

# CENTRALE LYON

UE PRO  
JEU DE GO

## PE50 : Rapport final

*Élèves :*

Charles BERGEAT  
Théo HUANGFU  
Rodrigue MILHOUD  
Timothée MIZON  
Lou THOMAS  
Sacha VANDERASPOILDEN

*Enseignants :*

Mohsen ARDABILIAN  
Philippe MICHEL  
Alexandre SAIDI  
Abdel-Malek ZINE

*Président du jury :*  
Thierry FARGÈRE

## Résumé

Ce travail vise à concevoir une intelligence artificielle jouant au jeu de Go, intégrée à une interface graphique intuitive destinée à en faciliter l'exploitation et l'analyse. Malgré des connaissances limitées au départ, nous avons progressivement exploré les aspects essentiels de ce jeu de stratégie réputé pour sa profondeur tactique.

Pour rendre notre IA compétente, nous nous sommes appuyés sur des méthodes modernes comme la méthode de Monte-Carlo Tree Search (MCTS), qui sélectionne les meilleurs coups en simulant des parties aléatoires. Cette approche, efficace face à la complexité du Go, est renforcée par l'utilisation de réseaux de neurones. Ces derniers permettent à l'IA d'anticiper les coups en s'appuyant sur des bases de données de parties historiques, améliorant ainsi sa prise de décision.

L'ensemble s'intègre dans une interface graphique pensée pour visualiser les parties, interagir avec l'IA et offrir une expérience claire et accessible, aussi bien pour le développement que pour l'utilisateur final.

## Abstract

This work aims to design an artificial intelligence capable of playing the game of Go, integrated into an intuitive graphical interface intended to facilitate its use and analysis. Despite our limited initial knowledge, we progressively explored the key aspects of this strategic game, renowned for its depth and complexity.

To build a competent AI, we relied on modern approaches such as Monte Carlo Tree Search (MCTS), which selects promising moves by simulating random games. This method, well-suited to the vast search space of Go, is enhanced by the use of neural networks. These allow the AI to anticipate moves by learning from large databases of past games, thus improving its decision-making.

All these elements are integrated into a graphical interface designed to visualize games, interact with the AI, and provide a clear and accessible experience-for both development and user interaction.

## Remerciements

Nous tenons à exprimer notre sincère gratitude à toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce projet.

Nous remercions tout d'abord nos tuteurs, Philippe MICHEL, Alexandre SAIDI et Abdel-Malek ZINE pour avoir conçu le sujet de ce projet et nous avoir offert l'opportunité de travailler sur ce sujet à la fois stimulant et formateur. Leur accompagnement et leurs retours ont été indispensables au bon déroulement du projet.

Nous adressons également nos remerciements à notre conseiller, Mohsen ARDABILLIAN, pour son aide précieuse apportée tout au long du projet, notamment pour la gestion des différentes étapes importantes, l'organisation de notre travail, ainsi que la rédaction des rapports. Son encadrement et ses conseils ont largement contribué à structurer notre démarche.

Enfin, nous remercions chaleureusement l'École Centrale de Lyon, pour les ressources mises à notre disposition et le cadre propice offert pour la réalisation de notre projet, ainsi que M. Emmanuel DELLANDREA pour nous avoir donné l'accès aux serveurs GPU de l'école, ce qui a été crucial pour le bon déroulement de notre travail.

# Table des matières

<b>Introduction</b>	<b>4</b>
<b>1 Contexte du projet</b>	<b>5</b>
<b>2 Problématique et objectifs</b>	<b>6</b>
<b>3 État des connaissances</b>	<b>7</b>
3.1 Contexte historique . . . . .	7
3.2 Etat de l'art . . . . .	7
<b>4 Présentation de la démarche effectuée et résultats</b>	<b>10</b>
4.1 Construction de l'interface . . . . .	10
4.2 Exploration par l'algorithme MCTS . . . . .	12
<b>5 Réseaux de neurones intégrés à MCTS</b>	<b>15</b>
5.1 Fonctionnement . . . . .	15
5.1.1 Typologie des réseaux . . . . .	15
5.1.2 Intégration des réseaux dans MCTS . . . . .	16
5.1.3 Données d'entraînement . . . . .	17
5.2 Policy Network . . . . .	18
5.2.1 Architecture détaillée . . . . .	18
5.2.2 Augmentation des données . . . . .	19
5.2.3 Optimisation et fonctions de perte . . . . .	20
5.2.4 Stabilisation à long terme . . . . .	22
5.3 Value Network . . . . .	24
5.3.1 Architecture détaillée . . . . .	24
5.3.2 Techniques partagées avec le réseau de politique . . . . .	24
5.3.3 Spécificités d'entraînement . . . . .	25
5.3.4 Métriques de performance et monitoring . . . . .	25
5.4 Renforcement par auto-jeu . . . . .	27
5.4.1 Concept et comparaison . . . . .	27
5.4.2 Cycle d'entraînement . . . . .	27
5.4.3 Mise à jour des réseaux . . . . .	27
5.4.4 Lancement de l'entraînement . . . . .	30
5.4.5 Évolution du taux de victoire . . . . .	31
5.5 Évaluation finale : MCTS Neuronal face à un MCTS « classique » . . . . .	32
5.6 Vers des évaluations plus fiables . . . . .	32
<b>6 Budget</b>	<b>34</b>
<b>Conclusion</b>	<b>36</b>
<b>Bibliographie</b>	<b>37</b>
<b>Annexes</b>	<b>38</b>

## Introduction

Avec l'accélération d'une part des capacités des intelligences artificielles et d'autre part des projets de recherche dans ce domaine, le développement de l'IA s'est imposé comme un sujet d'intérêt majeur dans les dernières années.

Dans ce contexte, le projet d'étude n°50 (PE 50) est proposé aux élèves de première année de l'École Centrale de Lyon. Ce projet d'étude a pour objectif de mettre au point un système d'intelligence artificielle capable de jouer au jeu de Go tout en étant le plus performant possible.

Le jeu de Go est un jeu de société originaire de Chine. Il oppose deux adversaires qui placent à tour de rôle des pierres, respectivement noires et blanches, sur les intersections d'un tablier quadrillé appelé goban en japonais. Le but est de contrôler le plan de jeu en y construisant des « territoires ». Les pierres encerclées deviennent des « prisonniers », le gagnant étant le joueur ayant totalisé le plus de territoires et de prisonniers. C'est un jeu connu pour sa complexité combinatoire, mais pour lequel l'intelligence artificielle a récemment fait une percée, le logiciel AlphaGo ayant battu les meilleurs joueurs du monde dans la dernière décennie.

Ce rapport vise à rendre compte du travail mené et des raisonnements conduits par l'équipe des élèves du PE 50. Dans un premier temps, le groupe a constitué un état de l'art relatif au sujet, sur lequel il s'est appuyé pour réaliser le projet. Ensuite, il a fallu mettre au point une interface graphique pour jouer des parties. Parallèlement, le modèle utilisé pour l'IA était en conception. Une fois choisi, une base de données a été acquise pour alimenter le modèle.

Premièrement, problématique du sujet et les objectifs (déterminés à partir de celle-ci) vont être rappelés, puis la démarche effectuée va être présentée ; ensuite l'utilisation des réseaux de neurones va être expliquée, et pour finir le budget va être exposé.

Les principales notions techniques et définitions sont regroupées en annexe pour faciliter la lecture du rapport. De plus, chaque terme technique apparaissant pour la première fois dans le texte est défini en note de bas de page.

## 1 Contexte du projet

En Mars 2016, la victoire du programme informatique AlphaGo, contre Lee Sedol, l'un des meilleurs joueurs de Go au monde, fait grand bruit. C'est la première fois de l'histoire qu'une intelligence artificielle parvient à gagner contre un joueur de ce niveau. Cette victoire marque un réel tournant dans l'histoire des intelligences artificielles.

En effet, le jeu de Go est considéré comme un des jeux les plus complexes (bien plus que le jeu d'échecs par exemple), en raison de l'immense nombre de configurations possibles ( $10^{10^{48}}$ ), mais aussi parce qu'il n'existe pas de méthode simple et fiable d'évaluation du plateau à un stade donné de la partie. L'intuition humaine, l'anticipation et la vision globale du jeu y jouent un grand rôle.

La victoire historique d'AlphaGo, bien que révolutionnaire, ne clôt pas la recherche sur les intelligences artificielles appliquées aux jeux de plateau. Bien au contraire, au Go, tout comme aux échecs, le domaine de recherche reste très actif, avec la volonté de créer des programmes toujours plus performants et innovants.

Ici, le but du projet n'est pas de développer une intelligence artificielle aussi performante qu'AlphaGo, les moyens mis à disposition étant bien inférieurs à ceux de l'entreprise DeepMind qui a développé celle-ci ; l'objectif est justement de développer un programme de bon niveau, à partir de peu de connaissances sur le sujet et d'une faible quantité de ressources disponibles.

## 2 Problématique et objectifs

La problématique du projet d'étude est la conception d'une intelligence artificielle jouant au jeu de Go, intégrée à une interface graphique intuitive.

Trois objectifs ont été tirés de cette problématique :

Objectif	Détail de l'objectif	Priorité	Mesure	Évaluation
Conception logicielle	Créer une interface graphique et coder un jeu de Go qui suit toutes les règles	1	Vérifier si toutes les règles sont respectées	100% si l'interface graphique est claire et toutes les règles sont respectées.
Intelligence artificielle	Coder une intelligence artificielle jouant au jeu de Go à un bon niveau amateur.	2	En fonction du niveau des joueurs battus par l'IA	100% si capable de battre un joueur de niveau 3 dan amateur.
Fonctionnalités en plus	Faire en sorte qu'il soit possible pour deux humains de jouer l'un contre l'autre sur deux ordinateurs différents, que notre interface contienne un tutoriel ou qu'il soit possible d'avoir accès à l'historique des coups.	3	En fonction de l'étendue des fonctionnalités présentées par notre programme.	100% si toutes les fonctionnalités indiquées sont présentes.

TABLE 1 – Objectifs du projet

Les objectifs ont été classés par ordre de priorité dans le tableau 1. Les objectifs globaux que sont le fait de coder un jeu de Go avec interface graphique et d'implémenter une intelligence artificielle sont plus importants que les fonctionnalités à ajouter.

En plus des objectifs directement liés à la réalisation technique du projet, nous poursuivons également des objectifs d'apprentissage, tant sur le plan des connaissances que des compétences transversales (soft skills). Sur le plan technique, il s'agit notamment de comprendre le fonctionnement de différents algorithmes, ainsi que les principes de conception d'une intelligence artificielle. Du côté des compétences humaines, ce projet est l'occasion de développer notre capacité à travailler en équipe, ainsi qu'à gérer un projet dans toutes ses dimensions : organisation, planification, communication et adaptation.

### 3 État des connaissances

Le jeu de Go est un jeu de plateau d'origine chinoise. Il oppose deux joueurs sur un plateau quadrillé de taille variable : 9x9, 13x13 ou 19x19. Les joueurs posent chacun leur tours des pierres sur les intersections du quadrillage, leur but étant de capturer le plus de territoire. Les règles sont expliquées en détail en annexe dans la figure 13. Même si les règles du jeu sont relativement simples, le jeu de Go est connu pour sa grande complexité.

#### 3.1 Contexte historique

Le PE réalisé n'est pas la continuation d'un précédent PE et il n'est pas accompagné d'un PAR. De nombreux sujets de thèse ont été réalisés autour de ce sujet. En effet, de nombreux chercheurs ont toujours été intéressés par la réalisation d'une intelligence artificielle capable de battre des vrais joueurs au jeu de Go [1].

Les premiers programmes capables de jouer au jeu de Go sortent au début des années 90, ils consistent en un parcours d'arbre qui est ensuite tronqué à l'aide d'une bonne fonction d'évaluation, mais comme celle-ci n'était pas assez bien définie, les premiers programmes n'étaient capables de jouer qu'à un niveau amateur. Les programmes suivants utilisent la méthode de Monte-Carlo, qui permet d'ajouter de l'aléatoire en simulant des parties, et en supprimant les branches les moins favorables. Cette méthode est utilisée dans des programmes tels que Crazy Stone qui a été capable de gagner pour la première fois contre des joueurs professionnels.

Et enfin en cumulant la méthode de Monte-Carlo et celle des réseaux de neurones, le programme AlphaGo créé par l'entreprise DeepMind a été capable de vaincre les meilleurs du monde. En 2016, elle bat l'un des meilleurs joueurs du monde Lee Sedol, combat qui sera retenu pour le 37ème coup joué par AlphaGo qui était complètement inattendu et très ingénieux. En 2017, AlphaGo gagne aisément contre le numéro 1 mondial, montrant sa dominance [2].

Mais AlphaGo n'est pas la seule intelligence artificielle avec une approche moderne ayant vu le jour récemment : Par exemple, AlphaGo Zero, logiciel aussi développé par Google DeepMind, consiste en un apprentissage autonome, contrairement à AlphaGo auquel on donnait des bases de données de parties jouées par des humains.

#### 3.2 Etat de l'art

La Programmation Orientée Objet (POO) est un paradigme qui organise le code en différentes classes d'objets. Tous les objets ont des propriétés qui leur sont propres, et des fonctions qui leur sont associées. La POO permet d'obtenir un code plus facile d'utilisation et mieux organisé. S'il y a plusieurs classes, celles-ci peuvent être mises en relation dans un diagramme UML qui explicite les classes et leurs relations.

La POO est très utilisée pour coder des jeux de plateaux, dans lesquels une classe est associée au plateau et d'autres à des éléments du plateau. Par exemple, au jeu de Go, on définit les classes Board, Piece, ScoreBoard, etc. Chacune de ces classes est représentée dans un diagramme UML.

Dans la théorie des jeux, on appelle état une configuration du jeu, état initial l'état

au début de la partie et état final un état qui met fin au jeu. Une fonction d'évaluation heuristique est une fonction qui attribue une valeur à un état du jeu, représentation de l'avantage qu'a un joueur dans la partie.

Pour les jeux à deux joueurs, un algorithme utilisé pour trouver le meilleur coup possible est l'algorithme Minimax : On représente chaque état possible dans un arbre, et pour la position dans laquelle on se trouve on regarde toutes les positions suivantes possibles dans l'arbre jusqu'à une certaine profondeur. Ceci fait, on évalue les états dans lesquels on se trouve avec une fonction heuristique et on choisit le coup qui permet d'arriver au noeud avec la plus grande valeur possible, c'est à dire qui est la plus favorable pour le joueur qui joue [3]. Cet algorithme est très efficace pour des jeux avec très peu d'états comme le tic tac toe, mais il l'est moins pour le go, car il peut y avoir au moins 101048 plateaux possibles, ce qui rend l'algorithme compliqué à mettre en place en termes de complexité. [4].

Une alternative plus rapide de l'algorithme Minimax est l'algorithme Alpha Beta. Il élague l'arbre en supprimant les branches de l'arbre que l'algorithme trouve inutile. Il y a deux types d'élagage, l'élagage alpha : si la valeur du noeud est inférieure à alpha alors on supprime le noeud ; et l'élagage beta : si la valeur est supérieure à beta on supprime le noeud aussi. Cette méthode est assez efficace pour des jeux plus complexes tels que les échecs, mais reste quand même inefficace pour le jeu de Go. Il y a deux problèmes principaux : encore une fois le grand nombre de positions possibles au go, mais il est aussi compliqué d'établir une fonction d'évaluation heuristique fiable. En effet le go est aussi très complexe dans l'évaluation d'un plateau de jeu, car les interactions entre les différents groupes de pierres sont difficiles à analyser. De plus, il y a une grande liberté de mouvement, (presque) toutes les intersections libres du plateau étant jouables.

Une autre méthode utilisée en théorie des jeux est la méthode de Monte-Carlo, aussi appelée Monte-Carlo Tree Search (MCTS) [5]. La méthode de Monte-Carlo permet elle aussi d'élaguer un arbre. Mais elle le fait en choisissant des échantillons de manières aléatoires. Elle est utile lorsqu'il y a beaucoup trop de cas à traiter, en élaguant l'arbre beaucoup plus qu'AlphaBeta. Pour choisir les échantillons, la méthode garde à la fois les échantillons qui ont la probabilité de mener à une victoire (principe d'exploitation) et les échantillons qui sont les moins joués et qui peuvent surprendre (principe d'exploration) [5].

La méthode Monte Carlo Tree Search est très utile au jeu de Go, et il a été choisi que l'intelligence artificielle créée dans ce projet utilise en premier lieu celle-ci. La façon dont elle a été implémentée sera explicitée plus en détail dans la partie 4.2 du rapport.

Les réseaux de neurones sont une structure inspirée par le cerveau humain. Ils servent à résoudre des problèmes complexes en apprenant à partir de grandes bases de données. Cette méthode est très utilisée dans des domaines comme la reconnaissance d'image, la traduction automatique ou encore dans les jeux de stratégie.

Un réseau de neurones est constitué de plusieurs couches de neurones, les neurones sont reliés entre eux et permettent de prendre des décisions à l'aide de fonctions mathématiques appelées fonctions d'activation. Chaque connexion entre les neurones est associée à un poids qui représente l'importance de la liaison et un biais qui permet d'ajuster la sortie donnée par le neurone en fonction de l'entrée. Pour optimiser un tel réseau, il faut donner

beaucoup d'information à l'intelligence artificielle, pour que celle-ci puisse ajuster les poids et les biais en fonction des situations. Par exemple, au jeu de Go, il faut entraîner le réseau de neurones sur un grand nombre de parties qui ont déjà été jouées.

## 4 Présentation de la démarche effectuée et résultats

### 4.1 Construction de l'interface

Avant de pouvoir nous concentrer sur l'implémentation d'une intelligence artificielle pour le jeu de Go, il nous a semblé indispensable de développer une interface graphique simple d'utilisation. Cette interface devait permettre non seulement de représenter fidèlement un plateau de jeu, en respectant les règles fondamentales du Go (placement des pierres, alternance des tours, gestion des captures, etc.), mais aussi d'offrir une expérience utilisateur claire et intuitive.

Parmi les fonctionnalités essentielles, il y avait l'intégration d'un menu d'accueil permettant à l'utilisateur de sélectionner le mode de jeu souhaité : un affrontement contre une intelligence artificielle, une partie en mode joueur contre joueur (1 vs 1) ou encore un mode bac à sable (sandbox) facilitant les tests. Il était également important pour nous d'offrir le choix entre plusieurs tailles de plateau, allant du format compact au standard :  $5 \times 5$ ,  $9 \times 9$ ,  $13 \times 13$  et  $19 \times 19$ . Le menu que nous avons réalisé est illustré dans la figure 1 ci-dessous, et permet à l'utilisateur de choisir le mode qu'il veut avant de commencer la partie.



FIGURE 1 – Menu

En nous appuyant sur ce cahier des charges et sur notre état de l'art, nous avons déterminé que la bibliothèque Tkinter, intégrée à Python, constituait la solution la plus adaptée à notre projet.

L'interface graphique que nous avons conçue est illustrée dans la figure 2 ci-dessous. Cette base visuelle robuste a ainsi constitué un socle essentiel pour la suite de nos travaux, notamment pour l'intégration progressive des algorithmes d'intelligence artificielle.

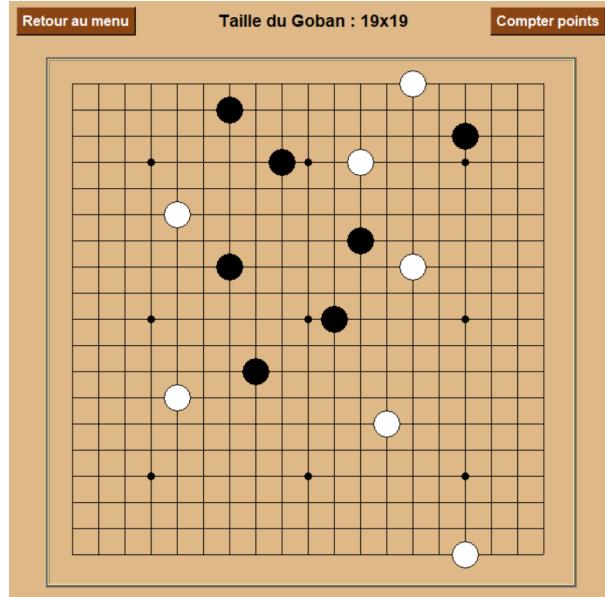


FIGURE 2 – Plateau  $19 \times 19$

Pour l’architecture globale de notre application nous avons respecté les principes de la POO qui sont de découper notre application en différents objets ayant leurs attributs et leurs méthodes propres. Le diagramme UML simplifié dans la figure 3 ci-dessous présente les relations entre les différents objets créés pour notre application. La classe Goban regroupe les attributs et méthodes relatives à l’affichage de la partie sur le goban et contient le moteur de jeu, la classe Menu regroupe les éléments nécessaires au menu et contient la classe Goban : c’est en créant une instance de cette classe que nous lançons notre application. La classe MoteurDeJeu permet de gérer la logique du jeu et le déroulement de la partie tout en intégrant la fonctionnalité de jeu contre notre intelligence artificielle. La classe MCTS est notre intelligence artificielle et la classe Plateau représente informatiquement notre goban à l’aide d’une matrice Numpy et intègre les fonctions relatives au plateau(territoires, libertés des pierres).

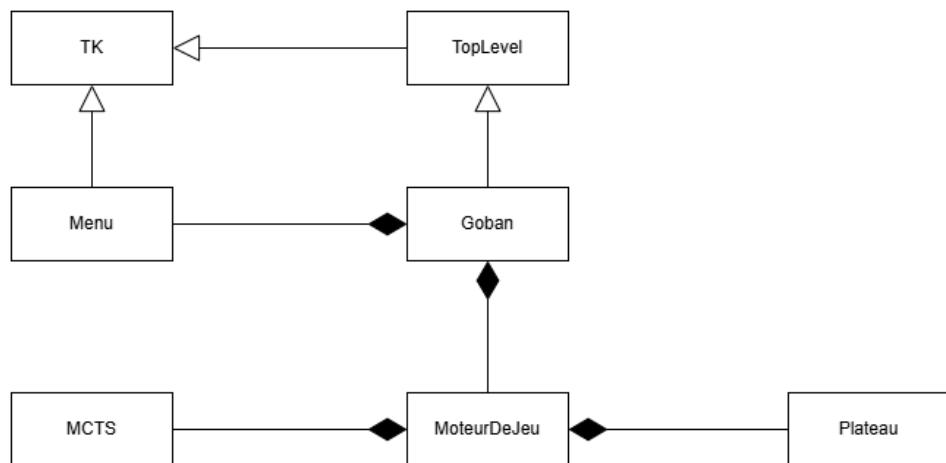


FIGURE 3 – Diagramme UML simplifié de l’application

## 4.2 Exploration par l'algorithme MCTS

En parallèle du développement de notre interface graphique, nous avons également entamé la conception de notre intelligence artificielle. L'analyse menée dans notre état de l'art nous a permis d'identifier la méthode principale que nous allions implémenter pour doter notre IA de capacités adaptées au jeu de Go : l'algorithme MCTS (Monte Carlo Tree Search).

Cet algorithme permet d'explorer l'arbre des coups possibles du jeu de manière asymétrique, tout en limitant l'exploration à un nombre raisonnable de noeuds. Il repose sur une construction progressive de l'arbre de jeu au fil de l'exploration. L'algorithme MCTS s'appuie sur un cycle de quatre phases, répétées à chaque itération : la sélection, l'expansion, la simulation et la rétropropagation (backpropagation) [5].

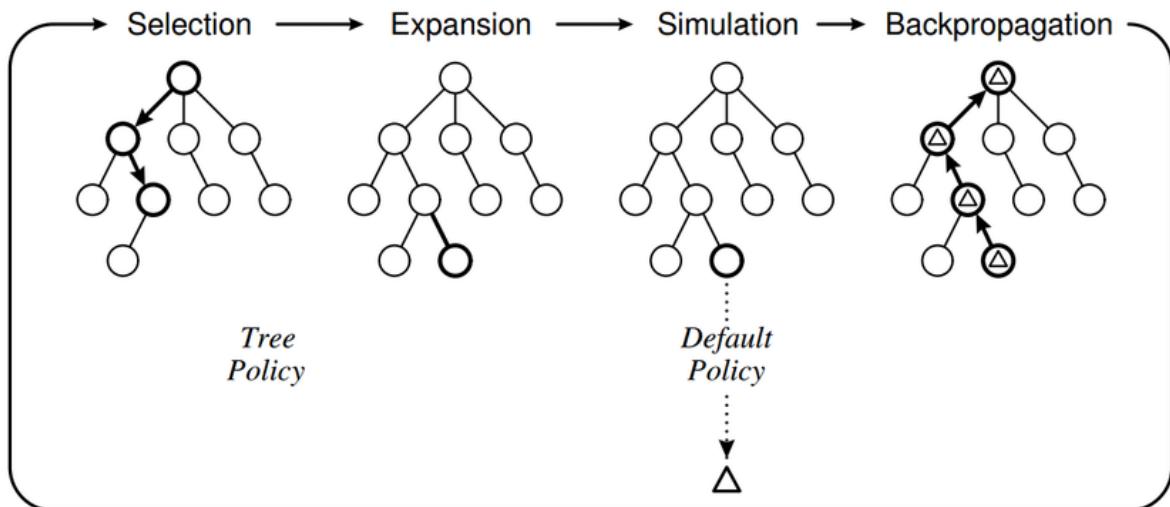


FIGURE 4 – Une itération de l'algorithme MCTS [5]

La phase de sélection consiste à choisir le noeud à partir duquel nous allons continuer de construire l'arbre. Cette sélection se fait en suivant une politique de décision. La phase d'expansion consiste à rajouter à notre arbre les noeuds correspondant aux états suivant possibles à notre arbre. La particularité de l'algorithme MCTS est la phase de simulation où on joue des coups au hasard jusqu'à la fin de la partie afin de pouvoir donner une valeur au coup choisi que l'on remonte sur tous les noeuds parents lors de la phase de backpropagation. Les différentes étapes sont illustrées dans la figure 4.

Cette méthode est donc très utile dans le cadre du jeu de Go car ce jeu a un arbre d'état très large et profond ce qui rend son exploration très longue et on ne peut pas évaluer une position du plateau au cours de la partie ce qui rend la phase de simulation très utile car elle permet de simuler la partie jusqu'à sa fin où l'on peut alors évaluer le plateau. Pour mettre en place cet algorithme nous avons tout d'abord créé deux classes nécessaires à la construction de l'arbre de jeu : une classe noeud pour stocker un état du jeu et les paramètres nécessaires à l'algorithme MCTS et une classe arbre regroupant différents noeuds pour pouvoir construire l'arbre avec MCTS. Le diagramme UML de notre arbre est représenté dans la figure 5 ci-dessous.

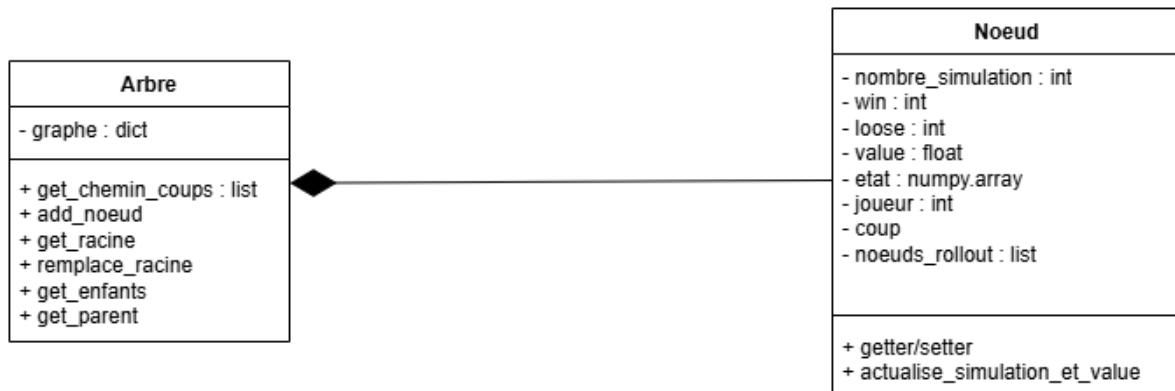


FIGURE 5 – Diagramme UML de notre arbre de jeu

Comme politique de décision nous avons décidé d'utiliser la méthode la plus utilisée, la formule UCT (Upper confident bound for Tree) [5] :

$$\frac{w}{n} + c\sqrt{\frac{N}{n}}$$

Cette formule peut se voir comme la somme de deux termes : le terme d'exploitation et le terme d'exploration. Le facteur  $c$  correspond à une constante permettant de régler le compromis entre exploration et exploitation. Nous avons fixé cette valeur à  $c = \sqrt{2}$  qui est la valeur optimale théorique.

Avec cette implémentation basique de l'algorithme nous avions une première IA fonctionnelle ; cependant, elle était assez lente et de moins bon niveau qu'un joueur débutant. Nous avons donc réfléchi à comment améliorer les coups joués par l'algorithme. Pour cela nous avons décidé de simuler plusieurs parties lors de la phase de simulation grâce à du multithreading avec la bibliothèque 'multithreading' de python afin de moyenner les résultats obtenus à la fin de chaque simulation pour avoir un score plus fiable à remonter lors de la backpropagation.

Cependant cela a encore augmenté le temps de calcul, donc nous avons ensuite réfléchi à comment réduire le temps de calcul. Nous avons donc défini un nombre maximal de coups à jouer lors de la simulation et nous avons mis en place une méthode d'évaluation du plateau une fois ce nombre de coups maximum atteint. Notre méthode d'évaluation repose sur le principe d'influence des pierres sur les cases voisines et sur la notion d'ensemble flou [6].

À partir d'un état donné du plateau (figure 6a) nous affectons une valeur de +128 aux pierres noires et de -128 aux pierres blanches puis nous calculons l'influence de ces cases sur les autres cases du plateau en répétant plusieurs phases de dilatation. Enfin plusieurs phases d'érosion sont aussi réalisées pour limiter l'influence des pierres afin que ces zones d'influence correspondent à celles visualisées par un joueur expérimenté. Une fois ces différentes étapes réalisées un score compris entre -1 et 1 est calculé pour chaque case, ce qui permet de quantifier l'appartenance des cases au territoire noir ou blanc (figure 6b).

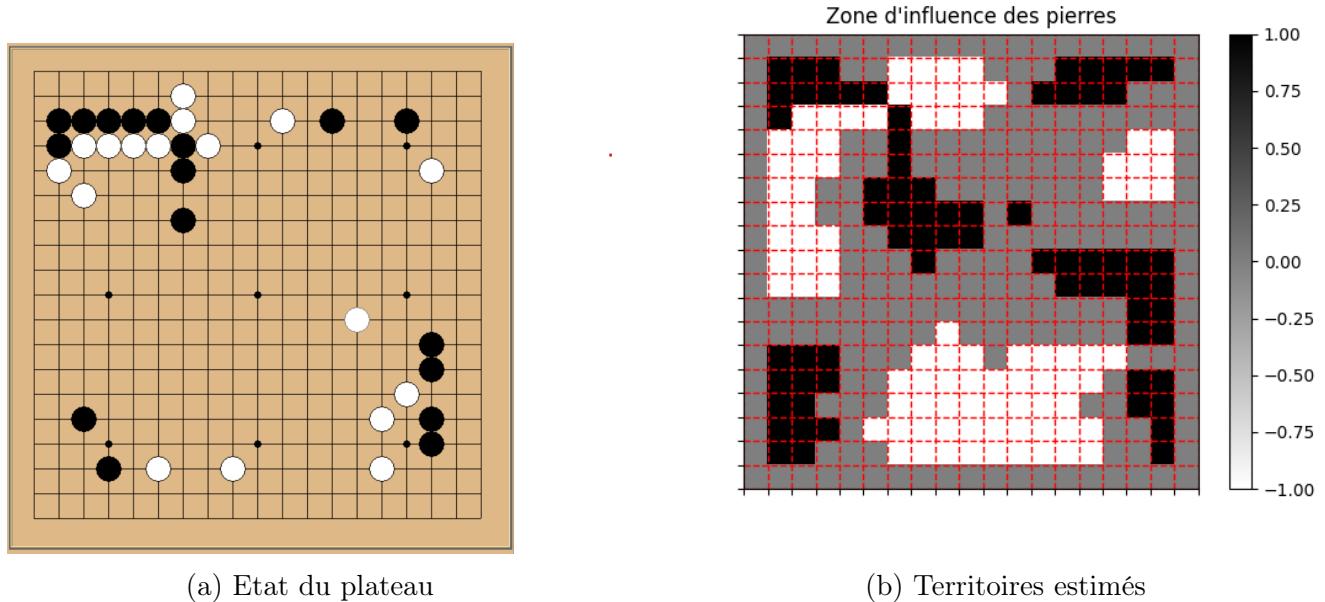


FIGURE 6 – Exemple de territoires estimés à partir d'un état du plateau

Après toutes ces améliorations le niveau obtenu avec notre intelligence artificielle n'était toujours pas suffisant pour nos objectifs. Nous avons donc commencé à envisager la mise en place d'un réseau de neurones en parallèle de l'algorithme MCTS afin d'améliorer son niveau de jeu.

## 5 Réseaux de neurones intégrés à MCTS

### 5.1 Fonctionnement

Nous avons vu jusque-là l'implémentation d'un algorithme MCTS qui utilisait un protocole de sélection pour élaguer notre arbre. Cet élagage fait gagner un temps de calcul considérable à côté d'autres algorithmes comme min-max, ou alpha-bêta. Toutefois, il reste toujours conséquent. Ce temps de calcul nous constraint notamment à réduire le nombre d'itérations pour calculer le prochain coup avec MCTS. Ainsi, nous ne pouvions jusque-là pas allier rapidité d'exécution et performances de l'algorithme.

Pour pallier ce problème, la littérature nous suggère de s'intéresser à l'intégration de réseaux neuronaux dans MCTS, ce qui démontrerait un gain de performances significatif.

À l'instar d'AlphaGo, nous avons essayé d'implémenter un réseau de politique (Policy Network) pour prédire une distribution des meilleurs coups possibles sur le plateau, ainsi qu'un réseau de valeurs (Value Network) pour prédire l'avantage d'un joueur à un instant de la partie et en les intégrant dans MCTS.

#### 5.1.1 Typologie des réseaux

**Réseau convolutionnel** L'architecture des réseaux de neurones convolutionnels (CNN) est particulièrement adaptée pour les jeux de plateau, notamment le jeu de Go. En effet, ce type de réseau est conçu pour l'analyse d'images et de structures spatiales. Les CNN utilisent des filtres glissants (appelés *convolutions*) qui permettent d'extraire des motifs locaux. Ces caractéristiques sont idéales pour représenter les configurations d'un jeu de plateau, où la relation spatiale entre les éléments est essentielle pour déterminer une stratégie optimale.

Un plateau de jeu peut être comparé à une image, dans la mesure où une image est généralement représentée par plusieurs matrices, chaque matrice correspondant à un *canal* (par exemple, les canaux *R*, *G* et *B* pour les couleurs dans une image). De manière similaire, un plateau de Go peut être représenté par deux matrices, ou *canaux* :

- Le premier canal indique les positions des pierres du joueur courant (de taille  $1 \times 1$ ).
- Le deuxième canal représente les pierres de l'adversaire.

Ainsi, les réseaux convolutionnels peuvent traiter ces deux canaux de manière simultanée pour capturer les informations cruciales du jeu et approximativement déterminer la meilleure stratégie à adopter à partir de l'état actuel du plateau.

**Réseau de politique** Le **réseau de politique** a pour objectif de prédire le prochain coup à jouer à partir d'un état donné du jeu. Il prend en entrée une représentation simplifiée du plateau, composée de deux canaux binaires de même taille que le plateau : un tableau de 0/1 indiquant la présence (1) ou l'absence (0) des pierres du joueur courant, et un second tableau 0/1 pour celles de l'adversaire.

Dans AlphaGo, d'autres canaux étaient utilisés pour représenter des informations complexes (par exemple, les pierres en atari, les états précédents du plateau, les coups possibles, etc.), avec un total de 19 canaux. Cependant, nous avons opté pour une version simplifiée en ne conservant que deux canaux (les deux principaux), permettant de capturer les informations essentielles pour le raisonnement tactique élémentaire.

Séparer le plateau en deux canaux distincts (1 : pierres alliées, 2 : pierres adverses) force le réseau à isoler visuellement les chaînes alliées et adverses. Il repère ainsi plus clairement les groupes de pierres, et se rapproche de l'intuition humaine qui distingue naturellement les blocs blancs ou noirs.

À partir de ces deux canaux, le réseau retourne une distribution de probabilités sur l'ensemble des coups possibles (un vecteur de taille plateau + 1 pour l'action « passer »), appelée **politique**, reflétant la probabilité estimée que chaque coup soit joué par un joueur fort.

**Réseau de valeur** Le **Value Network** a pour rôle d'évaluer la qualité d'une position donnée. À partir des mêmes entrées que le réseau de politique, il prédit un score scalaire dans l'intervalle [0, 1], obtenu à l'aide d'une fonction **sigmoïde** en sortie. Ce score représente l'estimation de la probabilité de victoire du joueur courant. Ce choix de sortie est motivé par la compatibilité avec notre implémentation de l'algorithme MCTS, qui utilise déjà des valeurs comprises entre 0 et 1.

Le réseau est entraîné à minimiser l'erreur quadratique moyenne (MSE) entre cette prédiction et l'issue réelle de la partie (victoire = 1, défaite = 0), en utilisant les résultats des parties générées lors de l'entraînement.

**Réseau Policy-Value (non retenu)** AlphaGo Zero utilise un réseau **Policy-Value** à double tête qui partage les premières couches convolutionnelles pour produire à la fois une politique et une estimation de valeur. Ce choix architectural permet de mutualiser les représentations internes et d'entraîner un seul modèle plus efficacement.

Dans notre implémentation, nous avons préféré séparer les réseaux de politique et de valeur afin de simplifier le processus d'entraînement, de faciliter le débogage, et d'avoir plus de flexibilité dans les expérimentations. Cette séparation rend également possible un entraînement supervisé indépendant pour chaque composant avant de les combiner dans la boucle de renforcement. Grâce au serveur GPU de Centrale Lyon, nous pouvons lancer plusieurs entraînements en simultané sans perte significative de performance.

### 5.1.2 Intégration des réseaux dans MCTS

L'algorithme MCTS utilisé ici suit les étapes classiques : **sélection**, **expansion**, **roll-out**, et **rétropropagation**. Les réseaux de neurones interviennent à deux niveaux précis pour remplacer des heuristiques classiques :

- Le **Policy Network** guide l'expansion de l'arbre ainsi que la sélection d'un noeud via les priors<sup>1</sup>.
- Le **Value Network** remplace les simulations aléatoires du roll-out par une évaluation directe de l'avantage d'un joueur.

**Sélection et calcul de PUCT.** Lors de la phase de sélection, chaque coup  $a$  depuis l'état  $s$  se voit attribuer un score PUCT qui combine l'estimation de sa valeur et la

1. Les **priors** (ou probabilités a priori) représentent les préférences initiales du MCTS pour chaque coup, fournies ici par le réseau de politique. Elles orientent l'exploration de l'arbre en attribuant plus de simulations aux actions jugées prometteuses dès le départ.

confiance que lui accorde le réseau de politique [7] :

$$\text{PUCT}(s, a) = Q(s, a) + c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}.$$

Ici,  $Q(s, a)$  est la moyenne des valeurs retournées par les simulations passées pour le coup  $a$ ,  $N(s, a)$  le nombre de fois où  $a$  a été exploré, et  $P(s, a)$  la probabilité prioritaire fournie par le *Policy Network*. La constante  $c_{\text{puct}}$  ajuste l'équilibre entre :

- **Exploitation** via  $Q(s, a)$ , qui favorise les coups déjà jugés forts,
- **Exploration** via le second terme, qui alloue plus de simulations aux coups préconisés par le réseau, surtout en début de partie.

Cette formule garantit que les coups les plus prometteurs sont rapidement explorés, tout en conservant une part d'exploration contrôlée par le réseau.

**Expansion.** Lorsqu'un noeud encore non développé est atteint, on effectue les étapes suivantes :

1. Passage de l'état  $s$  dans le *Policy Network* pour obtenir un vecteur de logits  $\ell(\cdot)$  couvrant tous les coups possibles.
2. Ces scores sont ensuite convertis en probabilités par la normalisation *softmax*<sup>2</sup> de sorte que chaque coup  $a$  se voit attribuer une probabilité :

$$P(s, a) = \text{softmax}(\ell)_a.$$

3. Pour limiter la taille de l'arbre, on ne conserve que les  $k$  coups les plus probables (paramètre `top_k`) pour générer les enfants ; les autres positions ne sont pas développées.

Cette stratégie garantit que le réseau concentre l'expansion sur les coups les plus prometteurs tout en maîtrisant la croissance de l'arbre de recherche.

**Rollout (sans simulations aléatoires).** Plutôt que d'effectuer des simulations aléatoires coûteuses, l'état  $s$  nouvellement exploré est directement passé au *Value Network*, qui estime la probabilité de victoire du joueur courant par une prédiction  $v(s) \in [0, 1]$ . Cette approche réduit significativement le coût de calcul en supprimant les playouts aléatoires, tout en améliorant la stabilité de l'apprentissage grâce à des évaluations cohérentes et répétables.

### 5.1.3 Données d'entraînement

**Essai d'un entraînement supervisé avec FoxDataset (19x19)** Dans un premier temps, nous avons entraîné un réseau de politique (type ResNet) en apprentissage supervisé sur le **FoxDataset**, composé de parties de joueurs professionnels (1p - 9p). L'entrée est constituée de deux canaux (pierres du joueur courant et de l'adversaire), et la sortie est une distribution de probabilité sur les  $19 \times 19 + 1$  coups possibles.

---

2. La fonction *softmax* transforme un vecteur de scores arbitraires en distribution de probabilité :  $\text{softmax}(\ell)_i = \exp(\ell_i) / \sum_j \exp(\ell_j)$ .

Ce modèle, bien qu'efficace pour des positions courantes, montre ses limites hors distribution (ex. : lors d'ouvertures inhabituelles pour un joueur professionnel, le modèle passe son tour), et reste trop coûteux pour être utilisé dans un MCTS sur plateau  $19 \times 19$ . Nous transposons le même problème au réseau de valeurs.

**Nous nous concentrerons donc, pour des raisons de simplicité d'entraînement, de robustesse et de performances computationnelles, sur des grilles  $9 \times 9$  dans la suite du projet.**

**Génération  $9 \times 9$  avec KataGo** Le Go en  $9 \times 9$  souffre d'un manque de données annotées, contrairement au format standard  $19 \times 19$ . Pour y remédier, nous avons utilisé **KataGo**, un moteur open source performant combinant MCTS et réseaux de neurones, capable d'analyser chaque position et de fournir directement une politique cible (distribution de probabilité sur les coups) ainsi qu'une valeur cible (probabilité de victoire du joueur actuel).

Nous avons téléchargé le modèle *kata1-b28c512nbt-s8925522176-d4787295149* (ELO :  $14017.1 \pm 17.4$ ) depuis [katagotraining.org](http://katagotraining.org), afin de générer automatiquement un corpus de 30 000 parties sur plateau  $9 \times 9$ .

KataGo propose de nombreux canaux d'entrée (historiques, libertés, patterns tactiques) et de sortie (score, territoire, probabilités). Pour simplifier l'apprentissage tout en conservant l'essentiel des dynamiques du jeu, nous avons restreint la configuration à :

- **Entrée** : 2 canaux binaires (vecteurs de taille 81) représentant les positions des pierres du joueur courant et les positions des pierres adverses.
- **Politique cible** : vecteur de taille 82 (81 cases + passe), normalisé en distribution de probabilité (entre 0 et 1).
- **Valeur cible** : Probabilité de victoire du joueur actuel, obtenue via la somme et la normalisation du canal de propriété finale.

Ce choix minimaliste permet un apprentissage rapide et efficace, parfaitement adapté à l'échelle du plateau  $9 \times 9$ . Une description complète des canaux d'entrée et de sortie proposés par KataGo est fournie en annexe.

Fort de cette configuration d'entrées et de cibles allégée mais représentative, nous pouvons maintenant détailler l'architecture du *Policy Network*.

## 5.2 Policy Network

### 5.2.1 Architecture détaillée

Le réseau de politique a pour objectif de prédire, pour une position donnée sur un plateau  $9 \times 9$ , une distribution de probabilité sur les coups jouables, reflétant la stratégie d'un joueur fort. L'architecture implémentée repose sur plusieurs choix structurants permettant à la fois une bonne capacité de généralisation et une stabilité à l'entraînement :

- **Stem** : le réseau commence par une convolution  $3 \times 3$  avec 128 filtres, suivie d'une normalisation par lots<sup>3</sup> (*Batch Normalization*) et d'une activation *ReLU*<sup>4</sup>. Cette

3. La *normalisation par lots* (*BatchNorm*) recentre et renormalise les activations d'une couche sur chaque mini-lot, ce qui accélère et stabilise l'apprentissage.

4. *ReLU* (*Rectified Linear Unit*) applique la fonction  $f(x) = \max(0, x)$ , introduisant la non-linéarité tout en évitant la saturation des gradients pour les valeurs positives.

couche d'entrée transforme les deux canaux binaires d'entrée (pièces du joueur courant et de l'adversaire) en une représentation initiale dense, apte à capturer les premiers motifs locaux sur le plateau.

- **Blocs résiduels ( $\times 8$ )** : on empile ensuite 8 blocs de type ResNet<sup>5</sup>, chacun contenant deux convolutions  $3 \times 3$ , précédées d'une normalisation par lot et d'une activation ReLU. Une connexion résiduelle (skip connection) permet de sommer directement l'entrée et la sortie du bloc. Cela facilite la propagation du gradient pendant l'apprentissage et atténue le risque de dégradation des performances dans les réseaux profonds.
- **Attention globale** : un module *Squeeze-and-Excitation* [8] clôture la pile convolutive : chaque carte de caractéristiques est d'abord comprimée par une moyenne globale (*squeeze*), puis un petit MLP<sup>6</sup> à deux couches apprend des coefficients multiplicatifs (*excitation*) appliqués canal par canal, recalibrant ainsi dynamiquement l'importance des filtres selon le contexte de la partie.
- **Tête de politique<sup>7</sup>** : la sortie du dernier bloc passe par une convolution  $1 \times 1$  afin de réduire la profondeur. Elle est ensuite aplatie (flatten) puis passée dans une couche linéaire vers un vecteur de 82 dimensions, correspondant aux 81 positions du plateau auxquelles ont ajouté l'action **pass**. Une fonction **softmax** est utilisée en sortie pour obtenir une distribution de probabilité.

**Comparaison** : contrairement à une simple CNN qui accumulerait des convolutions sans mécanisme de réutilisation du signal, notre réseau tire parti de la profondeur des blocs résiduels pour capturer des motifs de haut niveau, tout en restant entraînable efficacement. Le module d'attention complète cette capacité en introduisant une forme d'adaptation globale, très utile dans des configurations complexes.

### 5.2.2 Augmentation des données

Nous avons généré un corpus initial de 30 000 parties annotées par KataGo. Pour tirer parti de l'invariance du goban sous les rotations et réflexions du groupe  $D_4$ , nous appliquons, *à la volée* pendant l'entraînement, l'une des huit symétries possibles (rotations de  $0^\circ, 90^\circ, 180^\circ, 270^\circ$  et leurs reflets) à chaque position extraite. Autrement dit, chaque échantillon puise dans les 30 000 parties d'origine, puis l'une des huit transformations est choisie aléatoirement, sans aucun stockage préalable des variantes. Cette stratégie multiplie virtuellement la taille du dataset par huit : soit près de 240 000 parties équivalentes (environ 17 millions de positions) tout en conservant un coût de stockage et de prétraitement minimal.

- **Transformations géométriques** : on considère les 8 isométries du carré, issues de  $D_4$ , le groupe des symétries discrètes : rotations de  $0^\circ, 90^\circ, 180^\circ, 270^\circ$  combinées avec ou sans une réflexion horizontale.
- 5. **ResNet** insère une *skip connection* qui ajoute l'entrée d'un bloc à sa sortie : le bloc n'apprend alors qu'un « résidu ». Ce détour trivial garde les gradients stables et permet d'empiler un grand nombre de couches sans perte de performance.
- 6. **MLP** (Multilayer Perceptron) : réseau de neurones entièrement connecté, où chaque couche dense applique une transformation linéaire suivie d'une non-linéarité ; sans structure spatiale, il opère sur des vecteurs plutôt que sur des cartes de caractéristiques.
- 7. La tête de politique compresse les features, sort 82 scores (81 cases + **pass**) et les convertit en probabilités par softmax.

- **Application synchrone** : chaque transformation est appliquée simultanément à l'entrée (les deux canaux représentant les pierres) et à la cible (distribution de probabilité sur les coups). Cela garantit la cohérence entre l'état du plateau et la politique apprise.
- **Bénéfices** : l'augmentation multiplie par huit l'effectif d'exemples vus durant l'apprentissage, réduit le surapprentissage à des motifs orientés spécifiques et, surtout, contraint explicitement le réseau à respecter les symétries réelles du goban.

### 5.2.3 Optimisation et fonctions de perte

Plusieurs techniques d'optimisation ont été mises en oeuvre afin de stabiliser et d'accélérer l'apprentissage :

- **Perte de Kullback-Leibler avec label smoothing** : pour entraîner le réseau de politique, nous utilisons une divergence de Kullback-Leibler entre la distribution prédite  $p_\theta$  et la distribution cible  $p_{\text{target}}$  issue des données fournies par KataGO. Afin de rendre l'apprentissage plus stable et d'éviter que le réseau ne devienne trop sûr de lui en attribuant une probabilité écrasante à un seul coup (phénomène connu sous le nom de *surconfiance*<sup>8</sup>), nous appliquons une technique appelée *label smoothing* [9].

Cette technique consiste à lisser légèrement la distribution cible  $p_{\text{target}}$  en y injectant une petite composante uniforme. Formellement, on définit une distribution modifiée :

$$\tilde{p} = (1 - \epsilon)p_{\text{target}} + \epsilon \cdot U,$$

où  $\epsilon = 0.1$  est le coefficient de lissage, et  $U$  la distribution uniforme sur les 82 actions légales (81 positions + passe).

La perte finale est simplement la divergence de Kullback–Leibler entre cette cible lissée  $\tilde{p}$  et la distribution prédite  $p_\theta$  ; elle pousse ainsi le réseau à rapprocher ses probabilités de celles fournies par KataGo tout en restant régularisé.

- **Optimiseur**<sup>9</sup> : nous avons opté pour **RAdam** (*Rectified Adam*), un optimiseur adaptatif plus stable qu'Adam<sup>10 11</sup> parce qu'il introduit un terme de *rectification* qui retarde l'échelle adaptative tant que le second moment  $v_t$  n'est pas encore fiable. Cela évite les pas surdimensionnés au démarrage et assure une convergence plus

8. La *surconfiance* (overconfidence) désigne le biais de calibration où le modèle attribue des probabilités excessivement élevées à ses prédictions, même lorsqu'elles sont erronées ; elle se distingue de l'*overfitting*, qui résulte d'une sur-adaptation du modèle aux données d'entraînement et d'une perte de capacité à généraliser sur de nouvelles données.

9. Un *optimiseur* est l'algorithme numérique qui met à jour les poids d'un réseau de neurones à partir des gradients afin de minimiser la fonction de perte [10].

10. Adam calcule deux moyennes mobiles :  $m_t$  (gradient) et  $v_t$  (carré du gradient). Le pas est alors  $\eta m_t / (\sqrt{v_t} + \epsilon)$ . Durant les premières itérations,  $v_t$  sous-estime la véritable variance du gradient, d'où un pas initial trop grand et souvent oscillant.

11. Comme Adam, **RAdam** maintient deux moyennes exponentielles :  $\beta_1 = 0,9$  lisse le *gradient brut* (90% d'historique + 10% du nouveau gradient) : c'est le **momentum**, qui stabilise la direction.  $\beta_2 = 0,999$  lisse le *carré du gradient* (99,9 % d'historique) : il mesure la **variance** pour ajuster la taille du pas. Formules :  $m_t = \beta_1 m_{t-1} + (1 - \beta_1)g_t$ ,  $v_t = \beta_2 v_{t-1} + (1 - \beta_2)g_t^2$ , puis  $\theta_{t+1} = \theta_t - \eta m_t / (\sqrt{v_t} + \epsilon)$ . En clair :  $\beta_1$  commande la *mémoire directionnelle*,  $\beta_2$  la *mémoire d'amplitude*. RAdam n'active le facteur adaptatif  $1/\sqrt{v_t}$  qu'après quelques itérations, le temps que  $v_t$  devienne fiable, évitant ainsi les pas trop grands au démarrage.

régulière. Nous avons également fixé le paramètre `weight decay`<sup>12</sup> à  $2 \times 10^{-4}$  afin de contrôler la norme des poids tout au long de l'apprentissage.

- **Scheduler**<sup>13</sup> : `ReduceLROnPlateau` réduit dynamiquement le taux d'apprentissage d'un facteur 0.5 lorsqu'aucune amélioration significative de la perte de validation n'est observée. (documentation PyTorch)
- **Refresh \_ model** : tous les 50 cycles d'entraînement, l'optimiseur est réinitialisé (sans altérer les poids du réseau). Cette opération vise à débloquer des plateaux de convergence où les gradients deviennent peu informatifs malgré une performance apparente stable. Nous avons observé empiriquement qu'une telle réinitialisation entraîne systématiquement une chute significative de la `train_loss` et de la `val_loss`, ce qui suggère une relance efficace de la dynamique d'apprentissage (voir figure 7). Nous justifierons l'efficacité de cette méthode dans le paragraphe **impact du refresh\_model**.

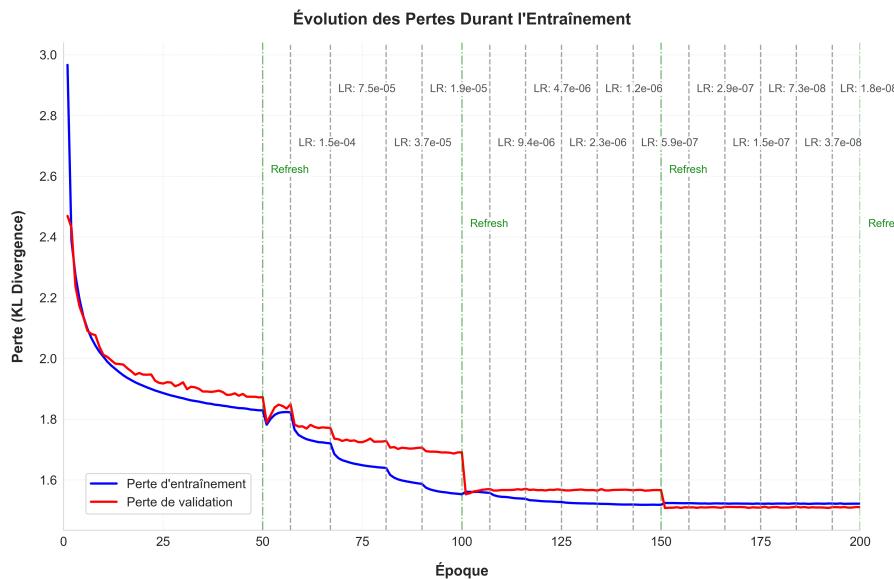


FIGURE 7 – Évolution de la divergence KL du réseau de politique au cours de l'entraînement, avec indication des phases de réduction du taux d'apprentissage (`ReduceLROnPlateau`) et des relances de l'optimiseur.

La figure 7 illustre l'évolution de la **KL divergence** durant l'entraînement du réseau de politique, mesurée à la fois sur les données d'entraînement (courbe bleue) et de validation (courbe rouge). On y observe plusieurs points clés confirmant l'efficacité des stratégies adoptées :

- **Diminution progressive et régulière des pertes** : les premières phases montrent une décroissance fluide de la perte, traduisant la stabilité offerte par l'optimiseur `RAdam`, qui amortit les fortes variations initiales sans compromettre la dynamique de convergence.

12. Le `weight decay` soustrait à chaque mise à jour une petite fraction de la valeur des poids, limitant ainsi leur amplitude et réduisant le sur-apprentissage sans surcoût de calcul.

13. Un `scheduler` d'apprentissage ajuste automatiquement le learning rate au cours de l'entraînement ; `ReduceLROnPlateau` le divise (ici par 2) quand la perte de validation ne progresse plus, permettant des pas plus fins sans intervention manuelle.

— **Effets du scheduler ReduceLROnPlateau** : les lignes grises verticales indiquent les instants où le scheduler détecte une stagnation de la perte de validation et réduit automatiquement le learning rate par deux. Ce réglage adapte dynamiquement le pas de mise à jour, permettant d'affiner les poids dès qu'un plateau est atteint et d'éviter ainsi les oscillations persistantes.

— **Impact du refresh\_model** : les lignes vertes verticales signalent les instants où l'optimiseur est réinitialisé (sans toucher aux poids). Chaque *refresh* est suivi d'une chute marquée de la **train\_loss** et de la **val\_loss**, traduisant la sortie du modèle d'un plateau de convergence (baisses nettes autour des époques 50, 100 et 150).

#### Pourquoi cela fonctionne-t-il ?

- (i) *Architecture sensible au momentum*<sup>14</sup> : la PolicyNet9x9 combine blocs résiduels, modules Squeeze-and-Excitation et attention globale ; un momentum trop accumulé peut « geler » ces coefficients d'attention, et le reset les libère.
- (ii) *Optimiseur RAdam + weight-decay élevé* : RAdam maintient des estimateurs adaptatifs qui, avec un weight-decay de  $2 \times 10^{-4}$ , peuvent se verrouiller dans des vallées sous-optimales. Réinitialiser ces états internes relance la descente.
- (iii) *Perte KL + label smoothing* : la divergence KL est sensible aux décalages historiques de distribution ; un reset recalcibre les sorties et atténue la surconfiance.
- (iv) *Mise en oeuvre pratique* : réinitialiser l'optimiseur (ou diviser le **momentum\_buffer** par 2) et recharger les mêmes poids débloque de nouvelles directions d'optimisation, comme l'illustrent les chutes immédiates de perte.

Ces observations sont cohérentes avec les travaux sur les redémarrages planifiés et adaptatifs [11, 12], ce qui justifie empiriquement l'emploi de **refresh\_model**.

— **Stabilisation en fin d'entraînement** : après l'époque 150, les pertes cessent de décroître significativement, ce qui reflète une convergence du modèle. L'absence de surapprentissage manifeste entre les deux courbes (train vs validation) indique également une bonne régularisation.

Ces résultats démontrent que le couplage d'un optimiseur moderne comme RAdam avec des mécanismes dynamiques de scheduler et de rafraîchissement périodique permet un apprentissage efficace et stable, même sur des problèmes complexes comme la prédiction de coups en Go. La visualisation temporelle de ces pertes constitue un outil précieux pour interpréter l'efficacité des choix méthodologiques retenus.

#### 5.2.4 Stabilisation à long terme

Pour garantir une convergence stable sur plusieurs dizaines d'itérations, plusieurs mécanismes de robustesse sont en place :

— **Monitoring de l'entropie**<sup>15</sup> : un suivi régulier de l'entropie de la distribution prédictive assure un équilibre entre confiance et diversité.

14. Le terme *momentum* désigne la mémoire exponentielle qu'utilise l'optimiseur **RAdam** pour lisser les gradients (paramètres  $\beta_1 = 0.9$  et  $\beta_2 = 0.999$  par défaut). C'est précisément ce buffer que la fonction **refresh\_model** remet à zéro (ou divise par deux) pour relancer l'apprentissage.

15. L'entropie mesure la dispersion d'une distribution de probabilité : pour  $p = (p_1, \dots, p_n)$ ,  $\mathcal{H}(p) = -\sum_{i=1}^n p_i \log p_i$ . Une entropie élevée indique une politique proche de l'uniforme (modèle hésitant), une entropie basse une politique très concentrée (modèle trop confiant). Suivre cette valeur permet de détecter les dérives : surexploitation si l'entropie chute trop, ou manque de convergence si elle reste élevée.

- **Anti-surapprentissage**<sup>16</sup> : un *dropout*<sup>17</sup> léger est appliqué dans les couches denses pour prévenir le surapprentissage. De plus, un *gradient clipping*<sup>18</sup> limite la norme globale du gradient pour prévenir les mises à jour trop brutales.
- **Reprise sur checkpoint** : chaque modèle est sauvegardé à intervalles réguliers, et l'apprentissage peut reprendre précisément là où il s'est arrêté, garantissant la continuité du processus même en cas d'interruption.

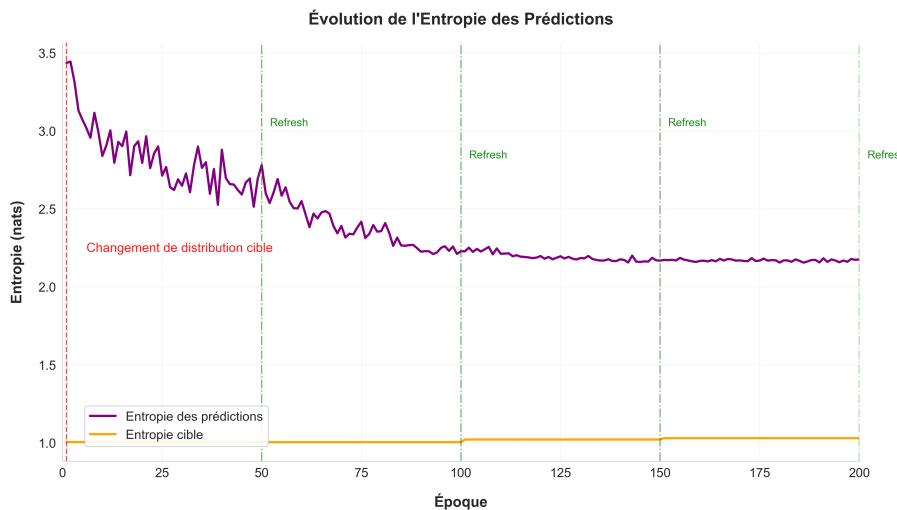


FIGURE 8 – Évolution de l'entropie des prédictions de la politique au cours de l'entraînement. Les lignes vertes verticales indiquent les phases de `refresh_model`.

**Analyse de l'évolution de l'entropie.** La figure 8 montre une décroissance régulière de l'entropie des politiques prédites durant l'entraînement. Initialement élevée ( $\sim 3.5$  nats<sup>19</sup>), l'entropie diminue progressivement, traduisant une montée en confiance du réseau sur ses prédictions.

Les phases de `refresh_model` (lignes vertes) relancent ponctuellement la dynamique d'apprentissage sans modifier l'entropie. L'entropie se stabilise ensuite autour de 2.2 nats, reflétant un bon équilibre entre confiance et diversité des politiques.

Ce suivi complète les courbes de perte, assurant que le réseau ne devient ni trop rigide ni trop aléatoire.

16. On parle d'*overfitting* (surapprentissage) lorsqu'un modèle s'adapte excessivement aux données d'entraînement, au point de perdre sa capacité à généraliser à de nouveaux exemples.

17. Le *dropout* consiste à désactiver aléatoirement un sous-ensemble de neurones pendant l'entraînement, ce qui force le réseau à apprendre des représentations plus robustes et évite la dépendance excessive à certains motifs spécifiques.

18. Le *gradient clipping* consiste à plafonner la norme du vecteur de gradient lorsqu'elle dépasse un certain seuil, stabilisant ainsi l'apprentissage, en particulier dans les réseaux profonds où des gradients excessifs peuvent causer une divergence de l'optimisation.

19. La *nat* est l'unité d'entropie utilisée lorsque le logarithme est en base  $e$  (logarithme népérien), par opposition au *bit* qui utilise la base 2. Une entropie de 1 nat correspond à  $\log_e(1/p)$  pour une probabilité  $p = e^{-1} \approx 0.368$ .

## 5.3 Value Network

### 5.3.1 Architecture détaillée

Le **Value Network** est conçu pour estimer, à partir d'une configuration du plateau, la probabilité que le joueur courant remporte la partie. Contrairement à un simple modèle de régression entièrement connecté (MLP), notre architecture exploite les propriétés spatiales du Go via un réseau convolutionnel résiduel enrichi [13].

- **Blocs résiduels avec attention globale** : le cœur du réseau repose sur **6 blocs résiduels**, chacun composé de deux convolutions  $3 \times 3$  avec normalisation par lot et fonction d'activation ReLU. À la sortie de chaque bloc, une *skip connection*<sup>20</sup> permet d'ajouter directement l'entrée à la sortie, facilitant ainsi la propagation des gradients lors de l'apprentissage profond. Cela réduit le risque de disparition ou d'explosion du gradient. Ces blocs sont enrichis par des **mécanismes d'attention de type Squeeze-and-Excitation (SE)**, qui recalibrent dynamiquement les canaux de caractéristiques pour favoriser ceux les plus pertinents dans la configuration actuelle du jeu.
- **Tête de valeur**<sup>21</sup> : à la suite des couches convolutionnelles, une **branche d'attention globale** est appliquée, suivie d'un **ensemble de couches fully-connected** (linéaires) de plus en plus réduites. Chaque couche est suivie d'un **dropout progressif**, c'est-à-dire que le taux de dropout augmente à mesure que l'on progresse dans la tête, afin de régulariser efficacement les couches profondes. La dernière couche applique une fonction d'activation **sigmoïde**, garantissant une sortie scalaire dans l'intervalle  $[0, 1]$ , interprétée comme la probabilité de victoire du joueur courant.

**Comparaison** : contrairement à une architecture entièrement connectée (MLP) quelconque, cette structure permet de capter efficacement les **motifs locaux et globaux** du plateau, comme les territoires ou les groupes capturables. L'ajout des blocs SE permet également une adaptation contextuelle des filtres appris, ce qui augmente la capacité du modèle à faire face à des configurations variées. Cela s'avère essentiel dans un jeu où les stratégies sont souvent basées sur des motifs spatiaux complexes.

### 5.3.2 Techniques partagées avec le réseau de politique

Les mêmes stratégies d'augmentation et de stabilisation que pour le réseau de politique sont reprises ici :

- **Augmentation des données** : 8 transformations géométriques (4 rotations, 2 symétries) sont appliquées à la fois à l'entrée et à la cible. Cela permet d'assurer l'**invariance par rotation et symétrie** du modèle, cohérente avec les symétries du goban.
- **Refresh\_model et scheduler** : comme pour le *Policy Network*, l'optimiseur est *rafraîchi* tous les 50 cycles (remise à zéro de ses buffers) et un scheduler *ReduceLROnPlateau*

20. Une *skip connection* ajoute l'entrée  $x$  du bloc à sa sortie  $F(x)$  pour former  $y = F(x) + x$ ; le réseau n'apprend ainsi que le « résidu »  $F(x)$ . Cette passerelle identitaire crée un chemin de gradient direct et prévient la disparition ou l'explosion du gradient dans les réseaux profonds.

21. La tête de valeur aplatis les features, les traverse via quelques couches linéaires, puis une sigmoïde rend la probabilité de victoire.

réduit automatiquement le learning rate lorsque la perte de validation stagne ; la combinaison prévient le sur-apprentissage et aide à sortir des plateaux d'optimisation.

### 5.3.3 Spécificités d'entraînement

- **Préparation de la cible** : la cible de sortie du réseau est dérivée du *résultat final d'une partie self-play*, extrait de la base de données générée par KataGo. Ce résultat (victoire/défaite) est transformé pour correspondre à un **scalaire entre 0 et 1** via la fonction sigmoïde, car notre implémentation MCTS interne fonctionne avec des valeurs positives.
- **Fonction de perte MSE** : l'apprentissage repose sur la *Mean Squared Error (MSE)*, adaptée à notre tâche de régression :

$$\ell = \frac{1}{N} \sum_{i=1}^N (v_\theta(x_i) - y_i)^2,$$

où  $v_\theta(x_i)$  est la prédiction du réseau pour l'entrée  $x_i$  et  $y_i$  la valeur cible fournie par KataGO. La régularisation  $L_2$  est appliquée séparément, via le paramètre `weight_decay` de l'optimiseur, de sorte que les poids trop grands sont pénalisés sans être inclus explicitement dans la formule de perte.

- **Optimiseur** : comme pour le *Policy Network*, nous employons **RAdam** (Rectified Adam), qui amortit les premiers pas d'apprentissage tout en conservant la mise à jour adaptative d'Adam.

### 5.3.4 Métriques de performance et monitoring

- **Suivi de la performance** : à chaque époque, nous mesurons deux métriques principales :
  - **MSE (Mean Squared Error)** : évalue l'écart quadratique moyen entre prédictions et valeurs cibles.
  - **R<sup>2</sup> (coefficient de détermination)** : mesure la proportion de la variance expliquée par le modèle.

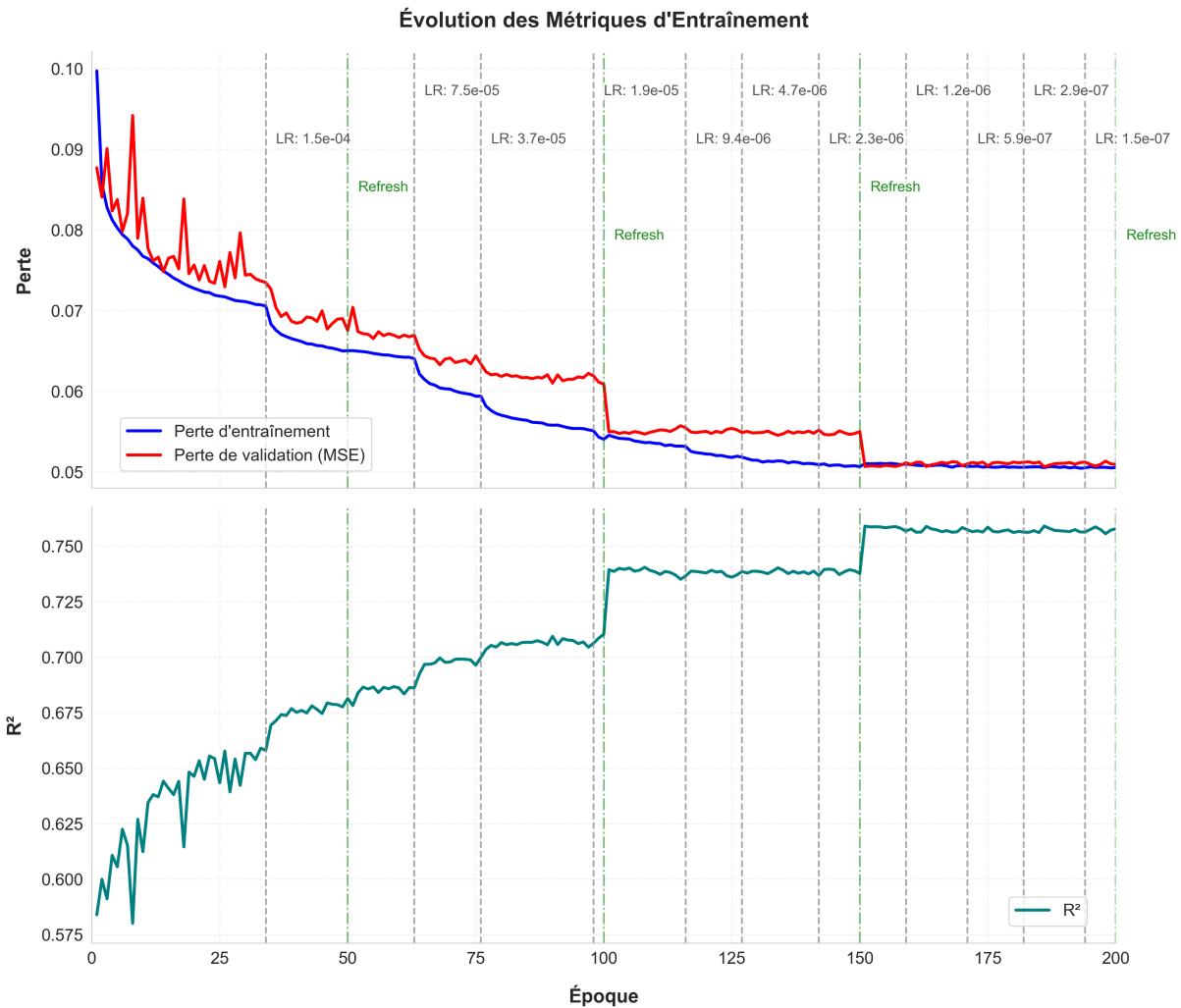


FIGURE 9 – Évolution des métriques d’entraînement du réseau de valeur (MSE et  $R^2$ ) au cours des 200 époques. Les lignes vertes indiquent les instants de `refresh_model`.

**Analyse des métriques combinées.** La figure 9 présente deux indicateurs clés pour le suivi du Value Network : la **perte quadratique moyenne** (MSE) et le **coefficients de détermination** ( $R^2$ ).

La MSE (panneau supérieur) décroît régulièrement, puis se stabilise après l'époque 150 autour de  $\sim 0.052$  (train) et  $\sim 0.057$  (val). Les phases de `refresh_model` (lignes vertes) relancent efficacement l'optimisation, tandis que le scheduler affine progressivement l'apprentissage.

Le score  $R^2$  (panneau inférieur) progresse nettement et atteint  $\sim 0.75$  en fin d'entraînement, ce qui indique que 75 % de la variance des cibles est capturée par le réseau.

L'interprétation conjointe de ces métriques confirme que le modèle est précis, stable et qu'il généralise sans sur-apprentissage visible, constituant ainsi une base solide pour la phase d'apprentissage par renforcement auto-supervisé.

## 5.4 Renforcement par auto-jeu

### 5.4.1 Concept et comparaison

L'*auto-jeu (self-play)* est une routine d'entraînement dans laquelle l'agent génère lui-même ses données en jouant contre sa dernière version, à la manière d'AlphaGo et d'AlphaGo Zero. À chaque coup, une recherche MCTS guidée par deux réseaux neuronaux (un réseau de politique pour proposer des coups, un réseau de valeur pour évaluer les positions) choisit l'action la plus prometteuse. Une fois la partie terminée, chaque position  $s$ , la politique cible  $\pi$  issue des visites MCTS et le résultat final  $z$  (1=gagné, 0=perdu, 0,5=nul) servent de cibles pour réentraîner ces deux réseaux de façon supervisée. Aucun signal de récompense externe ni partie humaine n'est nécessaire : l'agent s'améliore en se mesurant continuellement à son « moi » le plus récent.

Contrairement aux algorithmes **Q-learning**<sup>22</sup> ou **REINFORCE**<sup>23</sup>, qui exigent une fonction de récompense explicitement définie, l'auto-jeu ne retient que l'issu binaire de la partie et exploite la richesse de l'intégration du réseau de politique et de valeur dans MCTS pour guider l'apprentissage.

### 5.4.2 Cycle d'entraînement

Le processus itératif de l'entraînement par auto-jeu se déroule en trois étapes :

1. **Self-play** – Génération de parties : à chaque itération, l'agent joue 50 parties contre lui-même. MCTS interroge la *PolicyNet9x9* pour les priors et la *ValueNet9x9* pour évaluer les feuilles, exécutant plusieurs centaines de simulations par coup.
2. **Entraînement supervisé** – Mise à jour des réseaux : les triplets  $(s, \pi, z)$  collectés alimentent un *buffer d'expérience* à double file circulaire (capacité 1 000 000 d'expériences). Chaque nouvelle expérience (état du plateau, distribution MCTS, issue de la partie) y est ajoutée, écrasant les plus anciennes lorsque la capacité est atteinte. Lors de chaque mise à jour, on prélève un mini-batch de 512 positions puis on applique huit transformations diédrales (4 rotations  $\times$  2 symétries) pour l'augmentation de données. Sur ce lot, on entraîne le *PolicyNet9x9* en minimisant l'entropie croisée entre la politique prédite et  $\pi$ , et le *ValueNet9x9* en minimisant la MSE entre la valeur prédite  $v(s)$  et le résultat  $z$ , sur 25 époques.
3. **Évaluation** – Matchs de test : la version mise à jour affronte 20 parties contre la version précédente. Le taux de victoire sert à décider si l'itération est validée.

Cette boucle (génération → entraînement → test) permet une amélioration autonome et stable de l'agent, sans recours à des données externes.

### 5.4.3 Mise à jour des réseaux

**Mise à jour du réseau de politique (*PolicyNet9x9*)** Lors du self-play, *PolicyNet9x9* est interrogé par MCTS pour fournir des priors  $P(s, a)$  sur chaque coup  $a$  en position  $s$ , remplaçant ainsi les rollouts aléatoires et réduisant le coût de calcul. Avant

22. Q-learning (Watkins Dayan, 1992) : met à jour le score  $Q(s, a)$  vers « récompense + meilleur futur ». Méthode simple, sans modèle, rendue célèbre par le DQN de DeepMind qui a battu plusieurs jeux Atari.

23. REINFORCE (Williams, 1992) : après chaque épisode, augmente la probabilité des actions payantes et réduit celle des mauvaises. Exemple emblématique : tenir le pendule du Cart-Pole en équilibre.

chaque recherche, on injecte un léger **bruit de Dirichlet**<sup>24</sup> pour assurer une exploration minimale, puis, après quelques centaines de simulations, on normalise les visites  $N(s, a)$  selon une **température**<sup>25</sup>  $T$  pour obtenir la politique cible

$$\pi_a = \frac{N(s, a)^{1/T}}{\sum_b N(s, b)^{1/T}}.$$

À la fin de la partie, on met à jour les poids de *PolicyNet9x9* en minimisant la loss suivante :

$$\mathcal{L}_{\text{policy}} = - \sum_a \pi_a \ln p_{\theta}(a | s) + \epsilon \mathcal{L}_{\text{LS}} + \lambda_{\text{KL}} \text{KL}[p_{\theta_{\text{old}}} \| p_{\theta}].$$

Dans cette formule :

- $\epsilon \mathcal{L}_{\text{LS}}$  est le terme de *label smoothing* ( $\epsilon = 0,1$ )<sup>26</sup>.
- $\lambda_{\text{KL}} \text{KL}[\dots]$  est la pénalisation KL ( $\lambda_{\text{KL}} = 0,05$ )<sup>27</sup>.

Le premier terme (entropie croisée) aligne la politique apprise sur la cible  $\pi$ , tandis que les deux termes ci-dessus assurent une évolution progressive et stable du réseau. Si les logits de l'itération précédente ne sont pas disponibles, on se limite au seul label smoothing pour la régularisation.

---

24. Le **bruit de Dirichlet** consiste à mélanger la probabilité initiale  $P(s, a)$  avec un tirage  $\eta \sim \text{Dir}(\alpha)$  (par ex.  $\alpha = 0,03$ ) :  $\tilde{P} = (1 - \varepsilon)P + \varepsilon \eta$ . Cette méthode, popularisée par AlphaGo Zero, garantit qu'aucun coup n'ait une probabilité nulle lors des premières simulations.

25. La **température**  $T$  règle le compromis exploration/exploitation :  $T > 1$  aplatis la distribution pour favoriser l'exploration, tandis que  $T < 1$  concentre la probabilité sur les coups les plus visités.

26. Le *label smoothing* mélange la cible MCTS avec une distribution uniforme pour limiter la surconfiance du réseau et améliorer sa généralisation.

27. Le `kl_weight` détermine la force de la pénalité KL : à 0 aucune régularisation, à 0,05 un compromis stable entre conservatisme et exploration, au-delà de 0,1 la régularisation devient trop contraignante, ralentissant l'adaptation aux nouvelles stratégies.

### Remarque

**Température vs bruit de Dirichlet** La température agit directement sur la distribution de visites normalisée :

$$\pi_a = \frac{N(s, a)^{1/T}}{\sum_b N(s, b)^{1/T}},$$

où un  $T > 1$  aplatis la distribution et favorise l'exploration, tandis qu'un  $T < 1$  concentre la probabilité sur les coups les plus visités. En pratique, la température décroît au fil de la partie pour passer d'une recherche large à des choix plus déterministes. Le *bruit de Dirichlet* est injecté avant même le comptage des visites : on mélange la distribution initiale  $P(s, a)$  avec un échantillon  $\eta \sim \text{Dir}(\alpha)^a$ ,

$$\tilde{P} = (1 - \varepsilon) P + \varepsilon \eta,$$

assurant qu'aucun coup n'ait une probabilité nulle dès les premières simulations. Alors que la température module  $\pi$  après la recherche, le bruit de Dirichlet garantit une exploration minimale dès l'expansion de l'arbre.

Ces deux mécanismes, complémentaires, équilibrivent exploration et exploitation tout au long du self-play.

a. La distribution de Dirichlet  $\text{Dir}(\alpha)$  est une loi de probabilité continue définie sur le simplexe  $\{x \in R^K \mid x_i \geq 0, \sum_i x_i = 1\}$ . Pour un paramètre de concentration unique  $\alpha > 0$  (cas symétrique), elle génère un vecteur aléatoire  $\eta = (\eta_1, \dots, \eta_K)$  tel que

$$f(\eta_1, \dots, \eta_K) \propto \prod_{i=1}^K \eta_i^{\alpha-1}.$$

Un  $\alpha < 1$  favorise des distributions « sparse » (peu de composantes non nulles), tandis qu'un  $\alpha > 1$  tend vers une distribution plus uniforme.

**Mise à jour du réseau de valeur (*ValueNet9x9*)** Lors du self-play, chaque nouvel état  $s$  exploré par le MCTS est évalué par *ValueNet9x9*, qui retourne une prédiction  $v(s) \in [0, 1]$  de la probabilité de victoire du joueur au trait. Cette estimation remplace les simulations aléatoires « rollouts », réduisant considérablement le coût de calcul tout en assurant une évaluation uniforme tout au long de l'arbre.

Après la fin de la partie, on récupère le résultat final  $z \in \{0, 0.5, 1\}$  (0 pour une défaite, 1 pour une victoire, 0.5 pour un nul). Pour chaque position  $s_i$  rencontrée pendant la partie, on calcule l'erreur quadratique moyenne :

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (v(s_i) - z)^2$$

et on ajuste les poids de *ValueNet9x9* afin de minimiser cette MSE sur l'ensemble des positions. Ainsi, le réseau apprend à prédire de plus en plus précisément l'issue des parties qu'il génère.

#### 5.4.4 Lancement de l'entraînement

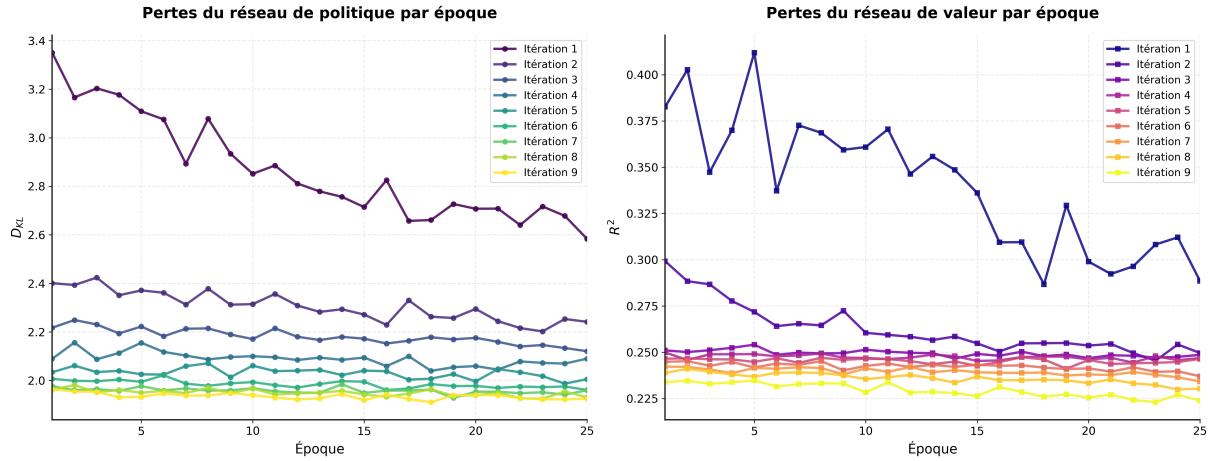


FIGURE 10 – Pertes des réseaux de politique (gauche) et de valeur (droite).

**Lecture détaillée de la figure 10.** Les deux graphiques rapportent l'évolution de la perte sur **25 époques** pour les **9 premières itérations** du cycle d'auto-jeu, chaque itération reposant sur **50 nouvelles parties**. Chaque partie génère en moyenne une soixantaine de positions (coups), produisant ainsi environ 3 000 triplets  $(s, \pi, z)$  par itération, qui sont automatiquement ajoutés au buffer d'expérience décrit plus haut. Ces triplets alimentent ensuite l'échantillonnage et l'augmentation de données avant la mise à jour des réseaux, comme expliqué précédemment. La courbe violette la plus sombre représente l'itération 1, puis chaque teinte s'éclaire jusqu'au jaune de l'itération 9, ce code-couleur étant rappelé dans la légende.

- **Politique (gauche).** L'itération 1 débute à 3,3 et termine autour de 2,6. À l'itération 4 la perte s'ouvre déjà sous 2,2, et l'itération 9 flotte juste au-dessus de 1,9 avec une pente bien plus plate, signe que la politique converge plus vite parce que les cibles MCTS sont devenues plus informatives.
- **Valeur (droite).** La même dynamique est visible : on part de 0,40, on descend rapidement à 0,30 dès l'itération 2, puis on plafonne à 0,23–0,24 aux itérations 8–9. Le modèle de valeur gagne donc en précision au même rythme que se sophistique le jeu produit par le self-play.

Grâce à l'ajout de 50 parties par itération (soit environ 3 500 positions), la courbe de perte décroît plus rapidement et atteint son plateau plus tôt. Avec davantage de temps et de puissance de calcul, nous aurions pu générer un nombre plus important de parties d'auto-jeu sans avoir à gérer un buffer d'expérience, ce qui aurait permis un apprentissage encore plus robuste des deux réseaux. Pour vérifier que ces progrès numériques se traduisent effectivement sur le goban, nous examinons désormais le taux de victoire de chaque nouvelle version face à la précédente.

### 5.4.5 Évolution du taux de victoire

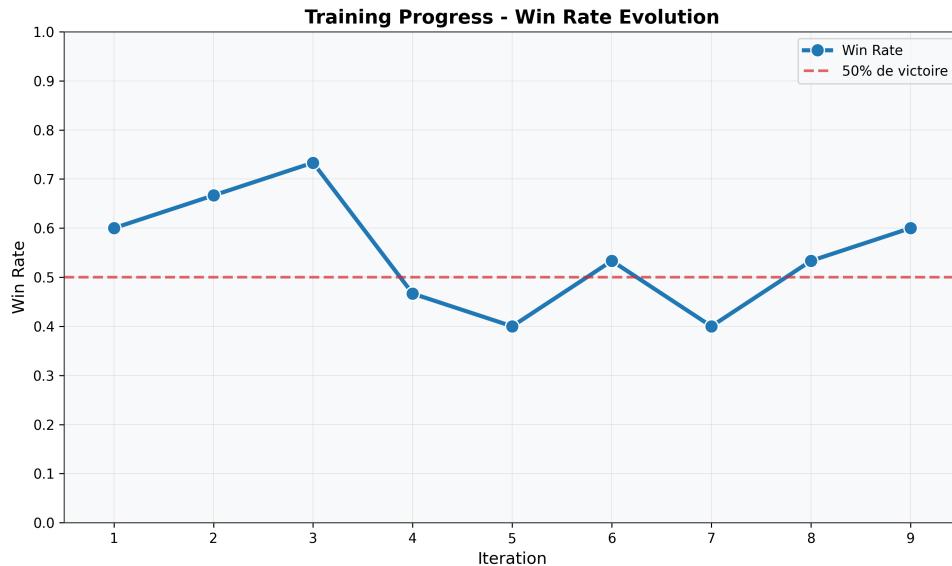


FIGURE 11 – Taux de victoire de l’itération  $k$  face à l’itération  $k - 1$  (20 parties par point, ligne pointillée : seuil de 50 %).

Chaque cycle suivait le même schéma : 50 nouvelles parties d’auto-jeu, 25 époques d’apprentissage, puis 20 parties d’évaluation. Nous visualisons ici les résultats des parties d’évaluation. Les trois premières itérations profitent pleinement de cette routine : le taux de victoire bondit de 60 % à 73 %, preuve que le réseau découvre encore des motifs très différents de ceux de la version précédente. À partir de la quatrième, l’écart se réduit : la courbe oscille autour de la barre des 50 % et ne la franchit plus qu’épisodiquement (rebonds des itérations 6, 8 et 9). En d’autres termes, même si les réseaux deviennent plus précis à chaque itération (comme le montre la figure 10), cette amélioration ne se traduit plus forcément par de meilleures performances en jeu : l’agent atteint un palier où chaque nouveau progrès se fait plus rare.

Pour repousser ce palier, il faudrait augmenter le nombre de parties d’auto-jeu (voire se passer du buffer d’expérience actuel) afin de relancer l’exploration et échapper à la zone de rendement décroissant. Malheureusement, chaque cycle complet (auto-jeu, entraînement et test) prenait près de 4 heures sur notre matériel ( $2 \times$  Nvidia Titan XP), ce qui nous empêchait d’enchaîner un plus grand nombre de parties en raison du temps limité.

Nous avons donc cherché à déterminer si, malgré cette stagnation, notre modèle restait tout de même supérieur à un MCTS classique dépourvu de toute aide neuronale.

## 5.5 Évaluation finale : MCTS Neuronal face à un MCTS « classique »

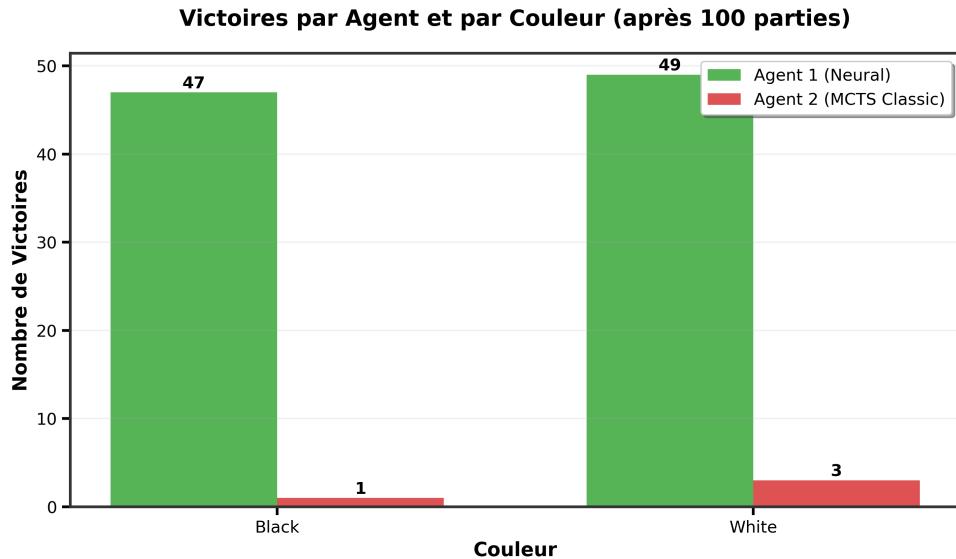


FIGURE 12 – Bilan des **100 parties** jouées entre notre *Neural MCTS* et un MCTS « classique » (800 simulations / coup). Chaque barre indique le nombre de victoires obtenues lorsqu'un agent joue Noir ou Blanc.

**Protocole.** Nous avons opposé la version finale de notre agent *Neural MCTS* à un MCTS dépourvu de réseau (politique uniformément aléatoire et valeur terminale uniquement), en lui accordant exactement le même budget de 800 simulations par coup. Les deux moteurs ont disputé 100 parties, chacun prenant 50 fois la couleur noire et 50 fois la couleur blanche, sous les mêmes règles de komi (6.5). L'évaluation totale a duré 5 jours.

**Résultats.** La figure 12 met en évidence une supériorité écrasante de la version neuronale : 47 victoires sur 50 avec les pierres noires et 49 sur 50 avec les pierres blanches, soit un total de **96 %** de parties gagnées.

**Analyse.** Ces résultats confirment que l'adjonction des réseaux de politique et de valeur ne se limite pas à faire baisser les courbes de perte : elle améliore concrètement le niveau de jeu. Le réseau de politique *PolicyNet9x9* guide le MCTS vers les coups les plus prometteurs dès les premières simulations, tandis que le réseau de valeur *ValueNet9x9* évalue rapidement les positions, réduisant le bruit inhérent aux simulations aléatoires. À budget égal, la version neuronale tire donc un bien meilleur parti de ses 800 simulations, validant la stratégie d'apprentissage supervisé sur données de self-play adoptée.

## 5.6 Vers des évaluations plus fiables

Sous les règles japonaises, le score final dépend de la bonne identification des *groupes morts*, c'est-à-dire des chaînes condamnées à la capture faute de posséder deux « yeux »<sup>28</sup>. Notre script applique, après les passes, quelques heuristiques maison pour déterminer les territoires des deux joueurs. Or, sur des positions complexes, nous avons relevé jusqu'à **40**

<sup>28</sup>. Un **œil** est une intersection vide entièrement entourée par des pierres d'une même couleur. Avec deux yeux, un groupe est considéré vivant, car l'adversaire ne peut le capturer sans se suicider.

**points d'écart** par rapport au comptage automatique de **Katrain**<sup>29</sup>. Ce problème est principalement dû à la difficulté de détecter des groupes morts, car il s'agit d'un commun accord entre les adversaires à la fin de la partie. Dans certains cas, même Katrain réclame une validation manuelle. Le problème n'est donc pas anecdotique.

Lorsque deux modèles ont des niveaux très différents (par exemple notre Neural MCTS contre le MCTS «classique»), une erreur de comptage de quelques points reste négligeable : l'un des deux agents l'emporte haut la main, et le verdict ne change pas. En revanche, quand on compare deux versions successives de notre agent—dont la force est presque identique—une unique capture mal identifiée suffit à inverser l'issue d'une partie. Dans ces conditions, le critère de promotion fixé à « 55% de victoires » devient trop instable : un léger décalage aléatoire peut faire passer le score de 54% à 56%, ou inversement. C'est pourquoi nous avons préféré conserver la version tout juste entraînée (la plus récente) lors de l'entraînement par auto-jeu, même lorsque son avantage statistique par rapport à l'itération précédente était inférieure.

Effectuer le marquage manuellement n'est pas viable : il faudrait corriger plusieurs centaines de parties pour obtenir une statistique significative, soit plus de seize heures de travail humain à chaque comparaison interne. Pour fiabiliser nos évaluations, plusieurs pistes s'offrent à nous :

1. **Réseau « vie–mort » dédié** : entraîner un nouveau réseau de neurones capable d'identifier automatiquement les groupes condamnés.
2. **Passage aux règles chinoises**<sup>30</sup> : ainsi le problème de marquage des groupes morts disparaît. Seulement, KataGO est initialisé avec les règles japonaises.
3. **Évaluation par moteur de référence**, : s'appuyer sur des moteurs existants (KataGo, KaTrain ou Leela Zero) pour recalculer automatiquement le score en fin de partie et repérer les écarts anormaux. Par manque de temps, cette solution n'a pas encore été implémentée.
4. **Self-play prolongé** : après les passes, relancer un sous-MCTS jusqu'à la capture effective des groupes douteux, afin de figer un résultat incontestable.

Tant qu'une de ces solutions n'est pas en place, nos métriques resteront parfaitement fiables pour des écarts de niveau marqués, mais devront être interprétées avec prudence lorsque l'on compare deux réseaux de force comparable.

---

29. Interface basée sur KataGo, dotée d'une détection de groupes morts.

30. Sous les **règles chinoises**, le décompte se limite aux intersections occupées plus territoires. Les pierres capturées sont simplement retirées, ce qui supprime la notion de groupes morts. Notre choix initial des règles japonaises vient du paramétrage par défaut de KataGo, mais il n'est pas figé.

## 6 Budget

Le coût total théorique de notre projet s'élève à 22 280 €. Ce budget a été établi dans une optique d'évaluation économique du travail fourni, bien que certaines dépenses soient fictives dans le cadre de ce projet étudiant. En particulier, la rémunération des étudiants n'est pas réelle : elle permet simplement d'estimer la valeur de leur contribution si elle avait été rémunérée dans un contexte professionnel. En revanche, les rémunérations des encadrants (enseignants, conseiller de projet) sont effectives, car ces heures correspondent à un encadrement réel fourni dans le cadre de leur activité professionnelle.

Le poste de dépense principal est celui du personnel, qui représente 17 082 €, soit la quasi-totalité du budget. Ce poste regroupe les 1 080 heures de travail des étudiants, ainsi que celles des encadrants académiques. Le second poste de dépense non négligeable est celui lié à l'utilisation des équipements. En effet, il a fallu entraîner notre réseau de neurones. Pour ce faire nous avons utilisé deux GPU coûtant 2 134€ chacune. Les autres dépenses budgétaires sont secondaires : les coûts liés à l'utilisation des équipements autres que les GPU (690 €), incluant notamment la location ponctuelle d'un espace, et les coûts techniques complémentaires, tels que l'hébergement web ou l'usage de services de cloud computing (totalisant à peine 150 €).

Bien que notre projet reste modeste dans ses moyens, il est intéressant de le mettre en parallèle avec un projet de référence dans le domaine de l'intelligence artificielle appliquée au jeu de Go : AlphaGo, développé par DeepMind. Le coût de développement d'AlphaGo est estimé à environ 50 millions de dollars, incluant des infrastructures matérielles de pointe (GPU en masse, serveurs), une équipe de chercheurs en IA, des ingénieurs logiciels, ainsi que des experts du jeu de Go.

Notre budget représente donc 1.1 % de celui d'AlphaGo. Cette différence d'échelle met en évidence les objectifs distincts : alors qu'AlphaGo visait la performance ultime et la victoire contre les meilleurs joueurs professionnels mondiaux, notre projet s'inscrit avant tout dans une démarche pédagogique et exploratoire, centrée sur la compréhension des algorithmes, la conception d'un environnement fonctionnel, et la mise en oeuvre d'une IA adaptée à des plateaux de tailles variées.

<b>1. Coût des achats</b>	Quantité	Coût unitaire (€)	Coût total TTC (€)
Matières premières	0	0,00	0,00
Composants	0	0,00	0,00
Documents	0	0,00	0,00
Sous-traitance	0	0,00	0,00
<b>Total achats</b>			<b>0,00</b>

<b>2. Utilisation des équipements</b>	Qté / h	Coût unitaire (€)	Coût total TTC (€)
Cartes GPU (2 × Titan XP)	2	2 134,00	4 268,00
Autres équipements divers	1	690,00	690,00
Location ponctuelle d'un espace (h)	9	10,00	90,00
<b>Total équipements</b>			<b>5 048,00</b>

<b>3. Coût de personnel</b>	Nb d'heures	Tarif brut (€ / h)	Coût total (€)
Élèves du PE (6 étudiants)	1 080	11,65	12 582,00
Enseignant tuteur pédagogique	27	100,00	2 700,00
Conseiller de projet + Président de jury	18	100,00	1 800,00
Personnel technique	0	18,00	0,00
<b>Total personnel</b>	1 125	—	<b>17 082,00</b>

<b>4. Communication</b>	Quantité	Coût unitaire (€)	Coût total TTC (€)
Hébergement site web (24 mois)	24	2,50	60,00
<b>Total communication</b>			<b>60,00</b>

<b>5. Autres coûts</b>	Quantité	Coût unitaire (€)	Coût total TTC (€)
Cloud computing (heures)	30	3,00	90,00

<b>TOTAL projet (TTC)</b>	<b>22 280,00</b>
Trésorerie (subventions)	300,00
<b>Consommation prévisionnelle de trésorerie</b>	<b>90,00</b>

TABLE 2 – Budget récapitulatif du projet

## Conclusion

Ce projet avait pour ambition de concevoir une intelligence artificielle capable de jouer au Go à un niveau amateur, intégrée dans une interface graphique interactive et accessible. Les objectifs principaux (créer une interface respectant toutes les règles du jeu, développer une IA compétitive, et ajouter des fonctionnalités annexes) ont été atteints pour l'essentiel, en particulier l'interface complète et une IA compétitive, tandis que certaines fonctionnalités optionnelles (mode réseau, tutoriel) restent à développer. Nous avons d'abord réalisé une interface complète et fonctionnelle permettant de jouer sur différentes tailles de goban, aussi bien en mode "bac à sable" qu'en affrontement entre deux humains ou contre l'IA. Cette interface respecte l'ensemble des règles du Go et propose une expérience fluide pour l'utilisateur comme pour le développeur (historique des coups, ergonomie claire, etc.).

Concernant les performances de l'IA, la version initiale reposant uniquement sur Monte Carlo Tree Search offrait un niveau de jeu trop limité. Nous avons donc intégré un réseau de politique supervisé, entraîné sur une base de parties issues de KataGo, ainsi qu'un réseau de valeur chargé d'estimer la qualité des positions : ces deux réseaux guident désormais le MCTS, améliorant significativement la pertinence des coups. Un entraînement par auto-jeu (self-play) a ensuite permis de constituer une seconde base de parties, distincte de celle utilisée pour l'apprentissage supervisé, renforçant la coordination entre les deux réseaux. Par ailleurs, la stratégie d'apprentissage inclut un mécanisme original de réinitialisation périodique du momentum de l'optimiseur, ce qui stabilise et accélère la convergence du modèle. Malgré ces avancées, le niveau de jeu obtenu reste modeste comparé aux moteurs professionnels, en raison des ressources limitées allouées au projet.

Ce socle ouvre plusieurs perspectives d'amélioration. D'une part, l'IA pourrait être encore renforcée par un entraînement sur un volume bien plus important de parties, voire par davantage d'auto-jeu, afin d'affiner ses réseaux de neurones. D'autre part, l'interface elle-même pourrait évoluer vers une version web et intégrer des outils pédagogiques supplémentaires (aide contextuelle, tutoriel interactif) pour enrichir l'expérience utilisateur. Enfin, la mise en place d'un mode en réseau (permettant à deux joueurs distants de s'affronter via un serveur) constituerait une extension naturelle pour élargir l'usage de l'application : non seulement ce mode faciliterait le jeu à distance, mais il permettrait aussi de récolter automatiquement des parties en ligne pour constituer notre propre base de données. Ces données pourraient ensuite être publiées sous l'égide de l'École Centrale de Lyon dans le cadre du « Jeu de Go de l'École Centrale Lyon ».

## Bibliographie

- [1] Didier LESESVRE. *Le Go et les mathématiques, un voyage au confluent des deux mondes*. URL : <https://lesesvre.perso.math.cnrs.fr/go.pdf>.
- [2] British Go ASSOCIATION. *History of Go-playing Programs*. URL : <https://www.britgo.org/computergo/history>.
- [3] Gauthier PICARD. *Algorithme Minimax et élagage*.
- [4] Gunnar FARNEBÄK JONH TROMP. *Combinatorics of GO*.
- [5] Cameron B. BROWNE et al. “A Survey of Monte Carlo Tree Search Methods”. In : *IEEE Transactions on Computational Intelligence and AI in Games* (2012). DOI : 10.1109/TCIAIG.2012.2186810.
- [6] Bruno BOUZY. *Les Ensembles Flous Au Jeu De Go*. 1996.
- [7] Adrian GLAUBEN. *Replacing PUCT with a planing model*. URL : <https://ml-research.github.io/papers/glauben2022replacing.pdf>.
- [8] Jie HU, Li SHEN et Gang SUN. “Squeeze-and-Excitation Networks”. In : *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, p. 7132-7141.
- [9] Ryan Cotterell CLARA MEISTER Elizabeth Salesky. *Generalized Entropy Regularization or : There's Nothing Special about Label Smoothing*. URL : <https://arxiv.org/pdf/2005.00820>.
- [10] Ian GOODFELLOW, Yoshua BENGIO et Aaron COURVILLE. *Deep Learning*. MIT Press, 2016. URL : <https://www.deeplearningbook.org/>.
- [11] Bao WANG et al. “Scheduled Restart Momentum for Accelerated Stochastic Gradient Descent”. In : *Advances in Neural Information Processing Systems*. 2020. URL : <https://arxiv.org/pdf/2002.10583>.
- [12] Brendan O'DONOUGHUE et Emmanuel CANDÈS. *Adaptive Restart for Accelerated Gradient Schemes*. 2012. arXiv : 1204.3982 [cs.LG]. URL : <https://arxiv.org/pdf/1204.3982>.
- [13] Ilya Sutskever CHRIS J. MADDISON Aja Huang et David SILVER. *Move Evaluation in Go Using Deep Convolutional Neural Networks*. URL : <https://arxiv.org/pdf/1412.6564>.
- [14] Jean Hossenlopp FRANÇOIS MIZESSYN. *Règle du jeu de go*. URL : [https://jeudego.org/\\_pdf/regleGoCourte.pdf](https://jeudego.org/_pdf/regleGoCourte.pdf).

## Annexes

- **Algorithme Alpha-Bêta** : Algorithme d'optimisation de la recherche MiniMax qui évalue les branches non explorées d'un arbre de jeu en s'appuyant sur les résultats déjà calculés, éliminant ainsi les sous-arbres qui ne peuvent pas influencer le résultat.
- **AlphaGo** : Programme d'intelligence artificielle développé par DeepMind, capable de jouer au go à très haut niveau. Il a été le premier logiciel à vaincre un champion humain de go (Lee Sedol en 2016) en combinant des réseaux de neurones de politique et de valeur avec l'algorithme.
- **AlphaGo Zero** : Version ultérieure d'AlphaGo (DeepMind, 2017) s'entraînant par auto-apprentissage sans données humaines. Doté d'un réseau de neurones unique à deux têtes (policy et value) et entraîné uniquement par auto-jeu, AlphaGo Zero a rapidement surpassé la performance de la version originale d'AlphaGo.
- **Apprentissage par renforcement** : Paradigme d'apprentissage où un agent apprend à prendre des décisions en interagissant avec un environnement et en recevant des récompenses. L'agent améliore graduellement sa stratégie (politique) afin de maximiser la récompense cumulative reçue au cours du temps.
- **Apprentissage supervisé** : Mode d'entraînement d'un modèle à partir d'exemples étiquetés. Le modèle reçoit des entrées avec les sorties attendues (labels) et ajuste ses paramètres en minimisant l'erreur entre sa prédiction et la sortie correcte, jusqu'à apprendre la correspondance entrée-sortie.
- **Atari** : Terme du jeu de go désignant la situation où une pierre ou un groupe de pierres n'a plus qu'une seule liberté restante, c'est-à-dire qu'au prochain coup de l'adversaire ce groupe sera capturé s'il n'est pas sauvé.
- **Auto-jeu (self-play)** : Méthode où une intelligence artificielle joue des parties contre elle-même afin de générer ses propres données d'entraînement ou d'améliorer sa force de jeu sans intervention humaine extérieure.
- **Divergence de Kullback–Leibler (KL)** : Mesure de dissimilarité entre deux distributions de probabilité. La KL quantifie la perte d'information lorsqu'une distribution approximative  $q$  est utilisée pour estimer une distribution réelle  $p$  – plus la divergence KL  $D_{KL}(p\|q)$  est élevée, plus  $q$  s'écarte de  $p$ . Dans le projet, la KL sert de fonction de perte pour rapprocher la distribution prédictive de celle de référence.
- **Dropout** : Technique de régularisation consistant à désactiver aléatoirement une fraction des neurones d'un réseau pendant l'apprentissage, afin de forcer le réseau à apprendre des caractéristiques plus générales et à éviter une trop forte spécialisation sur certains neurones.
- **Early stopping** : Stratégie d'arrêt anticipé de l'entraînement d'un modèle lorsque la performance de validation ne s'améliore plus (voire se dégrade). En stoppant l'apprentissage avant sur-ajustement, on prévient le surapprentissage et on obtient un modèle qui généralise mieux.
- **Entropie** : Mesure de la dispersion d'une distribution de probabilité. Pour une distribution  $p = (p_1, \dots, p_n)$ , l'entropie est définie par  $H(p) = -\sum_{i=1}^n p_i \log p_i$ . Une entropie élevée indique une distribution presque uniforme (incertitude élevée), tandis qu'une entropie faible correspond à une distribution très concentrée (confiance élevée).

- **FoxDataset** : Ensemble de données de parties de go utilisées pour un entraînement supervisé initial du réseau de politique ( $19 \times 19$ ). Il est composé de parties de joueurs professionnels (du niveau 1p à 9p), fournissant des positions d'entrée et les coups joués correspondants comme cibles à apprendre.
- **Goban** : Terme désignant le plateau de jeu de go. Un goban standard est une grille de  $19 \times 19$  intersections, mais des formats réduits (par ex.  $9 \times 9$ ) sont utilisés pour l'entraînement et les tests. Les pierres sont placées sur les intersections de la grille.
- **Gradient clipping** : Technique consistant à limiter la norme du vecteur de gradient lors de la rétropropagation, en la plafonnant à un seuil maximal fixé. Cela empêche les mises à jour de poids excessives et stabilise l'apprentissage, en particulier dans les réseaux profonds où de très grands gradients peuvent faire diverger l'optimisation.
- **KataGo** : Programme d'IA open-source pour le jeu de go (publié en 2019) qui implémente un entraînement de type AlphaGo Zero amélioré. KataGo est l'un des moteurs de go open-source les plus puissants, capable de battre des joueurs professionnels humains, et a été utilisé dans ce projet pour générer automatiquement des parties annotées pour l'apprentissage.
- **Label smoothing** : Technique de régularisation qui consiste à lisser les étiquettes de sortie lors de l'entraînement d'un modèle de classification. Plutôt que d'utiliser des sorties cibles « dures » (par ex. 1 pour la classe correcte, 0 sinon), on attribue une petite probabilité non nulle aux autres classes. Ce lissage des labels (par ex. distribuer un  $\varepsilon$  de probabilité uniformément sur les classes non-cibles) évite que le modèle ne soit trop confiant et améliore sa généralisation.
- **Liberté** (au go) : Intersection vide adjacente à une pierre ou à un groupe de pierres sur le plateau. Chaque pierre a jusqu'à 4 libertés initialement (une par direction orthogonale). Si un groupe de pierres n'a plus de libertés disponibles, il est immédiatement capturé par l'adversaire.
- **MAE** (Mean Absolute Error) : Erreur moyenne absolue, une métrique mesurant la moyenne des valeurs absolues des écarts entre les prédictions du modèle et les valeurs réelles. Contrairement à la MSE, la MAE pénalise de manière linéaire les erreurs, la rendant moins sensible aux valeurs aberrantes.
- **MCTS** (Monte Carlo Tree Search) : Algorithme d'exploration de l'espace de jeu par simulations aléatoires. MCTS construit progressivement un arbre de recherche en répétant quatre étapes (sélection, expansion, simulation (jeu aléatoire jusqu'à une fin de partie) et rétropropagation du résultat) afin d'estimer la valeur des coups possibles. Il équilibre exploration (tester de nouveaux coups) et exploitation (approfondir les coups prometteurs) pour converger vers le meilleur coup à jouer.
- **MLP** (Multilayer Perceptron) : Réseau de neurones constitué exclusivement de couches entièrement connectées (denses). Chaque couche réalise une transformation linéaire suivie d'une fonction d'activation non-linéaire. Ce type de réseau, ne comportant pas de structure spatiale, opère sur des données vectorielles plutôt que sur des images ou des grilles.
- **MSE** (Mean Squared Error) : Erreur quadratique moyenne, métrique calculant la moyenne des carrés des écarts entre les prédictions et les valeurs cibles. Couramment utilisée comme fonction de perte en régression, la MSE pénalise fortement les erreurs les plus grandes (du fait de l'élévation au carré), ce qui peut la rendre sensible aux valeurs aberrantes.

- **Momentum** : Terme désignant l'inertie utilisée dans certains algorithmes d'optimisation pour accélérer la convergence. Le momentum accumule une fraction (par ex. 90%) du gradient précédent dans la mise à jour courante, ce qui a pour effet de lisser les variations de gradient et d'orienter la descente de gradient dans une direction plus stable.
- **Nat** : Unité d'entropie utilisant le logarithme naturel (base  $e$ ) au lieu du logarithme base 2 (bit). Par exemple, une entropie de 1 nat correspond à la quantité d'information d'un événement de probabilité  $p = e^{-1} \approx 0,368$ . (À titre de comparaison, 1 bit d'entropie correspond à  $p = 0,5$ ).
- **Normalisation par lots** (Batch Normalization) : Technique consistant à recentrer et re-normaliser les activations d'une couche de neurones pour chaque mini-lot de données pendant l'entraînement. En contrôlant la distribution des sorties de couche (moyenne et variance) à chaque batch, on accélère et stabilise l'apprentissage des réseaux profonds.
- **Optimiseur** : Algorithme chargé de mettre à jour les poids d'un réseau de neurones lors de l'apprentissage, en se basant sur le gradient de la fonction de perte afin de minimiser celle-ci. Exemples d'optimiseurs courants : SGD, Adam, RMSProp, etc.
- **Réseau de politique** (Policy Network) : Réseau de neurones qui prédit, pour une position donnée du plateau, une distribution de probabilité sur les coups jouables. Il évalue ainsi les mouvements les plus prometteurs selon l'expérience de jeu apprise, et guide la recherche du MCTS vers ces coups candidats.
- **Réseau de valeur** (Value Network) : Réseau de neurones qui estime, pour une position donnée, la probabilité de victoire du joueur courant. Il fournit une évaluation numérique de la position (généralement entre 0 et 1) indiquant les chances de gain pour le joueur actif, et sert de retour pour orienter les simulations de MCTS.
- **Réseau policy-value** : Architecture de réseau de neurones combinée comprenant deux têtes de sortie distinctes – l'une pour la politique (distribution des coups) et l'autre pour la valeur (évaluation de la position). Les premières couches du réseau sont partagées entre ces deux objectifs, ce qui permet au modèle d'apprendre des représentations communes servant à la fois à prédire le meilleur coup et la probabilité de victoire.
- **ReLU** (Rectified Linear Unit) : Fonction d'activation définie par  $\text{ReLU}(x) = \max(0, x)$ . Elle remplace les valeurs négatives par 0 et laisse passer les positives linéairement, introduisant de la non-linéarité tout en évitant la saturation du gradient pour  $x > 0$ , ce qui facilite l'apprentissage des réseaux profonds.
- **RAdam** (Rectified Adam) : Variante de l'optimiseur Adam qui intègre un mécanisme de « rectification » de la variance afin d'améliorer la stabilité en début d'apprentissage. Concrètement, RAdam temporise l'adaptation du taux d'apprentissage tant que la variance estimée des gradients (second moment) est insuffisante, évitant ainsi des pas initiaux trop grands et une divergence potentielle de l'optimisation.
- **ResNet** (Réseau Résiduel) : Architecture de réseau de neurones introduisant des connexions résiduelles (*skip connections*) entre l'entrée et la sortie de certains blocs de couches. Une connexion résiduelle ajoute directement le signal d'entrée d'un bloc à sa sortie, de sorte que le bloc n'apprenne que la différence (« résidu ») à ajouter. Ce procédé maintient des gradients plus stables et permet d'empiler de très nombreuses couches sans dégradation des performances.

- **Scheduler** (planificateur de taux d'apprentissage) : Mécanisme qui ajuste automatiquement le *learning rate* (taux d'apprentissage) au cours de l'entraînement en fonction du progrès. Par exemple, le scheduler `ReduceLROnPlateau` diminue le taux d'apprentissage d'un certain facteur (par ex.  $0,5\times$ ) lorsqu'aucune amélioration de la perte de validation n'est observée pendant un certain nombre d'époques, afin de continuer à affiner les poids du réseau.
- **Sigmoïde** : Fonction d'activation logistique définie par  $\sigma(x) = \frac{1}{1+e^{-x}}$ . Elle transforme toute valeur réelle en un nombre dans l'intervalle  $(0, 1)$ , pouvant être interprété comme une probabilité. En réseau de neurones, une couche de sortie sigmoïde est souvent utilisée pour prédire une probabilité (par exemple, de victoire) dans le cas binaire.
- **Softmax** : Fonction d'activation qui convertit un vecteur de scores arbitraires  $(\ell_1, \dots, \ell_n)$  en une distribution de probabilité  $(p_1, \dots, p_n)$ . Elle applique l'exponentielle à chaque score puis le normalise par la somme de toutes les exponentiations : 
$$\text{softmax}(\ell)_i = \frac{\exp(\ell_i)}{\sum_{j=1}^n \exp(\ell_j)}$$
.
- **Squeeze-and-Excitation (SE)** : Module d'attention pour réseaux convolutionnels qui recalibre dynamiquement l'importance de chaque canal de caractéristiques. Un bloc SE effectue d'abord un *squeeze* (agrégation globale, typiquement une moyenne globale) des cartes de caractéristiques, puis passe ce vecteur agrégé dans un petit réseau entièrement connecté pour générer des coefficients d'activation (*excitation*) appliqués canal par canal. Cela permet de moduler les filtres du réseau en fonction du contexte de l'entrée.
- **Weight decay** : Technique de régularisation consistant à pénaliser l'amplitude des poids du réseau pendant l'entraînement. À chaque mise à jour, une petite fraction de chaque poids est soustraite (proportionnelle à sa valeur), ce qui tend à les maintenir plus petits. Le weight decay limite ainsi le surapprentissage sans coût de calcul significatif.



## Règle du jeu de go

### Déroulement du jeu

Le go se joue à deux sur une **grille** de 19x19 lignes, avec des **pierres** noires et blanches. Parfois, des grilles plus petites sont utilisées, en particulier pour l'initiation.

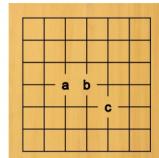
**Celui qui commence joue avec les pierres noires.** À tour de rôle, les joueurs posent une pierre de leur couleur sur une intersection inoccupée de la grille ou bien ils passent (essentiellement pour indiquer qu'ils pensent la partie terminée).

Le but du jeu est d'occuper ou d'entourer avec ses pierres le plus grand nombre possible d'intersections de la grille, la partie s'arrêtant lorsque les deux joueurs pensent que ce nombre ne variera plus.

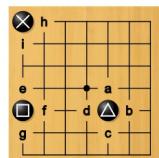
Une fois posées, les pierres ne se déplacent pas, mais elles peuvent être éventuellement capturées et retirées de la grille par l'adversaire.

### Libertés

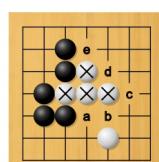
Des intersections sont **voisines** si elles sont reliées par une ligne de la grille et sans autre intersection entre elles. Les **libertés** d'une pierre sont les intersections inoccupées voisines de l'intersection sur laquelle est cette pierre. Des pierres de même couleur sont **connectées** si elles sont sur des intersections voisines. Elles mettent alors leurs libertés en commun.



'a' et 'b' sont voisines, mais 'b' et 'c' ne le sont pas.



La pierre marquée d'un triangle à quatre libertés ('a', 'b', 'c' et 'd'), celle marquée d'un carré trois ('e', 'f' et 'g') et celle marquée d'une croix deux ('h' et 'i').

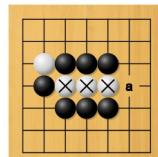


Les quatre pierres blanches marquées d'une croix sont connectées et ont cinq libertés ('a', 'b', 'c', 'd', et 'e').

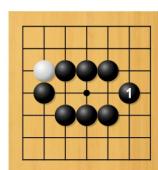
### Capture

Si un joueur supprime la dernière liberté de pierres adverses, il les capture en les retirant de la grille. De plus, il ne doit pas supprimer la dernière liberté de ses propres pierres, sauf s'il capture au moins une pierre adverse.

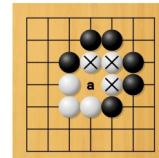
La capture permet de déterminer qui occupe ou entoure quoi.



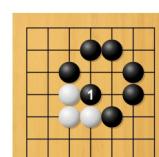
Si Noir joue en 'a', il supprime la dernière liberté des pierres blanches marquées d'une croix...



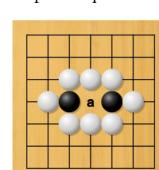
...alors Noir capture ces pierres en les retirant de la grille, ce qui donne deux libertés à la pierre 1 qu'il vient de jouer.



Si Noir joue en 'a', il n'a pas de liberté, mais il supprime la dernière liberté des pierres blanches marquées d'une croix...



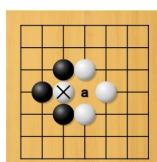
...alors Noir capture ces pierres en les retirant de la grille, ce qui donne deux libertés à la pierre 1 qu'il vient de jouer.



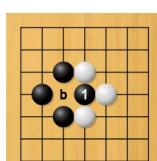
Si Noir joue en 'a', il supprime la dernière liberté de ses pierres et ne capture rien. C'est interdit.

### Répétition

Un joueur ne doit pas, en posant une pierre, ramener la grille dans un état identique à l'un de ceux qu'il lui a lui-même déjà donné.



Si Noir joue en 'a', il capture la pierre marquée d'une croix (il lui enlève sa dernière liberté).



Une fois que Noir a joué 1, Blanc ne peut pas rejouer en 'b' immédiatement et capturer la pierre noire car dans ce cas, il reproduirait la situation du diagramme précédent.

Blanc doit donc jouer ailleurs. Si Noir répond en jouant lui aussi ailleurs, Blanc pourra à nouveau jouer en 'b', puisque l'état de la grille aura changé. Alors ce sera au tour de Noir de devoir jouer

ailleurs, et ainsi de suite, tant qu'aucun des deux joueurs ne connecte ses pierres, empêchant ainsi une nouvelle capture.

### Fin de la partie

Des intersections inoccupées sont **entourées** par un joueur si toutes leurs voisines sont occupées par ses pierres.

La partie s'arrête lorsque les deux joueurs passent consécutivement. Ils peuvent alors retirer de la grille toutes les pierres adverses qu'ils sont certains de pouvoir capturer. Puis ils comptent leurs points qui sont le nombre d'intersections qu'ils sont certains de pouvoir occuper ou entourer avec leurs pierres.

En cas de désaccord des joueurs sur le statut de certaines intersections, la partie reprend dans l'état où elle s'était arrêté jusqu'à deux nouveaux passes consécutifs, sachant qu'ensuite, aucune des pierres encore sur la grille ne pourra être retirée, et que les points des joueurs seront le nombre d'intersections qu'ils occupent ou entourent avec leurs pierres.

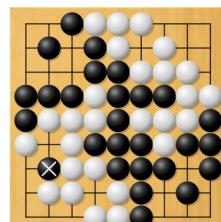
Enfin, Blanc reçoit des **points de compensation** (le plus souvent entre 5,5 et 7,5 points), Noir ayant eu l'avantage de commencer.

**Le gagnant est celui qui a le plus de points.**

Si des intersections ne sont entourées par aucun des joueurs, elles sont **neutres**, et

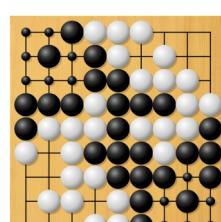
ne donnent de points à personne.

### Exemple de décompte

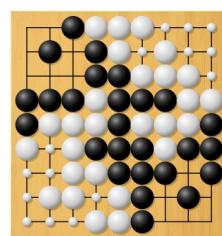


Les joueurs viennent de passer. Blanc retire la pierre marquée d'une croix qu'il est certain de pouvoir capturer.

Comme il ne reste plus de pierres capturables sur la grille, on peut procéder au décompte des points.



Noir entoure 7 intersections en haut à gauche, 6 en bas à droite, et à 26 pierres sur la grille, soit 39 points.



Blanc entoure 8 intersections en bas à gauche, 7 en haut à droite, et à 27 pierres sur la grille, soit 42 points.

Blanc a donc 3 points de plus que Noir. Si on lui ajoute les points de compensation (par exemple 7,5 points), il gagne de 10,5 points.

Pour compter plus rapidement, on peut donner durant la partie une pierre à l'adversaire à chaque fois que l'on passe, obliger Blanc à passer le dernier, et placer à la fin de la partie les pierres que l'on a capturées dans le territoire adverse.

On sera alors certain qu'il y aura autant de pierres noires que de pierres blanches sur la grille, et on pourra donc se contenter de ne compter que les intersections inoccupées restantes.

### Handicap

Parfois on donnera un handicap à l'un des joueurs, consistant à le laisser jouer plusieurs coups de suite en début de partie, ou à modifier le nombre de points de compensation.

FIGURE 13 – Résumé des règles du Go [14]

## Features Spatiales (binaryInputNCHWPacked)

Les vingt-deux plans binaires listés dans le tableau suivant décrivent l'état instantané du goban, l'historique tactile immédiat ainsi que plusieurs contraintes de règles (ko, phases d'*encore*). Les quatre premiers canaux suffisent à reconstruire la position pure ; les suivants ajoutent la notion de libertés, d'échelle ou de territoire pressenti, de sorte que le réseau dispose dès l'entrée d'indices tactiques et stratégiques sans avoir à les redécouvrir.

Canal	Description	Valeurs	Détails
0	Position sur le plateau	0–1	Masque des intersections valides sur le goban
1	Pierres du joueur actuel	0–1	Emplacements occupés par les pierres du joueur actuel
2	Pierres de l'adversaire	0–1	Emplacements occupés par les pierres de l'adversaire
3	Pierres en atari (1 liberté)	0–1	Pierres ou groupes à une seule liberté restante
4	Pierres à 2 libertés	0–1	Groupes de pierres disposant exactement de deux libertés
5	Pierres à 3 libertés	0–1	Groupes de pierres disposant exactement de trois libertés
6	Positions interdites par Ko	0–1	Cases bloquées par la règle du ko simple
7	Interdictions Ko de l'encore	0–1	Cases interdites en phase d' <i>« encore »</i> (ko étendu)
8	Canal réservé	0–1	Usage interne : suivi de l' <i>encore</i>
9	Dernier coup (adversaire)	0–1	Position du dernier coup joué par l'adversaire
10	Avant-dernier coup (joueur)	0–1	Position de l'avant-dernier coup du joueur au trait
11	3 dernier coup (adversaire)	0–1	Position du troisième dernier coup adverse
12	4 dernier coup (joueur)	0–1	Position du quatrième dernier coup joueur
13	5 dernier coup (adversaire)	0–1	Position du cinquième dernier coup adverse
14	Échelles actuelles	0–1	Intersections faisant partie d'une échelle active (ladder)
15	Échelles (coup précédent)	0–1	Échelle amorcée lors du coup précédent
16	Échelles (2 coups avant)	0–1	Échelle amorcée deux coups plus tôt
17	Coups d'échappement d'échelle	0–1	Coups possibles pour briser ou échapper à l'échelle

18	Territoire du joueur	0–1	Cases considérées comme territoire sécurisé du joueur
19	Territoire de l'adversaire	0–1	Cases considérées comme territoire sécurisé de l'adversaire
20	Pierres d'encore du joueur	0–1	Pierres en vigueur au début de la deuxième phase d'« encore » du joueur
21	Pierres d'encore de l'adversaire	0–1	Pierres en vigueur au début de la deuxième phase d'« encore » adverse

*Ces cartes binaires offrent donc au modèle une “photographie” détaillée : pierres, libertés, historique des six derniers coups, menaces d'échelle et règles de ko.*

Canaux associé au modèle b28c512nbt-s8925522176-d4787295149 trouvés sur : model-version.cpp (chercher modèle 7 : <https://github.com/lightvector/KataGo/blob/master/cpp/neuralnet/neuralnet.cpp>)

## Cibles de politique (policyTargetsNCMove).

Pour entraîner la tête de politique, nous retenons deux plans. Le premier contient le nombre brut de visites MCTS pour chaque intersection ainsi que pour l'action **pass**; le second décale cette information d'un coup pour servir de cible auxiliaire. Ces comptages, encore non normalisés, reflètent la confiance différenciée du MCTS : plus un coup est exploré, plus il est jugé prometteur.

Canal	Description
C0	Nombre de visites MCTS pour chaque coup (81 intersections + passe).
C1	Nombre de visites pour le coup suivant (cible auxiliaire).

*La tête de politique apprend ainsi à approcher la distribution de visites MCTS plutôt qu'une simple étiquette « meilleur coup ».*

## Cibles de valeur (valueTargetsNCHW).

La tête de valeur reçoit une supervision plus fine qu'un simple « victoire / défaite » : le canal C0 assigne à chaque intersection sa propriété finale (+1/–1/0), tandis que C1 corrige les cas de *seki*. Deux autres plans (C2–C3) demandent au réseau de prédire l'état des pierres à court terme. Enfin le canal C4 encode le territoire final en tenant compte des règles de scoring, et huit masques binaires indiquent quels plans sont valides. Cette supervision dense oblige le modèle à comprendre la topologie complète du goban plutôt que de se contenter d'un signal global.

Canal	Description
C0	Propriété finale de chaque case ( $-1 = \text{adverse}$ , $0 = \text{neutre}$ , $+1 = \text{joueur actuel}$ ).
C1	Écart entre propriété et aire naïve (gestion des seki).
C2–C3	Prédiction des positions des pierres à horizon fixe (futur).
C4	Aire finale/territoire avec règles de scoring (taxe de groupe).
C27–C34	Poids binaires indiquant la validité de chaque canal (0 = ignoré, 1 = valide).

*Ces targets spatiales permettent un apprentissage fin de la propriété du plateau, offrant un signal beaucoup plus riche qu'un simple résultat binaire victoire/défaite.*

## Features globales.

Certaines informations — komi, handicap, phases d'*encore*, variante de règle de ko, etc. — ne peuvent pas être exprimées dans une carte spatiale. Nous les rangeons donc dans un vecteur de dix-neuf scalaires, concaténé à la sortie de la partie convolutionnelle avant la tête de valeur et la tête de politique. Le réseau sait ainsi quelle variante de règle s'applique et dans quelle phase la partie se trouve, ce qui serait impossible à déduire uniquement avec les pierres.

Index	Description	Valeurs	Détails
0–4	Passes récentes	0–1	Indique si les 5 derniers coups étaient des passes
5	Komi	float	Komi normalisé ( $\div 20$ )
6–7	Règles de Ko	0–1	Simple / positionnel / situationnel
8	Suicide multiple	0–1	1 si suicide multiple légal
9–11	Règles de scoring	0–1	Territory vs. area et taxation
12–13	Phase d' <i>encore</i>	0–1	Indique les phases d' <i>encore</i>
14	Fin de phase par passe	0–1	1 si une passe termine la phase
15–16	Handicap	0–1, float	Paramètres de handicap
17	Bouton	0–1	Règle du bouton (first-pass-collect)
18	Parité komi/plateau	float	Ajustement de parité komi/aire

*Grâce à cette séparation claire entre informations spatiales et globales, le modèle dispose à la fois du contexte de règles et de la géométrie détaillée du plateau, condition indispensable pour rivaliser avec les moteurs professionnels récents.*

## Correspondance modèle–features

Pour déterminer précisément l'encodage des entrées du réseau, nous avons suivi trois étapes :

1. Dans KataGo/python/katago/train/modelconfigs.py, le modèle nommé b28c512nbt est déclaré à la version 15 :

```
b28c512nbt = {  
    "version": 15,  
}
```

2. Nous avons ensuite ouvert KataGo/cpp/neuralnet/modelversion.cpp pour vérifier à quelle génération de features cette version 15 correspond. La ligne 22 y indique :

```
// 15 = V7 features, Extra nonlinearity for pass output
```

3. Enfin, dans KataGo/cpp/neuralnet/nninputs.cpp, la section INPUTSVERSION 7 définit, via la fonction NNInputs::fillRowV7(...), la liste exacte des plans binaires et scalaires utilisés comme entrées du réseau.

La cohérence de ces canaux a été contrôlée grâce aux tests d'exécution figurant dans KataGo/cpp/tests/results/runOutputTests.txt. Par ailleurs, le fichier <https://katagoarchive.org/g170/selfplay/README.txt> nous a éclairés sur la structure des cibles de politique et de valeur produites par KataGo.