Reinforcement Learning

MorpionZero, utilisation du programme AlphaZéro adapté au TicTacToe

---

AXEL MARCHAND, RODRIGUE RILLARDON

1 Introduction

Le programme de Deepmind AlphaGo a réussi l'incroyable prouesse en mars 2016 de réussire à battre le 18 fois champion du monde de Go Lee Sedol 4-1[4] dans un match regardé par plus de 200 millions de personnes. Si cette prouesse est surprenante, c'est parce que le jeu de Go avait pour réputation d'être très difficile pour l'ordinateur, au contraire par exemple du jeu d'échec pour lequel des stratégies informatiques capables de battre l'homme existent depuis de nombreuses années, on peut par exemple parler du retentissant match entre Kasparov et DeepBlue[13].
Cependant, cette prouesse d'AlphaGo a été récemment éclipsée, le 18 octobre 2017, par le nouveau programme de DeepMind, AlphaGo Zero, qui a réussi à battre son parent AlphaGo 100 à 0[1]. Encore plus incroyable, cette nouvelle variante de l'algorithme n'avait besoin d'aucun apport humain. En partant de zéro, et en jouant uniquement contre lui même, AlphaGo Zero avait réussi à surpasser toutes les meilleures intelligences artificielles dans le domaine, sans avoir besoin de bases de données de jeu d'humains.
48 jours plus tards, le 5 décembre 2017, DeepMind publie l'article ***Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm***, montrant comment l'algorithme initial d'AlphaGo Zero pouvait être adaptée pour des jeux comme les échecs ou le shogi.
Lors de notre projet, nous avons voulu adapter l'algorithme AlphaZero à un jeu plus simple, le morpion.

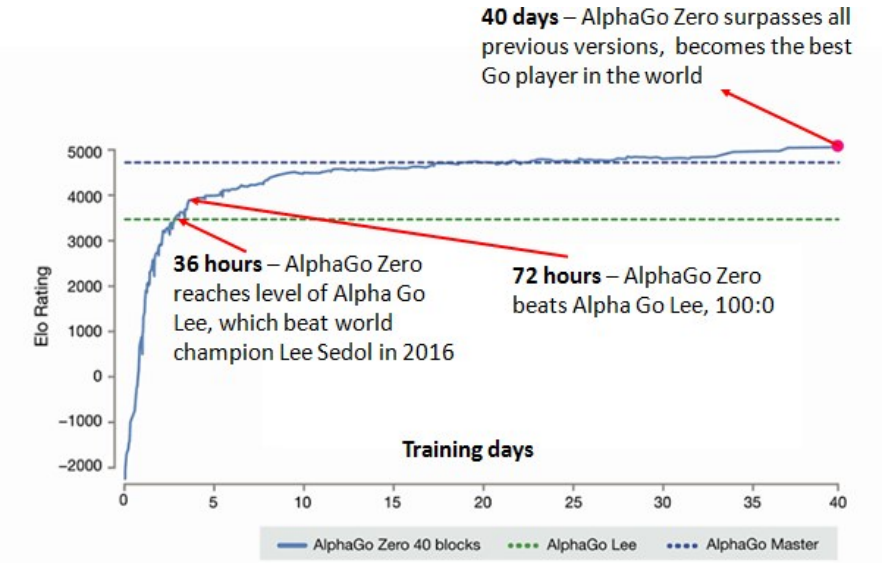


FIGURE 1: Performances d'AlphaGo Zero contre ses pairs

2 Caractéristiques d'AlphaZero

2.1 Absence d'input humain

La plupart des programmes d'intelligence artificielle existant jusqu'à présent reposaient sur des algorithmes basés sur des principes de jeux existant et déterminés par des grands maîtres et une exploration du jeu *alpha beta*[13].
L'exploration *Alpha Beta* est une optimisation de l'algorithme minmax permettant de n'explorer qu'une partie de l'arbre des possibles.

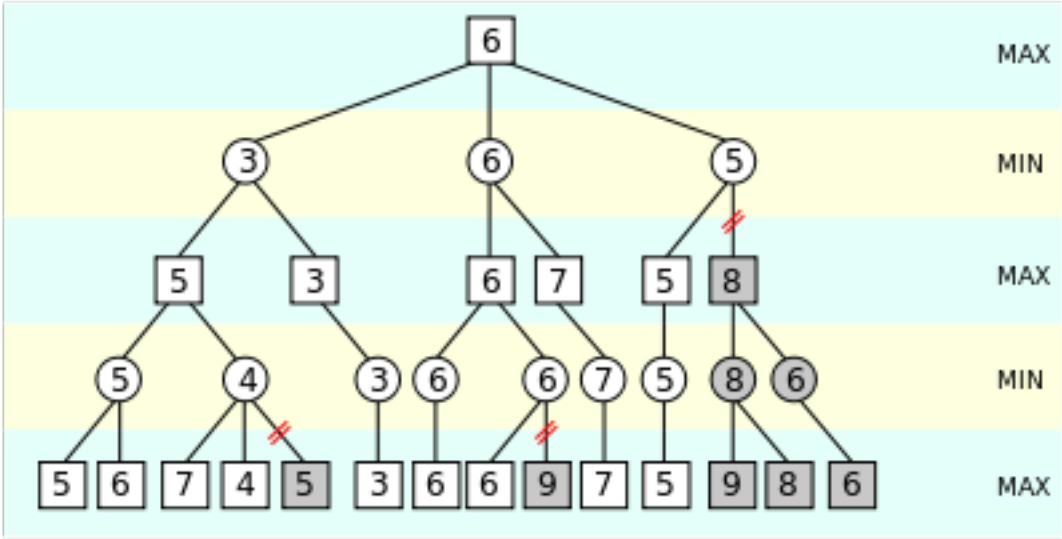


FIGURE 2: Exemple de recherche Alpha Beta

En revanche, l'algorithme AlphaZero n'a besoin d'aucune aide humaine. Une fois les règles du jeu implémentées, le jeu va apprendre de lui même à l'aide de *self play* quelles sont les combinaisons optimales.

2.2 Un algorithme compréhensible

L'algorithme régissant le comportement d'AlphaZero est très facilement compréhensible. On peut résumer ses différentes étapes de la façon suivante[7] :

1. Jouer mentalement avec les scénarios futurs possibles, en considérant les choix de l'adversaire et en donnant la priorité aux choix les plus avantageux (exploration exploitation)

2. Après avoir atteint un état inconnu, évaluer la performance de cet état et faire remonter le score à travers les positions précédentes qui ont conduit à ce point.
3. Après avoir fini de considérer les possibilités futures, prenez l'action la plus explorée.
4. À la fin du jeu, revenir en arrière et évaluer si la valeur des positions futures a été correctement prise en compte et actualiser la compréhension en conséquence.

Ces différentes étapes sont comparables à celle d'un homme cherchant à apprendre à jouer à un nouveau jeu. Nous allons maintenant rentrer un peu plus dans les détails et chercher à comprendre comment AlphaZero fonctionne.

3 Fonctionnement de l'algorithme

3.1 Réseau de Neurone

Le réseau de neurone utilisé à pour vocation de remplacer la fonction d'évaluation et de meilleurs move. En fonction des paramêtres θ du réseau de neurone, on obtient

$$(p, v) = f_\theta(s)$$

avec p la distribution de probabilités des différentes actions possibles et v la valeur associée à l'état du plateau.
Le réseau de neurone est constitué des couches suivantes[7].

1. Une couche de convolution

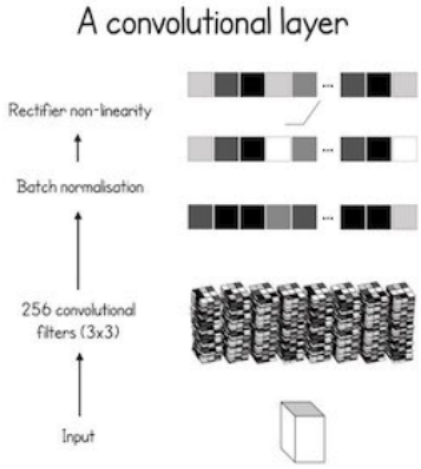


FIGURE 3: Couche de convolution du réseau de neurone

2. 19 couches résiduelles composées de la façon suivante

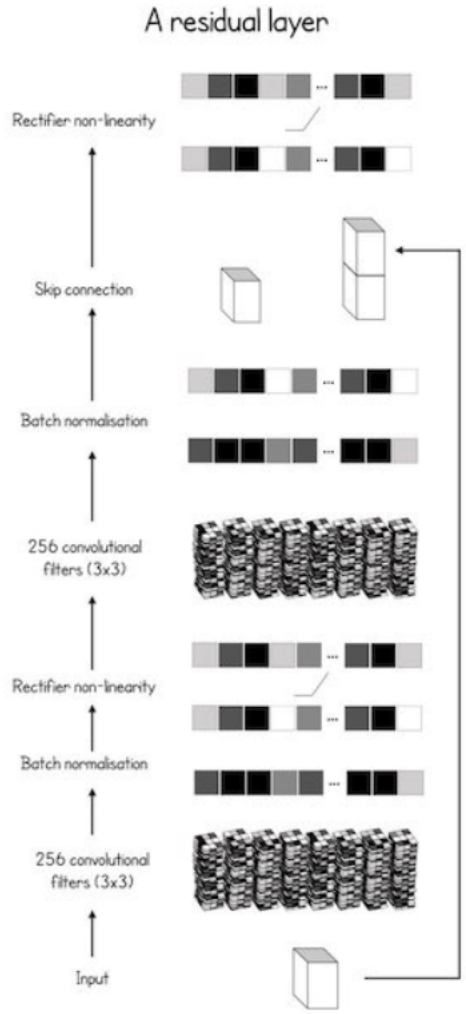


FIGURE 4: Residual layer

3. Les couches d'output, divisée en une partie *policy*, donnant la distribution des probabilités de sortie, et *value*, donnant la valeur de l'état du board.

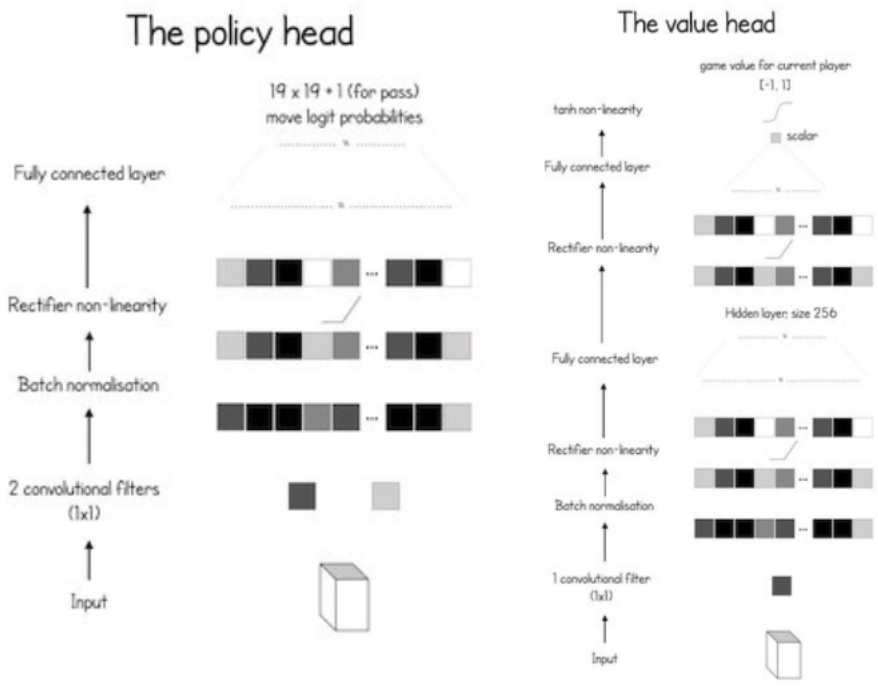


FIGURE 5: Output layer

Le réseau ainsi créé prend en entrée le plateau sous forme d'un tensor et renvoie d'un côté la *policy* et la *value*.

3.2 Loss Function

La loss de notre fonction doit être capable de prendre en compte à la fois les erreurs sur la value, mais aussi sur la policy. On va donc pour cela sommer une Mean Squared Error sur la value v comparée avec le vainqueur du jeu z et une cross entropy tel que

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

$c||\theta||^2$ contrôle les poids, tandis que le terme $\pi^T log(p)$ permet de comparer les similarités entre notre policy et les moves possibles par MCTS, que nous allons détailler.

3.3 Monte Carlo Tree Search

Dans une première partie nous utilisons un réseau pour évaluer la policy et la value associée à un board. C'est la première phase de **selection**.

Cette première phase effectuée, la deuxième phase est **expand**. Elle consiste à extrapoler une action de l'adversaire. On obtient donc un set d'action, auxquel est associé des values v qui désignent la probabilité de victoire[12].

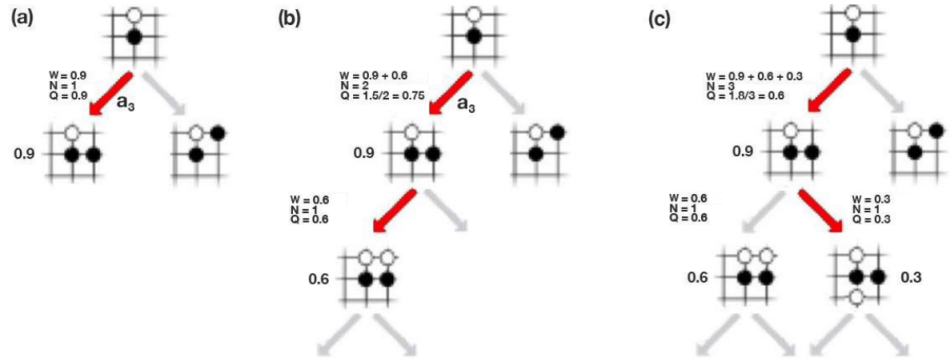


FIGURE 6: Illustration de l'algorithme MCTS

Dans la figure ci dessus, W désigne la probabilité de victoire, N le nombre de fois que a_3 est visité, et Q est l'indice dit *action value*.
Dans (a), on visite une fois a_3, $N = 1$, les chances de victoire associées au plateau sont 0.9, et on a donc $Q = 0.9$. Après avoir **expand**, on obtient (b). On a donc $N = 2$ car on observe une fois de plus a_3. De plus, comme $W = 0.6$ on a donc $Q = mean(W) = 0.75$ et de même dans (c), après avoir choisi une autre branche, $Q = 0.6$. Q est semblable à notre policy p. Cependant, dans le cas d'un jeu comme les échecs, l'arbre est immense et il faut chercher les états du plateau de manière plus efficace. On doit donc optimiser notre **exploitation-exploration**, tel que l'exploitation consiste à s'intéresser aux arbres avec une valeur Q élevée, et l'exploration des arbres avec une valeur N faible[10].
On choisit la prochaine action selon la formule

$$a_t = argmax_a(Q(s,a) + u(s,a))$$

avec Q qui contrôle l'exploitation et $u(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$ le terme d'exploration. Au début, notre algorithme aura tendance à explorer, mais plus les iterations augmentent plus Q deviendra précis. On utilise donc MCTS pour obtenir une nouvelle policy π, tel que

$$\pi_a \propto N(s,a)^{1/\tau}$$

$$\pi(a|s) = N(s,a)^{1/\tau} / (\sum_b N(s,b)^{1/\tau})$$

τ est appelée la température. En fixant τ à 1, on sélectionne la prochaine branche en prenant en compte le nombre de visites. Quant τ tend vers 0, seul l'action la plus visitée sera prise en compte. On effectue une **policy iteration**.
Pour résumer les différentes étapes de MCTS sont donc

1. **selection** : on observe une action que l'on veut analyser
2. **expansion** : on effectue une nouvelle action pour élargir notre arbre
3. **backup** : on update de manière récursive la valeur de notre branche
4. **repetition** : on répete ces mêmes étapes jusqu'à obtenir une policy π et on effectue un tirage à partir de cette policy.

Dans le cas d'AlphaZero, le nombre d'itérations était fixé à 1600.

4 Application

Nous avons cherché à appliquer les résultats d'AlphaZero à un jeu beaucoup plus simple, le TicTacToe.

4.1 Implémentation

Tous les détails de notre implémentation sont disponibles et détaillés dans le notebook *Demo.ipynb*. Nous avons également mis à disposition les différents fichiers dans le dossier *morpion*.
Le jeu du morpion est encodé de la façon suivante :

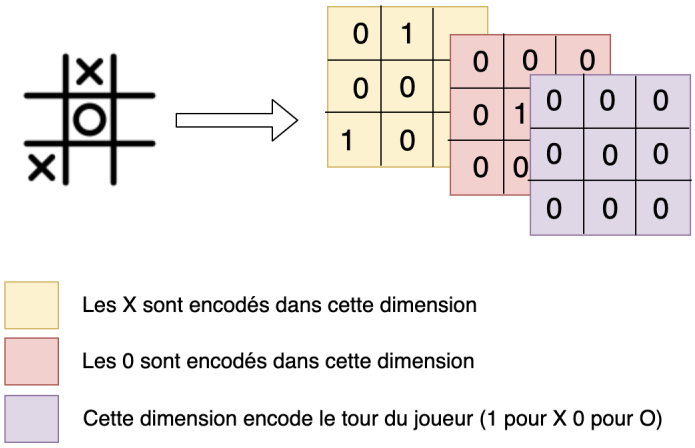


FIGURE 7: Transcription du jeu en tensor

Ce tensor est ensuite donné au réseau qui renvoie la policy et la value. Puis, le MCTS permet de d'obtenir la nouvelle policy π déterminant le choix du prochain move. Les ensembles (s, π, v) avec s l'état du plateau, π la bonne policy, et v le vainqueur est ensuite donné au réseau pour son entraînement.
Le fonctionnement complet de AlphaZero est donc le suivant[8] :

1. Self-play du réseau en utilisant MCTS pour générer un dataset (s, π, v)
2. Entrainement du réseau à l'aide des datasets (s, π, v) générés
3. Faire affronter le réseau contre le réseau précédent et garder celui qui gagne le plus en général
4. Répéter cette étape le nombre d'itération nécessaires

4.2 Déroulement

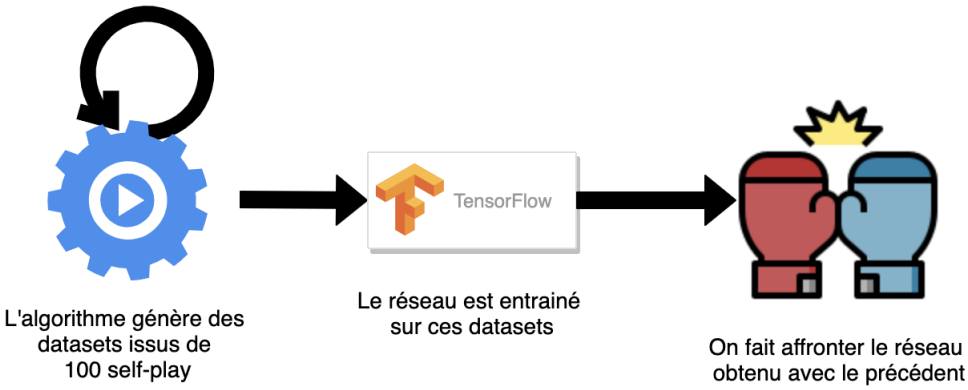


FIGURE 8: Déroulement d'une itération de l'algorithme

Nous avons dans notre expérience utilisé les paramètres suivants :

— Nombre de self-play par itération :100
— Nombre d'itérations : 10
— Nombre d'epoch : 10
— Temperature : 1
— Nombre de jeu à jouer contre la version précédente : 40
— Winrate nécessaire pour qu'un réseau soit meilleur qu'un autre : 0.6

4.3 Résultats

Notre recherche a été divisée en deux parties. Dans un premier temps nous avons implémenté la structure complète d'AlphaZéro, avec les 19 couches résiduelles (voir notebook). Cependant, au vu des performances décevantes et du temps d'entrainement long, nous avons décidé d'utiliser un réseau beaucoup plus petit, composé de 4 couches de convolutions et de la value head et policy head. Malgré un réseau théoriquement plus simple, on obtient un entrainement du jeu plutôt long. On obtient par itération une durée moyenne de 30 min. Mais qui cette fois ci donnait beaucoup plus de résultats.
On observe sur ce réseau bien la décroissance de la loss totale. Qui atteint néanmoins un plafond après la troisième itération.

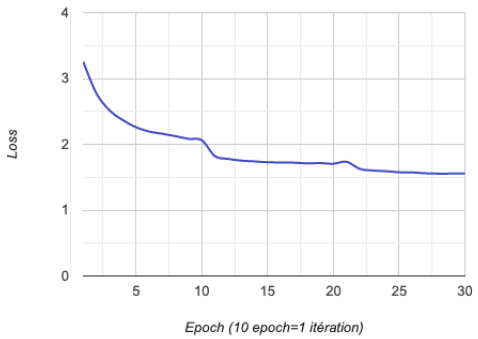
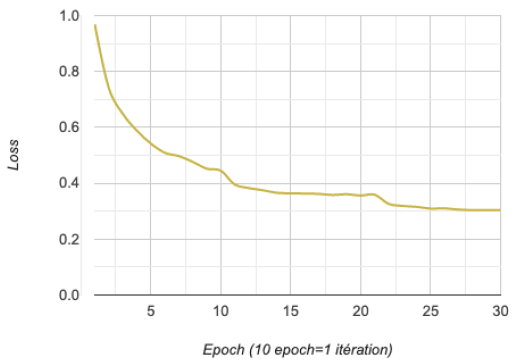
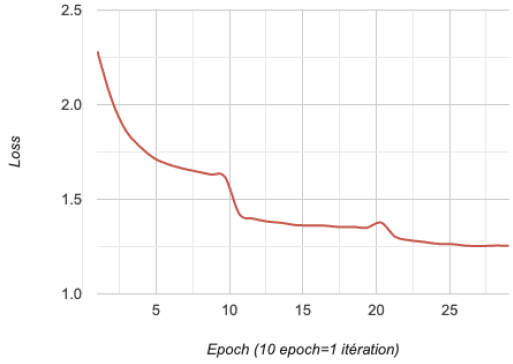


FIGURE 9: Totale loss

Après avoir décomposé la loss en policy loss et value loss, on observe que la policy loss est la plus importante.



(a) Evolution de la value loss



(b) Evolution de la policy loss

Un problème rencontré était le plafond atteint après la première itération. Notre modèle atteint en effet 40 égalités et n'est donc pas amélioré.

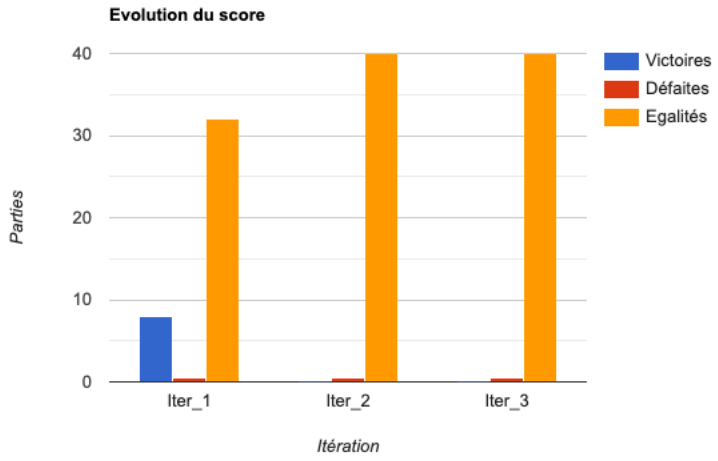


FIGURE 11: Performance du neural network contre sa version précédente

Cependant, il effectuait encore quelques erreurs, notamment contre l'homme. Après 40 parties jouées contre l'IA, le score était de 4 parties gagnées contre le joueur humain, 6 par l'intelligence artificielle, et 30 égalités. Nous avons donc réentrainé le network en utilisant un dropout et un learning rate plus important. Sans réussite puisqu'aucune des 10 itérations n'a été capable de battre de manière convainquante notre précédent modèle.

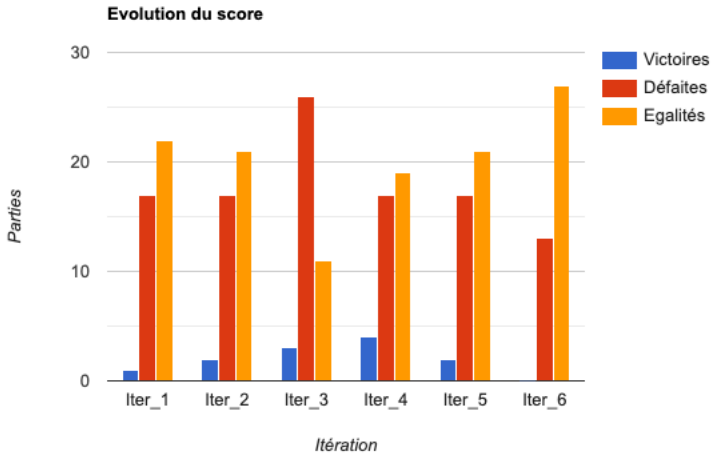


FIGURE 12: Avec un nouveau learning rate, le réseau obtenu n'est pas meilleur

Il est possible de jouer contre l'IA dans le fichier $play\_agains\_ai.py$ de notre github, ou à la fin du notebook $Demo.ipynb$.

5 Conclusion

Nous avons pu au cours de notre étude développer un algorithme identique à AlphaZéro pour un jeu beaucoup plus simple que son utilisation théorique. Sa philosophie et son fonctionnement simple, ainsi que sa portabilité à quasiment tous les jeux de plateau indiquent que cet algorithme est très prometteur. Au même titre que l'intelligence artificielle a bouleversée les échecs, les grands maîtres s'inspirent notamment beaucoup des nouvelles techniques inventées par l'IA, nous pensons qu'AlphaZéro peut virtuellement révolutionner n'importe quel jeu. Et il n'est pas étonnant qu'il ait pu être implémenté sur StarCraft[5]. Même si en l'occurrence l'algorithme ne partait pas de Zéro mais de parties jouées par des experts de ce jeu.
Nous relevons tout de même des éléments intéressant issus de l'article, les auteurs mentionnent la rapidité d'apprentissage d'AlphaZero (36 h pour atteindre AlphaGo), mais cette indication de durée n'a que peu de valeur quand on voit les importants moyens techniques développés (5000 TPUs). De plus, des restrictions de Stockfish mise en place par les auteurs (interdiction d'utiliser des ouvertures connues, temps fixe de 1min entre chaque coup), peuvent laisser à croire que la compétition avec Stockfish n'est pas encore finie. De notre côté, l'expérience d'être battu à un jeu aussi simple que celui du morpion nous a

montré la formidable performance de cet algorithme.
Enfin, si l'algorithme AlphaZero de DeepMind est un algorithme d'apprentissage général destiné à la maîtrise de jeux à deux joueurs, déterministes et en information parfaite, il a été adapté aux jeux multijoueurs et sous d'autres conditions[2].

Bibliographie

1. David Silver, Thomas Hubert, et Al., *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, DeepMind, 2018
2. Nick Pesota, Tucker Balch, *Multiplayer AlphaZero*, Georgia Institute of Technology, 2019
3. Tord Romstad, Marco Costalba, Joona Kiiski, et al. *Stockfish : A strong open source chess engine*, ,2017
4. David Silver et Al., *Mastering the game of Go without Human Knowledge*, 2017, DeepMind
5. Miguel Illescas, Garry Kasparov, *Alpha Zero*, 2018
6. James, Steven and Konidaris, George and Rosman, Benjamin, 2017, *An Analysis of Monte Carlo Tree Search*
7. David Foster, *Build your own Alpha Zero for Connect4*, Medium article, 2018
8. Wee Tee Soh *From-scratch implementation of AlphaZero for Connect4*, Towards Data Science, 2019
9. Sébastien Bubeck and Nicolò Cesa-Bianchi (2012), *Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems*, Foundations and Trends® in Machine Learning Vol. 5 : No. 1, pp 1-122.
10. A. Salomon, J. Y. Audibert, I. E. Alaoui, *Regret lower bounds and extended upper confidence bounds policies in stochastic multi-armed bandit problem*
11. J-Y. Audibert, S. Bubeck, and R. Munos. *Best arm identification in multi-armed bandits. In Conference on Learning Theory (COLT), 2010.*
12. Michèle Sebag, *Monte Carlo Tree Search*, 2012
13. David N. L. Levy and Monty Newborn, *How Computers Play Chess*, Ishi Press, 2009

Annexe

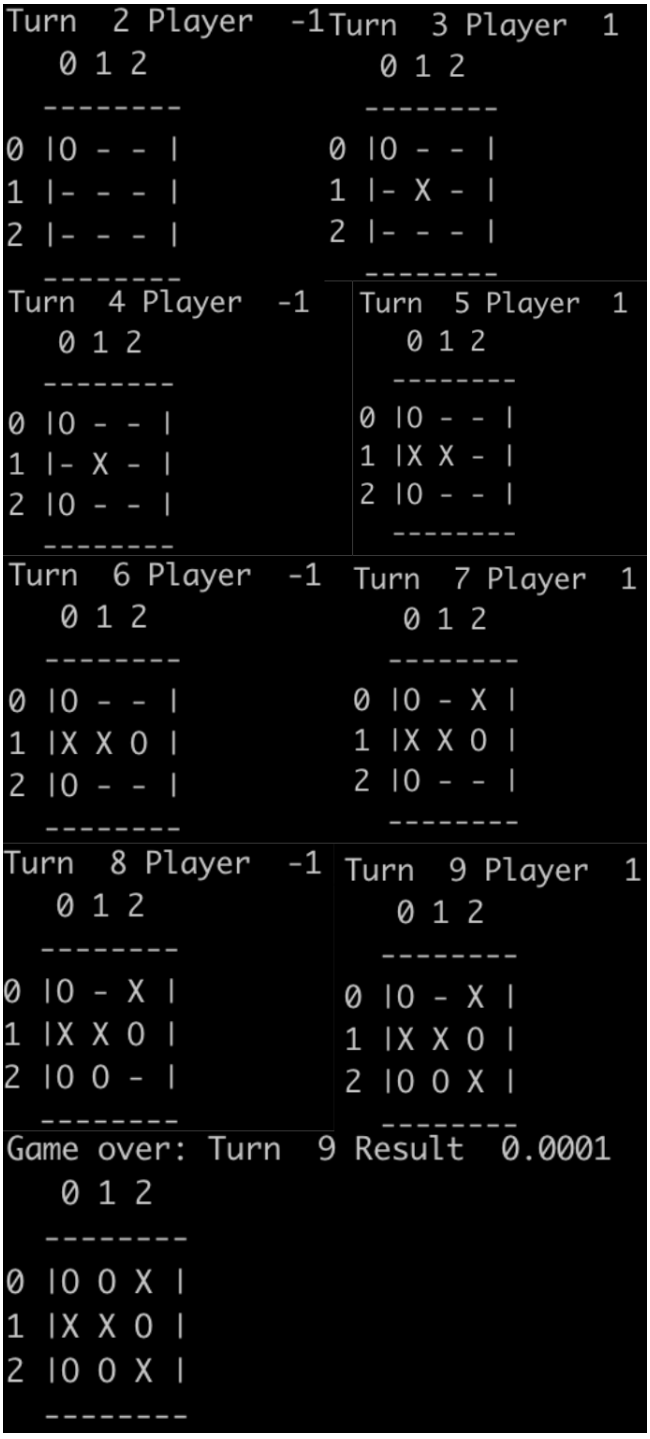


FIGURE 13: Déroulement d'une partie humain ordinateur, humain en O

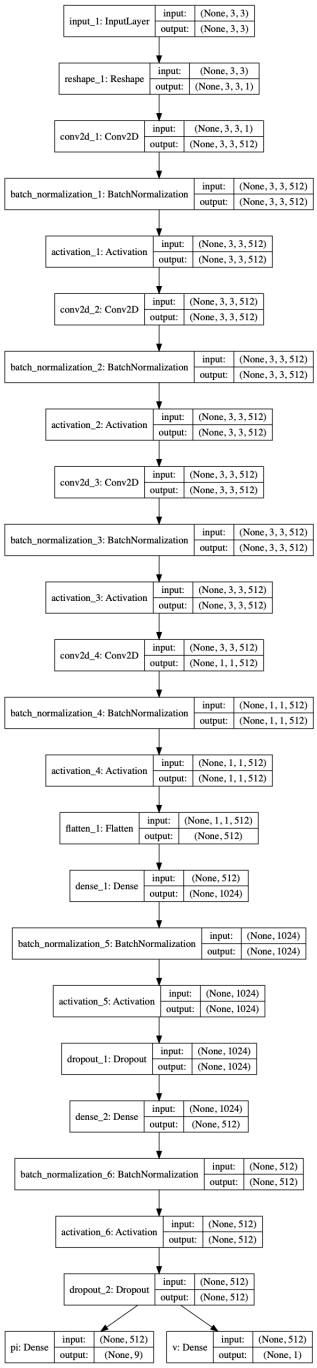


FIGURE 14: Architecture du modèle utilisé