Une application réelle de l'IA : le jeu de GO

Houcine Senoussi

March 18, 2019

Introduction

- Nous présentons un programme (**AlphaGo**) qui représente l'une des plus grandes réussites de l'IA de ces dernières années.
- Ce programme utilise plusieurs éléments que nous avons étudiés dans ce cours
  - Deep learning.
  - Apprentissage par renforcement.
  - Arbres de recherche.
  - Simulation de Monte carlo.

Description du jeux

- Un jeu de plateau. Le plateau se compose de 19×19 intersections.
- Un jeu de 2 joueurs. Chaque joueur dispose de pierres d'une couleur (Noit/Blanc).
- Le jeu consiste à poser à tour de rôle une pierre sur une intersection de manière à dessiner un territoire. Chaque intersection situé dans un territoire vaut un point.
- Le résultat de la partie dépend du nombre de points obtenu par chaque joueur.

Complexité

- Puisque chaque intersection peut être libre ou occupé par une pierre ou blanche, nous avons en tout $3^{19*19} = 3^{361}$ configurations de jeux possibles.
  - Cela fait donc $\approx 10^{172}$ configurations possibles.
- Cette complexité est bien plus grande que celles des autres jeux connus, et notamment celle des **échecs**.
  - Cela explique pourquoi l'IA n'a que tardivement réussi à produire des "champions" de GO.
- "For computers, Go is the most challenging game of all; solutions can only be computed close to the end of the game or on very small boards." [Réf 1 datant de 2002]

Description globale

1. Le "travail" d'AlphaGo commence par une très lourde phase d'apprentissage durant laquelle deux réseaux sont construits :
   1. Un réseau de stratégie (Policy network) : à chaque configuration du plateau, ce réseau associe une distribution de probabilité qui à chaque action associe une probabilité (probabilité élevée $\rightarrow$ ce mouvement est prometteur).
   2. Un réseau de valeurs : à chaque configuration, ce réseau associe une valeur (valeur élevée $\rightarrow$ état favorisant le gain de la partie).
2. Lorsqu'il jouera des parties, AlphaGo s'appuiera sur ces réseaux, fera beaucoup de simulations en utilisant une combinaison de la méthode de Monte carlo et les arbres de recherche.

Policy networks

1. Nous allons introduire 3 Policy networks. Les deux principaux sont :
   - Le réseau *SL* (pour supervived learning).
   - Le réseau *RL* (pour reinforcement learning).

SL Policy network

- Il s'agit d'un réseau de neurones convolutionnel (ConvNet) qui reçoit en entrée un état s (une configuration du plateau) et qui fournit en sortie une probabilité pour chaque mouvement légal a.
  - La fonction apprise par ce réseau est donc $prob(a \mid s)$. Nous noterons p_σ cette stratégie.
  - Mouvements légaux : 2 (couleurs) $\times$ 19 $\times$ 19.
- L'apprentissage du réseau se fait à l'aide d'un ensemble de couples $\{(s, a)\}$ extraits de parties jouées par des humains (champions).
  - Cet ensemble d'apprentissage contient $\approx$ 30 millions exemples extraits de 160000 parties.
  - L'apprentissage a duré 3 semaines sur plusieurs dizaines de *GPU*.

SL Policy network-2

- Le réseau comporte 13 niveaux comportant chacun une couche convolutionnelle suivie d'une couche *ReLu*, et se termine par une couche *Softmax*.
- L'input du réseau est une pile de $48 \times 19 \times 19$ 'images'. Autrement dit chaque configuration est décrite par 48 features (plutôt que seulement le contenu du plateau, c'est-à-dire les positions des pierres).
  - Le détail des features est données dans le tableau de la figure 1.
- Le premier niveau consiste en 192 convolutions avec des filtres 5×5.
- Les 12 niveaux suivants consistent en 192 convolutions avec des filtres 3×3.
- Le dernier niveau est un niveau fully connected utilisant une fonction *softmax*.

Policy network-3

Extended Data Table 2 | Input features for neural networks

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Figure: Liste des features

Rappels sur les types de couche

1. Couche *convolutionnelle* :
   - Composée d'une pile de tableaux $n \times n$ de neurons. À chaque tableau I on applique une convolution à l'aide d'un filtre T de taille $m \times m$. Au niveau suivant, nous avons donc un tableau de $(n-m+1) \times (n-m+1)$ neurones dont la sortie est définie par :
     $y(i,j) = \sum_{k=1}^{m} \sum_{l=1}^{m} T(k,l) * I(i+k, j+l)$.
   - Les éléments du filtre W jouent le rôle des poids. Bien noter qu'il sont communs à tous les neurones.
   - Pour éviter la réduction du nombre de neurones au niveau suivant, il arrive qu'on ajoute $(m-1)$ lignes et $(m-1)$ colonnes qu'on remplit de 0. On appelle cela du 'padding'. Ainsi le niveau suivant aura
     $(n+m-1-m+1) \times (n+m-1-m+1) = n \times n$ neurones.
   - Le réseau étudié ici utilise le padding.

Rappels sur les types de couche

2. Couche *ReLu*.
   - *ReLu* (pour Rectified Linear unit) est d'abord une fonction d'activation définie par $ReLu(x) = max(x, 0)$. Une couche *ReLu* est une couche dont chaque neurone reçoit la sortie x d'un neurone du niveau précédent et la transforme en $ReLu(x)$.
   - De telles couches sont ajoutées aux réseaux ConvNet car il a été établi qu'elles facilitent l'apprentissage.

Rappels sur les types de couche

3. Couche *Softmax*. La fonction *Softmax* prend en entrée un n-uplet de réels $(z_1, ..., z_n)$ et retourne un autre n-uplet $(y_1, ..., y_n)$ défini par :

$$y_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

Lorsque, comme c'est le cas ici, la couche de sortie d'un réseau de neurons est une couche *Softmax*, cela signifie que :

1. La somme des y_i étant égale à 1, chaque valeur y_i est la sortie de i^{eme} neurone de la couche et représente la probabilité de la i^{eme} classe.
2. La valeur z_i représente l'entrée du i^{eme} neurone, et la couche étant fully connected, cette valeur est une combinaison linéaire des sorties des neurones de la couche précédente plus un éventuel biais.

Performances du Policy network

- Le réseau arrive à reproduire les actions des joueurs humains avec une précision de 57%. À l'époque, la meilleure précision était de 44%.
- Il est capable de battre un "amateur avancé".
- Dans la suite, nous noterons SL ce réseau et p_σ la stratégie qu'il applique (σ fait référence à l'ensemble des poids appris par le réseau).

Rollout policy

- Un autre réseau est construit sur le même modèle mais en plus **rapide** et moins **précis**.
  - Rapidité : 1500 fois plus rapide que le SL policy network. le temps moyen pour choisir une action passe de ≈ 3 *ms* à ≈ 2 μs.
  - Précision : La performance dans la reprodutcion des mouvements humains passe à 24%.
  - Dans la suite ce réseau sera utilisée lorsque nous aurons besoin d'une décision rapide (même si moins précise).
- La stratégie apprise par ce réseau de neurones, est appelée Rollout policy et est notée p_π(π fait référence à l'ensemble des poids appris par le réseau).

Renforcement

- À présent que nous disposons d'une stratégie p_σ, nous allons la **renforcer**.
  - Le réseau actuel est capable de "**deviner**" les actions d'un champion, avec le renforcement il s'agit de faire en sorte qu'il puisse "**battre**" un champion.

Renforcement-2

- Le principe général du renforcement appliqué aux réseaux de neurones est le suivant :
  - Le réseau dans son ensemble ainsi que chaque neurone sont considérés comme des **agents** évoluant dans un environnement.
  - Le réseau reçoit son input de l'environnement. Cela joue le rôle de l'**état** dans l'AR.
  - Les données traversent le réseau qui finit par produire un output qu'il envoie à l'environnement (c'est l'équivalent de l'**action** dans la forme générale de l'AR).
  - L'environnement produit une récompense/punition en fontion de l'action du réseau. Sa valeur résulte de la comparaison entre la valeur que le réseau devait produire et celle qu'il a produite.
  - Chaque neurone/agent met à jour ses poids en fontion de la réaction de l'environnement.

Renforcement-Déroulement de l'apprentissage

- La stratégie à apprendre, qu'on note p_ρ, est initialisée à p_σ.
- Chaque itération = un mini-batch consistant en n parties jouées en parallèle entre le réseau p_ρ et une version précédente du même réseau qu'on note p_{ρ^-} tiré au hasard parmi un pool d'adversaires.
- Après chaque 500 itérations, la version courante de p_ρ est ajoutée au pool des adversaires.
- Durant chaque mini-batch, chaque partie est jouée jusqu'au bout et son résultat (gain et perte) est utilisée pour le renforcement (càd la mise à jour des poids).
  - Le renforcement utilise deux valeurs du reward : $+1$ pour une partie gagnée et -1 pour une partie perdue.

Performance après renforcement

- En tout, il a fallu 10.000 mini-batchs avec $n = 128$ parties chacun. Cette phase a occupé 50 GPU pendant une journée.
- À la fin du renforcement, le réseau *RL* gagnait 80% des parties qui l'opposait au réseau *SL* et 85% des parties qui l'opposait au meilleur logiciel de jeux de Go de l'époque (appelé Patchi).

Réseau des valeurs

- Le but de cette dernière phase est d'associer une valeur à chaque configuration du plateau. Plus exactement, étant donné un état s du plateau et une stratégie p, on définit :
  - $v_p(s) = E(z_T | s_t = s, a_{t..T} \in p)$.
- z_T est le résultat de la partie, a_t est l'action à l'instant t. Cette valeur est la donc la moyenne des résultats des parties en commençant à l'état s lorsque les deux joueurs jouent selon la stratégie p.
- La stratégie utilisée ici est p_ρ du réseau RL (qui est donc la meilleure stratégie dont nous disposons).
- Cette fonction $v_p(s)$ est apprise à l'aide d'un réseau de neurones qui a la même architecture que RL mais qui au lieu de retourner des probabilités retourne une seule valeur (censée donc représenter $v_p(s)$, s étant l'état input du réseau.

Réseau des valeurs-2

- L'apprentissage se fait à l'aide de 30 millions d'exemples extraites de parties jouées par *RL* contre lui-même (les deux adversaires utilisent donc la même stratéguie p_ρ).
  - De chaque partie on extrait une seule configuration (voir remarque ci-dessous).
- Chaque partie étant allée jusqu'au bout, nous connaissons la "vraie" valeur (actual value) de la configuration qu'on en extrait : 1 pour une partie gagnée, -1 pour une partie perdue.
- Après apprentissage, le réseau reçoit une configuration s en entrée et fournit sa valeur $v(s)$ (predicted value).

Réseau des valeurs-3

- Avec 30 millions d'exemples, nous avons 50 millions de batchs de 32 positions chacun. Cela a pris une semaine entière à 50 GPU.

Réseau des valeurs-Remarque

- Nous avons dit ci-dessus que de chaque partie, est extraite **une seule** configuration.
- Les auteurs justifient cela comme suit :
  - Dans une partie, toutes les configurations sont liées : elles se succèdent et aboutissent toutes au même résultat.
  - Dans les premiers essais, des configurations issues de mêmes parties ont été utilisées et il en a résulté le phénomène connu de **sur-apprentissage** (overfitting).
  - Le surapprentissage s'est traduit par un écart important entre les performances du réseau sur son ensemble d'apprentissage et sur l'ensemble de test.

Rappel MSE

- Le problème considéré par ce dernier réseau est un problème de regression : une seule valeur réelle est produite comme output.
- La calcul de l'erreur (écart entre predicted/actual values) se fait à l'aide de la méthode du mean squared error (MSE) qui peut être définie comme suit :
  - Nous disposons de N exemples.
  - Pour le i^{eme} exemple, notons Y_i la valeur réelle et $\hat{Y}$ la valeur prédite par le réseau.
  - L'erreur est alors $MSE = \frac{1}{N}\sum_i (Y_i - \hat{Y})^2$

Récapitulons

- Suite aux différents apprentissage nous disposons à présent de :
  1. un réseau de stratégie (policy network) p_ρ permettant de dire pour chaque état (configuration du plateau) quels sont les mouvements les plus prometteurs.
  2. un réseau de valeurs (value network) permettant de donner une valeur à chaque état.
- Notre programme va combiner ces deux réseaux pour jouer, mais pour cela il a besoin d'un autre élément : le MCTS (Monte Carlo Tree Search).

Motivation

- Il s'agit d'un algorithme probabiliste qui utilise une série de simulations pour construire un **arbre de jeux**.
  - Combinaison de l'échantillonage aléatoire de Monte Carlo et la construction des arbres de recherche.
- Son intérêt est d'autant plus important que l'espace de recherche du jeux est important.
  - Une taille d'espace de recherche trop importante rend impossible l'utilisation d'un algorithme 'force brute', càd la construction de l'arbre complet du jeux.
- Le jeux de Go se caractérise par un nombre de coups possibles à chaque étape trop important. Il a été aussi établi que chaque coup peut avoir des conséquences importantes 50 à 100 coups plus loin.
  - Conséquence : l'arbre de jeux du Go est trop large et trop profond pour être construit en entier.

Définition

- le MCTS représente un jeux sous la forme d'un arbre dans lequel **la racine représente l'état en cours**, les autres noeuds ses successeurs (donc des états possibles du jeux) et chaque arête une action (un coup joué par l'un des deux joueurs).
  - Exemple : la figure 2 représente une portion d'un MCTS possible pour le TicTacToe. Ici on suppose que c'est l'adversaire (qui joue avec X) qui commence. L'arbre correspond au premier coup de notre programme.
- Lorsque le programme choisit et exécute une action, le fils correspodant à cette action devient le nouvelle racine.
- Très important : chaque noeud a une **valeur** qui vient des simulations à l'intérieur de l'arbre dont il est la racine.
  - Les simulations pouvant modifier les valeurs d'un noeud/état : celles qui commencent en ce noeud ou en l'un de ses successeurs directs ou indirects.

Définition

- La valeur d'un noeud est d'autant plus élevée que le nombre de parties simulées en partant de cet état et gagnées est important.

MCTS-Exemple

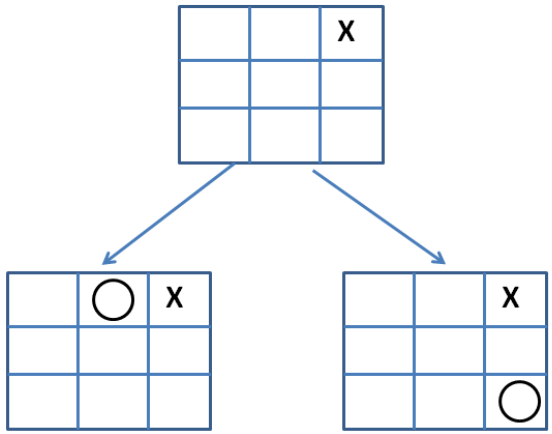

Figure: TicTacToe : un exemple de MCTS

Construction

- La construction d'un MCTS se fait selon des itérations comportant chacune 4 étapes :
  1. **Sélection** : Le programme traverse l'arbre en partant de la racine (état courant, donc) en utilisant une certaine stratégie jusqu'à atteindre un noeud *ns* auquel on peut ajouter des fils. On passe alors à la phase d'expansion.
  2. **Expansion** : Un fils *nf* de *ns* est ajouté à l'arbre. C'est le seul moment de chaque itération qu'un noeud est ajouté à l'arbre.
  3. **Simulation** : Une simulation du jeux est lancée à partir de *nf*. La simulation continue jusqu'à la fin de la partie (ou éventuellement jusqu'à ce qu'une autre condition d'arrêt soit atteinte). Les coups sont choisis de manière aléatoire ou selon une certaine stratégie qui privilégie certains états.

Construction-2

4. **Rétropropagation** : À la fin de la simulation on met à jour la valeur de *nf* et de tous ses prédécesseurs.

- Exemple : la figure3 montre un exemple d'un MCTS et une itération complète (d'après la référence 3).

Construction d'un MCTS-Exemple

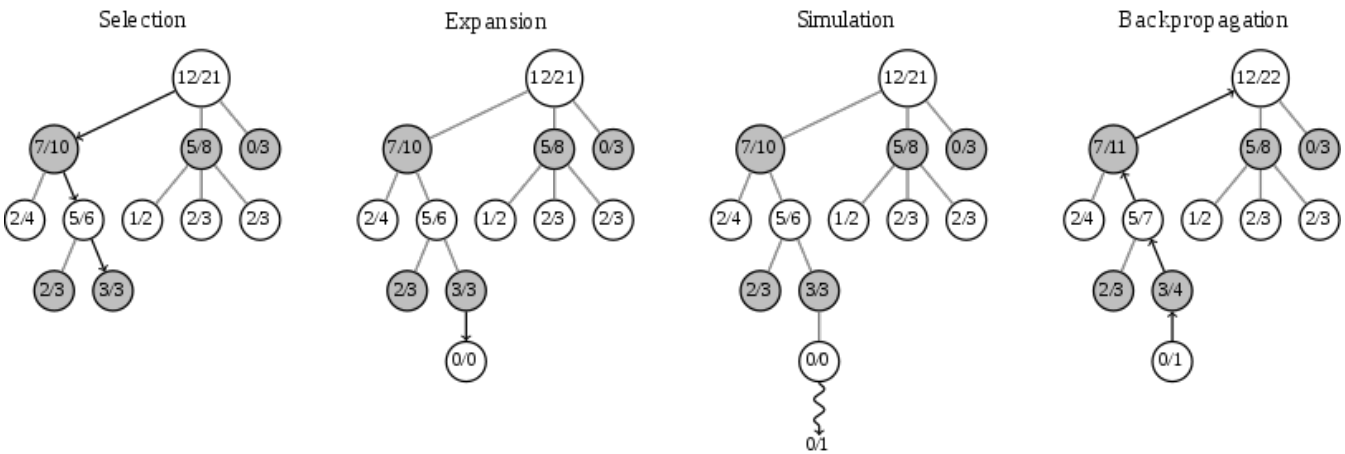

Figure: Les 4 étapes de la construction d'un MCTS

Utilisation

- À tout moment de la partie, le programme doit choisir une action à exécuter à partir de l'état courant. Pour ce faire il dispose en général d'un temps limité. Le programme utilise cette durée limitée pour construire ou compléter le MCTS puis il l'utilise pour choisir le mouvement suivant.

AlphaGo-Choix d'un coup en cours de partie

- Durant une partie, notre programme procèdera à chaque étape comme suit :
  - En partant de l'état actuel il fait des simulations qui iront jusqu'à la fin de la partie. Il construit donc un arbre de recherche dont la racine est l'état actuel.
  - L'espace de recherche étant de très grande taille, cette recherche-construction de l'arbre dispose d'un moyen de **prioriser** : une **combinaision** du réseau de **straégie** et du réseau de **valeurs**.
  - À ces deux moyens nous allons en ajouter deux autres : le **nombre de fois** où un état a été visité (privilégier ceux qui ne l'ont pas souvent été permet de faire de l'**exploration**) et bien sûr le résultats des **simulations en cours** (comme dans les MCTS du cas général).

Choix d'un coup en cours de partie-2

- À l'aide des 4 facteurs précédents, nous définissons deux fonctions $Q(s, a)$ et $u(s, a)$. Notre programme choisira l'action qui maximisera la somme des deux fonctions :
  - À tout moment $a_t = argmax_a(Q(s_t, a) + u(s_t, a))$
- Nous allons détailler les deux fonctions $u()$ et $Q()$. Pour ce faire nous avons besoin d'une notation supplémentaire :
  - $1(s, a, i)$ =1 si l'arête (s, a) a été visitée pendant la i^{eme} simulation, = 0 sinon.
  - i varie entre 1 et le nombre de simulations n.

Choix d'un coup en cours de partie-3

- Définition de la fonction $u(s, a)$:
  1. $N(s, a)$ = nombre de fois où l'action a a été choisie durant les simulations en cours. Autrement dit le nombre de fois où l'arête (s, a) a été traversée. On peut aussi noter :
     - $N(s, a) = \sum_{i=1}^{n} 1(s, a, i)$
  2. $P(s, a)$ = la probabilité $p_\sigma(a|s)$ calculée par le réseau SL.
  3. avec ces deux premières valeurs, nous calculons celle de $u(s, a)$:
     - $u(s, a) = \frac{P(s,a)}{1+N(s,a)}$.
  4. La fonction $u()$ favorise donc l'exploration : plus une action a été exécutée, moins elle a de chance de l'être à nouveau.

Choix d'un coup en cours de partie-4

- Définition de la fonctions $Q(s, a)$:
  1. L'**expansion** du MCTS nous conduit à ajouter une feuille/état qu'on va noter S_L.
     - Le réseau des valeurs attribue à cet état une valeur $v(S_L)$.
     - En partant de cette feuille et en poussant la simulation jusqu'à la fin (phase de simulation) nous allons à un résultat que nous notons z_L (gain ou perte).
  2. Nous allons combiner ces deux valeurs pour en avoir une nouvelle $V(S_L) = \lambda\ v(S_L) + (1\text{-}\lambda)z_L$, avec $0 \leq \lambda \leq 1$.
- Ensuite nous définissons $Q(s, a)$ comme suit :
  - $Q(s, a) = \frac{1}{N(s,a)} \sum_i^n 1(s, a, i) V(S_L^i)$

Choix d'un coup en cours de partie-5

- La troisième phase de chaque itération du MCTS est la **simulation**, c'est-à-dire partir de la feuille qu'on vient d'ajouter pour aller jusqu'à la fin de la partie.
  - Cette simulation se fait à l'aide du réseau rapide (et moins précis) dont nous avons appelé la stratégie **rollout policy** p_π. C'est la raison d'être de ce réseau, c'est pour cela qu'il a été construit.
- Quatrième phase (rétropropagation) : la fin de chaque simulation nous donne des élements pour mettre à jour les valeurs de $N(s, a)$, z_L donc aussi $V(s_L)$ et par conséquent $Q(s, a)$.

Choix d'un coup en cours de partie-6

- Une fois terminées les simulations, la construction du MCTS dont la racine est l'état courant s et l'évaluation de ses successeurs, *AlphaGo* choisit le prochain coup. Le critère est le suivant :
  - le coup a choisi est celui qui correspond à la plus grande valeur de $N(s, a)$.
  - Signification de ce choix (qui peut surprendre) : plus on avance dans la simulation plus l'**exploitation** est préférée à l'**exploration**, si un mouvement est le plus choisi c'est qu'il est le plus prometteur.

Conclusion

1. Nous avons décrit le fonctionnement du programme AlphaGo qui a battu le champion du monde de Go.
2. Ce fonctionnement peut être résumé comme suit :
   - Construction d'un réseau de neurones profond *SL* à partir de parties jouées par des champions. Ce réseau permet d'associer un mouvement à chaque état. Stratégie de ce réseau : p_σ.
   - Un réseau construit sur le même modèle mais moins précis et plus rapide. Stratégie de ce réseau : p_π.
   - Le réseau *SL* est renforcé pour donner naissance à un autre réseau appelé *RL*. Stratégie de ce réseau : p_ρ.
   - Le réseau *RL* est utilisé pour construire un réseau de valeurs. Ce réseau permet d'associer une valeur à chaque état. La fonction associée à ce réseau : v.

Conclusion

1. Ce fonctionnement peut être résumé comme suit (suite) :
   - Lorsqu'il joue ses parties, AlphaGo fait à chaque étape un grand nombre de simulation à l'aide d'un MCTS avant de choisir le coup à jouer. Pendant ces simulations, AlphaGo exploite les points forts de chaque réseau.

Bibliographie

1. Games solved: Now and in the future. H. Jaap van den Herik et al.
2. Monte Carlo Tree Search and Its Applications, M. Magnuson, 2015.
3. Par Mciura - Travail personnel, CC BY-SA 3.0, `https://commons.wikimedia.org/w/index.php?curid=25382061`.
4. et bien sûr la référence principale :
   - Mastering the game of Go with deep neural networks and tree search, D. Silver et al, 2017.