Résolution du jeu de plateau Avalam par apprentissage profond : méthode basée sur Alpha Go Zero

Louis Maestrati, Zeryab Moussaoui

Abstract

Si le domaine d'application de l'intelligence artificielle est vaste, celui de la création d'algorithme de résolution de jeu en est l'un des plus populaires. L'importante surface mémoire et la subtilité des raisonnements qu'il faut pouvoir maîtriser pour exceller dans un jeu de plateau sont autant de défis aux réseaux développés jusqu'ici. Ces jeux dépassent ainsi rapidement la simple vision ludique que l'on peut leur accorder et se révèlent être de véritables moyens d'appréciation de notre avancé dans le domaine. Le jeux d' Othello, plus particulièrement, présentait un véritable défie : on dénombrait par exemple plus d'atomes dans l'univers (10^{80} contre 10^{170}) que de configurations du plateau possibles ! On s'intéresse ici à l'application à un jeu à une composante dynamique : le jeu AVALAM [1]. Contrairement au jeu d'Othello, il ne s'agit pas de placer des pièces sur le plateau mais bien de les déplacer, ce jeu propose ainsi une variante intéressante et originale à la problématique initiale du modèle d'Alpha Zero. Suivant ce modèle, nous avons implémenté un réseau neuronal retournant la politique de jeu apprise, et un Monte Carlo Tree Search, permettant d'affiner cette politique, défiant en permanence ses capacités de résolution, lui fournissant des éléments d'apprentissage nouveau. L'entraînement s'effectue donc à partir d'un réseau aléatoire, s'affinant au fur et à mesure qu'il joue contre lui-même par l'intermédiaire de l'arche de recherche. La forme particulière du jeu Avalam nous a cependant obligé d'utiliser une taille importante de plateau (9x9), augmentant, comme il le sera expliqué, la durée du training du réseau. Afin d'obtenir une évaluation du modèle, nous avons également implémenté d'autres politiques de jeu : une Greedy policy ainsi qu'une Random policy. Il est également possible de jouer soit même face à l'algorithme afin d'apprécier le raisonnement établi face à un joueur humain.

1. Introduction

L'un des premiers travaux dans cette méthode de résolution de jeu a été un moteur de dames développé en 2000, qui a appris par auto-apprentissage, et non par des méthodes basées sur des règles. Un premier triomphe fut Deep Blue (Campbell, Hoane et hsiung Hsu 2002), un programme informatique capable d'exécuter des performances surhumaines au jeu d'échec, battant les meilleurs joueurs humains. Ces jeux demeurent néanmoins relativement simples, où le nombre maximum de successeurs possibles d'un état est petit (facteur de branchement aux échecs : 35), et il est facile d'évaluer à quel point une position non-terminale est bonne. Il a été estimé que des jeux comme Go, qui ont un grand facteur de branchement (250), et où il est très difficile de déterminer le gagnant probable à partir d'une position non terminale, ne seraient pas résolus avant plusieurs décennies. Cependant, AlphaGo (Silver et al., 2016), qui utilise l'apprentissage par réseau profond et les méthodes de Monte Carlo Tree Search, a réussi récemment à vaincre le meilleur joueur humain, grâce à une connaissance acquise phénoménale. En ce qui concerne le jeu Avalam, le facteur de branchement est limité par le nombre maximum de positions initiales (48), qui ne peut que décroître au cours du jeu, multiplié par le nombre de déplacement possible ((1,1),(1,0),(1,-1),(0,-1),(-1,-1),(-1,0),(-1,1),(0,1)), soit un facteur de branchement initiale borné par 380. La particularité du jeu est que celle-ci diminue fortement et rapidement au cours de la partie, se rapprochant d'une moyenne de 60.

La plupart des approches existantes pour la conception de systèmes permettant de jouer à des jeux reposaient sur la disponibilité de connaissances de mouvements performants pour former le modèle et évaluer rapidement les états non terminaux.
AlphaGo Zero (Silver et al., 2017) décrit une approche qui n'utilise absolument aucune connaissance pré-établie et qui est entièrement formée par le biais du self-play. Ce nouveau système surpasse même le modèle AlphaGo précédent. Ceci représente un résultat très excitant, les ordinateurs étant capables de performances surhumaines uniquement par le biais de l'auto-apprentissage, et sans aucun

1. Travail encadré de recherche. Il s'agit d'un travail de trois mois, encadré par monsieur Pascal Yim effectué durant le long du deuxième semestre de deuxième année, à l'école Centrale de Lille.

guidage humain.

Dans notre travail, nous extrayons des idées du papier AlphaGo Zero et les appliquons au jeu Avalam. Nous utilisons des tailles de tableau de 9x9, pour lesquelles l'apprentissage par le biais du self-play n'est pas traitable par le biais d'un simple CPU. En effet la caractéristique dynamique d'Avalam impose de considérer comme action, non plus des positions de placements (limités à une taille de 19x19 pour le jeu de Go) mais bien des mouvements (soit une taille limite de 9x9x9x9). C'est cette dimension qui fait exploser la taille du problème et présente le challenge le plus élevé. En effet, comme il le sera plus explicitement décrit par la suite, le réseaux neuronaux traiteront en sortie des vecteurs de cette taille, et le MCTS usera de vecteurs de probabilité de transition de cette même taille. Pourquoi a-t-il été fait ce choix au vue de la remarque précédente, stipulant que le facteur de branchement diminuait rapidement ? Car la forme du plateau ne permet pas de rendre usuelle une diminution progressive de la taille des actions acceptables... Elle ne présente aucune symétrie et les règles de déplacement rendent essentielle l'indexation des actions dans les tableaux utilisés !

Pour l'évaluation, nous comparons nos agents formés à des agents de bases aléatoires et avides. Nous faisons jouer également le modèle face à nous même afin de constater la présence ou non de performances surhumaines.

2. Etat de l'art

L'usage du self-play pour un apprentissage du jeu optimal a été largement étudié. Par exemple, le jeu de Go a été étudié dans (Gelly et Silver 2008). Les échecs, bien que largement joués en utilisant des stratégies de recherche alpha-bêta, ont également été programmés grâce à des méthodes de self-play dans (Heinz 2001). (Wiering 2010) étudient le problème de l'apprentissage du Backgammon par le biais d'une combinaison de méthodes d'auto-apprentissage et de connaissances spécialisées. En particulier, (Van Der Ree et Wiering 2013) apprennent à jouer à Othello par le biais de méthodes de self-play, et (Nijssen 2007) appliquent les méthodes de Monte Carlo à Othello. Par ailleurs, ce projet s'inspire et s'aide fortement du travail réalisé par messieurs Surag Nair, Shantanu Thakoor, Megha Jhunjhunwala en 2017 ; dont l'étude d'un jeu d' Othello 6x6 a été la base de mon travail. Ceux-ci ont établi une méthode de self-play en se basant sur le modèle d'Alpha Zero.

3. Le jeu Avalam

Le jeu avalam est constitué d'un plateau oblongue dans lequel se trouvent des pièces noires et blanches empilées sur des cases. Le jeu se joue à deux et, si chacun des joueurs peut déplacer n'importe quelle couleur de jeton, il leur est associé respectivement une couleur à chacun.
Le partie consiste en ce que chaque joueur déplace à tour de rôle des pièces d'une case à une autre. Celles-ci ne peuvent être déplacées uniquement que dans des cases non vides et à la restriction que :
-l'ensemble des pièces de la case d'origine doit être déplacé au cours d'un déplacement
-les empilements de pièces ne peuvent excéder une hauteur de 5.
-les pièces d'une case ne peuvent être déplacées que vers une case voisine (en diagonale, horizontal, ou vertical)

Il est interdit de passer son tour et le décompte des points est effectué à la fin de la partie (plus de coups jouables) en considérant les couleurs des pièces situé au sommet des tours formées.
Si le joueur s'est vu attribué la couleur noire, il gagne un point pour une tour à sommet noire et ainsi de suite...
En cas d'égalité, on ne compte que les tours de taille 5 !

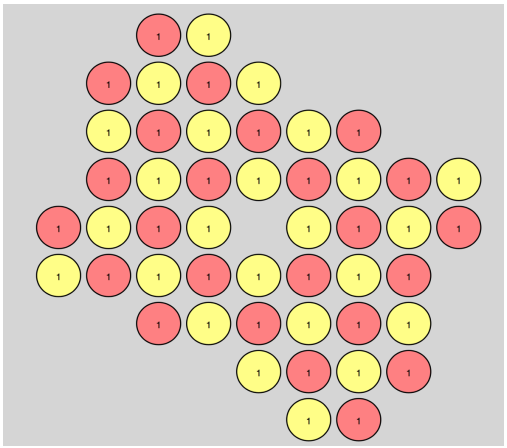

Figure 1: Représentation du plateau à l'état initial (début de partie)

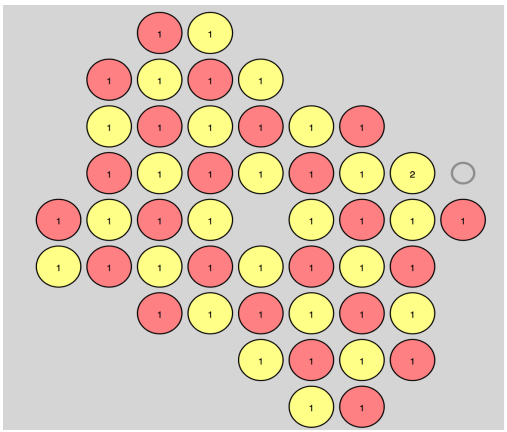

Figure 2: Représentation du plateau après un déplacement

4. Travail réalisé

4.1. Méthode générale

Comme décrit plus haut, l'algorithme ne s'entraîne qu'en jouant avec lui-même.

Deux principaux constituants sont à signaler :
Tout d'abord un réseau neuronal prenant en entrée le plateau et renvoyant deux sorties : une politique de jeu décrit par un vecteur de probabilité $\vec{p}$ et une valeur v associés à l'état du plateau.
Cette dernière correspond à une indexation de la proximité entre l'état courant du plateau à un état d'échec ou de gain de la partie. Ce réseau est notre banque de données, la mémoire de l'apprentissage du jeu effectué lorsque l'on fait jouer l'algorithme contre lui-même.
La deuxième principale composante, allant de paire avec le réseau entraîné, est le MCTS, acteur véritable du jeu, permettant :
- de jouer de manière active au jeu en usant initialement de la politique renvoyée par le réseau neuronal
-affiner cette politique par un jeu de poids de transition dépendant récursivement des valeurs de gain ou de perte obtenue en fin de partie
- l'établissement d'un trainset donné à apprendre au réseau lorsque l'on a effectué un certain nombre de parties (fixé à 70), le réseau défiant alors ce nouveau réseau obtenu au cours de nbparties (fixé à 40). Si il est moins performant, il est remplacé.

4.2. Le réseau neuronal

Le réseau est constitué de 4 CNN placés en série, suivie de deux couches de sorties, deux MLP à 1 seul neurone mis en parallèle.
Nous noterons l'indice indexant le réseau θ. Le réseau prend en entrée l'état du plateau s et renvoie respectivement :

- Une valeur v présentant la valeur continue associée à l'état de jeu/plateau, avec $v_\theta \in [-1, 1]$, -1 correspondant à la valeur de défaite, 1 à la valeur de gain pour un état de plateau terminal.

- un vecteur de probabilité $\vec{p_\theta}$ qui décrit la politique d'action à suivre en fonction de l'état du plateau placé en entrée. Sa taille est donc celle du nombre d'actions maximale depuis un état (soit $9^4 = 6561$), et l'écriture en base 9 de l'indice de l'action fournit sa description complète : (i,j,k,l), le déplacement $(i, j) \to (k, l)$

La boucle principale de notre algorithme est celle qui fournie un trainset à ce réseau. Initialisé de façon random, il reçoit à chaque itération des exemples de la forme $(s, \vec{\pi}, v)$, valeurs transmises grâce à l'utilisation du MCTS dont le rôle est d'affiner la politique de jeu au cours de son utilisation par self play.

Le réseau neuronale est alors entraîné de sorte à affiner la fonction de coût suivante :

$$C = \sum_t (v_\theta(s) - z)^2 + \log\left(\vec{p_\theta}(s)\right) * \vec{\pi_\theta}(s)$$

La forme du plateau a dû être agrandie, pour raison d'accessibilité des positions, et facilitation du code, nous en avons fait une matrice 9x9, en complétant les positions par des zéros inaccessibles.

```
[ [ 0,  0,  1, -1,  0,  0,  0,  0,  0],
  [ 0,  1, -1,  1, -1,  0,  0,  0,  0],
  [ 0, -1,  1, -1,  1, -1,  1,  0,  0],
  [ 0,  1, -1,  1, -1,  1, -1,  1, -1],
  [ 1, -1,  1, -1,  0, -1,  1, -1,  1],
  [-1,  1, -1,  1, -1,  1, -1,  1,  0],
  [ 0,  0,  1, -1,  1, -1,  1, -1,  0],
  [ 0,  0,  0,  0, -1,  1, -1,  1,  0],
  [ 0,  0,  0,  0,  0, -1,  1,  0,  0] ]
```

FIGURE 3: Représentation du plateau à l'état initial sous forme de tableau

4.3. Le Monte Carlo Tree Search

C'est l'élément central de l'algorithme, celui dont le rôle est d'affiner la politique de jeu donné à apprendre au réseau. Le MCTS est un algorithme de recherche qui équilibre l'exploration et l'exploitation pour produire une politique améliorée après un certain nombre de simulations du jeu.

4.3.1. La structure du MCTS

Le MCTS est un arbre dont les nœuds représentent différentes configurations du plateau, une arrête $(i \to j)$ représente une action valide permettant de passer du plateau i au plateau j.

Pour chaque arête, nous maintenons à jour plusieurs valeurs :
-une valeur Q notée $Q(s, a) \in [-1, 1]$ qui est la récompense moyenne attendue pour cette action (à la fin de la partie)
-N(s,a) qui représente le nombre de fois que l'on a effectué l'action a depuis l'état s.
-Nous gardons aussi trace de $P(s, a) = \vec{p_\theta}(s)$, probabilité de transition renvoyée par le réseau neuronal.

Cet ensemble de valeurs nous permet d'obtenir U(s,a) qui pondère notre MCTS et décrit la descente que l'on y effectue afin d'obtenir la valeur auxiliaire $v_{aux} \in \{-1, 1\}$ correspondant à la valeur finale du jeu vers lequel "se dirige" l'action a depuis l'état s et qui permet de mettre à jour les précieux paramètres pondérants définis plus haut. U(s,a) est définit ainsi :

$$U(s, a) = Q(s, a) + c_{exp} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

Ici, c_{exp} est un hyperparamètre contrôlant le degré d'exploration (que nous avons fixé à 1.0 dans nos expériences).

Revenons sur cette valeur utile v_{aux} paramétrant les poids de l'arbre.
Le fonctionnement globale de la boucle MCTS est le suivant :
L'état d'entrée s dont on veut obtenir une valeur v_{aux} associée est matché avec les états présents dans l'arbre, si le MCTS possèdent déjà cet état en mémoire , il se rend à un état fils associé à l'action maximisant la valeur U(s,a) et ainsi de suite... jusqu'à un état terminal de gain/perte associé à une valeur $v_{aux} \in \{-1, 1\}$, cette valeur étant associée ensuite par remonté récursive à l'état s d'entrée.
Si cependant, au cours de la descente, le prochain état s ne forme pas un noeud du graphe, il est créé et la boucle s'arrête en renvoyant la valeur $v_{aux} = v_\theta(s)$ apprise par le réseau.

Au cours de cette descente, il est évident que les valeurs N(s,a), Q(s,a) sont modifiés en conséquence récursivement, ce sont ces actions qui permettent l'affinement de la politique de jeu du couple (MCTS/réseau). Si t est le prochain sommet désigné par U(s,a), il vient :

$$Q(s,a) = \frac{N(s,a) * Q(s,a) + MCTS(t)}{N(s,a) + 1}$$

$$N(s,a) = N(s,a) + 1$$

Il faut bien percevoir que cette valeur v_{aux} n'est en aucun cas celle qui sera attribuée plus tard dans le trainset à l'état s, elle n'a qu'un rôle de "mise à jour" du MCTS : parfois issu du réseau qui devient de plus en plus performant et fin dans son indexation des états, parfois issu de la valeur binaire que l'on attribue à un état terminal rencontré dans la descente , il sert avant tout à affiner les paramètres de l'arbre (et plus particulièrement ceux du noeud associé à l'état s dont part la descente), le but ultime étant que ces deux valeurs prises potentiellement (réseau/terminal) se confondent au cours de l'apprentissage (même si c'est inenvisageable évidemment)... Une valeur terminale $v_{aux} \in \{-1, 1\}$ est une **vérité absolue**, nous sommes certains de son association à un gain/perte, la valeur $v_{aux,\theta}$ renvoyé par le réseau est son approximation.
On peut voir cette chimère (de la correspondance entre les deux valeurs) comme l'image d'un joueur sûr de lui à toute étape de jeu, il sait pertinemment ce que son action engendrera dans le futur comme gain ou perte, car il anticipe les actions choisis par son adversaire qu'il suppose prendre toujours les meilleurs décisions (semblable à lui finalement). On voit ainsi réapparaître la façon dont il s'entraîne : face à lui même.
Afin de favoriser l'exploration, et d'empêcher l'arbre de nous renvoyer trop souvent la valeur v_θ du réseau en tant que valeur v_{aux}, nous décidons d'effectuer pour chaque état du self play plusieurs simulations (numSims=20) du MCTS, ceci favorise le développement de notre arbre.

4.3.2. Le MCTS durant le self play

Les grands nombres d'itérations successives de l'arbre MCTS depuis différents états donnés s laissent toujours

Algorithm 1 Monte Carlo Tree Search

procedure MCTS(s, θ)
  if *s is terminal* **then**
    return *game_ result*
  if $s \notin$ *Tree* **then**
    Tree $\leftarrow$ *Tree* $\cup\, s$
    $Q(s, \cdot) \leftarrow 0$
    $N(s, \cdot) \leftarrow 0$
    $P(s, \cdot) \leftarrow \vec{p_\theta}(s)$
    return $v_\theta(s)$
  else
    $a \leftarrow \text{argmax}_{a' \in A} U(s, a')$
    $s' \leftarrow$getNextState(s, a)
    $v \leftarrow$MCTS(s')
    $Q(s,a) \leftarrow \frac{N(s,a)*Q(s,a)+v}{N(s,a)+1}$
    $N(s,a) \leftarrow N(s,a) + 1$
    return v

Figure 4: Boucle du MCTS du MCTS

notre triplet d'apprentissage fortement incomplet (s_,_). En effet lorsqu'un nombre suffisant de simulations MCTS ont été effectué (y compris lors des parties antérieurs...), les valeurs N(s,a) fournissent une bonne approximation pour le processus stochastique optimal de choix d'action.

Par conséquent, cette valeur fournie un vecteur de probabilité de transition idoine (toujours de taille 9x9x9x9) :

$$tab_s = [N(s,a)^{1/t}, \text{pour a} \in 6561]$$

$$\pi_s = [\frac{tab_s(a)}{\sum_{b \in 6561} tab_s(b)}, \text{pour a} \in 6561]$$

t est un paramètre de température (hyperparamêtre d'exploration) : initialisé à un 1 dans nos tests, il décrit le rapprochement avec une politique "greedy", correspondant à t=0, et à une politique random, pour $t+ \to \infty$.

Ce vecteur sera ainsi utilisé principalement pour deux rôles principaux :

-Tout d'abord lors du self play du MCTS avec lui même bien sûr : Reprenons l'action qui se déroule lorsque le MCTS effectue une phase de self play : à partir d'un état courant s, la boucle MCTS (s, θ) met à jour notre arbre de recherche par exploration. Le MCTS doit alors choisir l'action finale qu'il choisit d'effectuer afin de prolonger son jeu contre lui même, celle-ci s'effectue suivant le vecteur π_s.

-comme son nom l'indique, elle est également la politique de jeu π_s apprise par le réseau neuronale et vient compléter le triplet d'apprentissage $(s, \pi_s, \_)$

Ainsi au fur et mesure que le MCTS joue face à lui même (suivant la politique π_s qui ne cesse d'évoluer...),

nous obtenons une batterie incomplète de triplets $(s, \pi_s, \_)$ qui seront complétés à la toute fin de la partie. Le résultat final r du dernier état du plateau permet de définir le résultat de la politique du MCTS face à lui même, les suites pairs et impairs d'action (on a deux joueurs qui utilisent l'arbre en réalité) étant donc logiquement assignés à cette valeur au signe près :

$$[(s, \pi_s, v)][i] = [(s, \pi_s, \_)][i] \leftarrow r * (-1)^i$$

Un self play est nommé épisode. Soit n le nombre d'état rencontrés lors de la partie. En plus de développer notre arbre à chaque état rencontré ($n * numSims$ descentes), il fournie donc n triplets d'apprentissage (s, π_s, v).

Algorithm 3 Execute Episode

```
procedure ExecuteEpisode(θ)
    examples ← []
    s ← gameStartState()
    while True do
        for i in [1, ..., numSims] do
            MCTS(s, θ)
        examples.add((s, π_s, _))
        a* ~ π_s
        s ← gameNextState(s, a*)
        if gameEnded(s) then
            //fill _ in examples with reward
            examples ←assignRewards(examples)
            return examples
```

Figure 5: Boucle d'un épisode : une partie en self play qui forme des exemples pour le trainset

4.3.3. L'établissement d'un trainset complet et mise à jour du réseau neuronale

Il s'agit de la boucle principale de notre algorithme, permettant la mise à jour de notre réseau.
Son fonctionnement est simple :
A partir du réseau courant, un MCTS est créé et joue numEpisodes parties face à lui même. Ceci complète notre base d'exemples de $numEpisodes * n_{episode}$ triplets d'apprentissage. Une fois cette complétion effectuée, le réseau est réentraîné avec ce nouveau trainset de tailles plus importante. Vient alors une compétition entre les deux réseaux θ_1 et θ_2(ancien et nouveau). On crée deux MCTS (MCTS(θ_1), MCTS(θ_2)) et les faisons s'affronter [2] de manière gloutonne, suivant la même politique que lors de la phase d'entraînement, sur 40 parties... Soit n_{res} (p_{res}) le nombre de parties gagnantes du nouveau réseau, n_{nul} le nombre de parties nuls. Si $\frac{n_{res}}{p_{res}+n_{nul}} > \omega$ [3], alors le nouveau réseau remplace le nouveau. Notre système d'apprentissage s'en trouve amélioré !

Un dernier paramètre important est à signaler : comme il l'est clairement expliqué, ce modèle stipule que le trainset absorbe de manière systématique les nouveaux triplets (s, π_s, v) rencontrés. Si cette structure de retraining est classique du domaine du Deep Reinforcement Learning, elle présente un inconvénient intuitif (autre que celui de saturer l'espace mémoire alloué) : au fur et à mesure du ré-entraînement du réseau et des parties effectuées, il peut y avoir redondance des couples (s,a) rencontrés et surtout redondance contradictoire, les triplets de début d'entraînement issus des premières mises à jour de mon réseau sont mises au même plan que les plus récents... surtout le nombre de triplets les plus "fins" rajoutés au trainset lors d'une itération globale est quasi invariante ($numEpisodes * n_{episode}$, et nous avons observé logiquement que $n_{episode}$ ne cesse de diminuer au cours des itérations, le réseau devenant plus performant, pour illustration cf figure 7) et celles-ci deviennent a fortiori négligeable au vue de la taille importante acquise par le trainset. Nous avons ainsi décidé d'introduire un paramètre limitant B définissant la taille maximale conférée à celui-ci. Si sa taille dépasse la valeur B (fixée à 200 000) les nouveaux triplets remplacent les plus anciens !

Algorithm 2 Policy Iteration through Self-Play

```
procedure PolicyIterationSP
    θ ←initNN()
    trainExamples ← []
    for i in [1, ..., numIters] do
        for e in [1, ..., numEpisodes] do
            ex ← executeEpisode(nn)
            trainExamples.append(ex)
        θ_new ← trainNN(trainExamples)
        if θ_new beats θ ≥ thresh then
            θ ← θ_new
    return θ
```

Figure 6: Boucle d'un épisode : une partie en self play qui forme des exemples pour le trainset

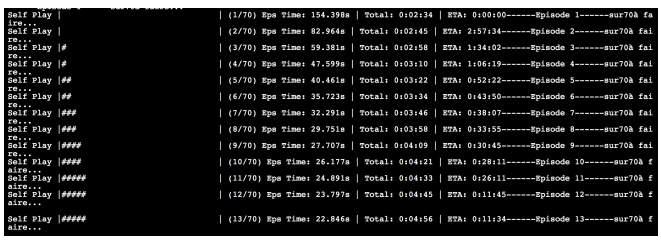

Figure 7: Illustration de l'augmentation de rapidité de résolution d'un épisode (self play)

4.4. Une allure globale du training

Un schéma présentant une allure globale du training :

2. On aurait pu faire le choix de se faire affronter les réseaux directement lors de cette phase de concurrence des réseaux.
3. ω est la valeur référence de mise à jour, fixé à 0.55 dans le modèle d'AlphaZero, nous l'avons fixé à 0.6

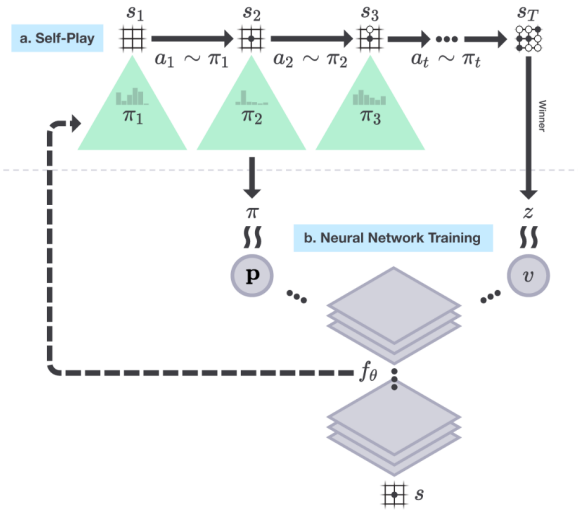

FIGURE 8: Allure globale du training

5. Quelques premiers résultats

La principale difficulté rencontrée fut celle d'allouer un serveur gpu à notre apprentissage (nous travaillions initialement sur le CPU de notre ordinateur). Par manque de temps nous n'avons pu pour l'instant effectuer que 16 boucles principales de 70 épisodes (parties).
Présentons quelques résultats comparatifs de performance :
En présence :

-le réseau neuronal initial avant entraînement (r_0)
-le réseau entraîné 1h30 sur un gpu, ayant effectué 4 itérations (r_4)
-le réseau entrainé 6h sur un gpu, ayant effectué 16 itérations (r_{16})
-une politique "greedy"
-une politique "random"
On effectue 40 parties entre eux, les résultats sont les suivants :

...vs...	r_0	r_4	r_{16}	greedy	random
r_0	ne	13g/26d/1n	4g/35d/1n	0g/40d/0n	ne
r_4	26g/13d/1n	ne	10g/28d/2n	0g/40d/0n	ne
r_{16}	35g/4d/1n	28g/10d/2n	14v/22d/4n	1g/39d/0n	34g/6d/0n
random	ne	ne	6g/34d/0n	0g/40d/0n	ne
greedy	40g/0d/0n	40g/0d/0n	39g/1d/0n	20g/20d/0n	40g/0d/0n

(ne :non exécuté, g :gagnant, p :perdant, n :nul)

Ainsi on obtient bien un modèle en constante progression (dont les performances dépassent de loin celle d'une politique random), cependant le nombre d'itérations pour l'instant effectué est trop faible pour défier la politique greedy, beaucoup plus performante pour l'instant.
Notre gpu est en ce moment même en train d'entraîner notre réseau, nous espérons obtenir un réseau entraîné sur au moins 80 itérations principales, afin d'obtenir des résultats plus conséquents.

6. Annexe

Le print que nous avons fait affiche ceci lors de l'entraînement (1ère itération uniquement) :

`https://drive.google.com/open?id=1siHSxJVsqByebdRV_SL-eimgbs-crCxPFcfSuQ-FkjI`

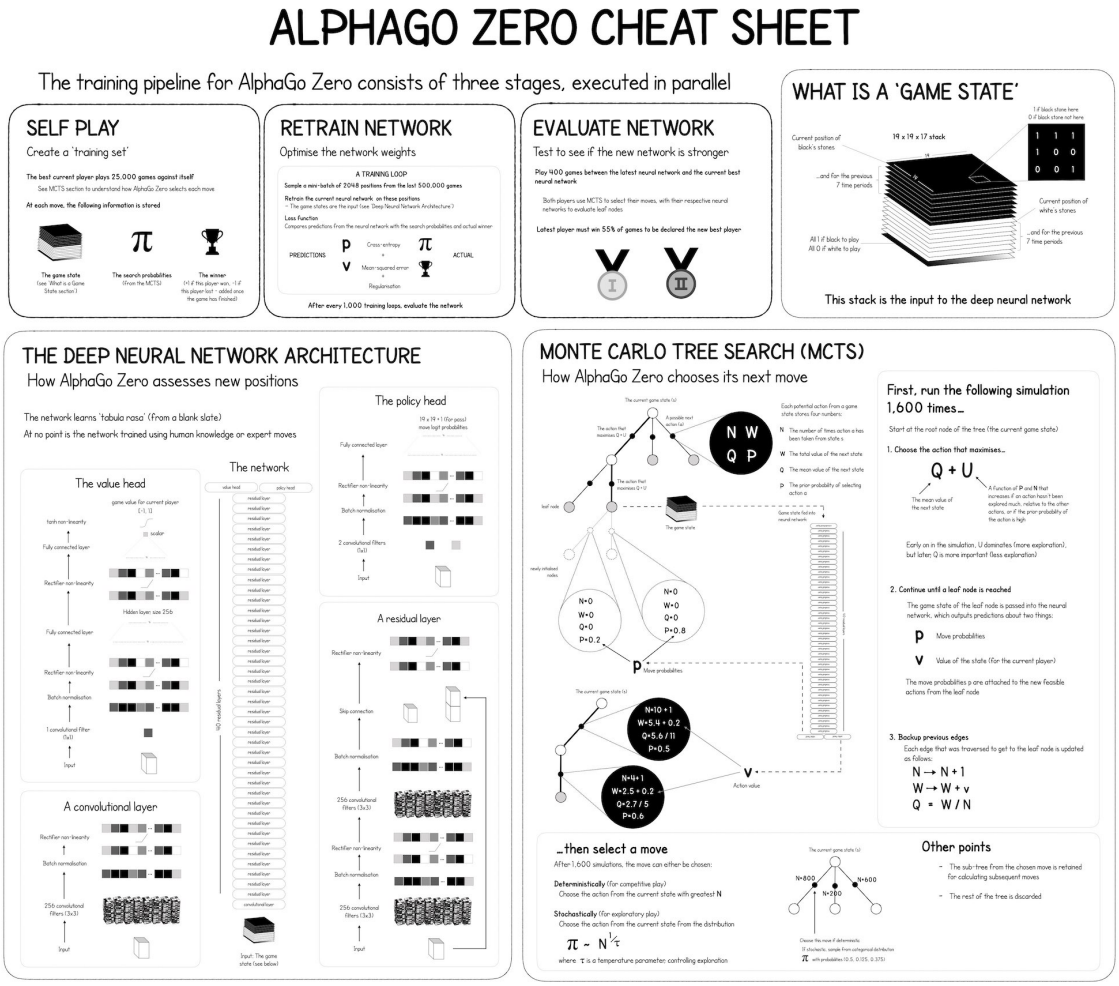

Figure 9: Une petite feuille explicative

7. Bibliographie

[Yoshua Bengio 2016] Deeplearningbook
Machine Learning Course by Andrew Ng (Stanford)
Surag Nair «Alpha Zero Général», `https://github.com/suragnair`
[Heinz 2001]Heinz, E. A. 2001. New Self-Play Results in Computer Chess. Berlin, Heidelberg : Springer Berlin Heidelberg.
[Ioffe and Szegedy 2015] Ioffe, S., and Szegedy, C. 2015. Batch normalization : Accelerating deep network training by reducing internal covariate shift. In International Conference on Machine Learning.
[Kingma and Ba 2014] Kingma, D., and Ba, J. 2014. Adam : A method for stochastic optimization. [Nijssen 2007] Nijssen, J. 2007. Playing othello using monte carlo. Strategies.
[Silver et al. 2016] Silver, D. ; Huang, A. ; Maddison, C. J. ; Guez, A. ; Sifre, L. ; van den Driessche, G. ; Schrittwieser, J. ; Antonoglou, I. ; Panneershelvam, V. ; Lanctot, M. ; Dieleman, S. ; Grewe, D. ; Nham, J. ; Kalchbrenner, N. ; Sutskever, I. ; Lillicrap, T. ; Leach, M. ; Kavukcuoglu, K. ; Graepel, T. ; and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search.
[Silver et al. 2017] Silver, D. ; Hubert, T. ; Schrittwieser, J. ; Antonoglou, I. ; Lai, M. ; Guez, A. ; Lanctot, M. ; Sifre, L. ; Kumaran, D. ; Graepel, T. ; Lillicrap, T. ; Simonyan, K. ; and Hassabis, D. 2017a. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. ArXiv e-prints. [Silver et al. 2017] Silver, D. ; Schrittwieser, J. ; Simonyan, K. ; Antonoglou, I. ; Huang, A. ; Guez, A. ; Hubert, T. ; Baker, L. ; Lai, M. ; Bolton, A. ; et al. 2017b. Mastering the game of go without human knowledge. Nature 550(7676).
[Srivastava et al. 2014] Srivastava, N. ; Hinton, G. E. ; Krizhevsky, A. ; Sutskever, I. ; and Salakhutdinov, R. 2014. Dropout : a simple way to prevent neural networks from overfitting. Journal of machine learning research.
[Van Der Ree and Wiering 2013] Van Der Ree, M., and Wier- ing, M. 2013. Reinforcement learning in the game

of othello : learning against a fixed opponent and learning from self- play. In Adaptive Dynamic Programming And Reinforcement Learning.
[Wiering 2010] Wiering, M. A. 2010. Self-play and using an expert to learn to play backgammon with temporal difference learning. Journal of Intelligent Learning Systems and Applications.