



Reinforcement Learning

Paper Analysis :

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

Github link :

<https://hrialan.github.io/m2ds-reinforcement-learning/>

Master 2 - Data Science

Alexandre Perbet - Cyril Nérin - Hugo Rialan

Février 2022

Version 1.0

Table des Matières

Introduction	3
I - Historique et introduction à AlphaZero	4
I.1 - L'IA appliquée aux jeux d'échecs et de GO	4
I.1.a - Rappel des règles	4
I.1.b - Complexité des jeux pour un ordinateur	5
I.1.c - Algorithme minimax	6
I.1.d - Historique et résultats obtenus jusqu'à aujourd'hui	6
I.2 - AlphaGO Zero	8
I.2.a - Historique de l'algorithme	8
I.2.b - Principe de fonctionnement	8
I.2.c - Résultats	10
I.3 - AlphaZero	11
I.3.a - Historique de l'algorithme	11
I.3.b - Résultats et performance d'AlphaZero comparées aux autres logiciels de jeu d'échec ou de go	11
II Description du logiciel AlphaZero	12
II-1 Rappel des principes du Deep Reinforcement Learning	12
II-2 Les principaux composants du logiciel AlphaZero	14
II-3 Sélection des déplacements à explorer avec la recherche arborescente de Monte Carlo (MCTS)	14
II-4 Le réseau de neurones profond d'AlphaZero	18
III Implémentation d'une version simplifiée d'AlphaZero et application au jeu Puissance 4	21
Conclusion	22
Sources	23

Introduction

Nous avons étudié l'article « Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm » [1] écrit en 2017 par des chercheurs de DeepMind.

Cet article présente un algorithme d'apprentissage par renforcement, AlphaZero appliqué aux jeux d'échecs, de go et de Shogi. Il s'agit d'une généralisation d'un précédent logiciel (AlphaGo Zero) qui avait été conçu pour apprendre à jouer uniquement au jeu de go. Alors que les précédentes versions de l'algorithme utilisaient des tables d'ouverture et de fin de partie établies par des joueurs experts pour fournir au programme d'excellents coups de départ et de fin, AlphaZero ne connaît que les règles du jeu. Il apprend en jouant contre lui-même de nombreuses parties et atteint un niveau très supérieur au niveau humain. Il a également battu tous les programmes champions du monde dans chaque catégorie : shogi, go et échecs.

Il s'agit d'une avancée majeure dans le domaine de l'Intelligence Artificielle : AlphaZero remplace toutes les connaissances a priori transmises par des experts du domaine, par des réseaux neuronaux profonds et un algorithme d'apprentissage par renforcement de type « table rase » (c'est-à-dire sans informations préalables).

Notre rapport se décompose en quatre parties :

- Rappel historique de l'Intelligence Artificielle appliquée aux jeux d'échecs et de go jusqu'à l'arrivée d'AlphaZero ;
- Description des deux principaux composants d'AlphaZero : l'Arbre de Recherche de Monte Carlo et le réseau de neurones profond ;
- Comparaison des performances d'AlphaZero avec celles des autres logiciels de jeu d'échecs ou de go.
- Description de notre logiciel.

I - Historique et introduction à AlphaZero

I.1 - L'IA appliquée aux jeux d'échecs et de GO

I.1.a - Rappel des règles

Les jeux d'échecs et de go ont toujours été très étudiés dans le domaine de l'intelligence artificielle. Ce sont deux jeux assez différents mais leur complexité les lie souvent lorsqu'il s'agit d'appliquer des algorithmes d'intelligence artificielle pour rechercher des stratégies gagnantes.

Rappel des règles:

Le jeu d'échecs est un jeu de société opposant deux joueurs. Le plateau de jeu, ou l'échiquier, est composé de 64 cases, 32 cases blanches et 32 cases noires. Les joueurs jouent à tour de rôle en déplaçant leur pièce suivant des mouvements autorisés. Le but du jeu est "l'échec et mat", c'est à dire qu'un joueur fait tomber le roi de l'autre joueur sans que celui ne puisse rien faire. C'est un jeu très ancien, il a été introduit dans le sud de l'Europe à partir du X^e siècle. On ignore quand et où il a été inventé exactement. Malgré son ancienneté, c'est un des jeux de réflexion les plus populaires dans le monde entier.

Le jeu de go ou simplement go est un jeu de société originaire de Chine. Il oppose également deux adversaires qui déplacent à tour de rôle non pas des pions mais des pierres. Le but de ce jeu est de construire des territoires et d'encercler les pierres de l'autre joueur. Chaque pierre encerclée devient un "prisonnier". Le joueur qui gagne est celui qui a totalisé le plus de territoires et de prisonniers. Comme l'échec, les premières utilisations sous sa forme actuelle ont vu le jour vers le XV^e siècle. Ce jeu est moins populaire en France que les échecs. Son nom est surtout connu pour l'utilisation du jeu avec des algorithmes d'intelligence artificielle.



exemple de jeu de GO

I.1.b - Complexité des jeux pour un ordinateur

Les deux sont des jeux de plateau, les règles ne sont pas difficiles à comprendre. Cependant, le nombre de stratégies de jeu possibles est si grand qu'il est nécessaire de faire preuve d'expérience, d'analyse et d'intelligence pour gagner une partie. C'est toute la complexité d'essayer de faire jouer un ordinateur à la place d'un humain. Il n'est pas possible de créer un algorithme qui calcule toutes les stratégies possibles avant de jouer car il en existe beaucoup trop. En 2021, un programmeur a estimé avec précision que le nombre de positions de pièces légales aux échecs était d'environ 10^{44} [2]. Le plateau de Go est plus grand, 19x19. Le nombre possible estimé de positions légales est de 10^{170} . En comparaison, le nombre d'atomes dans l'univers est estimé à 10^{80} .

Le jeu d'échecs est plutôt dit tactique alors que le jeu de Go est lui plutôt stratégique. La différence entre les deux est que la stratégie vise le long terme, la tactique s'applique à des actions ponctuelles. Pour gagner au Go il faut donc planifier ses coups sur le long terme. C'est la même chose pour les échecs mais en moindre mesure. C'est pour cela que le jeu de Go est une base extrêmement intéressante à l'application d'algorithmes d'intelligence artificielle. Ce jeu permet d'avoir une base commune pour être capable de comparer les différents algorithmes.

I.1.c - Algorithme minimax

L'algorithme minimax est très important dans la théorie des jeux. Il constitue la base du premier algorithme qui a battu un champion du monde d'échecs. Cet algorithme permet de parcourir l'arbre des possibilités de coups jusqu'à une profondeur prédéfinie puis de retourner le prochain meilleur coup estimé pour la machine.

L'algorithme consiste à passer en revue chaque possibilité de coups jusqu'à une limite prédéfinie. Pour chacune de ces possibilités, une fonction d'évaluation permet de calculer un poids. Cette fonction d'évaluation peut être différente selon les jeux mais elle a le même objectif pour chacun d'eux. Le poid qu'elle retourne doit être d'autant plus grand que le coup est bon pour la machine, il doit être d'autant plus petit que le coup est bon pour le joueur. Un exemple de fonction d'évaluation serait de simuler des centaines de parties au hasard puis de compter celles gagnées par la machine (cf II-3 Monte Carlo).

A partir des valeurs calculées par la fonction d'évaluation, l'algorithme remonte petit à petit l'arbre des possibilités en sélectionnant comme poid pour la feuille parent :

- Le minimum des poids des enfants si le coup est joué par l'ordinateur
- Le maximum des poids des enfants si le coup est joué par le joueur

A la fin de l'algorithme, l'ordinateur sait quelle option choisir pour espérer gagner sur le long terme. Cependant, tout cet algorithme repose sur la performance de la fonction d'évaluation. Celle-ci n'est pas si simple à trouver, voire très difficile et peu performante dans certains cas. Nous verrons dans la suite du rapport l'algorithme de Monte Carlo qui permet de sélectionner les meilleurs coups efficacement.

Un autre inconvénient de cet algorithme est qu'il faut parcourir tout l'arbre pour sélectionner les meilleurs coups. Des techniques existent pour éviter de passer en revue tout l'arbre. La plus connue est l'élagage alpha beta.

I.1.d - Historique et résultats obtenus jusqu'à aujourd'hui

Le point de départ de l'intelligence artificielle se situe dans les années 1950 avec les travaux d'Alan Turing et avec l'apparition des premiers ordinateurs. C'est depuis 1980 environ que l'IA a pris son envol et que des outils efficaces ont vu le jour.

En 1990, la société IBM a commencé à développer son superordinateur 'Deep Blue' spécialisé dans le jeu d'échecs. En 1996, Deep Blue avait affronté le champion du monde d'échecs de l'époque, Garry Kasparov. C'était Kasparov qui avait remporté le premier match. Cependant, en 1997 lors d'un match de revanche, l'ordinateur Deep

Blue a pris le dessus et a remporté le jeu. Cette victoire a fait le tour du monde et a montré que l'IA avait encore de beaux jours devant elle.



Deep Blue 1997

Cependant, en 1997, les ordinateurs jouant au Go ne pouvait gagner que contre des amateurs inexpérimentés. De nombreuses recherches ont débuté et le challenge de battre un joueur de Go expérimenté a motivé le monde de l'intelligence artificielle.

A partir de 2006, les algorithmes font des progrès considérables notamment grâce à la méthode de Monte Carlo dont nous allons développer quelques aspects dans ce rapport.

En 2015, la combinaison du deep learning et du reinforcement learning ont permis au programme Alpha Go de la société DeepMind de battre pour la première fois un joueur humain professionnel.

En mars 2016, à Séoul, ce même programme affronte Lee Sedol, un des meilleurs joueurs mondiaux. Puis un an plus tard, le champion du monde Ke Jie est battu par l'algorithme. Ces deux matchs sont historiques, ils ont permis de définitivement démontrer la performance des algorithmes d'IA dans ces tâches complexes.

En 2017, une nouvelle version d'Alpha Go est développée, Alpha Go Zero. Cette version est encore plus compétente et intéressante. En effet, elle n'utilise aucune connaissance humaine préalable sur le jeu de Go. C'est à dire qu'elle ne demande que les règles du jeu puis par auto apprentissage, elle a obtenu des performances inégalables.

Des recherches n'ont cessé d'être effectuées jusqu'à aujourd'hui.

I.2 - AlphaGO Zero

I.2.a - Historique de l'algorithme

Alpha Go Zero est une version améliorée de l'algorithme initial Alpha Go. Il a été publié dans un article en Octobre 2019. Cet algorithme a été développé par la société Deep Mind.

Deep Mind est une entreprise anglaise fondée en 2010 et rachetée en 2014 par Google.

Après avoir créé l'algorithme Alpha Go en 2016, la nouvelle version Alpha Go Zero se voulait être une évolution, un algorithme plus généraliste que le précédent. Cette version n'utilise aucune donnée de jeux précédemment joué par des humains. Elle commence son apprentissage uniquement avec les règles du jeu de Go, en jouant à des jeux avec elle-même. C'est une révolution dans le monde de l'apprentissage par renforcement. Il a fallu 40 jours à cette IA pour battre toutes les précédentes versions créées auparavant.

Le PDG de Deep Mind, Demis Hassabis, disait alors que Alpha Go Zero était si performant du fait qu'il n'était jamais limité par les limites des connaissances humaines. Tout était appris de A à Z par la machine à partir des règles du jeu.

L'algorithme a été entraîné en utilisant TensorFlow avec 64 GPU et 19 CPU. Les 3 premiers jours, Alpha Go Zero a joué plus de 4.9 millions de parties contre elle-même. Elle pouvait battre des joueurs professionnels seulement après quelques jours alors que l'algorithme Alhpa Go initial aurait eu besoin de plusieurs mois pour atteindre ce niveau.

Le coût matériel du système Alpha Go Zero entier a été estimé à 25 millions de dollars.

I.2.b - Principe de fonctionnement

Comme vu précédemment, il est totalement impossible pour un algorithme de gagner une partie de Go par la force brute, en simulant tous les états possibles. Il y a beaucoup trop de possibilités et l'algorithme tournerait indéfiniment.

Alpha Go Zero utilise l'apprentissage par renforcement. Au lieu de prédire de manière déterministe toutes les stratégies gagnantes, l'algorithme va jouer les stratégies qui ont le plus de chances de le faire gagner.

Pour ce faire, l'algorithme utilise la méthode de **Monte Carlo**. Cette méthode permet de pouvoir simuler un nombre de parties à partir d'un état donné. Imaginons que nous soyons arrivé à un état quelconque du jeu, la technique consiste à simuler un nombre très important d'actions différentes. La réponse est alors stochastique, en moyenne on saura quelle est l'action qui rapporte le plus de gain. Ce n'est pas déterministe mais la technique s'approche au plus proche de la meilleure stratégie plus le nombre d'entraînement préalable est grand.

Le problème est maintenant le suivant: comme toutes les actions ne peuvent pas être simulées, il faut choisir quelles actions sont à simuler. Les actions à simuler doivent en effet être celles qui permettent sûrement de rapporter le plus de gain. Pour cela, AlphaGo Zero se base sur deux réseaux de neurones nommés 'Policy Network' et le 'Value network'.

Le but du Policy Network est, en fonction de l'état actuel du jeu, de prédire quelle est l'action la plus intelligente à prendre pour pouvoir gagner. Le Value Network estime à quel point l'état du jeu à un temps donné ou non. A l'aide de ces deux réseaux, on peut donc savoir à n'importe quel moment du jeu si un joueur est plus apte à gagner qu'un autre.

Il y a une différence notable entre Alpha Go et Alpha Go Zero dans le fonctionnement et lors de l'apprentissage. Alpha Go était un algorithme supervisé. Il devait apprendre sur des parties réellement jouées par des joueurs qualifiés. Il pouvait donc apprendre les meilleurs coups, ceux qui faisaient perdre ou ceux qui faisaient gagner. L'algorithme ne pouvait donc pas être vraiment plus fort que les joueurs sur lesquels il avait appris. Alpha Go Zero est venu régler ce problème. C'est un algorithme non supervisé, il apprend tout seul et par lui-même. Il ne nécessite pas de parties réelles pour apprendre. Il peut donc imaginer des coups jamais vus et faire autant de parties souhaitées.

Alpha Go Zero n'est donc pas biaisé par la manière dont jouent les humains, il est totalement libre dans la création des stratégies de jeu.

I.2.c - Résultats

L'augmentation de performance entre Alpha Go et Alpha Go Zero est très significative. Le tableau ci dessous récapitule les matchs les plus notables :

Année	Affrontement	Ressources	Résultat
2016	AlphaGo / Lee Sedol	48 TPUs distribués sur plusieurs machines	Alpha Go <= 4 / 1 => Lee Sedol
	AlphaGo Zero / AlphaGo Lee *	4 TPUs sur une machine	AlphaGo Zero <= 100 / 0 => AlphaGo Lee

(*) Version donnée à l'algorithme AlphaGo entraîné pour jouer contre Lee Sedol en 2016.

I.3 - AlphaZero

I.3.a - Historique de l'algorithme

AlphaZero est l'ultime version du logiciel initial Alpha Go. Cette version combine les performances de Alpha Go Zero avec, en plus, une adaptation à d'autres jeux.

AlphaZero peut s'adapter aux jeux d'échecs et de shogi en plus du jeu de Go. Il suffit, de la même manière qu'avec Alpha Go Zero, de spécifier les règles du jeu avant l'entraînement. AlphaZero s'entraîne tout seul en jouant contre lui-même. L'algorithme est une généralisation d'Alpha Go Zero.

C'est en Décembre 2017 que la société Deepmind a sorti le papier de recherche concernant AlphaZero [1]. Selon Deepmind, le nouvel algorithme serait capable en seulement 24h de battre des joueurs expérimentés d'échecs, de shogi et de Go. Le Shogi est un jeu traditionnel Japonais qui se rapproche des échecs.

Nous allons développer plus en détail le fonctionnement de AlphaZero dans ce rapport puis réaliser un exemple d'implémentation.

En 2019, DeepMind a annoncé la réalisation de MuZero. Ce logiciel ne se base même plus sur les règles du jeu. Il a juste besoin de connaître en jouant si il peut ou non réaliser un coup particulier ainsi que les conséquences de ce coup. Il peut maintenant s'adapter à un ensemble de jeux sur Atari en plus du Go, Shogi et des échecs. Encore une fois, ce logiciel à des performances égales ou supérieures à tous les algorithmes préexistants, dans tous les jeux auxquels il sait jouer.

I.3.b - Résultats et performance d'AlphaZero comparées aux autres logiciels de jeu d'échec ou de go

Jeu	Affrontement	Ressources	Résultat
Go	AlphaGo Zero / AlphaGo Lee *	4 TPUs sur une machine	AlphaGo Zero <= 100 / 0 => AlphaGo Lee
Go	Alpha Zero / AlphaGo Zero	4 TPUs sur une machine	Alpha Zero <= 60 / 40 => AlphaGo Zero
Échecs	Alpha Zero / Stockfish 8	AlphaZero entraîné sur un total de 9 heures	Alpha Zero gagne 290 fois, nul 886 fois et perds 24
Shogi	Alpha Zero / elmo (World Computer Shogi Championship 27 summer 2017)	AlphaZero entraîné sur un total de 2 heures	AlphaZero gagne 90 fois, perds 8 et nul 2 fois

II Description du logiciel AlphaZero

II-1 Rappel des principes du Deep Reinforcement Learning

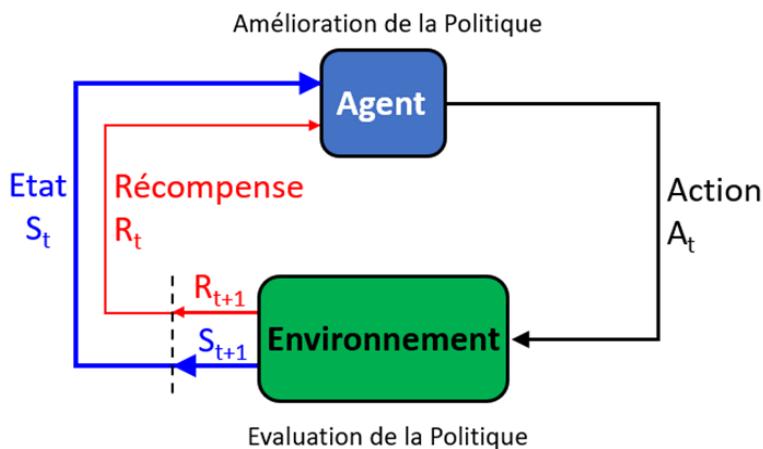
L'Apprentissage par Renforcement (Reinforcement Learning en anglais) est un type d'algorithme qui permet à un Agent de maximiser ses gains dans un environnement où chaque action lui donne une récompense (positive ou négative).

De manière générale, l'Apprentissage par Renforcement permet de traiter des Processus de Décision de Markov finis (Markov Decision Processes ou MDPs en anglais). Les MDP sont une formalisation classique de la prise de décision séquentielle, où les actions effectuées influencent non seulement les récompenses immédiates, mais aussi les situations ou les états ultérieurs, et à travers eux les récompenses futures : il faut donc toujours arbitrer entre récompense immédiate et différée.

L'interaction entre un Agent et son Environnement dans un processus de décision Markovien peut se décomposer en 6 étapes :

- Observer l'environnement à travers l'état S_t
- Décider comment agir en utilisant une stratégie donnée : politique p
- Agir en conséquence (A_t)
- Recevoir une récompense ou une pénalité (R_t)
- Apprendre des expériences passées et affiner la stratégie
- Itérer jusqu'à trouver une stratégie optimale

Ces différentes étapes sont synthétisées dans la figure suivante tirée de la référence [5].



Pour trouver la meilleure politique, l'Agent utilise une fonction d'évaluation $Q(S, A)$ qui, à partir de l'état du système S et de l'action A qui sera effectuée, calcule la récompense complète potentielle qui sera obtenue à long terme en prenant en compte la totalité des actions effectuées. La politique optimale ne doit pas sélectionner les actions qui augmentent la récompense sur le court-terme mais celles qui augmentent la récompense de l'Agent sur le long terme.

Si la dimension de l'espace d'état du système et la dimension de l'ensemble des actions sont petites, il est possible de calculer la fonction Q pour toutes les combinaisons possibles et on obtiendra un résultat précis. Cependant, cette condition n'est pas vérifiée pour le jeu d'échec ou de go car les dimensions sont très grandes comme nous l'avons vu au paragraphe I.1.b.

La fonction d'évaluation Q peut alors être modélisée de manière approchée avec un réseau de neurones ; ce qui permet d'éviter de parcourir l'ensemble des états, grâce à la capacité de généralisation de ce type de réseau : il va appliquer ses connaissances acquises à l'entraînement sur des états qu'il n'a encore jamais rencontrés.

En machine Learning, pour entraîner un réseau de neurones, on n'utilise pas un seul échantillon mais un ensemble de données appelé « *batch* ». Cela permet à l'algorithme de gradient de choisir une direction moyenne ; ce qui conduit globalement à de meilleurs résultats.

Dans le cas du Reinforcement Learning, chaque action ne génère qu'un seul échantillon de variable d'état ; on ne peut donc pas l'utiliser directement dans l'entraînement du réseau de neurones car cela ralentirait la convergence et pourrait même conduire à des oscillations. De plus, un état du système dépend de l'ensemble des états précédents : la chronologie des actions effectuées a donc un impact sur les récompenses obtenues. C'est pourquoi, après chaque action de l'Agent, on stocke dans une mémoire les paramètres d'entraînement (état de départ, action, état d'arrivée, récompense, fin du jeu). Ensuite, régulièrement, on va piocher au hasard un certain nombre d'éléments dans cette mémoire pour constituer un mini-batch d'entraînement du réseau de neurones. Cette méthode s'appelle la répétition d'expérience (« *experience replay* » en anglais).

II-2 Les principaux composants du logiciel AlphaZero

AlphaZero a deux composants que nous allons détailler dans les paragraphes suivants :

- Un **réseau de neurones profond** qui calcule une politique et une valeur estimée à partir d'un état,
- Un **arbre de recherche de Monte Carlo** qui utilise le réseau de neurones pour évaluer de manière répétée les états et mettre à jour sa règle de sélection des actions afin de n'explorer qu'une partie de l'arbre.

Dans le cas des jeux qui sont traités dans l'article, un « état » est une position et éventuellement un historique des positions précédentes. Il est représenté comme un vecteur à valeur réelle S_0 .

Le réseau de neurones prédit deux quantités qui sont apprises à partir de jeux de données d'entraînement et qui sont utilisées dans l'arbre de Monte Carlo :

$$p, v = f_{\theta}(S_0)$$

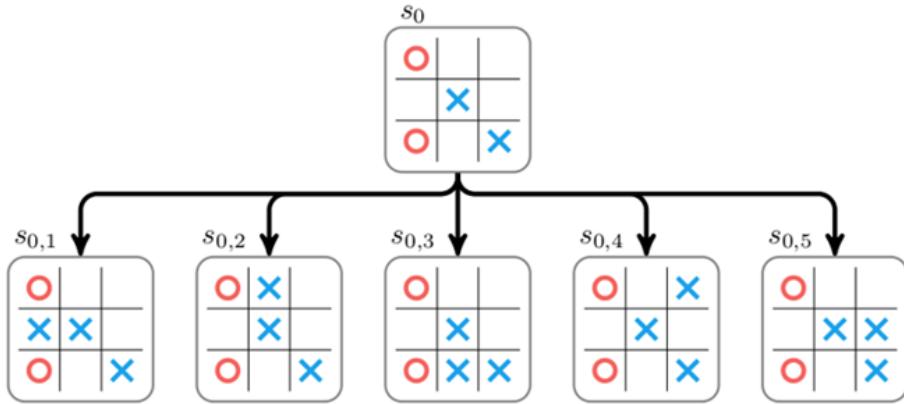
v est une estimation du résultat du jeu attendu à partir de la position actuelle des pièces sur la table de jeu et p est la distribution de probabilité sur le prochain mouvement de pièces du jeu.

Les paramètres θ du réseau de neurones sont initialisés de manière aléatoire. Puis AlphaZero joue de nombreuses fois contre lui-même et les données de ces parties sont insérées dans une file d'attente pour servir de données d'entraînement au réseau de neurones. L'utilisation de l'arbre de recherche de Monte Carlo permet d'avoir des jeux de bonne qualité ; les meilleurs sont intégrés dans la file d'attente et les jeux antérieurs sont abandonnés.

Simultanément, on utilise la descente de gradient pour minimiser la différence entre les prédictions actuelles du réseau de neurones et des positions extraites de la file d'attente qui donnent le coup joué à partir d'une position et le résultat de la partie.

II-3 Sélection des déplacements à explorer avec la recherche arborescente de Monte Carlo (MCTS)

Les arbres sont les structures de données très utilisées pour représenter un jeu. Chaque nœud de l'arbre représente un état particulier dans le jeu. En effectuant un déplacement autorisé par la règle du jeu, on effectue une transition d'un nœud vers ses enfants comme on peut le voir sur la figure suivante tirée de [6] qui modélise un jeu de morpion.



Ce processus se poursuit jusqu'à ce que l'on atteigne le nœud feuille où le résultat de la partie est évalué : +1 pour une victoire, -1 pour une défaite et 0 en cas d'égalité. On construit ainsi l'arbre de recherche, nœud par nœud comme on peut le voir sur la figure suivante tirée de [7] :

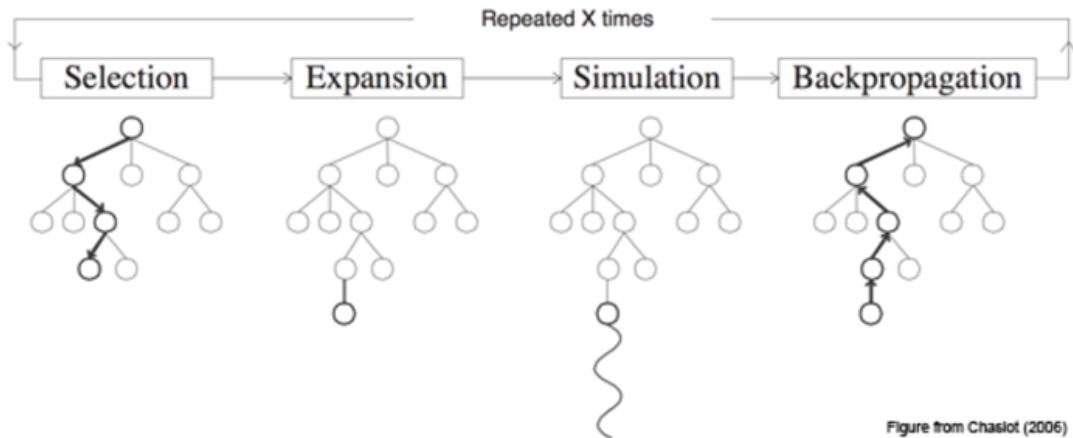


Figure from Chaslot (2006)

La recherche de l'arbre de Monte Carlo se poursuit pendant de multiples itérations consistant à sélectionner un nœud, à le développer en créant un nœud enfant, à simuler le résultat du jeu et à propager vers le haut les nouvelles informations.

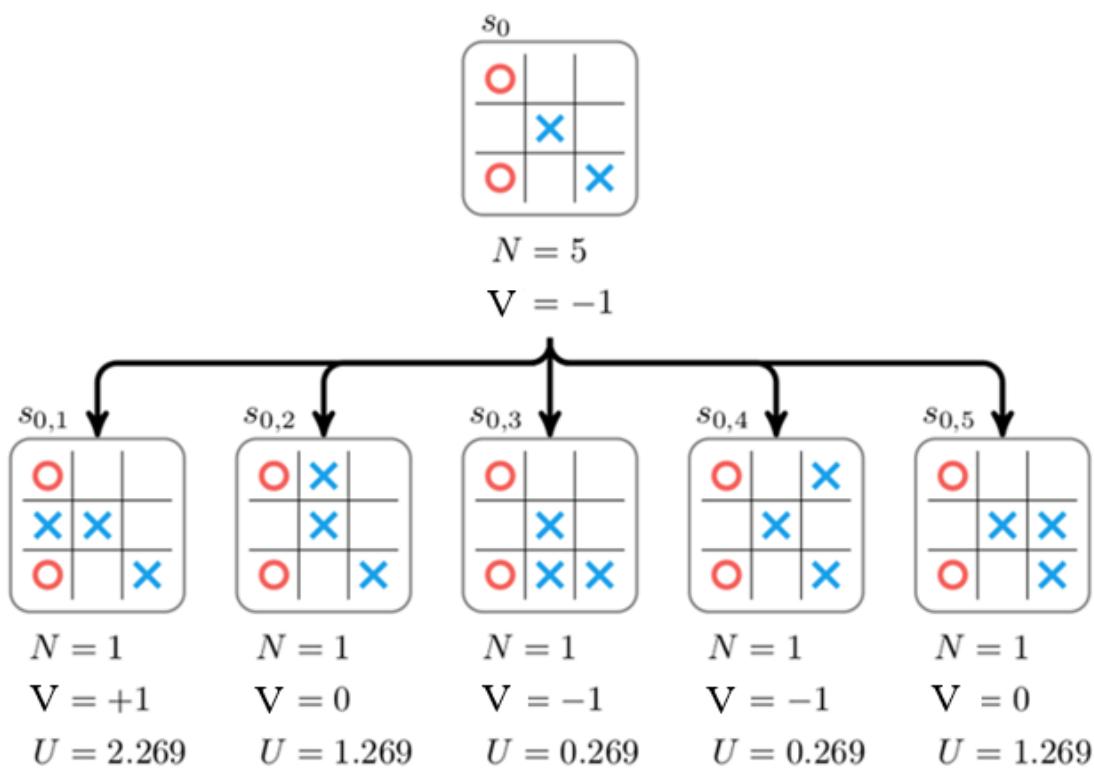
On ne développe cependant pas tous les nœuds feuilles, car cela serait très coûteux. On sélectionne les nœuds qui ont une estimation de gain élevée et qui sont relativement inexplorés (faible nombre de visites). En parcourant l'arbre depuis le nœud racine, on choisit donc le nœud enfant qui a le plus grand score de confiance UCT, noté U_i :

$$U_i = \frac{V_i}{N_i} + c \sqrt{\frac{\ln N_p}{N_i}}$$

Où :

- V_i est la valeur cumulée du i ème nœud enfant,
- N_i est le nombre de visites pour le i ème nœud enfant,
- N_p est le nombre de visites pour le nœud parent.
- Le paramètre $c \geq 0$ contrôle le compromis entre le choix des nœuds lucratifs (c faible) et l'exploration des nœuds à faible nombre de visites (c élevé). Il est souvent fixé de manière empirique.

En reprenant l'exemple du jeu de morpion, nous avons par exemple :



Nous choisissons donc de développer le nœud $S_{0,1}$ qui a la plus grande valeur de U .

Dans le cas des jeux d'échecs et de go, l'arbre a un très grand nombre de ramifications et l'évaluation des mouvements avec la recherche arborescente de Monte Carlo que nous venons de décrire n'est pas assez efficace.

Supposons donc que nous ayons également une politique d'expert π qui nous indique la probabilité de choisir la i ème action a_i , étant donné l'état racine S_0 :

$$P_i = \pi(a_i | s_0)$$

Nous allons utiliser cette probabilité d'action pour ajuster le score du nœud lors de la sélection :

$$U_i = \frac{V_i}{N_i} + cP_i \sqrt{\frac{\ln N_p}{1 + N_i}}$$

Comme précédemment, le score est un compromis entre les nœuds qui produisent constamment des scores élevés et les nœuds qui ne sont pas explorés. Maintenant, l'exploration des nœuds est orientée vers les actions que la politique de l'expert considère comme probables.

Les performances peuvent être également améliorées en évitant de développer les nœuds. et en estimant directement la valeur d'un état dans [-1,1] avec une fonction d'approximation de la valeur : $\hat{V}(x)$.

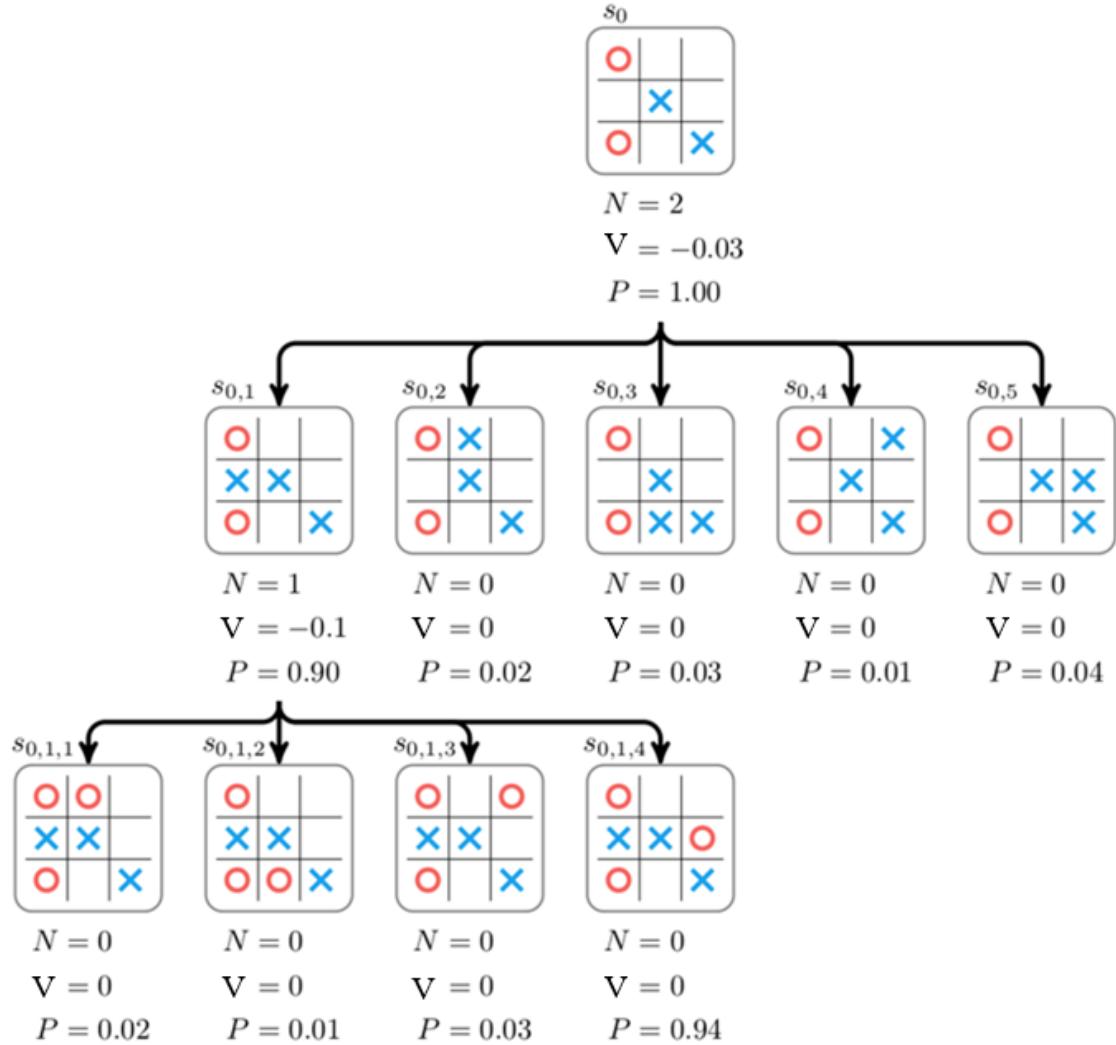
L'algorithme Alpha Zero produit des politiques expertes et des fonctions d'évaluation de plus en plus performantes au fil du temps en jouant de nombreuses parties contre lui-même et en utilisant une recherche accélérée d'arbres de Monte Carlo.

La politique experte π et la fonction d'évaluation approximative \hat{V} sont toutes deux représentées par des réseaux de neurones profonds. En fait, pour augmenter l'efficacité, Alpha Zéro utilise un seul réseau de neurones f qui prend en entrée l'état du jeu et produit à la fois les probabilités associées au prochain coup et une valeur approximative de l'état correspondant :

$$f(S) \rightarrow [p, V]$$

Les feuilles de l'arbre de recherche sont développées en les évaluant directement avec le réseau de neurones. Chaque nœud enfant est initialisé avec $N=0$, $V=0$, et avec une probabilité p correspondant à la prédiction du réseau de neurones. La valeur du nœud développé est fixée à la valeur prédite et cette valeur est ensuite remontée dans l'arbre jusqu'à la racine. Les étapes de sélection et de remontée des valeurs sont inchangées : pendant la remontée, le nombre de visites d'un parent est incrémenté et sa valeur est augmentée en fonction de V .

En reprenant l'exemple du jeu de morpion, on obtient l'arbre de recherche suivant :

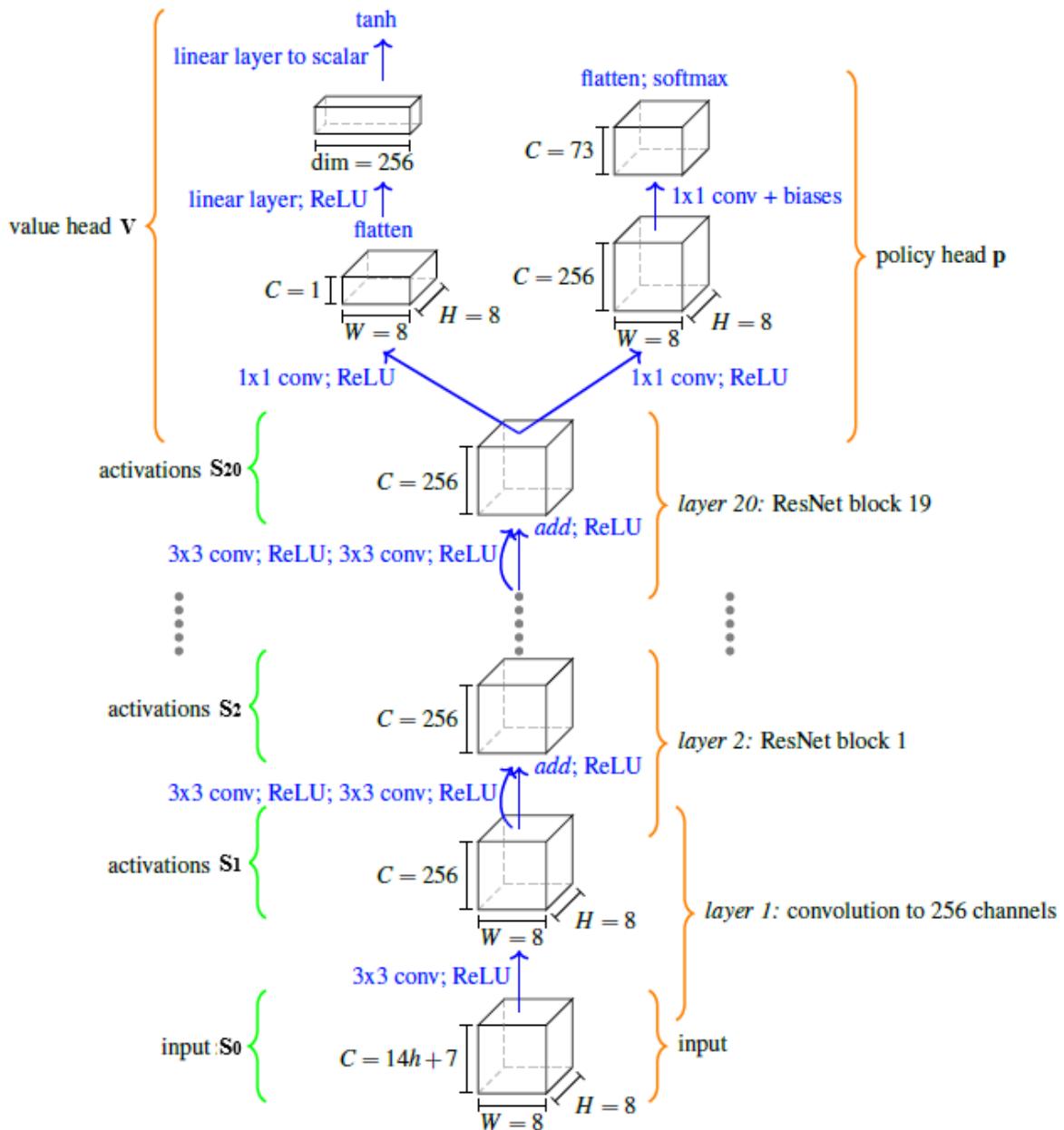


L'idée centrale de l'algorithme Alpha Zero est que les prédictions du réseau de neurones peuvent être améliorées, et que le jeu généré par la recherche d'arbre de Monte Carlo peut être utilisé pour fournir les données d'entraînement.

La partie politique du réseau de neurones est améliorée en entraînant les probabilités prédites p pour S_0 afin qu'elles correspondent à la probabilité améliorée π obtenue en exécutant l'arbre de Monte Carlo sur S_0 .

II-4 Le réseau de neurones profond d'AlphaZero

Le réseau de neurones profond d'AlphaZero est représenté sur la figure suivante tirée de [8]. Il est composé de 20 couches convolutives et possède deux sorties comme nous l'avons dit au paragraphe précédent : la politique experte p et la fonction d'évaluation V qui est utilisée dans l'arbre de recherche de Monte Carlo.



AlphaZero a une ossature constituée d'une séquence de Réseaux Neuronaux Résiduels (ResNet) qui forment des couches indicées de 1 à 19. Chaque convolution 3×3 applique 256 filtres de taille de kernel 3×3 avec stride 1. Un bloc ResNet contient deux couches convolutionnelles normalisées par batch avec une connexion saute-couche (skip connection).

L'entrée S_0 permet de prendre en compte une longueur d'historique h . On choisit généralement un historique de 8 coups qui code la position courante du plateau de jeu et celle des 7 coups précédents. L'entrée du réseau de neurones est alors un tenseur à $8 \times 8 \times 119$ dimensions.

Le réseau transforme progressivement l'entrée S_0 en S_1 , puis S_2 , et ainsi de suite jusqu'à aboutir au couple de sorties : politique/valeur finale. Les activations des couches $k \geq 2$, sont données par la fonction :

$$S_k = f_{\theta^k}(S_{k-1}) = \text{ReLU}(S_{k-1} + g^k(S_{k-1}))$$

qui copie directement les activations S_{k-1} , y ajoute une fonction non linéaire $g^k(S_{k-1})$ composée de deux couches de convolution supplémentaires, et force le résultat à être non négatif en utilisant une fonction d'activation linéaire rectifiée (ReLU).

Comme nous l'avons déjà dit, les paramètres θ du réseau de neurones sont initialisés aléatoirement puis sont entraînés à partir des résultats de parties jouées par AlphaZero contre lui-même. Chacune de ces parties est jouée en sélectionnant successivement les coups de chaque joueur (action $a(t)$) avec un Arbre de recherche de Monte-Carlo : $a(t) \sim \pi(t)$. Le résultat final de chaque partie détermine la valeur de Z pour cette partie comme nous l'avons vu au paragraphe précédent.

Les paramètres θ du réseau de neurones sont mis à jour afin de minimiser l'erreur entre le résultat prédit $V(t)$ et le résultat du jeu Z , et afin de maximiser la similarité du vecteur de politique $p(t)$ avec les probabilités de recherche $\pi(t)$ calculées avec l'arbre de Monte Carlo. Plus précisément, les paramètres θ sont ajustés par descente de gradient sur une fonction de perte (loss) qui additionne respectivement l'erreur quadratique moyenne sur la valeur et la perte d'entropie croisée (cross-entropy) sur la politique :

$$\left\{ \begin{array}{l} (\mathbf{p}, V) = f_{\theta}(s) \\ \text{loss} = (Z - V)^2 + \pi^T \log \mathbf{p} + c \|\theta\|^2 \end{array} \right.$$

où $c \geq 0$ est un paramètre contrôlant le niveau de régularisation des poids.

Les paramètres mis à jour sont utilisés dans les parties suivantes d'AlphaZero contre lui-même.

III Implémentation d'une version simplifiée d'AlphaZero et application au jeu Puissance 4

Conclusion

Grâce à l'analyse du papier de recherche “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”, nous avons pu implémenter notre propre algorithme AlphaZero qui joue au puissance 4. Nous avons également découvert l'historique de l'intelligence artificielle dans les jeux d'échecs et de go. Ensuite, nous avons pu analyser les principales techniques qui fondent ces types d'algorithmes, comme l'algorithme minimax ou l'algorithme de Monte Carlo. Pour terminer, nous avons travaillé sur l'application du deep learning dans les techniques de Reinforcement Learning.

Cela a été très formateur de travailler sur ce sujet. En effet, nous avons pu mettre en pratique la théorie que nous avions étudiée lors du cours de Reinforcement Learning.

Sources

- 1) David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis: "Mastering chess and shogi by self-play with a general reinforcement learning algorithm." arXiv preprint arXiv:1712.01815 (2017). <https://arxiv.org/abs/1712.01815>
- 2) ChessPositionRanking : <https://github.com/tromp/ChessPositionRanking>
- 3) Le jeu de Go : [https://fr.wikipedia.org/wiki/Go_\(jeu\)#Intelligence_artificielle](https://fr.wikipedia.org/wiki/Go_(jeu)#Intelligence_artificielle)
- 4) Alpha Go Zero : https://en.wikipedia.org/wiki/AlphaGo_Zero
- 5) R. Sutton and A. Barto: Reinforcement Learning, an Introduction (2nd ed.) MIT Press, 2018 <http://incompleteideas.net/book/RLbook2020.pdf>
- 6) Tim Wheeler - 2017 : AlphaGo Zero – How and Why it Works - <http://tim.hibal.org/blog/alpha-zero-how-and-why-it-works/>
- 7) Ankit Choudhary - 2019 : Introduction to Monte Carlo Tree Search: The Game-Changing Algorithm behind DeepMind's AlphaGo - <https://medium.com/analytics-vidhya/introduction-to-monte-carlo-tree-search-the-game-changing-algorithm-behind-deepminds-alphago-554a9017f0c2>
- 8) Thomas McGrath, Andrei Kapishnikov, Nenad Tomasev, Adam Pearce, Demis Hassabis, Been Kim, Ulrich Paquet, and Vladimir Kramnik: "Acquisition of Chess Knowledge in AlphaZero." arXiv preprint arXiv:2111.09259v2 [cs.AI] (2021). <https://arxiv.org/abs/2111.09259>