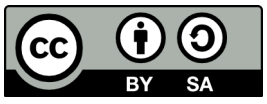
Deep learning et apprentissage par renforcement pour la conception d'une Intelligence Artificielle pour le jeu Yokai No Mori

David Roche



Remerciements

Je tiens à remercier Raphaël Couturier et Michel Salomon pour l'aide qu'ils m'ont apportés tout au long de la mise au point de ce document.

Table des matières

1 Introduction

En 1997 le super ordinateur d'IBM, Deep Blue, battait le champion du monde d'échec Gary Kasparov pour la première fois (deux parties à quatre). Deep Blue utilisait un algorithme relativement classique, l'algorithme Minimax couplé avec une énorme base de données de 60 000 parties.

Il a fallu attendre presque 30 ans pour voir l'ordinateur s'imposer face au multiple champion du monde de Go, Lee Sedol. Pourquoi tant de temps ? Simplement parce que le jeu de Go (Figure 1) n'est pas adapté aux méthodes algorithmiques « classiques » comme Minimax. Ce jeu est beaucoup plus complexe avec notamment un nombre de coups possibles qui dépasse l'entendement (10^{170}). Il est aussi extrêmement difficile de déterminer si une position donnée est plutôt favorable aux blancs ou plutôt favorable aux noirs.

FIGURE 1 – Jeu de Go

David Silver et son équipe (Figure 2) de la société Deep Mind ont donc dû mettre au point de nouvelles méthodes afin de développer un programme capable de rivaliser avec les meilleurs joueurs de Go de la planète. AlphaGo, le programme qui a battu Lee Sedol (quatre parties à une), utilise deux réseaux de neurones et une recherche arborescente de Monte-Carlo, le tout utilisé pour faire de l'apprentissage par renforcement [1].

AlphaGo Zero [2] est une évolution majeure d'AlphaGo. En effet, avec AlphaGo, une partie de l'entraînement des réseaux de neurones était réalisée grâce à une base de données de parties de Go réalisées par des experts humains. AlphaGo Zero n'utilise aucune connaissance humaine préalable, seules les règles du jeu sont programmées. AlphaGo Zero a très rapidement pris le dessus sur AlphaGo, montrant l'intérêt de ce système.

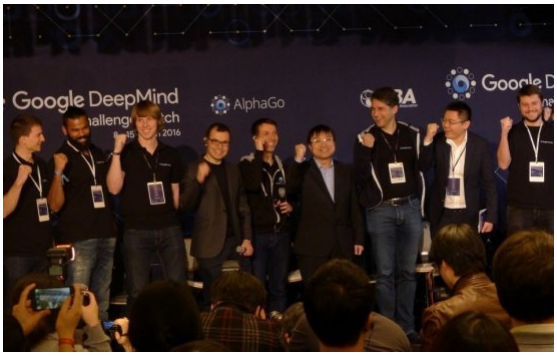


FIGURE 2 – Équipe AlphaGo de Deep Mind

Fin 2017 une équipe de DeepMind, toujours dirigée par David Silver, a appliqué les concepts mis au point avec AlphaGo Zero afin de créer une intelligence artificielle (AlphaZero) capable de jouer aux Echecs et au Shogi [3] avec, ici aussi, un succès remarquable.

Dans ce projet nous allons utiliser les recherches menées pour AlphaZero, afin de mettre au point une intelligence artificielle capable de joueur au Yokai No Mori. Dans un premier temps nous effectuerons des rappels sur les principales notions utilisées dans AlphaZero : l'apprentissage par renforcement, les réseaux neuronaux profonds et les arbres de recherche de Monte-Carlo. Ensuite, nous nous intéresserons à l'implémentation des algorithmes utilisés dans AlphaZero. Enfin, après avoir exposé le principe du jeu Yokai No Mori, nous adapterons ces algorithmes afin de créer une intelligence artificielle capable de jouer au Yokai No Mori.

2 État de l'art

2.1 Apprentissage par renforcement

2.1.1 Principe

L'apprentissage par renforcement est à classer dans les systèmes d'apprentissage automatique. Il a été popularisé en 1998 par Sulton et Barto avec leur livre « An introduction to Reinforcement Learning » [4]. C'est un apprentissage non supervisé qui fonctionne avec un système de récompense. Au cours d'un épisode, un agent va effectuer des actions en fonction des informations en provenance d'un environnement (l'environnement correspond à l'ensemble de

l'univers à l'exception de l'agent). L'agent va donc recevoir des récompenses en fonction de « l'efficacité » des actions entreprises. Personne ne dit à l'agent quelle action il doit effectuer à un instant t. Grâce à ce système de récompense, l'agent devra découvrir par lui-même les actions à entreprendre en fonction de la situation. Nous avons donc, avec l'apprentissage par renforcement, un système basé sur la « recherche par essai-erreur ». Une action qui se révélera à plus ou moins long terme peu judicieuse sera sanctionnée par une récompense négative (par exemple -1). A contrario, si l'action effectuée s'avère positive (toujours à plus ou moins long terme), l'agent recevra une récompense positive (par exemple +1). À chaque pas de temps t, l'agent reçoit une représentation de l'état de l'environnement notée s_t. L'ensemble du processus est résumé par la Figure 3

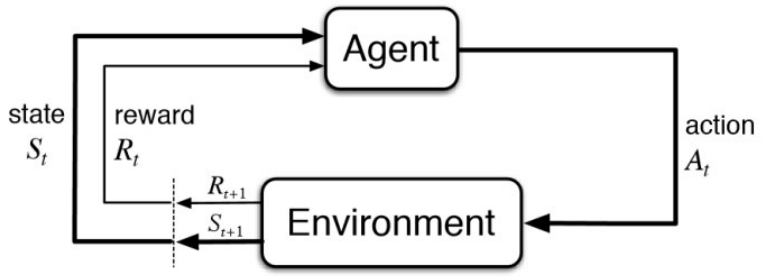


FIGURE 3 – Principe de l'apprentissage par renforcement

2.1.2 Processus markovien

On parlera d'état markovien si $\mathbb{P}[s_{t+1}|s_t] = \mathbb{P}[s_{t+1}|s_1, s_2, s_3, ...s_t]$. Autrement dit, la probabilité de se retrouver dans l'état s_{t+1} sachant que l'on se trouve dans l'état s_t, dépend uniquement de s_t et pas des états antérieurs (de s_1 à s_{t-1}). Dans un processus markovien, le futur dépend uniquement du présent et pas du passé ; remonter dans le passé n'apporte pas d'information. Dans le cadre de l'apprentissage par renforcement on utilisera des processus markovien.

2.1.3 Politique suivie par un agent

L'agent, pour choisir l'action à effectuer depuis l'état s, va suivre une politique $\pi(s|a)$ qui pour un état s renvoie la probabilité de choisir l'action

$a : \pi(s|a) = \mathbb{P}(A = a|S = s)$ avec A l'ensemble des actions possibles et S l'ensemble des états possibles. L'agent va changer de politique au cours du temps en fonction de l'expérience acquise.

2.1.4 Récompense reçue par un agent

On définit G_t la somme des récompenses futures qui seront reçues par l'agent depuis l'instant t. Nous avons :

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

avec R les récompenses reçues aux différents instants et γ le taux d'escompte : une récompense reçue dans k pas de temps dans le futur vaut γ^{k-1} fois ce qu'elle aurait valu si elle était reçue immédiatement. Nous avons $0 \leq \gamma \leq 1$, plus γ est petit et moins la récompense reçue dans le futur aura de l'importance dans la politique choisie par l'agent.

2.1.5 Fonction de valeur et équation de Bellman

$V^\pi(s)$ est la fonction de valeur, nous avons $V^\pi(s) = \mathbb{E}[G_t|s = s_t]$. La fonction de valeur représente donc les récompenses espérées dans le futur, sachant que nous sommes actuellement dans l'état s_t. Nous avons donc aussi $V^\pi(s) = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|s = s_t]$, soit après quelques lignes de calculs : $V^\pi(s) = \mathbb{E}[R_{t+1} + \gamma V^\pi(s_{t+1})|s = s_t]$. On nomme cette équation l'équation de Bellman. Il existe différentes méthodes afin de résoudre l'équation de Bellman, mais ces méthodes ne seront pas abordées dans ce document (voir [4] pour en savoir plus). De manière similaire on trouve la fonction $Q(s, a)$ qui donne la somme des récompenses futures espérées par l'agent à partir de l'état courant s s'il effectue l'action a. Nous aurons l'occasion d'utiliser certaines notions vues dans cette partie (par exemple $\pi(s|a)$ et $Q(s, a)$) plus loin dans ce document.

2.1.6 « Exploration versus Exploitation »

Le dilemme « Exploration versus Exploitation » est un aspect fondamental de l'apprentissage par renforcement. Faut-il choisir d'exécuter une série d'actions déjà explorée par l'agent au préalable où la récompense finale est déjà plus ou moins connue et ainsi exploiter les connaissances déjà acquises ?

Ou choisir d'effectuer des enchaînements d'action inconnus jusqu'alors avec, au bout, une récompense qui pourrait potentiellement être intéressante ? Dans le travail présenté dans ce document, certains paramètres permettront de privilégier l'exploitation ou l'exploration. Le choix de ces paramètres sera discuté plus loin.

2.2 Les réseaux de neurones

2.2.1 Le neurone formel

C'est en s'inspirant du neurone biologique que McCulloch et Pitts ont eu l'idée du neurone formel (Figure 4) dès 1943 [5]. Soit un vecteur $\vec{x}$ de dimension m, de coordonnées $x_1, x_2, \ldots, x_m$ et $\hat{y}$ un scalaire. Le vecteur $\vec{x}$ constitue l'entrée du neurone formel et le scalaire $\hat{y}$ la sortie.

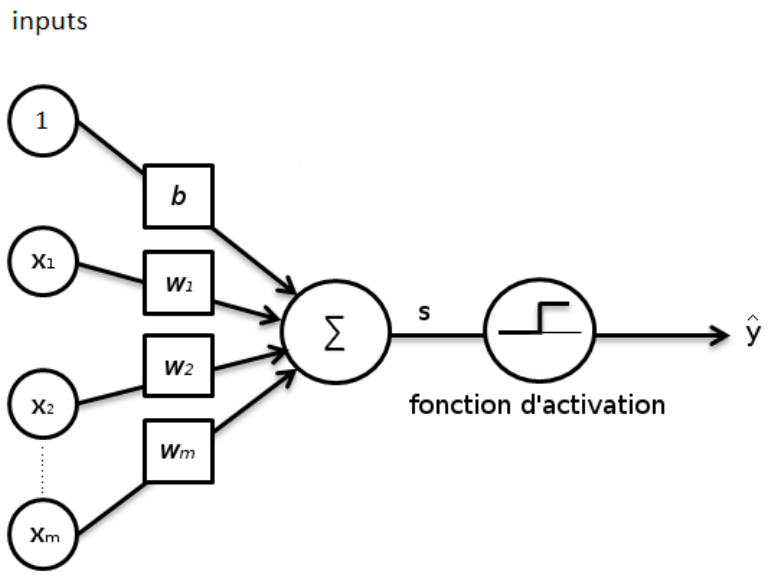


FIGURE 4 – Neurone formel

Un neurone formel est constitué de deux parties :
— une partie linéaire telle que $s = \vec{w}^T.\vec{x} + b$ avec $\vec{w}$ de coordonnées $w_1, w_2, \ldots, w_m$, le vecteur de poids synaptiques et b le biais ;

— une partie « non linéaire » où l'on applique une fonction d'activation telle que $\hat{y} = f(s)$. Historiquement cette fonction d'activation était la fonction « échelle » ($H(x) = 0$ si $x \leq 0$ et $H(x) = 1$ si $x > 0$), mais il existe aujourd'hui un grand nombre de fonctions d'activation sur lesquelles nous reviendrons plus tard.

Le neurone formel, malgré sa simplicité, est capable de résoudre des problèmes de classification binaire : $\hat{y}$ peut s'interpréter comme la probabilité d'être dans la « classe 1 » si $s = \vec{w}^T.\vec{x} + b > 0$ et dans la « classe 0 » si $s \leq 0$. Le vecteur $\vec{w}$ est perpendiculaire à l'hyperplan d'équation $\vec{w}^T.\vec{x} + b = 0$. Cet hyperplan constitue la frontière de décision (séparation entre les classes 0 et 1 évoquées ci-dessus).

2.2.2 Couche de neurones formels

Un neurone formel est uniquement capable de traiter des problèmes de classification à 2 classes. Pour traiter les problèmes multiclasses, il est nécessaire d'utiliser plusieurs neurones formels. Si nous avons k classes, il faudra utiliser k neurones formels (voir la Figure 5).

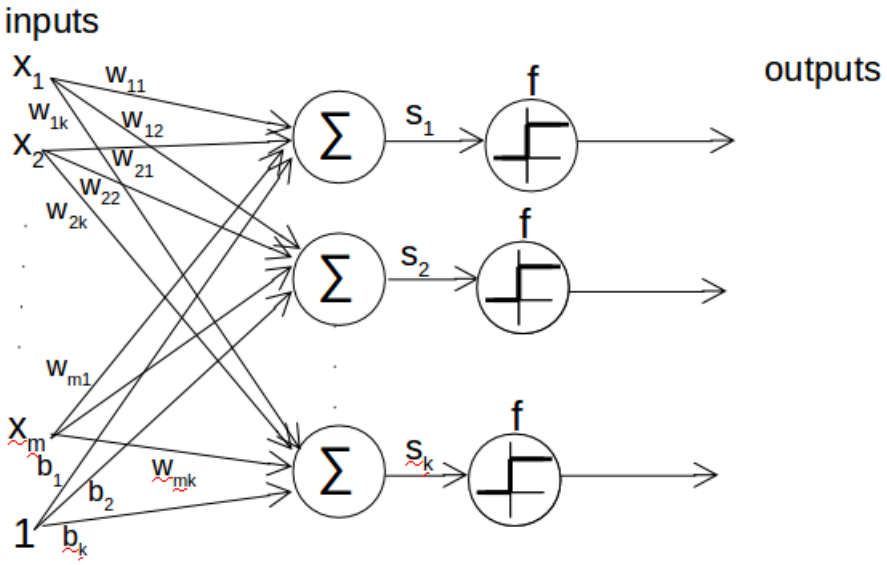


FIGURE 5 – Association de neurones formels

Avec k classes, nous aurons alors : $\vec{s} = \vec{x}^T.W + \vec{b}$, avec W la matrice de poids de taille $m \times k$ et $\vec{b}$ un vecteur biais de taille k. Dans le cas de

la classification multiclasses nous utiliserons la fonction d'activation softmax pour chaque sortie $\hat{y}_n$:

$$\hat{y}_n = \frac{\exp s_n}{\sum_{p=1}^{k} \exp s_p}.$$

2.2.3 Réseau multicouches et réseau profond

Le réseau de neurones monocouche vu précédemment est insuffisant pour traiter des problèmes avec des frontières de décision non linéaires (voir le problème XOR [6]). Pour traiter ce genre de problème, il est nécessaire d'ajouter une couche cachée, on parle alors de perceptrons multicouches (voir la Figure 6). Afin d'augmenter l'expressivité du système (capacité à « simuler »

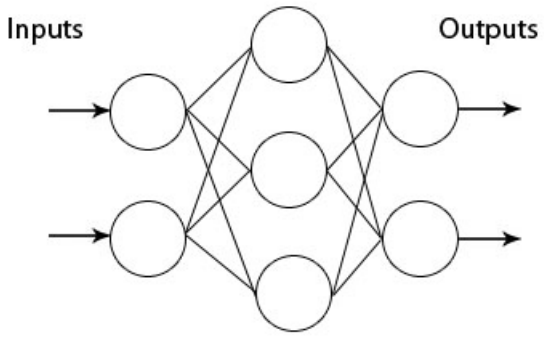


FIGURE 6 – Perceptron multicouche

des fonctions de plus en plus complexes), il est possible de rajouter un grand nombre de couches cachées, nous avons alors affaire à un réseau de neurones profond (voir la Figure 7). Chaque sortie de neurone est connectée aux entrées des neurones de la couche suivante, on parle alors de réseau de neurones « complètement connecté ».

2.2.4 Apprentissage supervisé

Les paramètres d'un réseau de neurones (la matrice de poids W et les vecteurs biais $\vec{b}$ que nous regrouperons désormais sous la notation θ), peuvent être déterminés grâce à un apprentissage supervisé. L'idée est d'utiliser un ensemble d'apprentissage $A = \{(\vec{x_1}, \vec{y_1^*}), (\vec{x_2}, \vec{y_2^*}), \ldots, (\vec{x_N}, \vec{y_N^*})\}$, afin d'entraîner le réseau en ajustant les paramètres θ du réseau f_θ. Il sera ensuite possible

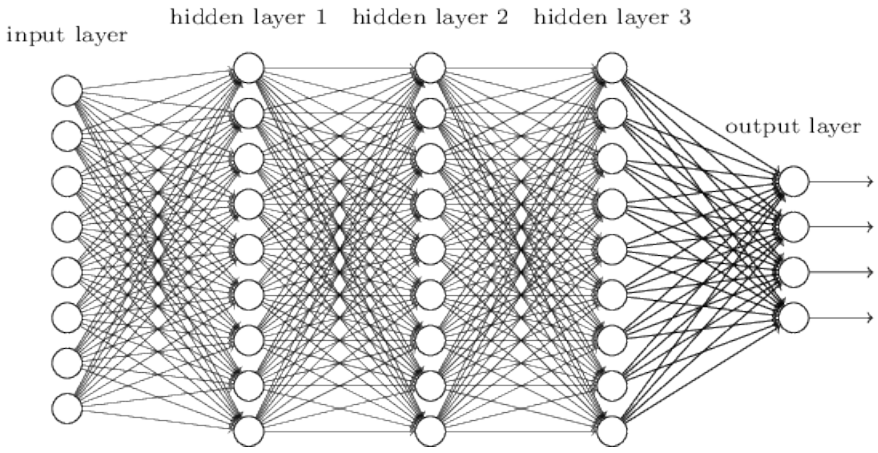


FIGURE 7 – Réseau profond

d'utiliser ce réseau f_θ afin de déterminer une valeur $\vec{\hat{y}}=f_\theta(\vec{x})$ à partir d'une entrée $\vec{x}$ n'appartenant pas à l'ensemble d'apprentissage A. Pour réaliser cet entraînement, on utilisera une fonction de coût :

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} g(f_\theta(\vec{x_i}), \vec{y_i^*})$$

avec N le nombre de couple $(\vec{x},\vec{y^*})$ contenu dans l'ensemble d'apprentissage A. Il existe de nombreuses possibilités, que nous verrons plus loin, pour le choix de la fonction de coût, mais cette fonction doit être différentiable et continue. L'idée étant de minimiser cette fonction de coût $J(\theta)$ sur l'ensemble d'apprentissage, nous allons utiliser l'algorithme de descente de gradient afin d'ajuster les paramètres θ du réseau de neurones :

$$\theta_{t+1} = \theta_t - \eta\frac{\partial J}{\partial \theta}$$

avec $\frac{\partial J}{\partial \theta}$ le gradient de la fonction de coût par rapport à θ et η le taux d'apprentissage. On applique cet algorithme de descente de gradient jusqu'à la convergence ($\|\nabla_\theta J\|^2 = 0$). Le choix du taux d'apprentissage a son importance, en effet si η est trop petit, nous aurons une convergence trop lente, alors qu'un η trop grand amènera à une oscillation qui rendra aussi la convergence difficile.

L'algorithme de descente de gradient utilise la rétropropagation du gradient de la fonction de coût. Les calculs se font en deux étapes : une première

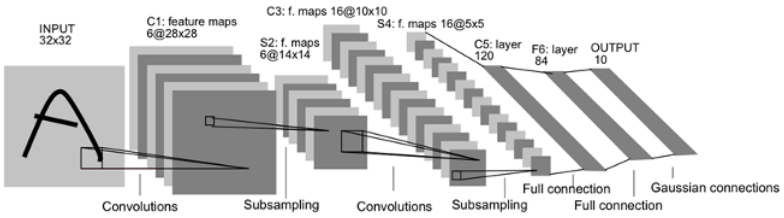
étape dite « forward », où on parcourt le réseau de neurones de l'entrée vers la sortie afin de calculer la fonction de coût $J(\theta)$. Nous avons ensuite une étape dite « backward », où on parcourt le réseau de la sortie vers l'entrée afin de calculer les dérivées nécessaires au calcul du gradient de la fonction de coût $\nabla_\theta J$.

Pour obtenir le gradient, on devrait normalement effectuer les calculs sur l'ensemble des exemples d'apprentissage, mais avec un grand nombre de données, ces calculs sont très longs à réaliser. Au lieu d'effectuer ces calculs sur l'ensemble des données, l'algorithme de descente de gradient stochastique (*SGD*) calcule une approximation du gradient sur chaque donnée (l'ordre de traitement des données est aléatoire, d'où le terme stochastique). Afin d'obtenir une meilleure approximation du gradient, il est aussi possible de choisir une solution intermédiaire : au lieu d'effectuer les calculs après chaque exemple, on peut travailler sur un sous-ensemble de l'ensemble d'entraînement (ce sous-ensemble est appelé « mini-batch »). Au cours d'une session d'entraînement (appelée « epoch »), tous les exemples auront été utilisés. À noter que par abus de langage le terme *SGD* est souvent utilisé pour ce traitement par mini-batch.

2.2.5 Les réseaux convolutifs

Si la dimension des données en entrée d'un réseau de neurones complètement connecté est importante, par exemple dans le cas d'une image (une image de 28x28 pixels correspondra à un vecteur d'entrée de dimensions 784), le nombre de paramètres θ du réseau sera tellement important que les calculs liés à la phase d'apprentissage risquent de prendre un temps beaucoup trop long.

Pour répondre à la problématique de la reconnaissance d'image, Yann Lecun a mis au point dans les années 90, en s'inspirant du fonctionnement du cortex visuel des animaux, les réseaux de neurones convolutifs (voir la Figure 8). Ce type de réseau utilise des masques de convolution (voir la Figure 9) afin de détecter des caractéristiques dans une image. Une couche de convolution contient un nombre variable de masques de convolution, afin de détecter plusieurs caractéristiques au sein de la même couche. Après l'application des différents masques de convolution, on applique une non-linéarité. Classiquement la non-linéarité appliquée est de type ReLU (*Rectified Linear Unit* voir la Figure 10) : $f(x) = \max(0, x)$. Les valeurs des différents masques de convolution ne sont pas prédéfinies, mais apprises grâce à un ensemble



An early (Le-Net5) Convolutional Neural Network design, LeNet-5, used for recognition of digits

FIGURE 8 – Réseau « LeNet 5 » développé par Y Lecun

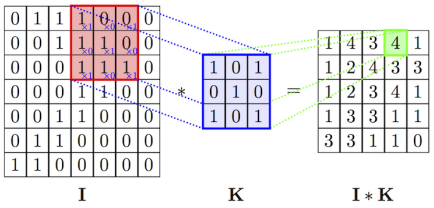


FIGURE 9 – Exemple de masque de convolution

d'entraînement. Les réseaux convolutionnels sont des réseaux profonds : on enchaîne un grand nombre de couches, ce grand nombre de couches permet d'augmenter l'expressivité du réseau.

Un autre type de couche est souvent utilisé dans les réseaux convolutifs : la couche de Pooling. Le Pooling permet de réduire la taille des images intermédiaires et améliore ainsi les performances du réseau (gain en puissance de calcul). Sachant que la couche de Pooling ne sera pas utilisée dans ce travail, son fonctionnement ne sera pas détaillé (pour en savoir plus voir [6])

2.2.6 L'architecture résiduelle dans les réseaux convolutifs

Naïvement on pourrait penser qu'en utilisant des réseaux de plus en plus profonds, on améliorerait la performance des réseaux. L'expérience montre le contraire : à partir d'une certaine profondeur, les performances des réseaux plafonnent, voir même régressent. Pour pallier à ce problème, une nouvelle architecture dite « architecture résiduelle » a vu le jour fin 2015 [7] (voir la Figure 11). Il peut être nécessaire d'obtenir la fonction identité en sortie d'une couche de neurones. Les couches de neurones ont du mal à simuler cette fonction identité. En ayant en sortie d'une couche $F(x) + x$ à la place de $F(x)$, la fonction identité devient plus facile à obtenir. En jouant sur les

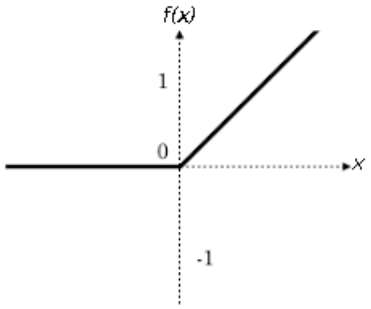


FIGURE 10 – Fonction ReLU

poids synaptiques, il est en effet plus aisé d'obtenir $F(x) = 0$ que $F(x) = x$, or, en ayant $F(x) = 0$ on obtient bien la fonction identité si l'on a en sortie de la couche $F(x)+x$. Avec ce genre d'architecture, on constate une amélioration

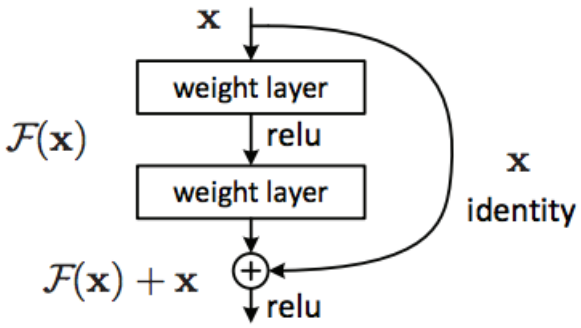


FIGURE 11 – Principe de l'architecture résiduelle

des performances avec des réseaux très profonds.

2.2.7 Méthodes d'optimisation

Au cours de la rétropropagation, le gradient peut, en « remontant » les couches, devenir de plus en plus petit et même, dans certains cas, tendre vers 0, ce qui pose un problème important au niveau de « l'apprentissage ». Ce phénomène est connu sous le nom de « *vanishing gradient* ». On peut aussi rencontrer le phénomène inverse : le gradient devient de plus en plus grand, on parle alors de « *exploding gradient* ». Dans le cas de l'exploding gradient, on obtient des poids très grands ce qui fait diverger l'algorithme de

rétropropagation du gradient. Le vanishing/exploding gradient entraîne un ralentissement significatif de la phase d'entraînement du réseau de neurones. Aussi, il est nécessaire de trouver des méthodes permettant d'éviter ces deux phénomènes.

En 2010, Xavier Glorot et Yoshua Benigio ont montré [8] que ces problèmes de gradient avaient plusieurs origines, dont les fonctions d'activation utilisées jusqu'alors (type sigmoïd, voir Figure 12) et l'initialisation des poids du réseau. Les fonctions de type sigmoïd entraînent un phénomène de saturation (à 0 ou à 1). Afin d'éviter ce phénomène de saturation, Glorot et Bengio

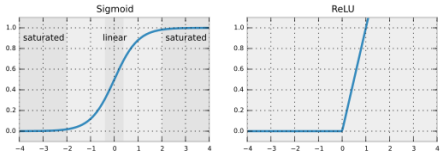


FIGURE 12 – Fonction sigmoïd et fonction ReLU

ont proposé d'autres fonctions d'activation, par exemple la fonction ReLU.

Pour ce qui est de l'initialisation des poids du réseau, Glorot et Bengio ont mis au point une méthode (qui ne sera pas expliquée ici) souvent appelée « initialisation Xavier ».

Sergey Ioffe et Christian Szegedy ont proposé, en 2015, une autre méthode pour éviter le phénomène de vanishing gradient : la normalisation par batch [9]. La normalisation par batch appartient, comme son nom l'indique, à la famille des méthodes de normalisation des données : au lieu d'utiliser les données brutes x en entrée du réseau, on utilise x' tel que $x' = \frac{x-\mu}{\sigma}$ avec μ la moyenne de toutes les entrées et σ l'écart type aussi calculé sur toutes les entrées. Dans les méthodes de normalisation classique, on normalise uniquement les données à l'entrée du réseau de neurones, dans le cas de la normalisation par batch, la normalisation est appliquée à la sortie de chaque couche cachée, juste avant la fonction d'activation (voir Figure 13). Au lieu de calculer la moyenne et l'écart type sur l'ensemble des données, ces deux grandeurs sont calculées sur un batch de données. Ioffe et Szegedy ont montré que cette technique améliore grandement l'apprentissage dans le cas des réseaux profonds en réduisant drastiquement le problème du vanishing gradient et améliore donc les performances des réseaux profonds.

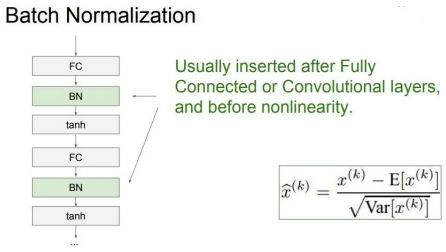


FIGURE 13 – Normalisation par batch

2.3 Recherche arborescente Monte Carlo (MCTS)

2.3.1 Algorithme Minimax

L'algorithme Minimax [10] est un algorithme très utilisé dans les jeux dits à « somme nulle » et à « information parfaite » (exemples : le morpion, les Echecs, les dames, le Go...). Le but de l'algorithme Minimax est de déterminer le meilleur coup à jouer à partir d'un état donné du jeu. L'algorithme construit un arbre de jeu qui représente tous les états possibles de jeu à partir de l'état actuel.

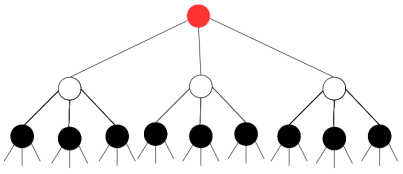


FIGURE 14 – Minimax

L'algorithme « simule » 2 joueurs : l'un est appelé max et l'autre min. L'état courant est représenté en rouge sur la Figure 14, c'est au tour de max de jouer. Toujours sur la Figure 14, les cercles blancs représentent les différents états possibles après un coup joué par max, les cercles noirs représentent les différents états possibles après un coup joué par min. L'arbre est développé de façon récursive pour chaque branche jusqu'au moment où max ou min remporte la partie. Un score est attribué à chaque état final (feuille de l'arbre). Par exemple, si nous nous trouvons dans un état final où max a remporté la partie, on attribuera à cet état le score de +1, si nous nous trouvons dans un état final où min a remporté la partie, on attribuera un score de -1 et enfin, si nous nous trouvons dans un état final où il y a eu

match nul, on attribuera un score de 0.

À chaque fois que cela sera au tour de max de jouer, on part du principe qu'il choisira, parmi tous les états possibles, celui qui lui apportera le score le plus grand possible. On aura exactement l'inverse quand cela sera au tour de min de jouer, il choisira le coup qui lui permettra de se retrouver dans l'état où le score sera le plus petit possible. L'algorithme part des feuilles et remonte l'arbre de façon récursive en partant du principe que max et min joueront à chaque fois au mieux de leur intérêt respectif. Une fois la racine atteinte, max pourra choisir le « meilleur » coup à effectuer compte tenu de l'état courant.

L'algorithme alpha-béta est une amélioration de l'algorithme minimax puisqu'il permet d'explorer seulement une partie de l'arbre. Nous ne l'aborderons pas dans ce document.

Selon la complexité du jeu, l'arbre de jeu peut atteindre des profondeurs très importantes et même trop importantes pour permettre un traitement par l'ordinateur dans un temps raisonnable. Il est donc souvent nécessaire de limiter la profondeur de l'arbre. Problème : si nous limitons la profondeur de l'arbre, selon l'état d'avancement de la partie, nous avons de fortes chances de ne pas atteindre un état terminal. Dans ce cas, comment attribuer un score (dans l'exemple évoqué ci-dessus +1, -1 ou 0) à la feuille de l'arbre ? Nous devons définir une fonction dite fonction d'évaluation qui prend en paramètre un état du jeu et renvoi un score. Ces fonctions d'évaluations peuvent être, selon les jeux, très complexes à mettre au point. Une mauvaise fonction d'évaluation rend l'algorithme Minimax complètement inefficient.

2.3.2 Principe du Monte Carlo Tree Search

C'est pour répondre aux problèmes évoqués ci-dessus (trop grand nombre d'états à gérer et fonction d'évaluation difficile à mettre au point) que l'algorithme du Monte Carlo Tree Search (MCTS) a été mis au point. L'algorithme MCTS [11] permet d'explorer un arbre de recherche en effectuant des simulations. Dans le cas d'un jeu à somme nulle, l'algorithme MCTS permet de déterminer le coup suivant sans être obligé de considérer tous les coups possibles jusqu'à un état terminal. L'algorithme MCTS se décompose en 4 phases :

1. la sélection ;
2. l'expansion ;

3. la simulation ;
4. la rétropropagation.

Une fois la rétropropagation réalisée on reprend un nouveau cycle en effectuant une nouvelle sélection (voir la Figure 15).

On commence par calculer la valeur *ucb* (*upper confidence bounds*) pour chaque état s_i de l'arbre (chaque nœud de l'arbre) :

$$ucb = \overline{V_i} + C\sqrt{\frac{\ln N}{n_i}}$$

avec :

— $\overline{V_i}$ la valeur moyenne (score moyen) de l'état s_i
— N le nombre total de visites
— n_i le nombre de visites pour l'état s_i
— C une constante permettant de favoriser l'exploration ou l'exploitation : plus la constante C est grande et plus on favorisera l'exploration au détriment de l'exploitation.

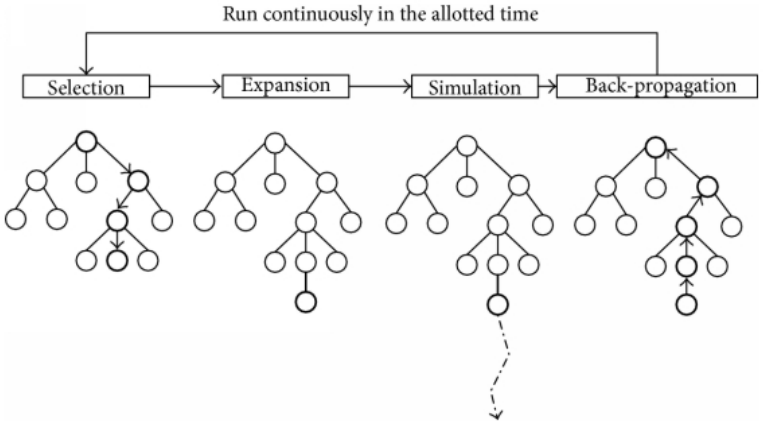


FIGURE 15 – Monte-Carlo Tree Search

On choisira l'état qui obtiendra la plus grande valeur *ucb*. Si cet état n'a pas encore été visité, on effectuera une simulation permettant d'atteindre un état terminal. Un score est attribué à l'état terminal, ce score « remonte » le long de l'arbre (rétropropagation) jusqu'à la racine. Dans le cas où le nœud ayant obtenu la plus grande valeur *ucb* a déjà été visité, on aura alors une phase d'expansion à partir de ce nœud (apparition de nouveaux nœuds).

L'algorithme MCTS peut itérer un certain nombre de fois ou pendant un temps limité. Une fois l'exécution de l'algorithme terminée, il suffira, à partir de l'état courant s_i, de choisir l'état suivant s_{i+1} qui aura obtenu le plus grand score.

2.4 AlphaZero pour Othello

En s'inspirant grandement des travaux de David Silver et de son équipe sur AlphaZero, Shantanu Thakoor, Surag Nair et Megha Jhunjhunwala, tous les trois étudiants à l'université de Stanford, ont mis au point une intelligence artificielle conçue pour jouer à Othello [12]. Le code source de leur IA est disponible sous licence libre [13].

Nair *et al.* utilisent un réseau de neurones f_θ relativement modeste puisqu'il est composé de quatre couches de convolution suivies de deux couches complètements connectées. Ce réseau prend en entrée b_t qui correspond à la représentation du plateau à l'état s_t : une matrice de 6 par 6 avec un « 0 » si la case est vide, un « 1 » si la case est occupée par une pièce blanche et un « -1 » si la case est occupée par une pièce noire (voir la Figure 16). Ce réseau

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

FIGURE 16 – b_t : matrice représentant le plateau en début de partie

va permettre de modéliser la fonction valeur $V_\theta(s_t)$ (telle qu'elle a été définie dans la section 2.1.5) et un vecteur politique que l'on notera $\vec{p_\theta}(s_t)$. $\vec{p_\theta}$ a pour coordonnées $(p_1, p_2, p_3, \ldots, p_n)$ avec n le nombre total d'actions possibles et p_1 la probabilité de choisir l'action a_1, p_2 la probabilité de choisir l'action $a_2, \ldots, p_n$ la probabilité de choisir l'action a_n. Le réseau de neurones est initialisé avec des valeurs de paramètres aléatoires. Les séances de « self-play » (voir plus loin pour plus de détails) permettent d'obtenir deux grandeurs, $\vec{\pi_t}$ et z_t pour un état donné s_t :

— $\vec{\pi_t}$ est de même nature que $\vec{p_\theta}(s_t)$. C'est un vecteur qui donne la probabilité de choisir une action a parmi toutes les actions possibles. $\vec{\pi_t}$ est issu du « self-play » alors que $\vec{p_\theta}(s_t)$ est lui issu directement du réseau de neurones ;

— z_t correspond à la valeur de l'état s_t du point de vue du joueur courant. z_t est donc de même nature que V_θ défini précédemment. Ici aussi z_t est le résultat du « self-play » alors que V_θ est issu du réseau de neurones.

À l'issu du « self-play » on obtient donc une série de triplets $(b_t, \vec{\pi_t}, z_t)$.

L'apprentissage au niveau du réseau de neurones se fait grâce à une fonction de coût $J(\theta)$ telle que :

$$J(\theta) = \sum_t (V_\theta(s_t) - z_t)^2 + \vec{\pi_t} \log \vec{p_\theta}(s_t)$$

On utilise donc la combinaison d'une fonction quadratique et d'une cross entropy. L'entraînement a ainsi pour objectif de minimiser l'écart entre $V_\theta(s_t)$ et z_t d'un côté et $\vec{\pi_t}$ et $\vec{p_\theta}(s_t)$ d'un autre côté.

Intéressons-nous maintenant aux méthodes employées pour obtenir $\vec{\pi_t}$ et z_t, c'est à dire à la phase de « self-play ». L'idée du « self-play » est d'améliorer la politique permettant à l'IA de jouer le meilleur coup possible pour un état donné s_t. Cette phase de « self-play » utilise principalement un arbre de recherche de Monte-Carlo (MCTS) qui repose sur l'Algorithme 1 décrit ci-après (par souci de clarté le nom donné à cet algorithme correspond au nom donné à la méthode utilisée dans l'implémentation en Python).

Afin de mieux comprendre le fonctionnement de cet algorithme fondamental, nous allons simuler son fonctionnement avec un exemple (voir aussi la Figure 17) :

étape 1

Première exécution de la fonction *search*. L'état s_0 n'a pas encore été rencontré : $N_s(s_0) = 0$ et $v = V_\theta(s_0)$. La fonction renvoie $-v$. On sort de la fonction *search*.

étape 2

On exécute à nouveau la fonction *search*. On repart de l'état s_0. L'état s_0 a déjà été rencontré . Pour chaque action a possible depuis s_0, on calcule

$$u(a) = cpuct.P(s_0, a).\sqrt{N_s(s_0) + \epsilon}$$

avec ϵ une valeur proche de zéro, mais non nulle afin d'éviter d'avoir $u(a) = 0$, avec $P(s_0, a)$ la probabilité de choisir l'action a depuis l'état s_0 ($P(s_0, a)$ a été fournie par le réseau de neurones). Partons du principe que $u(a_0)$ soit la plus grande valeur obtenue, on choisit donc d'effectuer l'action a_0. Depuis l'état s_0, l'action a_0 nous amène à l'état $s_{0,0}$ à partir duquel la fonction *search* est appelée (appel récursif).

Algorithm 1 search(s)

if s est terminal **then**
  return $-resultatPartie$
end if
if $s \notin arbre$ **then**
  $arbre \leftarrow arbre \cup s$
  $N_s(s) \leftarrow 0$
  $P(s, .) \leftarrow p_\theta(s)$
  $v \leftarrow V_\theta(s)$
  return $-v$
end if
$curBest \leftarrow -\infty$
$bestAct \leftarrow -1$
for all action a possible depuis s **do**
  if $Q_{sa}(s,a)existe$ **then**
    $u \leftarrow Q_{sa}(s,a) + cpuct.P(s,a).\frac{\sqrt{N_s(s)}}{1+N_{sa}(s,a)}$
  else
    $u \leftarrow cpuct.P(s,a).\sqrt{N_s(s)+\epsilon}$
  end if
  if $u > curBest$ **then**
    $curBest \leftarrow u$
    $bestAct \leftarrow a$
  end if
end for
$s' \leftarrow nextState(s,a)$
$v \leftarrow search(s')$
if $Q_{sa}(s,a)existe$ **then**
  $Q_{sa}(s,a) \leftarrow \frac{N_{sa}(s,a).Q_{sa}(s,a)+v}{N_{sa}+1}$
  $N_{sa}(s,a) \leftarrow N_{sa}(s,a) + 1$
else
  $Q_{sa}(s,a) \leftarrow v$
  $N_{sa}(s,a) \leftarrow 1$
end if
$N_s(s) \leftarrow N_s(s) + 1$
return $-v$

étape 3
$s_{0,0}$ n'a pas encore été rencontré, nous avons donc $N_s(s_{0,0}) = 0$. Soit $v = V_\theta(s_{0,0})$, la fonction *search* renvoie $-v$. Pour notre exemple, partons du principe que $v = 0.2$, la fonction renvoie donc -0.2.
étape 4
On dépile l'appel récursif précédent, on retourne à l'état s_0. On a donc $Q_{sa}(s_0, a_0)$ la valeur retournée par $s_{0,0} = -0.2$, $N_{sa}(s_0, a_0) = 1$ et $N_s(s_0) = 1$. La fonction renvoie $-v$, c'est-à-dire 0.2, on quitte la fonction *search*.
étape 5
On exécute la fonction *search* pour la 3e fois. On repart de l'état s_0. On calcule $u(a)$ pour chaque action possible depuis s_0 :

— pour une action a_1 non encore explorée, nous aurons :

$$u(a_1) = cpuct.P(s_0, a_1).\sqrt{N_s(s_0) + \epsilon}$$

— pour une action a_0 déjà explorée nous aurons :

$$u(a_0) = Q_{sa}(s_0, a_0) + cpuct.P(s_0, a_0).\frac{\sqrt{N_s(s_0)}}{1 + N_{sa}(s_0, a_0)}$$

Partons du principe que $u(a_1)$ est la plus grande valeur trouvée.
étape 6
Depuis s_0 on effectue l'action a_1, on se retrouve dans l'état $s_{0,1}$. L'état $s_{0,1}$ n'a pas encore été rencontré : $N_s(s_{0,1}) = 0$ et $v = V_\theta(s_{0,1})$, la fonction renvoie $-v$. Pour notre exemple, partons du principe que $v = 0.1$, la fonction renvoie donc -0.1.
étape 7
On dépile l'appel récursif, on se retrouve dans l'état s_0. On a $Q_{sa}(s_0, a_1)$ la valeur retournée par $s_{0,1} = -0.1$, $N_{sa}(s_0, a_1) = 1$ et $N_s(s_0) = 2$. La fonction retourne $-v$, c'est-à-dire 0.1, on quitte la fonction *search*.
étape 8
On exécute la fonction *search* pour la 4e fois. On repart de l'état s_0. On calcule $u(a)$ comme expliqué dans l'étape 5. Partons du principe que la valeur de $u(a)$ la plus élevée est trouvée pour l'action a_1.
étape 9
Depuis s_0 on effectue l'action a_1 qui nous amène à l'état $s_{0,1}$. L'état $s_{0,1}$ a déjà été exploré. On calcule $u(a)$ pour chaque action possible depuis $s_{0,1}$. Partons du principe que $u(a_4)$ correspond à la valeur la plus grande possible,

on exécute donc a_4 depuis $s_{0,1}$ ce qui nous emmène vers l'état $s_{0,1,4}$.
étape 10
$s_{0,1,4}$ n'a pas encore été rencontré. $N_s(s_{0,1,4}) = 0$ et $v = V_\theta(s_{0,1,4})$, la fonction renvoie $-v$. Pour notre exemple, partons du principe que $v = 0.3$, la fonction renvoie donc -0.3.
étape 11
On dépile l'appel récursif, on retourne à l'état $s_{0,1}$. On a $Q_{sa}(s_{0,1}, a_4)$ la valeur retournée par $s_{0,1,4} = -0.3$, $N_{sa}(s_{0,1}, a_4) = 1$ et $N_s(s_{0,1}) = 1$. La fonction retourne $-v$, c'est à dire 0.3
étape 12
On dépile l'appel récursif, on retourne à l'état s_0. $Q_{sa}(s_0, a_1)$ existe déjà, on effectue une mise à jour : $Q_{sa}(s_0, a_1) = \frac{N_{sa}(s_0,a_1).Q_{sa}(s_0,a_1)+v}{N_{sa}(s_0,a_1)+1}$ avec ici $v = 0.3$, $N_{sa}(s_0, a_1) = 2$ et $N_s(s_{0,1}) = 3$. On quitte la fonction *search*
étape 13
On exécute la fonction *search* pour la 5e fois, on repart de l'état s_0...

Ce processus va se poursuivre comme expliqué ci-dessus. Le nombre de fois où l'on repart de l'état s_0 est donné par le paramètre $numMCTSSims$. Ce paramètre va jouer un rôle majeur dans la suite de ce travail. En effet, il va déterminer le nombre d'états explorés au cours d'un processus de MCTS (à chaque fois que l'on rencontre un état non encore exploré, après avoir « dépilés » les appels récursifs, on repart de l'état s_0) et donc, par voie de conséquence, la profondeur de l'arbre de recherche. On remarque aussi un autre paramètre, *cpuct*, qui permet de favoriser plutôt l'exploration ou plutôt l'exploitation. En effet nous avons

$$u = Q_{sa}(s, a) + cpuct.P(s, a).\frac{\sqrt{N_s(s)}}{1 + N_{sa}(s, a)}$$

plus *cpuct* sera grand et plus l'exploration sera favorisée par rapport à l'exploitation. *cpuct* est donc, comme $numMCTSSims$, un paramètre qu'il faudra définir dans la suite de ce travail. À noter qu'au cours du processus de MCTS, il est possible de rencontrer un état terminal (c'est-à-dire un état où un des deux joueurs aura remporté la partie). La fonction *search* retourne alors 1 ou -1 en fonction du joueur qui aura remporté la partie. Ce sujet sera développé au moment où nous aborderons l'adaptation de la fonction *search* au cas du Yokai No Mori. Il est aussi important de constater que si une position est favorable à un joueur (par exemple avec un $v = 0.8$), elle

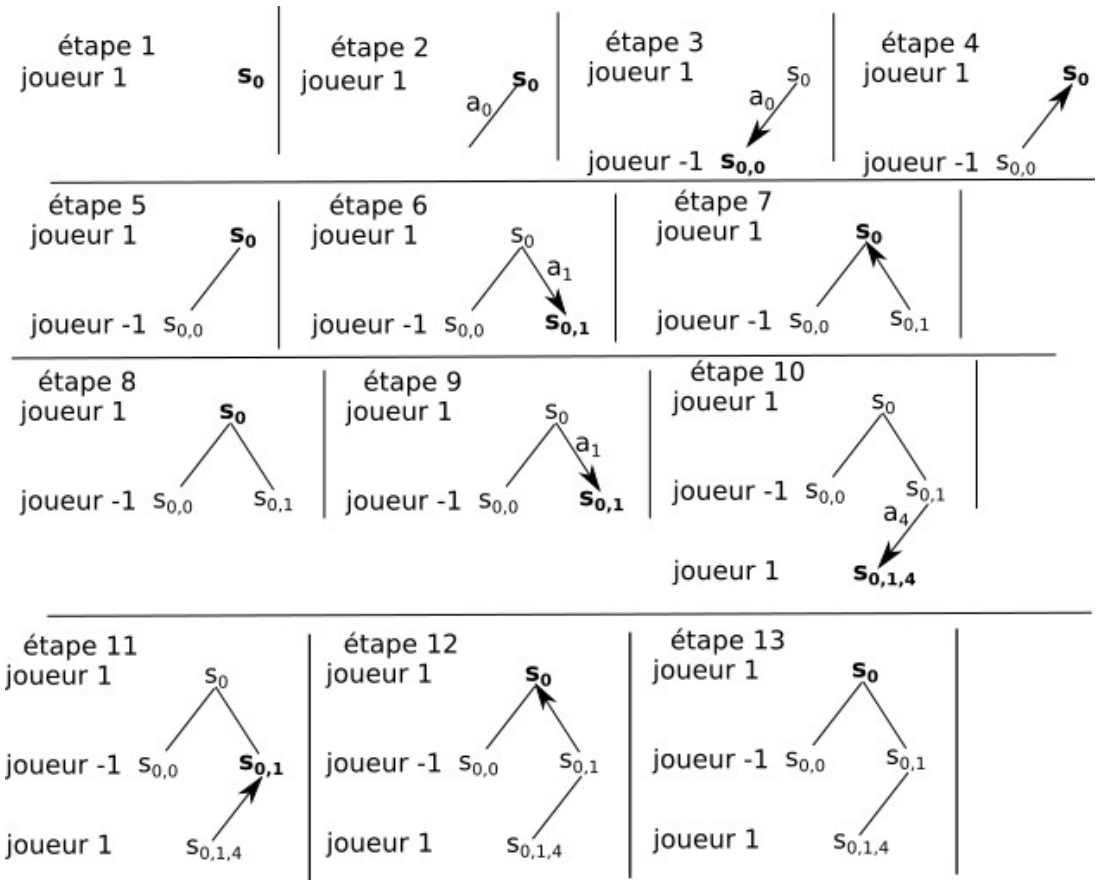


FIGURE 17 – Simulation de l'exécution de l'algorithme *search*

est forcément défavorable pour son adversaire (du point de vue de l'adversaire, la position vaut $v = -0.8$). Voilà pourquoi, la fonction *search* retourne systématiquement l'opposé de v.

Une fois le processus de MCTS terminé pour l'état s_0, nous allons, grâce aux données récoltées pendant ce MCTS, obtenir le vecteur politique $\vec{\pi_t}$ qui va contenir, pour toutes les actions possibles à partir de s_0, la probabilité d'exécuter chaque action. Par exemple, si l'on obtient $\vec{\pi_t} = (0.1, 0.3)$, la probabilité de choisir l'action a_0 est de 0.1 et la probabilité de choisir l'action a_1 est de 0.3. Si une action est impossible à partir d'un état s_t donné, la probabilité de cette action sera nulle. Pour obtenir ce vecteur $\vec{\pi_t}$, on part du principe que pour une action a donnée, la probabilité d'effectuer cette action est proportionnelle à $N_{sa}(s_0, a)^{\frac{1}{\tau}}$ avec τ le paramètre « température ».

Le paramètre « température » va, comme *cpuct*, permettre d'ajuster le taux d'exploration. Une « température » élevée aura tendance à uniformiser la distribution des probabilités, on favorisera donc l'exploration. À l'inverse, dans le cas où la valeur du paramètre τ est choisie nulle, la probabilité de toutes les actions possibles depuis s_0 seront mises à zéro à l'exception de l'action qui aura le $N_{sa}(s_0, a)$ le plus grand qui verra sa probabilité égale à 1, on sera alors typiquement dans un cas où l'exploitation sera grandement favorisée. Nous pouvons donc bien considérer que $\vec{\pi_t}$ est une amélioration de $\vec{p_\theta}(s_t)$, amélioration due aux simulations effectuées durant le MCTS.

Une fois ce vecteur $\vec{\pi_t}$ construit pour l'état s_0, l'agent va pouvoir choisir l'action à effectuer. Cette action est choisie aléatoirement en utilisant la distribution de probabilités proposée par $\vec{\pi_t}$. Si la probabilité de choisir l'action a est nulle, alors cette action n'aura aucune chance d'être choisie. Le choix de cette action a permet de passer de l'état s_0 à l'état s_1. Une fois dans l'état s_1, on procède de nouveau à un MCTS, exactement comme cela a été fait pour l'état s_0. Depuis s_1, l'agent effectue de nouveau un choix aléatoire en utilisant le vecteur $\vec{\pi_t}$ construit grâce au second MCTS, ceci va nous permettre de nous retrouver dans l'état s_2...et ainsi de suite jusqu'au moment où nous allons rencontrer un état terminal s_f (voir Figure 18). Si le joueur

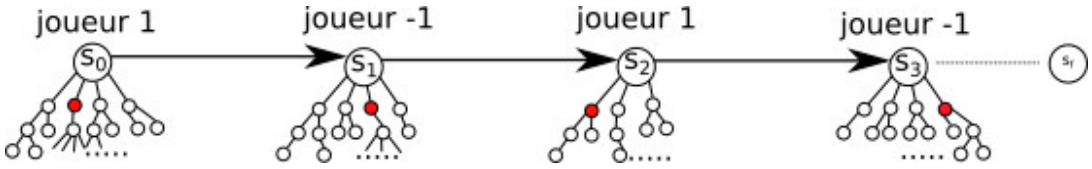


FIGURE 18 – déroulement d'un épisode

1 est vainqueur : tous les états où c'est au tour du joueur 1 de jouer (s_0, s_2, ... dans notre exemple) se verront attribuer une « récompense » de +1 (nous aurons $z_t = +1$ pour ces états). A contrario, tous les états où c'est au tour du joueur -1 (s_1, s_3,...) se verront attribuer une récompense de -1 (nous aurons $z_t = -1$ pour ces états). De la même manière, si c'est le joueur -1 qui sort vainqueur de la partie, z_t sera égal à -1 pour tous les états « c'est au tour du joueur 1 » et égal à +1 pour tous les états « c'est au tour du joueur -1 ».

Tout ce processus (une partie de « self-play » joueur 1 contre joueur -1) constitue un épisode. L'algorithme va enchaîner plusieurs épisodes (le nombre d'épisodes à effectuer est donné par le paramètre *numEps*). Chaque épisode repose sur le même principe : on part de l'état initial s_0, pour chaque état

on effectue une série de simulations (MCTS) qui utilise les données issues du réseau de neurones ($\vec{p_\theta}(s_t)$ et $V_\theta(s_t)$). À la fin de cette phase de simulation pour un état donné on obtient un vecteur $\vec{\pi_t}$ qui va permettre de déterminer l'action à effectuer. Une fois un état terminal atteint, une mise à jour de chaque état rencontré au cours de l'épisode est effectuée, en attribuant à chaque état une récompense z_t égale à -1 ou +1 en fonction du vainqueur de l'épisode. À la fin d'un épisode, on obtient donc un triplet (b_t, $\vec{\pi_t}$, z_t) pour chaque état (avec b_t la représentation du plateau de jeu pour l'état s_t, $\vec{\pi_t}$ le vecteur politique et z_t la récompense). Après un certain nombre d'épisodes, nous avons à notre disposition un grand nombre de triplets (b_t, $\vec{\pi_t}$, z_t). Ces triplets seront utilisés pour entraîner le réseau de neurones : on utilise b_t en entrée du réseau de neurones, $\vec{\pi_t}$ et z_t en sortie (fonction de coût $J(\theta) = \sum_t (V_\theta(s_t) - z_t)^2 + \vec{\pi_t} \log \vec{p_\theta}(s_t)$). Le nombre d'epochs utilisé pour entraîner le réseau est donné par le paramètre *epochs*.

Une fois l'entraînement du réseau de neurones terminé, afin de tester l'efficacité des modifications apportées à ce réseau de neurones, une nouvelle séance de « self-play » va avoir lieu en utilisant deux versions du réseau de neurones : θ_{New} qui correspond au réseau de neurones modifié après la phase d'entraînement et θ_{Old} qui correspond à ce même réseau de neurones, mais dans une version antérieure à la phase d'entraînement. Au cours de ce « self-play », le joueur 1 va utiliser le réseau de neurones θ_{New} alors que le joueur -1 utilisera le réseau θ_{Old}. L'algorithme va réaliser un certain nombre d'épisodes (ce nombre d'épisodes est donné par le paramètre $arenaCompare$). Les épisodes se dérouleront comme expliqué précédemment : pour chaque état on effectue des simulations (MCTS) qui permettent de générer un vecteur $\vec{\pi_t}$... À la fin de cette phase que l'on nommera dans la suite « θ_{New} vs θ_{Old} », on comptabilise les victoires remportées par le joueur -1 (Vi_{-1}) et les victoires remportées par le joueur +1 (Vi_{+1}). Si le joueur 1 a remporté le plus de victoires, et si on a :

$$\frac{Vi_{+1}}{Vi_{+1} + Vi_{-1}} \geqslant updateThreshold$$

avec $updateThreshold$ un paramètre compris entre 0 et 1, le réseau de neurones θ_{New} devient le réseau de neurones de référence et θ_{Old} est abandonné. Dans le cas contraire θ_{Old} reste le réseau de référence et θ_{New} est abandonné.

En résumé, on appelle itération l'ensemble des processus suivants :

— $numEps$ épisodes de « self-play » qui vont permettre de générer un grand nombre de triplets (b_t, $\vec{\pi_t}$, z_t) ;

— la mise à jour des paramètres du réseau de neurones grâce aux triplets ;
— « θ_{New} vs θ_{Old} ».

Dès qu'une itération est terminée, une nouvelle recommence. Le nombre total d'itérations effectué est donné par le paramètre $numIters$.

Une fois les itérations terminées (fin de la phase d'entraînement), il est possible de jouer contre l'IA. L'IA utilisera quasiment les mêmes algorithmes que pendant les itérations (voir Figure 19). Pour un état donné, l'action à effectuer par l'IA sera choisie aléatoirement en utilisant la distribution de probabilité donnée par $\vec{\pi_t}$ ($\vec{\pi_t}$ est ici aussi issu d'une phase de simulation).

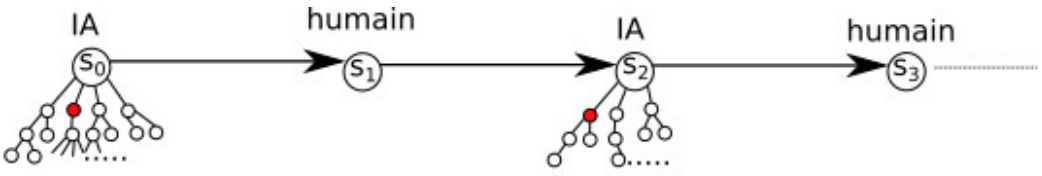


FIGURE 19 – Humain vs IA

3 Création d'une IA pour jouer au jeu Yokai No Mori

3.1 Présentation du Yokai No Mori

3.1.1 Introduction

Yokai No Mori (voir Figure 20) est un jeu de plateau créé en 2008 par Madoka Kitao et illustré par Naiade. Yokai No Mori se veut une introduction au Shogi (jeu traditionnel japonais, souvent présenté comme « le jeu d'Echecs japonais », voir la Figure 21). Il existe deux versions de Yokai No Mori, une avec un plateau de 3x4 et une autre avec un plateau de 5x6.

3.1.2 Version 3 x 4

Chaque joueur dispose en début de partie de quatre pièces :
— un Koropokkuru ;

FIGURE 20 – Boite du jeu Yokai No Mori

FIGURE 21 – Jeu de Shogi

— un Kitsune ;
— un Tanuki ;
— un Kodama.

Ces pièces sont disposées sur le plateau comme indiqué sur la Figure 22. Le but du jeu est :

— soit de capturer le Koropokkuru de l'adversaire ;
— soit d'amener son propre Koropokkuru sur la dernière rangée du plateau.

Les pièces se déplacent d'une seule case à la fois. Les mouvements possibles sont indiqués par les points situés sur les pièces. Par exemple un Kodama peut uniquement se déplacer vers l'avant (voir la Figure 23). En cas de mouvement d'une pièce, il y a trois possibilités :

1. la case de destination est vide, la pièce prend place sur la case de destination ;

FIGURE 22 – Disposition des pièces en début de partie version 3x4

FIGURE 23 – Kodama

2. la case de destination est occupée par une pièce « amie », le mouvement n'est pas autorisé ;
3. la case de destination est occupée par une pièce appartenant à l'adversaire, la pièce de l'adversaire est alors capturée et rejoint la « réserve » du joueur ayant effectué le mouvement. La pièce déplacée prend la place de la pièce capturée.

La notion de réserve est très importante dans ce jeu. En effet, à la place de déplacer une pièce déjà présente sur le plateau, un joueur peut « parachuter » une des pièces capturées à l'adversaire. Le parachutage consiste à déposer une des pièces présentes dans la réserve sur n'importe quelle case vide du plateau.

Autre particularité : la promotion. Si un joueur amène son Kodama sur la dernière ligne du plateau, ce Kodama se transforme alors en Kodama samuraï (le Kodama samuraï possède une liberté de mouvement supérieure au Kodama (voir Figure 24)). Cependant, si un joueur parachute son Kodama directement sur la dernière ligne, ce dernier ne sera promu.

FIGURE 24 – Kodama et Kodama samuraï

3.1.3 Version 5 x 6

La version 5x6 diffère de la version 3x4 uniquement sur quelques points, les grands principes restent identiques :

- la taille du plateau est de 5x6 ;
- le seul moyen de gagner une partie est de capturer le Koropokku de l'adversaire ;
- les pièces sont différentes. Chaque joueur possède en début de partie : trois Kodamas (qui se transforment en Kodama samuraï en cas de promotion), deux Onis (qui se transforment en supers Onis en cas de promotion), deux Kirins et un Koropokku.
- la zone de promotion correspond aux deux dernières rangées (cette zone est délimitée par une ligne orange sur le plateau) ;
- deux Kodamas d'un même joueur ne peuvent pas se trouver sur la même colonne.

FIGURE 25 – Disposition des pièces en début de partie version 5x6

3.2 Plateforme de calcul et logiciels utilisés

3.2.1 Machine utilisée pour l'entraînement

L'ordinateur utilisé pour la phase d'entraînement de l'intelligence artificielle est équipé d'un processeur Intel(R) Xeon(R) CPU E5-2407 v2 cadencé à 2.40 GHz, de 16 Go de RAM et d'un GPU Nvidia Titan X (utilisation de CUDA pour effectuer les calculs liés au réseau de neurones). Cet ordinateur est utilisé à distance grâce à SSH. Le logiciel GNU Screen a été utilisé afin de pouvoir laisser une session s'exécuter sans pour autant garder une connexion ouverte en permanence. Cette machine utilise la distribution GNU/Linux Debian comme système d'exploitation.

3.2.2 Bibliothèque utilisée

L'intelligence artificielle a été développée exclusivement en Python. Toute la partie liée au réseau de neurones a été développée à l'aide de la bibliothèque créée par François Chollet, appelée Keras (disponible sous licence MIT) [14]. Keras peut être utilisé comme surcouche de la bibliothèque Theano (développée par l'Institut des algorithmes d'apprentissage de Montréal), mais dans ce projet, Keras a été utilisé comme surcouche de Tensorflow (bibliothèque dotée d'une interface Python, développée par Google sous licence Apache). Keras facilite grandement l'utilisation de TensorFlow en permettant de construire des réseaux de neurones complexes en quelques lignes de codes.

3.3 Création d'une IA

Le travail qui suit se base principalement sur le projet de Surag Nair *et al.* [12]. Dans la suite nous évoquerons les modifications qu'il a fallu apporter afin d'obtenir une intelligence artificielle capable de jouer correctement au Yokai No Mori.

3.3.1 Codage du plateau de jeu et des pièces

Le plateau est encodé sous la forme d'une matrice de 6 par 6 pour la version 5x6 du jeu et d'une matrice de 5 par 4 pour la version 3x4 du jeu. Dans la version 5x6 du jeu une ligne de la matrice est utilisée pour la réserve des joueurs (voir Figure 26). Dans la version 3x4, une ligne de la matrice est

utilisé pour la réserve du joueur 1 et une autre pour la réserve du second joueur appelé joueur -1 (voir Figure 27)

Les pièces sont codées à l'aide de nombres (voir Tableau 1 et Tableau 2), de façon à ce que plus une pièce est importante plus la valeur absolue de son code est grand.

Kodama joueur 1	1
Kodama Samurai joueur 1	2
Oni joueur 1	3
Super Oni joueur 1	4
Kirin joueur 1	5
Koropokkuru joueur 1	6
Kodama joueur -1	-1
Kodama Samurai joueur -1	-2
Oni joueur -1	-3
Super Oni joueur -1	-4
Kirin joueur -1	-5
Koropokkuru joueur -1	-6

TABLE 1 – Encodage des pièces version 5x6

Kodama joueur 1	1
Kodama Samurai joueur 1	2
Tanuki joueur 1	3
Kitsune joueur 1	4
Koropokkuru joueur 1	5
Kodama joueur -1	-1
Kodama Samurai joueur -1	-2
Tanuki joueur -1	-3
Kitsune joueur -1	-4
Koropokkuru joueur -1	-5

TABLE 2 – Encodage des pièces version 3x4

Toutes les cases vides contiennent des zéros.

Au lieu de coder le joueur courant (« Au tour du joueur 1 » ou « Au tour du joueur -1 »), il a été choisi, comme dans le travail réalisé par Nair *et al.* sur Othello, d'utiliser la forme canonique de la représentation. Pour passer

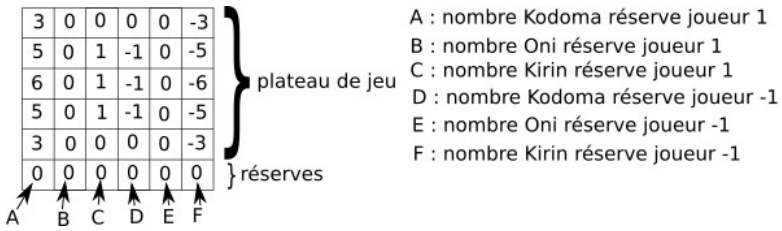


FIGURE 26 – Matrice version 5x6 (début partie)

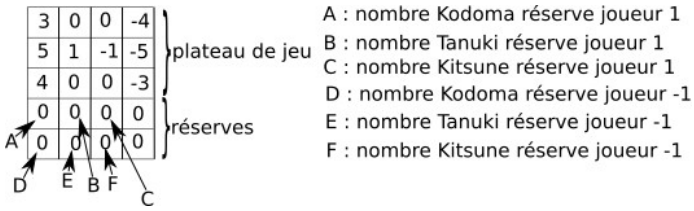


FIGURE 27 – Matrice version 3x4 (début partie)

de la forme de base à la forme canonique, il suffit de multiplier par -1 chaque élément de la matrice, à l'exception de la réserve, si le joueur courant est le joueur -1. Dans le cas où le joueur courant est le joueur 1, il n'y a rien à faire (multiplication par 1). Tout se passe comme si le joueur courant était le joueur 1 à chaque tour, il n'est donc plus nécessaire d'encoder le joueur courant.

Une autre solution a été envisagée : utiliser le codage adopté par Silver *et al.* [2]. La représentation du plateau de jeu et des réserves n'est plus une matrice mais un tenseur. Dans ce tenseur, pour la version 5x6 du jeu, on a pour chaque joueur les 9 matrices suivantes :

— une matrice 5x6 avec la position des Kodamas ;
— une matrice 5x6 avec la position des Kodamas Samourai ;
— une matrice 5x6 avec la position des Onis ;
— une matrice 5x6 avec la position des Super Onis ;
— une matrice 5x6 avec la position des Kirins ;
— une matrice 5x6 avec la position du Koropokkuru ;
— une matrice 5x6 avec les Kodamas en réserve ;
— une matrice 5x6 avec les Onis en réserve ;
— une matrice 5x6 avec les Kirins en réserve.

Enfin une matrice permettant d'encoder le joueur courant. Soit, en tout, 19 matrices et donc un tenseur de 19x5x6 en entrée du réseau de neurones.

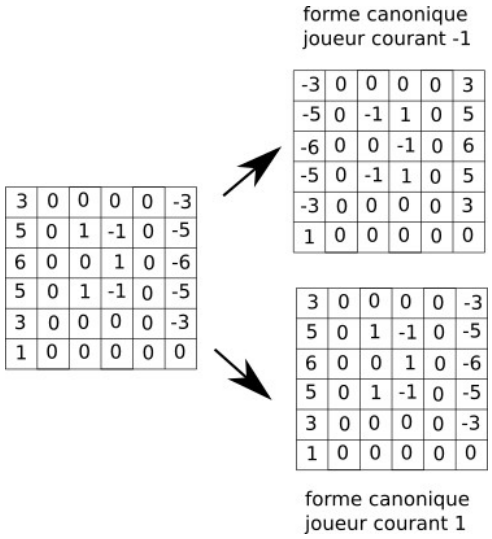


FIGURE 28 – Forme canonique

Cette solution a été testée et très vite abandonnée (aucun entraînement n'a été mené à son terme) vu les lenteurs constatées par rapport à la version « forme canonique ».

3.3.2 Description des fonctions propres au Yokai No Mori

Dans l'idée de fournir des informations pertinentes afin d'effectuer les phases de « self-play », certaines fonctions propres au Yokai No Mori ont dû être développées, voici la description de quelques unes de ces fonctions.

— **getGameEnded**
La fonction *getGameEnded* est utilisée pour savoir si un état est ou non terminal. Dans le cas où l'état est non terminal, la fonction retourne 0. Dans le cas contraire, elle retourne 1 si le vainqueur est le joueur 1 ou -1 si le joueur -1 est vainqueur. Pour déterminer si un état est ou non terminal, il suffit de vérifier si les deux Koropokkuru sont bien présents. Si un des deux manque à l'appel, cela signifie qu'un des deux joueurs a perdu la partie. Le vainqueur est simple à déterminer puisqu'il s'agit du dernier joueur ayant joué. On part du principe que s'il manque un Koropokkuru, ce dernier a été pris lors du dernier coup joué (la fonction *getGameEnded* est utilisée pour chaque nouvel état). À noter que dans la version 3x4 du jeu il est aussi nécessaire de

vérifier qu'aucun des Koropokkuru a atteint la dernière ligne du camp adverse.

— **getValidMoves**
La fonction *getValidMoves* permet de recenser toutes les actions autorisées pour un état s_t donné. Cette fonction s'appuie sur une liste nommé *allPos*. Cette liste référence toutes actions possibles, chaque élément de cette liste est un quadruplet, chaque quadruplet représente une action : (i_1, j_1, i_2, j_2) avec (i_1, j_1) les coordonnées du point de départ de l'action et (i_2, j_2) les coordonnées du point d'arrivée de l'action. *allPos* contient deux types d'actions : les déplacements des pièces (avec éventuellement la prise d'une pièce adverse) et le parachutage d'une pièce, par exemple dans la version 5x6 du jeu, l'action de parachutage d'un Kodama du joueur 1 sur la case de coordonnées i_2, j_2, correspondra au quadruplet (5, 0, i_2, j_2). Il faut bien noter que la liste *allPos* contient toutes les actions possibles (voir Figure 29) : nous avons donc 8 actions possibles pour chaque case « centrale », 5 actions possibles pour chaque case située sur un bord et 3 actions possibles pour chaque case située dans un coin. À tout cela il faut ajouter les parachutages depuis la réserve vers n'importe quelle case du plateau. On dénombre donc 358 actions possibles pour la version 5x6 du jeu et 130 actions possibles pour la version 3x4.

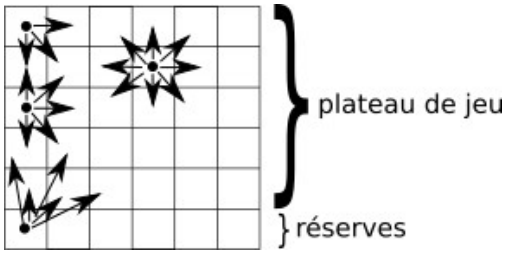


FIGURE 29 – Actions possibles

Pour un état s_t donné, la fonction *getValidMoves* parcourt toute la liste *allPos* et pour chaque action possible a, elle détermine si a est une action possible dans l'état s_t. Une liste *validMove*, de même taille que *allPos*, est complétée avec, pour une action a d'index i dans *allPos*, un 1 à l'index i de *validMove* si l'action a est possible dans l'état s_t et un 0 dans le cas contraire. À la fin de ce processus, nous avons donc une liste contenant des 1 et des 0. Cette liste sera utilisée lors des

phases de « self-play », mais aussi quand l'IA affrontera un humain pour vérifier si les coups proposés par l'humain sont valides.

— **getNextState**
La fonction *getNextState* renvoie la matrice correspondant à l'état s_{t+1} ainsi que le nouveau joueur courant. Elle prend en paramètre la matrice correspondant à l'état s_t, le joueur courant et l'action à effectuer. Cette fonction est utilisée à de nombreuses reprises, que cela soit pendant l'entraînement ou pendant les parties « IA vs Humain ».

D'autres fonctions spécifiques ont du être élaborées mais leur simplicité n'appelle aucun commentaire, elles ne seront pas évoquées ici.

3.3.3 Problème rencontrés

Au niveau du MCTS, dans certaines situations, les états peuvent boucler sur eux-mêmes : à partir d'un état s_t il est possible de passer successivement par les états s_{t+1}, s_{t+2}, s_{t+3} et de revenir à l'état s_t pour recommencer la boucle s_{t+1}, s_{t+2},... (voir un exemple Figure 30). Le problème de ce bouclage

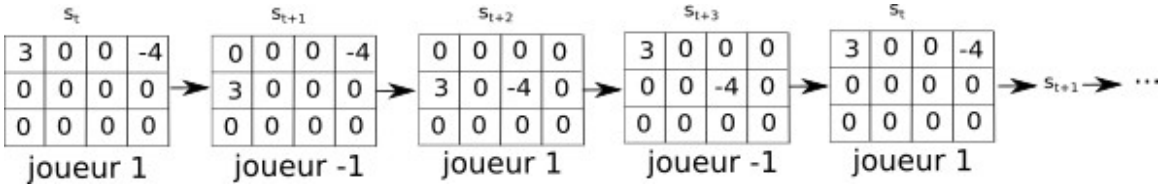


FIGURE 30 – Exemple de boucle MCTS

dans le MCTS est qu'il engendre un nombre potentiellement infini d'appels récursifs. En cas de boucle infinie, on atteint rapidement la limite de profondeur des appels récursifs imposée par le système. Dans ce cas nous avons la levée d'une exception. Ce problème ne se rencontre pas avec la version Othello puisque ce genre de boucle infinie n'est pas possible : en effet, sachant que chaque joueur pose tour à tour ses pions, il est impossible de « repasser » par un état déjà rencontré précédemment. Afin de pallier ce problème, un compteur permet de limiter la profondeur. Si cette profondeur dépasse la valeur $numMCTSSims$, la fonction *search* retourne 0 et stoppe ainsi la boucle.

Ce même genre de boucle peut aussi se rencontrer au niveau du « self-play ». Ici aussi une limite est imposée afin d'éviter les boucles infinies. Une

partie de « self-play » ne peut pas excéder 200 coups (valeur choisie arbitrairement), ici aussi la valeur 0 est retournée en cas de dépassement et la partie concernée n'est pas ajoutée à l'ensemble des exemples.

Enfin dans la phase « θ_{New} vs θ_{Old} », afin ici aussi d'éviter les boucles infinies, le nombre de coups par partie est limité à 500 (valeur arbitraire), si une partie va au-delà, la partie sera considérée comme nulle.

3.3.4 Réseau de neurones utilisé

Afin d'essayer d'augmenter l'expressivité du réseau de neurones et ainsi augmenter l'efficacité de l'intelligence artificielle développée dans ce projet, la profondeur du réseau de neurones a été augmentée par rapport à la version Othello. Pour des raisons qui ont déjà été évoquées plus haut, les blocs résiduels et la normalisation par batch ont été utilisés.

Voici une brève description de ce réseau de neurones (voir Figure 31) :

- **conv-1**
  Couche de convolution (masques de convolution de taille 5x5, output : 256 matrices) avec normalisation par batch et une fonction d'activation ReLU.
- **conv-2 et conv-3**
  Nous avons ici un bloc résiduel composé de deux couches de convolution (pour chaque couche : masques de convolution de 3x3, output : 256 matrices) avec, dans les deux cas, une normalisation par batch et une fonction d'activation ReLU. Ce bloc résiduel étant répété 7 fois on a donc 14 couches.

Le réseau se sépare ensuite en deux parties : une première partie qui aura en sortie le vecteur $\vec{p_\theta}$ et une seconde partie qui aura en sortie V_θ.

- **conv-4** (partie « $\vec{p_\theta}$ »)
  Couche de convolution (masques de convolution de taille 1x1, output : 2 matrices) avec normalisation par batch et une fonction d'activation ReLU.
- **conv-5** (partie « V_θ »)
  Couche de convolution (masques de convolution de taille 1x1, output : 4 matrices) avec normalisation par batch et une fonction d'activation ReLU.
- **Flatten**
  Ces deux couches permettent de passer d'une matrice à un vecteur

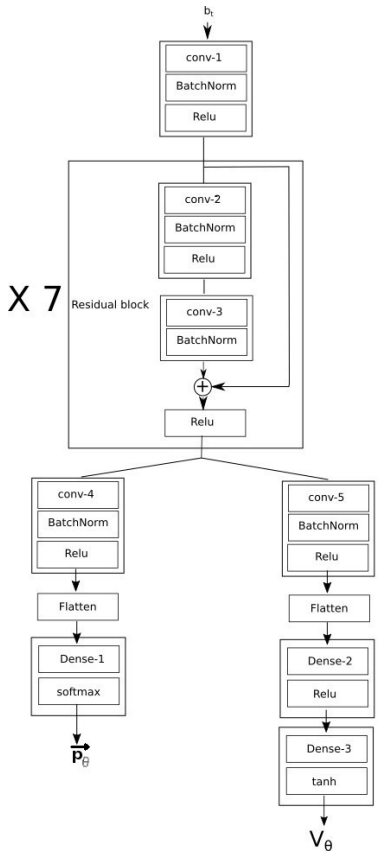


FIGURE 31 – Réseau de neurones

(en « aplatissant » la matrice)

— **Dense-1** (partie « $\vec{p_\theta}$ »)
Couche complètement connectée avec une fonction d'activation softmax. En sortie de cette couche on trouve un vecteur de taille 358 pour la version 5x6 du jeu et 130 pour la version 3x4 (dans les deux cas cela correspond au nombre de coups possibles)

— **Dense-2** (partie « V_θ »)
Couche complètement connectée avec une fonction d'activation ReLU. En sortie de cette couche on trouve un vecteur de taille 256.

— **Dense-3** (partie « V_θ »)
Couche complètement connectée avec une fonction d'activation ReLU. En sortie de cette couche on trouve un scalaire (V_θ).

Nous avons donc en tout 17 couches pour la partie « $\vec{p_\theta}$ » et 18 couches

pour la partie « V_θ ». À noter que les paramètres de ce réseau, ainsi que son architecture, s'inspirent de l'article de David Silver *et al.* [3].

3.3.5 Extension du jeu de données

Toujours dans le souci d'essayer d'améliorer l'IA, des données (triplet (b'_t, $\vec{\pi'_t}$, z_t)) ont été générées à partir des données existantes (triplet (b_t, $\vec{\pi_t}$, z_t)). Pour chaque matrice b_t on génère une matrice b'_t en se basant sur des

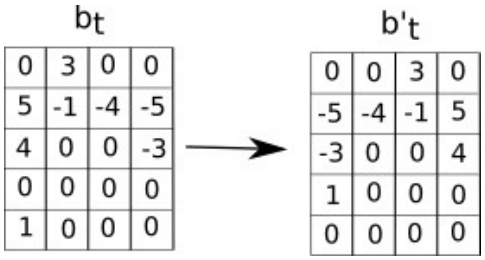


FIGURE 32 – Génération de b'_t

considérations de symétrie (l'axe de symétrie coupe le plateau de jeu dans le sens de la largeur et les réserves sont inversées) voir la Figure 32. Le vecteur $\vec{\pi'_t}$ est lui aussi généré à partir du vecteur $\vec{\pi_t}$ (voir le code Python ci-dessous pour les transformations b_t en b'_t et $\vec{\pi_t}$ en $\vec{\pi'_t}$ pour la version 3x4 du jeu). z_t n'est pas modifiée car la situation du joueur courant n'est pas modifiée. Ce processus permet donc de doubler la taille de l'ensemble d'entraînement et ainsi espérer un réseau de neurones plus efficace.

```
def getSymetrie(self,board,pi,r):
        size=self.getActionSize()
        nPi=[0]*self.getActionSize()
        nBoard=[[0]*4]*5
        # creation de nPi a partir de pi
        for i in range(size):
            if pi[i]!=0:
                i1,j1,i2,j2=self.allPos[i]
                if i1<3:
                    j1=3-j1
                else:
                    if i1==3:
                        i1=4
                    else:
                        i1=3
                j2=3-j2
                index=self.allPos.index((i1,j1,i2,j2))
                nPi[index]=pi[i]
        # creation de nboard a partir de board
        for i in range(3):
            for j in range(4):
                if board[i][j]!=0:
                    nBoard[i][3-j]=board[i][j]
        # inversion des reserves
        nBoard[3][0]=board[4][0]
        nBoard[3][1]=board[4][1]
        nBoard[3][2]=board[4][2]
        nBoard[4][0]=board[3][0]
        nBoard[4][1]=board[3][1]
        nBoard[4][2]=board[3][2]
```

3.3.6 Choix des paramètres

Le choix des différents paramètres a été quelque peu difficile. Les expérimentations auraient dû permettre l'ajustement de ces derniers, mais le temps nécessaire à l'obtention de résultats corrects (voir la section suivante), rend extrêmement difficile ce travail d'expérimentation. La plupart des valeurs n'ont donc pas été modifiées par rapport à la version Othello.

numIters : 1000

Les 1000 itérations ne sont jamais atteintes, le processus d'entraînement est arrêté avant d'atteindre cette valeur. Cependant cela permet d'avoir une cer-

taine latitude sur le nombre d'itération à effectuer.
numEps : 100
la valeur de ce paramètre n'a pas été modifiée.
tempThreshold : 15
Ce paramètre entre dans le choix de la « température » τ : au cours d'un épisode si moins de 15 (*tempThreshold*) coups ont été joués, le paramètre de température sera égal à 1, dans le cas contraire il sera égal à 0. L'idée est donc de favoriser l'exploration en début de partie (moins de 15 coups joués) et l'exploitation par la suite. La valeur de ce paramètre n'a pas été modifiée.
updateThreshold : 0.6
La valeur de ce paramètre n'a pas été modifiée.
numMCTSSims : 50
L'expérimentation a principalement porté sur ce paramètre. La valeur proposé dans la version Othello est de 25. Afin d'augmenter la profondeur du MCTS, cette valeur a été portée jusqu'à 200, sans succès puisque cela rallonge de façon rédhibitoire la durée d'un épisode du « self-play » (jusqu'à plus de 20 minutes par épisode). Une valeur intermédiaire a donc été choisi : 50
arenaCompare : 50
La valeur de ce paramètre n'a pas été modifiée.
cpuct : 1
La valeur de ce paramètre n'a pas été modifiée.
epochs : 10
La valeur de ce paramètre n'a pas été modifiée.

À noter que les paramètres au niveau du réseau de neurones n'ont pas non plus été modifiés à des fins d'expérimentation, toujours pour la même raison, la durée avant d'obtenir des résultats probants pour ces expérimentations est beaucoup trop longue.

3.4 Résultats obtenus

À l'origine, le but de ce projet était de montrer qu'un système de type AlphaZero était capable d'apprendre à jouer au Yokai No Mori. Afin de « prouver » qu'il y avait bien apprentissage, une série de parties, « IA vs joueur qui joue au hasard » (noté « JH » dans la suite), a été menée. Les résultats de ces parties « IA vs JH » sont sans appel, dans plus de 95% des cas l'IA l'emporte, même avec une version ayant effectuée moins d'une

dizaine d'itérations. On peut donc affirmer que l'IA a bien « appris » à jouer au Yokai No Mori.

Ce seul test ne permet pas de juger du niveau atteint par l'IA. Afin de tester plus en profondeur ses capacités, deux solutions s'offraient à nous : organiser une série de parties « IA vs Humain » ou « IA vs IA d'un autre type » (par « IA vs IA d'un autre type », on entend une IA basée sur des algorithmes « classiques », par exemple l'algorithme Minimax). Cette deuxième solution n'a pas été retenue faute de trouver une « IA classique » fiable pour joueur au Yokai No Mori. L'étude plus en profondeur des capacités de l'IA s'est donc basée uniquement sur des parties « IA vs Humain ». Gros défaut de cette solution : il est impossible d'automatiser la procédure et donc d'établir des statistiques fiables. Étant dans l'impossibilité de produire des statistiques, nous devons nous contenter d'un ressenti (toujours subjectif) pour évaluer le niveau atteint par l'IA.

Première chose à noter, si le joueur met volontairement son Koropokkuru dans une position extrêmement défavorable (risque de prise de ce dernier au prochain coup), l'IA profite de cet avantage dans la plupart des cas, même avec peu d'entraînement (5 itérations). Cela aurait donc tendance à montrer que peu d'itérations sont nécessaires pour que l'IA « comprenne » le but du jeu.

Plus généralement, les résultats obtenus diffèrent de façon importante selon la version du jeu (3x4 ou 5x6). Plus précisément voici ce que l'on observe :

— **version 3x4**
  Après une centaine d'itérations, il est difficile, pour un joueur moyen, de battre l'IA. En revanche, si le joueur évite les erreurs grossières, l'IA aura aussi énormément de mal à gagner. Nous avons donc souvent des parties qui s'éternisent. À noter qu'au-delà d'une centaine d'itérations, on constate une stagnation des progrès, même si cette notion de progrès reste très subjective vues les conditions de l'expérimentation.

— **version 5x6**
  Ici aussi l'expérimentation porte sur une version de l'IA qui a effectué une centaine d'itérations. Les résultats obtenus sont moins positifs que pour la version 3x4. En effet, un joueur moyen gagne relativement facilement ses matchs. Au cours d'une partie, l'IA finit souvent par commettre une erreur qui lui sera fatale quelques coups plus tard. Il est assez simple de prendre les pièces de l'IA, elle a en effet la fâcheuse tendance à ne pas les protéger (comportement qui est plus difficile à mettre en évidence dans la version 3x4 vue la taille réduite du plateau

de jeu). Afin d'essayer d'améliorer la situation, il a été tenté d'augmenter le nombre d'itérations, mais comme dans le cas de la version 3x4, le sentiment que l'IA ne progresse plus s'impose assez rapidement. Autre piste suivie, l'augmentation de la valeur de $numMCTSSims$, mais l'inflation de la durée des séquences de « self-play » fait qu'il devient difficile d'atteindre la centaine d'itérations. On pourrait aussi envisager de jouer sur les paramètres *cpuct* et $tempThreshold$ mais comme déjà précisé, il est difficile, vues les durées mises en jeu, de mener a bien une telle expérimentation.

4 Conclusion

En partant du travail réalisé par l'équipe de DeepMind sur AlphaZero et sur l'intelligence artificielle pour jouer à Othello développé par Shantanu Thakoor, Surag Nair et Megha Jhunjhunwala, nous avons vu qu'il était relativement simple de concevoir une intelligence artificielle capable de jouer au Yokai No Mori. Cependant, le niveau atteint par cette intelligence artificielle ne permet pas de rivaliser avec un bon joueur de Yokai No Mori. L'amélioration des performances passe sans doute par une phase expérimentale plus poussée où les différents paramètres seraient testés un par un pendant au moins une centaine d'itérations. Problème : même avec une machine relativement bien équipée, cette centaine d'itérations peut prendre plus d'une semaine. Tester plusieurs valeurs pour un paramètre donné pourrait prendre au minimum un mois. Autre piste pour l'amélioration des performances, l'augmentation de la profondeur de recherche dans l'arbre de recherche de Monte-Carlo. Mais ici aussi, l'augmentation de la durée d'une itération rend l'étude de cette solution difficile. AlphaZero a réalisé 4,9 millions de parties de « self-play » avec pour chaque coup 1600 simulations au niveau du MCTS (contre 50 dans ce projet !). Malgré ce nombre important de simulations, la durée de « réflexion » pour un coup était en moyenne de 0.4 seconde. Il faut dire que DeepMind avait à sa disposition des machines extrêmement performantes : plusieurs cartes graphiques spécialisées, 4 TPU...

Plus généralement, il apparaît que ce type d'intelligence artificielle basée sur les réseaux de neurones profonds nécessite une grande puissance de calcul et énormément de données (même si dans le cas qui nous intéresse ici, ces données sont « autogénérées » grâce à l'apprentissage par renforcement). Peut-être que le but à atteindre dans le futur serait de mettre au point des

systèmes au moins aussi efficaces que ceux qui existent aujourd'hui, mais qui seraient beaucoup moins « gourmands » en terme de ressources (calculs et données). Après tout, un enfant a besoin d'observer uniquement quelques parties de Yokai No Mori pour en comprendre les bases et commencer à y jouer. Depuis l'essor des réseaux de neurones profonds, le terme « intelligence artificielle » est redevenu à la mode dans le grand public. Pourtant on pourrait s'interroger sur l'utilisation du mot « intelligence » quand on constate qu'il faut des millions de photos pour qu'une IA apprenne à différencier un chat d'un chien, alors qu'il suffit de quelques exemples pour qu'un enfant réalise la même distinction. Alors oui, les réseaux de neurones profonds peuvent être très utiles dans certaines tâches bien précises, mais nous sommes encore très loin de Terminator ou de HAL 9000 en matière d'intelligence artificielle.

Références

[1] David Silver and Aja Huang. Mastering the game of go with deep neural networks and tree search. *Nature*, 529 :484–489, janvier 2016.

[2] David Silver, Julian Schrittwieser, and Karen Simonyan. Mastering the game of go without human knowledge. *Nature*, 550 :354–358, octobre 2017.

[3] David Silver, Thomas Hubert, and Julian Schrittwieser. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. décembre 2017.

[4] Richard Sutton and Andrew Barto. *Reinforcement Learning*. MIT Press, 1998.

[5] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow*. O'Reilly, 2017.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. décembre 2015.

[8] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. 2010.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization : Accelerating deep network training by reducing internal covariate shift. mars 2015.

[10] Stuart Russel and Peter Norvig. *Intelligence artificielle*. Pearson Education, 3e edition, 2010.

[11] Cameron Brown, Edward Powley, Daniel Whitehouse, and Simon Lucas. A survey of monte carlo tree search methods. décembre 2012.

[12] Shantanu Thakoor, Surag Nair, and Megha Jhunjhunwala. Learning to play othello without human knowledge. 2017.

[13] Shantanu Thakoor, Surag Nair, and Megha Jhunjhunwala. Learning to play othello without human knowledge projet github. `https://github.com/suragnair/alpha-zero-general`, 2017.

[14] François Chollet. Documentation keras. `https://keras.io/`, 2018.