Reinforcement Learning : AlphaZero

Adnan Asadullah Imad Al Moslli

February 2020

# Partie 1 : Présentation de l'article

## 1 Contexte : AlphaZero

Nous avons choisi d'étudier l'article "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play" publié par Google DeepMind en décembre 2018. L'article présente le tout nouvel algorithme développé pour les jeux d'échecs, Shogi et Go : AlphaZero.

Contrairement aux algorithmes classiques développés pour ce genre de jeux, AlphaZero apprend tout de lui-même, on lui donne seulement les règles du jeu. C'était également le cas de l'algorithme AlphaGo Zero mais ce dernier ne jouait qu'au jeu de Go. AlphaZero a cette particularité qu'il n'est pas conçu pour un jeu en particulier.

L'apprentissage est composé de deux élements principaux : un réseau de neurones convolué et un arbre de recherche de type Monte Carlo (MCTS).

## 2 Deep neural network

On considère le réseau de neurones de paramètre $\theta$ suivant :

$$(\mathbf{p}, v) = f_\theta(s)$$

$s$ : position (entrée du réseau de neurones).

$\mathbf{p}$ : vecteur des probabilités des mouvements (sortie du réseau de neurones) composé des propabilités $p_a = P(a|s)$ pour chaque action a.

$v = E[z|s]$ avec $z$ le résultat de la partie ($v$ représente donc le résultat attendu sachant la position) : $z = \begin{cases} 1 & \text{si victoire} \\ 0 & \text{si match nul} \\ -1 & \text{si défaite} \end{cases}$

## 3 Monte Carlo Tree Search (MCTS)

Les algorithmes conçus pour ce type de problèmes sont souvent basés sur un arbre de recherche de type alpha-bêta. AlphaZero utilise lui un algorithme de type MCTS. L'algorithme est toujours basé sur un arbre de recherche :

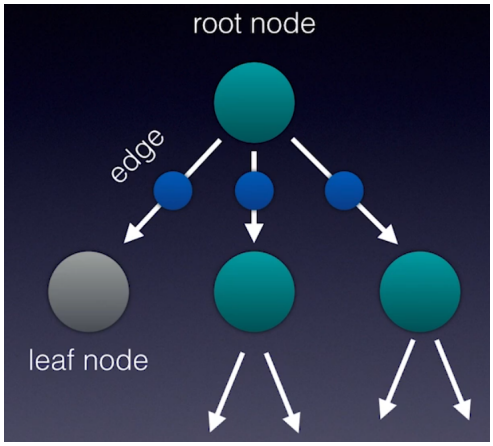


FIGURE 1 – arbre de recherche

Le noeud racine représente la position actuelle du joueur.
Chaque branche représente un mouvement potentiel pour une ceraine position
Chaque feuille représente la position après ce mouvement.

L'algorithme MCTS se décompose en 4 étapes.

### 3.1 Etape 1 : Sélectionner une action

On choisit l'action qui maximise $Q_t(a) + U(s,a)$ avec :

- $Q_t(a) = \frac{r_1+...+r_{ka}}{k_a} = \frac{W}{N}$ la valeur de l'action $a$ définie comme la moyenne des récompenses obtenues quand celle-ci est sélectionnée ;
- $U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$ avec $c_{puct}$ un hyperparamètre qui contrôle la tendance à plus explorer l'arbre, $N(s,a)$ le nombre de fois qu'une action a été explorée et $P(s,a)$ est la probabilité de réaliser une certaine action (retournée par le réseau de neurones).

### 3.2 Etape 2 : Expansion

Cette étape consiste à répéter l'étape 1 jusqu'à atteindre une feuille. On fait alors appel au réseau de neurones pour récupérer le vecteur des probabilités de mouvement $p$ attaché aux actions possibles depuis la nouvelle position ainsi que le résultat attendu $v$ depuis cette dernière.

### 3.3 Etape 3 : Mise à jour des noeuds (Backpropagation)

On met à jour chaque noeud de l'arbre par lequel on est passé :
$N \longleftarrow N + 1$
$W \longleftarrow W + v$
$Q \longleftarrow W/N$

### 3.4 Etape 4 : Simulation et sélection

On répète les étapes 2 et 3 un certain nombre de fois (correspond au nombre de simulations). Ensuite, on choisit l'action à réaliser en fonction du résultat des simulations : il s'agit tout simplement de choisir l'action qui a été la plus explorée, c'est-à-dire qui a la valeur $N(s,a)$ la plus grande.

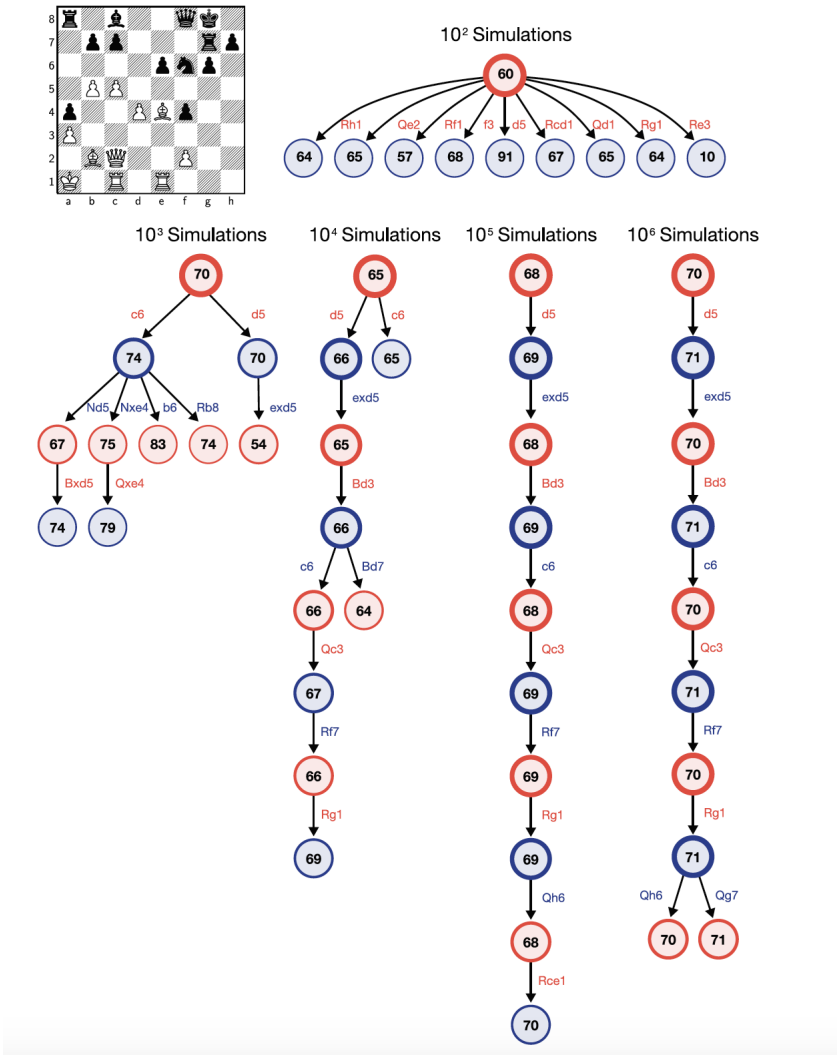


FIGURE 2 – Arbre de recherche dans une partie entre AlphaZero (blanc) et Stockfish (noir)

On peut voir ci-dessus un exemple sur une partie d'échec entre Alphazero et le programme d'échecs Stockfish. On commence par décrire toutes les possibilités de mouvement (position possible sur l'échiquier pour chaque pièce). Ensuite, on voit le nombre de fois qu'un noeud a été exploré pour différents nombres de simulations (réduit entre 0 et 100). Finalement, à ce moment là, Alphazero a choisi de bouger son pion à la position d5 de l'échiquier.

# Partie 2 : Expérimentations numériques

## 4 Introduction

Nous avons choisi d'exploiter la particularité de l'algorithme Alphazero qui peut être utilisé pour différents types de jeu et être performants pour des jeux qui demandent (un minimum de) réflexion, c'est-à-dire dont le résultat n'est pas aléatoire. Pour ce faire, nous allons nous concentrer sur un aspect (important) de l'algorithme qui est le MCTS et l'appliquer pour un jeu très simple qui est le jeu du morpion.

## 5 Implémentation

Pour implémenter un MCTS sur le jeu du morpion, nous nous sommes appuyé sur plusieurs librairies :

- Nous avons avons commencé par récupérer une implémentation basique du jeu du morpion en python (que nous avons nettoyé), disponible à cette adresse :
`https://python.jpvweb.com/python/mesrecettespython/doku.php?id=morpion_console` ;

- Ensuite, nous avons ajouté et adapté un algorithme MCTS à notre jeu, en utilisant la librairie mcts.py, que l'on peut retrouver ici :
`https://github.com/int8/monte-carlo-tree-search/blob/master/mctspy/tree/nodes.py` ;

- Enfin, nous avons ajouté un algorithme de type alpha beta et nous avons fait varier les paramètres pour comparer les performances. Nous nous sommes appuyés sur une implémentation en python, que nous avons adapté pour notre cas d'usage :
`https://stackabuse.com/minimax-and-alpha-beta-pruning-in-python/`.

Notre code est disponible sur Github : `https://github.com/imaddevinci/reenforcement_learning`

## 6 Exemple de partie

Commençons par regarder un exemple de partie entre l'algorithme AlphaBeta pour le morpion et l'algorithme Monte Carlo avec un nombre de simulations égale à 400 et un paramètre $c_{puct} = 1.4$. Evidemment, le MCTS l'emporte (ce n'est pas toujours le cas comme on va le voir plus tard). On l'appellera ici AlphaZero même si on a bien conscience qu'il s'agit d'un abus de langage puisqu'il s'agit seulement de sa composante MCTS en action.

On peut comparer rapidement son comportement avec celui de l'algorithme AlphaBeta. On rappelle qu'il faut aligner trois croix (pour AlphaZero) et trois ronds (pour son adversaire) pour remporter la partie. Alphabeta commence à jouer en placant son rond sur le bord gauche. On remarque qu'AlphaZero, lui, place tout de suite sa première croix au milieu, ce qui lui donne le maximum de possibilités de combinaisons possibles pour l'emporter ensuite.

Ensuite, AlphaBeta tente d'aligner trois ronds sur la ligne du haut, ce qu'AlphaZero lui empêche de réaliser. Par contre, ensuite, AlphaBeta n'empêche pas AlphaZero d'aligner trois croix sur la diagonale, ce qui permet à ce dernier de l'emporter.

```
Type de joueurs:
[1]: Alpha beta vs Alpha zero
[2]: Oridnateur vs Alpha zero
Quel choix voulez-vous? [2 par defaut]: 1
=====> le joueur1 a le pion 'O', et l'autre le pion 'X'
[' ', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
=====> début du tour 1
joueur1 joue case:  [0, 0]
['O', ' ', ' ']
[' ', ' ', ' ']
[' ', ' ', ' ']
joueur2 joue case:  [1 1]
['O', ' ', ' ']
[' ', 'X', ' ']
[' ', ' ', ' ']
=====> début du tour 2
joueur1 joue case:  [0, 1]
['O', 'O', ' ']
[' ', 'X', ' ']
[' ', ' ', ' ']
joueur2 joue case:  [0 2]
['O', 'O', 'X']
[' ', 'X', ' ']
[' ', ' ', ' ']
=====> début du tour 3
joueur1 joue case:  [1, 0]
['O', 'O', 'X']
['O', 'X', ' ']
[' ', ' ', ' ']
joueur2 joue case:  [2 0]
['O', 'O', 'X']
['O', 'X', ' ']
['X', ' ', ' ']
le gagnant est: Alpha Zero ('X')
fin du jeu
A bientôt pour un prochain jeu!
```

FIGURE 3 – Partie AlphaZero (4000 simulations, $c_{puct}$=1.4) vs programme basique

# 7 Résultats

Dans cette section, nous allons présenter les résultats après avoir fait jouer AlphaZero un certain nombre de fois, d'abord contre le programme basique puis contre l'algorithme AlphaBeta. Nous allons également étudier l'influence du nombre de simulations et du paramètre $c_{puct}$ sur les performances de AlphaZero.

## 7.1 AlphaZero vs Programme basique

Nous avons vu un exemple de partie entre la méthode MCTS et un programme de morpion classique. Cependant, cette partie n'est pas forcément représentative du rapport de force entre les deux. Pour avoir une idée plus précise de celui-ci, nous allons réaliser plusieurs parties.

### 7.1.1 Résultats en fonction du nombre de simulations

Il est intéressant d'étudier les performances de l'algorithme MCTS en fonction du nombre de simulations pour apercevoir à quel point il s'agit du coeur de la méthode, qui permet à celle-ci de donner des bons résultats. Pour cette expérience, nous avons lancé 10 parties avec différents nombres de simulations : 100, 500, 1000, 3000 et 5000. Le joueur qui commence la partie est choisi aléatoirement à chaque partie (une chance sur deux de commencer).

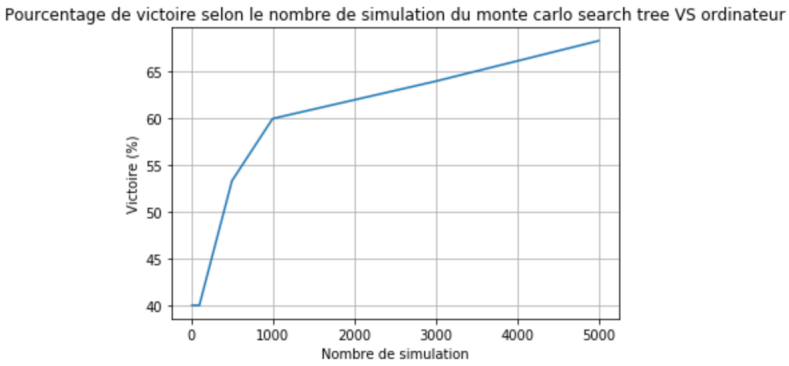


FIGURE 4

On peut voir sur la figure 4 que plus le nombre de simulations est élevé, meilleur sont les résultats de l'algorithme. On peut remarquer qu'entre 10 et 1000, les performances augmentent très fortement (on passe de 40% à 60% de victoire) mais qu'à partir de 1000, l'amélioration est plus lente (on gagne 10% entre 1000 et 5000 simulations).

On peut également voir que malgré un faible nombre de simulations (10), le taux de victoire est déjà de 40%, ce qui n'est pas mauvais sachant que le reste peut être un match nul ou une défaite.

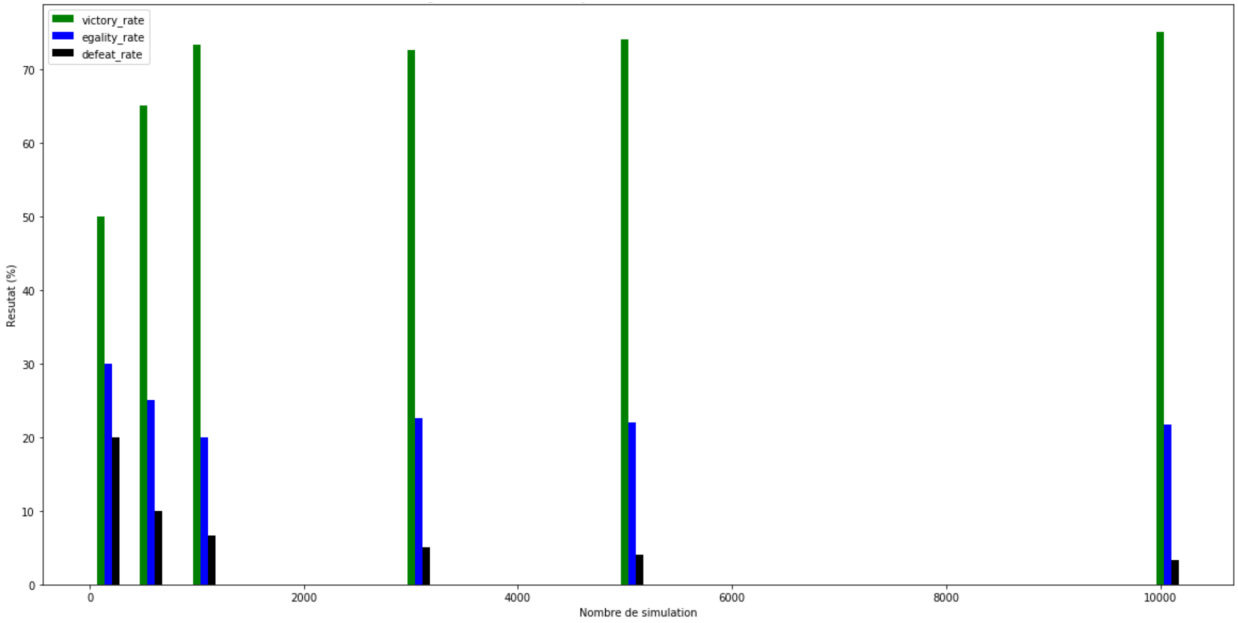


FIGURE 5 – Répartition entre victoire, match nul et défaite pour 10 parties de AlphaZero face à l'ordinateur

Sur la figure 5, on a relancé les parties en comptant cette fois le nombre de victoires, de match nul et de défaite. Les résultats peuvent différer (ici, ils sont un peu meilleur que précédemment) mais la tendance reste la même. On peut voir qu'en dehors des victoires de AlphaZero qui sont majoritaires viennent les matchs nuls et qu'au final, AlphaZero perd très peu de parties.

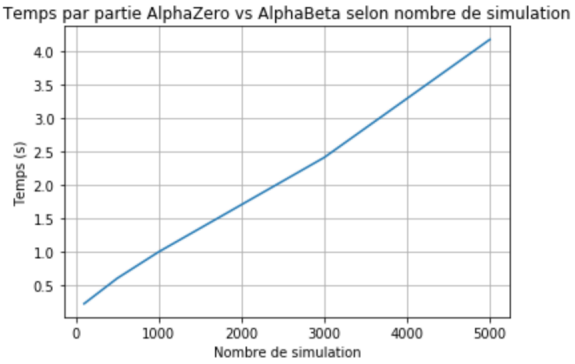


FIGURE 6

Comme on pouvait s'y attendre toutefois, plus on augmente le nombre de simulation, plus l'algorithme met du temps à agir. On peut voir sur la figure 6 l'évolution du temps moyen d'une partie en fonction du nombre de simulations. On discerne assez aisément une relation linéaire entre les deux.

Bien que le temps d'une partie reste peu important dans ce petit jeu du morpion, il risque d'être un problème pour des problèmes plus complexes. Ainsi, il faut avoir conscience que de ce paramètre dépend très fortement la performance du MCTS et qu'il faut le choisir de manière à améliorer autant que possible les résultats sans faire exploser le temps de calcul. On pourrait imaginer dans notre cas choisir un nombre de simulations égale à 1000 puisque cela suffit à avoir de bonnes performances.

### 7.1.2 Résultats en fonction de $c_{puct}$

Comme expliqué dans la première partie, il s'agit d'un paramètre qui permet de faire varier le niveau d'exploration de l'arbre de recherche. En effet, on rappelle que l'on choisit l'action qui maximise $Q_t(a) + U(s,a)$ avec $U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$.

Le terme $U(s,a)$ donne plus de poids aux noeuds qui n'ont pas beaucoup été exploré. Ainsi, le paramètre $c_{puct}$, égale à 1 par défaut, permet de contrôler cette tendance pour lui donner plus ou moins d'importance par rapport à la valeur intrinsèque de l'action.

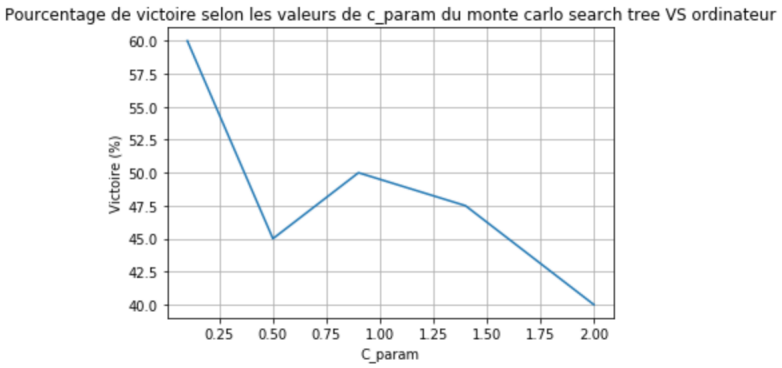


FIGURE 7

Sur la figure 7, on retrouve le pourcentage de victoire en fonction de ce paramètre (pour un nombre de simulations égal à 10). On peut voir qu'il y a plutôt une tendance à la baisse du pourcentage de victoire lorsque l'on augmente la valeur du paramètre. De manière assez surprenante, on peut voir que le taux de victoire le plus important est obtenu pour un paramètre proche de 0,

ce qui signifie qu'une politique qui cherche à favoriser plus d'exploration s'avère néfaste pour les performances de l'algorithme.

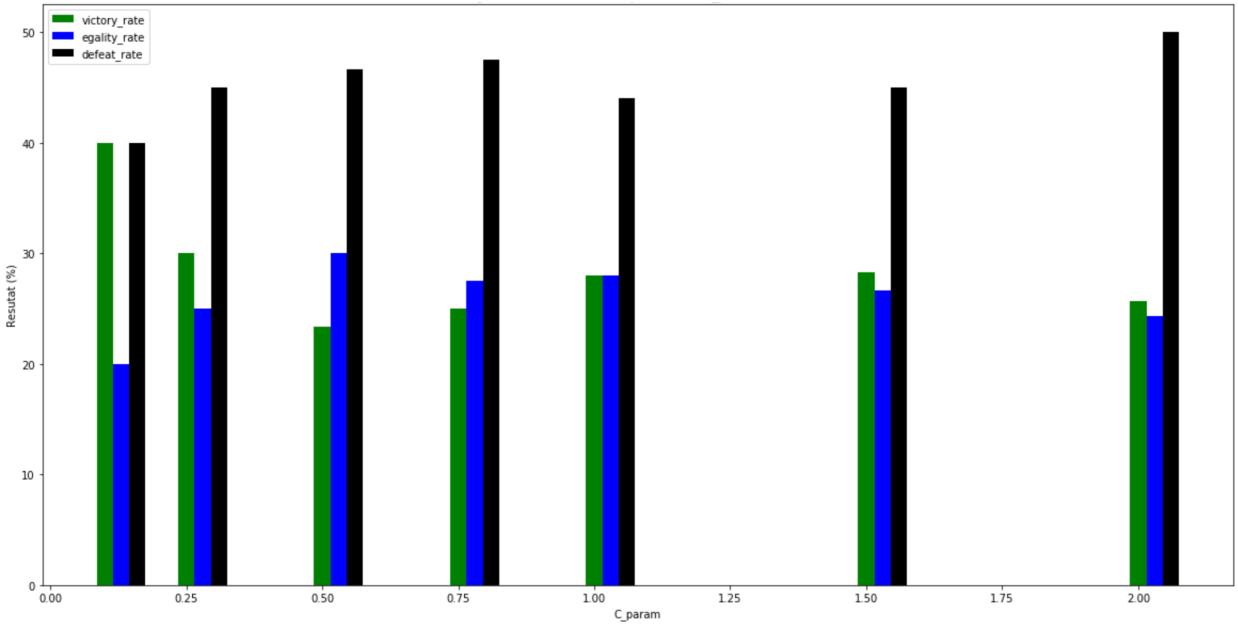


FIGURE 8 – Répartition entre victoire, match nul et défaite pour 10 parties de AlphaZero face à l'ordinateur en fonction du $c_{param}$

On peut voir sur la figure 8 la répartition victoire/match nul/défaite qui montre que le taux de victoire, au début égal au pourcentage de défaite, devient moins important tandis que le taux de défaite augmente lorsque l'on augmente la valeur du paramètre.

Cela peut s'expliquer par le fait que le morpion est un jeu très simple, dans lequel il n'y pas beaucoup de possibilités d'action, et encore moins lorsque l'on avance dans la partie. Ainsi, encourager l'exploration n'est pas forcément utile et a davantage pour effet d'ajouter du bruit à la fonction valeur et ainsi mener vers des mauvais choix d'action.

## 7.2 AlphaZero vs AlphaBeta

AlphaBeta est une adaptation de l'algorithme MiniMax. Cet algorithme cherche à réaliser une action qui va maximiser la récompense du joueur tout en minimisant celle de l'adversaire.

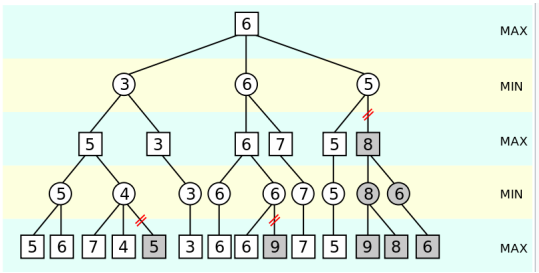


FIGURE 9

On part de la situation courante du jeu $s$, représentée par le noeud racine. Les premières branches représentent les différentes actions possibles : on va chosir l'action qui maximise le gain. Ensuite, à partir de cette action, on représente les différentes actions que peut réaliser l'adversaire : on va chercher à minimiser son gain.

Jusque là, il s'agit simplement d'un algorithme de type Minimax. Alphabeta introduit les quantités $\alpha$ et $\beta$ qui correspondent respectivement à la valeur minimum courante du gain à maximiser et à la valeur maximum du gain à minimiser. Si $\beta < \alpha$, nul besoin de continuer à regardet les noueuds descendants.

Par exemple, sur l'exemple ci-dessus (figure 9), regardons le côté gauche de l'arbre. Au dernier niveau de minimisation, lorsque l'on a vu qu'un des noeuds donnait une récompense de 4 ($\beta$), on savait que le maximum serait le noeud de gauche qui donne la valeur 5 ($\alpha$) donc il n'est pas nécessaire de regarder le noeud restant, puisque cela n'affectera nullement le résultat final.

### 7.2.1 Résultats en fonction du nombre de simulations

On peut voir sur la figure 10 l'évolution de la performance de l'algorithme AlphaZero face à AlphaBeta en fonction du nombre de simulations. Tout comme face au programme informatique, il y a une très forte progression entre 10 et 1000 itérations. Il y a ensuite un progrès relatif, voire une stagnation.

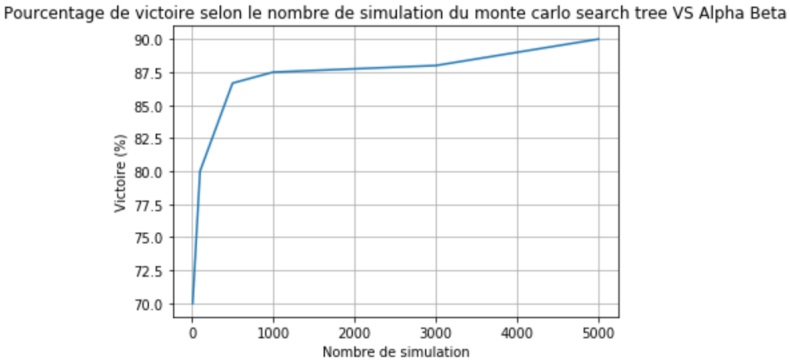


FIGURE 10

Toutefois, AlphaZero s'en sort mieux face à AlphaBeta que face au programme informatique. En effet, on commence dès le début avec un taux de victoire de 70% et on arrive à atteindre jusqu'à 90% de victoire.

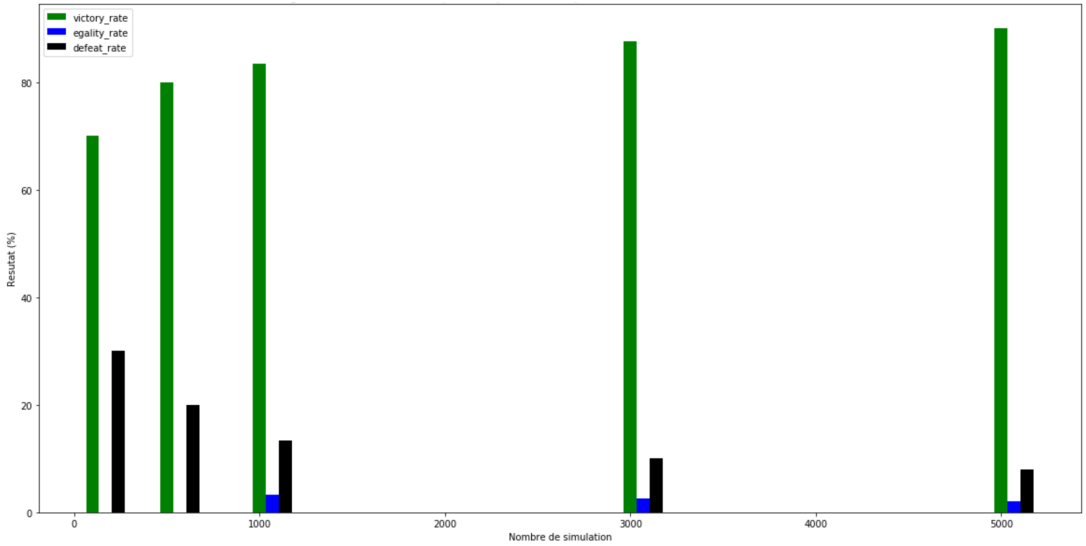


FIGURE 11 – Répartition entre victoire, match nul et défaite pour 10 parties de AlphaZero face à AlphaBeta en fonction du nombre de simulations

Par contre, lorsque l'on regarde la répartition entre victoires, défaites et match nul, on se rend compte que lorsque AlphaZero ne gagne pas, il perd la plupart du temps (très peu de matchs nuls).

Encore une fois, le jeu du morpion est très simple, ce qui fait qu'un programme qui décide de l'action à réaliser en fonction de règles définies à l'avance peut suffire à être très performant. Dans notre cas, on remarque qu'AlphaZero a plus de mal face au programme informatique que face à AlphaZero.

### 7.2.2 Résultats en fonction de $c_{puct}$

Face à AlphaBeta, on retrouve également la même tendance en ce qui concerne le paramètre $c_{puct}$, qui donne des meilleurs résultats lorsqu'il est proche de 0, même si les résultats restent assez bons jusqu'à 1. Ensuite, ils se dégradent assez fortement pour atteindre 60% de victoire pour $c_{puct} = 2$ (voir figure 12).

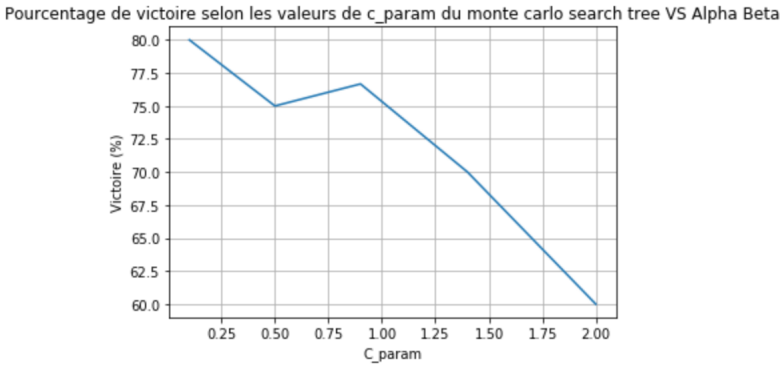


FIGURE 12

Si on regarde la répartition victoire/match nul/défaite (figure 13), on peut voir qu'il est d'autant plus important ici de ne pas trop augmenter la valeur du paramètre car il n'y a pas de match nul donc quand le pourcentage de victoire baisse, le pourcentage de défaite augmente d'autant plus.

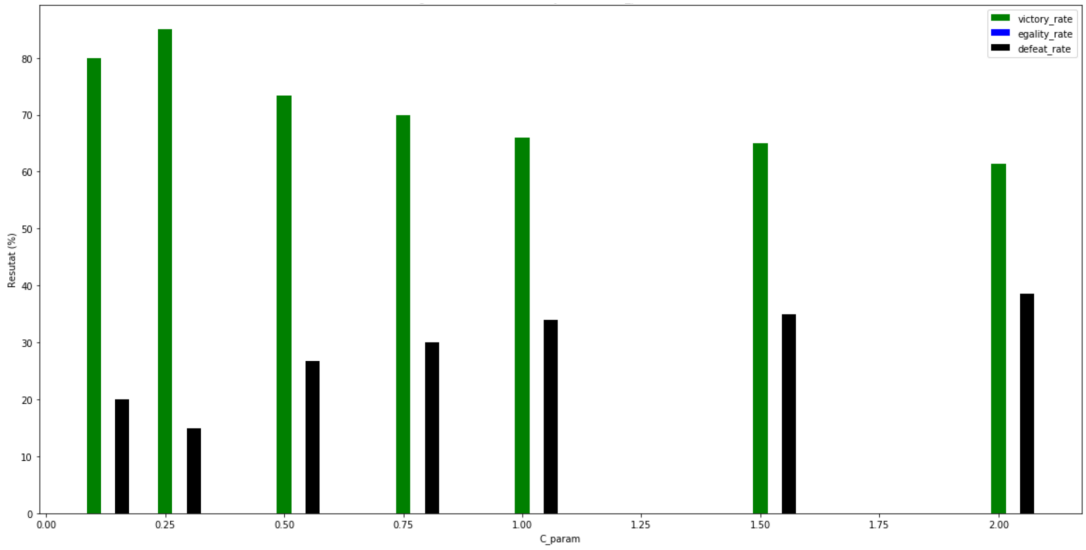


FIGURE 13 – Répartition entre victoire, match nul et défaite pour 10 parties de AlphaZero face à AlphaBeta en fonction du $c_{param}$

# 8 Spécifité du morpion : qui joue en premier ?

Quand on analyse les performances d'un algorithme dans un jeu particulier, il faut toujours prendre en compte les spécificités de ce dernier. Pour le morpion, on sait que le joueur qui commence en premier est avantagé, mais aussi que l'on peut toujours forcer un match nul. Il est donc intéressant de voir comment notre algorithme s'en sort lorsqu'il ne commence pas en premier.

## 8.1 Face à l'ordinateur

### 8.1.1 Lorsque AlphaZero commence

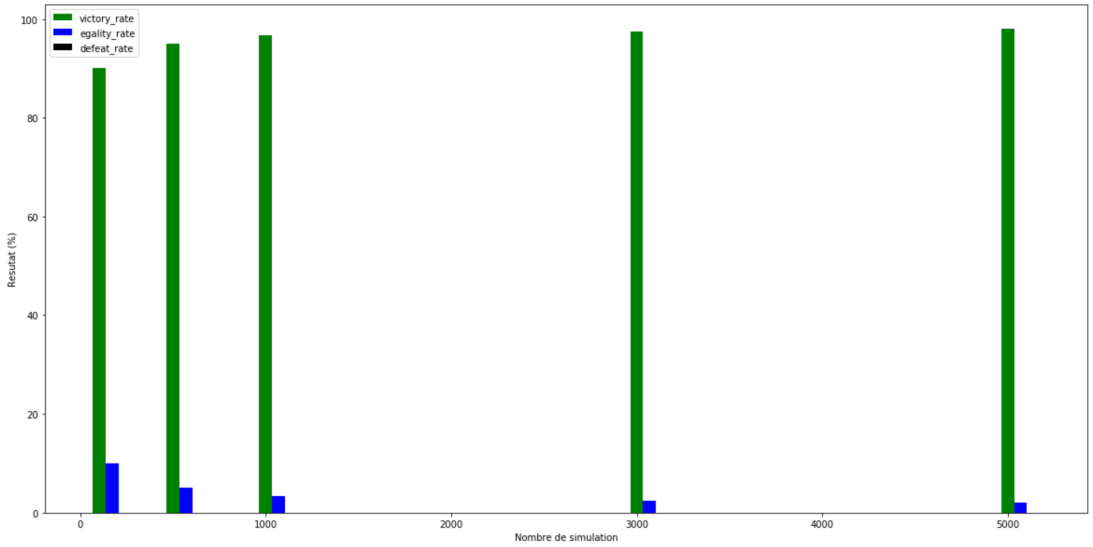


FIGURE 14 – Répartition entre victoire, match nul et défaite pour 10 parties de AlphaZero face à l'ordinateur (AlphaZero joue en premier)

Lorsque Alphazero commence face à l'ordinateur, on remarque tout de suite une chose : il ne perd jamais. Mieux encore, il gagne la plupart du temps et on s'approche même des 100% de victoire lorsque l'on augmente le nombre de simulations.

### 8.1.2 Lorsque l'ordinateur commence

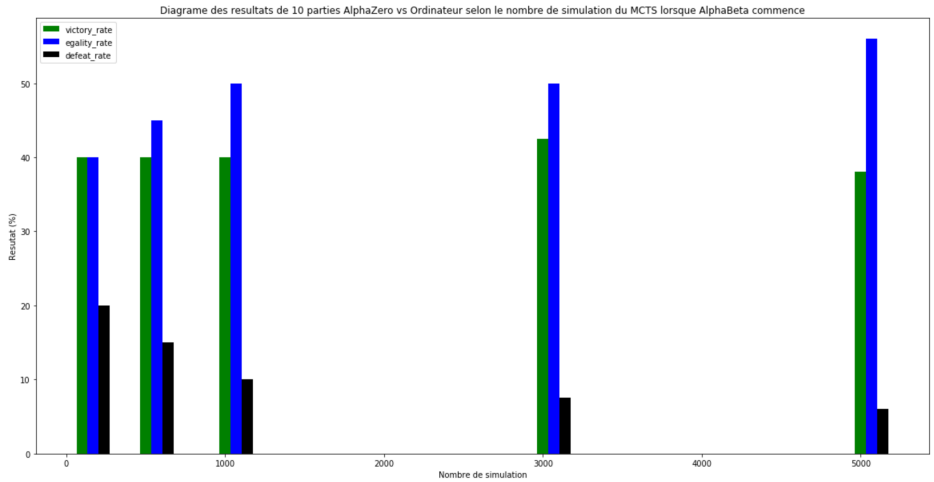


FIGURE 15 – Répartition entre victoire, match nul et défaite pour 10 parties de AlphaZero face à l'ordinateur (l'ordinateur joue en premier)

Lorsque l'ordinateur commence, augmenter le nombre de simulations ne permet pas à AlphaZero d'augmenter son pourcentage de victoire (qui stagne à 40%) mais il arrive à arracher davantage de match nul. Avec un nombre de simulations de 5000, AlphaZero ne perd que très peu face à son opposant.

## 8.2 Face à l'AlphaBeta

### 8.2.1 Lorsque AlphaZero commence

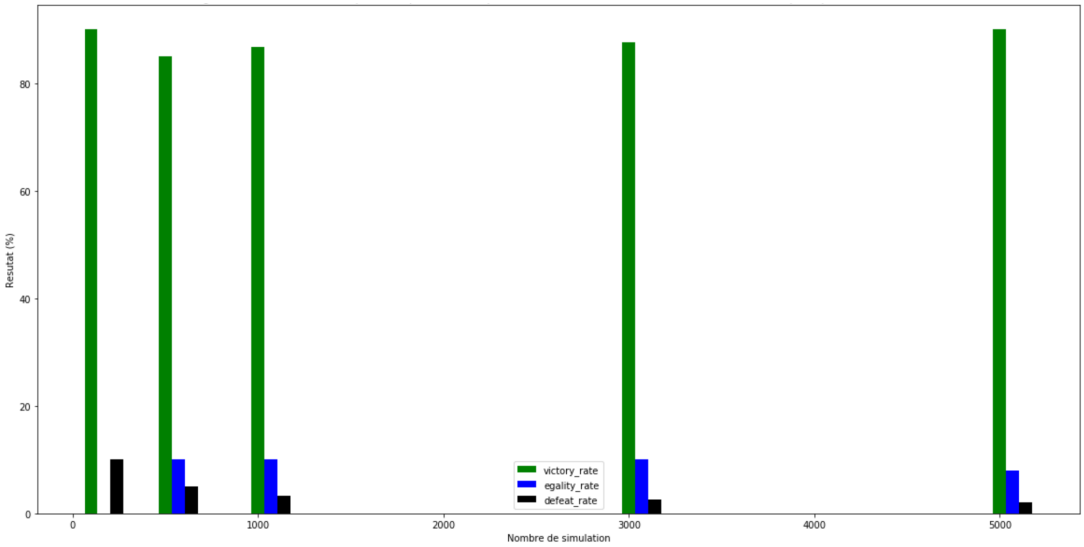


FIGURE 16 – Répartition entre victoire, match nul et défaite pour 10 parties de AlphaZero face à AlphaBeta (AlphaZero joue en premier)

Lorsque AlphaZero commence en premier face à AlphaBeta, le nombre de simulations n'a plus beaucoup d'influence : le pourcentage de victoire est toujours très élevé (autour de 80%). A noter qu'AlphaBeta arrive à décrocher une ou deux victoires sur dix à chaque fois.

### 8.2.2 Lorsque l'AlphaBeta commence

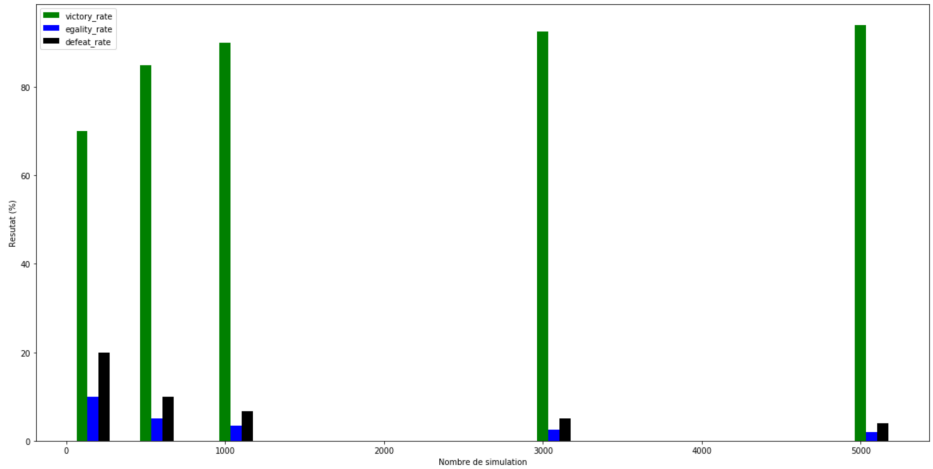


FIGURE 17 – Répartition entre victoire, match nul et défaite pour 10 parties de AlphaZero face à AlphaBeta (AlphaBeta joue en premier)

Lorsque AlphaBeta commence, il n'y a pas beaucoup de différence par rapport à lorsque AlphaZero commence au niveau des résultats. Seulement, augmenter le nombre de simulations permet

d'amméliorer un peu les performances, en passant d'un pourcentage de victoire légèrement au-dessous de 80% à un taux de victoire similaire à lorsque AlphaZero débute (environ 85%).

# 9 Conclusion

Le travail que l'on a réalisé ici avait pour but d'illustrer l'algorithme AlphaZero à travers une de ces composantes principale qu'est le MCTS. Pour cela, nous avons appliqué la méthode dans un cas pratique assez simple qu'est le jeu du morpion. Nous nous sommes penché sur la paramétrisation de la méthode en étudiant l'évolution des résultats pour différents paramètres : le nombre de simulations et le coefficient d'exploration. Enfin, nous l'avons comparé à une autre méthode plus classique et fortement utilisée pour ce genre de problème : l'algorithme AlphaBeta.

Les résultats que nous avons obtenu mettent en évidence les très bonnes performances du MCTS et l'importance des paramètres utilisés. Augmenter le nombre de simulations permet d'améliorer les performances mais augmente en même temps le temps de calcul : il faut donc trouver un équilibre entre les deux.

On a également pu constater que malgré le fait qu'AlphaZero n'ait pas été conçu pour un problème spécifique, l'utiliser pour un nouveau problème nécessite d'adapter les paramètres. En effet, pour le problème du morpion, le terme d'exploration dans la fonction objective à maximiser n'était pas adapté puisqu'il dégradait les performances de l'algorithme. Il faut également prendre en compte les spécifités de chaque jeu pour évaluer les performances. Pour le morpion, celui qui commence à jouer a un gros avantage.

On a également pu voir que pour un problème aussi simple où le nombre de combinaison est limité, un algorithme basé sur des règles de décision pouvait être difficile à battre. Malgré cela, lorsqu'il est correctement paramétrisé, le MCTS parvient à largement surpasser ce dernier, ce qui prouve sa capacité d'adaptation à différents types de problème.

Malgré notre abus de language répété consistant à assimiler l'algorithme que l'on a utilisé à AlphaZero, rappelons encore une fois qu'il ne s'agit que de la composante MCTS. Derrière AlphaZero, il y a aussi et surtout un réseau de neurones qui est entrainé par des parties dans lesquelles AlphaZero joue contre lui-même. Ainsi, durant cette phase d'entraînement, le MCTS est également encore mis à contribution puisque ces parties virtuelles sont jouées grâce à lui.

Le réseau de neurones est entrainé pour minimiser la différence entre le résultat qu'il a prédit $v_t$ et le résultat final $z$. De plus, durant la phase d'entraînement, les actions à chaque étape sont choisies de manière stochastique en tirant une action à partir de la distribution de probabilité du mouvement définie comme : $\pi = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$ où $\tau$ est appelé le paramètre de température qui contrôle la tendance à explorer.

Durant cette phase d'entrainement, on va davantage chercher à pousser l'exploration puisque l'on souhaite que l'algorithme essaye beaucoup plus de combinaisons pour apprendre le plus possible de l'expérience. Dans notre expérience, le réseau de neurones n'était pas vraiment nécessaire tout comme l'exploration mais pour des problèmes plus complexes, cela permet vraiment à l'algorithme de s'adapter à la complexité. C'est ce qui lui a permis d'obtenir les meilleurs résultats dans les jeux d'échecs, de Go et de Shogi.