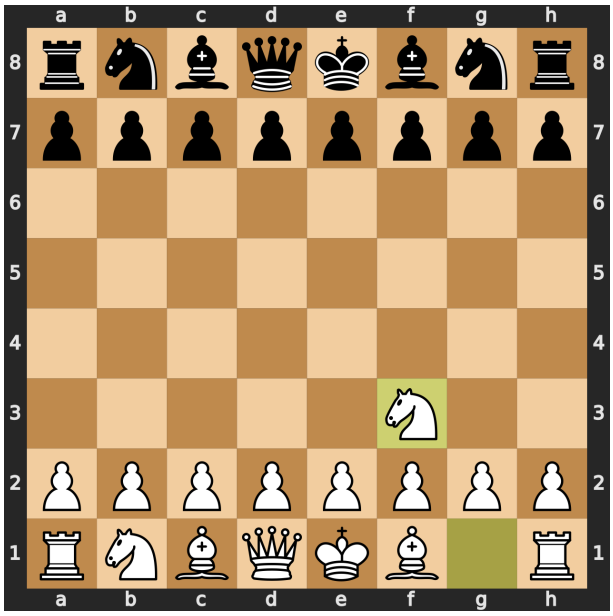
Projet - Reinforcement Learning

Analyse de l'article Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm



Master 2 - Data Science
Institut Polytechnique de Paris
Cours : Reinforcement Learning
Auteurs : Kévin Assobo - Robin Labbé - Joshua Wolff

Résumé

Ce document vise à résumer l'article Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm produit par DeepMind en 2017 (Silver et al. [2017b]). Nous faisons appel à d'autres travaux pour compléter cet article, notamment l'article Acquisition of Chess Knowledge in AlphaZero aussi produit par DeepMind (McGrath et al. [2021]), afin de comprendre finement la méthodologie d'AlphaZero. Le travail porte essentiellement sur l'étude de la méthode dans le cadre du jeu d'échecs, même si des allusions au jeu de Go ainsi qu'au Shogi sont faites. Avant d'entrer dans le vif du sujet, à savoir le fonctionnement d'AlphaZero, nous dressons un court état de l'art des moteurs d'échecs jusqu'ici utilisés. Par la suite, nous décrivons en détail le fonctionnement du moteur d'échec AlphaZero. Nous expliquons tout d'abord indépendemment le fonctionnement de chaque bloc élémentaire de la méthode, puis nous les regroupons pour montrer comment ces différents blocs interagissent pour aboutir à AlphaZero. Enfin, nous présentons une implémentation d'un des blocs élémentaires de cette méthode : la recherche arborescente Monte-Carlo (Monte Carlo Tree-Search, MCTS). Notre implémentation de cet algorithme est spécifique au jeu d'échecs, mais peut aisément être généralisée à une large classe de problèmes. La dernière partie du document porte sur les récentes évolutions proposées par DeepMind pour améliorer AlphaZero sur le plan des performances et surtout de l'interprétabilité.

Table des matières

1 Histoire des ordinateurs dans le jeu d'échecs

Le jeu d'échecs fascine le monde de la recherche depuis l'invention de l'ordinateur. La très grande richesse de ses positions (environ 10^{40} positions légales) entraîne une quantité gigantesque de parties possibles. Une estimation grossière de ce nombre est de 10^{120} selon le célèbre article Programming a Computer for Playing Chess écrit en 1950 par Claude Shannon, l'un des pères de la théorie de l'information (Shannon [1950]). Claude Shannon y fournit sa vision des moteurs d'échecs, et l'immense majorité des travaux ultérieurs à ce sujet reposent sur ce travail. Il y décrit notamment comment un échiquier peut être représenté par un ordinateur, comment évaluer une position, et exhibe deux types de recherche de coups pertinents : une approche par force brute, et une approche plus "humaine" visant à explorer uniquement les variantes intéressantes selon un point de vue anthropique. Il y expose enfin son processus de type minimax, qui servira de base aux moteurs les plus performants avant l'arrivée d'AlphaZero.

Un tournant majeur dans le développement des moteurs d'échecs est la défaite du multiple champion du monde Garry Kasparov contre le programme Deep Blue développé par IBM en 1997. Cet événement montre la capacité des algorithmes d'échecs à surpasser les meilleurs humains et participera à l'accélération de leur développement durant les décennies suivantes (Silver et al. [2017b]). Pendant plus de 20 ans, les meilleurs programmes seront basés sur des fonctions d'évaluation de positions faites à la main par des spécialistes du jeu combinées avec des recherches arborescentes très performantes, elles-mêmes guidées par des heuristiques spécifique au jeu. Bien que très performants (aucun joueur au monde ne peut actuellement rivaliser avec ce type de programmes), ils sont très limités dans leur capacité de généralisation. Les siècles de pratique échiquéenne ont permis à l'humain d'acquérir une riche vision du jeu d'échecs, mais il serait très difficile d'appliquer de tels algorithmes à une nouvelle situation dans la mesure où les fonctions d'évaluation dépendent uniquement de notre compréhension de cette situation.

En 2017, DeepMind publie un article nommé *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithmn* faisant suite au développement d'un algorithme pour jouer au jeu de Go, désormais appliqué au jeu d'échecs. L'approche est révolutionnaire dans la mesure ou la méthode est absolument nouvelle et mélange recherche arborescente et apprentissage profond, mais surtout car elle ne part d'aucune donnée / connaissance humaine. L'entièreté des données d'apprentissage est générée lors de phases de jeu contre lui même. En plus de faire table rase de toute connaissance humaine, cette approche permet à AlphaZero de battre le meilleur algorithme à cet période (StockFish, un programme d'échecs open-source multiple champion du

monde).

2 Fonctionnement général d'un moteur d'échec

Pour mieux comprendre l'apport d'AlphaZero par rapport aux programmes précédents, nous éclaircissons dans cette deuxième partie la différence entre les algorithmes "classiques" (i.e. basés sur des approches de type minimax/alpha-beta) et les algorithmes utilisant des approches basés sur l'apprentissage par renforcement.

2.1 Algorithmes classiques

Le jeu d'échecs est un jeu à deux joueurs à somme nulle (une victoire fait gagner +1, une partie nulle +0, et une défaite -1). Il peut être vu comme un arbre dont chaque branche représente un coup. L'idée principale des programmes classiques est de parcourir ces arbres aussi profondément que possible pour en déduire le meilleur coup suivant.

2.1.1 Algorithme minimax

De façon générique, l'algorithme minimax cherche à descendre dans l'arbre jusque à atteindre une feuille, à laquelle est associée une valeur. Le coup choisi par le joueur correspond au coup allant vers la feuille qui minimise les pertes du joueur en supposant que l'opposant cherche au contraire à les maximiser. En pratique cette approche est très coûteuse en temps de calcul et en espace de stockage. Une façon de contrer ces problèmes est de ne pas descendre jusqu'à une feuille, mais de s'arrêter à une profondeur arbitraire et d'évaluer la position à l'aide d'une fonction d'évaluation construite par des humains selon les connaissances des meilleurs joueurs. Par exemple, aux échecs, une approche simple consiste à affecter des valeurs aux pièces, un coup aura alors une valeur qui dépend des pièces prises grâce à ce coup (par exemple un coup permettant de s'emparer d'une reine vaut 9, un coup permettant de s'emparer d'un pion 1, un coup où l'on ne s'empare de rien 0 etc.). La profondeur d'évaluation de la position est limitée par les capacités du matériel informatique à disposition. Cette limitation réduit fortement les performances potentiels de la méthode car plus une exploration est profonde, meilleurs sont les coups joués. Cette limitation est cependant obligatoire, une partie d'échecs contient en moyenne 40 coups, on estime que cela constitue environ 10^{120} parties possibles, d'où l'impossibilité d'effectuer une exploration complète de l'arbre.

2.1.2 Élagage Alpha-Beta

Pour tenter de pallier ce problème des techniques d'élagages ont été développées. La technique d'élagage Alpha-Beta est employée pour améliorer les performances de l'algorithme minimax. L'objectif est d'évaluer moins de noeuds pour concentrer la puissance de calcul dans l'approfondissement de noeuds plus intéressants. Plus précisément, l'élagage Alpha-Beta consiste à suivre deux variables α et β qui contiennent respectivement la valeur minimale que le joueur peut obtenir et la valeur maximale que l'adversaire peut obtenir. Certains noeuds ne sont pas approfondis car ils indiquent la possibilité de faire un coup qui viole l'une des conditions suivante : α est le plus petit gain que le joueur sait pouvoir obtenir OU β est le plus grand gain que le joueur autorisera l'adversaire à obtenir. Cette méthode permet explorer plus profondément l'arbre des positions, cependant elle dépend de la méthode de valuation des noeuds car l'exploration, bien que plus profonde, ne se fera pas jusqu'à une position de fin de partie.

2.2 Algorithmes basés sur de l'apprentissage profond

Les avancées en apprentissage profond au cours de ces vingt dernières années ont ouvert la porte à un nouveau type de programmes, basés à la fois sur de la recherche dans des arbres et sur l'emploi de réseaux de neurones. L'idée derrière cette symbiose est d'utiliser les réseau de neurones comme des "experts" pouvant fournir une politique cohérente au cours du jeu. Cette politique d'expert est alors utilisée afin de restreindre le parcours de l'arbre aux branches les plus cohérentes.

Plusieurs démarches sont envisageables afin d'entraîner le réseau de neurones. La première consiste à utiliser des positions de "parties de maîtres" (parties jouées par les plus fort joueurs humains) comme données d'entraînement. L'idée sous-jacente est alors de "copier" le raisonnement humain dans la politique d'expert. Une perspective plus ambitieuse serait de ne partir d'aucune connaissance à l'exception des règles du jeu, et de générer les données d'entraînement par *self-play*, c'est à dire en faisant jouer le modèle contre lui même. De cette manière l'apprentisage est fait par renforcement en faisant table rase de toute a priori d'origine humaine. C'est précisément grâce à cette deuxième approche que les programmes AlphaGo Zero (Silver et al. [2017a]), puis AlphaZero (Silver et al. [2017b]), se sont démarqués des meilleurs programmes basés sur l'algorithme minimax avec élagage alpha-beta.

3 AlphaZero

Nous présentons dans cette partie le fonctionnement de l'algorithme AlphaZero développé par DeepMind. Nous verrons tout d'abord l'évolution entre AlphaGo Zero et AlphaZero, puis nous expliquerons les hypothèses sur lesquels se base la méthode. Nous nous focaliserons ensuite sur le coeur d'AlphaZero : à savoir le couplage apprentissage profond / recherche arborescente Monte Carlo, ainsi que la stratégie d'entraînement basée sur de l'apprentissage par renforcement.

3.1 D'AlphaGo Zero à AlphaZero

Les travaux initiaux de DeepMind portaient sur la réalisation d'un programme capable de rivaliser avec les tous meilleurs joueurs de Go au monde. En effet, la complexité combinatoire de ce jeu est encore plus importante que celle des échecs et aucun ordinateur n'avait jusqu'alors réussi à surpasser un maître humain du jeu de Go. Ils développent pour cela un programme, nommé AlphaGo, basé sur de l'apprentissage profond couplé avec de la recherche dans des arbres, associé à de nombreux entraînements avec des humains, d'autres ordinateurs, et aussi contre lui-même. En mars 2016, AlphaGo bat Lee Sedol (un des meilleur joueurs mondiaux) puis Ke Jie (champion du monde) en mai 2017. Après ces succès, DeepMind publie l'article Mastering the game of go without human knowledge (Silver et al. [2017a]) qui présente leur nouveau programme AlphaGo Zero. Ce dernier, toujours appliqué au jeu de Go, est uniquement entraîné par *self-play* et réussi le tour de force de surpasser son prédécesseur AlphaGo.

Pour des raisons structurelles, le jeu de Go est parfaitement adapté au couches convolutionnelles des réseaux de neurones utilisés dans AlphaGo et AlphaGo Zero :

— Les règles du jeu de Go sont invariantes par translation, ce qui correspond au principe de partage des poids des convolutions.

— Elles sont invariantes par rotation et par réflexion, ce qui favorise l'utilisation d'une méthode classique d'apprentissage profond, l'augmentation de données, qui permet notamment de gagner en robustesse tout en augmentant la quantité de données d'apprentissage.

— Enfin l'espace d'état est simple (une *pierre* peut être posée sur chaque intersection du *goban*), et l'issue de la partie est nécessairement une victoire d'un des deux joueurs (il n'y a pas de partie nulles), ce qui participe à simplifier l'architecture du réseau de neurones.

Le programme AlphaZero est une généralisation d'AlphaGo Zero, avec une méthodologie très générique pouvant s'appliquer à de nombreux jeux, y compris ceux ne présentant pas des propriétés intéressantes comme le jeu de Go. En particulier, le jeu d'échecs ne semble pas bien se prêter à une structure classique de réseau de neurones convolutionnel :

— Les règles dépendent de la position des pièces sur l'échiquier et sont asymétriques.

— Les pièces peuvent avoir des interactions à longue distance, or les convolutions captent essentiellement les interactions locales.

— L'espace d'état est assez complexe, il contient toutes les positions atteignables légalement pour toutes les pièces des joueurs.

— Enfin, la possibilité d'avoir une partie nulle est possible, c'est même l'issue principale à haut niveau.

La méthodologie initiale a donc du être adaptée à toutes ces nouvelles propriétés, notamment en travaillant sur la structure de l'entrée du réseau de neurones que nous décrivons en partie 3.3.1. Tout comme pour AlphaGo Zero, les réseaux de neurones d'AlphaZero sont entraînés grâce à une démarche d'apprentissage par renforcement uniquement basée sur du *self-play*.

3.2 Connaissance du jeu

Comme expliqué précédemment, le tour de force réalisé par DeepMind dans la réalisation d'AlphaZero est d'avoir produit le meilleur programme d'échec à sa sortie sans avoir recourt à la moindre expertise humaine. Nous listons ici l'ensemble des connaissances échiquéennes utilisées au coeur d'AlphaZero :

— AlphaZero dispose d'une connaissance partfaite des règles du jeu d'échecs.

— Plus précisément, elles sont utilisées dans la phase de recherche arborescente Monte Carlo pour générer les coups potentiels découlant d'une position, mais aussi pour savoir si une position correspond à une fin de partie et évaluer l'issue d'une simulation MCTS si elle aboutit à une configuration correspondant à une fin de partie.

— Certaines règles, comme le roque, les répétitions ainsi que la règle des 50 coups sont encodés dans le format de l'entrée du réseau de neurones.

— La connaissance du nombre moyen de coups joués au cours d'une partie est utilisé pour dimensionner le bruitage des sorties du réseau de neurones afin d'assurer l'exploration de l'arbre lors de la recherche arborescente Monte Carlo.

Ces éléments constituent les seules connaissances du jeu d'échec à disposition du programme AlphaZero.

3.3 Apprentissage profond

Nous nous penchons dans cette partie sur la structure du réseau de neurones employé dans AlphaZero pour guider la recherche arborescente Monte Carlo.

3.3.1 L'entrée du réseau de neurones

La question de la représentation de l'échiquier qui sera utilisée comme format d'entrée du réseau de neurones est fondamentale. Elle doit décrire de façon exhaustive la position, en prenant notamment en compte toutes les subtilités liés aux règles complexes des échecs comme expliqué en partie 3.1. Par ailleurs il est important de noter que la taille de l'entrée du réseau de neurones influence très fortement le choix de sa structure, dans la mesure ou une entrée de grande dimension entraîne souvent une augmentation de la taille du réseau, et donc un entraînement plus coûteux en calculs.

L'entrée du réseau de neurones est un objet de $\mathbb{R}^{8\times8\times(14h+7)}$, qui s'interprète comme $14h + 7$ matrices de taille 8×8 décrivant chacune soit la position de certaines pièces, soit l'état de la partie vis à vis des règles spécifiques des échecs (roque, règle des 50 coups...). Nous précisons ci-dessous la signification de chacune des ces matrices dans le cas où $h = 1$ (non préciserons la signification de h par la suite) :

— Les 12 premières matrices 8×8 sont binaires et représentent la position de chaque type de pièce de chaque joueur sur l'échiquier. En effet, chaque joueur dispose de 6 types de pièces différentes (pions, cavaliers, fous, tours, dame, roi).
— 4 matrices sont utilisés pour indiquer la possibilité de roquer (petit roque et grand roque) pour chaque joueurs.
— 2 matrices dont utilisés pour compter le nombre de coups (dans le cadre de la règle des 50 coups et aussi pour compter le nombre de coups total).
— Les matrices restantes décrivent le nombre de répétition de la position (une partie est déclarée nulle si la même position se produit 3 fois), ainsi que lequel des deux joueurs doit jouer.

Le paramètre h désigne le nombre de demi-coups passés que l'on souhaite considérer en entrée du réseau. Bien que la position soit parfaitement décrite avec $h = 1$, les développeurs d'AlphaZero on décidé de choisir $h = 8$ pour des raisons empiriques, les résultats étant en pratique meilleurs lorsque l'on considère quelques demi-coups en amont de la position. Notons que l'entrée du

réseau de neurones est systématiquement transposée de façon à être orientée du point de vue du camp qui doit jouer son coup.

3.3.2 La sortie du réseau de neurones

La sortie du réseau de neurones est en réalité constituée de deux sorties. On peut écrire de façon générale le modèle sous la forme $f_\theta(s) = (\mathbf{p}, v)$ avec s la position telle que nous l'avons décrite en partie 3.3.1, θ les paramètres (poids, biais) du réseau, $\mathbf{p}$ un vecteur de probabilités et enfin $v \in [-1, 1]$ qui évalue la position (proche de 1 si c'est une position gagnante pour le joueur, proche de -1 elle est en faveur de son adversaire). Le vecteur $\mathbf{p}$ est de dimension 4672 pour décrire l'ensemble des coups possibles dans une partie (*c.f.* la convention Universal Chess Interface / UCI). Il est important de remarquer que la dimension de $\mathbf{p}$ est bien supérieur au nombre de coups légalement jouables dans une position (il y en a en moyenne 20), mais du fait de la nécessité d'avoir une sortie de taille fixe, il est impératif de considérer toutes les possibilités de déplacement de pièces sur l'échiquier mais aussi les possibilités de roques, de promotions...

3.3.3 Structure du réseau

Le format de l'entrée et de la sortie du réseau a été traité en partie 3.3.1 et 3.3.2, nous décrivons maintenant la structure des couches cachées du réseau de neurones. L'article original sur AlphaZero (Silver et al. [2017b]) contient peu d'informations à ce sujet, mais un article plus récent (McGrath et al. [2021]) mentionne ce sujet en détail.

Le réseau de neurones est principalement constitué de 18 réseaux de neurones résiduels. La première couche est une couche convolutionnelle suivie des 18 ResNet. La sortie du dernier réseau de neurones résiduel débouche sur deux structures indépendantes, l'une responsable de produire le vecteur de probabilité $\mathbf{p}$ et l'autre de fournir l'évaluation de la position v. Remarquons que v est issue d'une application de la fonction tangente hyperbolique et est donc nécessairement comprise entre -1 et 1, tandis qu'une fonction *softmax* est appliqué pour produire le vecteur de probabilité ce qui assure que tous les coefficients de ı sont positifs et que leur somme vaut bien 1. Il faut cependant noter que cela n'empêche en rien que des probabilités non nulles soient associées à des coups illégaux selon les règles du jeu d'échecs. Cela ne pose pas de problème en pratique, car l'algorithme de recherche arborescente Monte Carlo prend en compte ces règles et ne considère pas les coups illégaux.

Une représentation graphique de l'architecture du réseau de neurones d'AlphaZero est disponible en figure 1.

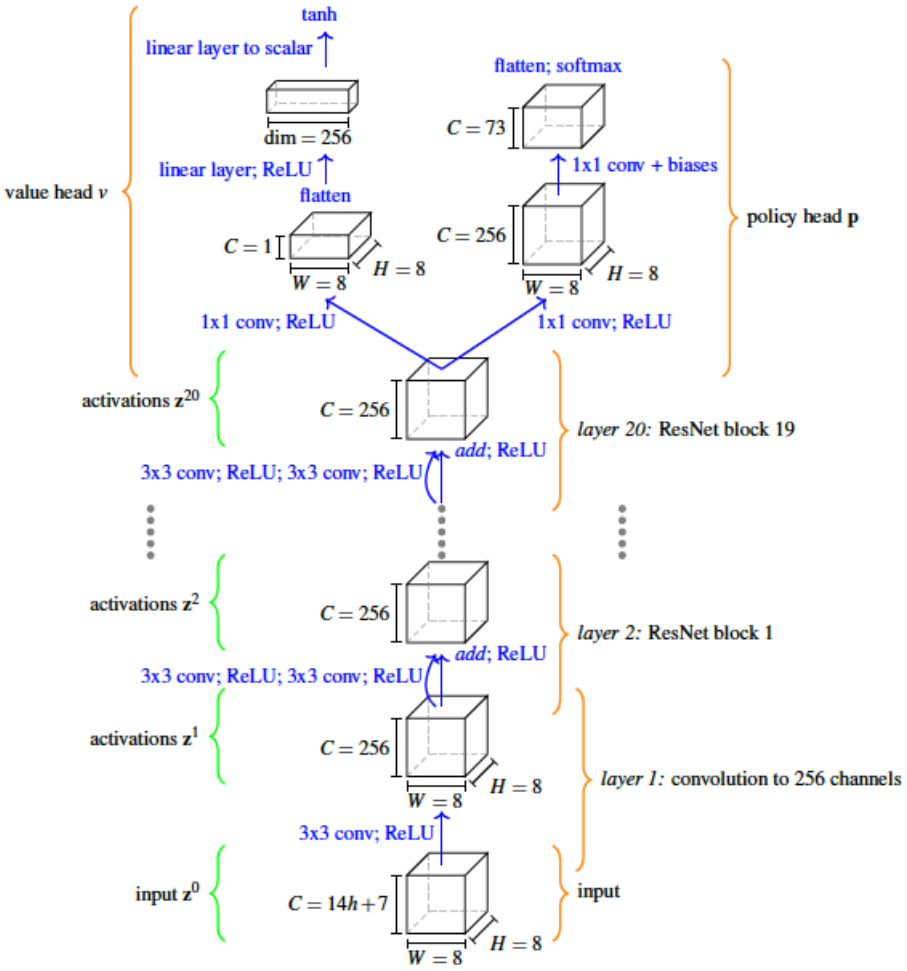


FIGURE 1 – Structure du réseau de neurones utilisé dans AlphaZero (schéma issue de l'article McGrath et al. [2021])

3.4 Recherche arborescente Monte-Carlo (MCTS)

Nous étudions dans cette partie la recherche arborescente Monte Carlo, en premier lieu sous sa forme générique puis sa version adaptée au sein d'AlphaZero.

3.4.1 Algorithme générique

L'algorithme MCTS (Monte Carlo Tree Search) est de façon générale découpée en 4 parties. Ces 4 parties, effectuées consécutivement, constituent une itération de l'algorithme. Pour aboutir à des résultats cohérents, il est nécessaire de performer un nombre d'itération suffisamment grand par rapport à

la complexité de l'arbre exploré, ceci explique pour quoi l'algorithme MCTS générique n'est pas viable dans le cas du jeu d'échecs : il s'agit d'un jeu trop complexe pour qu'une politique intéressante soit trouvée en un nombre raisonnable d'itérations.

L'arbre à explorer est constitué de noeuds associés à des positions sur l'échiquier contenant les informations suivantes : w la somme des gains obtenus en étant passé par ce noeud, et n le nombre de fois où le noeud a été visité. L'algorithme utilise ces informations pour construire une statistique qui lui permettra de choisir le noeud enfant à explorer. Ces deux quantités sont bien sûr initialement égales à zéro. Intéressons nous maintenant aux 4 phases caractéristiques de l'algorithme :

Phase 1 : Sélection

À chaque nouvelle itération, le départ est effectué de la racine de l'arbre (qui correspond à la position pour laquelle ou souhaite déterminer un coup nous donnant de bonnes chances de gagner la partie). Pour choisir quel noeud enfant va être parcouru, on calcule une statistique basée sur les quantités w et n définies ci-dessus. Cette statistique a pour but de fournir un équilibre entre l'exploitation des noeuds qui ont l'air "bons" (*i.e.* qui semblent mener vers une victoire), et l'exploration de noeuds jusqu'alors peu visités. Nous donnons à titre indicatif l'expression de la statistique la plus communément utilisée dans l'algoritme MCTS, nommée UCT (Upper Confidence Bound 1 applied to Trees) :

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln(\sum_j n_j)}{n_i}}$$

Avec w_i et n_i correspondant aux informations du i-ème du noeud dans lequel on se trouve, et c une constante qui quantifie a quel point on favorise l'exploration par rapport à l'exploitation. Le premier terme de la somme est celui qui quantifie la qualité du noeud, le second exprime quant à lui à quel point il a été exploré par rapport aux autres noeuds enfants.

Tant que que l'on a pas atteint une feuille (un noeud sans enfants), on choisit le noeud suivant à explorer en prenant le noeud enfant ayant la plus grande valeur de la statistique choisie (par exemple UCT).

Phase 2 : Expansion

Une fois arrivé dans une feuille la phase de sélection se termine, et celle d'expansion commence. En utilisant les règles du jeu, on "crée" des noeuds enfants qui correspondent au suites possibles de la partie. On choisit alors

selon une règle par défaut quel noeud enfant visiter en premier puisque la statistique UCT n'est pas encore calculable.

Phase 3 : Simulation

On arrive alors dans la phase de simulation, qui consiste à continuer la partie à partir de la position correspondant au noeud enfant choisit en phase d'expansion. La politique pour mener le jeu peut être aléatoire tout comme elle peut être définie par l'utilisateur. La partie continue alors jusqu'à aboutir à une issue à laquelle un score est associé.

Phase 4 : Rétropropagation

La dernière phase de l'algorithme MCTS constitue simplement à faire remonter l'issue de la partie à travers les noeuds parcourus. Cela consiste à incrémenter la quantité w de chaque noeud parcouru avec le score correspondant à l'issue de la partie simulée ainsi qu'à ajouter 1 au nombre de fois où ils ont été visités.

Le schéma ci-dessous récapitule graphiquement les 4 phases caractéristiques d'une itération de l'algorithme de recherche arborescente Monte Carlo.

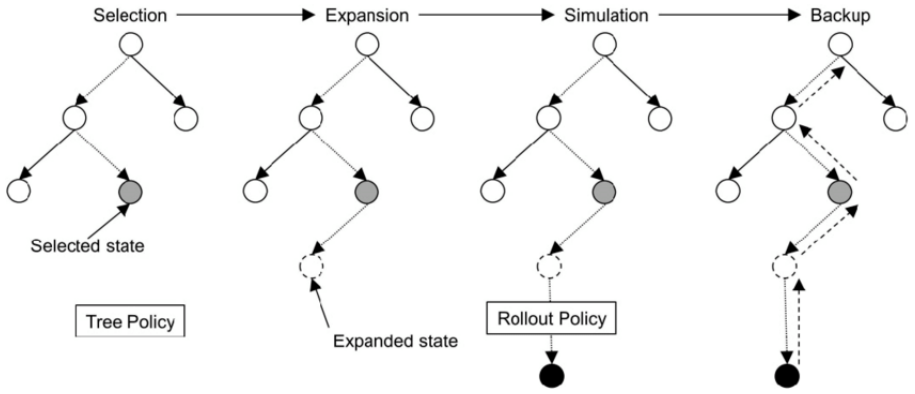


FIGURE 2 – Description d'une itération de l'algorithme de recherche arborescente Monte Carlo (MCTS), schéma issu de l'article Duarte et al. [2020]

3.4.2 Algorithme employé dans AlphaZero

L'algorithme précédent est légèrement modifié dans le cas d'AlphaZero afin de prendre en compte l'expertise provenant du réseau de neurones. Ce dernier a en effet pour objectif de renseigner sur les bons noeuds enfants à emprunter dans la phase de sélection, et permet aussi d'éviter la phase de simulation. On peut donc résumer cette partie d'AlphaZero en trois parties :

Les noeuds constituant l'arbre à parcourir contiennent une information supplémentaire, à savoir la probabilité d'avoir emprunté un noeud enfant sachant que l'on était dans un noeud parent. On notera cette probabilité p_i pour le i-ème noeud enfant du noeud considéré.

Phase 1 : Sélection

Le principe de cette étape est exactement le même que dans le cas de l'algorithme générique. La seul différence tient au fait que la statistique permettant de choisir quel noeud enfant va être parcouru est modifiée pour prendre en compte les informations issues du réseau des neurones. La nouvelle statistique est la suivante :

$$\frac{w_i}{1+n_i} + cp_i \frac{\sqrt{\sum_j n_j}}{1+n_i}$$

On voit désormais que le terme d'exploration est proportionnel à la probabilité de choisir le noeud enfant i sachant que l'on est dans son noeud parent.

Phase 2 : Expansion / Évaluation

Une fois dans une feuille, les noeuds enfants sont crées de la même manière que dans l'algorithme générique. Cependant, au lien de passer à la phase de simulation et de mener une partie jusqu'à son terme à l'aide d'une politique par défaut, on utilise simplement la sortie du réseau de neurones pour évaluer la position. Les probabilités issues du réseau de neurones sont assignées aux noeuds enfants correspondant et on récupère par ailleurs l'évaluation v de la position.

Phase 3 : Rétropropagation

Durant cette phase, on fait remonter l'évaluation de la position à travers les noeuds parcourus pendant l'itération en cours. Il s'agit simplement d'incrémenter les valeurs w des noeuds parcourus avec l'évaluation v de la position donnée par le réseau de neurones.

Le schéma ci-dessous récapitule graphiquement les 3 phases caractéristiquesd'une itération de l'algorithme de recherche arborescente Monte Carlo adapté pour AlphaZero. ‘

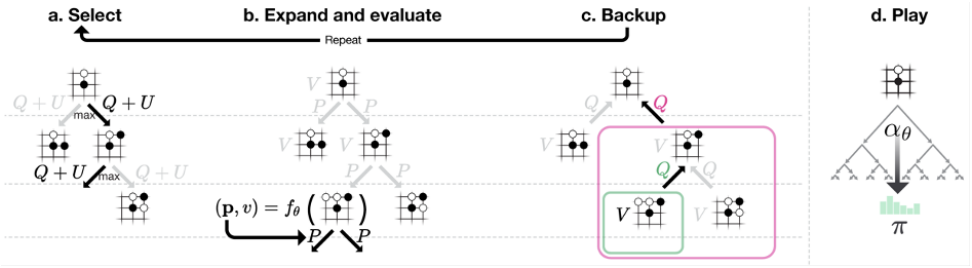


FIGURE 3 – Description d'une itération de l'algorithme de recherche arborescente Monte Carlo (MCTS) utilisé dans AlphaGo Zero et AlphaZero (schéma issue de l'article Silver et al. [2017a])

Notons par ailleurs qu'un bruit est ajouté aux probabilités a priori lors du calcul de la statistique de choix du noeud enfant à parcourir. Dans le cas particulier des échecs, le bruit est simulé par une loi de Dirichlet de paramètre $\alpha = 0.3$. Ce bruitage a pour but de favoriser l'exploration de l'arbre. Enfin, après un certain nombre d'itérations de l'algorithme, la politique à la racine de l'arbre π est finalement calculée de la façon suivante :

$$\pi_i = \left(\frac{n_i}{\sum_j n_j}\right)^{\frac{1}{\tau}}$$

Le paramètre n_i est le nombre de fois où le i-ème enfant de la racine a été visité, et τ est un paramètre dit de "température" qui contrôle le niveau d'exploration. La politique π ainsi que la position correspondant à la racine de l'arbre serviront comme données d'entraînement lors de la phase d'apprentissage du réseau de neurones.

3.5 Procédure d'entraînement

Comme expliqué précédemment, la procédure d'entraînement d'AlphaZero est uniquement basée sur de l'apprentissage par renforcement. Après avoir initialisée le réseau de neurone décrit en partie 3.3 avec des paramètres aléatoires, la démarche d'apprentissage peut être séparée en 2 phases caractéristiques : la phase de *self-play* et la phase d'entraînement du réseau de neurones.

3.5.1 Phase de *self-play*

La première étape constitue simplement à faire jouer l'agent contre lui même, en suivant le processus décrit dans l'algorithme MCTS adapté à AlphaZero. Après 800 itérations de l'algorithme MCTS, un coup est joué en fonction de la politique π obtenue. Cette opération est répétée jusqu'à arriver à l'issue de la partie, où l'on récupère de score z correspondant au résultat de la partie.

Si la partie a duré N coups, on obtient donc N données potentielles d'entraînement constituées d'une position s (entrée du réseau de neurones), d'un vecteur π contenant la politique correspondant à la position s, et de l'issue de la partie z qui est la même pour toutes les données issues de cette partie. En pratique les données sont échantillonnées pour ne pas que les données soient trop corrélées entre elles. Les données sélectionnées aléatoirement sont ajoutées au jeu de données d'entraînement. Lorsque le nombre de données d'entraînement dépasse un certain seuil les données les plus vieilles sont jetées.

3.5.2 Entraînement du réseau de neurones

Le réseau de neurones, que l'on peut simplement écrire $f_\theta(s) = (\mathbf{p}, v)$, est entraîné de façon à minimiser la fonction de perte l suivante :

$$l = (z - v)^2 - \pi^T \log(\mathbf{p}) + c||\theta||_2^2$$

Le paramètre c permet de contrôler le niveau de régularisation L^2. Une fois l'entraînement terminé on obtient un nouveau réseau $f_{\theta'}$.

Dans AlphaGo Zero, pour tester si la nouvelle séquence d'entraînement a amélioré le modèle, on fait jouer un nombre arbitraires de parties opposant le programme avec le réseau de neurones f_θ face à celui avec le nouveau réseau candidat $f_{\theta'}$. Si plus de 55% des parties sont remportées par le programme avec le nouveau réseau il remplace l'ancien. Néanmoins, dans AlphaZero, le réseau de neurone est simplement mis à jour continuellement.

Le schéma ci-dessous récapitule graphiquement la méthodologie d'entraînement d'une instance d'AlphaZero.

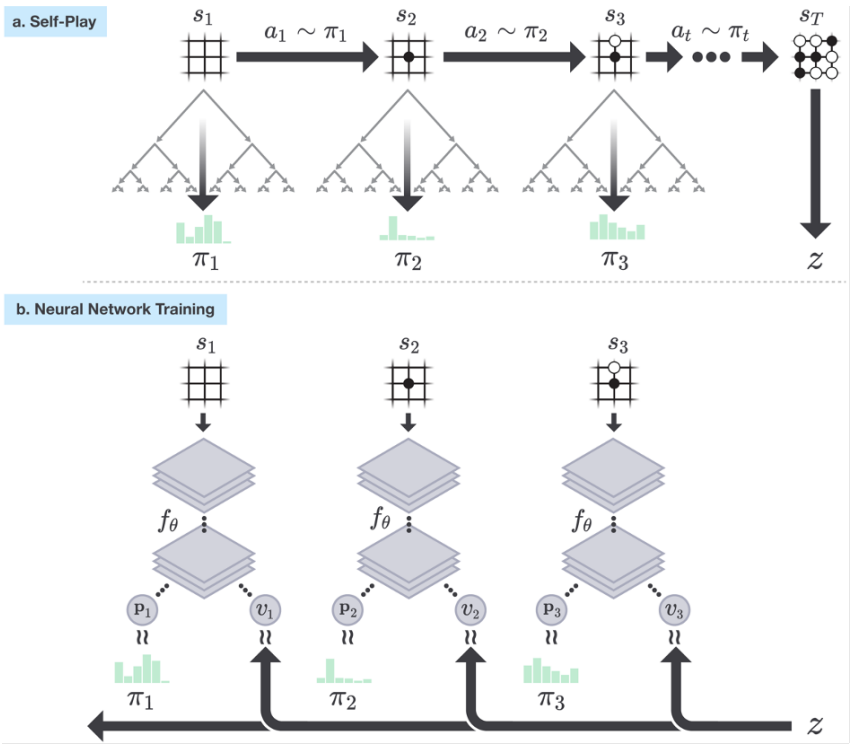


FIGURE 4 – Schéma de la stratégie d'entraînement d'AlphaZero (schéma issue de l'article Silver et al. [2017a])

4 Implémentation - MCTS

4.1 Généralités

Nous avons implémenté un algorithme de recherche arborescente Monte Carlo utilisant un réseau de neurones pré-entraîné. En effet, nous n'avons pas pu réaliser nous même l'algorithme complet mettent en place la stratégie d'apprentissage par renforcement pour des questions de puissance de calcul accessible. Le matériel utilisé par DeepMind pour entraîner le réseau de neurones est malheureusement bien plus conséquent que celui dont nous disposions.

Le réseau de neurones est issu du répertoire github disponible à l'adresse suivante https ://github.com/Zeta36/chess-alpha-zero. Il a été entraîné en suivant la méthodologie décrite dans les différents articles produits par DeepMind et disponibles dans les références de ce rapport.

Nous avons initialement codé une version générique de l'algorithme de recherche arborescente Monte-Carlo, mais avec des résultats très médiocres qui ne s'éloignaient guère des performances d'un agent jouant des coups aléatoires. Nous avons donc décidé d'utiliser un réseau de neurones pré-entraîné pour

pouvoir aboutir à des performances satisfaisantes. Nous avons porté une attention particulière à respecter les préconisations faites dans l'article Silver et al. [2017b], notamment sur la nécessité de rajouter du bruit sur les probabilités a priori fournies par le réseau.

Il est possible de faire jouer l'algorithme contre lui même ou contre un agent aléatoire, une étape de démonstration est fournie en fin de code. Malgré des performances très honorables en ouvertures et en milieu de jeu, nous avons remarqué que notre implémentation conduit souvent à des parties nulles malgré une superiorité matérielle écrasante. Nous n'avons pas réussi à comprendre l'origine ce problème.

4.2 Remarque importante pour l'exécution du code

Notre implémentation est en Python sous la forme d'un notebook Jupyter. **Il est impératif de tester le code sur Google collaboratory avec une instance GPU active. En effet, pour des raisons d'implémentations, le réseau de neurones pré-entraîné ne peut être évalué que si une instance GPU est active. Dans le cas contraire le code échouera.** Toutes les démarches permettant le bon fonctionnement du code sont expliqués dans le fichier texte *README.txt* contenu dans l'archive.

5 Conclusion

Ce projet nous a permis de prendre en main une méthodologie utilisant une stratégie d'apprentissage par renforcement. En parallèle de ces notions, nous avons pu nous pencher sur différents types d'algorithmes permettant d'explorer des arbres de grande dimension. Nous avons finalement pu implémenter l'algorithme de recherche arborescente Monte Carlo dans le même esprit que celui décrit dans l'article *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm* (Silver et al. [2017b]).

Actuellement, les principales recherches sur la méthodologie développée pour AlphaZero portent sur la notion d'interprétabilité. Ce point est particulièrement étudié dans l'article récemment produit par DeepMind *Acquisition of Chess Knowledge in AlphaZero* (McGrath et al. [2021]), pour lequel le célèbre joueur d'échec Vladimir Kramnik a notamment apporté sa contribution.

Références

Fernando Duarte, Nuno Lau, Artur Pereira, and Luís Reis. A survey of planning and learning in games, 2020.

Thomas McGrath, Andrei Kapishnikov, Nenad Tomasev, Demis Pearce, Adam Hassabis, Been Kim, Ulrich Paquet, and Vladimir Kramnik. Acquisition of chess knowledge in alphazero, 2021.

Claude Shannon. Programming a computer for playing chess, 1950.

D. Silver, J. Schrittwieser, and K. et al Simonyan. Mastering the game of go without human knowledge, 2017a.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017b.