LittleGo : Intelligence Artificielle pour le jeu de Go

Samuel Batissou et Mathieu Pont

Mai 2019

Abstract

1 Introduction

Dans le cadre du master informatique première année de l'université de Paris (anciennement Paris Descartes) nous avions un projet au sujet libre dans le cours "Apprentissage Automatique". Nous devions utiliser une des méthodes vues en cours.

Etant intéressés par les réseaux de neurones et le jeu de Go nous nous sommes dirigés vers les travaux de DeepMind sur AlphaGo Zero [D. Silver et al., 2017]. C'est une intelligence artificielle qui a appris à jouer au jeu de Go en ayant joué uniquement contre elle même, le tout reposant sur un réseau de neurones et *Monte Carlo Tree Search* (MCTS).

Les moyens mis en jeu pour l'apprentissage par l'algorithme sont énormes (64 GPU, 19 CPU, 4 TPU, coût estimé à environ 25 millions de dollars pour 40 jours d'entrainement pour la plus grosse version).

L'objectif ici n'est donc pas tant de reproduire leurs résultats mais de reprendre le principe de leur méthode et essayer de la faire marcher sur du matériel avec peu de moyen (sur une machine, ayant 1 seul GPU, estimée à environ 1500 euros) en modifiant des hyper-paramètres ou autres pour l'adapter à notre matériel.

2 Librairie pour le jeu de Go : libshusaku

Pour développer le cœur décisionnel nous avons utilisé le langage Python, pour ces grandes librairies très bien supportées comme Tensorflow ou Numpy. Néanmoins dans l'optique d'imiter les résultats de DeepMind, le goulot d'étranglement devait être le réseau de neurones et non la gestion du jeu de Go (comme la génération de coups pour le MCTS par exemple). C'est pour cette raison que nous avons choisi de ne pas utiliser Python (ayant des performances d'exécutions plus faibles que d'autres langages) pour cette partie.

Nous avons ainsi créé notre bibliothèque pour gérer tous les aspects et règles du jeu de Go. Pour la développer nous avons choisi "Rust" un langage compilé sans GC[1] et moderne, pour les performances[2] et aisance de développement.

[1] Garbage collector

[2] Performances égales au langage C: https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/rust.html

L'interface entre Python et Rust se fait naturellement avec une librairie très puissante qui génère les bindings Python depuis le code Rust. La librairie python est trouvable sur pypi.org[3] et la librairie rust sur crates.io[4].

3 Méthodologie

3.1 Architecture du réseau de neurones

Entrée. AlphaGo Zero utilise un réseau de neurones prenant en entrée les 8 derniers plateaux de la partie du jeu de Go et un plan supplémentaire pour indiquer quel joueur est en train de jouer (tout à 0 pour dire que c'est au joueur noir, et tout à 1 pour le joueur blanc). Chaque plateau est séparé en deux plans, un contenant uniquement les pierres du joueur noir et l'autre uniquement celles du joueur blanc. Cela fait donc en tout $8 \times 2 + 1 = 17$ plans, soit une entrée de $19 \times 19 \times 17$ (19×19 étant la taille d'un plateau).

L'ensemble des plans forment la séquence: $s_t = \{X_t, Y_t, X_{t-1}, Y_{t-1} ... X_{t-7}, Y_{t-7}, C\}$. Avec X le plan correspondant aux pierres du joueur auquel c'est le tour, Y celui du joueur adverse et C le plan indiquant quel joueur est en train de jouer.

Sortie. L'architecture possède deux sorties. La première est la politique d'action $\mathbf{p}$, c'est un vecteur de dimension $19 \times 19 + 1 = 362$ (une valeur pour chaque case du plateau et une pour le coup correspondant à passer son tour), elle permet de savoir quel est le meilleur coup à jouer pour un état donné. La deuxième sortie est une fonction d'évaluation v du plateau, c'est un scalaire dans l'intervalle [-1, 1], 1 symbolise que le plateau est plutôt favorable pour le joueur actuel et -1 pour le joueur adverse.

Le réseau de neurones peut être vu comme la fonction suivante:

$$f_\theta(s) = (\mathbf{p}, v) \tag{1}$$

Architecture interne. Les premières couches du réseau sont des couches de convolution ayant des connexions résiduelles, elles sont agencées par bloc. Chacun d'eux est construit de la manière suivante:

- Couche de convolution, 256 filtres, kernel 3x3, stride 1 (+ Batch normalisation + ReLU)
- Couche de convolution, 256 filtres, kernel 3x3, stride 1 (+ Batch normalisation)
- Connexion résiduelle connectant l'entrée du bloc
- ReLU

Ils utilisent 19 ou 39 blocs (en les mettant l'un après l'autre) et appellent ceci la tour résiduelle. En amont de cette tour il y a une couche de convolution de mêmes paramètres qu'une des couches d'un bloc (suivi de batch normalisation et ReLU). En aval, le réseau est séparé en deux "têtes". Une pour la politique d'action et l'autre pour la fonction d'évaluation présentées précédemment.

[3] https://pypi.org/project/libshusaku/
[4] https://crates.io/crates/goban

Tête de la politique d'action:

- Couche de convolution, 2 filtres, kernel 1x1, stride 1 (+ Batch normalisation + ReLU)
- Couche entièrement connectée de $19 \times 19 + 1 = 362$ neurones (+ Softmax)

Tête de la fonction d'évaluation:

- Couche de convolution, 1 filtre, kernel 1x1, stride 1 (+ Batch normalisation + ReLU)
- Couche entièrement connectée de 256 neurones (+ ReLU)
- Couche entièrement connectée de 1 neurone (+ TanH donnant une valeur entre -1 et 1)

3.2 Méthode d'apprentissage pour le réseau de neurones

Comme la rétropropagation est utilisée ils faut arriver à former les labels à la fois pour la politique d'action et pour la fonction d'évaluation. Chaque donnée d'apprentissage peut être vu comme un tuplet $\{s_t, \boldsymbol{\pi}_t, z_t\}$ où s_t est l'état du plateau à l'instant t (sous la forme des différents plans consécutifs comme décrit plus haut). $\boldsymbol{\pi}_t$ le label pour la politique d'action. z_t le label pour la fonction d'évaluation étant -1 ou 1 en fonction de si le joueur jouant à l'instant t a gagné (1) ou non (-1).

Chaque état du plateau est enregistré tout au long d'une partie mais n'est labellisé qu'à la fin de celle-ci car le label z_t (pour la fonction d'évaluation) n'est connu qu'à ce moment là.

Pour $\boldsymbol{\pi}_t$ ils utilisent le résultat du MCTS comme label, cela a pour objectif d'essayer d'apprendre et d'estimer les prédictions faites par le MCTS.

Le réseau utilise l'erreur quadratique moyenne (pour la fonction d'évaluation) et l'entropie croisée (pour la politique d'action) comme fonction d'erreur:

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

avec c le paramètre controlant la regularisation L2 pour empêcher le sur-apprentissage.

3.3 Variante de la méthode d'apprentissage par renforcement

Ici nous proposons une variante pour la méthode d'apprentissage par renforcement qui ne fait pas intervenir le MCTS. Au lieu d'essayer d'apprendre les prédictions du MCTS nous allons essayer de faire apprendre l'évaluation du plateau en utilisant la fonction d'évaluation.

L'objectif est que la sortie $\mathbf{p}$ nous donne l'évaluation du plateau pour chaque coup possible que l'on peut jouer. La valeur la plus haute sera le coup donnant le plateau le plus favorable et ce sera donc celui qui sera choisi.

Pour ce faire, à un instant donné, pour chaque coup possible nous allons refaire une passe dans le réseau afin de simuler ce coup et de voir la valeur de la fonction d'évaluation. Etant donné que la valeur de la fonction d'évaluation se base sur des données non biaisées (victoire ou non victoire, il n'y a pas d'approximation ou d'ambiguité) on peut donc compter sur sa fiabilité pour assurer un bon apprentissage.

C'est bien évidemment une version extrêmement simplifiée par rapport au MCTS, et surement moins performante, mais elle permet de faire moins de passes dans le réseau (au maximum $19 \times 19 + 1 = 362$ contre 1600 avec le MCTS) et permet une implantation plus simple.

4 Expériences et résultats

Nous avons utilisé la librairie Python Tensorflow pour implanter le réseau de neurones.

4.1 Modification des hyper-paramètres originaux

Dans un premier temps nous avons décidé de changer l'architecture afin de la réduire pour qu'elle soit plus facilement entrainable sur notre machine. Nous utilisons 5 blocs (au lieu de 19 ou 39) et 64 filtres sur les couches de convolution (au lieu de 256).

Nous avons cependant augmenté le nombre de filtres à 32 sur la couche de convolution de la tête de la politique d'action et celle de la fonction d'évaluation (au lieu de respectivement 2 et 1 auparavant). En effet, nous avons vu que cela facilitait l'entraînement.

Pour ce qui est des plans en entrée nous avons considérablement réduit la dimension, au lieu de prendre les 8 derniers plateaux nous prenons uniquement les 2 derniers. Cela nous permet de passer de 17 plans ($8 \times 2 + 1$) à seulement 5 ($2 \times 2 + 1$) et ainsi de diminuer grandement les calculs et la mémoire nécessaire.

4.2 Apprentissage

Afin de valider l'architecture (avant de faire l'apprentissage par renforcement) DeepMind l'entraîne via de l'apprentissage supervisé sur le jeu de données KGS[5] qui contient 29.4 millions de positions de jeu (état de plateau). Cela permet de s'assurer de bons hyper-paramètres, bien évidemment le réseau est réinitialisé avant de lancer l'apprentissage par renforcement.

Ils utilisent KGS comme jeu d'entraînement et de test, comme ensemble de validation ils utilisent le jeu de données GoKifu[6] (que nous n'avons pas eu le temps d'utiliser).

Malheureusement, de part la capacité de notre machine nous ne pouvions charger tout le jeu de données KGS en mémoire et avons pu prendre seulement 25000 positions.

Cependant nous pouvons augmenter le jeu de données de part les règles du jeu de Go et de la forme carrée du plateau, nous pouvons ainsi faire des rotations et des réflexions pour chaque plateau et ainsi multiplier par 8 chacun d'eux (4 rotations, puis 4 rotations avec le plateau en miroir).

		KGS train	KGS test	GoKifu validation
Supervisé	AlphaGo Zero (20 blocs)	62.0	60.4	54.3
	AlphaGo (12 couches)	59.1	55.9	-
	LittleGo	43.8	37.0	-
Renforcement	AlphaGo Zero (20 blocs)	-	-	49.0
	LittleGo	-	-	-

Table 1: Précision sur la prédiction de la politique d'action.

Nous remarquons sur la table 1 que LittleGo ne parvient pas à égaler AlphaGo ou AlphaGo Zero au niveau de la précision de la prédiction de la politique d'action. Cependant nous

[5]disponible sur https://u-go.net/gamerecords/
[6]http://gokifu.com/

utilisons uniquement les 2 derniers plateaux pour représenter l'état contre les 8 derniers pour les différents AlphaGo.

Nous avons à la base utilisé seulement le dernier plateau, utiliser les deux derniers a considérablement augmenté la précision (presque 2 fois supérieur). Nous pouvons donc supposer qu'ajouter des plateaux augmente la précision. Cela fait sens, car prendre plus en compte les derniers coups peuvent aider dans la prise de décision du prochain coup.

		KGS train	KGS test	GoKifu validation
Supervisé	AlphaGo Zero (20 blocs)	0.177	0.185	0.207
	AlphaGo (12 couches)	0.19	0.37	-
	LittleGo	0.27	0.26	-
Renforcement	AlphaGo Zero (20 blocs)	-	-	0.177
	LittleGo	-	-	-

Table 2: Erreur sur la prédiction de la fonction d'évaluation.

Sur la table 2 nous voyons que LittleGo arrive à battre AlphaGo sur le jeu de test (valeur la plus importante à prendre en compte) mais pas AlphaGo Zero pour la prédiction de la fonction d'évaluation. C'est un résultat tout de même intéressant de se retrouver entre les deux versions et d'avoir réussi à faire mieux que AlphaGo.

Il est important de garder en tête que nous travaillons sur uniquement 25000 positions de jeu (multiplié par 8 grâce à l'augmentation de données décrite plus haut) contre 29.4 millions pour DeepMind. Faire analyser plus d'exemples peut d'un côté rendre plus performant l'apprentissage (dans le sens d'avoir une plus grande précision car on analyse plus de cas différents) mais d'un autre côté le ralentir (car l'inférence sur bien plus de données est plus longue).

Comme nous pouvons le voir sur les différentes tables nous n'avons pas pu évaluer notre algorithme avec l'apprentissage par renforcement par manque de temps.

5 Conclusions et futurs travaux

Nous voyons à travers ce travail que nous avons pu reproduire relativement partiellement les résultats de AlphaGo Zero à l'aide de moyens extrêmement inférieurs à ceux utilisés par DeepMind. Nous avons réussi à faire mieux sur la fonction d'évaluation que la version originale, AlphaGo, mais en utilisant un sous-ensemble du jeu de données. Nous voyons donc à travers ce travail qu'en réduisant considérablement l'architecture nous pouvons tout de même atteindre des résultats convenables.

Comme travail futur nous pourrions évaluer l'impact du nombre de plateaux pour représenter un état sur la précision de la politique d'action.

Il serait aussi intéressant d'évaluer notre architecture avec l'apprentissage par renforcement et ainsi comparer la méthode proposée par DeepMind avec la notre. Cela permettrait de voir si une version simplifiée de l'apprentissage permettrait d'obtenir tout de même des bons résultats.

Le pré-apprentissage dans les réseaux de neurones à l'aide d'algorithmes non supervisés a fait largement ses preuves ([Erhan et al., 2010]) et notamment dans l'apprentissage par renforcement ([Abtahi et al., 2011]). C'est une des possibles pistes que nous avons prévu d'explorer afin d'améliorer les résultats.

De plus, la récente invention des capsules ([Hinton et al., 2017]) ayant pour objectif d'améliorer les réseaux de neurones à convolution pourrait aussi aider dans l'obtention de meilleurs résultats. Cependant les capsules semblent plus difficilement entraînable et pourraient donc certes améliorer les résultats mais ralentir l'entraînement.

De même que pour la couche récurrente LSTM, elle permettrait de ne pas entasser les plans comme on le fait et de garder une trace temporelle entre chaque donnée mais elle est plus difficilement entraînable.

Le répertoire github associé à ce projet et ouvert à tous et libre d'utilisation[7].

References

Erhan et al., 2010 -Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol and Pascal Vincent. Why Does Unsupervised Pre-training Help Deep Learning? 2010.

Abtahi et al., 2011 - Farnaz Abtahi and Ian Fasel. Deep Belief Nets as Function Approximators for Reinforcement Learning. 2011.

Silver et al., 2016 - D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. 2016.

Hinton et al., 2017 - Geoffrey E Hinton, Sara Sabour and Nicholas Frosst. Dynamic Routing Between Capsules. 2017.

Silver et al., 2017 - D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel and D. Hassabis. Mastering the Game of Go without Human Knowledge. 2017.

[7] https://github.com/SamBlaise/shusaku