L'IA championne des jeux d'échecs et de Go (partie 2)

Franco, Nicolas

Published in:
Losanges

Publication date:
2025

Document Version
le PDF de l'éditeur

L'IA championne des jeux d'échecs et de Go (2)

Nicolas Franco

Mots clés : Intelligence artificielle, automate d'échecs et de Go, recherche arborescente Monte-Carlo, réseaux de neurones, descente de gradient.

Résumé. *Cet article en deux parties explore l'évolution historique de l'intelligence artificielle et des concepts mathématiques sous-jacents, en se concentrant sur la théorie des jeux comme étalon dans le développement des IA. Lors de cette seconde partie, nous montrons comment l'introduction de l'aléatoire puis par la suite des réseaux de neurones artificiels a permis de créer une nouvelle génération de programmes capables de surpasser les humains aux échecs et au jeu de Go. Nous explorons un peu les mathématiques derrière la notion de réseau de neurones artificiels.*

1. Introduction

Dans le Losanges 65, nous avions développé la construction historique d'ordinateurs capables de battre les plus grands joueurs d'échecs sur la période 1945–2015. Pour cela, nous avions montré comment la théorie des graphes, et en particulier le concept d'arbre *min-max*, permet de modéliser un jeu à deux joueurs, à tour de rôle, à somme nulle et à information parfaite. La racine d'un tel arbre correspond à l'état actuel du jeu et les fils successifs correspondent à l'ensemble des coups autorisés. À chaque sommet est alors attribué une valeur correspondant à l'évaluation de l'état de jeu, calculée comme le min ou max de ses fils (le joueur dont c'est le tour cherchant à maximiser ou minimiser l'évaluation) ou utilisant une fonction d'évaluation bien précise dans le cas d'une feuille. L'exploration de l'arbre était effectuée par une méthode de backtracking jusqu'à une certaine profondeur donnée. La technique pouvait alors être améliorée en utilisant un élagage alpha-bêta, des tables de finales ou de transposition, etc.

Si cette méthode a porté ses fruits pour les IA d'échecs, produisant jusque récemment les meilleurs programmes au monde, pour le jeu de Go les performances n'ont pas été si fructueuses, en raison du nombre excessivement important de possibilités de jeu à explorer (plus de 300 fils à chaque coup, contre environ 38 pour les échecs), mais également de la très grande difficulté à créer une fonction d'évaluation pertinente, car pour le jeu de Go le nombre de pièces n'a pas une grande importance par rapport à leurs positions respectives.

Ainsi il a fallu attendre de nouvelles techniques avant d'avoir une IA performante pour le jeu de Go. Ces techniques sont la recherche arborescente Monte-Carlo (apparue vers 2005) puis l'introduction des réseaux de neurones artificiels (vers 2015). Le programme initial utilisant ces techniques est appelé AlphaGo et sa progression est fulgurante. Il est par la suite amélioré en AlphaGo Zero et enfin AlphaZero permettant de jouer à d'autres jeux comme les échecs (avec ses nombreux programmes dérivés comme MuZero ou Leela Chess Zero, ce dernier étant open source).

Cette seconde partie de l'article présente les nouvelles techniques utilisées par AlphaGo/AlphaZero ainsi que certains concepts mathématiques bien connus qui se cachent derrière.

2. La recherche arborescente Monte-Carlo

L'idée naïve de la recherche arborescente Monte-Carlo (ou *Monte Carlo Tree Search* – MCTS) est assez simple : remplacer l'évaluation d'une branche à une certaine profondeur, qui auparavant se faisait à l'aide d'une fonction d'évaluation principalement

basée sur le nombre de pièces, par la simulation d'un certain nombre de « fins de partie aléatoires », que nous appellerons simulations. Le résultat espéré est que, si un coup mène plus souvent qu'un autre à la victoire lors de ces fins de partie aléatoires, alors il devrait être choisi.

Pour plus de clarté, nous allons considérer que l'arbre total des possibilités de jeu est divisé en deux parties :

- Une partie déjà « visitée », constituée des coups ou successions de coups qui ont déjà été envisagés, les derniers sommets de ce sous-arbre étant appelés abusivement feuilles dans cette méthode (même s'ils ne correspondent pas nécessairement à une situation de fin de partie) ;
- Une partie « non visitée », sauf par des simulations, donc une continuation du sous-arbre précédent mais non connue exhaustivement et seulement parcourue par des fins de partie jouées totalement aléatoirement.

Sur la partie de l'arbre déjà visitée, nous allons considérer deux statistiques sur chaque sommet s :

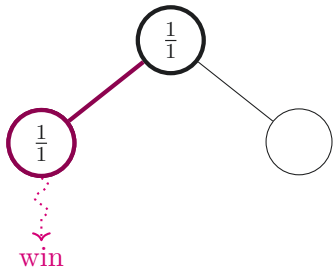
- $Q(s)$, appelé Total simulation reward, est la somme des résultats des simulations qui ont été obtenues en passant préalablement par le sommet s ;
- $N(s)$, appelé Total number of visits, est le nombre de simulations qui ont été effectuées en passant préalablement par le sommet s.

Si nous supposons que les résultats des simulations sont 1 (victoire du joueur 1), -1 (victoire du joueur 2) et 0 (match nul), alors le signe de $Q(s)$ indique si le passage par le sommet s mène plus probablement à une victoire ou une défaite, du moins sur base du nombre limité de simulations effectuées, et le ratio $\frac{Q(s)}{N(s)}$ se rapproche de 1 lorsque la probabilité de gagner est forte (note : il est également possible d'utiliser la convention 1-0-0.5). Le ratio $\frac{Q(s)}{N(s)}$ est donc une évaluation moyenne du sommet en question, remplaçant la fonction d'évaluation.

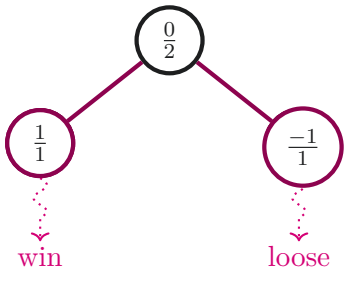
L'algorithme Monte Carlo Tree Search consiste à répéter la procédure suivante à partir de la racine (correspondant à l'état actuel du jeu), et comprenant quatre phases :

- Sélection : Depuis la racine, on construit un chemin parmi des sommets déjà visités en choisissant à chaque fois un fils à l'aide d'une fonction de sélection, jusqu'à atteindre un sommet dont l'expansion n'est pas complète (un sommet dont l'expansion est complète est un sommet dont tous les fils ont été marqués comme visités). Ce sommet est appelé *feuille*.
- Expansion : Si cette feuille n'est pas finale (i.e. une réelle feuille de l'arbre correspondant à une fin de partie), alors on la considère comme sommet d'expansion (*expansion node*) : on construit ses fils en utilisant les règles du jeu, et on choisit aléatoirement un de ces fils, noté s. On fixe $Q(s) = 0$ et $N(s) = 0$.
- Simulation : On crée une simulation depuis le fils s, c'est-à-dire une fin de partie totalement aléatoire depuis ce sommet. Le sommet s est alors considéré comme visité.
- Backpropagation : On met à jour les valeurs Q et N en reprenant le chemin à l'envers depuis s jusqu'à la racine, en augmentant Q de 1 si la simulation a mené à une victoire et en diminuant de 1 si défaite (0 si match nul), et en augmentant N de 1 dans tous les cas.

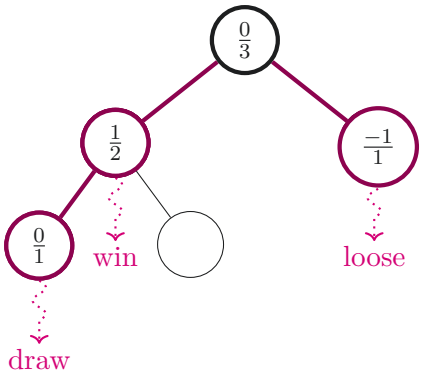
Exemple : La racine possède deux fils, une simulation est lancée à partir du premier fils, qui est alors considéré comme visité et possède l'évaluation moyenne $\frac{1}{1}$.



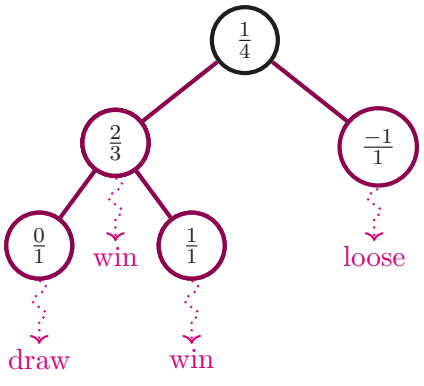
Le second fils est visité, lançant une seconde simulation.



Le premier fils est considéré comme sommet d'expansion, et une simulation est lancée à partir du premier petit-fils. Le score est alors propagé en arrière au niveau des parents.



Le second petit-fils modifie encore l'évaluation moyenne de ses parents.



On peut remarquer que la particularité de cet algorithme est qu'une simulation est toujours effectuée depuis un sommet qui n'a jamais été « visité », ce qui veut dire que chaque sommet est le point de départ d'une simulation une fois au maximum. Au début de l'algorithme, ce sont évidemment les fils non visités de la racine qui sont choisis en premier, après cela la racine est considérée comme ayant eu une expansion complète, et les simulations suivantes démarreront de profondeurs plus importantes.

Un des intérêts principaux de l'algorithme MCTS est la façon dont les petits-fils et descendants suivants sont choisis. En effet, il n'est pas opportun d'explorer et de lancer des simulations depuis tous les descendants profondeur par profondeur. Si une branche de l'arbre mène à des simulations aux résultats très négatifs, il n'est guère intéressant de l'explorer beaucoup plus loin, à la façon d'un cerveau humain qui ne s'amuserait pas à perdre du temps à tester de nombreux coups supplémentaires à partir d'un très mauvais coup. S'il y a des coups déjà identifiés comme assez prometteurs, il est intéressant d'étudier un peu plus en détail cette partie de l'arbre. Cependant, il est préférable de ne pas ignorer totalement les mauvaises branches, car les mauvais scores ne sont que le résultat de quelques fins de partie aléatoires et l'algorithme pourrait alors passer à côté de réels coups prometteurs.

Un bon compromis doit donc être trouvé entre :

- l'exploitation : qui consiste à tester davantage vers des fils ou descendants qui paraissent prometteurs ;
- l'exploration : qui consiste à tester des fils ou descendants qui ont été moins explorés.

L'algorithme utilise alors une fonction de sélection qui permet de choisir parmi les sommets à l'expansion complète le prochain qui servira de sommet d'expansion. La fonction choisie s'appelle UCT (pour « Upper Confidence Bound applied to Trees ») et est définie de la façon suivante, pour un fils s_i d'un parent s :

$$UCT(s_i, s) = \frac{Q(s_i)}{N(s_i)} + c\sqrt{\frac{\ln N(s)}{N(s_i)}}$$

où nous avons en fait que $N(s) = \sum_i N(s_i)$ sur tous les fils s_i de s.

Cette fonction s'interprète de la façon suivante :

- Le premier terme correspond à l'évaluation moyenne du sommet obtenue jusqu'ici, c'est donc l'information relative à l'exploitation.
- Le second terme correspond lui à l'exploration. Il est très élevé lorsque peu de simulations ont été effectuées à partir de chemins initiaux utilisant le choix du sommet s_i (infini lorsque le sommet n'a pas encore été visité).

Le paramètre c est choisi de façon à favoriser davantage l'exploitation ou l'exploration. Sa valeur théorique est $c = \sqrt{2}$.

Après répétition de la procédure en fonction du temps alloué ou d'un nombre de répétitions fixé, le fils de la racine qui est choisi est celui qui a le meilleur score moyen $\frac{Q(s_i)}{N(s_i)}$. Notez que les informations Q et N calculées peuvent être réutilisées lors du coup suivant, en restreignant l'arbre exploré au sous-arbre dont la racine est le nouvel état du jeu. Sous la supposition d'un temps et d'une capacité mémorielle infinie, l'algorithme MCTS converge de façon théorique vers le même résultat que l'algorithme *min-max*.

3. L'ajout d'un réseau de neurones artificiels

Si l'utilisation du MCTS a permis entre 2005 et 2016 une progression très importante des pro-

grammes de Go, permettant de dépasser le simple stade amateur, le niveau atteint n'est pas encore celui des plus grands professionnels. En effet, l'algorithme présente deux principales faiblesses : chaque simulation effectuée jusqu'au bout prend du temps et ne représente qu'une unique possibilité de jeu peu représentative de l'évaluation réelle, et la partie exploration pourrait être davantage guidée en fonction de connaissances antérieures.

Ces limitations ont conduit à l'intégration de réseaux de neurones à l'algorithme MCTS. L'idée intuitive est ici de créer un « cerveau » qui s'améliorerait au fil du temps, et qui serait capable à partir d'une situation donnée (parent s avec des fils s_i) de donner deux valeurs bien précises :

- une valeur $v(s)$ donnant une évaluation de la position s et appelée value network (score entre -1 et 1) ;
- nn vecteur $P(s_i)$, donnant pour chaque fils de s une valeur déterminant si le choix de ce fils comme prochain coup est prometteur ou non et appelé policy network (probabilités entre 0 et 1 normalisées sous forme de distribution).

La valeur $v(s)$ est ici un remplacement de la valeur résultante d'une simulation. L'idée est donc d'utiliser les connaissances préalables du « cerveau » plutôt que de jouer des longues fins de parties. Ainsi les différents $Q(s)$ sont calculés non plus par Backpropagation de la valeur $1, -1, 0$ renvoyée par une simulation, mais par Backpropagation de la valeur $v(s) \in [-1, 1]$ depuis le fils s actuellement visité. Les valeurs $P(s_i)$ permettent de davantage guider la partie exploration, en utilisant la variante suivante de la fonction de sélection :

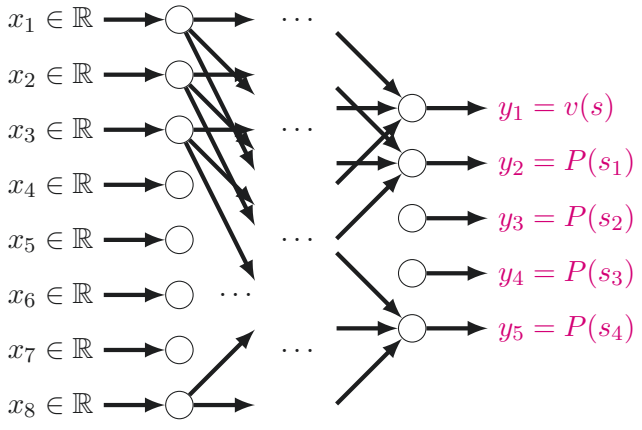
$$U\hat{C}T(s_i, s) = \frac{Q(s_i)}{N(s_i)} + \sqrt{2}\, P(s_i)\, \frac{\sqrt{N(s)}}{1 + N(s_i)}.$$

En résumé, $v(s)$ est une indication du fait que l'état du jeu s est bon ou non alors que $P(s_i)$ est une distribution de probabilité signalant s'il est préférable d'effectuer tel ou tel mouvement à partir de l'état s.

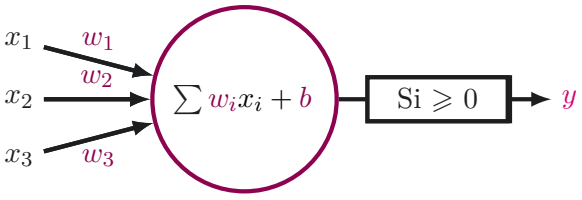
3.1. L'idée d'un cerveau artificiel mathématique

Un réseau de neurones artificiels est un système inspiré du fonctionnement du cerveau et des neurones biologiques. Un neurone biologique reçoit en entrée des signaux électriques (influx nerveux) via des *dendrites*. Ces signaux transitent ensuite par le corps cellulaire et un signal de sortie est envoyé via l'*axone* jusqu'aux terminaisons nerveuses, qui vont permettre de transmettre l'information aux neurones suivants. Le signal de sortie dépend des signaux d'entrée. Un cerveau est composé de plusieurs milliards de neurones connectés les uns aux autres.

Mathématiquement (et informatiquement), il est possible de mimer ce fonctionnement, avec une structure recevant des signaux sous forme de nombres réels via plusieurs entrées et produisant en sortie plusieurs nombres réels. Dans le cas de notre algorithme MCTS, les entrées correspondent à l'information sur l'état du jeu s, ce qui peut se faire facilement en imaginant une entrée pour chaque case du jeu et un codage pour le pion ou la pièce présente. En sortie, nous nous attendons à ce que notre cerveau produise les quantités $v(s)$ et $P(s_i)$.



L'élément de base est appelé neurone artificiel ou neurone formel. Il prend en entrée un certain nombre de réels x_i qui vont être combinés à l'aide de poids w_i. Comme un neurone biologique ne réagit que si les signaux d'entrées sont suffisamment importants, nous allons décider que notre neurone artificiel ne produira une sortie que si le résultat du calcul $\sum_i x_i w_i$ dépasse un certain seuil $-b$, appelé biais. Notons que cela revient à demander que le calcul $\sum_i x_i w_i + b$ dépasse la valeur zéro, voire à intégrer ce biais aux poids sous la forme d'un $w_0 = b$ associé à une entrée supplémentaire constante $x_0 = 1$.



En pratique, le test de positivité est effectué via une fonction particulière appelée fonction d'activation φ. Pour des raisons qui paraitront évidentes par la suite, nous allons exiger que cette fonction soit dérivable, plutôt que d'utiliser la fonction de Heaviside qui renverrait 0 ou 1 mais serait discontinue. Ainsi des fonctions de type sigmoïde $\left(\varphi(x) = \dfrac{1}{1+\mathrm{e}^{-x}}\right)$ ou tangente hyperbolique sont privilégiées, car elles permettent de renvoyer un signal dans les intervalles désirés ($[0,1]$ ou $[-1,1]$) avec une valeur se rapprochant du minimum lorsque le résultat du calcul est négatif et se rapprochant du maximum lorsque le résultat du calcul est positif.

Au final, d'un point de vue mathématique, un neurone artificiel est simplement une fonction de type :

$$\begin{aligned} y &= \varphi(w_1x_1 + w_2x_2 + w_3x_3 + b) \\ &= \varphi(w_1x_1 + w_2x_2 + w_3x_3 + w_0 1) \end{aligned}$$

ou plus simplement sous une formulation vectorielle :

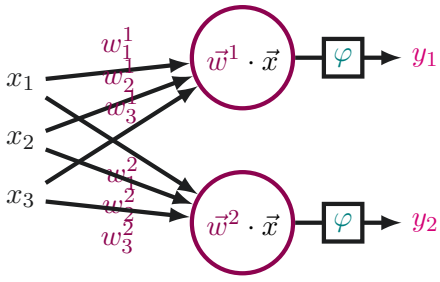
$$\vec{w} = (w_0, w_1, w_2, w_3), \quad \vec{x} = (1, x_1, x_2, x_3),$$

$$y = \varphi(\vec{w} \cdot \vec{x}).$$

Cependant, un seul neurone ne permet de renvoyer qu'une valeur unique, et nous avons besoin de plusieurs sorties. La construction suivante est une couche de neurones, qui consiste à mettre un certain nombre de neurones en parallèle, utilisant les mêmes entrées $\vec{x} = (x_1, \ldots, x_n)$ et produisant chacun une sortie, ce qui donne une sortie totale $\vec{y} = (y_1, \ldots, y_m)$ (le nombre de sorties ne doit pas nécessairement égaler le nombre d'entrées). La fonction d'activation est la même partout, mais les poids et le biais sont différents.

On trouve donc un système de type (la fonction φ étant appliquée sur chaque composante) :

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \varphi\left(\begin{pmatrix} \vec{w}^1 \\ \vec{w}^2 \end{pmatrix} \cdot \vec{x}\right),$$

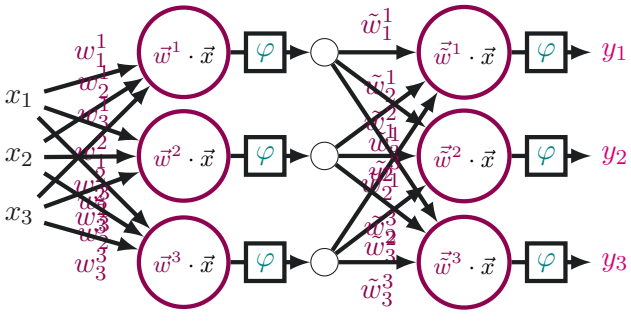


ou sous formulation matricielle :

$$\vec{y} = (y_1, y_2), \qquad W = \begin{pmatrix} \vec{w}^1 \\ \vec{w}^2 \end{pmatrix},$$

$$\vec{y} = \varphi(W\vec{x}).$$

L'étape ultime est d'utiliser le fait que les sorties y_i peuvent correspondre aux entrées x_i d'autres couches de neurones, à la façon du cerveau humain. Ainsi un réseau de neurones multicouche est simplement la composition en série de plusieurs couches de neurones :



soit sous forme mathématique :

$$\vec{y} = \varphi(W^{(n)} \cdot \varphi(W^{(n-1)} \cdot \cdots \varphi(W^{(2)} \cdot \varphi(W^{(1)}\vec{x})))).$$

Un réseau de neurones artificiels est donc simplement une immense fonction presque linéaire composée d'un très grand nombre de coefficients (les poids) qui doivent être ajustés de façon à produire les sorties voulues. La façon dont ces poids sont ajustés sera présentée à la fin de cet article. En fonction de la taille du réseau, le nombre de poids peut être extrêmement important (plusieurs centaines de milliards). Notre qualification de « presque linéaire » vient du fait que la non-linéarité est seulement due à la présence de la fonction d'activation φ. En l'absence de cette fonction, un réseau de neurones se résumerait simplement à :

$$\vec{y} = W^{(n)}W^{(n-1)} \ldots W^{(1)}\vec{x},$$

soit un simple système linéaire où toutes les matrices pourraient être multipliées entre elles de façon à n'obtenir qu'une matrice unique de la taille des entrées et sorties finales, ce qui réduirait très fortement le nombre de paramètres. Le découpage en couches séparées par cette fonction d'activation permet donc à un réseau de neurones de devenir fortement *surdimensionné* en terme de nombre de paramètres par rapport au nombres d'entrées et de sorties.

3.2. Les différents entrainements du programme AlphaZero

Revenons à notre programme AlphaZero et montrons comment la notion de réseau de neurones peut être combinée à l'algorithme MCTS. Pour cela, nous allons supposer que nous possédons un réseau de neurones artificiels prenant en entrée un état du jeu s et donnant en sortie le scalaire $v(s)$ et le vecteur $P(s_i)$, censés représenter l'évaluation moyenne du sommet s et les probabilités de sélection de chacun de ses fils. Nous supposons également que nous avons une méthode extrêmement rapide pour ajuster les poids du réseau de neurones de façon à produire les sorties voulues.

La particularité du programme AlphaZero est qu'il n'utilise pas un réseau de neurones, mais plusieurs. Le programme va partir d'un réseau « bidon », où les poids ont été générés au hasard, et donc retournant des valeurs $v(s)$ et $P(s_i)$ au hasard (respectant tout de même les contraintes $v(s) \in [-1, 1]$ et $\sum_i P(s_i) = 1$), puis va en créer un nouveau qui sera un peu amélioré, et ainsi de suite.

AlphaZero va utiliser trois processus d'entrainements différents pour créer le nouveau réseau à partir du premier. Pour les comprendre, nous allons supposer que nous avons à notre disposition un réseau de neurones déjà assez performant (donc renvoyant des valeurs $v(s)$ et $P(s_i)$ assez bonnes), et nous essayons d'en créer un second encore meilleur.

Tout d'abord, nous avons un processus d'entrainement assez rapide (∼0.4s) qui permet, à partir d'un sommet s, de générer une distribution de probabilités sur l'ensemble de ses fils, permettant de choisir le coup suivant. Ce processus consiste à effectuer un Monte Carlo Tree Search comme décrit précédemment, mais avec les différences notables suivantes :

- Pour la partie Sélection, la variation $U\hat{C}T(s_i, s) = \frac{Q(s_i)}{N(s_i)} + \sqrt{2}\ P(s_i)\ \frac{\sqrt{N(s)}}{1+N(s_i)}$ est utilisée, où $P(s_i)$ est donné par notre réseau de neurones. Ainsi l'exploration est davantage guidée par la connaissance déjà présente dans notre réseau de neurones, afin de favoriser les meilleurs coups.
- Pour la partie Simulation, lorsqu'un nouveau sommet s est visité, plutôt que d'effectuer une simulation jusqu'au bout de la partie (ce qui est très approximatif, car représentant une seule fin de partie aléatoire), l'algorithme va simplement prendre la valeur $Q(s) = v(s)$ donnée par le réseau de neurones, sauf si s est une feuille finale, dans ce cas on lui donne la valeur 1, 0, ou −1. C'est cette valeur $v(s)$ qui sera alors propagée lors de la Backpropagation.

Ce processus va être répété 1600 fois, et va permettre de générer une nouvelle distribution de probabilités en comptabilisant combien de fois chaque fils du sommet s a été choisi lors de ces 1600 répétitions, soit :

$$P'(s_i) = \frac{N(s_i)}{N(s)},$$

ce qui respecte bien $\sum_i P'(s_i) = 1$. Ces $P'(s_i)$ représentent (peut-être) une amélioration des $P(s_i)$ donnés initialement par le réseau de neurones, vu que l'on suppose que la fonction de sélection $U\hat{C}T$ conduit plus souvent le MCTS à passer par des sommets prometteurs (exploitation).

À partir de ce processus de détermination des $P'(s_i)$ améliorés, AlphaZero va pouvoir lancer un processus d'auto-entrainement : le programme effectue un grand nombre de parties (25 000) depuis la racine en jouant au hasard, mais en utilisant à chaque sommet s le processus MCTS précédent et les $P'(s_i)$ obtenus comme probabilité de choix à chaque coup. L'idée de jouer au hasard en suivant ces probabilités est d'éviter de jouer 25 000 fois des parties similaires dans le cas où seul le fils le plus probable serait choisi. Cette fois-ci, les parties sont jouées jusqu'au bout et de nouvelles évaluations moyennes sont obtenues via

$$v'(s) = \frac{Q(s)}{N(s)},$$

du moins pour les sommets parcourus lors des parties. À la fin, les différent résultats obtenus (nouveaux $v'(s)$ et $P'(s_i)$) sont utilisés pour entrainer

un nouveau réseau de neurones qui devra donner les nouvelles valeurs de sorties améliorées pour chaque sommet s.

Mais ces deux processus ne sont pas suffisants. En effet, si le premier réseau de neurones est un réseau « bidon » dont les sorties sont totalement aléatoires, rien ne dit que les parties générées vont être intéressantes (car basées sur des $v(s)$ et $P(s_i)$ au hasard) et que le nouveau réseau de neurones créé rendra réellement des valeurs améliorées. AlphaZero va alors passer par un processus d'entrainement supplémentaire en jouant contre lui-même 400 parties en utilisant le nouveau réseau de neurones contre l'ancien. Si le nouveau gagne un nombre suffisant de parties (55%), alors il est choisi, sinon l'ancien est conservé. Les tout premiers réseaux de neurones créés ainsi sont assez mauvais mais, progressivement, ils commencent à s'améliorer.

Le processus de création d'un nouveau réseau de neurones est alors répété un certain nombre de fois (au moins 200), et le réseau final obtenu est prêt à battre tous les humains du monde entier.

Sachant que la création de chacun de ces 200 réseaux de neurones a nécessité 25 000 auto-parties dont chaque coup nécessitait 1 600 MCTS ainsi que 400 parties de test, nous laissons au lecteur le soin d'estimer le nombre total de simulations effectuées ainsi que le temps total requis (qui se mesure en dizaines de jours sur des supercalculateurs). Notons que les détails et valeurs présentés ici ne correspondent qu'à une version particulière du programme et que ses variantes et adaptations ont chacune leurs spécificités.

4. Les mathématiques derrière l'IA

Nous avons présenté la notion de réseau de neurones artificiels et sa capacité à fournir des sorties bien précises en fonction des entrées, mais nous n'avons pas encore expliqué comment ajuster les fameux poids afin de garantir la pertinence des signaux de sortie. En particulier pour nos programmes d'échecs et de Go, il convient d'améliorer constamment le réseau de neurones de façon à ce que pour chaque sommet s connu les valeurs de sorties $v(s)$ et $P(s_i)$ soient les plus proches possibles des valeurs cibles calculées $v'(s)$ et $P'(s_i)$, et ce de façon particulièrement rapide vu que le processus doit être répété un grand nombre de fois.

Nous allons regrouper nos sorties dans un vecteur $\vec{y} = (v(s), P(s_i))$ et nos valeurs cibles dans un autre vecteur $\vec{t} = (v'(s), P'(s_i))$. Les entrées sont regroupées dans un vecteur $\vec{x}$ qui n'est autre qu'un codage d'un état ou sommet s. Notre rôle est de minimiser une certaine distance entre les vecteurs $\vec{y}$ et $\vec{t}$ pour chaque entrée $\vec{x}$. Cette distance est appelée fonction objectif ou loss function L. Un choix classique pour cette distance est d'utiliser le principe des *moindres carrés*, soit

$$L = \frac{1}{2}\left\|\vec{y} - \vec{t}\right\|^2.$$

AlphaZero utilise en réalité une fonction légèrement différente combinant moindres carrés pour $v(s)$ et *entropie croisée* pour $P(s_i)$ car ce dernier représente une distribution, soit :

$$L = \sum_s \left\|v(s) - v'(s)\right\|^2 - \sum_{s_i} \ln(P(s_i))P'(s_i),$$

mais nous poursuivrons notre raisonnement en nous limitant aux moindres carrés pour plus de simplicité.

Le but est donc de trouver le minimum de la fonction objectif L dépendante du vecteur de sortie $\vec{y}$, lui-même dépendant du vecteur d'entrée $\vec{x}$ via :

$$\vec{y} = \varphi(W^{(n)} \cdot \varphi(W^{(n-1)} \cdot \ldots \varphi(W^{(2)} \cdot \varphi(W^{(1)}\vec{x})))).$$

Mais comme les entrées $\vec{x}$ sont connues (états du jeu) et que la fonction d'activation φ est préalablement choisie, les seuls éléments réellement inconnus sont les poids w_i composant les matrices $W^{(n)}$, …, $W^{(1)}$ pour chaque couche de neurones. Notre problème revient donc à déterminer les w_i donnant le minimum global de la fonction L.

La technique utilisée est une technique itérative connue sous le nom de descente de gradient. Lorsque l'on cherche un minimum (local) d'une fonction de plusieurs variables $f(x_1, x_2, \ldots, x_n)$ différentiable, nous pouvons utiliser l'information des dérivées de cette fonction pour se rapprocher de ce minimum. Une fonction de n variables possède en effet n dérivées partielles $\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}$ (dérivées suivant chaque variable isolée) qui sont regroupées sous forme d'un vecteur appelé gradient :

$$\nabla f(x_1, x_2, \ldots, x_n) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right).$$

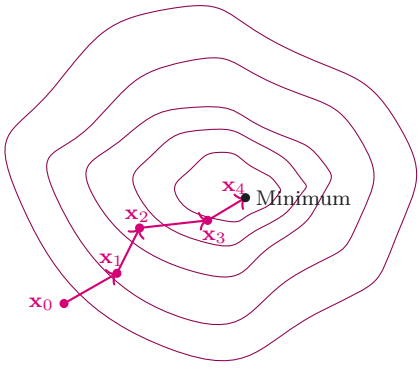
Chaque dérivée partielle donnant l'information sur la pente de la fonction dans la direction de la variable considérée, le gradient possède la propriété de

représenter, en tant que vecteur, la direction de la plus forte pente, c'est-à-dire la direction dans l'espace des variables $(x_1, x_2, \ldots, x_n)$ de dimension n qui permet de faire augmenter la valeur de la fonction $f(x_1, x_2, \ldots, x_n)$ le plus fortement possible. Ce qui signifie que la direction opposée, à savoir $-\nabla f(x_1, x_2, \ldots, x_n)$ pointe dans le sens permettant de diminuer cette valeur le plus fortement possible. Il est donc intéressant de suivre cette direction, à partir d'un point de départ $\vec{x}^0$, afin de se rapprocher du minimum.

Cependant, cette direction n'est valable que localement, similairement à l'idée que, lors d'une randonnée en montagne, un chemin de descente n'est pas rectiligne. Il convient donc d'avancer dans cette direction d'une petite distance seulement, d'une quantité $\alpha \in]0, 1]$ appelée learning rate dans le cas d'une IA, pour diminuer la valeur de la fonction avant de recalculer le gradient à la nouvelle position obtenue, et de recommencer le processus. Nous avons donc l'algorithme itératif suivant :

$$\vec{x}^{n+1} = \vec{x}^n - \alpha \nabla f(\vec{x}^n),$$

qui converge vers un minimum local, et qui est stoppé une fois que $\|\nabla f(\vec{x}^n)\| < \epsilon$, c'est-à-dire lorsque la pente devient plus faible qu'un seuil de tolérance ϵ fixé.



Essayons maintenant d'appliquer cette technique à notre réseau de neurones. Il convient de calculer le gradient de la fonction L, c'est-à-dire ses dérivées partielles en fonction de tous les poids w_i. $L = \frac{1}{2}\left\|\vec{y} - \vec{t}\right\|^2$ est bien une fonction dérivable (et même différentiable) car

$$\vec{y} = \varphi(W^{(n)} \cdot \varphi(W^{(n-1)} \cdot \ldots \cdot \varphi(W^{(2)} \cdot \varphi(W^{(1)}\vec{x}))))$$

est différentiable, car constitué uniquement de produits scalaires et de la fonction d'activation φ. Nous voyons ici toute l'importance de choisir une fonction d'activation qui soit dérivable en tout point.

Analysons tout d'abord le cas le plus facile : celui d'un neurone artificiel unique :

$$y = \varphi(\vec{w} \cdot \vec{x}) = \varphi(z)$$

avec

$$z = 1x_0 + \vec{w}_1 x_1 + \cdots + \vec{w}_n x_n = \vec{w} \cdot \vec{x},$$

z étant un élément intermédiaire qui nous sera bien utile par la suite. La fonction objectif se résume à

$$L = \frac{1}{2}(y - t)^2.$$

Par la *règle de dérivation en chaine*, nous trouvons :

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w_i} = (y - t)\varphi'(z)x_i,$$

ou en considérant $\frac{\partial L}{\partial \vec{w}}$ comme le vecteur gradient par rapport aux variables contenues dans $\vec{w}$:

$$\frac{\partial L}{\partial \vec{w}} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial \vec{w}} = (y - t)\varphi'(z)\vec{x},$$

ce qui est très aisément calculable à partir du moment où la dérivée de la fonction d'activation φ' est une fonction bien connue.

Le cas d'une couche unique de neurones est une simple généralisation (chaque indice i représentant un neurone de la couche) :

$$\vec{y} = \varphi(W\vec{x}) = \varphi(\vec{z}), \quad \vec{z} = W\vec{x},$$

$$\implies \frac{\partial L}{\partial \vec{w}^i} = (y_i - t_i)\varphi'(z_i)\vec{x}.$$

Mais dans un réseau multicouche, la règle de dérivation en chaine nous conduit à l'immense formule suivante (où les vecteurs apparentés sous soumis à la règle de sommation sur leurs indices, les sommes étant ici omises par concision) :

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \vec{x}^{(n)}}\frac{\partial \vec{x}^{(n)}}{\partial \vec{z}^{(n)}}\frac{\partial \vec{z}^{(n)}}{\partial \vec{x}^{(n-1)}}\frac{\partial \vec{x}^{(n-1)}}{\partial \vec{z}^{(n-1)}}\frac{\partial \vec{z}^{(n-1)}}{\partial \vec{x}^{(n-2)}} \cdots \frac{\partial \vec{x}^{(1)}}{\partial \vec{z}^{(1)}}\frac{\partial \vec{z}^{(1)}}{\partial w}$$

où nous avons dû poser les résultats intermédiaires suivants :

- $\vec{z}^{(1)} = W^{(1)} \cdot \vec{x}^{(0)}$, $\vec{x}^{(1)} = \varphi(\vec{z}^{(1)})$
- $\vec{z}^{(2)} = W^{(2)} \cdot \vec{x}^{(1)}$, $\vec{x}^{(2)} = \varphi(\vec{z}^{(2)})$
- ...
- $\vec{z}^{(n-1)} = W^{(n-1)} \cdot \vec{x}^{(n-2)}$, $\vec{x}^{(n-1)} = \varphi(\vec{z}^{(n-1)})$
- $\vec{z}^{(n)} = W^{(n)} \cdot \vec{x}^{(n-1)}$, $\vec{x}^{(n)} = \varphi(\vec{z}^{(n)}) = \vec{y}$,

les différents $\vec{x}^{(1)}$, ..., $\vec{x}^{(n-1)}$ étant à la fois les sorties des couches 1 à $n-1$ et les entrées des couches 2 à n, $\vec{x}^{(n)}$ correspondant à la sortie finale $\vec{y}$.

Un tel calcul à effectuer pour chaque poids w_i serait colossal (il peut y avoir des centaines de couches et des milliards de poids). Mais heureusement, le miracle des mathématiques fait qu'il est possible d'effectuer ce calcul couche par couche (et donc en se limitant aux poids de la couche en question) en partant de la dernière. En effet, il suffit tout d'abord de propager le signal depuis la première vers la dernière couche pour générer toutes les sorties $\vec{x}^{(1)}$, ..., $\vec{x}^{(n)}$ intermédiaires. Ensuite, si nous cherchons à calculer le gradient en fonction des poids $\vec{w}^{i(n)}$ d'un neurone i de la couche n, nous trouvons :

$$\begin{aligned}\frac{\partial L}{\partial \vec{w}^{i(n)}} &= \frac{\partial L}{\partial \vec{x}^{(n)}}\frac{\partial \vec{x}^{(n)}}{\partial \vec{z}^{(n)}}\frac{\partial \vec{z}^{(n)}}{\partial \vec{w}^{i(n)}} \\ &= (x_i^{(n)} - t_i)\varphi'(z_i^{(n)})\vec{x}^{(n-1)},\end{aligned}$$

ce qui est aisément calculable rien qu'avec la connaissance de $\vec{x}^{(n)}$ et $\vec{x}^{(n-1)}$ ($z_i^{(n)} = \vec{w}^{i(n)}.\vec{x}^{(n-1)}$). Notons que $e_i^{(n)} = (x_i^{(n)} - t_i)\varphi'(z_i^{(n)})$ est appelé erreur de niveau n pour la couche i.

Si maintenant nous calculons la quantité suivante :

$$\begin{aligned}\frac{\partial L}{\partial \vec{x}^{(n-1)}} &= \frac{\partial L}{\partial \vec{x}^{(n)}}\frac{\partial \vec{x}^{(n)}}{\partial \vec{z}^{(n)}}\frac{\partial \vec{z}^{(n)}}{\partial \vec{x}^{(n-1)}} \\ &= \sum_i (x_i^{(n)} - t_i)\varphi'(z_i^{(n)})\vec{w}^{i(n)} \\ &= \sum_i e_i^{(n)}\vec{w}^{i(n)},\end{aligned}$$

alors en regardant le gradient en fonction des poids $\vec{w}^{i(n-1)}$ d'un neurone i de la couche précédente, on retrouve cette quantité calculée dans les trois premiers termes du développement :

$$\begin{aligned}\frac{\partial L}{\partial \vec{w}^{i(n-1)}} &= \underbrace{\frac{\partial L}{\partial \vec{x}^{(n)}}\frac{\partial \vec{x}^{(n)}}{\partial \vec{z}^{(n)}}\frac{\partial \vec{z}^{(n)}}{\partial \vec{x}^{(n-1)}}}_{\frac{\partial L}{\partial \vec{x}^{(n-1)}} = \sum_j e_j^{(n)}\vec{w}^{j(n)}}\frac{\partial \vec{x}^{(n-1)}}{\partial \vec{z}^{(n-1)}}\frac{\partial \vec{z}^{(n-1)}}{\partial \vec{w}^{i(n-1)}} \\ &= \sum_j e_j^{(n)} w_i^{j(n)}\varphi'(z_i^{(n-1)})\vec{x}^{(n-2)} \\ &= e_i^{(n-1)}\vec{x}^{(n-2)}.\end{aligned}$$

Nous pouvons donc définir récursivement des erreurs de niveau n, $n-1$, etc. par :

- $e_i^{(n)} = (x_i^{(n)} - \vec{t}_i)\varphi'(\vec{z}_i^{(n)})$
- $e_i^{(n-1)} = \sum_j e_j^{(n)} w_i^{j(n)}\varphi'(z_i^{(n-1)})$
- etc.

et enfin mettre à jour tous les poids en faisant une étape de descente de gradient de pas α via :

$$\vec{w}^{i(k)} = \vec{w}^{i(k)} - \alpha\, e_i^{(k)}\vec{x}^{(k-1)}$$

pour chaque neurone i de chaque couche k.

Cette méthode extrêmement rapide est appelée algorithme de rétropropagation du gradient.

5. Conclusion

Nous avons présenté dans cet article l'évolution de l'IA depuis plus de 70 ans, en nous focalisant sur la construction de programmes capables de jouer aux jeux d'échecs et de Go. La première partie de l'article avait détaillé la longue période que l'on peut qualifier de procédurale où les IA étaient principalement construites sous forme d'analyse plus ou moins exhaustive de possibilités. Cette seconde partie a présenté la révolution intervenue d'abord par le passage à des algorithmes faisant intervenir l'aléatoire, puis, sur ces 10 dernières années, à l'ajout de réseaux de neurones artificiels.

Le programme initial AlphaGo, généralisé en AlphaZero et suivi de ses successeurs, a permis enfin à l'IA d'atteindre le niveau professionnel au jeu de Go en 2015 et de remporter 60 victoires contre des joueurs professionnels en 2017 sans en concéder une seule. Au niveau des échecs, le niveau des IA était déjà très haut avec l'algorithme *min-max* et le programme Stockfish, et la dernière version open source Leela Chess Zero utilisant la recherche arborescente Monte-Carlo combinée à des réseaux de neurones parvient seulement à égaler la performance, les deux programmes étant depuis plus de 5 ans respectivement premier et second au *Top Chess Engine Championship*. Mais en voyant que les réseaux de neurones ont permis de rattraper en moins de 10 ans le niveau des anciennes IA développées depuis plus de 70 ans, il est à parier que ce niveau sera bientôt dépassé.

Le succès d'AlphaGo n'est pas anodin, car il a permis de montrer toute la puissance de l'utilisation de réseaux de neurones et de favoriser leur développement notamment dans l'IA dite *générative*, avec entre autres la percée auprès du grand public du maintenant célèbre ChatGPT. Le fonctionnement de ces IA n'est autre que l'évolution de ce que nous avons présenté ici, la principale différence étant que

l'entrainement du réseau de neurones (l'ajustement des poids) se fait non pas sur des parties d'échecs ou de Go, mais sur des quantités gigantesques de données réelles. Par exemple, ChatGPT ajuste ses poids à partir de données disponibles sur internet (sites web, Wikipedia, livres, articles scientifiques, etc.) en prenant des bouts de phrases comme entrée et en essayant de prédire les mots suivants en sortie, ceux-ci étant comparés aux données. Mais ce type d'IA nécessite des éléments additionnels comme un processus de *tokenisation* permettant de coder des (parties ou combinaisons de) mots de façon à les transformer en nombres réels nécessaires aux entrées et sorties des réseaux de neurones (ce qu'on appelle un *grand modèle de langage* ou *LLM*).

Les tailles de ces réseaux de neurones sont par ailleurs impressionnantes. Par exemple, pour Leela Chess Zero, il y a environ 40 couches (sans compter les couches initiales et finales) d'au moins $256 \times 8 \times 8 = 16384$ neurones, ce qui donne environ 800 000 neurones pour un total de 40 millions de poids à ajuster correctement. Les IA génératives sont d'une échelle incomparable, ChatGPT pour sa version 3 utilise 175 milliards de poids ! On comprend dès lors la nécessité de posséder un algorithme mathématique permettant de trouver rapidement le minimum d'une fonction à 175 milliards de variables.

Pour conclure, il faut se rendre compte qu'il convient de démystifier l'IA, surtout auprès du grand public qui n'a généralement aucune idée de ce qui se cache derrière et peut dès lors imaginer les pires films apocalyptiques. Les IA récentes ne sont que des fonctions mathématiques simples mais aux innombrables paramètres qui essayent de s'ajuster automatiquement en fonction de ce qu'on leur donne, que ce soit un jeu avec des règles ou des données produites ailleurs, et qui, à la façon d'un enfant en bas âge, essayent de reproduire les résultats en faisant un minimum d'erreurs. Ce sont donc des immenses « perroquets » ayant nécessité parfois plusieurs mois d'entrainement avant d'être utilisés, et on peut même se demander pour quelle raison cela fonctionne si bien.

Pour en savoir plus

[1] Silver, D. et al., Mastering the game of Go with deep neural networks and tree search, *Nature*, vol. 529, pp. 48–489, 2016.

[2] Silver, D. et al., Mastering the game of Go without human knowledge, *Nature*, vol. 550, pp. 354–359, 2017.

[3] Silver, D. et al., A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play, *Science*, vol 362 issue 6419, pp. 1140–1144, 2018.

[4] Czarnogórski, K., Monte Carlo Tree Search – beginners guide, 2018 : https://int8.io/monte-carlo-tree-search-beginners-guide/

[5] Nair, S., A Simple Alpha(Go) Zero Tutorial, 2017 : https://suragnair.github.io/posts/alphazero.html

[6] Khovanskiy, M., AlphaZero Chess : How It Works, What Sets It Apart, and What It Can Tell Us, *Towards Data Science*, 2022 : https://towardsdatascience.com/alphazero-chess-how-it-works-what-sets-it-apart-and-what-it-can-tell-us-4ab3d2d08867

[7] Förster, R., AlphaZero from Scratch – Machine Learning Tutorial, Youtube (freeCodeCamp.org channel), 2023, https://youtu.be/wuSQpLinRB4.

[8] Chess programming wiki : https://www.chessprogramming.org/

Nicolas Franco est chargé de cours à l'université de Namur. ✉ nicolas.franco@unamur.be