Reinforcement Learning

—

Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm

Cédric Allain Alexis Gerbeaux

6 mars 2020

Table des matières

1 Introduction

Depuis son apparition en Inde vers la fin du VI^{e} siècle, le jeu d'échecs n'a cessé de fasciner les Hommes. Avec l'émergence des algorithmes et en particulier de l'intelligence artificielle, ces derniers ont cherché à développer des machines capables de battre les humains à ce jeu complexe, une volonté qui remonte la fin du XVIII^{e} siècle avec le Turc mécanique de Johann Wolfgang von Kempelen.

La première avancée significative dans ce domaine arrive le 11 mai 1997, lorsque l'ordinateur Deep Blue, développé par la société américaine IBM, bat le champion du monde en titre, le Russe Garry Kasparov. C'est la première fois qu'un programme informatique bat un champion du monde au jeu d'échecs ; l'Homme perd alors ce jour sa supériorité dans ce domaine. Depuis 2004, le programme Open Source Stockfish est considéré comme la référence des algorithmes pour le jeu d'échecs, avec un Elo score bien supérieur aux meilleurs joueurs mondiaux.

Cependant, un nouveau défi voit le jour pour les chercheurs : le jeu de go, également un jeu de plateau, dont la forme actuelle à vu le jour au XV^{e} siècle au Japon. Les techniques d'attaque par force brute (*brute force* en anglais) employées par Deep Blue (l'ordinateur était capable d'évaluer 200 millions de positions par seconde) ne sont d'aucune utilité pour le jeu de go, à cause notamment du plus grand nombre de combinaisons possibles [1]. Ce défi est résolu le 27 mai 2017, lorsque le champion du monde de go, le Chinois Ke Jie, est battu par *AlphaGo*, un programme développé par Google DeepMind qui fait intervenir des techniques d'apprentissage par renforcement (*Reinforcement Learning*, ou RL).

Malgré toutes ces avancées remarquables, un problème subsiste : le caractère spécialisé des méthodes développées. En effet, Deep Blue et Stockfish sont incapables de jouer au jeu de go, tandis qu'à l'inverse, *AlphaGo* ne saurait jouer une partie d'échecs. La problématique est alors d'essayer de développer un algorithme plus général qui serait capable de jouer à différents jeux, en indiquant uniquement à l'algorithme les règles des jeux en question. David Silver et son équipe de Google DeepMind apporte en 2017 une solution à cette problématique avec l'algorithme *AlphaZero* [1]. Nous étudierons l'algorithme décrit dans cet article dans la suite de ce rapport.

2 Résumé de l'article

Dans cette partie, on se propose de résumer l'article de Silver et al., en présentant les principes fondamentaux de l'algorithme *AlphaZero*, ainsi que les résultats des auteurs pour trois types de jeux : jeu d'échecs, jeu de shogi et jeu de go.

Comme évoqué précédemment, la motivation principale des auteurs pour le développement d'*AlphaZero* est d'obtenir un algorithme aussi performant que Deep Blue ou *AlphaGo* et qui est

1. Il est estimé à 10^{600} le nombre de possibilités de jeu pour le go, contre 10^{120} pour les échecs.

également capable de jouer à la fois aux échecs et au jeu de go, mais aussi à n'importe quel jeu, à partir du moment où on lui en donne les règles.

2.1 Des capacités surhumaines

L'algorithme *AlphaGo* décrit dans l'article [1] permet d'atteindre un niveau de jeu surhumain aux jeux de Go, d'échecs et de shogi (échecs japonais), et cela sans utiliser des combinaisons particulières, ou des stratégies sophistiquées spécifiques à certain jeux, le rendant ainsi plus génral.

Des trois jeux étudiés dans l'article, le shogi est plus compliqué d'un point de vue computationnel, principalement parce que la grille de jeu est plus grande et parce que les règles offrent plus de libertés. Le jeu de Go se distingue des deux précédents car ses règles du jeu sont invariantes par translation et possèdent de nombreuses symétries, ce qui en fait un candidat intéressant pour une modélisation par réseau de neurones. Une différence importante entre ces trois jeux est qu'au jeu de Go les états finaux pour un joueur sont binaires (gagné ou perdu), alors qu'aux jeux de shogi et d'échecs il peut y avoir des cas d'égalité.

Réussir à développer un algorithme de jeu général avec des performances équivalentes à celles de Stockfish ou *AlphaGo* aurait été déjà une incroyable avancée, car cela aurait permit de le réutiliser pour de multiples taches diverses à moindre coût. Cependant, il se trouve qu'*AlphaGo Zero* fait preuve de performances bien supérieures. En effet *AlphaZero* rivalise et surpasse les algorithmes spécialisés les plus performants pour les différents jeux (*Stockfish* pour les échecs, *Elmo* pour le shogi et *AlphaGo Zero* pour le Go [2]).

Ainsi, sans adaptation humaine importante, à l'exception des règles du jeu [3], *AlphaZero* peut atteindre un niveau surhumain de performances en moins de 24 heures d'entraînement. Pour se faire, il joue contre lui-même et entraîne, par *reinforcement learning*, un réseau de neurones qui permet, à partir d'un état du plateau, de donner le meilleur coup possible.
Avant l'introduction de *AlphaGo*, ces trois jeux étaient résolus à l'aide d'une recherche alpha-beta optimisée.

Pour se rendre compte des capacités atteintes par *AlphaZero*, on peut rapidement évoquer les résultats obtenus pour le jeu d'échecs [4]. Aux échecs, et comme dans d'autres jeux similaire, on utilise le plus souvent le classement Elo (*Elo score*), qui permet de comparer les joueurs entre eux. Par exemple, à la date de publication du papier en décembre 2017, le meilleur joueur mondial était le Norvégien Magnus Carlsen [5], qui avait alors un Elo score de 2 800. Afin de se donner une idée des capacités de Magnus Carlsen, on peut évoquer son exploit du 6 octobre 205 à Vienne, lorsqu'il a joué

2. *AlphaGo Zero* étant déjà une amélioration d'*AlphaGo* développée par Google DeepMind
3. *cf.* Section 2.3
4. On détaillera par la suite plus en détail les résultats des auteurs, *cf.* Section 2.5
5. Il est toujours le premier joueur mondial en 2020 : `https://ratings.fide.com`

simultanément 5 parties d'échecs les yeux bandés, assis sur une chaise, tournant le dos aux plateaux d'échecs, et a réussi à gagner 3 des 5 parties. Le programme Stockfish a un Elo score de 3 300, et cette différence de 500 points avec Magnus Carlsen signifie que si Stockfish joue contre Magnus Carlsen, l'algorithme gagnera 95 % du temps. La puissance d'*AlphaZero* nous apparaît alors, lorsque l'on sait que ce dernier ne perd jamais contre Stockfish.

2.2 Une adaptation d'un algorithme déjà existant : *AlphaGo Zero*

AlphaZero est une adaptation de l'algorithme *AlphaGo Zero* [2] qui permet de jouer à différents jeux de plateau et pas seulement au jeu de Go. *AlphaZero* diffère de *AlphaGo Zero* notamment sur les points suivants. D'une part *AlphaZero* prend en compte les jeux où le résultat peut être une égalité. Aux échecs par exemple, la partie peut se terminer sur un match nul, alors que pour le jeu de go, la situation finale est binaire, soit l'on perd, soit l'on gagne. D'autre part, *AlphaZero* n'augmente pas son échantillon d'entraînement en se servant des rotations ou réflexions qui sont propres au jeu de Go, ce qui permet à *AlphaGo Zero* d'augmenter les exemples pour l'entraînement de son réseau de neuronnes.

Il existe un point commun entre *AlphaGo Zero* et *AlphaZero*, à savoir l'utilisation de réseaux de neurones profonds avec des algorithmes de *reinforcement learning* pour s'améliorer, simplement à partir de simulations de parties. Le réseau de neurones utilisé prend en entrée la position s du joueur sur la grille (ou, en d'autres termes, l'état du plateau à un moment donné) et prédit notamment en sortie un vecteur p de probabilité, dont les composantes p_a donnent les probabilités d'effectuer les différentes actions a possibles, $p_a = \mathbb{P}(a|s)$, de façon à gagner la partie.

2.3 Connaissances données à l'algorithme

Afin de fonctionner, l'algorithme *AlphaZero* doit avoir connaissance des points suivants.

— Les règles du jeu. Elles sont utilisées pour le MCTS pendant la phase d'expension, pour déterminer la fin d'une partie et le score associé.
— La caractérisation des positions sur le plateau ainsi que des déplacements et des types de pièces.
— Le nombre de coups autorisés.
— Pour les jeux d'échecs et de shogi, un nombre maximum de coups à partir duquel la partie était considérée comme match nul.

2.4 Le fonctionnement : Monte-Carlo tree search et réseau de neurones

Pour comprendre le fonctionnement de l'algorithme *AlphaZero*, considérons la phase d'entraînement pendant laquelle l'algorithme joue contre lui-même en simulant des parties successives.

Rappelons que l'objectif de cette phase d'apprentissage est d'entraîner un réseau de neurones permettant de dire quelle est la meilleure action à faire à partir d'un état du jeu. C'est uniquement ce réseau qui sera utilisé lors des « vraies » parties, contre Stockfish par exemple.

Ainsi, on se place à un état du jeu s_t pendant la phase d'apprentissage. *AlphaZero* remplace la recherche alpha-beta, spécifique au domaine d'application, par l'algorithme Monte Carlo Tree Search (MCTS). *AlphaZero* effectue alors plusieurs itérations successives de MCTS en partant de s_t.

En partant de cet état s_t, l'algorithme sélectionne le chemin le plus avantageux qui l'amène à une feuille en choisissant les branches avec les valeurs $q + u$ les plus élevées. Le terme q représente la valeur moyenne générée par cette action sur tous les passages et le terme u permet de favoriser l'exploration de chemins n'ayant pas encore été beaucoup explorés.
Soit la feuille sélectionnée est une feuille finale et l'issue du jeu peut-être calculée directement, soit on lui crée des feuilles filles qui représentent les états possibles à partir de cette dernière. Une des feuilles filles, représentant l'état s_0, est choisie. Cet état s_0 est ensuite donné au réseau de neurones qui renvoie une valeur v correspondant à l'estimation de l'espérance du gain sachant que le jeu est dans l'état s_0. Cela permet de ne pas évaluer tous les états successifs possibles jusqu'à la fin du jeu, ce qui représenterait trop de calculs et pourrait être très long pour un jeu nécessitant en moyenne beaucoup de coups. Il faut ensuite mettre à jour les valeurs q et u des branches composant le chemin parcouru. Alors q devient $q = \frac{w}{N}$, où w est la somme cumulée des v obtenus pour cette branche et v est l'espérance du gain sachant s_0 et N le nombre de fois que le MCTS a parcouru cette arrête. u est une fonction décroissante de N qui augmente si une action n'a pas été beaucoup explorée en comparaison des autres actions. Pour le calcul de u, un hyper-paramètre multiplicatif, noté c_{param} dans notre cas, permet également d'augmenter le poids de u par rapport à q et ainsi de favoriser davantage l'exploration [6]. On répète alors l'opération à partir du même état de départ s_t, sachant que cette mise à jour des poids $q + u$ amènera peut-être à une autre feuille de l'arbre. Au cours des itération le poids respectif de u par rapport à q va diminuer laissant de moins en moins de place à l'exploration et profitant de l'exploitation.

Une fois cette opération répétée un grand nombre de fois (1 000 par exemple), on obtient un arbre d'états possibles du jeu à partir de l'état s_t, dans lequel le jeu se trouve réellement. C'est à partir des valeurs $N_{s_t}(a)$ des arrêtes du premier étage de l'arbre que l'on définit le vecteur π_t, représentant la probabilité de distribution des actions a possibles en étant dans l'état s_t. On a alors $\pi(a \mid s_t) = \frac{N_{s_t}(a)}{\sum_b N_{s_t}(b)}$ avec $N_{s_t}(a)$ le nombre de fois que l'action a a été choisie à partir de l'état s_t. Pour la phase d'entraînement, un bruit peut également être ajouté à cette distribution. En mettant l'état s_t dans le réseau de neurones, on obtient p_t une estimation de π_t et v_t, l'espérance de gain sachant cet état.

6. On a $u(s_t, a) = c_{param}\mathbb{P}(a|s)\frac{\sqrt{\sum_b N_{s_t}(b)}}{N_{s_t}(a)}$

L'algorithme avance ensuite à l'état suivant s_{t+1} en choisissant une action $a_t \sim \pi_t$ et recommence alors la procédure décrite jusqu'à effectuer une action qui amène le jeu dans un état final. Le gain final du jeu est noté z. Dans le cas où l'issue du jeu est binaire (victoire/défaite), $z \in \{-1, +1\}$, et dans le cas où un match nul est possible, $z \in \{-1, 0, +1\}$

C'est une fois la partie simulée terminée que s'opère la mise à jour des paramètres du réseau de neurones. v_t doit être proche de z, le gain obtenu dans notre état terminal, et p_t doit être proche de π_t pour tout les t correspondants aux états intermédiaires traversés lors du jeu. Pour la mise à jour, la perte utilisée est $l_t = (z - v_t)^2 - \pi_t^T \log p_t + c\|\theta\|^2$, où θ représente les paramètres du réseau.

Le processus est répété autant de fois que les paramètres ont besoin d'être mis à jour. Les auteurs de [1] entraînent le modèle avec 700 000 itérations. Une fois le modèle entraîné, le réseau de neurones peut jouer contre un joueur donné. Pour cela, à partir d'une position de jeu donné, le réseau permet de savoir quelle action choisir à partir du vecteur de probabilité des actions possibles.

Question : Comment pour tout état s_t en entrée du réseau on obtient un vecteur de proba dont la taille dépend du nombre d'état fils possibles après s_t ?

2.5 Résultats des auteurs

Comme évoqué précédemment, pour évaluer et comparer la performance des algorithmes, le classement Elo (*Elo score*) est souvent utilisé. Ce classement consiste à attribuer un nombre de points à tous les joueurs d'un jeu et plus un joueur dispose de points, plus son niveau attendu est élevé. Pour deux joueurs de même niveau, le même nombre de points leur ait attribué. Un joueur gagne des points en gagnant face à un joueur mieux classé que lui, et perd des points s'il subit une défaite face à un joueur moins bien classé que lui.

Les auteurs ont entraîné *AlphaZero* pour les jeux de Go, d'échecs et de shogi séparément mais avec les mêmes hyper-paramètres. Comme nous le montre la Figure 1, *AlphaZero* surpasse les algorithmes spécialisés pour les différents jeux (*Stockfish* pour les échecs, *Elmo* pour le shogi et *AlphaGo Zero* pour le Go) à un moment pendant la phase d'apprentissage comprenant 700 000 itérations (avec des mini-batchs de 4 096).

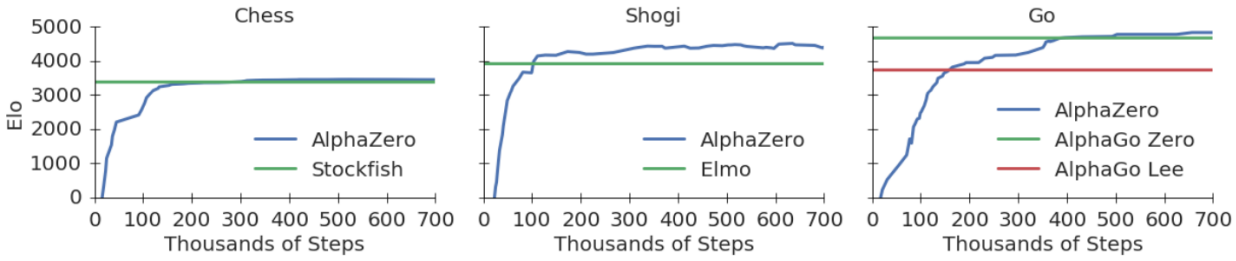

FIGURE 1

Pour le jeu d'échecs, *AlphaZero* surpasse *Stockfish* après 300 000 itérations d'entraînement soit environ 4 heures. Il lui faut 2 heures et 110 000 itérations pour dépasser *Elmo* au jeu de shogi et 8 heures et 165 000 itérations pour être meilleur que *AlphaGo Lee* au Go. Le nombre d'itération par heure montre que l'entraînement au jeu de Go est beaucoup plus long que l'entraînement aux jeux d'échecs et de shogi, comme attendu.

Sur 100 parties jouées entre *AlphaZero* et ses meilleurs adversaires dans les différents jeux, *AlphaZero* a perdu seulement 8 fois au jeu de shogi et n'a jamais été battu aux échecs. D'ailleurs, il est intéressant de voir que parmi les ouvertures les plus connues aux échecs, *AlphaZero* gagne systématiquement contre *Stockfish*. Les performances d'*AlphaZero* sont moins marquées face à *AlphaGo Zero*, avec 62 % de chance de victoire lorsque *AlphaZero* joue avec les blancs comparé aux 58 % de victoires lorsque c'est *AlphaGo Zero* qui joue les blancs.

Sur ces applications, les auteurs comparent également la recherche par MCTS avec la recherche alpha-beta utilisée par *Stockfish* et *Elmo*. La recherche par MCTS est moins rapide puisqu'elle permet à *AlphaZero* d'évaluer 80 000 positions par secondes dans le cas du jeu d'échecs contre 70 millions avec *Stockfish*. Dans le même laps de temps *AlphaZero* pourra évaluer moins de positions que *Stockfish* mais les positions évaluées seront plus prometteuses.

Enfin, avec ses MCTS *AlphaZero* est plus lent que ses adversaires mais il est intéressant de voir qu'il joue déjà bien avec un temps de jeu très réduit et que ses performances augmentent plus vite que celle de ses adversaires avec la durée par coup. Ainsi, avec 0,5 seconde par coup *AlphaZero* est du même niveau que *Stockfish* mais il le dépasse dès que la durée du coup augmente comme on peut le voir dans la figure 2.

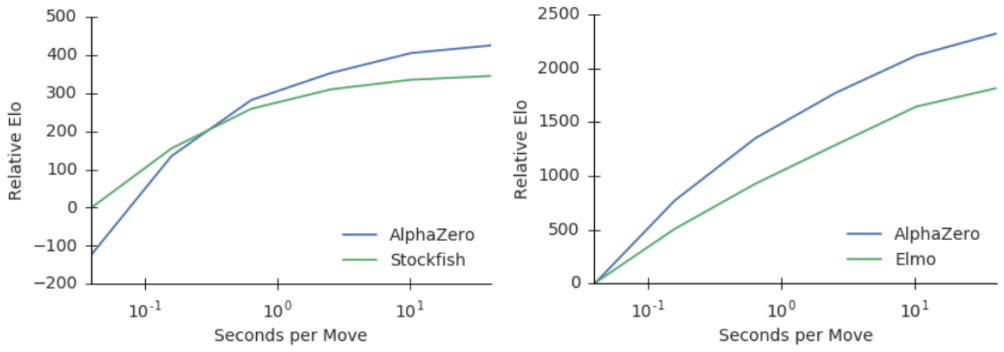

FIGURE 2 – Mesure de performance avec le score Elo en fonction du temps de jeu par coup. A gauche le jeu d'échecs et à droite le jeu de shogi.

3 Jeu du morpion

Nous avons cherché à illustrer ces méthodes et en particulier le MCTS, dans le cadre plus simple du jeu du morpion (*tic-tac-toe* en anglais). Cette simplification est à la fois technique, les règles du jeu étant plus simple à implémenter, et computationnelle, avec un plateau de taille très inférieure à un plateau de jeu de go ou même d'échecs.

La différence principale avec l'article [1] est que nous n'utilisons pas de réseau de neurones. En effet, l'introduction d'un réseau de neurones dans l'algorithme *AlphaZero* permet de pallier le manque de rapidité induit par le MCTS. Il permet ainsi, une fois le réseau entraîné, de ne plus avoir à recourir aux MCTS et il permet également, pendant l'entraînement, de ne pas avoir besoin que les arbres du MCTS aillent jusqu'à un état final du jeu. Le réseau de neurones permet également de ne pas avoir besoin de garder en mémoire les arbres du MCTS.

Dans notre cas, le MCTS est donc utilisé directement pendant les phases de jeu en simulant des parties jusqu'à atteindre un état final du jeu. De ce fait, il n'y a pas de « phase d'apprentissage », seulement des phases de jeu.

3.1 Implémentation du jeu

Nous sommes partis d'une implémentation du jeu du morpion déjà existante (`https://gist.github.com/eaorak/3966315`) que nous avons ensuite adapté à notre convenance, notamment en faisant une classe Python.

Nous avons implémenté différentes méthodes de jeu, de façon à pouvoir faire varier la difficulté. Ainsi, nous avons trois méthodes de jeu différentes : "random", "smart" et "mcts", qui fonctionnent de la façon suivante.

Méthode "random" Cette méthode consiste simplement pour le joueur à jouer sur une case disponible de façon totalement aléatoire.

Méthode "smart" Cette méthode « intermédiaire » consiste pour un joueur à regarder l'évolution du jeu au temps d'après et à jouer en conséquence. Ainsi, si le joueur peut gagner au coup d'après alors il joue ce coup, si c'est l'adversaire qui peut gagner au tour suivant, alors le joueur le bloque. Dans les cas restants, le joueur joue de façon aléatoire, comme dans la méthode précédente.

Méthode "mcts" Cette méthode est la plus évoluée, dans le sens où un joueur jouant avec cette technique va, afin de décider quelle action faire, effectuer un MCTS à partir de la situation dans laquelle il se trouve. Dans cette méthode, deux hyper-paramètres sont importants : le nombre de simu-

lations à effectuer ainsi que la valeur de la constante d'exploration, qui intervient dans le choix de l'action à faire. Par la suite, on précisera toujours les valeurs de ces paramètres. Afin d'implémenter cette méthode, on utilise le package `mctspy` (`https://github.com/int8/monte-carlo-tree-search`), que l'on modifie légèrement afin de pouvoir changer plus facilement les paramètres décrits précédemment.

3.2 Résultats préliminaires

Afin d'évaluer les différences entre ces différentes méthodes, on définit une fonction qui permet de faire jouer automatiquement une partie de morpion à deux joueurs ayant une méthode différente.

Deux joueurs aléatoires Tout d'abord, on propose de faire jouer entre eux, deux joueurs avec la méthode "random". Comme présenté dans la Table 1 (première ligne), lorsque l'on fait jouer deux joueurs aléatoires l'un contre l'autre, celui qui commence à l'avantage sur le second. En effet, pour 1 000 parties effectuées, celui qui joue en premier (*play order* vaut 1) gagne 562 parties, soit plus de 50 % du temps, et perd moins d'une partie sur 3. Ceci est un résultat intuitif car le jeu de morpion ayant 9 cases, le joueur qui joue en premier peut poser un marqueur de plus que celui qui joue en second. Ainsi, lorsqu'on joue de façon aléatoire, il y a plus de chances de gagner si l'on est le premier à poser un marqueur, ce que nous confirment nos résultats.

C'est pour cette raison que dans la suite, lorsque deux joueurs jouent avec des méthodes différentes, on distingue les résultats en fonction de qui a joué en premier.

Le joueur "smart" Lorsque c'est un joueur "smart" qui joue face à un joueur aléatoire, celui-ci gagne plus souvent qu'un autre joueur aléatoire et ce, qu'il soit le premier joueur ou le second, comme le montre la Table 1. On retrouve ici encore un avantage à jouer en premier : un joueur "smart" gagne 90 % du temps s'il joue en premier, alors que s'il joue en second, il ne gagne plus que 68 % du temps. Cependant, lorsque deux joueurs "smart" s'affrontent, la probabilité que la partie se termine en match nul est supérieure à 50 %, comme le montre les résultats de la Table 2 (première ligne), avec de nouveau un avantage pour le joueur qui commence.

Le joueur "mcts", le joueur ultime ? En théorie, on pourrait avancer le fait qu'un joueur qui regarde dans le futur ne peux que gagner (surtout s'il joue en premier), ou du moins, ne peux pas perdre. Dans cette partie, on fixe le nombre de simulations à 500 et la constante d'exploration à 1. Contre un joueur aléatoire, le joueur MCTS gagne la très grande majorité du temps (plus de 90 % des parties), comme le montrent les résultats présentés dans la Table 1, et même ne perd jamais lorsqu'il joue en premier.

Method	Play order	Victory	Draw	Defeat
random	1	562	116	322
smart	1	900	90	10
	2	679	252	69
MCTS	1	980	20	0
	2	914	82	4

TABLE 1 – Performances face à un joueur aléatoire sur 1 000 jeux (pour MCTS : 500 simulations par coup et `c_param`=1).
Lecture : un MCTS entraîné avec 500 simulations par coup qui joue en premier face à un joueur aléatoire gagne 980 parties sur 1 000 et concède 20 parties.

Face à un joueur smart, la nuance est plus marquée. En effet, lorsque le joueur MCTS commence, il gagne 60 % des parties, les matchs nuls représentant un peu moins de 40 %. Cependant, lorsqu'il joue en second, la très grande majorité des parties se terminent en un match nul (plus de 80 %). Encore une fois, cela met en évidence l'avantage donné au premier joueur. À noter tout de même qu'il est possible pour le joueur MCTS de perdre face à un joueur aléatoire ou "smart", alors qu'il a joué en premier [7]. Cela peut être soit dû au pur hasard, soit au nombre de simulations pour le MCTS.

On peut également regarder ce que l'on obtient en faisant joueur deux joueurs MCTS l'un contre l'autre. En théorie, le joueur qui commence ne peut pas perdre, il peut simplement concéder un match nul ou gagner bien évidement. Donc, l'issue d'un match entre deux joueurs avec la meilleur stratégie devrait systématiquement être le match nul. On obtient alors les résultats suivants (pour le joueur ayant commencé la partie) : 82 victoires, 918 matchs nuls et 0 défaites. Ainsi, cela confirme notre intuition, la très grande majorité des parties se terminent en un match nul. Cependant, il reste possible pour le joueur qui commence de gagner la partie.

On peut ainsi se demander si le MCTS pourrait atteindre de meilleures performances si l'on augmenter le nombre de simulations effectuées avant chaque coup. À l'inverse, obtient-on des performances équivalentes si l'on réduit le nombre de simulations, ou si l'on réduit la constante d'exploration ? Cette question est légitime car le jeu du morpion reste un jeu simple, avec que très peu de situations uniques, et qui donc a priori ne nécessite pas une phase d'apprentissage très longue.

3.3 Rôle des hyper-paramètres du MCTS

Pour répondre aux questions précédentes, on se propose dans cette partie de s'intéresser à l'évolution des performances d'un joueur MCTS face à un joueur "smart" lorsque l'on modifie soit le nombre de simulations, soit la valeur de la constante d'exploration.

7. Pour des exemples de parties où c'est le cas, voir le notebook associé au rapport : `https://github.com/CedricAllainEnsae/TicTacToe_MCTS/blob/master/TicTacToe_mcts.ipynb`

Method	Play order	Victory	Draw	Defeat
smart	1	283	524	193
MCTS	1	613	386	1
	2	114	823	33

TABLE 2 – Performances face à un joueur "smart" sur 1 000 jeux (pour MCTS : 500 simulations par coup et `c_param`=1).
Lecture : un MCTS entraîné avec 500 simulations par coup qui joue en premier face à un joueur "smart" gagne 613 parties sur 1 000.

Dans le premier cas, on fait varier le nombre de simulations de 50 à 1 000 en conservant la constante d'exploration à 1. Les résultats sont présentés dans la Table 3 et représentés dans les Figures 3, 4, 5. Dans le second cas, on conserve le nombre de simulations à 500 et on fait varier la constante d'exploration de 0 à 1. Les résultats sont présentés dans la Table 4 et représentés dans les Figures 6, 7, 8.

On observe ainsi que plus le nombre de simulations augmente, plus le nombre de victoires du joueur MCTS lorsqu'il joue en premier augmente. On observe cependant une diminution de ce nombre lorsque le nombre de simulations est de 1 000, ce qui est contre intuitif, et que l'on ne saurait expliquer [8]. On observe également que lorsqu'il joue en second, le joueur MCTS a des performances (en nombre de victoires), systématiquement en deçà de celles d'un joueur "smart". Cependant, en moyenne (c'est-à-dire si le joueur MCTS a une chance sur deux de commencer, dans le cas où le choix du joueur qui commence est fait de façon aléatoire), les performances passent au dessus de celle d'un joueur "smart" pour un nombre assez élevé de simulations.

Lorsque l'on fait varier la valeur de la constante d'exploration, on observe que les performances restent assez stables, laissant croire que la constante d'exploration ne joue qu'un rôle mineure dans notre jeu du morpion. Cela est certainement le cas puisque, comme mentionné précédemment, le jeu du morpion n'as que très peu de situations uniques, surtout après deux ou trois tours de jeu. Ce paramètre a cependant un rôle indéniable dans les jeux tel que les échecs, où le très grand nombre de possibilités d'états à partir d'un état donné rend nécessaire un bon compromis entre exploration et exploitation.

8. Il faudrait probablement refaire les calculs sur un nombre plus grands de parties, mais nous ne disposons pas de la puissance de calcul nécessaire pour effectuer cela en un temps raisonnable.

Method	Play order	Victory	Draw	Defeat
smart	1	283	524	193
MCTS 10	1	270	341	389
	2	87	220	693
MCTS 50	1	482	458	59
	2	137	482	381
MCTS 100	1	534	439	27
	2	175	584	241
MCTS 500	1	613	386	1
	2	114	823	33
MCTS 1000	1	525	475	0
	2	90	899	11

TABLE 3 – Performances face à un joueur "smart" sur 1 000 jeux, en faisant varier le nombre de simulations par coup pour le MCTS (`c_param`=1).
Lecture : un MCTS entraîné avec 500 simulations par coup qui joue en premier face à un joueur "smart" gagne 613 parties sur 1 000.

method	play_order	victory	draw	defeat
MCTS 0	1	578	422	0
	2	119	847	34
MCTS 0.2	1	568	432	0
	2	125	849	26
MCTS 0.4	1	571	429	0
	2	116	845	39
MCTS 0.6	1	595	405	0
	2	131	839	30
MCTS 0.8	1	602	398	0
	2	120	840	40
MCTS 1	1	623	377	0
	2	135	834	31

TABLE 4 – Performances face à un joueur "smart" sur 1 000 jeux, en faisant varier la constante d'exploration `c_param` pour le MCTS (500 simulations par coup).
Lecture : un MCTS entraîné avec 500 simulations par coup avec une constante d'exploration de 0.6 qui joue en premier face à un joueur "smart" gagne 595 parties sur 1 000.

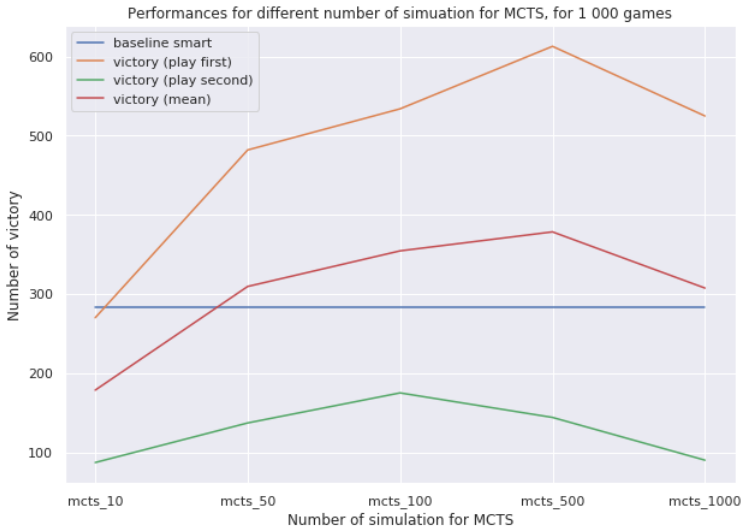

FIGURE 3

FIGURE 4

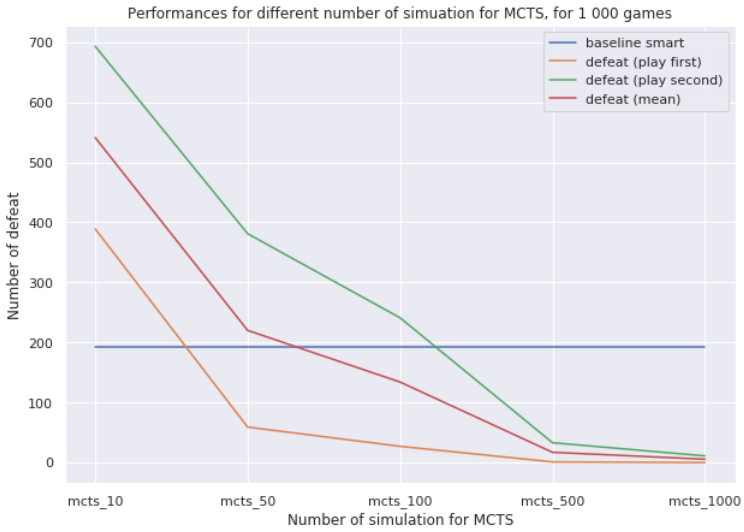

FIGURE 5

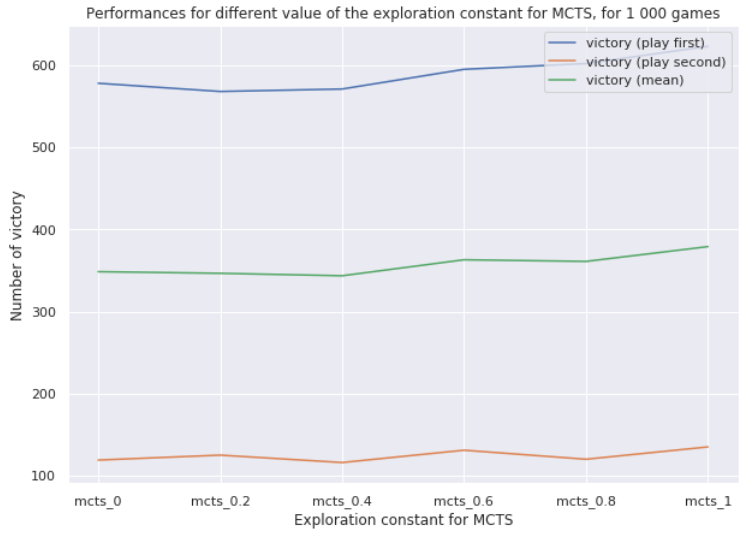

FIGURE 6

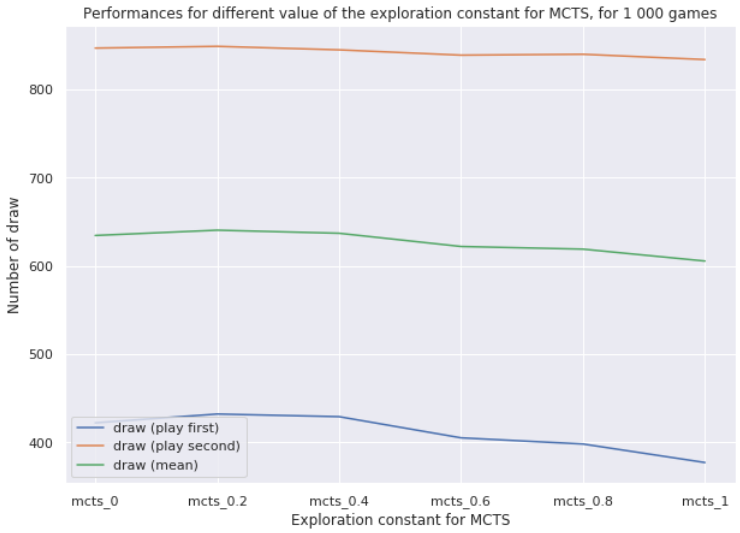

FIGURE 7

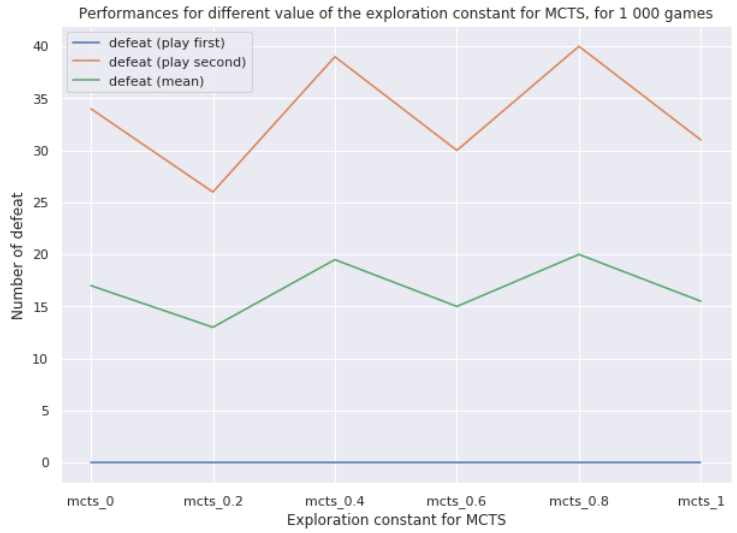

FIGURE 8

Références

[1] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv :1712.01815*, 2017.

[2] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676) :354–359, 2017.