



ÉCOLE  
**CENTRALE** LYON

Projet d'études

PE 110 : ShogIA

Rapport final

---

## Développement d'une application pour le jeu de shogi

---

### *Auteurs :*

M. Dylan DUMAY  
M. Ibrahim EZ-ZAHRAOUI  
M. Léo GONÇALVES  
M. Arthur LEITE  
M<sup>me</sup> Jinglei LIU  
M. Antony PARODI

### *Encadrants :*

M. Baptiste CELLE  
M. Philippe MICHEL  
M. Hai-Son NGUYEN  
M. Alexandre SAIDI  
M. Abdel-Malek ZINE

5 juin 2020

## Résumé

Le Shogi, également connu sous le nom d'échecs japonais, est une version japonaise du Chaturanga. Celui-ci est un jeu de société indien qui est théoriquement à l'origine des échecs européens et de ses semblables comme le Shogi et le Makruk [14].

Connu comme la variante la plus complexe des échecs, le jeu a attiré l'attention des développeurs d'intelligence artificielle depuis les années 70. Cependant, en raison de sa très grande complexité et sa faible popularité, il a fallu 35 ans pour que la première IA développée soit capable de battre un joueur professionnel en 2010 lors d'un match d'une durée de 6 heures.

L'entreprise DeepMind, grâce à son AlphaZero, dispose de l'IA la plus performante du monde pour le jeu de Shogi. L'utilisation de sa méthode d'auto-apprentissage a de loin dépassé les méthodes antérieures telles que Alpha-bêta ou encore Monte Carlo. Toutefois, cela n'enlève rien à l'efficacité de ces méthodes et encore moins à leur importance aujourd'hui. C'est en y réfléchissant que le groupe d'étudiants de l'École Centrale de Lyon du PE 110 a été formé dans le but de développer une IA et d'élaborer un tutoriel pour initier de nouveaux joueurs à cette formidable variante des échecs.

## Abstract

Shogi, also known as Japanese chess, is a Japanese version of Chaturanga, an Indian board game which is theoretically the origin of European chess and its counterparts like Shogi and Makruk [14].

Known as the most complex variant of chess, the game has attracted the attention of artificial intelligence developers since the 1970s. However, due to its high complexity and its lower popularity, it took 35 years for the first AI developed to be able to beat a professional player in a 6-hour match in 2010.

DeepMind, thanks to AlphaZero, has the world's most powerful AI for Computer Shogi. The use of its self-learning algorithm has meant that previous methods such as Alpha-beta itself or Monte Carlo have proven somewhat outdated. However, this does not detract from the effectiveness of these methods and even less from their importance today. It is by thinking about it that the group of students of the Ecole Centrale de Lyon of PE 110 was formed, with the aim to develop an AI and also to bring a tutorial to initiate new players to this formidable chess variant.

# Table des matières

<b>1 Présentation de l'équipe</b>	<b>5</b>
<b>2 Introduction</b>	<b>6</b>
2.1 Contexte . . . . .	6
2.2 Présentation du jeu . . . . .	7
2.3 État de l'art . . . . .	7
<b>3 Présentation de l'application</b>	<b>9</b>
3.1 Cahier des charges . . . . .	9
3.2 Produit final . . . . .	10
3.2.1 Menu principal . . . . .	10
3.2.2 Options . . . . .	10
3.2.3 Tutoriel . . . . .	10
3.2.4 Jeu . . . . .	11
<b>4 Réalisation et solutions techniques</b>	<b>14</b>
4.1 Interface graphique . . . . .	14
4.1.1 Menu . . . . .	14
4.1.2 Tutoriel . . . . .	14
4.1.3 Jeu . . . . .	14
4.2 Moteur de jeu . . . . .	15
4.2.1 Structure du moteur . . . . .	15
4.2.2 Implémentation des règles du jeu . . . . .	16
4.3 Intelligence artificielle avec la méthode alpha-bêta . . . . .	17
4.3.1 Livre d'ouverture . . . . .	20
4.3.2 Parcours d'arbre . . . . .	21
4.3.3 Fonction d'évaluation . . . . .	21
4.3.4 Résultats . . . . .	24
4.4 Intelligence artificielle avec la méthode de Monte Carlo (MCTS) et les réseaux de neurones . . . . .	24
4.4.1 Recherche arborescente Monte Carlo . . . . .	25

4.4.2	Alpha Zero Junior . . . . .	27
4.4.3	Simple réseau de neurones (SNN) . . . . .	32
<b>5</b>	<b>Conclusion</b>	<b>35</b>
<b>A</b>	<b>Règles du Jeu de Shogi</b>	<b>36</b>
A.1	Objectif . . . . .	36
A.2	Joueurs . . . . .	36
A.3	Plateau . . . . .	36
A.4	Pièces . . . . .	36
A.5	Déplacements . . . . .	37
A.6	Prise . . . . .	38
A.7	Promotion . . . . .	38
A.8	Parachutage . . . . .	38
A.9	Nulles . . . . .	39
<b>B</b>	<b>Détails du moteur de jeu</b>	<b>39</b>
<b>C</b>	<b>Architecture du réseau de neurones</b>	<b>42</b>
<b>D</b>	<b>Budget</b>	<b>44</b>
<b>E</b>	<b>Tableau de vérification</b>	<b>45</b>

## Table des figures

1	Comparaison entre les plateaux de shogi et d'échecs . . . . .	6
2	Menu principal de l'application . . . . .	10
3	Menu Tutoriel . . . . .	11
4	Interface de jeu au lancement d'une partie . . . . .	12
5	Interface de jeu au milieu d'une partie . . . . .	12
6	Déplacements possibles de la tour . . . . .	13
7	Fin de partie . . . . .	13
8	Exemple d'arbre de l'algorithme Minimax . . . . .	18
9	Example d'arbre de l'algorithme Minimax avec élagage . . . . .	19
10	Exemple de calcul de squareControl . . . . .	22
11	Schéma de la zone de danger . . . . .	23
12	Table de Shogi . . . . .	37
13	Mouvements des pièces . . . . .	37

## 1 Présentation de l'équipe

L'équipe de ce projet est composée de six membres :

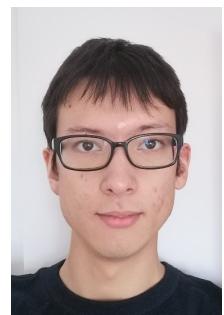
- M. Dylan DUMAY, chef de projet
- M. Antony PARODI, chargé de communication
- M. Léo GONÇALVES
- M. Ibrahim EZ-ZAHRAOUI
- M. Arthur LEITE
- Mme Jinglei LIU



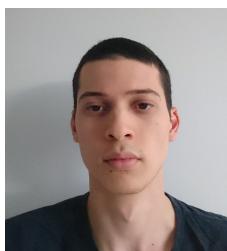
Dylan DUMAY



Ibrahim EZ-ZAHRAOUI



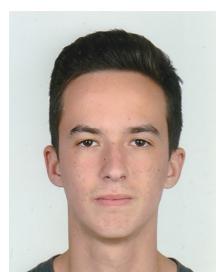
Léo GONÇALVES



Arthur LEITE



Jinglei LIU

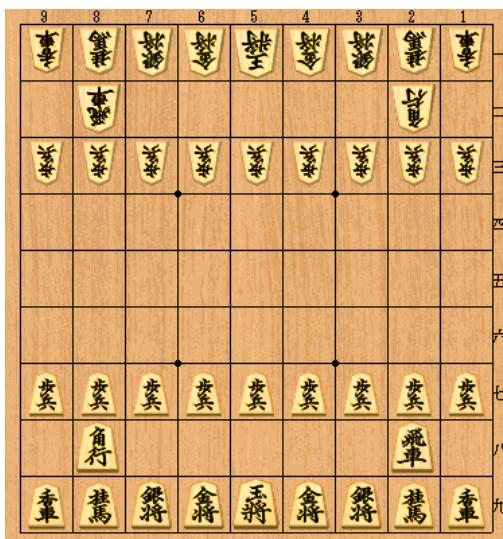


Antony PARODI

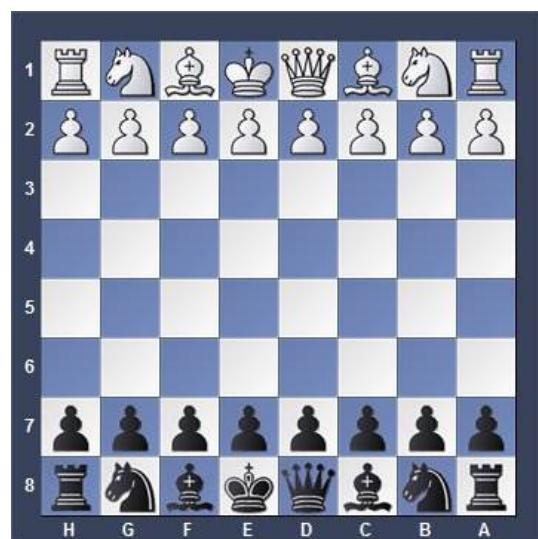
## 2 Introduction

### 2.1 Contexte

Le jeu de shogi est un jeu similaire aux échecs très populaire en Asie mais comptant des joueurs à travers le monde. Ce jeu comportant des pièces et des règles supplémentaires est réputé plus difficile et complexe que le jeu d'échec. C'est pourquoi il a intéressé de nombreux chercheurs en intelligence artificielle.



Plateau du jeu de shogi



Plateau du jeu d'échec

**FIGURE 1 –** Comparaison entre les plateaux de shogi et d'échecs

De part sa complexité supérieure, il fut, après le jeu d'échec, une cible privilégiée des concepteurs d'IA à travers le monde. En effet, les premiers succès des programmes informatiques contre des professionnels sont survenus dans les années 2010, c'est-à-dire 20 ans après la victoire de Deep Blue aux échecs.

Aujourd'hui, le programme AlphaZero est devenu le programme le plus performant en 2017. C'est un programme développé par DeepMind, une entreprise appartenant aux groupes de Google qui a appris par lui-même la plupart des stratégies existantes jusqu'alors.

Malgré cela, la conception d'intelligence artificielle reste toujours d'actualité que ce soit sur le shogi ou sur d'autres jeux encore plus complexes (jeux de stratégie en temps réel). C'est à partir de ce constat que s'est formé un projet mené par l'équipe d'étudiants de l'École Centrale de Lyon (ECL) du PE 110, de septembre 2019 à juin 2020. Notre projet a avant tout un but formateur, c'est à-dire, qu'en premier lieu, l'objectif est d'initier les membres du groupe à la réalisation d'une IA la plus performante possible qui pourra jouer dans des délais les plus courts possibles. La mesure de cette performance pourra

être réalisée à l'aide du système de ranking du shogi (mesure en ELO) ou en affrontant d'autres intelligences artificielles déjà existantes.

Il pourrait également être intéressant d'intégrer dans le programme un système de tutoriel permettant d'initier le joueur au shogi.

## 2.2 *Présentation du jeu*

Le jeu de shogi (littéralement « jeu des généraux ») est un jeu de plateau combinatoire partageant de très grandes similarités avec le jeu d'échec. Son origine demeure floue mais il prend ses racines en Asie et est devenu un jeu traditionnel japonais. Ce jeu est très populaire en Asie mais reste méconnu dans les autres régions du Monde. Par exemple, la fédération française de shogi n'a été fondée qu'en 2006. La principale différence avec le jeu d'échec est l'existence de nouvelles pièces et de deux nouvelles mécaniques, appelées "parachutage" et "promotion".

En effet, on peut constater l'apparition de pièces comme le Lancier, le Général d'argent et le Général d'or.

Chaque pièce atteignant le camp adverse peut être promue et acquérir de nouveaux mouvements.

Enfin, la mécanique de "parachutage" consiste en la possibilité de rejouer les pièces capturées de son adversaire directement sur le plateau en suivant certaines règles spécifiques.

## 2.3 *État de l'art*

En repensant à l'histoire, de 1997 à nos jours, l'intelligence artificielle a créé à plusieurs reprises de splendides records contre les meilleurs joueurs d'échecs humains dans des jeux tels que les échecs, le Go ou autres jeux combinatoires. Au niveau du Shogi, le programme Ponanza de Yamamoto a battu le meilleur joueur de Shogi japonais Sato Shinichi en 2013, et n'a connu aucune défaite pendant quatre années consécutives. Ponanza, DeepBlue d'IBM et AlphaGo de Google, sont appelés les trois points de repère dans l'histoire de l'intelligence artificielle.

La première Ponanza a été conçue sur la base du principe de "recherche + évaluation". Yamamoto estime que presque tous les comportements intelligents répètent le processus de recherche et d'évaluation, et essaient longuement de choisir celui qui semble le plus avantageux, puis passent à l'étape suivante, l'accumulation, la répétition et la superposition. Cependant, le logiciel était encore faible à l'époque. Au Championnat du Monde de Shogi informatique 2010, il n'a remporté que la 11e place.

Depuis lors, Yamamoto a également introduit la technique d'apprentissage

automatique et la technique d'apprentissage en profondeur, ce qui a amélioré l'efficacité de calcul du logiciel et réduit le taux d'erreur. En 2016 et 2017, dans la compétition d'Esport de Shogi et le Championnat du Monde de Shogi informatique, Ponanza a remporté la première place. Malheureusement, après la fin du cinquième Championnat Shogi Den-O en novembre 2017, Yamamoto a annoncé la retraite de Ponanza. Les mouvements et tactiques de Ponanza sont appelés "Styles de Ponanza", et ils sont maintenant acceptés par les joueurs de Shogi professionnels. Bien que les programmes qui jouent au Shogi aient fait de grandes réalisations, il reste encore des améliorations à apporter.

### 3 Présentation de l'application

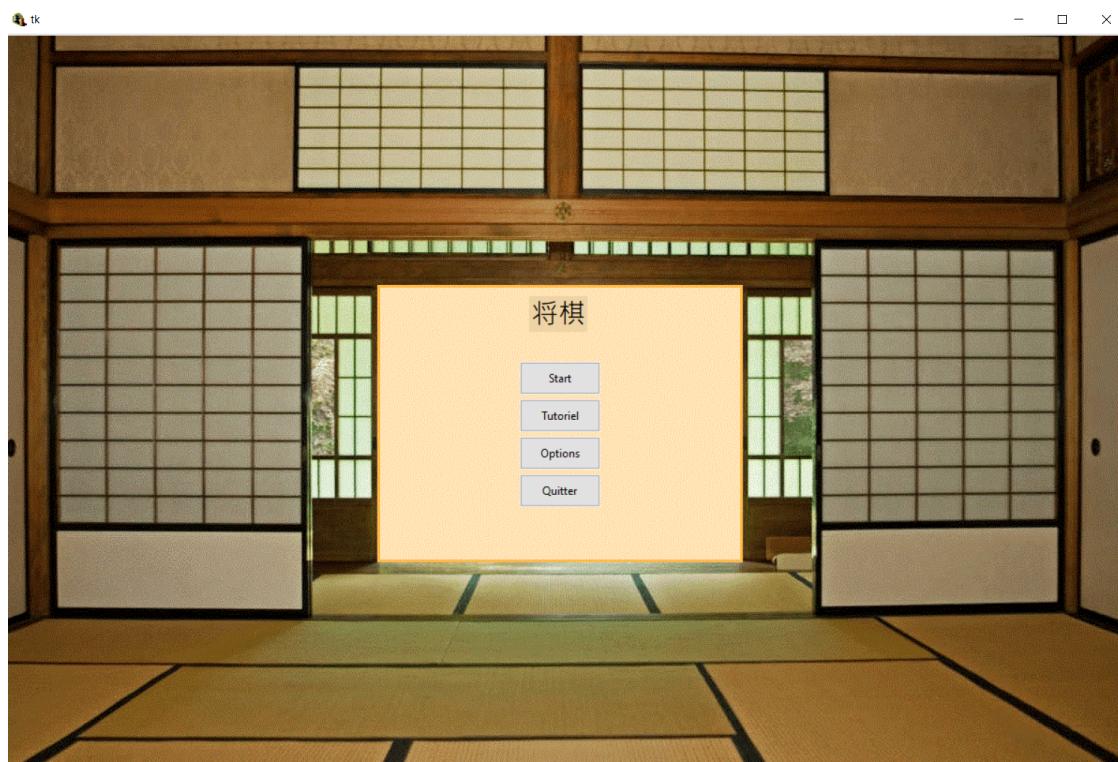
#### 3.1 Cahier des charges

Fonctions principales	Sous-fonctions
Permettre des parties entre joueurs	Mettre à disposition un moteur de jeu stockant et mettant à jour toutes les informations décrivant l'état d'une partie. Le respect des règles du jeu doit être assuré.
	Afficher un plateau de jeu ainsi que les coups possibles d'une pièce sélectionnée par un joueur.
Permettre des parties joueur contre IA.	Pouvoir jouer contre une IA par l'intermédiaire de l'interface graphique.
	L'IA doit pouvoir battre un joueur débutant.
Permettre des parties avec une IA	Afficher l'évolution de la recherche de coup par les IA
	Lancer un très grand nombre de parties avec une IA donnée contre différentes IA afin d'évaluer l'Elo de l'IA.
Apprendre à jouer au shogi	Mettre à disposition un tutoriel initiant aux règles du shogi.
	Pouvoir régler facilement les paramètres de l'IA afin que son niveau soit adapté à celui du joueur.

## 3.2 *Produit final*

### 3.2.1 Menu principal

Le menu du jeu (visible figure suivante) est l'interface principale avec l'utilisateur. Celui-ci dispose de plusieurs boutons permettant de lancer une nouvelle partie de shogi, ou d'ouvrir un menu d'options. De plus, le shogi étant un jeu méconnu, une partie tutoriel a été prévue afin de comprendre les règles et les principes de base du jeu.



**FIGURE 2 –** Menu principal de l'application

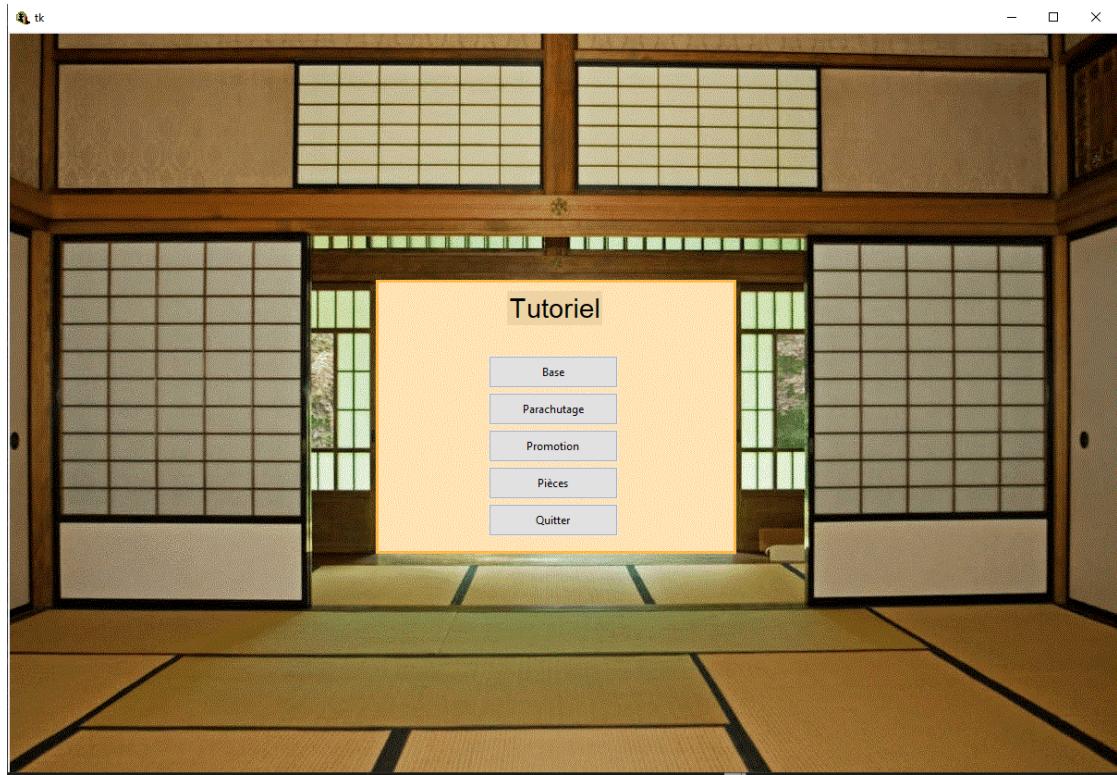
### 3.2.2 Options

La partie Options permet à l'utilisateur de choisir s'il veut jouer contre un autre utilisateur, ou s'il veut jouer contre une IA. Si l'utilisateur veut jouer contre une IA, il peut aussi sélectionner son niveau : 1, 2 ou 3. Plus ce nombre est élevé, plus l'IA sera forte, mais plus elle sera lente.

### 3.2.3 Tutoriel

Le tutoriel prévu pour aider les débutants à comprendre le jeu est décomposé en 4 parties (Voir figure suivante) et est sous forme de texte agrémenté d'images

animées afin de bien s'approprier les règles.



**FIGURE 3 – Menu Tutoriel**

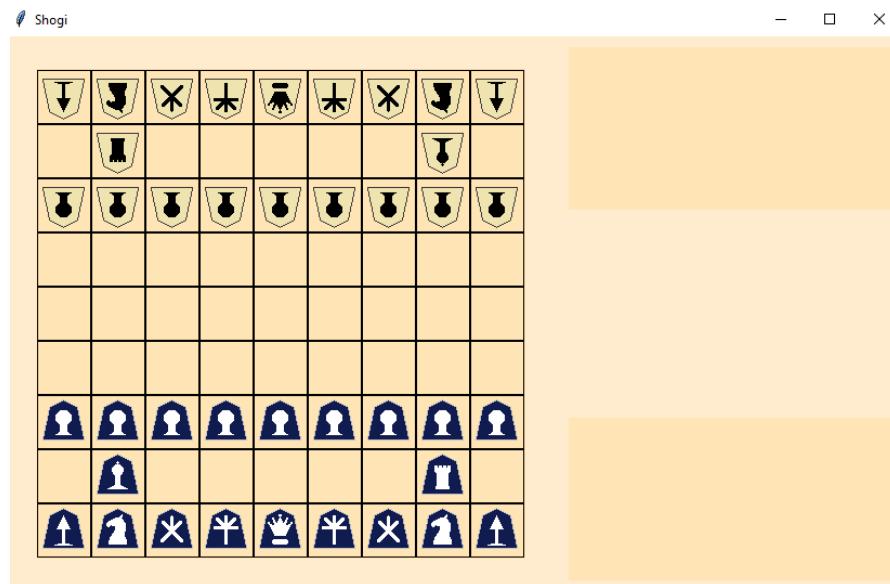
La première partie présente les bases du jeu tel que le plateau, les différentes pièces, le but du jeu ou encore la prise de pièces adversaires (mécanique similaire au jeu d'échec).

La deuxième présente la mécanique inédite de parachutage propre au shogi qui permet sous certaines conditions de placer une pièce capturée de l'adversaire sur notre plateau.

Puis vient la partie sur la promotion de pièces permettant d'améliorer les pièces existantes d'un joueur et enfin la partie montrant sous forme d'images animées les mouvements possibles pour chacune des pièces du shogi.

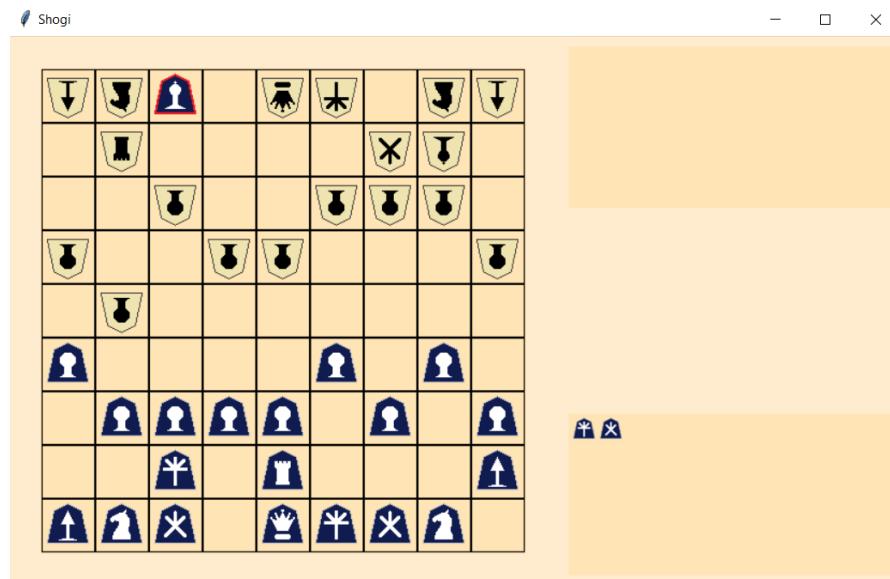
### 3.2.4 Jeu

C'est la partie principale de l'application qui permet de jouer entièrement une partie de shogi en prenant en compte les règles.



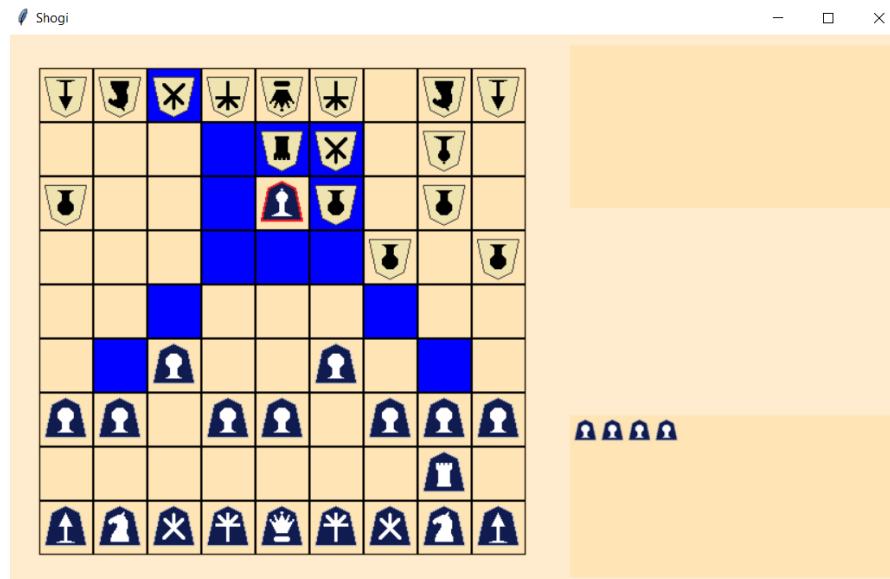
**FIGURE 4 – Interface de jeu au lancement d'une partie**

On peut donc voir sur la figure précédente le plateau de jeu au lancement d'une partie avec toutes les pièces encore présentes et à leur position initiale. Les parties jaune à droite du plateau représentent les zones où les pièces qui pourront être parachutées seront affichées.



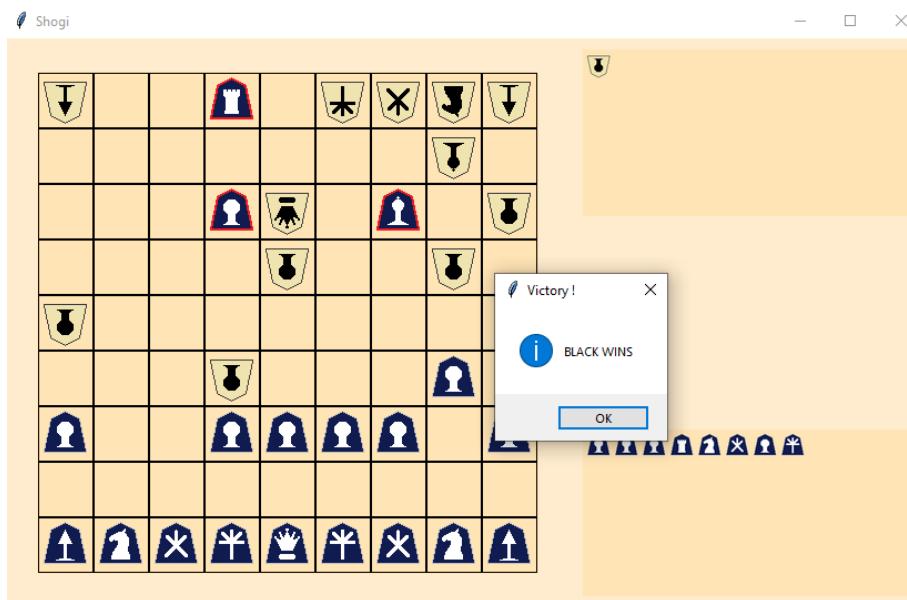
**FIGURE 5 – Interface de jeu au milieu d'une partie**

On voit sur cette figure une partie dans un état un peu plus avancé. On remarque les pièces promues (entouré d'un contour rouge) et les pièces parachutables sur la droite de l'application.



**FIGURE 6 – Déplacements possibles de la tour**

De plus, tout au long de la partie, l’application peut afficher les déplacements possibles d’une pièce en cliquant dessus. La promotion de pièce est aussi gérée et l’application demande au joueur si oui ou non il veut promouvoir sa pièce si il remplit les conditions nécessaires.



**FIGURE 7 – Fin de partie**

Enfin, lors de la fin de la partie, le jeu la détecte et affiche le gagnant dans une nouvelle fenêtre.

## 4 Réalisation et solutions techniques

L'application a été entièrement codée en Python. Coder dans un autre langage tel que C++ par exemple aurait pu permettre d'obtenir des performances accrues et un style graphique plus affiné, cependant en considérant l'ampleur du projet, le groupe a préféré opter pour Python qui est un langage connu de la majorité du groupe et qui est globalement plus simple.

### 4.1 Interface graphique

L'interface graphique a été réalisée avec le module `tkinter` de Python et à l'aide de la programmation orientée objet.

#### 4.1.1 Menu

Le menu du jeu est composé de la fenêtre principale et de plusieurs boutons permettant l'accès aux différentes fonctions de l'application.

Il utilise des fonctions de base du module `tkinter`. On crée une classe héritant de la classe `Tk` qui correspond à la fenêtre principale dans laquelle on va pouvoir afficher les différents objets que l'on souhaite. Les objets tels que le titre ou les différents boutons sont créés à l'aide des classes déjà existante du module telles que `Button` ou `Label`.

#### 4.1.2 Tutoriel

De la même manière, le tutoriel est composé de boutons permettant d'afficher les différents textes expliquant les différentes parties du jeu. Celui-ci repose sur les mêmes principes que le menu du jeu. En outre, on utilise la classe `Text` et la classe `Scrollbar` de `tkinter` permettant d'afficher des longs textes et de pouvoir naviguer à travers ceux-ci avec une barre de déroulement. Les textes sont eux-mêmes stockés dans des fichier à part et sont lus par l'application. Le seul point délicat du tutoriel est l'affichage d'images animées ou 'gif' car mal pris en charge par le module. On doit donc afficher manuellement les différentes images composant cette image animée par une boucle et un indice incrémenté.

#### 4.1.3 Jeu

La partie qui permet à l'utilisateur de jouer se base entièrement sur la classe `GameState` du moteur de jeu. En communiquant avec elle, elle affiche de manière dynamique l'état du jeu.

Elle peut aussi créer 2 sous fenêtres, une au début de la partie pour choisir son côté quand on joue contre une IA, et une autre dans le cas d'une promotion où on peut choisir si oui ou non on veut promouvoir une pièce.

## 4.2 *Moteur de jeu*

Le moteur de jeu est un élément essentiel du projet et doit être optimisé au mieux. En effet, celui-ci sert de base pour toutes les autres parties du projet (interface graphique, intelligences artificielles).

### 4.2.1 Structure du moteur

Le moteur de jeu est organisé par classes (programmation orientée objet).

La classe `Piece` représente une pièce tout type confondu. Elle a pour attributs sa position, sa couleur, si elle est prisonnière, et si elle est promue. Elle possède une méthode indiquant si elle peut être promue et une autre méthode indiquant si elle a l'obligation d'être promue.

Chaque type de pièce possède sa propre classe (pion, lance, cavalier, général d'argent, général d'or, tour, fou, roi) et chacune de ces classes hérite de la classe `Piece`. Ces classes n'existent que pour renseigner sur les mouvements possible de la pièce qu'elles représentent.

La classe `GameState` représente une position du jeu, c'est-à-dire un état possible du jeu. C'est le cœur du moteur de jeu car en plus de l'état de la partie, elle contient aussi toutes les règles du jeu en son sein.

Ses principaux attributs sont les suivants :

- Le plateau de jeu. Il s'agit d'un tableau avec les pièces du jeu sur le plateau. Chaque pièce est une instance de la classe de son type de pièce (les classes décrites précédemment).
- La liste de toutes les pièces du jeu, capturées ou non.
- La couleur qui doit jouer à partir de la position actuelle.
- Le nombre de coups joués jusque-là.
- La liste `legal_moves` des coups légaux à partir de la position actuelle. C'est la génération de cette liste qui constitue le cœur du moteur de jeu avec toutes les règles. Cette liste est constituée de tous les coups qu'il est possible de jouer à partir la position.

La méthode principale de `GameState` est `update` qui prend un coup légal en argument et met à jour les attributs, notamment la liste `legal_moves`.

La partie suivante expose les idées importantes concernant l'implémentation des règles du jeu dans le moteur.

### 4.2.2 Implémentation des règles du jeu

Le fonctionnement détaillé du moteur de jeu est donné en annexe.

#### Fonctionnement global

La position de départ d'une partie de shogi est représentée par l'état initial d'une instance `s` de `GameState`. Au début, noir joue un coup présent dans la liste `legal_moves`. Puis `s` est mise à jour avec `update` en donnant le coup joué en argument. Blanc joue ensuite un coup présent dans la nouvelle liste `legal_moves`, et ainsi de suite jusqu'à la fin de la partie. Une partie est terminée lorsque la liste `legal_moves` est vide, signifiant que le joueur qui devait jouer n'a plus aucun coup légal à jouer car il est en échec et mat. Le joueur précédent ayant joué le dernier coup est alors le vainqueur.

Les coups du shogi sont représentés avec la notation décrite dans le protocole USI (Universal Shogi Interface) [16]. Il s'agit d'un protocole pour la communication entre un moteur et une interface graphique de shogi. Chaque coup est ainsi une chaîne de caractères.

#### Performances des différentes versions

Au cours du projet, trois versions du moteur de jeu ont vu le jour. C'est la fonction `update_legal_moves` mettant à jour `legal_moves` qui a été améliorée à chaque fois, dans l'optique d'une plus grande rapidité de calcul. La performance de chaque version a été évaluée en utilisant une IA aléatoire, c'est-à-dire une IA sélectionnant un coup de manière aléatoire dans `legal_moves`. Nous avons calculé le nombre de coups joués par l'IA pour chaque version du moteur :

- version 1 : 8 coups/seconde.
- version 2 : 428 coups/seconde (version 1 à 2 : x54).
- version 3 : 3993 coups/seconde (version 2 à 3 : x9).

Afin de tester l'absence d'erreurs dans le moteur, 10 000 parties aléatoires ont été effectuées. Pour vérifier la validité des coups légaux, 100 parties ont été effectuées sur le logiciel *shogidokoro* (l'interface graphique la plus utilisée par les créateurs d'IA de Shogi) avec notre moteur.

#### Règles omises

Il est à noter que le moteur de jeu omet deux règles du jeu :

- L'égalité d'une partie si une même position se reproduit quatre fois.
- L'égalité d'une partie si les deux joueurs ont leur roi dans le camp adverse sans qu'il soit aisé de trouver un mat.

L'implémentation de ces deux règles aurait considérablement alourdi le moteur de jeu en plus du travail que cela aurait représenté. Comme ces règles ne s'appliquent que très rarement et qu'elles ne sont pas indispensables même dans les situations où elles doivent s'appliquer, elles n'ont pas été implémentées. Néanmoins, une version complète du moteur de jeu devrait les prendre en compte.

### 4.3 *Intelligence artificielle avec la méthode alpha-bêta*

L'élagage alpha bêta [12], connu comme *alpha-beta pruning* en anglais, est un algorithme de recherche qui vise à réduire le nombre de noeuds évalués dans un arbre de jeu par la méthode Minimax. Cela fonctionne de telle manière qu'il arrête d'évaluer un mouvement lorsqu'il trouve que ce mouvement est pire qu'un précédent déjà analysé : il n'y a alors pas besoin d'évaluer ses fils.

#### **Algorithme Minimax**

Pour comprendre la méthode alpha-bêta, il faut avant comprendre la méthode Minimax. L'idée est relativement simple, on part de la position (noeud) actuelle du jeu et on considère certains mouvements, puis sont considère les mouvements de l'adversaire et après le snôtre à nouveau, etc.., jusqu'à ce que l'on arrive au niveau de profondeur souhaité ou à un noeud terminal (ici, une fin de partie). À partir de la fin de l'arbre, on évalue le score de cette situation de jeu, et si le noeud parent correspond à un tour de l'adversaire, on prend le minimum de ses fils (parce qu'il est le 'minimizingPlayer'), si c'est l'IA, on prend le maximum (on est le 'maximizingPlayer') et ainsi de suite jusqu'à ce qu'on remonte à la première position.

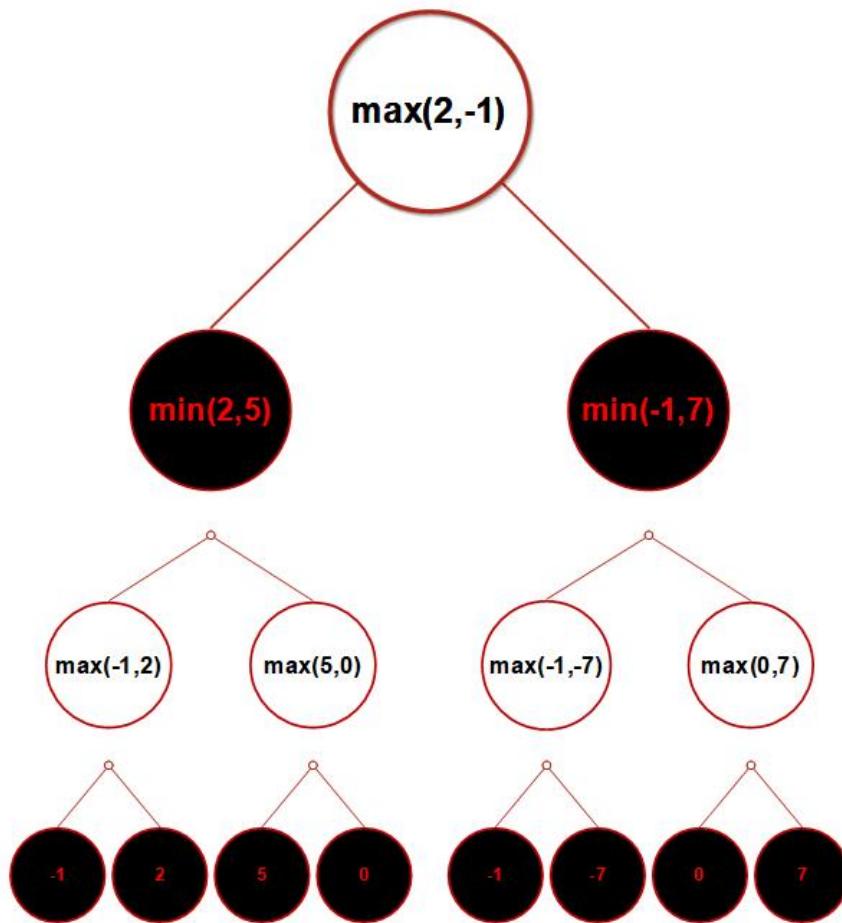
Ce processus peut être illustré par l'arbre logique dans la figure 8 et le pseudo-code qui suit :

```
function minimax(node, depth, isMaxPlayer):
    if depth == 0 or game over in node:
        return value of node
    if isMaxPlayer:
        maxValue = -inf
        for each child in node:
            value = minimax(child, depth - 1, False)
            maxValue = max(maxValue,value)
        return value
    else:
        minValue = inf
```

```

for each child in node:
    value = minimax(child, depth - 1, True)
    minValue = min(minValue,value)
return value

```



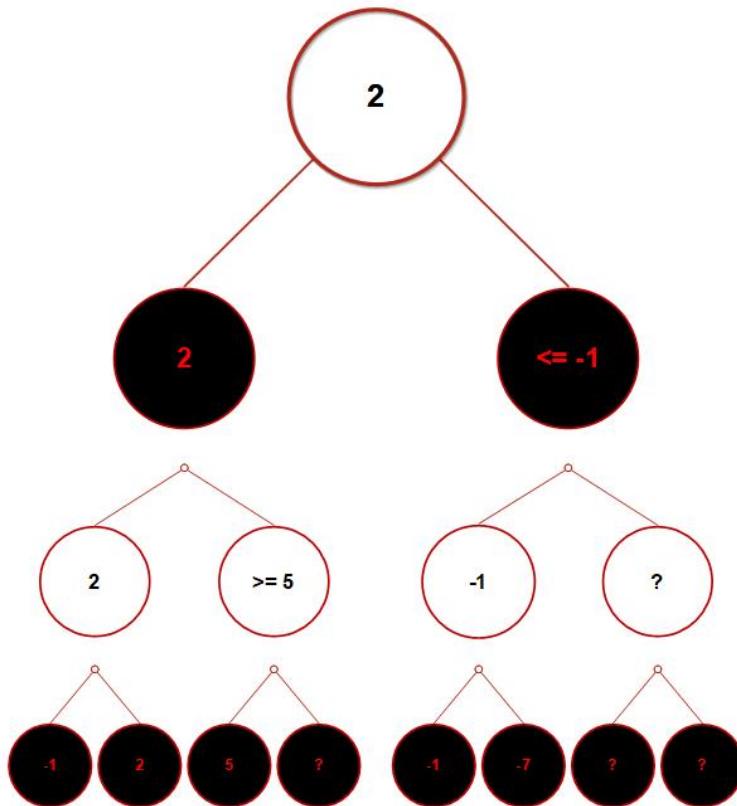
**FIGURE 8 –** Exemple d'arbre de l'algorithme Minimax

On peut voir que dans l'arbre exemple de la figure 8, le mouvement considéré sera le mouvement 2 à gauche. Quand on remonte l'arbre, on effectue les calculs écrits sur chaque noeud pour trouver la valeur finale, celle du haut de l'arbre.

### Implémentation alpha-bêta sur Minimax

Si on regarde le processus qu'on vient de faire avec attention, on peut voir que certaines évaluations n'étaient pas nécessaires, et on aurrait pu les couper sans

changement dans le résultat final. Cela peut être illustré avec une modification de l'arbre de la figure 8.



**FIGURE 9 –** Example d'arbre de l'algorithme Minimax avec élagage

Comme chaque calcul non nécessaire est une augmentation du temps de calcul, il faut alors réaliser un algorithme capable de reconnaître quand une évaluation sera inutile et empêcher le programme principal de la calculer. À partir de cela seront ajoutés deux paramètres supplémentaires à la fonction Minimax, alpha et bêta, qui suivront le meilleur score possible pour chaque joueur (alpha pour nous et bêta pour l'adversaire).

```
function alphabeta(node, depth, alpha, beta, isMaxPlayer):
    if depth == 0 or game over in node:
        return value of node
    if isMaxPlayer:
        maxValue = -inf
        for each child in node:
            value = alphabeta(child, depth - 1, alpha, beta, False)
            maxValue = max(maxValue,value)
        alpha = max(alpha, maxValue)
    else:
        minValue = inf
        for each child in node:
            value = alphabeta(child, depth - 1, alpha, beta, True)
            minValue = min(minValue,value)
            beta = min(beta, minValue)
        beta = min(beta, minValue)
    return value
```

```

alpha = max(alpha,value)
if beta <= alpha:
    break
return value
else:
    minValue = inf
    for each child in node:
        value = alphabeta(child, depth - 1, alpha, beta, True)
        minValue = min(minValue,value)
        beta = min(beta,value)
        if beta <= alpha:
            break
    return value

```

Pour utiliser la nouvelle fonction alphabeta, il va falloir définir  $\alpha = -\infty$  et  $\beta = +\infty$ .

### 4.3.1 Livre d'ouverture

Au début de la partie, il y a beaucoup de possibilités de déplacement qui ne semblent amener à rien pour l'IA. Le parcours d'arbre est donc assez long, et aussi peu précis car il est difficile de juger l'efficacité d'un choix si tôt dans la partie.

C'est pour cela qu'on a doté l'IA de ce qu'on appelle un livre d'ouverture. Un livre d'ouverture est un répertoire des coups classiques et efficaces que les joueurs humains utilisent en début de partie.

Pour le mettre en place, on met en relation l'IA avec un site contenant un livre d'ouverture. Au début, on avait essayé d'utiliser des techniques de web scraping pour parcourir le site et sauvegarder le résultat dans un fichier. Cependant, il suffit d'aller jusqu'à 6 coups pour créer un fichier d'environ 100 Mo, cette idée a alors été abandonnée.

A la place, avant de parcourir l'arbre, on vérifie que la partie en soit bien à ses débuts, et si c'est le cas, on se réfère au livre. Si l'adversaire joue un coup qui apparaît dans le livre, on continue de s'y référer. Sinon, c'est à l'IA de sélectionner un coup par elle-même. Ce processus est très rapide, et le temps qu'on perd à communiquer avec le site à chaque fois est minimal (de l'ordre d'une demi-seconde).

Les communications se font à l'aide du module Selenium pour récupérer les informations du site, et BeautifulSoup pour les formater. Le livre utilisé [15] consiste d'environ quelques dizaines de milliers de combinaison.

### 4.3.2 Parcours d'arbre

Les noeuds de l'arbre sont des classes qui comportent 6 attributs et 2 méthodes.

L'attribut `gameState` relie le noeud au plateau et au moteur de jeu, et les attributs `value`, `depth`, `moveHistory`, `sons` et `aiSide` servent dans le parcours d'arbre. Pour les méthodes, `computeValue` sert à évaluer le noeud, et `generateSons` sert à générer les fils du noeud.

Le but est de parcourir l'arbre le plus rapidement possible, c'est pour cela qu'on a eu recours à la technique de l'élagage alpha-bêta. Cependant, ce n'est pas suffisant car dès que des parachutages commencent à apparaître, beaucoup plus de fils sont créés à chaque itération, et le nombre de calculs augmente donc énormément. Pour palier à cela, on met en place 2 améliorations.

La première est le fait de trier les fils. En effet, en appliquant l'algorithme du quick sort pour trier les fils d'un noeud en fonction de la valeur que leur a attribué la fonction d'évaluation, on a beaucoup plus de chance de créer une situation où on va couper une branche avec l'élagage alpha-bêta. En implémentant cette amélioration, le parcours d'arbre a accéléré d'environ 15%.

Ensuite, il est nécessaire de volontairement négliger certains parachutages. Ces derniers augmentent énormément le nombre de calculs car ils créent beaucoup de possibilités. On n'en étudiera qu'une poignée parmi ceux disponibles de manière aléatoire, en les coupant volontairement.

Avec un peu plus de temps, on aurait pu essayer d'utiliser des tables de hash pour accélérer les cas de transpositions, ou permettre à l'IA de réfléchir pendant que son adversaire humain réfléchit.

### 4.3.3 Fonction d'évaluation

La fonction d'évaluation est la partie de l'algorithme qui a pour but d'attribuer aux noeuds de l'arbre des valeurs : plus cette valeur est élevée, plus l'état du jeu est favorable à l'intelligence artificielle. Pour cela, on se base sur trois principes différents [10], qui sont aussi utilisés par les joueurs humains.

#### Valeur des pièces

Premièrement, on calcule `pieceValue`. Cette variable est calculée en attribuant une valeur à chaque pièce : plus la pièce est importante et utile, plus la valeur qu'on lui attribue est élevée. Voici le tableau des valeurs qu'on a utilisé, qui est le même que celui de YSS 7.0, un algorithme de Computer Shogi datant de 1997 :

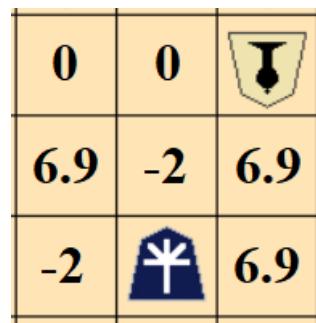
Pièce	Valeur	Valeur (capturée)	Valeur (promue)
Pion	1.00	1.15	4.20
Lance	4.30	4.80	6.30
Cavalier	4.50	5.10	6.40
Général d'argent	6.40	7.20	6.70
Général d'or	6.90	7.80	-
Fou	8.90	11.10	11.50
Tour	10.40	12.70	13.00

On va donc sommer la somme des valeurs des pièces de l'IA à chaque noeud, et ensuite on y soustrait la somme des des valeurs des pièces de son adversaire. Cette partie permet donc de reconnaître quel joueur est le plus puissant en termes de pièces possédées seulement.

Cependant, il serait trop simpliste de réduire la valeur d'un noeud à la valeur des pièces. Le positionnement de ces dernières est aussi extrêmement important, et il sera évalué par les deux prochaines variables.

### Contrôle du plateau

Deuxièmement, on va évaluer la variable `squareControl`. Cette variable sert à donner une idée de qui a le plus de contrôle sur le plateau de jeu, qui peut attaquer et/ou défendre le plus de cases sur le plateau. Pour évaluer cette valeur, on choisit une case, et on additionne la valeur de toutes les pièces pouvant atteindre cette case (on soustrait si les pièces appartiennent à l'ennemi).



**FIGURE 10 –** Exemple de calcul de `squareControl`

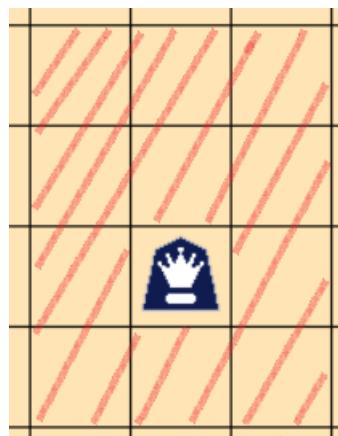
En faisant cela, on a donc une idée de qui contrôle une case, et en répétant sur toutes les cases, on aura une idée de qui contrôle le plateau entier.

### État du roi

Dernièrement, on va essayer de déterminer si le roi est en bonne position, étant donné que c'est la pièce la plus importante du jeu. On divise cette mesure

en 3 parties :

- le roi peut-il se déplacer librement ? On étudie toutes les cases autour du roi, et on vérifie si ces cases sont disponibles pour s'y déplacer. Si oui, le roi a plus de liberté et il est donc plus aisé pour lui d'échapper au danger.
- le roi est-il vulnérable aux attaques ? On définit une zone de danger (voir figure ci-dessous), qu'on va scanner pour chercher des attaques potentiels. Dès qu'on y trouve une pièce ennemie, on la considère donc comme un attaquant potentiel, et on garde sa valeur en mémoire. Une fois que la zone est scannée, on somme toutes les valeurs retenues, modulée par un poids qui varie en fonction du nombre d'attaquants. En effet, s'il n'y a qu'un seul attaquant dans la zone de danger, il est peu probable que l'attaque envers le roi soit très sérieuse. En revanche, dès qu'on y trouve 2 ou 3 pièces, il y a probablement un sérieux danger.



**FIGURE 11 – Schéma de la zone de danger**

- le roi est-il bien défendu ? On reprend là aussi le concept de zone de danger, mais cette fois-ci, on compte les pièces alliées. La seule différence est que le concept de poids ne s'applique plus, car même une seul pièce bien placée peut faire office d'une solide défense.

On additionne ces 3 composantes dans la variable `kingSafety` qui traduira donc si le roi est en bonne position ou non.

Avec ces 3 variables, on a donc essayé de prendre un compte un maximum de facteurs qui pourraient permettre à l'IA de savoir si la position est favorable ou non pour elle. On finit par sommer ces 3 variables qu'on vient de calculer pour avoir la valeur du noeud. On a attribué un poids à ces variables en les sommant pour leur donner un ordre d'importance : `pieceCount` est de loin la plus importante et prendra toujours le dessus sur les deux autres, alors que `boardControl` est légèrement plus importante que `kingSafety`. Cette fonction a un temps d'exécution très rapide et n'influe que très peu sur le temps d'exécution total du programme.

#### 4.3.4 Résultats

En implémentant un parcours d'arbre de cette manière, on arrive à chercher jusqu'à 3 coups plus tard en un temps raisonnable ( $\leq 30$  secondes). L'IA atteint un niveau décent, mais un humain normal qui réfléchit bien à ses coups arrivera en général à la battre. De plus, ce n'est pas forcément agréable de devoir attendre aussi longtemps entre chaque coup pour pouvoir jouer, et c'est pour ça qu'il est préférable de jouer contre une IA de profondeur 1 ou 2, mais ces dernières sont très faibles.

Peut être qu'en implémentant des tables de hash selon le principe de Zobrist ou en donnant à l'IA la possibilité de réfléchir pendant que son adversaire réfléchit, on aurait pu avoir de meilleurs résultats, mais nous n'avons pas eu le temps d'implémenter tout cela.

### 4.4 *Intelligence artificielle avec la méthode de Monte Carlo (MCTS) et les réseaux de neurones*

La conception d'une IA basée sur une approche différente a été menée, celle utilisant la recherche arborescente de Monte Carlo (MCTS pour Monte Carlo Tree Search). Comme l'algorithme alpha-bêta, cet algorithme fait intervenir un parcours d'arbre en évaluant les positions, mais sa philosophie est assez différente de celle d'alpha-bêta. L'IA que nous avons implémentée avec cette méthode n'est pas assez puissante à cause des trop longs calculs qu'elle requiert.

Une version alternative du MCTS a alors été implementée en incorporant un réseau de neurones, comme l'a fait DeepMind pour son IA AlphaZero. Mais là aussi les calculs étaient trop longs pour que l'IA soit opérationnelle, même en n'exigeant d'elle qu'un niveau modeste.

Une autre version a ensuite été mise en œuvre. Elle utilise simplement un réseau de neurones sans de MCTS dans le but d'accélérer les calculs. L'algorithme est en effet bien plus rapide, mais l'IA n'est pas performante.

Ces différentes tentatives auront néanmoins permis de découvrir une nouvelle approche à la conception d'une IA et d'explorer l'utilisation des réseaux de neurones, même si au final l'IA n'est pas opérationnelle avec cette méthode.

En lançant l'équipe sur deux méthodes différentes, le but était d'optimiser les chances de succès de la conception d'une IA répondant au cahier des charges.

Alpha-bêta s'est révélé être une bonne méthode pour atteindre l'objectif, tandis que Monte Carlo avait moins de chance de succès. Il était cependant difficile d'estimer l'efficacité de la méthode de Monte Carlo classique avant de l'implémenter.

Face au constat de son échec, il a été décidé d'implémenter l'algorithme d'AlphaZero malgré de nombreuses incertitudes concernant ses chances de succès à cause de sa difficulté d'implémentation et la puissance de calcul qu'elle requiert. Un risque considérable a été pris avec ce choix, risque qui n'a pas payé. Comme la méthode alpha-bêta assurait déjà une IA de niveau raisonnable, cette prise de risque était justifiée. Si elle aboutissait à un succès, elle aurait permis de concevoir une IA de niveau bien plus élevé. Par ailleurs, même si ceci sort du cadre fixé par le cahier des charges, l'IA aurait été utilisable pour de nombreux autres jeux, contrairement à alpha-bêta à cause de sa fonction d'évaluation spécifique au shogi.

#### 4.4.1 Recherche arborescente Monte Carlo

##### Description de l'algorithme

Le MCTS (Monte Carlo Tree Search) combine des simulations dites de Monte Carlo avec un arbre de recherche. Comme pour alpha-bêta, chaque nœud de l'arbre correspond à une position du jeu. Cependant, l'évaluation des nœuds se fait de manière différente avec des simulations de Monte Carlo. Alpha-bêta exige la construction d'une fonction d'évaluation, ce qui peut s'avérer très complexe si on cherche à atteindre des niveaux élevés car il faut être expert dans la connaissance du jeu (néanmoins, pour concevoir une IA de niveau modeste comme ici, la fonction d'évaluation ne pose pas de problème). Avec le MCTS, la fonction d'évaluation n'est pas directement construite mais une politique dictant la manière de jouer est élaborée. Pour évaluer un état  $s$  (*state*), on simule une partie à partir de  $s$  en suivant la politique qui décrit comment choisir le prochain coup. Une fois arrivé à un état terminal, on se sert du résultat de la partie pour évaluer  $s$ . Le problème de cette approche est que si la politique est mal choisie, l'évaluation de certains états peut être très mauvaise. Il est assez difficile de prédire l'efficacité d'une politique donnée si le jeu est complexe comme le shogi. La méthode de Monte Carlo règle ce problème de la manière suivante : on effectue plusieurs simulations à partir du même état en ajoutant à chaque fois une part d'aléatoire dans les simulations. La multiplicité de simulations pseudo-aléatoires auront des erreurs qui vont se compenser et la moyenne des résultats sera au final une bonne évaluation de l'état. [7]

L'algorithme du MCTS fait intervenir deux politiques, la politique de l'arbre et la politique stochastique. Le MCTS peut se décomposer en 4 phases répétées jusqu'à ce qu'un critère d'arrêt soit vérifié (dans l'algorithme ici élaboré, c'est le nombre d'itérations) :

- Sélection : Depuis la racine, ie. l'état  $s$  actuel, on sélectionne successivement l'enfant (c'est-à-dire le coup) selon la politique de l'arbre, c'est-à-dire l'enfant possédant le meilleur score (explicite par la suite), jusqu'à ce qu'on atteigne une feuille de l'arbre actuel.

- Expansion : Tous les enfants du nœud sélectionné sont ajoutés dans l’arbre en initialisant leur nombre de visites  $N$  et leur nombre de parties gagnées  $W$  à 0.  $N$  et  $W$  serviront à calculer le score du nœud utilisé lors de la sélection.
- Simulation : À partir d’un des nœuds rajoutés pris au hasard, on simule une partie en sélectionnant les coups selon la politique stochastique jusqu’à atteindre un nœud terminal. Dans notre cas, la politique stochastique est simplement une sélection aléatoire.
- Rétropropagation : Avec le score donné par le nœud terminal, et en partant du nœud à partir duquel on avait commencé la simulation, on met à jour le nombre de visites  $N$  et le nombre de victoires  $W$  de chaque nœud jusqu’à remonter à la racine.

Au début, l’arbre de recherche ne contient que la racine, puis au fur et à mesure des itérations, il s’agrandit avec des statistiques sur chaque nœud qui sont de plus en plus précises.

Le score d’un nœud caractérisé par son père  $s$  (state) et le coup  $a$  (action) qu’il faut jouer pour l’atteindre, est donné par la formule de l’UCT (Upper Confidence Bounds for Trees) :

$$Q(s, a) = \frac{W(s, a)}{N(s, a)} + C \sqrt{\frac{\log(N(s))}{N(s, a)}}$$

avec :

- $W(s, a)$  le nombre de parties gagnées à partir du nœud  $(s, a)$ .
- $N(s, a)$  le nombre de visites du nœud  $(s, a)$ .
- $C$  une constante ici fixée à 1.
- $N(s)$  le nombre de visites du père  $s$ .

Le terme de gauche correspond à un terme d’exploitation. Il tend à favoriser le nœud avec le meilleur score moyen et il exploite donc ce qui a été trouvé jusque-là. Le terme de droite correspond à un terme d’exploration. Il tend à favoriser les nœuds qui ont été peu explorés afin de ne pas passer à côté d’un bon coup. La formule parvient à trouver un équilibre entre exploitation et exploration lors de la phase de sélection. Lorsque  $N(s, a) = 0$ , c’est-à-dire lorsque le nœud n’a pas encore été visité, le terme d’exploration est infini et le nœud encore non exploré sera systématiquement choisi. La racine a initialement un  $N$  égal à 1.

Une fois l’arbre construit, le meilleur coup à partir de la position actuelle (racine de l’arbre) est choisie en prenant le coup avec le plus grand de nombre de visites. En effet, les meilleurs nœuds sont ceux qui ont été le plus visités comme indiqué par la formule de l’UCT (le coup avec la plus grande moyenne des parties gagnées peut aussi être choisi, c’est à peu près pareil).

Il a été prouvé que l'arbre construit par le MCTS converge vers l'arbre construit par le minimax si on lui donnait un temps infini de calcul, ce qui prouve son optimalité[2].

## Résultats

L'IA implémentée par l'équipe avec le MCTS s'est avérée bien trop lente pour être opérationnelle. En effet, pour chaque coup à choisir à partir d'un état, étant donné la complexité d'un jeu comme le shogi, il faudrait simuler au minimum de l'ordre de 100 000 parties pour que l'arbre construit ait des valeurs pertinentes. Or avec la dernière version du moteur de jeu conçu, capable de simuler environ 10 parties par seconde, ceci prendrait au minimum de l'ordre de 2h30 pour chaque coup, ce qui est évidemment inexploitable.

Pour pallier à ce problème, plusieurs solutions sont envisageables. Par exemple, la simulation aléatoire pourrait être arrêtée à une certaine profondeur pour ensuite estimer la valeur de la position atteinte avec une fonction d'évaluation comme celle utilisée pour alpha-bêta. Ceci augmenterait considérablement le nombre de simulations par coup qu'il serait possible de faire en temps limité.

Une autre solution serait aussi d'utiliser un MCTS asynchrone capable d'effectuer plusieurs simulations à la fois en effectuant des calculs parallèles.

Une dernière solution envisagée est d'essayer d'implémenter l'algorithme d'AlphaZero conçu par DeepMind. Cette solution utilise un réseau de neurones et nécessite d'effectuer beaucoup moins de simulations par coup (AlphaZero effectue 800 simulations de MCTS par coup).

La solution la plus simple est la première, mais son efficacité n'est pas du tout sûre, car même 100 000 simulations pourraient ne pas suffire. De plus, avec cette solution, l'algorithme serait très similaire à celui d'alpha-bêta. En effet, les parcours d'arbre sont certes différents, mais leur implémentation est assez rapide que ce soit pour alpha-bêta ou Monte Carlo. Le cœur de l'IA aurait été la fonction d'évaluation. Par ailleurs, la méthode utilisée par AlphaZero a suscité notre intérêt.

### 4.4.2 Alpha Zero Junior

Cette partie du projet s'appuie principalement sur les articles publiés par DeepMind sur AlphaGo Zero et sur AlphaZero [6, 4, 3], ainsi que sur le pseudo-code d'AlphaZero fournie par DeepMind [9]. Des sites internet expliquant le fonctionnement d'AlphaZero ont également été consultés [13, 1]. Ces ressources ont permis l'élaboration d'une version d'AlphaZero baptisée AlphaZero Junior. La plupart des paramètres du programme d'AlphaZero Junior ont les mêmes valeurs que celles utilisées dans l'implémentation originale.

Plusieurs projets d'une implémentation de l'algorithme d'AlphaZero ont vu le jour pour des jeux simples comme le puissance 4 ou Othello [8, 13]. Un projet sérieux l'implémentant pour le shogi a également vu le jour [11].

### Présentation d'AlphaZero

AlphaZero est une IA conçue par DeepMind faisant partie des meilleures IA dans les jeux de Go, d'échecs, et de shogi. Sa particularité est qu'elle utilise le MCTS couplé à l'utilisation d'un réseau de neurones, alors que les meilleures IA utilisaient toutes jusque-là alpha-bêta avec une fonction d'évaluation extrêmement sophistiquée élaborée à l'aide de connaissances poussées dans les stratégies du jeu. AlphaZero apprend à jouer de lui-même en n'ayant connaissance que des règles du jeu, sans aucune intervention humaine. Il découvre les stratégies du jeu par lui-même.

AlphaZero passe d'abord par une phase d'apprentissage durant laquelle il joue un grand nombre de parties contre lui-même (de l'ordre de 10 millions). Ces parties permettent à son réseau de neurones de s'améliorer. Avant de surpasser le niveau des meilleurs humains et des meilleures IA, AlphaZero s'est entraîné pendant 3 jours avec une très grande capacité de calcul mis à sa disposition. Il était impossible avec notre matériel d'atteindre le niveau de performance d'AlphaZero. Cependant, l'objectif étant simplement d'avoir une IA modeste capable de jouer des coups cohérents, atteindre un niveau raisonnable pouvait être espéré avec cet algorithme même si la phase d'apprentissage n'est pas aussi intense que celle d'AlphaZero. En effet, en analysant la courbe d'évolution de l'Elo d'AlphaZero en fonction du nombre d'itérations de la phase d'apprentissage donnée dans l'article de DeepMind [3], il est estimé qu'il avait atteint un Elo de 1000 pour le shogi (ce qui est largement au-dessus de nos objectifs) en seulement 1h (et 8 min pour les échecs). La suite montre que les temps de calcul avec notre matériel restent quand même bien trop longs et AlphaZero Junior n'est donc pas opérationnelle.

Une fois la phase d'apprentissage terminée, le temps de recherche d'un coup est assez court même avec notre matériel.

### Réseau de neurones

Dans un premier temps, on explique le concept de réseau de neurones qu'AlphaZero utilise.

Un réseau de neurones est un système prenant une entrée et donnant une sortie. Ce système doit être capable, étant donnée une entrée, de fournir la bonne sortie correspondant à l'entrée (par exemple en lui donnant une image de voiture, il doit donner la couleur de la voiture). Un réseau de neurones est capable de modéliser des fonctions très complexes. Plus précisément, un réseau de neurones

est constitué d'un très grand nombre de paramètres et prend en entrée un tenseur (un tableau de nombres multi-dimensionnel) sur lequel il effectue des opérations à partir de ses paramètres, puis il renvoie un tenseur en sortie. Tout au long des différentes opérations, le tenseur donné en entrée peut changer de forme, et se transforme peu à peu en le tenseur de sortie. Le réseau est organisé en plusieurs couches et le tenseur d'entrée passe par chaque couche où il subit à chaque fois des opérations à partir des paramètres. Le nombre de paramètres d'un réseau de neurones peut être de l'ordre du million, même si ceci dépend fortement de son architecture. Le but est de trouver les bonnes valeurs des paramètres afin que le réseau puisse donner la bonne sortie en réponse à une entrée. Pour cela, le réseau doit passer par une phase d'entraînement. Initialement, les paramètres du réseau ont des valeurs aléatoires. On fournit au réseau des exemples d'associations entrée-sortie. Le réseau va apprendre de ces exemples quelles sont les bonnes valeurs de sortie pour les différentes entrées qu'ils pourraient recevoir. Il existe différents algorithmes permettant d'optimiser les paramètres du réseau étant données des exemples. La plupart de ces algorithmes sont basés sur l'algorithme du gradient stochastique. Si on donne aux réseaux suffisamment d'exemples, il pourra être capable, dans une certaine mesure, de généraliser ses réponses à des entrées qu'il n'a jamais rencontrées dans les exemples.

Dans le cas du réseau de neurones d'AlphaZero, celui-ci prend en entrée une position du jeu qu'on met sous forme de tenseur. En sortie, il renvoie une évaluation de la position entre -1 et 1 (plus le nombre est grand, meilleure est la position) ainsi qu'une distribution de probabilités pour chaque coup possible du jeu, c'est-à-dire qu'à chaque coup lui est associé une probabilité qu'on peut interpréter comme étant la probabilité que ce soit le meilleur coup possible. À partir d'une certaine couche, le réseau se sépare en deux parties, l'une donnant l'évaluation, l'autre la distribution de probabilités.

Pour l'implémentation du réseau, PyTorch a été utilisé. Il s'agit d'une librairie de machine learning pour python.

L'architecture du réseau de neurones d'AlphaZero Junior est similaire à celle d'AlphaZero [4, 3]. Seule la forme de son entrée est différente. Les détails de son architecture sont fournis en annexe.

### MCTS avec un réseau de neurones

Pour déterminer le meilleur coup pour une position donnée, AlphaZero utilise un MCTS incorporant le réseau de neurones qui a été présenté. La formule de l'UCT utilisée dans la phase de sélection est modifiée comme suit :

$$Q(s, a) = \frac{W(s, a)}{N(s, a)} + C \cdot p(s, a) \sqrt{\frac{N(s)}{1 + N(s, a)}}$$

La principale différence avec la formule originale de l'UCT est la présence du  $p(s, a)$  qui est la probabilité associée au coup  $(s, a)$  renvoyée par le réseau de neurones. La formule montre que la probabilité correspond à une probabilité a priori, obtenue avant toute recherche dans l'arbre de jeu. Cette valeur permet de diriger la recherche dans ses débuts afin de se focaliser sur les coups vraiment pertinents sans perdre de temps sur les coups de toute évidence mauvais. Cette façon de raisonner est similaire à ce qu'un joueur de shogi (ou d'échecs) peut faire. Un bon joueur laissera en effet tomber les mauvais coups par un seul coup d'œil du plateau de manière instinctive. Une fois la recherche bien entamée, le terme d'exploitation prendra progressivement le dessus dans le score, de la même manière qu'un bon joueur peaufine son jeu au fur et à mesure de sa réflexion.

La phase d'expansion n'est pas modifiée.

La phase de simulation est différente. Au lieu de simuler une partie aléatoire à partir du nœud  $n$  sélectionné afin de déterminer quel est le score de  $n$ , on prend simplement l'évaluation  $v$  de  $n$  donnée par le réseau de neurones.

Enfin, la rétropropagation va consister à ajouter  $v$  au  $W$  de chaque nœud rencontré lors de la phase de sélection.

Jusqu'ici, cette version du MCTS n'a aucun part d'aléatoire et est parfaitement déterministe. Pour rajouter de l'aléatoire, essentiel pour l'exploration, on rajoute à la distribution de probabilités  $p$  un bruit de Dirichlet :

$$p(s) \leftarrow (1 - \epsilon)p(s) + \epsilon\eta$$

où  $\eta \sim Dir(0, 15)$  et  $\epsilon = 0,25$ .

Une fois l'arbre du MCTS construit, pour obtenir le meilleur coup possible, le coup ayant le plus grand  $N$  est choisi comme pour le MCTS classique.

## Construction des données d'entraînement

Pour construire les données d'entraînement à fournir au réseau de neurones, AlphaZero Junior génère des parties contre elle-même avec le MCTS et le réseau de neurones qu'il possède actuellement. Mais il ne suffit pas de faire des parties et sauvegarder les positions obtenues, il faut aussi donner un label à ces positions, c'est-à-dire ici une distribution de probabilités et une évaluation pour chaque position.

Pour la distribution de probabilités, une fois construit l'arbre du MCTS, on pose :

$$p(s, a) = \frac{N(s, a)^\tau}{\sum_b N(s, b)^\tau}$$

où la somme parcourt tous les enfants de  $s$ , et où  $\tau$  est un paramètre valant ici 1.

Les coups illégaux ont une probabilité nulle qui leur est attachée. Cette distribution correspond à une version améliorée de la distribution initiale fournie par le réseau de neurones. En effet, cette nouvelle distribution est le fruit d'une recherche dans l'arbre ayant exploité les évaluations des différentes positions fournies par le réseau.

Pour l'évaluation, une fois la partie de l'IA contre elle-même terminée, le résultat de la partie est prise, c'est-à-dire  $v = -1$  si la couleur qui joue à la position évaluée est le gagnant (car alors cette position sera perdante pour la couleur précédente, et c'est elle qui a besoin de savoir si cette position est bonne ou mauvaise),  $v = 1$  si c'est le perdant, et  $v = 0$  s'il y a égalité.

### Déroulement de la phase d'apprentissage

La phase d'apprentissage d'AlphaZero Junior est la répétition de l'itération décrite ci-dessous.

D'abord, AlphaZero Junior joue 25 000 parties contre lui-même. Ces parties sont sauvegardées dans un buffer de capacité de stockage limitée à 100 000 parties. Chaque partie est limitée à un maximum de 512 coups. Si la partie ne se termine pas avant, on la considère comme étant une égalité. Si le buffer atteint sa capacité maximale, les premières parties sont supprimées et les nouvelles sont ajoutées à la fin.

Une fois les nouveaux exemples de position générés, AlphaZero Junior procède à l'amélioration de son réseau de neurones. Il effectue 4000 pas d'entraînement, chacun de ces pas correspondant à un pas de l'optimisation d'Adam avec un taux d'apprentissage fixé à 0,01. On prélève aléatoirement du buffer un batch de 1024 positions qu'on associe chacune à un label défini précédemment. Cette sélection aléatoire de positions pour l'entraînement permet de limiter le phénomène d'overfitting [5], c'est-à-dire le fait que l'IA ne fasse que mémoriser les parties d'entraînement au lieu d'essayer de généraliser à partir de celles-ci.

Cette itération est répétée un maximum de fois (AlphaZero l'a répété 700 fois pour 3 jours de calcul). À chaque itération, le réseau de neurones s'améliore, ce qui a pour effet d'améliorer la qualité des parties générées et la qualité des labels associés aux positions, ce qui améliore encore le réseau de neurones, et ainsi de suite.

La phase d'apprentissage est alors achevée. Il n'y a plus qu'à utiliser ce réseau de neurones entraîné couplé au MCTS pour jouer des parties contre d'autres joueurs.

## Résultats

La phase d'apprentissage d'AlphaZero Junior a été lancée sur Google Colab qui propose de bons GPU (un seul est disponible par session), permettant de paralléliser de manière importante les calculs faits par le réseau de neurones, ce qui accélère considérablement le temps de calcul. La parallélisation a été faite avec cuda qui est une API permettant la parallélisation de certains calculs et prise en charge par pytorch.

AlphaZero Junior est bien trop lent durant sa phase d'apprentissage pour être exploitable. Il fait environ 100 parties en 16 heures. Donc pour faire 10 000 parties, ce qui est le strict minimum (juste une itération complète devrait en réalité faire 25 000 parties), cela prendrait plus de 2 mois de calculs.

Pour pallier à ce problème, le MCTS asynchrone peut être employé, mais il est très probable que cela ne suffise pas.

Une autre solution est d'utiliser des parties déjà faites par des joueurs humains ou d'autres IA et de les fournir à AlphaZero Junior pour son entraînement. Il n'aurait alors plus besoin de générer lui-même les parties d'entraînement, ce qui faisait en très grande partie la lenteur de la phase d'apprentissage. Cependant, à ce stade du projet, nous manquions de temps pour essayer cette solution.

Il a alors été décidé de tester une solution plus simple et plus rapide qui consiste à concevoir une IA qui n'utilise pas de MCTS, mais juste un réseau de neurones. Ce sera l'objet de la partie suivante.

### 4.4.3 Simple réseau de neurones (SNN)

Une version très simplifiée d'AlphaZero Junior a été implémentée afin de réduire au maximum le temps de calcul. Néanmoins, même si le temps de calcul a effectivement été très largement réduit, l'IA obtenue, baptisée SNN (Simple Neural Network), n'est pas exploitable.

#### Description de l'algorithme

L'algorithme proposé consiste à ne plus utiliser de MCTS et à s'appuyer entièrement sur le réseau de neurones. Cette idée s'appuie sur l'article d'AlphaGo Zero [6] à la page 12 qui montre qu'une IA basée seulement sur le réseau de neurones d'AlphaGo Zero atteint tout de même un Elo de 3055 au Go, ce qui est le niveau d'un joueur top professionnel. Néanmoins, le réseau a bien été entraîné en se servant du MCTS, ce qui laissait un doute quant à la performance de SNN, finalement trop mauvaise.

Le déroulement global de la phase d'apprentissage est identique à celle d'AlphaZero. Les seules différences sont le réseau de neurones, la façon de

choisir les coups (sans MCTS), et la manière de fixer un label à une position.

Le réseau est légèrement modifié. Au lieu de renvoyer en sortie la distribution de probabilités et l'évaluation de la position, il renvoie un tenseur  $V$  de même forme que celle de la distribution de probabilités, et associant à chaque coup du jeu une évaluation de celui-ci entre -1 et 1 à partir de la position donnée en entrée. Ceci équivaut à évaluer toutes les positions après la position actuelle en une seule fois. L'architecture du réseau de neurones est la même que celle d'AlphaZero mais sans la partie donnant la valeur de l'évaluation de la position, et en appliquant  $\tanh$  au tenseur de sortie au lieu de  $\log_{softmax}$ .

Une fois l'évaluation faite, le coup ayant la meilleure évaluation est choisi, sans MCTS.

Pour une position donnée, le label qui lui est associé est le tenseur des évaluations des coups renvoyé par le réseau de neurones actuel (donc aucune modification par rapport à ce que produit déjà le réseau), auquel nous modifions seulement l'évaluation du coup qui a été joué à cette position. Si le joueur ayant joué ce coup est le gagnant de la partie, le coup est évalué à 1. Si ce joueur est perdant, l'évaluation vaut -1. Dans le cas d'une égalité, l'évaluation vaut 0. Finalement, ce label est le même utilisé que pour le label de l'évaluation d'une position pour AlphaZero Junior. On a également la possibilité de donner une évaluation de -1 à tous les coups illégaux, mais ceci n'a pas modifié les résultats.

### Améliorations additionnelles

Les parties jouées par SNN ont été en partie parallélisées. Ce dernier évalue 25 000 positions en même temps, ce qui accélère significativement le temps de calcul (environ 3 fois plus rapide).

Durant la phase d'entraînement du réseau de neurones, lorsqu'une position est sélectionnée pour l'ajouter dans le batch, celle-ci est identifiée par le nombre de coups qui la précédent. Il est alors nécessaire de générer cette position avant de la donner au réseau en partant de la position initiale et en jouant les coups qui précèdent la position voulue (les coups sont récupérés dans l'historique de la partie). Ce processus prend du temps car on fait beaucoup intervenir le moteur de jeu. Pour l'accélérer, durant la génération d'une partie, une position est sauvegardée toutes les 50 positions. Pour retrouver une position donnée à partir de l'historique des coups, il suffit alors de partir de la dernière position sauvegardée avant la position voulue au lieu de partir du début, ce qui accélère le temps de calcul. Un compromis entre mémoire utilisée (sauvegarde des positions), et temps de calcul est ici trouvé.

Le suivi de la phase d'apprentissage a été amélioré afin de savoir où en est SNN dans son apprentissage. Plusieurs variables importantes ont été introduites à cette fin : nombre d'itérations effectuées, nombre de parties générées, nombre

de pas d'entraînement effectués, temps de calcul, pourcentage d'égalités, pourcentage de parties identiques générées, valeur moyenne du maximum du tenseur des évaluations décomposée en la partie sans le bruit et la partie correspondant au bruit, pourcentage des coups décidés par le réseau de neurones et non à cause du bruit, etc.

La structure du code a également été améliorée avec l'introduction d'une classe RunManager permettant le suivi simple des différentes variables importantes tout en gardant un programme propre et organisé. Tous les paramètres du programme ont été regroupés en tant qu'attributs de la classe Configuration, dont une instance se trouve dans RunManager.

Des sauvegardes automatiques de l'état de progression de l'apprentissage ont été mises en place. Les sauvegardes concernent l'instance de RunManager, le buffer avec les dernières parties générées, les sauvegardes spontanées des positions, et les paramètres du réseau de neurones.

## Résultats

SNN ne fait pas de recherche dans un arbre. Sans les améliorations ci-dessus, il est donc de l'ordre de 800 fois plus rapide qu'AlphaZero Junior car celui-ci fait 800 simulations avec le MCTS pour chaque coup. Au final, durant la phase d'apprentissage, SNN effectue une itération complète en environ 8h30, ce qui est environ 950 fois plus rapide qu'AlphaZero Junior.

Le revers de ce gain drastique en temps de calcul est l'intelligence de SNN. Celui-ci ne fait plus de recherche dans un arbre et ne s'appuie donc que sur le réseau de neurones. Ceci s'est avéré fatal sur la performance de SNN finalement non exploitable. En effet, à certains moments de la partie, celui-ci répète les mêmes coups pendant longtemps (par exemple en déplaçant une tour à une certaine position puis en la faisant revenir, et ce plusieurs fois). Cependant, ce problème ne semble pas être exclusif à SNN car le même problème était apparu avec AlphaZero Junior (la lenteur de son apprentissage le rendait de toute façon complètement inexploitable et il est donc difficile de le comparer avec SNN). Pour AlphaZero Junior, nous avions tenté de régler le problème en pénalisant les répétitions mais ceci ne s'est pas avéré fructueux, car au final l'IA tentait toujours de faire des répétitions même en étant pénalisée. Malgré de nombreux tests effectués, notamment en analysant les différentes positions avec leur label données en entraînement, nous n'avons à ce jour pas trouvé la véritable cause de ce comportement. Une raison possible est l'overfitting du réseau de neurones aux positions générées. En effet, ceci semblerait cohérent avec le fait que le réseau s'acharne sur un coup en particulier malgré les différences entre les positions qu'on lui demande d'analyser.

Afin d'écartier l'existence de bugs, SNN a été testée avec le jeu du morpion en conservant au maximum le code original. Sur le morpion, dont on sait qu'il

est toujours possible de finir en égalité dans le pire des cas, SNN fonctionne correctement, ce qui semble écarter l'existence de bugs. Mais ses résultats ne sont pas optimaux. Il parvient à atteindre un taux de victoire de 94 % contre une IA aléatoire, avec 2 % d'égalités, 100 % de victoires s'il joue en premier et 88 % de victoires s'il joue en second. Ces performances sont atteintes après environ 15 itérations de 2500 parties et de 100 pas d'entraînement, après quoi il ne parvient pas à progresser davantage. Il y a donc environ 4 % des parties qu'il perd alors qu'il pourrait théoriquement l'éviter. Même sur un jeu aussi simple que le morpion, SNN ne parvient donc pas à être optimal. SNN semble donc avoir un problème de convergence vers une bonne solution. Il est au mieux extrêmement lent à converger vers une bonne solution. Transposé à un jeu aussi complexe que le shogi, il semble ainsi cohérent que SNN ait du mal à apprendre correctement. La recherche dans un arbre semble donc être assez important pour l'apprentissage de l'IA.

Une piste non explorée pour une IA performante est de collecter des parties déjà faites sur des sites internet et de les fournir comme données d'entraînement à l'IA. Cette méthode est employée par AlphaGo, le prédecesseur d'AlphaGo Zero et d'AlphaZero. Cependant, l'article d'AlphaGo Zero [6] montre que ce dernier est largement plus performant qu'AlphaGo qui s'est pourtant entraîné pendant plusieurs mois, bien plus comparé aux 3 jours d'entraînement d'AlphaGo Zero.

## 5 Conclusion

En somme, ce projet fut une expérience très enrichissante. Toute l'équipe a su se servir de nouveaux outils, et la visée éducative a donc été remplie.

En termes d'objectifs, nous sommes tout de même heureux d'avoir rempli une bonne partie de ceux qu'on s'était fixés au départ. Le tutoriel remplit bien sa fonction, et l'interface graphique est fonctionnelle et intuitive. Malheureusement, le calcul d'elo fut impossible, et n'ayant qu'une IA fonctionnelle, nous n'avons pas pu permettre les parties entre IA. Pour les IA, MCTS a eu beaucoup de mal à fonctionner car elle nécessite une très grosse puissance de calcul, et elle n'a donc abouti à rien. De l'autre côté, Alpha-Bêta n'a pas brillé par son niveau ou sa vitesse non plus, mais elle a tout de même réussi à faire un adversaire viable lorsque son niveau est suffisamment élevé.

Avec un peu plus de temps, peut être que les IA auraient pu être plus efficaces : nous avons déjà parlé précédemment de ce qui aurait pu être implémenté en plus. Il aurait peut être aussi été possible d'écrire un tutoriel plus extensif, ou de créer une interface plus belle.

## Annexes

### A Règles du Jeu de Shogi

#### A.1 Objectif

L'objectif est de **prendre le roi adverse**.

Quand le roi ne peut éviter d'être pris au coup suivant, la partie est terminée. On dit que le roi est "Mat".

#### A.2 Joueurs

Le Shogi oppose deux adversaires qui jouent chacun à son tour, un avec des pièces noires et l'autre blanche. En général, c'est le joueur noir qui commence la partie.

#### A.3 Plateau

Le plateau de Shogi se nomme "Shogi ban" (table de Shogi) et comporte **81 cases** ( $9 \times 9$ ). Les trois lignes les plus éloignées des joueurs constituent leur **zone de promotion**.

#### A.4 Pièces

Au début de la partie, **chaque joueur dispose de 20 pièces** : un **roi**, une **tour**, un **fou**, deux **généraux d'or**, deux **généraux d'argent**, deux **cavaliers**, deux **lances** et neuf **pions**. Toutes les pièces sont plates, de forme légèrement pointue et de couleur claire. **Les pièces de chaque camp sont reconnaissables à la direction qu'elles indiquent par leur pointe.**

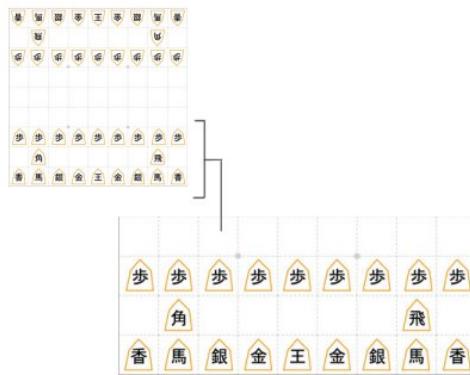


FIGURE 12 – Table de Shogi

### A.5 Déplacements

**Flèche bleue** = déplacement d'une case dans la direction indiquée.

**Flèche rouge** = déplacement d'autant de cases que l'on veut dans la direction indiquée.

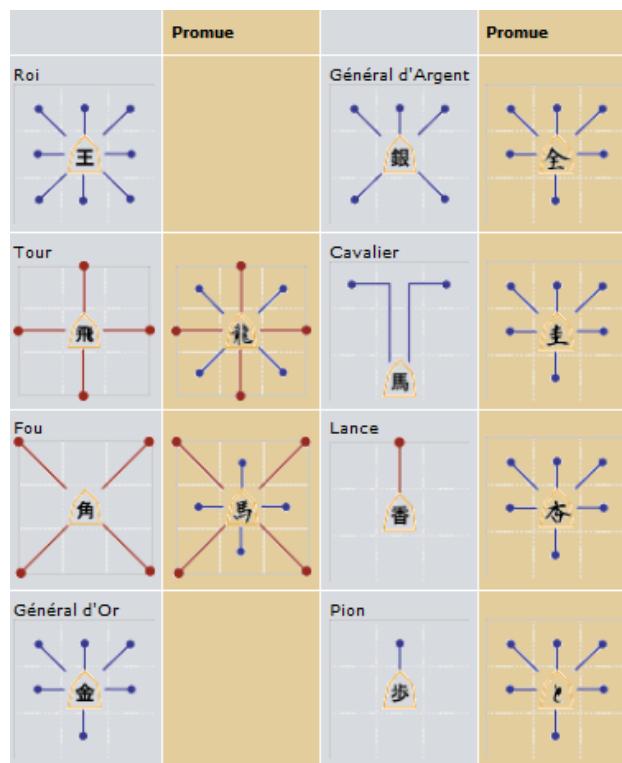


FIGURE 13 – Mouvement des pièces

## A.6 Prise

Lorsqu'une pièce effectue un déplacement et se trouve sur une case déjà occupée par une pièce du camp adverse, celle-ci est prise et enlevée de l'échiquier. Lorsqu'un joueur prend ces pièces à l'adversaire, il doit les garder bien en évidence sur le côté à sa droite de l'échiquier.

## A.7 Promotion

Les deux lignes horizontales supportant les points noirs sur le dessin sont nommées "lignes de promotions". **Quand les pièces pénètrent dans la zone de promotion, elles peuvent être promues**, si on le souhaite, sous certaines conditions. Pour ce faire, la pièce est retournée, révélant ainsi sa qualité de pièces promue. Elle gagne alors les **possibilités des mouvements supplémentaires**. (*cf. tableau des déplacements*).

Si, dans un premier temps, le joueur, le joueur choisit de ne pas promouvoir une pièce, celle-ci peut être promue plus tard, à n'importe quel moment, en la déplaçant sur le Shogi ban, soit entièrement soit partiellement dans la zone de promotion. Si le déplacement a lieu uniquement hors de la zone de promotion, la promotion n'est pas possible. **La promotion reste permanente tant que la pièce reste sur l'échiquier.**

La promotion est obligatoire pour le pion ou la lance arrivant sur la dernière du plateau, et pour le cavalier sur l'avant dernière ou la dernière ligne. En effet, à ce stade, ces pièces non promues ne pourraient plus se déplacer.

## A.8 Parachutage

Les joueurs peuvent **réutiliser les pièces qu'ils ont prises pour leur propre camp et les replacer sur n'importe quelle case vide du jeu**. Ceci compte pour un coup. Cette règle comporte néanmoins quelques particularités :

1. Une pièce est toujours parachutée non promue.
2. Pour promouvoir une pièce parachutée, il faut la déplacer entièrement ou partiellement dans la zone de promotion.
3. On ne peut pas parachuter une pièce sur une case d'où elle ne pourrait plus partir.
4. Un pion ne peut pas être parachuté sur une colonne où se trouve déjà un pion non promu du même camp.
5. Un pion ne peut pas être parachuté devant le roi pour faire échec et mat directement.

### A.9 Nulles

Les parties nulles sont rares au Shogi. Néanmoins, quelques cas sont considérés.

1. Une partie est nulle si une position se répète à 4 reprises, pièces à parachuter comprises ("**Sennichite**"). En cas de nulle, les joueurs jouent une nouvelle partie, Sente (qui commence le premier coup) devenant Gote (qui fait le deuxième coup) et vice-versa, avec le temps restant. Si cette séquence provoque une mise en échec continue, l'attaquant perd la partie.
2. Une partie peut aussi être considérée comme nulle en cas de "**Jishogi**" : les deux rois arrivent à rentrer dans le camp adverse et ne peuvent pas être mis mat. On additionne alors la valeur des pièces possédées par chaque joueur comme suit : Roi = 0, Tour/Fou = 5, les autres = 1. Si chaque joueur a 24 points au moins, la partie est nulle. Si un joueur refuse de compter ses pièces, la partie est continuée.

## B Détails du moteur de jeu

La 3<sup>ème</sup> version du moteur de jeu est ici décrite.

Tout d'abord, il est assez simple de déterminer les coups pseudo-légaux, c'est-à-dire les coups déduits des mouvements des pièces, y compris ceux qui mettent le roi allié en échec à la fin du coup (ce qui est normalement illégal).

S'il s'agit d'un coup déplaçant une pièce sur le plateau (donc pas un parachutage), il suffit de partir de la position de la pièce et de se déplacer selon les mouvements autorisés par la pièce. Si la pièce est glissante, c'est-à-dire qu'elle peut se déplacer de plusieurs cases en un coup (lance, tour, et fou, le cavalier étant exclu), il suffit d'autoriser toutes les cases en partant de la position de la pièce et en se déplaçant dans les directions autorisées jusqu'à rencontrer une autre pièce ou à atteindre la bordure du plateau.

En ce qui concerne les promotions, pour un déplacement vers une case donnée :

- Si la pièce a l'obligation d'être promue, on n'autorise que le coup qui promeut.
- Si ce n'est pas le cas mais que la pièce peut être promue, on autorise le coup qui promeut et celui qui ne promeut pas.
- Sinon, on n'autorise que le coup qui ne promeut pas.

Pour les parachutages, toutes les cases libres peuvent être choisies, sauf pour les cas suivants :

- La dernière ligne pour un pion ou une lance car ils ne pourraient plus bouger.
- Les deux dernières lignes pour un cavalier car il ne pourrait plus bouger.
- Pour le cas d'un pion, une colonne où se trouve déjà un pion allié.
- Pour le cas d'un pion, un parachutage devant le roi adverse si ce coup fait mat (échec simple autorisé). Ce cas est délicat et sera traité par la suite.

Sachant cela, il s'agit d'éliminer tous les coups mettant le roi allié en échec. Nous pouvons par exemple tester le coup pseudo-légal et voir si le roi allié est en échec, ou encore parcourir toutes les pièces alliées pour voir lesquelles sont capables de protéger le roi. Mais ce genre de solution effectue beaucoup de calculs inutiles. Afin d'être le plus rapide possible en ne faisant que les calculs vraiment pertinents, nous pouvons distinguer trois cas :

- Situation 1 : Le roi allié est mis en échec par au moins deux pièces adverses. Dans ce cas, si on tente de sauver le roi en déplaçant une de nos pièces pour servir de couverture face à une pièce adverse glissante, ou en capturant une des pièces adverses mettant en échec, alors au moins l'une des autres pièces adverses met toujours le roi en échec. Nous n'avons donc d'autres choix que de déplacer le roi vers une case non atteignable par une pièce adverse (une case libre ou occupée par une pièce adverse).
- Situation 2 : Le roi allié est mis en échec par exactement une pièce adverse. On peut alors déplacer le roi en lieu sûr comme avant, mais on peut aussi faire intervenir une autre pièce. On peut en effet soit capturer la pièce adverse menaçant le roi, soit s'interposer entre cette pièce et le roi si la pièce est glissante (lance, tour et fou). Pour capturer ou s'interposer, on déplace une pièce du plateau qui ne protège pas déjà le roi contre une pièce glissante. Pour savoir si une pièce notée  $p_0$  est en train de protéger son roi, on peut regarder toutes les pièces glissantes atteignant la case sur laquelle  $p_0$  se trouve, et observer si  $p_0$ , son roi, et la pièce adverse sont alignés. Pour le cas d'une interposition, on peut aussi parachuter une pièce (ceci ne s'applique pas pour une capture car il est interdit de capturer par parachutage).
- Situation 3 : Le roi allié n'est pas mis en échec. On autorise alors tous les coups pseudo-légaux exceptés ceux qui enlèvent une couverture du roi comme précédemment.

Pour éviter le parachutage d'un pion faisant mat, on regarde l'un des deux cas suivants :

- Si le roi adverse pouvait se déplacer avant le parachutage, alors il n'y a pas mat car il peut toujours se déplacer. En effet, même si la seule case possible est celle où le pion est parachuté, il peut le capturer sans danger.
- Si une pièce adverse peut atteindre la case où le pion est parachuté, alors le pion pourra être capturé et il n'y a donc pas mat.

---

Si aucun de ces deux cas n'est vérifié, alors il y a mat et le parachutage est illégal.

Dans les différentes solutions proposées ci-dessus, un point essentiel est de savoir quelles sont les pièces pouvant atteindre une case donnée. Ceci doit pouvoir être su de manière rapide, sinon tout ce qui a été proposé est vain. Pour cela, nous utilisons un tableau représentant le plateau et qui, pour chaque case du plateau noté  $c_0$ , contient un dictionnaire contenant deux éléments. Le premier élément du dictionnaire est l'ensemble des pièces noires pouvant atteindre  $c_0$ , et le deuxième élément est l'ensemble des pièces blanches pouvant atteindre  $c_0$ . Nous utilisons les set de Python car ces structures de données permettent par exemple de tester l'existence d'un élément en temps constant. On considère qu'une pièce peut atteindre une case même si le coup serait illégal car mettant son roi en échec. On ne prend donc en compte que le mouvement des pièces. Nous excluons les pièces capturées et donc les parachutages car ceux-ci ne rentrent pas en jeu ici. Le tableau est appelé PRI (*Pieces Reaching It*).

Le dernier point essentiel du moteur de jeu concerne alors la mise à jour de ce tableau PRI. Après chaque coup, il est nécessaire de le mettre à jour. Au lieu de parcourir toutes les pièces du plateau pour reconstruire PRI de zéro à chaque fois, on peut garder le PRI de l'itération précédente et ne mettre à jour que les pièces concernées par le coup :

- Si le coup est le déplacement d'une pièce  $p_1$  de la case  $c_1$  à la case  $c_2$ , alors on traite la pièce  $p_1$  et les pièces atteignant  $c_1$  et celles atteignant  $c_2$ , pièces qu'on peut obtenir facilement justement grâce à PRI. Pour ce qui concerne  $p_1$ , il suffit de supprimer cette pièce des cases de PRI qu'elle pouvait atteindre avant le coup, puis de la rajouter dans les nouvelles cases qu'elle peut atteindre. On traite ensuite les autres pièces. Pour une case  $c_0$  de PRI, même si  $c_0$  est occupée par une pièce  $p_0$ , on comprend ici qu'il est judicieux d'inclure dans l'ensemble des pièces atteignant  $c_0$  les pièces des deux camps, y compris les pièces alliées à  $p_0$ , même si en réalité celles-ci ne peuvent atteindre  $c_0$ . Ceci est utile pour permettre une mise à jour de PRI qui soit simple et rapide. Avec cette convention, les pièces non glissantes (y compris le cavalier) ne sont pas affectées par la mise à jour de PRI. Il s'agit de ne traiter que les pièces glissantes (lance, tour et fou). Si une pièce glissante atteignait  $c_1$  (maintenant libre), alors la pièce peut maintenant atteindre les cases au-delà. Si elle atteignait  $c_2$ , la pièce est bloquée à partir de  $c_2$ . Il faudra cependant faire attention aux situations où la pièce glissante mise à jour se trouve sur le même axe de déplacement de la pièce déplacée (si cette dernière se rapproche ou s'éloigne de la pièce glissante). Si une pièce a été capturée, il faudra la supprimer des cases qu'elle pouvait atteindre.
- Si le coup est le parachutage d'une pièce  $p_1$  sur une case  $c_1$ , on rajoute  $p_1$  dans les cases qu'elle peut maintenant atteindre et on supprime les pièces glissantes atteignant  $c_1$  des cases au-delà de  $c_1$ .

---

Ceci complète la mise à jour de PRI.

Pour récapituler, les points les plus délicats du moteur de jeu sont les suivants :

- La mise en échec d'un roi, où il faut d'une part déterminer si le roi est mis en échec, et d'autre part quels sont les coups légaux si tel est le cas.
- La gestion des parachutages. Ceux-ci rentrent en jeu notamment dans la mise en échec d'un roi. Il faut aussi qu'un pion parachuté ne fasse pas mat.
- La mise à jour du tableau PRI.

## C Architecture du réseau de neurones

Architecture du réseau de neurones d'AlphaZero Junior :

- L'entrée est un tenseur de la forme  $43 \times 9 \times 9$ . Sa constitution est la même que celle décrite dans l'article d'AlphaZero [4] mais sans les tableaux pour les répétitions et pour l'historique. Pour résumer, il s'agit d'un tenseur constitué de 43 tableaux de  $9 \times 9$  (représentant le plateau de jeu) composés de 0 et de 1 indiquant la présence ou non du type de pièce que le tableau prend en charge.
- Un bloc de convolution :
  - Une couche de convolution de 256 filtres de taille  $3 \times 3$  avec un pas de 1.
  - Une normalisation du batch.
  - Un *ReLU*.
- Une succession de 19 blocs résiduels dont chacun est constitué de :
  - Une couche de convolution de 256 filtres de taille  $3 \times 3$  avec un pas de 1.
  - Une normalisation du batch.
  - Un *ReLU*.
  - Une couche de convolution de 256 filtres de taille  $3 \times 3$  avec un pas de 1.
  - Une normalisation du batch.
  - Une somme de l'entrée et du bloc.
  - Un *ReLU*.
- La partie donnant la distribution de probabilités :
  - Une couche de convolution de 256 filtres de taille  $1 \times 1$  avec un pas de 1.
  - Une normalisation du batch.

- Un *ReLU*.
- Une couche de convolution de 160 filtres de taille  $1 \times 1$  avec un pas de 1.
- Un *log\_softmax*.
- La partie donnant l'évaluation de la position :
  - Une couche de convolution de 256 filtres de taille  $1 \times 1$  avec un pas de 1.
  - Une normalisation du batch.
  - Un *ReLU*.
  - Une couche linéaire avec une entrée de taille  $9 \times 9$  et une sortie de taille 256.
  - Un *ReLU*.
  - Une couche linéaire avec une entrée de taille 256 et une sortie de taille 1.
  - Un *tanh*.
- La sortie est le couple  $\log_p$  (logarithme de la distribution de probabilités) et  $v$  (évaluation).  $v$  est un réel entre -1 et 1.  $\log_p$  est un tenseur de taille  $139 \times 9 \times 9$ . Sa constitution est exactement celle décrite dans l'article d'AlphaZero [4]. Chaque case du tenseur correspond à un coup possible du shogi. Certains coups représentés dans ce tenseur ne sont jamais possibles (par exemple débordant du plateau) et la plupart sont souvent illégaux pour une position donnée. Mais la sortie du réseau de neurones doit être un tenseur de dimensions fixées et cette représentation a donc été choisie en incluant tous les coups possibles.

## D Budget

Personnels	Coût horaire (euro)	Coût global (euro)
Tuteurs	41	820
Conseillers	20	400
<b>TOTAL</b>		<b>1220</b>

Tableau B1 : Rémunération des personnels

Amortissement	NB Heures estimé	Coût horaire estimé	TOTAL
Ordinateurs personnels (6)	180	3	540

Tableau B2 : Amortissement des équipements

Financement reçu de :	Valeur en euro
ECL (Tuteurs, ..)	1220
Groupe PE 110 (Ordinateurs)	540
Financement dépensé des 300 euros	0
<b>Total des financements</b>	<b>1760</b>

Tableau B3 : Financements

**FAISABILITÉ en terme de financement : Vérifiée**

## E Tableau de vérification

**Check-list de rapport de Projet d'Etudes**  
**A remplir par les rédacteurs (élèves)**  
**et à insérer en dernière page du rapport**

### A développer

Renseigner la case par le nom du responsable, ou la date ou une simple croix lorsque la vérification a été faite.

	Vérification présence	Vérification qualité
--	-----------------------	----------------------

### Contenu

Résumé en français	✓	✓
Résumé en anglais	✓	✓
Table des matières	✓	✓
Table des figures	✓	✓
Introduction	✓	✓
Conclusion générale	✓	✓
Bibliographie	✓	✓
Citation des références dans le texte	✓	✓

### Forme

Vérification orthographe		✓
Pagination		✓
Homogénéité de la mise en page		✓
Lisibilité des figures		✓

## Références

- [1] *A Simple Alpha(Go) Zero Tutorial*. URL : <https://web.stanford.edu/~surag/posts/alphazero.html>. (consulté : 01.06.2020).
- [2] Cameron Browne et AL. "A Survey of Monte Carlo Tree Search Methods". In : 2014, p. 9.
- [3] David Silver et AL. "A general reinforcement learning algorithm that masters chess, shogi and Go through self-play". In : (2018).
- [4] David Silver et AL. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In : (2017).
- [5] David Silver et AL. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In : *Nature* (2016).
- [6] David Silver et AL. "Mastering the Game of Go without Human Knowledge". In : *Nature* (2017).
- [7] Sylvain Gelly et AL. "The Grand Challenge of Computer Go : Monte Carlo Tree Search and Extensions". In : (2012).
- [8] *alpha-zero-general*. URL : <https://github.com/suragnair/alpha-zero-general>. (consulté : 01.06.2020).
- [9] *AlphaZero pseudocode*. URL : <https://github.com/jianpingliu/AlphaZero>. (consulté : 01.06.2020).
- [10] *An AI for Shogi*. URL : <http://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/restricted/posters/reguchi/poster.pdf>. (fonction d'évaluation).
- [11] *aobazero*. URL : <https://github.com/kobanium/aobazero>. (consulté : 01.06.2020).
- [12] *Artificial Intelligence | Alpha-Beta*. URL : <https://www.javatpoint.com/ai-alpha-beta-pruning>.
- [13] *Lessons From Implementing AlphaZero*. URL : <https://medium.com/oracledevs/lessons-from-implementing-alphazero-7e36e9054191>. (consulté : 01.06.2020).
- [14] H. J. R. MURRAY. *A history of chess*. Wiley, Oxford : Clarendon Press.
- [15] *Shogi Opening Book*. URL : <https://www.crazy-sensei.com/book/shogi>.
- [16] *The Universal Shogi Interface*. URL : <http://hgm.nubati.net/usи.html>. (consulté : 29.05.2020).