ÉCOLE CENTRALE LYON

RAPPORT PAR 174
2021/2022

Reinforcement Learning pour les jeux

Étudiants :
Roland ROBERT
Victor LUDVIG

Encadrants :
Abdelmalek ZINE
Alexandre SAIDI
Philippe MICHEL

15 septembre 2022

Table des matières

I Introduction

I.1 Histoire, contexte[5]

En 1842, Ada Lovelace écrit le premier programme informatique. Ce n'est que 100 ans plus tard, dans les années 1940 qu'Alan Turing formalise la science de l'informatique, qui est donc une discipline assez récente.
Le Machine Learning peut se définir comme le fait de donner la capacité à un programme d'apprendre des connaissances à partir de données. Cette définition fait tout de suite penser aux neurosciences, qui intuitivement pourraient sembler un bon modèle pour l'élaboration d'algorithmes de Machine Learning. Cependant, les neurones naturels transmettent des signaux électriques avec des fonctions très différentes des fonctions linéaires utilisées actuellement en Machine Learning. Un réalisme accru des fonctions utilisées en Machine Learning n'a pas permis d'amélioration des performance, c'est pourquoi de nombreux chercheurs étudient le Machine Learning indépendamment des neurosciences. Cela est probablement dû au fait que les réseaux de neurones actuels considérés comme larges sont plus petits que les réseaux neuronales de vertébrés plutôt primitifs tels les grenouilles. Au vu des vitesses d'avancées technologiques actuelles, avec la taille des réseaux de neurones qui double tous les 2.4 ans, ce n'est qu'en 2050 que les réseaux de neurones auront autant de neurones qu'un cerveau humain (100 milliards). Le Machine Learning a été étudié et développé durant trois phases. Dans les années 1940 − 1960, les chercheurs s'inspirent de la biologie, et s'inscrivent dans un courant intellectuel nommé la "Cybernétique". Dans les année 1980 − 1995, le "connectivisme" postule qu'un comportement intelligent peut émerger à partir de nombreux constituants effectuant des opérations de base. Bien que ce courant de recherche n'ai pas porté ses fruits, de nombreux algorithmes utilisés aujourd'hui ont été mis au point à cet époque, tel l'algorithme de rétropropagation, qui sera présenté. Enfin, la dernière vague dans laquelle s'inscrit notre PAr a débuté en 2006 et se poursuit encore de nos jours. Des avancées technologiques ont permis la mise en application d'algorithmes proposés durant la vague précédente de Machine Learning, ce qui a conduit à de nombreuses avancées. En 2012, Google a créé un programme capable d'atteindre un niveau superhumain contre les jeux Atari. Les algorithmes utilisés par Google seront étudiés et appliqués dans le cadre du PAr. En 2016, AlphaGo bat le champion du monde au jeu de GO, avec un algorithme qui ne prend en entrée que les pixels du jeu. Ce type d'algorithme sera mis en place dans le cadre du PAr. Ces algorithmes ont besoin de millions d'exemples pour s'entraîner, contrairement à un humain qui n'a besoin que de quelques exemples pour comprendre le fonctionnement d'un jeu basique. La recherche actuelle porte ainsi sur des algorithmes capables de classifier des images avec une seule

image d'entraînement.

I.2 Introduction au Machine Learning

Le Machine Learning peut être défini comme étant une technologie d'intelligence artificielle permettant l'apprentissage de taches sans avoir été programmé directement pour.

Dans le cadre général, un agent dédié au Machine Learning (très souvent un réseau de neurones) peut être vue comme un fonction transformant une entrée en sortie.

Par exemple, prendre une image d'animal en entrée et retourner en sortie l'animal dont il s'agit (reconnaissance) ou prendre en entrée les données sur un client et prédire quels produits l'intéresseraient (prédiction) ou bien prendre en entrée l'état d'un jeu et retourner en sortie l'action à jouer (reinforcement learning).

Au départ la relation entrée sortie est complètement aléatoire, mais grace à une grande quantité l'agent s'ajuste pour devenir performant (selon le critère d'erreur qu'on lui impose de minimiser)

Le domaine du "Machine Learning" est très vaste allant de simples algorithmes de régression linéaire aux réseaux de neurones complexes dans le cadre du Deep Learning. Les types d'apprentissage se divisent généralement en trois catégories distinctes selon le type de problème et le but attendu :

— **L'apprentissage supervisé** : c'est le cas le plus courant dans le cadre du "Big Data", ce type d'apprentissage a besoin d'une grande quantité de données étiquetées, ou "Labeled Data" c'est à dire des données (entrées) accompagnées de la réponse attendue (sortie). Le système s'ajuste grâce à des échantillons d'entraînement (avec solution), il apprend à prédire la sortie attendue étant donné une entrée. La performance du réseau est mesuré avec un échantillon test jamais vu auparavant.

— **L'apprentissage non supervisé** : pour certaines tache, l'accès à une grande quantité d'exemples avec solution est difficile et on a uniquement accès à des données non étiquetées. Le modèle aura alors aucune manière de mesurer son succès, l'algorithme en question cherchera à regrouper les données dans plusieurs catégories se basant sur des similarités. (regroupement de fleurs, segmentation de clients/de marché, détection d'anomalies...)

— **L'apprentissage par renforcement** : c'est le domaine au coeur de notre PAr, un agent apprend un choisir les "bonnes" actions dans un environnement doté de règles et de récompenses.

C'est le seul apprentissage qui ne nécessite pas directement de données, ces données sont crées à partir de l'environnement.

I.3 Objectifs

Ce Projet de Recherche s'intéresse à l'apprentissage par renforcement appliqué aux Jeux Vidéos. L'objectif est de créer des systèmes d'apprentissages cherchant à jouer le plus optimalement possibles à plusieurs types de jeux. C'est à dire des agent qui essaient de maximiser les récompenses obtenues.

Enjeu : Il n'existe pas de modèle, d'algorithme d'apprentissage fonctionnel pour tous les problèmes d'où la nécessité de comprendre différents systèmes d'apprentissage. Pour un problème donné il peut être nécessaire modifier et mélanger plusieurs méthodes, d'ajuster la structure de données, adapter les hyperparamètres et la fonction perte à un jeu de donnée, le tout pour arriver à des résultats convenables. On peut classifier différents types d'environnements, réponse continue ou discrète (classification), déterministe ou probabiliste, information complète ou incomplète, jeu fini, stationnaire, associatif ou non. Par exemple, une technique d'apprentissage par renforcement très rependue, le Q-Learning fonctionne très bien pour des jeux déterministe à information complète, cependant lorsqu'un jeu même très simple contient un peu de probabilité ces algorithmes peuvent échouer misérablement.
Un deuxième enjeu important est celui des récompenses clairsemés (rares). Un jeu ne vient pas avec son système de récompense, il doit être créée. Pour beaucoup de jeux il est impossible d'évaluer l'action d'un agent avant la toute fin de la partie (lorsqu'il gagne ou perd). Ainsi l'agent ne reçoit que très peu de récompense et il est difficile pour lui de corréler les récompenses avec les actions prises antérieurement. C'est un des plus gros enjeux du Reinforcement Learning.

$$V_{\Pi}(s) = \mathbb{E}\left[\sum \gamma^t r_t | s_0 = s\right]$$

Dans ce PAr nous allons regarder plusieurs jeux de natures différentes :

— Puissance4 : un jeu déterministe à information complète qui se joue contre un autre agent (facteur non prévisible). Ce jeu peut aussi facilement changer de dimension et donc de complexité. Le jeu est résolu en (6×7) et un algorithme MinMax est très efficace en petite dimensions (brute force, parcours dynamique de toutes les possibilités possibles). Il serait intéressant de voir comment apprendre à une IA à gagner à ce jeu.

— Breakout : Jeu à nombre d'état infini. Il faut trouver une méthode qui est capable de prendre une décision alors qu'elle ne peut pas percevoir l'ensemble des états du jeu.

II Etat de l'art

II.1 Apprentissage par renforcement

L'apprentissage par renforcement comme tout problème de Machine Learning, est un problème d'optimisation.
L'apprentissage par renforcement consiste à apprendre à un agent à maximiser ses récompenses futures en ajustant sa politique (comment elle choisit une action). Ainsi l'agent n'apprend pas explicitement à bien agir, ses bonnes actions sont simplement renforcées par une récompense. Cette méthode s'inspire de l'apprentissage d'animaux. L'agent ne reçoit jamais d'instructions explicites, mais ses bonnes actions sont récompensées et l'agent répète les actions qui lui sont fructueuses.

Le principe est illustré ci-dessous, un agent est plongé dans un environnement, à chaque pas de temps il doit choisir une action a_t étant donné l'état qu'il perçoit s_t, l'environnement lui transmet alors un feedback sur la valeur de son action (récompense) et le nouvel état au temps $t+1$.

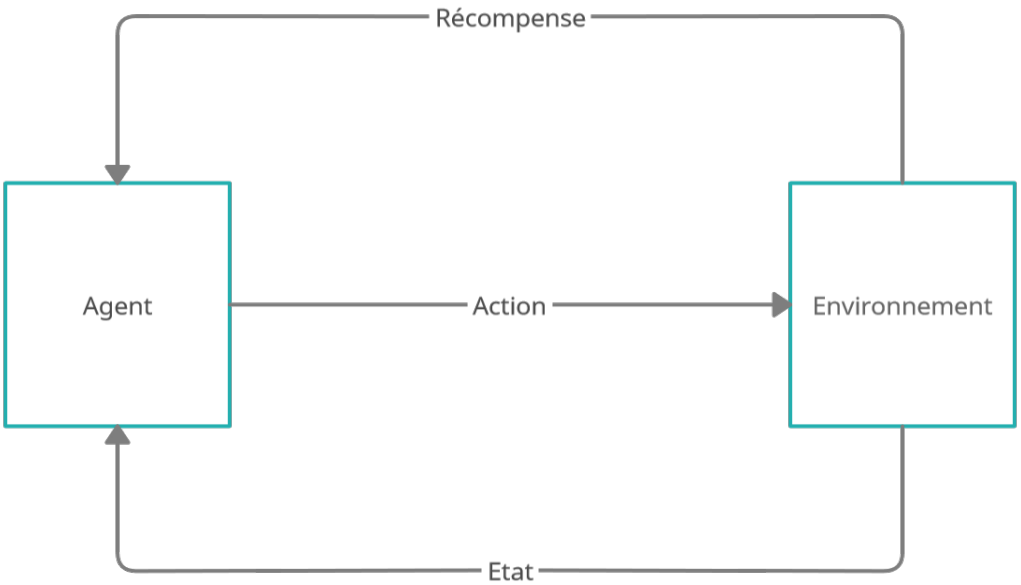

FIGURE 1 – Schéma représentant le contexte du RL

— Environnement : contexte dans lequel est plongé l'agent, l'environnement est définit par ses états, les action possibles et les lois régissant la transition d'état et la distribution de récompense. Les lois de l'environnement sont parfois mobilisables (si état s et action a alors état s' avec probabilité p), ou bien l'environnement peut agir comme une "boite noire".
— Système de récompenses : ce système est définit par l'environnement, il guide l'agent qui cherchera à maximiser cette quantité de par sa politique. La récompense est attribuée à une action par l'environnement.

— Agent : l'entité qui est en charge de la prise de décision.
— Politique : étant donné un état décrivant l'environnement, la politique définit comment choisir l'action. Cette politique est apprise et vise à obtenir le maximum de récompenses.

Le but dans toute la suite est de trouver des systèmes permettant d'apprendre une politique maximisant l'obtention de récompenses futures, étant donné un environnement. La politique est défini par la fonction $\Pi(s,a) = P(a_t = a|s_t = s)$.

II.2 Processus de décision Markovien

Lorsque la loi définissant la transition d'état et l'attribution de récompense dépend uniquement de l'état antérieur et de l'action choisie, alors la tache de l'agent est un processus de décision Markovien. Ce cadre permet de résoudre 90% des problèmes en RL ("Markov property" ref [11]).

$$P(s_{t+1}, r_{t+1}|s_1, a_1, r_1, s_2, ..., s_t, a_t, r_t) = P(s_{t+1}, r_{t+1}|s_t, a_t)$$

Ce cadre permet de définir tout environnement par le quadruplet S, A, T, R

— S : ensemble d'états dénombrable ou continu
— A : ensemble d'actions dénombrable ou continu, certaines restrictions (règles) peuvent rendre cet ensemble dépendant de l'état.
— T : fonction de transition, qui détermine la probabilité d'atterrir dans un nouvel état s' étant donné un état et une action.
$T(s,a,s') = P(s_{t+1} = s'|s_t = s, a_t = a)$
— R : le système de récompense, de façon générale elle associe à un état, une action et un état d'arrivé la probabilité d'obtenir une récompense r :
$R(s,a,s',v) = P(r_{t+1} = v|s_t = s, a_t = a, s_{t+1} = s')$. Souvent ce système ne dépend que de l'état d'arrivée.

Parfois l'environnement est modélisable (la distribution des probabilité des transitions et récompenses sont connus), dans ce cas on peut adopter un système d'apprentissage basé sur ce modèle (Model-based learning vs Model-free learning).

Dans toute la suite on considère être dans ce cadre, et on utilise les notations suivantes :

— Probabilité de transition :
$P(s_{t+1} = s'|s_t = s, a_t = a) = P(s'|s, a)$
— Espérance de la récompense pour une transition :
$\mathbb{E}[r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'] = \mathbb{E}[r|s, a, s']$

II.3 Apprentissage basé sur la valeur

La majorité des méthodes en apprentissage renforcé utilisent d'une façon ou d'une autre une fonction valeur qui estime les récompense futures à partir de cet état (Russell ref [9]).

II.3.a Définition de la fonction valeur

La valeur est définie relativement à une politique, la valeur d'un état peut être définie qualitativement par "les récompenses futures qu'on peut espérer obtenir à partir de cet état avec la politique Π".

Pour préciser cette définition, on introduit le terme de **résultat** R_t d'un épisode à partir d'un instant t : c'est la variable aléatoire égale à la somme des récompenses obtenues pour cet épisode. C'est la quantité que l'agent cherche à maximiser.

$$R_t = \sum_{k=1}^{k=+\infty} r_{t+k}$$

Pour le jeu du puissance 4 avec le système de récompense adopté, le résultat pour n'importe quel temps est tout simplement 10 si l'agent gagne, -10 si il perd et 0 en cas de match nul.

Lorsque l'environnement n'est pas épisodique, ou que la durée d'un épisode est très longue, on introduit le facteur d'atténuation $\gamma \in [0, 1]$ et on redéfinit R_t par :

$$R_t = \sum_{k=1}^{k=+\infty} \gamma^k r_{t+k}$$

Le facteur γ joue un rôle double, il permet de garantir la convergence de la somme et donne moins d'importance aux récompenses éloignées. (Pour $\gamma = 1$ on retrouve la formule précédente, donc dans la suite on considère ce cas général).

La valeur d'un état étant donné une politique peut alors être défini précisément.

Étant donné un état du jeu s et un agent avec une politique Π, la valeur d'un état est l'espérance du résultat d'un épisode à partir de cet état étant donné la processus de décision Π :

$$V_\Pi(s) = \mathbb{E}_\Pi[R_t|s_t = s]$$

De façon analogue, il est possible de définir la valeur (ou **qualité**) d'un couple (état, action) :

$$Q_\Pi(s,a) = \mathbb{E}_\Pi[R_t|s_t = s, a_t = a]$$

(Rq 1 : il est possible d'estimer l'espérance $V_\Pi(s)$ en prenant la moyenne des résultats obtenus en simulant des parties plusieurs fois, à partir de s, avec la politique considérée (Approximation de Monte Carlo)
(Rq 2 : toutes ces formules se simplifient lorsque l'environnement est déterministe)
(Rq 3 : l'utilisation du temps "t" sert juste à clarifier l'ordre chronologie pour bien définir les termes, dans tous les jeux considérés seul l'état en cours est important pour estimer les récompenses futures.)

II.3.b Équation de Bellman et fonction valeur optimale

L'objectif à atteindre peut alors être précisément défini : **trouver une politique qui maximise la valeur de chaque état**. La valeur d'un état obtenue pour cette politique optimale est alors appelée la **valeur optimale** de l'état $V_{opt}(s) = V_{\Pi_{opt}}(s)$. (Dans le cadre markovien, l'existence d'une politique qui maximise chaque état indépendemment est possible)

Dans ce but, Bellman en 1957 trouve une relation fondamentale dans le domaine du RL, l'équation permettant de caractériser récursivement la valeur d'un état, dans le cadre markovien (cf partie II.2) :

$$\begin{aligned} V_\Pi(s) &= \sum_{a\in A} \Pi(s,a) Q_\Pi(s,a) \\ Q_\Pi(s,a) &= \sum_{s'\in S} P(s'|s,a)(\mathbb{E}[r|s,a,s'] + \gamma V_\Pi(s')) \end{aligned} \tag{1}$$

La démonstration de ces résultats est présenté en annexe VII.6

Ensuite Bellman traduit ces relations pour la politique optimale :

$$\begin{aligned} V_{opt}(s) &= \max_{a\in A} Q_{opt}(s,a) \\ &= \max_{a\in A} \sum_{s'\in S} P(s'|s,a)(\mathbb{E}[r|s,a,s'] + \gamma V_{opt}(s')) \end{aligned} \tag{2}$$

OU

$$\begin{aligned} Q_{opt}(s,a) &= \sum_{s' \in S} P(s'|s,a)(\mathbb{E}[r|s,a,s'] + \gamma V_{opt}(s')) \\ &= \sum_{s' \in S} P(s'|s,a)(\mathbb{E}[r|s,a,s'] + \gamma \max_{a' \in A} Q_{opt}(s',a')) \end{aligned} \tag{3}$$

Cette équation semble compliqué, mais traduit simplement la phrase suivante :

Valeur(état,action) = Récompense instantanée + Valeur de l'état d'arrivée (réduite par γ).

Rq : Lorsque l'environnement est entièrement déterministe, c'est à dire que l'état d'arrivée s' et la récompense attribuée dépendent uniquement de l'état de départ et de l'action prise : $s' = s'(s,a)$, $r = r(s,a)$:

$$V_{opt,deterministe}(s) = \max_{a \in A}(r(s,a) + \gamma V_{opt,deterministe}(s'(s,a))) \tag{4}$$

Lorsqu'il est possible de calculer $Q_{opt}(s,a)$, alors le problème est résolu, il suffit d'adopter la politique :

$\Pi_{opt}(s) = argmax(Q_{opt}(s,\cdot)) \Leftrightarrow \Pi_{opt}(s,a) = \mathbf{1}_{a=argmax(Q_{opt}(s,\cdot))}$.

Cependant, dans la majorité des cas, ce calcul directe de V_{opt} n'est pas envisageable : soit l'espace des actions/états est beaucoup trop vaste (ex : jeu de Go) soit les lois régissant l'environnement sont inconnus (ex : prédiction Météo).

L'enjeu à présent est de savoir comment apprendre à un agent cette la politique Π_{opt} tel que $V_{opt} = V_{\Pi_{opt}}$.

II.3.c Programmation dynamique et convergence vers une politique optimale

Lorsque le modèle de l'environnement est connu, il est possible de définir un algorithme simple pour converger vers une politique optimale, les deux méthodes répendue basées sur l'équation de Bellaman sont le "policy iteration" et le "value iteration".

Le principe du "value iteration" est le suivant :

— Initialiser une fonction $V(s)$ aléatoirement (peut aussi être vu comme une grande table qui associe une valeur à un état)
— Pour chaque état $s \in S : V(s) \leftarrow \max_{a \in A} \sum_{s' \in S} P(s'|s,a)(\mathbb{E}[r|s,a,s'] + \gamma V(s'))$

Le principe du "policy iteration" est légèrement différent, en considèrant une politique déterministe : $\Pi(s) = a$.

— Initialiser une fonction $V(s)$ et une politique $\Pi(s)$ aléatoirement
— Pour chaque état $s \in S : V(s) \leftarrow \sum_{s' \in S} P(s'|s,a)(\mathbb{E}[r|s,a,s'] + \gamma V(s'))$
— Pour chaque état $s \in S : \Pi(s) \leftarrow arg\max_{a \in A} \sum_{s' \in S} P(s'|s,a)(\mathbb{E}[r|s,a,s'] + \gamma V(s'))$

Dans les deux cas, si le processus est itéré un grand nombre de fois, la convergence vers une politique optimale est garantie, (cf [1]).

Cependant ces algorithmes nécessitent une grande puissance de calcul puisque tous les états sont considérés ($O(n_{itrations} \times \#S^2)$).

La suite s'intéressera à des méthodes plus performantes, très répandues, basés sur l'apprentissage par l'expérience : jouer un grand nombre de partie, équilibrer exploitation et exploration, en ajustant après chaque partie la politique : au lieu d'itérer sur tous les états, on itère sur les états que la politique est susceptible d'atteindre.

II.4 QLearning et SARSA

Les algorthmes d'apprentissages QLearning et SARSA sont quasiment identiques, sauf que le SARSA à l'inverse du QLearning, est un apprentissage basé sur la politique (on-policy).

Ces systèmes d'apprentissage sont adaptés à des états discrétisés et une quantité finie d'actions. L'agent cherche à approximer la qualité d'une action donné, c'est à dire l'espérance des récompenses futures ajusté par un facteur d'actualisation $\gamma \in [0,1]$ comme vu dans la partie II.3.

Le principe du QLearning et du SARSA (State Action Reward State Action) est le suivant : l'IA utilise une table $Q(s,a)$, cette valeur représente la qualité de l'action a (cf II.3. La politique de l'IA est alors simple : pour un état donné, jouer aléatoirement avec une probabilité ϵ, sinon joueur $arg\max_{a \in A} Q(s,a)$, ce type de politique est appelé : ϵ-greedy policy.

Les hyperparamètres (fixés en amont de l'apprentissage) de ces algorithmes sont les suivants :

— $\alpha \in [0,1]$: Taux d'apprentissage ou "learning rate" qui définit la vitesse d'apprentissage de l'agent, plus ce coefficient est grand plus sa stratégie fluctue vite.

— $\gamma \in [0,1]$: Facteur de remise/actualisation/atténuation ou "discount factor", plus il est proche de 1, plus les récompenses futures sont considérées comme importantes.

— $\epsilon \in [0,1]$ facteur d'exploration ou "randomness", durant la phase d'entraînement la politique de l'agent est légèrement modifié, celui-ci choisit une action aléatoire avec une probabilité ϵ afin d'explorer l'espace d'action d'état. (Rq : il est possible de faire dépendre ϵ du nombre de parties jouées (époque), plus l'agent est entrainé moins il décide d'explorer)

Une table vide Q est initialisée (aléatoirement ou à 0), au départ cette table est une très mauvaise approximation de la qualité. Afin d'apprendre la qualité des couples (état, action), l'agent passe par la phase d'entraînement. Durant cette phase l'agent joue dans l'environnement selon la politique ϵ greedy.

Ensuite grâce aux réponses de l'environnement, l'agent met à jour les valeurs de son dictionnaire selon l'équation de Bellman en utilisant le quadruplet (s,a,s',r) = (état, action, nouvel état, récompense) :

— Pour le QLearning

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma max(Q(s',\cdot))$$

ou

$$\Delta Q(s,a) = \alpha(r + \gamma max(Q(s',\cdot)) - Q(s,a))$$

— Pour SARSA ($\Pi(s')$ est une fonction aléatoire, qui vaut $max(Q(s',\cdot)))$ avec probabilité $(1-\epsilon)$)

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma Q(s',\Pi(s')))$$

ou

$$\Delta Q(s,a) = \alpha(r + \gamma Q(s',\Pi(s')) - Q(s,a))$$

L'équation se traduit par le principe suivant : plus la récompense de l'action est grande ou l'état futur est prometteur, plus la valeur Q(Etat,Action) augmente. Ces équations de mise à jour sont basés sur l'erreur TD(0) (Temporal Difference) (cf [11]).

L'inconvénient majeur de ces méthode d'apprentissage est que lorsqu'un état n'a pas encore été visité un coup aléatoire est joué. De plus le dictionnaire contient des clefs

pour tous les cas possibles, et donc la complexité en mémoire devient vite très grande.

Pour le morpion avec un nombre de couple (Etat, Action) relativement faible (de l'ordre de 400 000), l'IA peut visiter toutes les possibilités et apprendre les actions à jouer efficacement.
Pour le jeu de Go cependant, avec 10^{160} états possible, le QLearning est pratiquement inutile, presque chaque partie jouée sera entièrement différente.
C'est pourquoi le Deep QLearning est préférable.

II.5 Deep QLearning

Le Deep QLearning est une extension du QLearning qui utilise un réseau de neurones. Le réseau de neuronne prend l'état du jeu en entrée, et retourne les QValues de cet état. La dimension de l'input est donc celle de l'état, et la dimension de l'output est égale au nombre d'actions possibles.

La complexité en espace est alors constante et le réseau de neurones peut identifier un état jamais vu auparavant avec des états similaires et avoir une réponse plus cohérente que pour le QLearning simple. Le Deep Q Learning peut aussi fonctionner sur des états continus.

Ainsi il est préférable d'utiliser une méthode utilisant un réseau de neurone. Pour le QLearning lorsque l'IA visite un état qu'il n'a jamais vu avant, il joue aléatoirement ce qui est l'inconvénient majeur de ce système d'apprentissage. Le réseau de neurones quant à lui peut identifier un état jamais vu auparavant avec des états similaires et avoir une réponse plus cohérente.

La politique de cet IA est analogue à la politique du QLearning. Il joue parfois aléatoirement pendant la phase d'entraînement. Sinon il choisit l'action avec la plus grande QValue. Ensuite l'équation de Bellman est utilisée afin de trouver artificiellement "la sortie attendue" pour l'état donnée en entrée.

$$Q_{attendu}(s,a) = Recompense + \gamma max(Q(s,:))$$

Ensuite les poids du réseau de neurones sont ajustés comme dans le cadre du Machine Learning Supervisé (Data science) (voir partie VII.2) :

$$poids_i \leftarrow poids_i - \alpha \nabla_i Erreur(s, Q_{attendu}(s))$$

II.6 Mémoire et batch learning

Un entraînement pas à pas du réseau à chaque coup fournit souvent de mauvais résultats, le réseau apprenant mal de cette manière.

Pour le Deep Q Learning, garder en mémoire un grand nombre de quadruplés (s, a, s, r) pour s'entraîner sur beaucoup de données permet un apprentissage plus stable et efficace.

Par exemple, une mémoire des 10000 derniers coups, puis à chaque partie le réseau est entraîné sur un échantillon aléatoire de 256 quadruplés.

Ensuite pour entraîner le réseau lui même on choisit un "batch size" lors de l'entraînement, il estime alors le gradient en moyennant sur plusieurs couples $(s, Q_{attendu}(s))$, le gradient sur les poids est alors estimée de façon plus précise.

III Étude Puissance 4

Tous les codes, ainsi qu'une interface graphique avec tous les agents entraînés sont disponibles sur le github : `https://github.com/roland-robert/puissance4_RL`

III.1 Objectif

Le but de cette partie est de créer un système d'apprentissage capable d'apprendre à jouer au puissance 4, et converger vers une politique maximisant les récompenses qu'il obtient.

Le système de récompenses pouvant être défini de différentes manières, classiquement on attribue +1 à une victoire, -1 à une défaite et 0 en cas d'égalité. (voir III.4)

Pour un agent donné, la maximisation des récompenses dépend de l'adversaire en face (voir III.5). La mesure de la performance d'un agent se fera en considérant des adversaires avec des politiques fixes (voir partie III.6).

III.2 Implémentation du jeu (Python et Tkinter)

Le jeu a été codé de sorte à pouvoir choisir les dimensions hauteur x largeur du plateau (typiquement h=6 et w=7) pour pouvoir comparer l'efficacité des systèmes d'apprentissage adoptés selon la taille du plateau.

Le plateau (board) est une matrice de dimension (h,w) qui contient des 0 pour les cases vides, des 1 pour les cases rouges et des -1 pour les cases bleues.

Le jeu, avec toutes ses règles et les méthodes nécessaires, a été codé dans un fichier puissance4.py, puis une interface graphique puissance4_GUI.py permet de jouer agréablement et de tester les différents agents (voir VII.11).

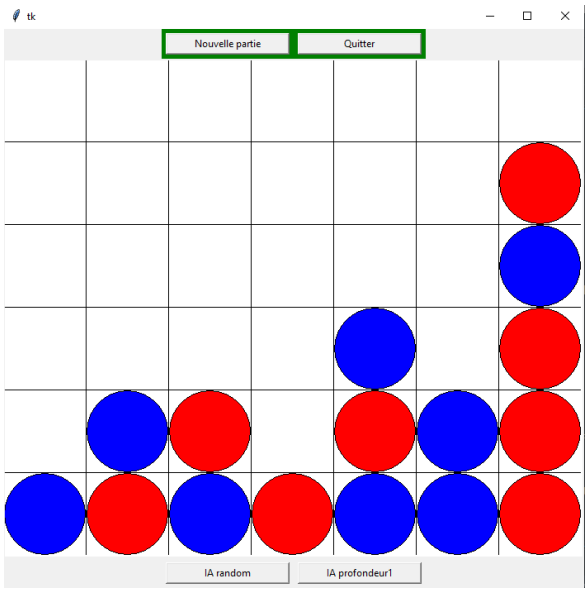

FIGURE 2 – Exemple de l'interface graphique h=6 w=7

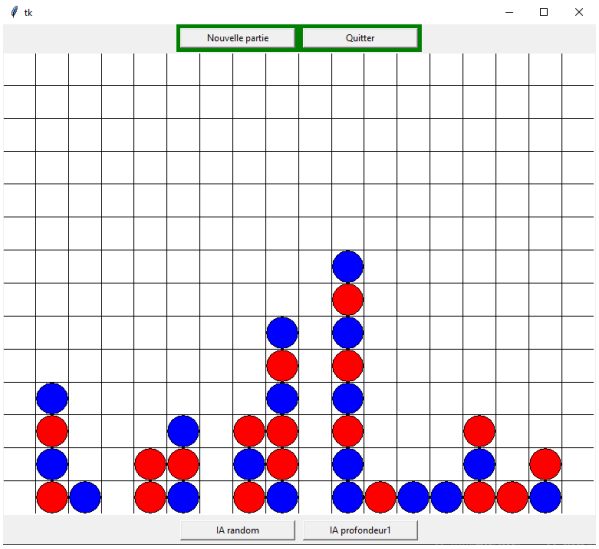

FIGURE 3 – Exemple de l'interface graphique h = 14, w = 18

Le fonctionnement général du code est présenté en annexe, figure 40.

III.3 Définition et modélisation

Une action est toujours modélisé par un entier $j \in [0, w-1]$ représentant la colonne sélectionnée.

L'état du jeu peut être modélisé de plusieurs façons, le choix de la modélisation de l'état joue un rôle clef dans l'apprentissage et peut faire la différence.
(Rq : pour le puissance 4, le tour du joueur n'est pas nécessairement à préciser dans l'état car celui-ci peut être déduit, cependant c'est une bonne idée de le faire).

Quatre modélisations d'états sont utilisé :

— Chaîne de caractères : pour le QLearning, l'état est la clef d'un dictionnaire
— Vecteur taille $h \times w$: pour le DQL avec réseau dense
— Matrice taille $(h, w, 1)$: pour le DQL avec réseau convolutif
— Matrice bitmap taille $(h, w, 3)$: pour la partie MCTS (réseau de valeur de politique)

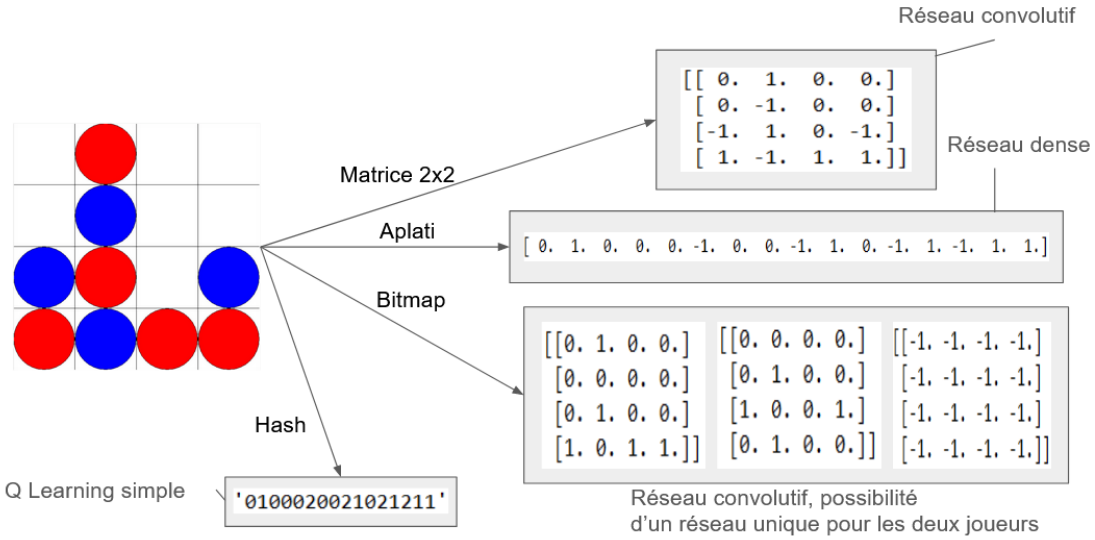

FIGURE 4 – Représentation de l'état

III.4 Système de récompense

Le jeu du puissance étant à somme nulle, un système de récompense logique serait d'attribuer une récompense de +1 lors d'une victoire, -1 lors d'une défaite et 0 sinon.

Les récompenses ne dépendent donc pas seulement de la transition (état, action, nouvel état) mais aussi de la couleur jouée. La récompense d'un état pour la couleur bleue est opposée à la récompense du même état pour la couleur rouge.

Ainsi il peut sembler pertinent de créer deux agents séparément pour chaque couleur, qui aura alors un résultat bien défini (récompense dépendant uniquement de l'état d'arrivée).

Le système de récompense attribue que très rarement des récompenses (fin de partie), ceci rend l'apprentissage difficile (aucun feedback direct sur les coups intermédiaires).

III.5 Étude théorique du puissance 4

L'objectif de tout Reinforcement Learning est de maximiser l'espérance des récompenses futures. Cependant pour le puissance 4, la politique maximisant cette espérance dépend intrinsèquement de la politique de l'adversaire.

Même si le puissance 4 est déterministe, du point de vue d'un agent, une action dans un état donné n'entraîne pas nécessairement toujours le même nouvel état et la probabilité de transition $P(s'|s,a)$ (cf markov partie II.2) est indéfinissable de façon générale.

En effet, $P(s'|s,a)$ est directement lié à la politique de l'adversaire, donc le puissance 4 n'est pas dans le cadre d'un processus de décisions markovien.

Trois solutions :

— L'agent joue contre un autre agent (les deux agents étant initialisés aléatoirement), la propriété de Markov n'est pas respectée, mais on peut toujours utiliser les méthodes habituelles et observer le résultat.
(Un entraînement des deux agents en simultané entraîne un changement continu des lois de l'environnement, il peut donc être préférable d'entraîner chaque couleur de façon différé).

— L'adversaire peut être intégré dans l'environnement pour retrouver la propriété de markov (II.2).
On entraîne un agent contre un algorithme suivant des règles précises (pas nécessairement déterministe) (ex : minmax). Avec les méthodes classiques (QLearning, SARSA...), on peut alors s'attendre à une convergence vers une politique optimale maximisant les récompenses futures contre cet algorithme.

— Abandonner complètement l'idée de "récompenses" et entraîner un réseau à estimer la valeur d'un état en utilisant les règles du jeu pour parcourir un arbre d'états. Approche adoptée à la toute fin dans la partie Recherche arborescente Monte Carlo.

Il est important de remarquer que la politique soi-disant "optimale" ne maximise pas nécessairement les récompenses. En effet, un "mauvais" coup peut être fructueux contre un mauvais adversaire. On remarquera que l'entraînement contre une IA aléatoire provoque des comportements étranges dus à ce phénomène, la maximisation des récompenses se fait en jouant des "mauvais" coups.

III.6 Mesure de la performance

Afin de mesurer la performance d'un agent, une IA minmax a été codé. La profondeur de l'IA détermine combien de coups en avance celle-ci regarde, elle joue alors le meilleur coup selon cette profondeur.

Lorsque plusieurs coups sont jugés de même qualité : choix d'un coup aléatoire parmi les meilleurs.
En résumé : pour une IA minmax de profondeur p, si elle peut gagner à coup sûr après p coups elle le fait, si un coup entraîne une défaite sûre après p coups elle évite ce coup. Sinon elle joue aléatoirement (ou le plus à gauche).

Un tableau en annexe présente les résultats en 4x4 et 6x7 des algorithmes MinMax jouant entre eux : VII.10

(profondeur 0 : joue aléatoirement, profondeur 1 : gagne en 1 si possible, profondeur 2 : gagne en 1 et bloque en 1 si possible, ...)

III.7 Méthode du QLearning

III.7.a Principe adapté au puissance 4

Dans un premier temps, il est possible d'implémenter un système d'apprentissage renforcé : le QLearning.
Cette méthode est expliquée en détail partie II.4.

Cette méthode doit être ajustée au puissance 4 qui inclus deux joueurs, pour une action donnée l'état suivant dépend de ce que joue l'adversaire.

Le principe d'apprentissage implémenté est résumé par le pseudo-code suivant :

```
pour i = 1, 2, ..., N_parties faire
    Initialiser nouvelle partie
    tant que Partie en cours faire
        Récupérer et jouer coup selon la politique du QLearning
        Stocker le quadruplet [s, a, s', r]
        Mettre à joue la table selon l'équation de Bellman
        Q[s][a] ← Q[s][a] + α(r + γmaxQ[s'][:])
    fin
fin
```

Algorithme 1 : Algorithme du QLearning simple

(Rq : le quadruplet [s, a , s', r] est récupéré de façon différée : après avoir joué un coup, l'état obtenu est le Nouvel état pour le coup précédent.)

III.7.b Résultats

On choisit dans un premier temps les paramètres $\alpha = 0.1, \gamma = 0.95$ et $\epsilon = 0.1$. On observe ensuite le résultat en fonction du nombre d'entraînements (résultats = taux de victoire - taux de défaite), tous les 10 000 entraînements, 6 x 300 parties testes sont effectuées pour chaque agent, contre chacune des 3 IAs MinMax.

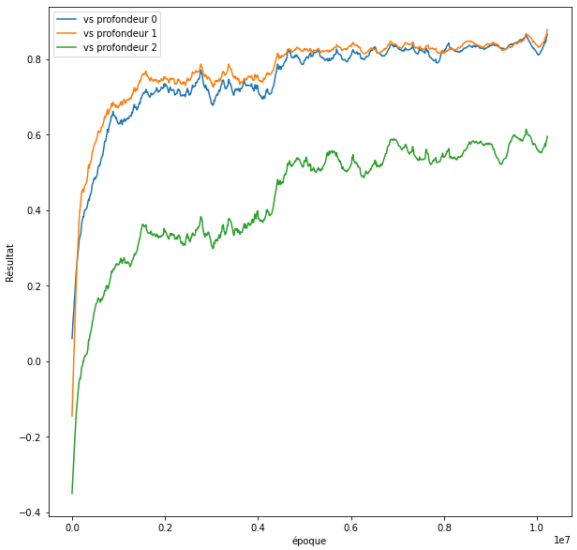

FIGURE 5 – Résultats du QL 4x4 pour l'agent X, interpolé (Savitzky Golay)

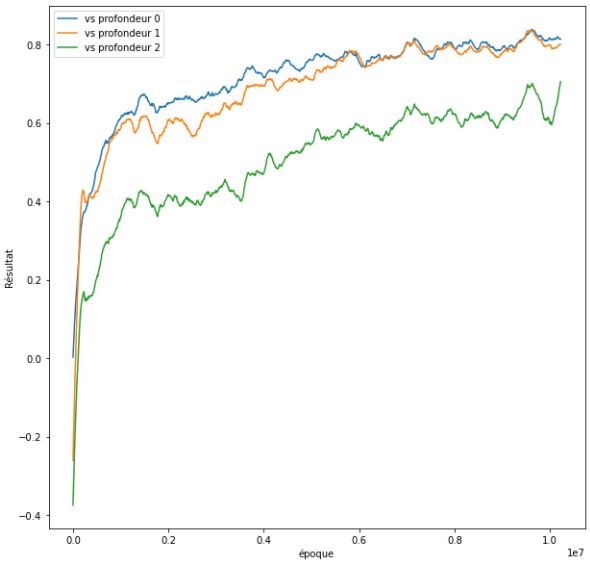

FIGURE 6 – Résultats du QL 4x4 pour l'agent O, interpolé (Savitzky Golay)

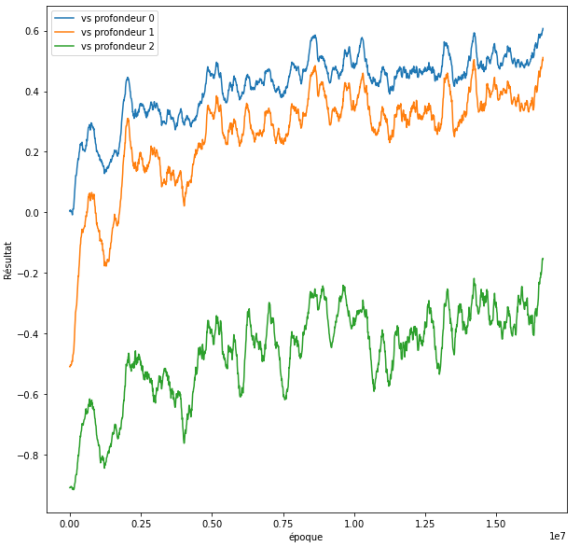

FIGURE 7 – Résultats du QL 6x7 pour l'agent X, interpolé (Savitzky Golay)

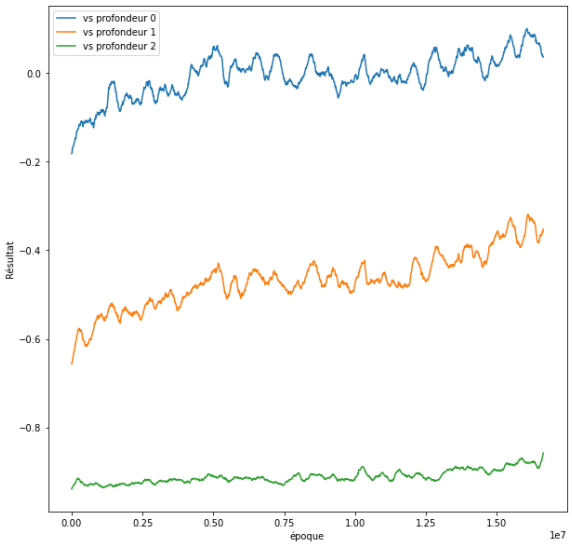

FIGURE 8 – Résultats du QL 6x7 pour l'agent O, interpolé (Savitzky Golay)

Le temps d'entraînement pour le 4x4 est : 1s pour 1000 parties environ.

Le temps d'entraînement pour le 6x47 est : 2s pour 1000 parties environ.

Pour un plateau 4x4, l'agent optimise rapidement ses récompenses, après un nombre de parties raisonnable l'algorithme finit par jouer presque optimalement. En comparant avec les résultats en annexe VII.10, un minmax profondeur 6 contre un minmax profondeur 2 a un résultat entre 0.1 et 0.2, ce qui montre que notre agent QL 4x4 est performant avec un résultat autour de 0.6.

Cependant pour un plateau de 6x7 l'espace État Action est trop vaste, l'apprentissage est très lent, et les résultats sont très médiocres.

De plus la complexité en mémoire est gigantesque :

— En 4x4 : En tout l'agent a visité 116 912 états distincts sur 14 Milions de parties jouées. Donc l'agent a pu itéré un grand nombre de fois sur chaque état, d'où les résultats satisfaisants.

— En 6x7 : En tout l'agent a visité 25 825 600 états sur 16 Millions de parties jouées. Moins de 1% des états ont été visités (environ 4×10^{12} états en tout en 6x7), et l'agent n'a pas itéré un grand nombre de fois sur les états qu'il a visité. D'où les résultats insatisfaisant.

Le dictionnaire enregistré après 16 Millions de parties fait 2.07 Go et prend plusieurs minutes à charger dans l'environnement python.

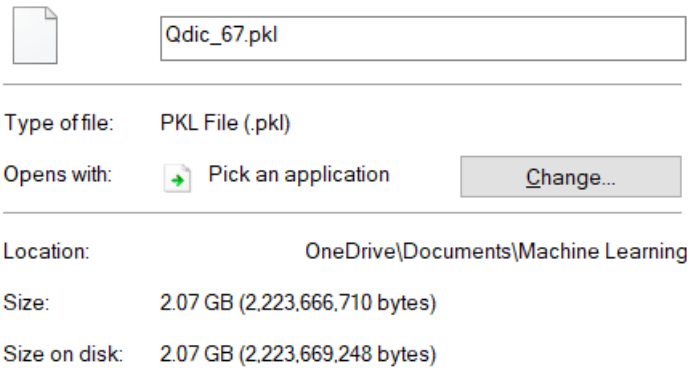

FIGURE 9 – Illustration de la taille en mémoire du dictionnaire obtenu après 16 Milions d'entraînements en 6x7

Ainsi, même avec un temps infini de calcul et un dictionnaire Q optimal, celui-ci utilisera environ 320 Terra octets de stockage.

La section suivante va s'intéresser au Deep Q Learning qui permet à la fois de compresser l'espace de stockage (un réseau de neurones, au lieu de 10^{10} lignes dans un tableau), et reconnaître les ressemblances entre états (évite de jouer aléatoirement sur un état inexploré).

III.8 Deep Q Learning

III.8.a Principe adapté au puissance 4

Lorsqu'il n'est plus envisageable d'explorer tous les états de l'environnement quand h x w devient grand, il est nécessaire de trouver un moyen pour généraliser les connaissances de certains états à des états similaires. L'idée étant de trouver des "pattern" sans devoir explorer tous les états.

Pour cela, l'état du jeu est placé à l'entrée d'un réseau de neurones, la sortie du réseau représentant les Q values des action pour cet état.

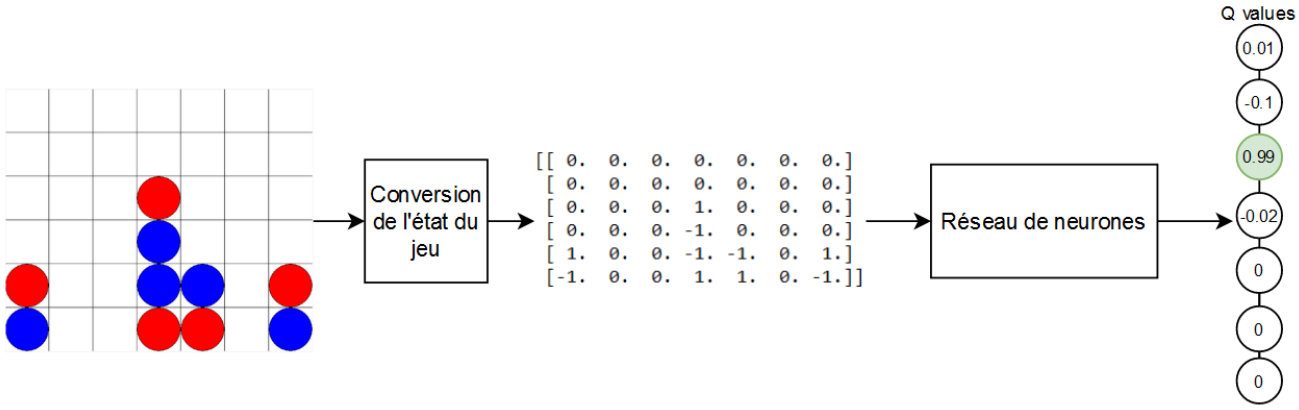

FIGURE 10 – Schéma du fonctionnement du DQL

Le principe du Deep Q Learning est expliqué plus en détail partie II.5

III.8.b Choix du réseau de neurones

Plusieurs choix de réseau de neurones sont envisageables. L'étude va se limiter à deux grandes catégories de réseaux : Réseau dense entièrement connecté ou Réseau convolutif. L'état du jeu étant représenté par un vecteur (h x w,) dans le premier, ou une matrice (h x w x 1) dans le second

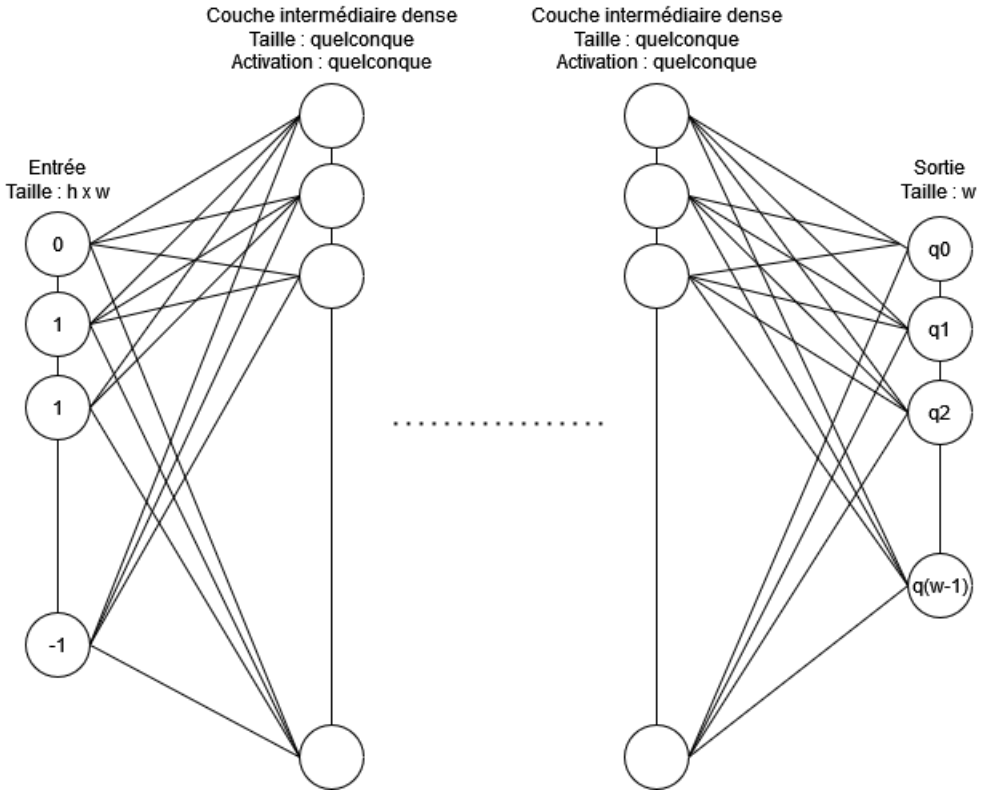

FIGURE 11 – Schéma général d'un réseau dense pour le puissance 4

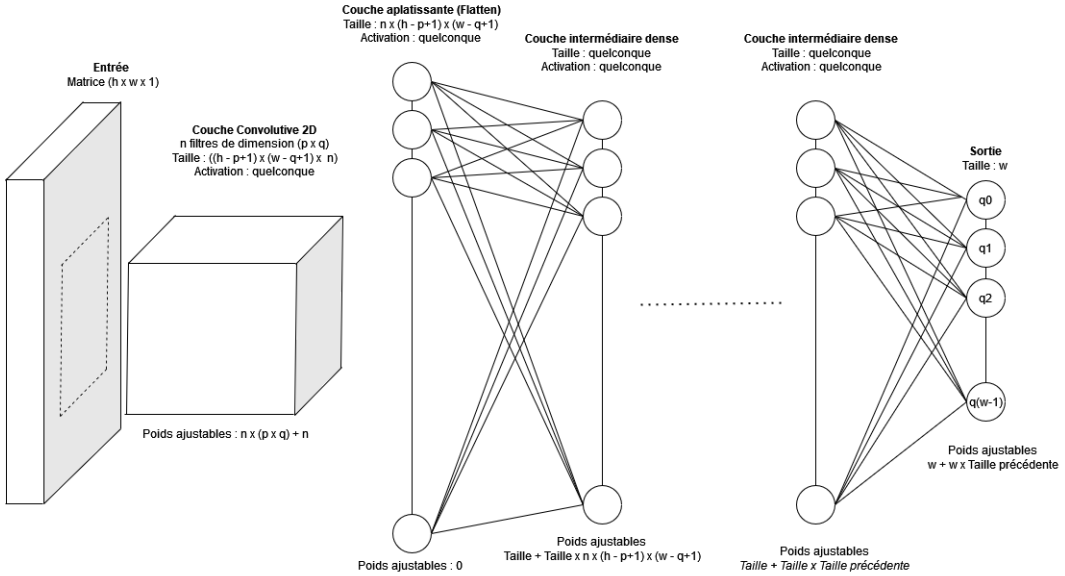

FIGURE 12 – Schéma général d'un réseau convolutif pour le puissance 4

Ensuite il reste à choisir le nombre de couches, la taille de chaque couche et les fonctions d'activations associées (reLU, sigmoid, tanh...). Les détails théoriques concernant les réseaux de neurones sont présentés en annexe VII.2.

III.8.c Choix des hyperparamètres

Une fois la structure du réseau, la taille de chaque couche et les fonctions d'activations choisies, il reste à choisir l'optimisateur du réseau, comment celui-ci va ajuster ses poids avec les données qu'il reçoit.

— Choix de la fonction perte : MSE, RMSE, Huber, Crossentropy...
— Choix de l'optimisateur : SGD, Adam...
— Choix du pas d'apprentissage (learning rate), et autres paramètres de l'optimisateur

Les choix effectués sont expliqué partie III.8.h

III.8.d Déroulement de l'entraînement et mémoire

Dans le cadre de la partie Deep Q Learning, deux réseaux indépendants sont créés, pour le joueur Rouge et Bleu. Ensuite ces réseaux jouent un grand nombre de fois. Les poids du réseau sont ajustés selon une descente du gradient grâce à l'historique des coups : (état, nouvel état, action, récompense) et l'équation de Bellman.

```
pour i = 1, 2, ..., N_parties faire
    Initialiser nouvelle partie
    tant que Partie en cours faire
        Récupérer et jouer coup selon la politique du DQLearning
        Stocker le quadruplet [s, a, s', r] dans la mémoire de l'agent (file)
    fin
fin
Entraîner l'agent après un certain nombre de partie*
```

Algorithme 2 : Algorithme du Deep Q Learning

Les détails de l'entrainement sont présentés ci-dessous :

```
Entrées : Mémoire, taille échantillon, batch_size
Récupérer un échantillon de la mémoire (liste de quadruplés)
pour i = 1, 2, ..., taille_chantillon faire
    [s, a, s', r] = échantillon[i]
    Récupérer Q(s) et Q(s') (à partir du RN)
    Q_attendu(s) ← Q(s)    (vecteur    (w, ))
    Q_attendu(s)[a] ← r + γ max Q(s')[:]
fin
Entraînement du Réseau sur les données : (états, Q_attendu(tats))
```

Algorithme 3 : Entrainement Deep Q Learning

III.8.e Résultats DQL

Remarque : Les choix des réseaux, hyperparamètres et conditions d'entraînements sont le fruit de beaucoup d'expériences et de tests. Dans les résultats seul un choix final est présenté.

III.8.f Entraînement seul

Lors d'un entraînement seul, deux problèmes théoriques sont présents : tout d'abord la propriété de Markov n'est pas vérifiée, selon chaque agent les lois de transition d'état varient après chaque partie. De plus l'agent s'entraîne contre un autre agent imparfait, donc une bonne performance contre une IA minmax (jamais rencontrée) n'est pas garantie.

Résultat en 4x4, en choissants le réseau convolutif suivant :
Entrée((4,4,1)), Conv2D(256, (4, 4), "selu"), Aplatir, Dense(128, "tanh"), Dense(128, "elu"), Sortie(4)
En tout 54,276 paramètres modifiables.

Les poids sont initialisés selon une Gaussiene centrée en 0 d'écart type 0.1. Le taux d'apprentissage utilisé est de 0.1, le facteur d'exploration varie entre 0.99 et 0.1 (selon l'époque), l'optimisateur utilisé est une descente du gradient stochastique, la fonction perte est le Mean Squared Error.

Ensuite l'entraînement se fait avec une mémoire des coups de 10000, et 256 entraînements après chaque partie (échantillon de la mémoire) avec à chaque fois 32 points pour estimer le gradient (batch size).

Le temps pour effectuer 1000 entraînements est d'environ 440s. On effectue 50000 entraînements au total.

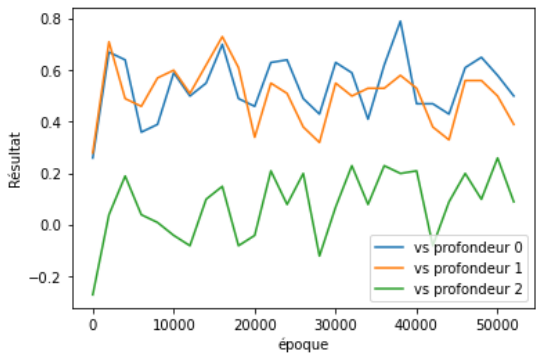

FIGURE 13 – Résultat pour le DQL 4x4 : Joueur rouge contre différentes IAs minmax

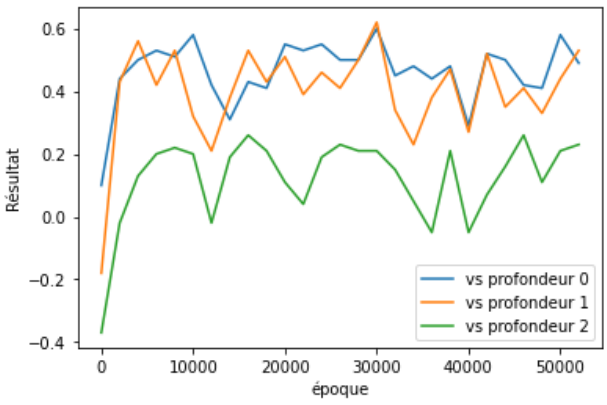

FIGURE 14 – Résultat pour le DQL 4x4 : Joueur bleu contre différentes IAs minmax

En 4x4 ou en 6x7 lorsque l'entraînement se fait seul le résultat est plutôt décevant. Les agents fonctionnent bien contre des IAs minmax de profondeur faible, mais lorsque la profondeur est supérieure à 2, peu de progression est constatée.

On voit sur les courbes que la performance en 4x4 évolue rapidement au départ, puis stagne à un niveau assez médiocre. En comparaison avec le QL en 4x4, on obtient un résultat nettement inférieur : 0.6 et 0.2 pour DQL, contre O.8 et 0.4 pour le QL. L'unique avantage est la complexité en mémoire inférieure : chaque réseau prend une place d'environ 300Ko au lieu de 6 Mo pour le Qdic en 4x4.

En 6x7, le résultat est encore pire : l'agent arrive à gagner contre les IAs minmax de profondeur 0 et 1 (résultat environ 0.75), mais aucune progression n'est observée contre une IA minmax profondeur 2 (résultat reste constant à -1).

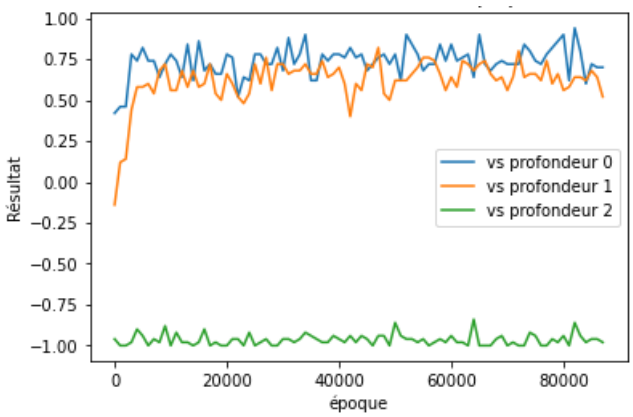

FIGURE 15 – Résultat pour le DQL 6x7 : Joueur rouge contre différentes IAs minmax

Exemple de partie jouée en 4x4 contre une IA minmax :

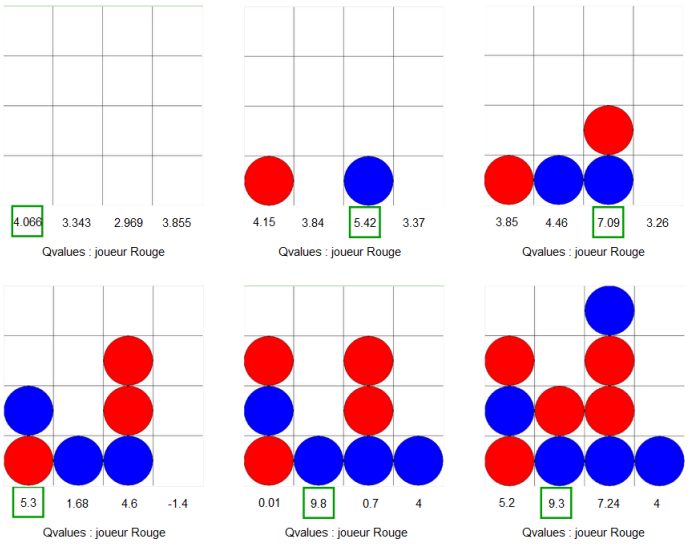

FIGURE 16 – Exemple de partie jouée contre IA minmax (ici une victoire)

III.8.g Entraînement contre une IA minmax

En définissant un adversaire précis qui suit des lois, comme une IA minmax, on retrouve la propriété de Markov : dans un état donné s, pour une action choisie a, $P(s'|s,a)$ est bien définit et constant pour tout s'. L'agent apprend comme si il était seul dans un environnement markovien qui englobe son adversaire.

Deux réseaux sont à nouveau initialisé aléatoirement comme en entraînement seul III.8.f.

En 6x7 on choisit le réseau suivant :
Entrée((6,7,1)), Conv2D(128, (4, 4), "relu"), Aplatir, Dense(64, "tanh"), Dense(128, "relu"), Sortie(7)
En tout : 105,159 paramètres modifiables.

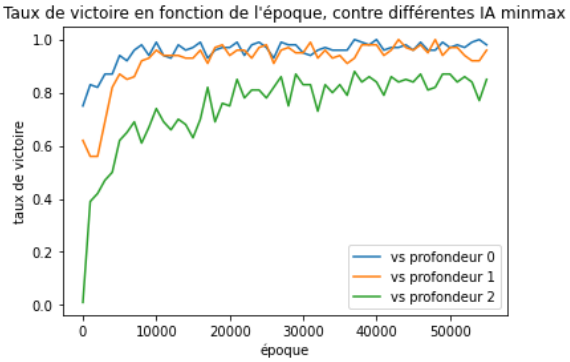

FIGURE 17 – Résultats du DQL 6x7 pour l'agent X

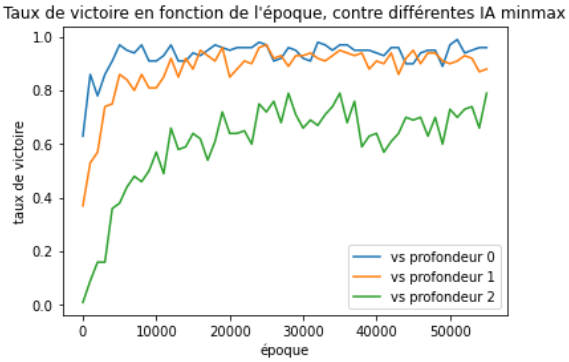

FIGURE 18 – Résultats du QL 6x7 pour l'agent O

On obtient alors de très bons résultats. Ici seul le taux de victoire est représenté, puisque la quantité de match nul est négligeable. Les deux agents aboutissents à des taux de victoire de 100% contre les minmax 0 et 1, et des taux de victoire d'environ 80% contre le minmax profondeur 2. Les agents rouge et bleu obtiennent des résultats similaires à une IA minmax profondeur 5 (voir annexe VII.10). La complexité en mémoire est très faible : chaque réseau est stocké avec 500Ko.

Il est alors intéressant de voir comment nos deux agents (Rouge et Bleu) se débrouillent contre une IA minmax de profondeur 5 :

Pour l'agent Rouge, sur 100 simulations : 80% de victoires, 17% de défaites.

Pour l'agent Bleu, sur 100 simulations : 70% de victoires, 23% de défaites.

Ces résultats sont assez satisfaisants, on remarque que notre agent apprend vite les coups qui favorisent une récompense.

L'agent apprend très vite à prendre le centre et créer des contraintes qui lui permettent de gagner :

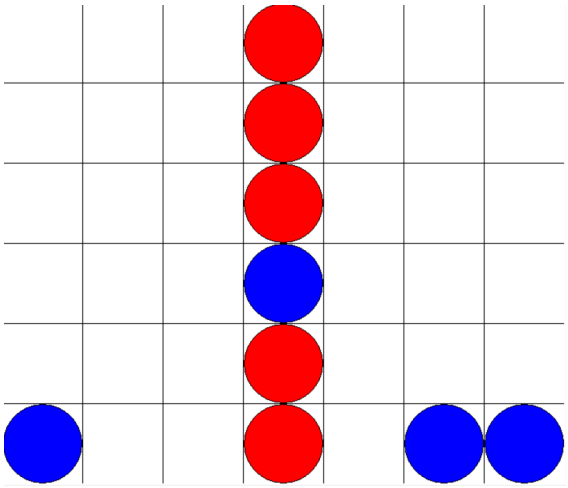

FIGURE 19 – Exemple de début de partie de l'agent rouge contre IA minmax profondeur 5

L'agent exploite le fait que l'agent minmax joue aléatoirement lorsqu'il ne voit pas de chemin gagnant ou perdant. En prenant le centre de cette manière, les bleu n'ont que très peu d'options pour gagner.

La Qvalue menant à l'état de l'image est de 5.52 (récompense pour une victoire = 10), donc l'agent est très confiant.

En observant à présent les Qvalues de l'état dans l'image ci-dessus selon l'agent bleu (c'est au tour des bleux) on voit que l'agent bleu sait qu'il perd dans cette position : [-12.142669 -9.461924 -7.8447905 -9.047617 -8.285687 -11.036969 -11.545659].

Cependant, la perfection est loin d'être atteinte : en effet, l'apprentissage des agents est biaisé, ils apprennent à battre un agent minmax de profondeur 2, ainsi ils n'explorent jamais certaines positions, comme celle ci-dessous :

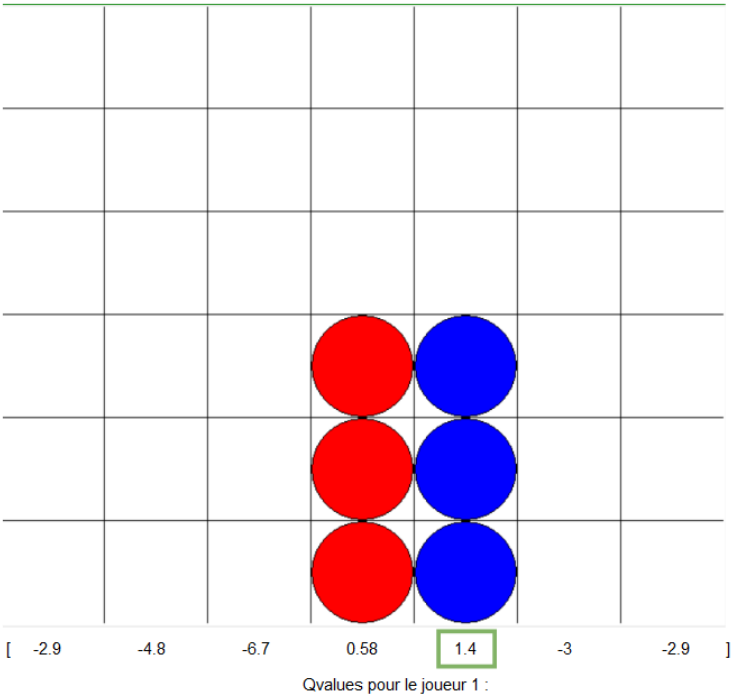

FIGURE 20 – Exemple de biaisage : l'agent rouge préfère bloquer le bleu au lieu de gagner (il n'a jamais vu cette position auparavant)

La position ci-dessus est entièrement gagnante, cependant notre agent ne l'a jamais vue auparavant, et ne voit donc pas la victoire instantanée. En s'entraînant uniquement contre une IA minmax profondeur 2, l'agent ne sait pas réagir lorsqu'un mauvais coup est joué.

III.8.h Analyse du choix des hyperparamètres

La sous partie précédente présente les résultats obtenus pour des choix fixes qui ont été étudiés au préalable.

Plusieurs structure de réseau sont possibles, le plus important étant de veiller à satisfaire 2 conditions : avoir assez de poids modifiables et de couches (selon la taille du jeu), et avoir avec des fonctions d'activations non-linéaires. Si l'une des ces condition n'est pas satisfaite, alors le réseau ne pourra jamais approximer correctement une politique optimale.

Ceci relève de la théorie de comment les réseaux de neurones approximent des fonctions [5]

Pour le puissance4 6x7, un réseau avec 3 couches, une activation "tanh" et 200 000 paramètres permet d'approximer des fonctions assez complexes pour jouer de façon cohérente.

— Choix de la taille mémoire de 10000 : dans le DQL, des réseaux de neurones sont

utilisés, ce type de modèle nécessite énormément de données pour apprendre. Ainsi utiliser un algorithme similaire au QL sans mémoire (entraînement unique après chaque coup) fournit des résultats incohérents.

— Choix du batch size de 32 : pour estimer le gradient, 32 échantillons sont utilisés. Avec un batch size de 1 le gradient est mal estimé et les résultats de l'agent oscillent sans tendances positives.

— Choix de 256 entraînements après chaque partie : ce choix est assez arbitraire, il s'agit simplement d'entraîner suffisamment le réseau (100, 300 ou 500 peuvent aussi fonctionner)

— Choix du pas d'apprentissage de 0.01 (pour le réseau) : un pas trop faible induit un apprentissage trop lent, un pas trop élevée induit une divergence. Empiriquement, un pas de 0.01 fournit de bons résultats.

— Choix du facteur d'exploration ϵ (fonction exponentiellement décroissante variant entre 0.9 et 0.1) : si l'agent n'explore pas suffisamment il bloque sur des politiques médiocres. Si l'agent explore trop, il aura une mauvaise représentation du jeu et n'explorera jamais jusqu'au bout un bon enchaînement de coups. Empiriquement, une exploration constante de 0.1 est un bon équilibre et une exploration décroissante permet de converger plus rapidement.

— Choix du facteur d'atténuation $\gamma \approx 0.95$: empiriquement, le choix de ce facteur pour le puissance4 n'a pas énormément d'importance. Il sert à atténuer les récompenses futures. En choisissant $\gamma \in [0.8, 1]$ des bons résultats sont obtenus. En effet une partie de puissance4 dure au maximum $h \times w$ coups, ce qui est relativement faible. Le facteur γ devient plus important quand une partie contient une grande quantité d'états (ex : $\approx 10^5$ pour le Breakout).

— Choix de la fonction erreur et de l'optimisateur : en essayant plusieurs fonctions pertes (MSE, RMSE, Huber, ...) et deux optimisateurs différents (SGD et Adam), la performance reste plus ou moins la même. Donc MSE + SGD a été globalement choisi, c'est l'erreur et l'optimisateur les plus classiques.

Rq : la mesure de l'impact des hyperparamètres est très coûteuse en temps (un entraînement complet de 50 000 parties, avec les tests de performances dure au moins 1 heure). De plus l'impact d'un hyperparamètre seul dépend des autres hyperparamètres (par exemple le pas d'apprentissage optimal dépend de la fonction perte et des fonctions d'activation). Ainsi seule une étude qualitative a pu être faite, guidée par la recherche théorique.

III.9 Recherche arborescente Monte-Carlo (MCTS)

Les deux approches précédentes (QLearning, Deep QLearning) sont des méthodes d'apprentissage "model free", c'est à dire qu'ils n'exploitent aucune propriétés de l'environnement. Ce type d'apprentissage est facilement adaptable à tout environnement, les formules ne varient pas.
Cependant, pour les jeux déterministes comme le puissance4, utiliser une méthode d'apprentissage "model based" est très prometteur.

Pour les jeux déterministes 1 contre 1, un algorithme très rependu est la rechercher arborescente Monte-Carlo. Utilisé par Deepmind pour maîtriser les échecs, le Go ou encore le Shogi [10].

Ainsi, afin d'exploiter la nature déterministe du jeu du puissance 4, une approche entièrement différente est adoptée.

III.9.a MCTS principe général

L'algorithme de la recherche arborescente Monte Carlo consiste à parcourir l'arbre du jeu, en parcourant les états de l'arbre les plus fructueux et les moins explorés. Ce parcours permet d'équilibrer l'exploitation et l'exploration.

L'algorithme permettant d'effectuer ce parcours d'arbre est présenté en annexe : VII.8.

La sélection de l'état à explorer se fait en choisissant le noeud qui maximise l'UCT (Upper Confidence bound applied to Trees) :

$$UCT = \frac{\pm valeur}{visites} + c\sqrt{\frac{ln(visites\_parent)}{visites}}$$

Ainsi, plus le gain moyen d'un noeud est élevé et moins il a été visité, plus il a de chance d'être choisi.

(Remarque : la valeur à maximiser est -valeur si c'est au tour des bleus)

Ensuite la simulation se fait classiquement en jouant des coups aléatoires, mais peut se faire en suivant une politique quelconques (minmax..). Cette simulation permet d'attribuer une valeur à un noeud.

III.10 MCTS adapté au Deep Reinforcement Learning

Cette section s'inspire du travail de Deepmind sur le jeu de Go [10]. La méthode utilisée est similaire avec beaucoup de simplifications (réseau de neurones simplifié,

aucun batch normalisation, aucune selection naturelle...).

III.10.a Réseau de Politique et Réseau de valeur

Deux réseaux de neurones sont créés : le "policy net" et le "value net".
Ces réseaux prennent en entrée l'état du jeu sous forme de bitmap (cube $h \times w \times 3$ avec une couche précisant le tour).

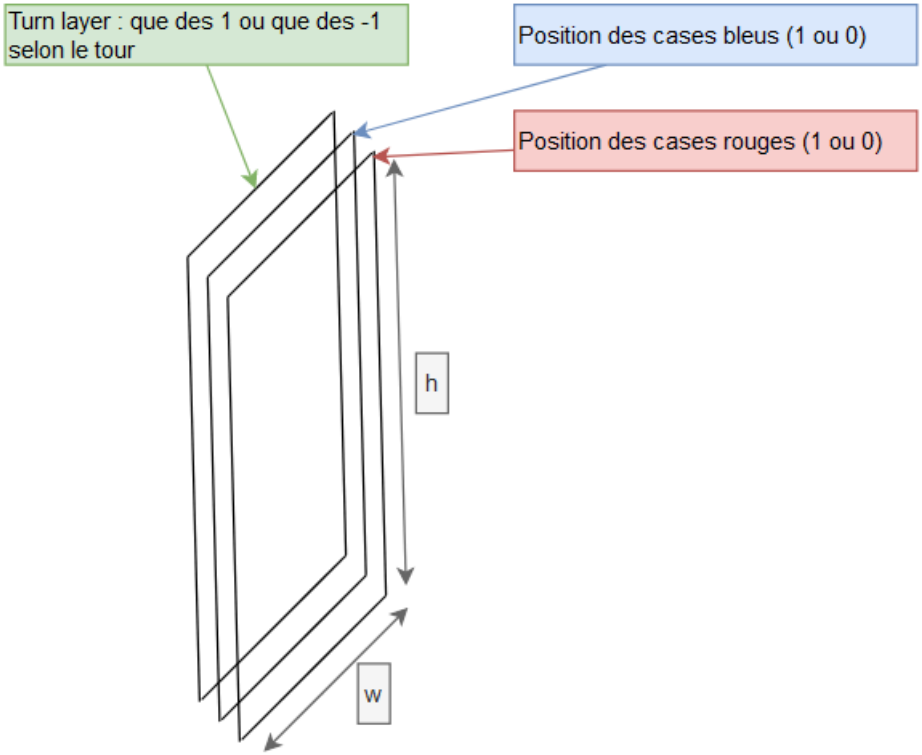

FIGURE 21 – Représentation d'un état pour les Policy et Value networks

Le Policy Network (Réseau Politique) prend en entrée l'état du jeu (bitmap), et renvoie une distribution de probabilités sur $[|0, w-1|]$, soit un vecteur de taille w, à valeurs dans $[0,1]$, telle que la somme est égale à 1. Cette distribution représente la probabilité de choisir un coup, p_0 étant la probabilité de jouer à gauche.
Pour se faire concrètement, la fonction softmax est utilisée comme activation en sortie du réseau. Cette fonction transforme un vecteur réel quelconque en vecteur probabilité. Plus un neurone a une valeur grande relativement aux autres, plus sa probabilité sera élevée.

$$softmax(x_i) = \frac{e^{x_i}}{\sum e^{x_j}} \in [0,1]$$

Le Value Network (Réseau Valeur) prend en entrée l'état du jeu et renvoie une valeur unique dans $[-1,1]$. Cette valeur représente une prédiction sur le résultat de la partie,

si elle vaut 1 alors le réseau pense que les rouges vont gagner, si elle vaut -1 le réseau pense que les bleus vont gagner.
Pour se faire concrètement, la fonction tangente hyperbolique est utilisée en sortie du réseau. Plus un réel x est négatif, plus tanh(x) est proche de -1.

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \in [-1, 1]$$

Rq : il n'est pas obligatoire de recaler la valeur dans $[-1, 1]$, la valeur serait alors un réel quelconque, ceci ne change rien à l'algorithme. Cependant borner la valeur par $[-1, 1]$ permet d'éviter une saturation (excès de la valeur de l'encodage float64).

Ensuite après plusieurs tests, le choix qui fournit les meilleurs résultats pour le puissance4 6x7 est le suivant :

Le Policy Netork a la structure suivante :

- Réseau : Entrée((6,7,3)),Conv2D(512, (4,4),"elu"), Dense(128,"tanh"), Dense(128, "lineaire"),Dense(128, "elu"), Sortie((7,), "softmax")
- Total : 845,575 poids ajustables
- Perte : cross entropy expliquée en annexe VII.5.a, c'est une erreur efficace lorsque la sortie du réseau représente des probabilités
- Optimisateur : SGD (Stochastic Gradient Descent)

Le Value Network a la structure suivante :

- Réseau : Entrée((6,7,3)),Conv2D(512, (4,4),"selu"), Dense(128,"tanh"), Dense(128, "linear"), Dense(128, "elu"), Sortie((1,), "tanh")
- Total : 844,801 poids ajustables
- Perte : MSE (Mean Squared Error)
- Optimisateur : SGD (Stochastic Gradient Descent)

La convolution (4,4) fournit de loin les meilleurs résultats. Ensuite un réseau assez profond sans trop de neurones est nécessaire (ici 4 couches intermédiaires). En effet les mêmes réseaux avec des couches Denses à 512 neurones n'arrivent pas à apprendre correctement.

Les réseaux choisis ci-dessus représentent un bon équilibre empiriquement.

Rq : les coups illégaux sont supprimés, càd que les probabilités données aux actions non autorisées ne sont pas prises en compte.

Policy Network

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 3, 4, 512)         25088
_________________________________________________________________
flatten (Flatten)            (None, 6144)              0
_________________________________________________________________
dense (Dense)                (None, 128)               786560
_________________________________________________________________
dense_1 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_3 (Dense)              (None, 7)                 903
=================================================================
Total params: 845,575
Trainable params: 845,575
Non-trainable params: 0
```

Value network

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 3, 4, 512)         25088
_________________________________________________________________
flatten_1 (Flatten)          (None, 6144)              0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               786560
_________________________________________________________________
dense_5 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_6 (Dense)              (None, 128)               16512
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 129
=================================================================
Total params: 844,801
Trainable params: 844,801
Non-trainable params: 0
```

FIGURE 22 – Résumé des réseaux de Politique et de Valeur (module tensorflow.keras)

III.10.b Déroulement de l'entraînement

Initialement, les réseaux fournissent des sorties aléatoires ne représentant rien.

L'agent joue contre lui-même, et à chaque fois qu'un état est rencontré, l'algorithme ci-dessous est enclenché :

L'arbre est constitué de noeuds avec les attributs suivants :

- État du noeud
- Parents = État menant au noeud
- Fils = États accessibles depuis le noeud
- Probabilité : probabilité indiqué par le policy net pour le coup menant au noeud, reste constant.
- Valeur accumulée : initialement la valeur indiqué par le value net pour l'état du noeud
- Visites : initialement 1 (valeur donné initialement compte comme une visite)

Entrée : un état du jeu Initialisation : création d'un Noeud* contenant la valeur de l'état considéré
pour $i = 1, 2, ..., N_{itrations}$ **faire**
    Sélection : on part de la racine, et on sélectionne le fils qui maximise un terme spécifique, jusqu'à un noeud qui peut être développé (expanded).
    Expansion : On choisit un coup aléatoire, on le joue et on créé le nouveau noeud contenant l'état obtenu. Le Value net attribue une valeur à celui ci.
    Retropropagation : On incrémente le nombre de visite (+1) et la valeur accumulée du nouvel état et tous ses parents.
fin

Algorithme 4 : Algorithme MCTS adapté au Deep Learning

Le point à détailler est la sélection. Afin de trouver le meilleur compromis entre exploration et exploitation, la quantité à maximiser pour la sélection est similaire au UCT classique vu en partie III.9.a :

$$UCT_{DeepL}(s,a) = \frac{\pm valeur\_cumule}{visites} + c^{ste} \times P(s,a)\frac{\sqrt{visites\_parent}}{visites}$$

Le Policy Network permet de guider l'apprentissage, en incitant les visites d'actions les plus probables, et le value net permet d'éviter l'étape de simulation et attribue une valeur souvent plus fiable.

Les illustrations ci-dessous permettent de comprendre en détail l'algorithme :

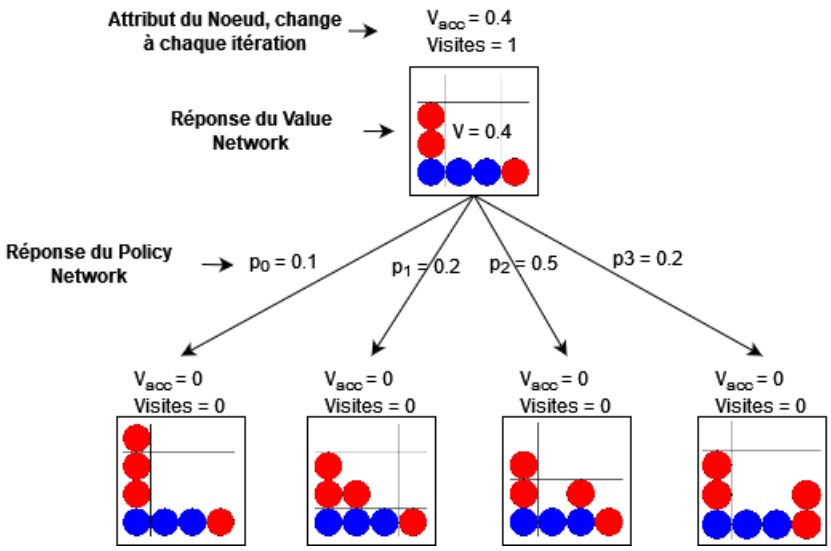

FIGURE 23 – Initialisation

Initialisation : l'état à étudier est mis à la racine de l'arbre, et tous ses fils sont initialisés. Les Policy et Value networks sont utilisés pour attribuer une première valeur à l'état initial et aux probabilités des actions à prendre.

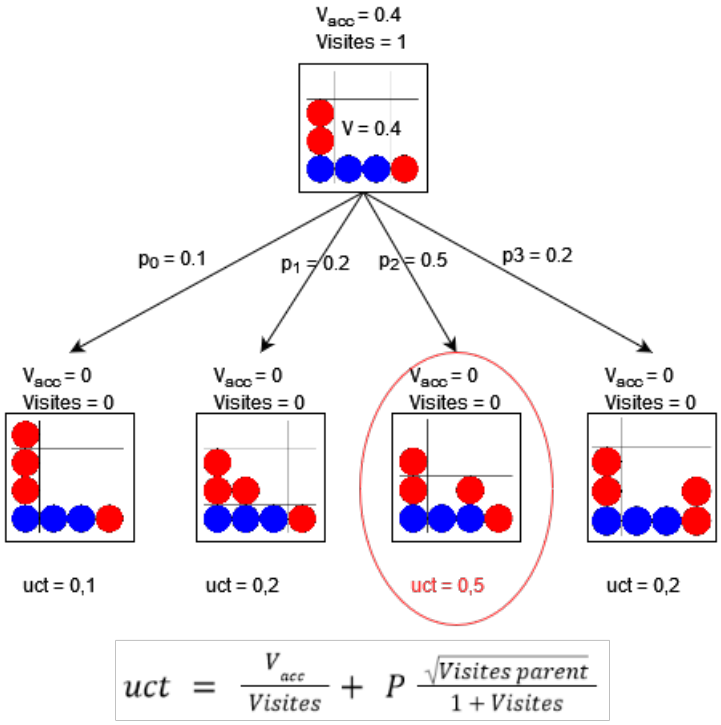

FIGURE 24 – Sélection

Sélection : on part de la racine et on descend jusqu'à tomber sur un noeud jamais visité ou terminal. La descente se fait en maximisant la quantité UCT.
Remarque : l'UCT utilise l'opposé de la valeur accumulée lorsque c'est au tour des bleus, les bleus cherchant à minimiser cette valeur.

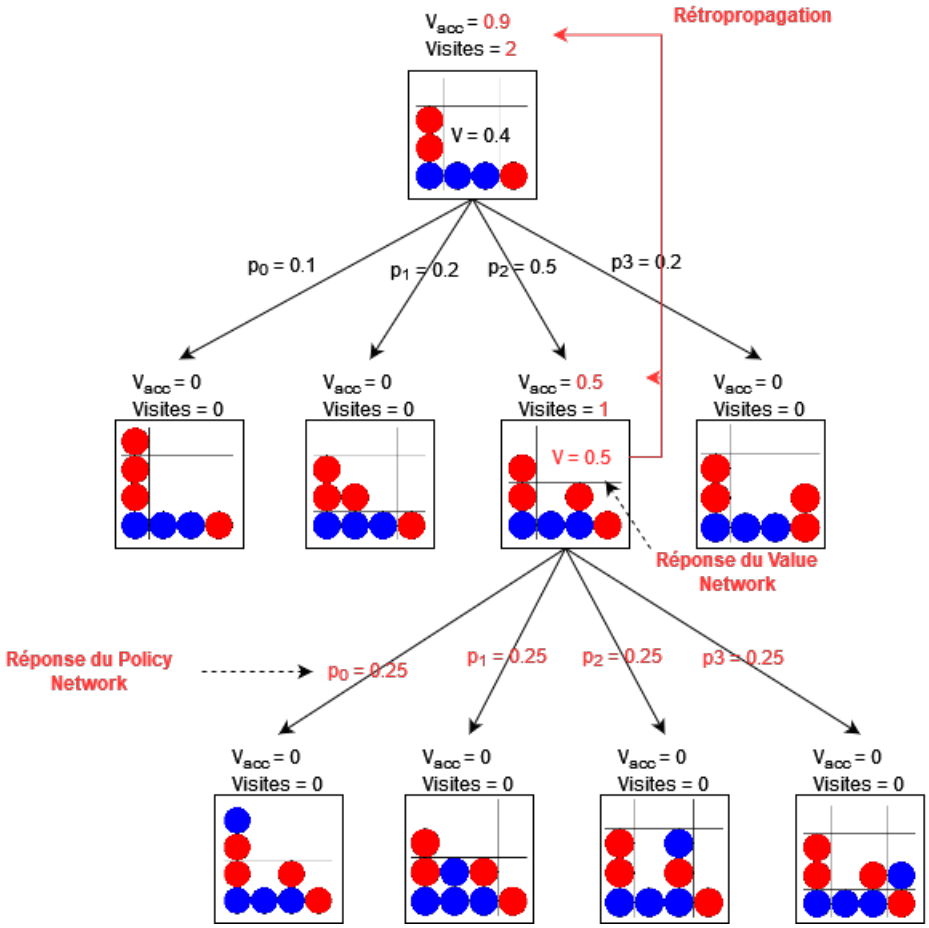

FIGURE 25 – Expansion et mise à jour

Expansion et mise à jour : Les Policy et Value networks sont utilisés à nouveau pour donner une valeur à l'état sélectionné et attribuer les probabilités des action. Ensuite les nouveaux états accessibles sont initialisés. Puis la valeur de l'état sélectionné permet de mettre à jour les noeuds parents.

Les 2 étapes précédentes correspondent à une itération, le même processus est répété un grand nombre de fois (autour de 100 pour l'algorithme implémenté ici) : la deuxième itération est illustrée ci-dessous :

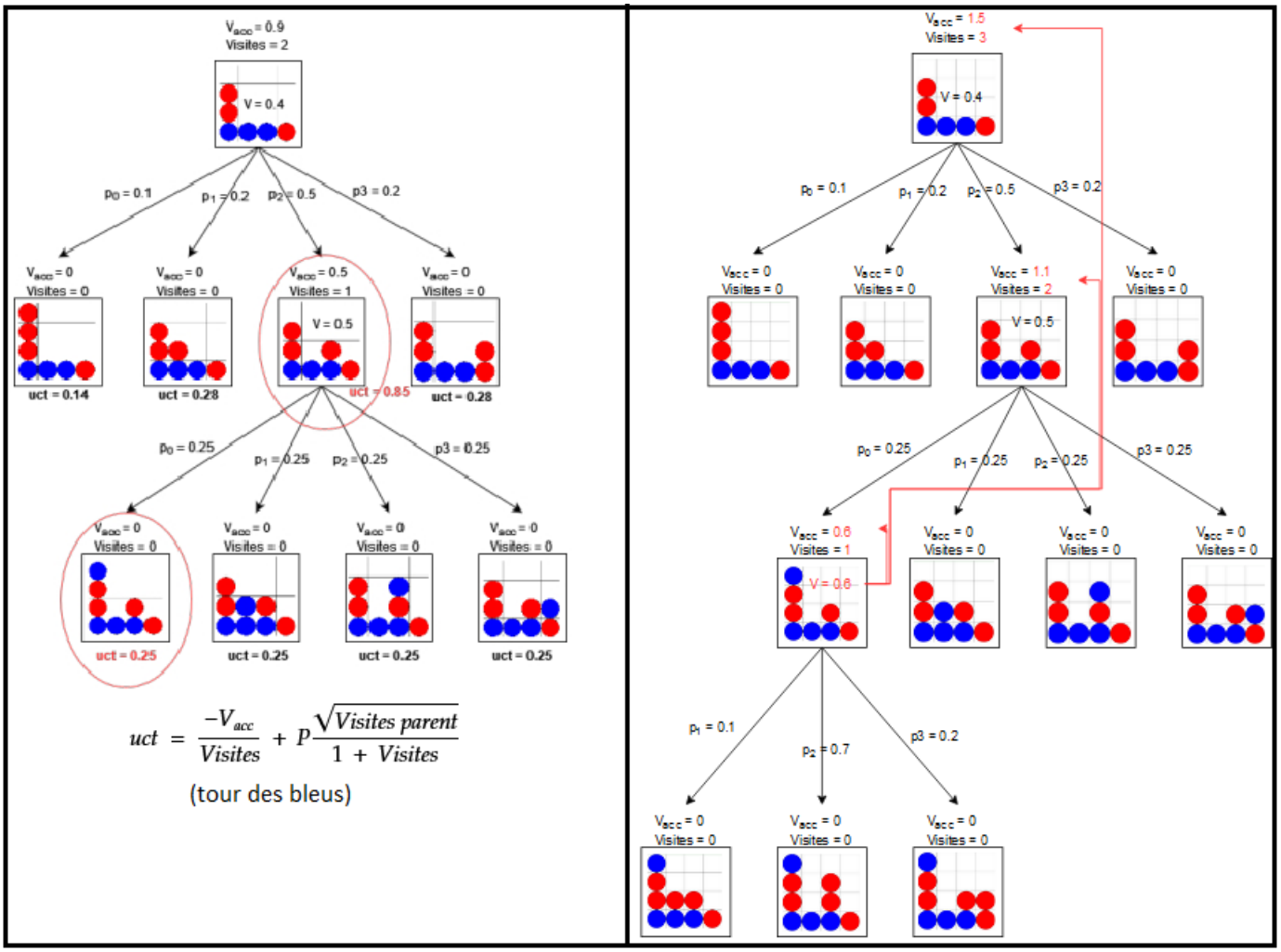

FIGURE 26 – Itération 2 : Sélection, Expansion et Mise à jour

Après suffisamment d'itérations (100-1000), de nouvelles probabilités d'action et une nouvelle valeur est obtenue.

Les réseaux sont alors entraînés sur ces données.

III.10.c Choix d'une action

Lorsque l'agent n'est pas en mode entraînement, le choix d'une action après itération sur un état peut se faire de plusieurs manières :

— Vers le noeud le plus visité
— Vers le noeud avec la plus grande valeur moyenne
— Maximiser une autre quantité cohérente (ex : LCB)

Le choix de la plus grande valeur moyenne a été adopté.

Lorsque l'agent s'entraîne, il choisit une action différemment afin de favoriser l'exploration. Il choisit son action selon une distribution de probabilité :

$$P(action\ i) = \frac{Visites\ noeud\ i}{Visites\ totales}$$

III.10.d Résultat final

Entraîner et tester l'agent est très coûteux en temps, donc aucune courbe de progression n'a été faite. Pour le Deep QLearning, jouer un coup prend peu de temps, il suffit de passer l'état du jeu dans le réseau et de choisir l'action avec le plus de valeur. Pour l'agent MCTS, avant de jouer un coup l'agent effectue 100 (ou plus) itérations en parcourant l'arbre, prenant ainsi 100x plus de temps que le DQN simple.

De ce fait, seul le résultat final sera présenté (par manque de temps). Tout d'abord, afin de choisir la bonne structure de réseau et les bons hyperparamètres, je commence par faire des mini entraînements, et j'observe la progression réalisée.

Ensuite, une fois une structure et des hyperparamètres cohérents, le réseau s'entraîne tout d'abord en utilisant une simulation aléatoire pour attribuer une valeur à un noeud (au lieu d'attribuer la valeur d'un noeud avec le réseau, une partie aléatoire est simulée et la valeur sera le résultat de la partie (0,1 ou -1)). Ceci a pour but d'accélérer l'apprentissage initialement, et de raccourcir l'entraînement. Ensuite, petit à petit, l'étape de simulation est supprimée, et le réseau s'entraîne normalement comme expliqué en partie III.10.

Après 5 jours d'entraînement et 500 000 coups joués, l'agent atteint enfin un niveau de performance très satisfaisant :

Sur 100 tests à chaque fois (pour les deux couleurs) :

— Minmax profondeur 2 : 100% de victoire.

— Minmax profondeur 5 : 98% de victoire, 1% défaite, 1% égalité.

L'agent joue presque parfaitement. Il peut être testé avec les fichier du github : `https://github.com/roland-robert/puissance4_RL`

IV Jeux à nombre d'états infinis[7]

IV.1 Introduction

Dans cette partie, on s'intéresse à des jeux à nombres d'états infinis, tels le Pong et le Breakout. On peut définir un état par l'ensemble des pixels d'une image. En tout rigueur, ce nombre d'état n'est donc pas infini, mais égal à la surface en $pixel^2$ de jeu. Cependant, notre système d'apprentissage ne pourra pas prendre en compte l'ensemble des pixels à chaque instant. Il faudra donc réaliser des transformations pour se ramener à des états plus compacts. Ces transformations utilisent des réseaux de neurones convolutifs $^{VII.3}$, grandement inspirés du traitement des images par le cerveau humain.

IV.2 Principe de la méthode

IV.2.a Alorithme utilisé

L'apprentissage du Breakout se fera avec l'algorithme du Deep Q-Learning (DQL), qui a été utilisé dans le cadre des jeux à états discrets. Il y a donc de nombreuses similarités dans la mise en application de la méthode. Le DQL est l'algorithme de base de l'apprentissage par renforcement profond, qui a été utilisé pour apprendre certains jeux à états finis dans la partie précédente. L'algorithme SARSA est inutilisable pour apprendre les jeux Atari, car ils sont trop complexes[7].

IV.2.b Exploration et apprentissage

Cette fois-ci, le nombre d'états est infini, et il faut un nombre plus important d'épisodes pour que l'agent puisse apprendre à jouer correctement. Le nombre exact varie selon les jeux, mais en général un jeu Atari nécessite plus d'un million d'épisodes. Il faudra donc une phase d'exploration plus importante que pour les jeux à états finis. On prendra donc un *epsilon_decay* plus important, égal à 10^6, afin de ralonger la phase d'exploration. Voici en figure suivante le graphe de epsilon en fonction du numéro de l'épisode :

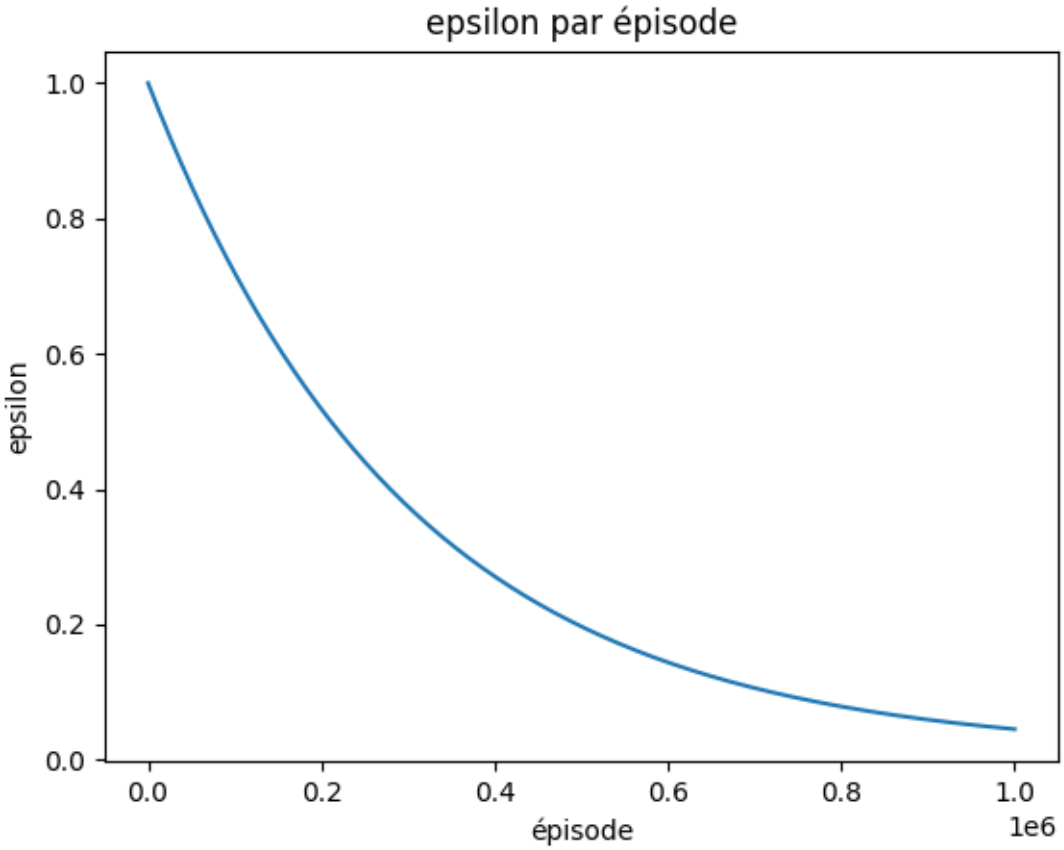

FIGURE 27 – Epsilon par épisode pour les jeux Atari

IV.3 Processus de décision markovien partiellement observable

Bien qu'un environnement à états continus puisse être discrétisé afin de permettre à l'agent d'intéragir avec, cette méthode est coûteuse en calculs et ne permet pas de gérer des environnements avec énormément d'état. On introduit donc le concept de processus de décision markovien partiellement observable : l'agent n'observera que certaines caractéristiques de l'environnement. C'est exactement le type d'intéraction qu'un humain a avec son environnement. Par exemple lorqu'un humain lit, son attention n'est donnée qu'à la page qu'il lit, et il n'observe pas les autres événements qui se produisent dans son champ vision. En fait, il n'effectue qu'une observation partielle de l'univers, qui est le processus de décision markovien global dans cette modélisation. La capacité de jongler entre différentes observations partielles d'un même processus de décision markovien infini est une pierre angulaire de la recherche actuelle. Deux branches du DRL tentent de résoudre ce problème : l'apprentissage par renforcement hiérarchique et le *Meta Reinforcement Learning*. Ce dernier introduit le temps dans l'apprentissage, et a recours a des réseaux de neurones récurrents, qui apprennent en tenant compte de la variable temporelle.

IV.4 Arcade Learning Environment (ALE) [8]

Afin de pouvoir tester des algorithmes d'apprentissage par renforcement de manière standardisée, la communauté scientifique a mis un point l'**Arcade Learning Environment (ALE)**. Cet environnement est une plateforme d'évaluation qui regroupe des dizaines de jeux Atari 2600.
Dans le cadre de ce PAr, nous nous intéressons en particulier à Breakout. Les jeux ont été développés de manière standardisée pour faciliter leur utilisation par les chercheurs, et sont disponibles avec le module python **Gym**. Nous utilisons donc ce module.
Dans le cas du jeu Breakout, une époque correspond à 5 parties. La récompense vaut 1 si l'agent détruit une brique, 0 sinon. Chaque époque est discrétisée en épisodes, de sorte qu'il y a en moyenne une centaine d'épisodes par époque. Ces données sont visibles dans le code du jeu, disponible en *open-source*.

FIGURE 28 – Jeu Breakout issu du module Gym

IV.5 Outils utilisés

De nombreuses bibliothèques sont utilisées dans le monde scientifique et de l'industrie pour implémenter des réseaux de neurones. Bien que nous pourrions coder ces réseaux nous-même, de nombreuses optimisations sont effectuées pour réduire le temps de calcul, et l'étude complète de la création d'un réseau de neurones pourrait faire l'objet d'un PAr à part entière. Les trois bibliothèques principales sont Keras, TensorFlow et PyTorch. Keras est la plus simple d'utilisation et sa popularité est en décroissance permanente. TensorFlow est la plus populaire et considérée comme à la pointe de la technologie, mais n'offre que peu de contrôle : c'est une bibliothèque haut

niveau. PyTorch est de plus en plus populaire, notamment dans les applications de *Deep Reinforcement Learning*, et offre un bon contrôle. Cette flexibilité est un avantage de taille qui va probablement faire d'elle la bibliothèque la plus populaire pour l'apprentissage par renforcement profond au cours des prochaines années. Notre choix s'est donc tourné vers PyTorch. Le réseau de neurones peut être construit à bas niveau en spécifiant explicitement les formules vectorielles pour le calcul de l'erreur et en faisant appel à des opérateurs de différentiation automatiques. PyTorch met à disposition des outils pour créer un graphe de calcul, qui représente le réseau de neurones. Cette approche a été utilisée pour se familiariser avec PyTorch et les réseaux de neurones, mais nous avons continué en utilisant des outils plus hauts niveaux, qui automatisent certaines étapes comme la définition vectorielle de la fonction de coût[7]. Ces outils restent plus bas niveau que ceux proposés par TensorFlow.

IV.6 Représentation des états

L'état du jeu est la matrice des pixels du jeu. Il faut donc déterminer une manière d'encoder cet état afin qu'il puisse être utilisé et interprété par notre agent. La méthode utilisée repose sur des réseaux convolutifs$^{VII.3}$, et a été mise en place au cours des 10 dernières années, après une trentaine année de recherche. Les pixels de l'image sont transmis à un réseau de convolution profond, qui apprend lui même à encoder ces états, au fil des entraînements.

IV.7 Batch Learning

Afin d'avoir un apprentissage qui sait gérer un grand nombre de situations et pas seulement quelques exemples, une technique classique en Deep Learning est le **Batch Learning**. Cette technique consiste à stocker les résultats d'un grand nombre d'épisodes, et d'attendre un certain nombre d'épisodes avant de calculer le coût de chaque décision et d'appliquer la rétropropagation du gradient. La mise à jour des poids est alors une moyenne des mises à jour sur chaque épisode.
Cette technique nécessite de stocker en mémoire de nombreux épisodes. Dans le cas de DQL, ces épisodes sont décrits par :

— L'état : entrée du réseau de neurones

— Le prochain état : renvoyé par l'environnement en fonction de l'action

— L'action : renvoyée par le réseau de neurones

— La récompense : renvoyée par l'environnement en fonction de l'action

— Done : booléen pour savoir si la partie est finie

Un objet spécifique ReplayBuffer a été créé afin de gérer tous ces exemples. La classe *deque*[3] du module *collections* a été utilisée pour stocker ces épisodes. Cette classe permet de stocker tout type d'objet, en particulier les tuples qui contiennent les différents paramètres des épisodes ici. On initialise cet objet en spécifiant la **capacité** du Buffer, i.e. le nombre d'exemples qu'il peut contenir. Ce paramètre est limité par les capacités mémoire de la machine.
Une méthode importante de la classe ReplayBuffer est la méthode **sample**. Elle permet de sélectionner aléatoirement un nombre *batch_size* d'exemples, parmi tous ceux qu'elle contient. Le nombre *batch_size* est un hyperparamètre important, car il détermine le degré de moyennage utilisé pour mettre à jour les poids du réseau de neurones. Dans le cas d'un jeu Atari, l'apprentissage nécessite un nombre très important d'exemples. Le replay buffer peut contenir jusqu'à plusieurs centaines de milliers d'épisodes. Voici en figures suivantes des exemples d'apprentissage d'un jeu très simple fourni par Gym, CartPole[2], en utilisant le batch_learning, pour différentes valeurs de batch_size et capacité. Le graphe de gauche représente la récompense moyenne au cours d'une époque, i.e. la somme des récompenses des épisodes d'une époque. A droite, on peut voir le coût de chaque action, épisode par épisode.

Les pics qu'on observe dans le coût des épisodes correspondent à de mauvaises actions effectuées par l'agent. Une fois qu'il les a effectuées, il ne commet plus ces erreurs, et les pics se font plus rares.

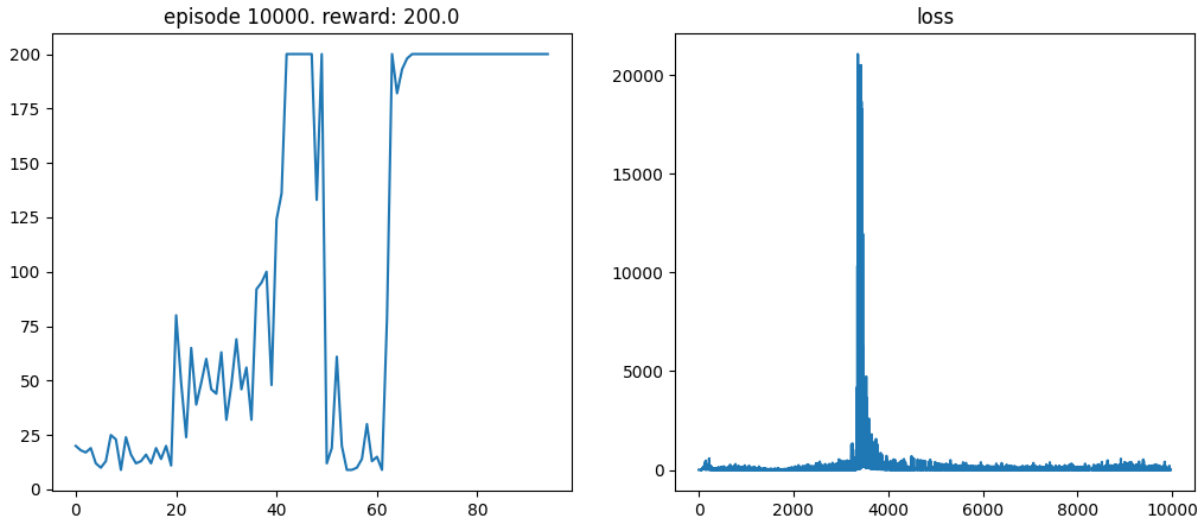

FIGURE 29 – Buffer de batch_size 32, et capacité 32. La moyennisation n'a aucun effet

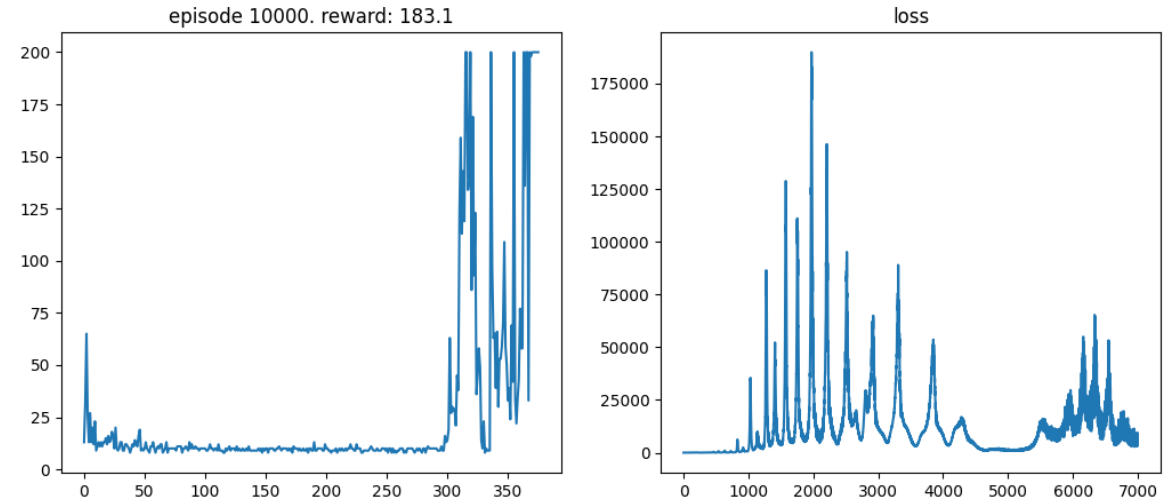

FIGURE 30 – Buffer de batch_size 3000 et capacité 10000. L'apprentissage est effectif mais plus lent, car à partir de 3 000 épisodes, 3000 rétropropagations du gradient sont effectuées à chaque épisode.

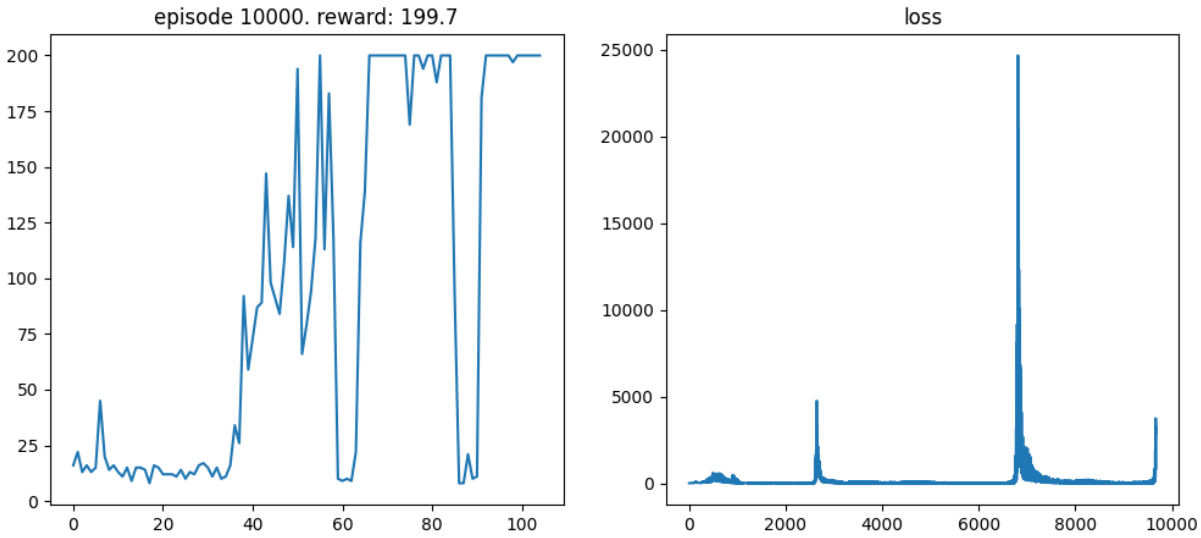

FIGURE 31 – Buffer batch_size 32 et capacité 333. La moyennisation a un bon effet lorsque la taille des batchs est environ 10 fois la capacité du buffer

IV.8 Paramètres de l'algorithme

Voici les paramètres utilisés pour l'algorithme du DQL appliqué au jeu Breakout :

episodes	batch_size	γ	buffer_size	hidden layers	activation	epsilon_decay
1400000	32	0.99	100000	3	ReLu	$1e^6$

La taille du buffer est très importante (100 000), afin de pouvoir entraîner le réseau sur de nombreuses situations. Chaque batch d'entraînement contiendra les résultats de 32 épisodes, choisis dans les 100 000 derniers résultats contenus dans le buffer. Il y a donc 32 entraînements effectués à la fin de chaque épisode.
Les entraînements sont effectués lorsque le buffer atteint une taille de 10 000, afin d'avoir une bonne diversité de situations dès le début de l'entraînement. Sur 1400000 épisodes, il y a donc 44 480 000 rétropropagations du gradient effectuées au total. Cela explique la durée très longue de l'entraînement (> 3 jours) .

IV.9 Résultats

Le réseau de neurones met plusieurs jours pour effectuer les 1400000 épisodes et s'entraîner. On observe une progression lente des récompenses obtenues par le réseau, qui justifie l'emploie d'un grand nombre d'épisodes. Au début, le coût des actions est souvent très élevé, et monte jusqu'à 1e6 : le réseau explore son environnement, et est amené à effectuer certaines des pires actions en terme de coût. Après les avoir rencontrées, il apprend à les éviter, et le coût ne remonte plus à 1e6.

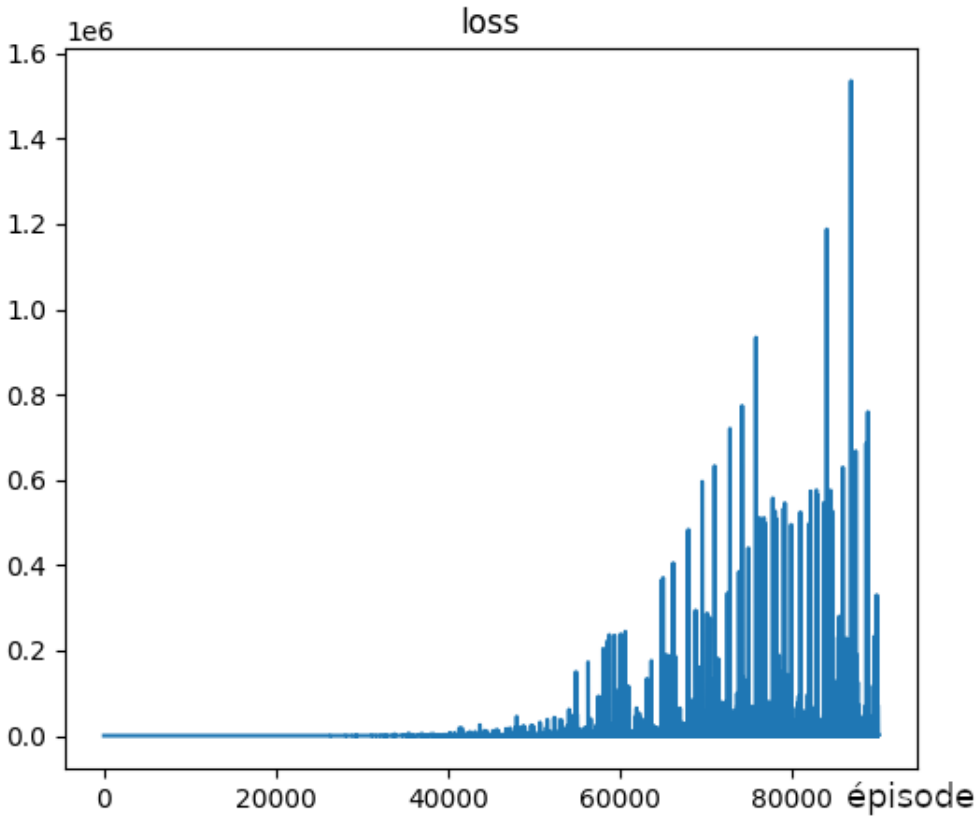

FIGURE 32 – Coût en fonction du nombre d'épisode pour le jeu Breakout, pour les 100 000 premiers épisodes

On remarque sur la figure 32 une forte augmentation du coût de chaque action, qui atteint un maximum d'environ 10^6 vers 100 000 épisodes.

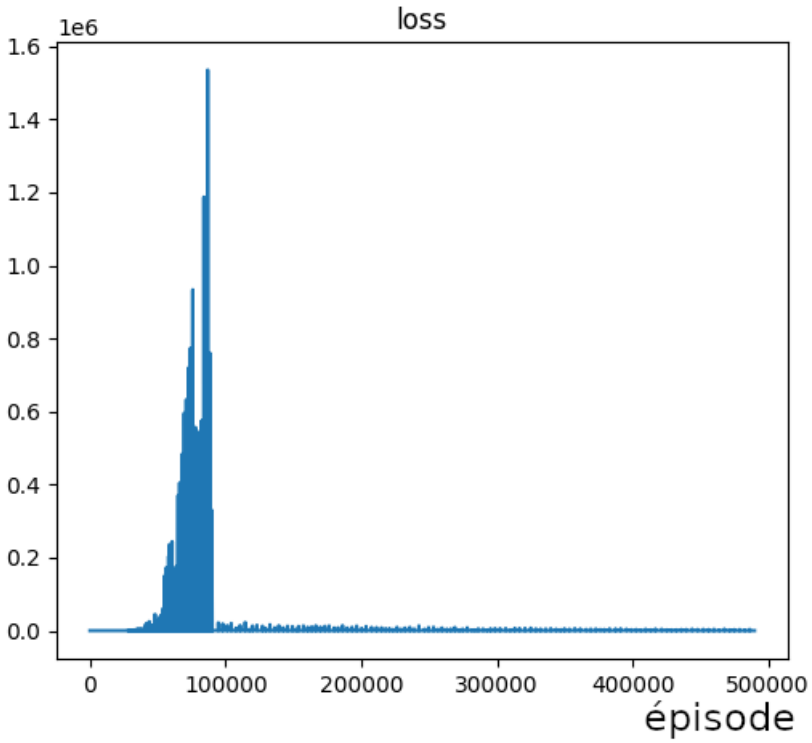

FIGURE 33 – Coût en fonction du nombre d'épisode pour le jeu Breakout, pour les 100 000 premiers épisodes

Sur la figure 33, on voit que le coût de chaque épisode diminue très fortement et a un ordre de grandeur de 1e4. Il diminue ensuite au fur et à meusre des épisodes.
Au bout d'un million d'épisodes, et au moins deux jours d'entraînement, l'agent prend la stratégie de se mettre dans un coin. Cette position lui permet de renvoyer les balles qui arrivent en diagonale de manière à ce qu'elles reviennent vers lui. Une fois les premières briques cassées, la balle passe par-dessus le mur de brique et offre de nombreuses récompenses à l'agent car de nombreuses briques sont ainsi brisées. Cependant, l'agent n'est pas encore capable de bien bouger pour récupérer une balle. Lorsque la balle arrive vers le sol, il se déplace mais pas forcément au bon endroit. Il faut attendre quelques millions d'épisodes de plus pour que ce soit le cas.

V Connaissances acquises

Voici une liste non exhaustive des connaissances acquises durant ce PAr :

1. Linux - le téléchargement de nouvelles bibliothèques est plus simple avec les commandes Linux, qui est plus rapide qui Windows. Un dual-boot Linux/Windows a été mis en place.
2. Théorie du Machine Learning et Reinforcement Learning
3. Création d'environment virtuels afin de garantir une certaine version de python.
4. Utilisation de Visual Studo (VS) Code pour les projets python .
5. Utilisation de la fonction *built-in* zip pour transformer une liste de tuple en un tuple de liste, le tout s'effectuant avec calcul en parallèle.
6. Utilisation du GPU pour améliorer la rapidité des réseaux de neurones et leur apprentissage.
7. PyTorch : création et utilisation de réseaux de neurones
8. Keras : module de création, utilisation et optimisation de réseaux de neurones.
9. Parcours d'arbre, utilisé dans la partie MCTS.

VI Conclusion

Dans ce PAr, on a exploré les différentes manières d'apprendre à un ordinateur à jouer à des jeux vidéos. Les algorithmes de la recherche actuels diffèrent selon le type de jeu : le Q Learning est très efficace sur des jeux avec peu d'états (puissance4 en 4x4), le Deep Q Learning remédie aux problèmes liés à de nombreux d'états en introduisant un réseau de neurones. Le Deep Q Learning permet d'apprendre à maximiser les récompenses d'un agent dans un environnement markovien comme le Breakout.

Le DQL permet de converger vers une politique optimale, mais le nombre d'épisodes nécessaires est souvent très important (notamment pour le Breakout avec une quantité d'états presque infini), et l'apprentissage prend plusieurs jours.

Pour le puissance4 lorsque l'environnement contient un autre agent, un simple apprentissage seul avec le DQL échoue. Pour un jeu déterministe comme le puissance4, un apprentissage utilisant les règles du jeu pour parcourir l'arbre des états guidé par deux réseaux de neurones (MCTS) s'avère être le plus efficace et produit une IA finale compétitive.

Pour toutes ses méthodes d'apprentissage, l'importance du choix des nombreux hyperparamètres (taux d'apprentissage, facteur d'atténuation,..) a été constaté. L'ordinateur apprend à jouer seul, à condition d'être bien paramètré.

Limité par le temps, seul une minorité du domaine du Reinforcement Learning a été exploré.
Avec plus de temps, des méthodes novatrices telles le *Meta Reinforcement Learning*, qui est une des avancées actuelles vers l'intelligence artificielle générale, et de jeux plus complexes non déterministes et 3D, auraient pu être étudiés.

VII Annexes

VII.1 Vocabulaire

MCTS : Monte Carlo Tree Search (Recherche arborescente Monte Carlo), parcours d'arbre utilisé dans la partie puissance4

DQL : Deep Q Learning

QL : Q Learning

Politique : comment un agent joue étant donné un état, c'est à dire la probabilité de jouer une action a dans un état s : $\Pi(a|s)$.

Deep Learning : permettre à un ordinateur d'apprendre à partir d'expériences et de comprendre le monde sous forme d'une hiérarchie de concepts, définis à partir de relations avec des concepts plus simples.

Knowledge base (learning) : Raisonnement à partir de règles prédéfinies. Ce type d'apprentissage n'a mené à aucune avancée majeure.

Machine Learning : Apprentissage de connaissances en extrayant des connaissances de données brut.
Representation learning : Apprentissage d'une représentation des données, et apprentissage de la représentation elle-même. Application : auto-encodeur, programme capable de coder et de ensuite de décoder les données.
Computational neuroscience : Comprendre le fonctionnement du cerveau à un niveau algorithmique.
Hyperparamètre : paramètre dans un système d'apprentissage non modifié à priori, en opposition avec les paramètres modifiables. Ce sont des paramètres qui doivent être choisi intelligemment.

Mini-lot (Mini-Batch) : petit lot de donnés choisi aléatoirement dans un lot beaucoup plus grand. Sert à approximer certaines valeurs (ex : descente du gradient).

Époque : Représente combien de fois on a parcouru toutes les données, après N étapes de mini batch (N = taille totale/ taille mini batch). En général on arrête l'apprentissage

lorsque il y a convergences du résultat, on sera à l'époque finale prête à être jugée.

Overfitting : quand un model est trop conditionné par ses données d'entrainement et s'adapte mal à des données plus générales

VII.2 Réseau de neurones feed-forward

La majorité des méthodes de machine learning utilisent des réseaux de neurones, c'est donc un concept clef. Les réseaux de neurones sont inspirés du cerveau humain. Le cas le plus classique étant le réseau de neurones à propagation directe.

Un réseau de neurones est constitué de plusieurs couches de neurones, chaque couche étant connectée à la précédente par des poids. Tout réseau de neurones contient une couche d'entrée et de sortie. Un réseau de neurones feed-forward récupère des valeurs en entrées, puis les valeurs des neurones des couches suivantes sont déterminées récursivement, en faisant une somme pondérées des neurones de la couche précédente auquel est appliquée une fonction d'activation. Ensuite les valeurs de la couche de sortie peuvent être exploitées.

Ainsi un réseau de neurones peut être définit par :

— Le nombre de couches

— Le nombre de neurones dans chaque couche

— Les types de connexion entre les couches

La dimension du réseau et les fonctions d'activation choisie influent directement sur l'exactitude des prédictions fournies, c'est un choix important et les paramètres optimaux varient selon les cas. Ensuite pour n couches contenant respectivement $n_i, \quad i \in [|1, n|]$ neurones il existe $\prod n_i$ (nombre d'arètes dans le réseau) poids ajustables si le réseau est dense (entièrement connecté). Souvent le réseau est initialisé avec des poids aléatoires, qui sont ajustés au fur et à mesure de son entraînement.

Un réseau de neurones peut être utilisé pour reconnaître des images, par exemple dire si une image contient un chat ou non. La couche d'entrée a une dimension fixe représentant les valeurs de chaque pixel de l'image (dimension $p \times q$ en greyscale ou $p \times q \times 3 en RGB$) et la couche de sortie contient un seul neurones représentant un booléen (vrai ou faux). Avec assez de couches et les bons poids, le réseau de neurones est capable de reconnaître efficacement des images de chat.

La valeur prise par un neurone individuel k est :

$$V = f_a(\sum_i w_i x_i + w_0)$$

(voir figure 34

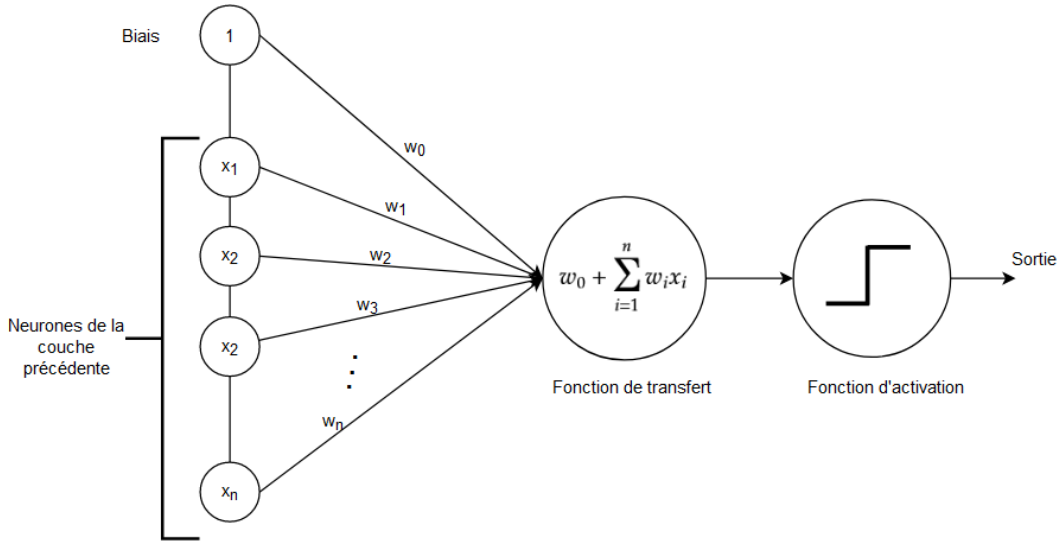

FIGURE 34 – Perceptron : valeur prise par un neurone

VII.2.a Les fonctions d'activation

Une fonction d'activation est une fonction de $\mathbb{R} \rightarrow \mathbb{R}$ qui sert à délinéariser le réseau ou restreindre les valeurs des neurones. Le choix des fonctions d'activation est important et dépend de l'objectif attendu.

Voici quelques exemples de fonctions d'activation classiques qui ont chacun leurs cas d'application :

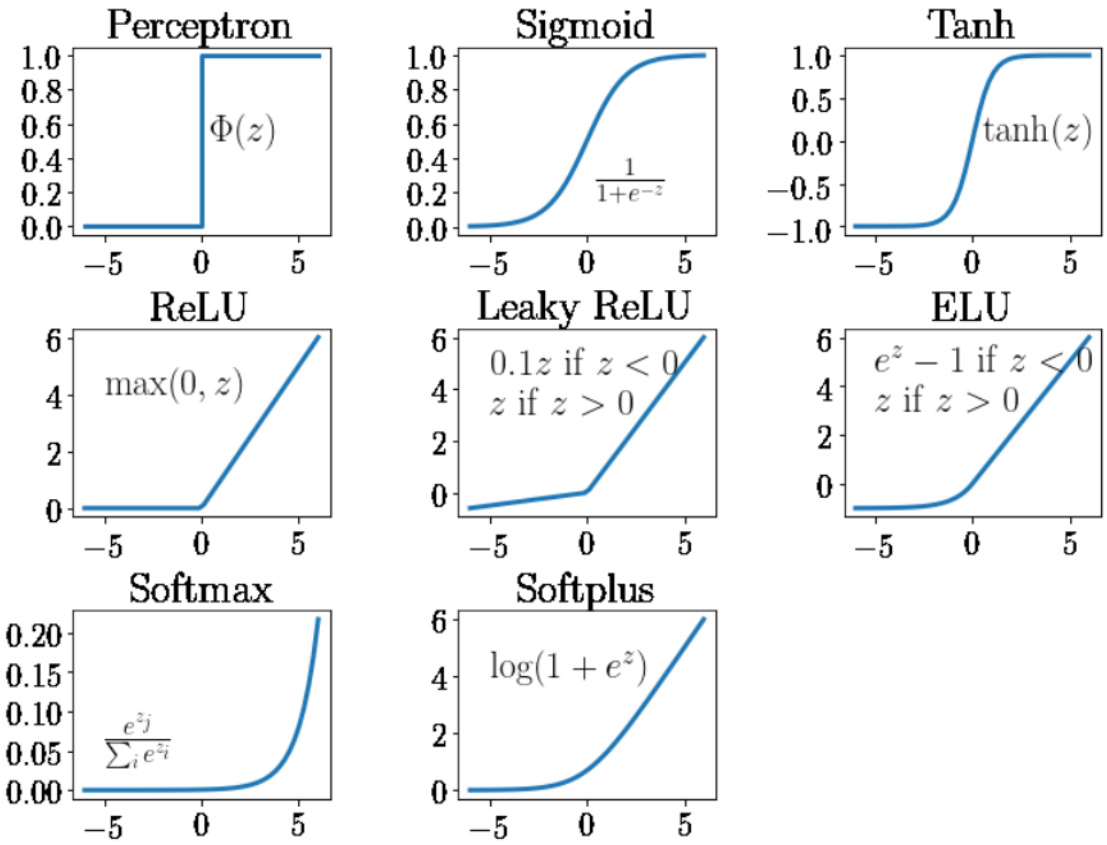

FIGURE 35 – Fonctions d'activation classiques [4]

VII.3 Réseaux convolutifs (CNNs)[5]

VII.3.a Motivation

Les réseaux de neurones convolutifs (CNNs) sont utilisés pour traiter des données qui peuvent être représentées par une grille. Par exemple, une liste de temps peut être représentée par une grille 1D, une image par une grille 2D. Ces réseaux sont appelés "convolutifs" car ils utilisent l'opérateur mathématique de convolution. L'étude de l'architecture de ces réseaux fait partie intégrante de la recherche actuelle en Deep Learning, avec de nouvelles architectures découvertes presque tous les mois.
Dans le cadre de notre PAr, ces réseaux de convolution sont utilisés pour encoder l'état d'un jeu, qui est représenté par une matrice de pixels. Le réseau de convolution est capable d'encoder certaines caractéristiques de l'état visuel du jeu, sans que ces dernières aient été explicitées. L'encodage est retenu uniquement si le résultat est bon, et le réseau met à jour cet encodage jusqu'à ce que l'erreur commise atteigne un minimum locale. Chaque couche du réseau applique des filtres aux images intermédiaires, afin d'extraire certaines caractéristiques. Le site tensorspace.org permet de mieux cerner ces opérations de convolution et de filtrage. Les paragraphes suivants formalisent mathématiquement l'opérateur de convolution.

VII.3.b Opérateur de convolution

Soient x, y deux fonctions réelles ou complexes, intégrales au sens de Lebesgues. Leur produit de convolution $(f * g)$ est défini pour presque tout réel t par :

$$(x * y)(t) = \int_{-\infty}^{+\infty} x(a)y(t-a)da \tag{5}$$

Dans le cas des réseaux de convolution, le premier argument (x) est appelée l'*input*, et le second (y) le *kernel*.

Dans le cas réel, les signaux x et y prennent leurs valeurs sur un ensemble discret, et leur convolution s'écrit :

$$x * y(t) = \sum_{a=-\infty}^{+\infty} x(a)y(t-a) \tag{6}$$

Dans les application en Machine Learning, l'*input* est en général un tableau multidimensionnel, et le *kernel* est un tableau multidimensionnel de paramètres adaptés par l'algorithme d'apprentissage. On appelle ces tableaux multidimensionnels **tenseurs**.
La somme infinie dans la définition est effectuée sur les tenseurs, et est donc finie en pratique.
Dans le cas de traitement d'images, l'*input* I est bi-dimensionnel et réprésente les pixels de l'image. Le *kernel* k est lui aussi choisi bidimenssionnel :

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{7}$$

En traitement du signal, on utilise l'intercorrélation (**cross-correlation** en anglais) des signaux. Si x, y sont deux signaux à énergie finie (donc dans $L_2(\mathbb{R})$), leur intercorrélation est définie par :

$$R_{xy}(\tau) = (x * y)(t) = \int_{-\infty}^{+\infty} x(t+a)y(a)da \tag{8}$$

Le produit de convolution d'un signal avec lui même est l'autocorrélation. Ces opérateurs permettent de mesurer la similitude entre deux signaux.
Beaucoup de bibliothèque de Machine Learning implémente l'interrcorrélation et l'appelle la convolution. Dans le cas du traitement d'image, l'intercorrélation entre l'*input* et le *kernel* s'écrit :

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{9}$$

En Machine Learning, on appelle donc convolution l'intercorrélation.
La convolution pour les images en machine utilise en réalité des tenseurs à 4 axes. Un développement sera effectué ultérieurement.

VII.3.c Matrice de Toeplitz et matrice circulante

La convolution discrète peut s'effectuer à l'aide de ces matrices. L'intérêt est qu'il existe des algorithmes pour optimiser le temps de calcul des multiplications matricielles, qui peuvent être effectuées avec calcul en parallèle.

VII.3.d Base neuroscientifique des réseaux de convolution

Les réseaux de convolution sont fortement inspirés du traitement des images par le cerveau humain[5].

VII.3.e Exemple de convolution

Voici un exemple classique de convolution qui permet de faire apparaître les contours d'une image. On prend pour une exemple l'image I du jeu breakout en niveaux de gris :

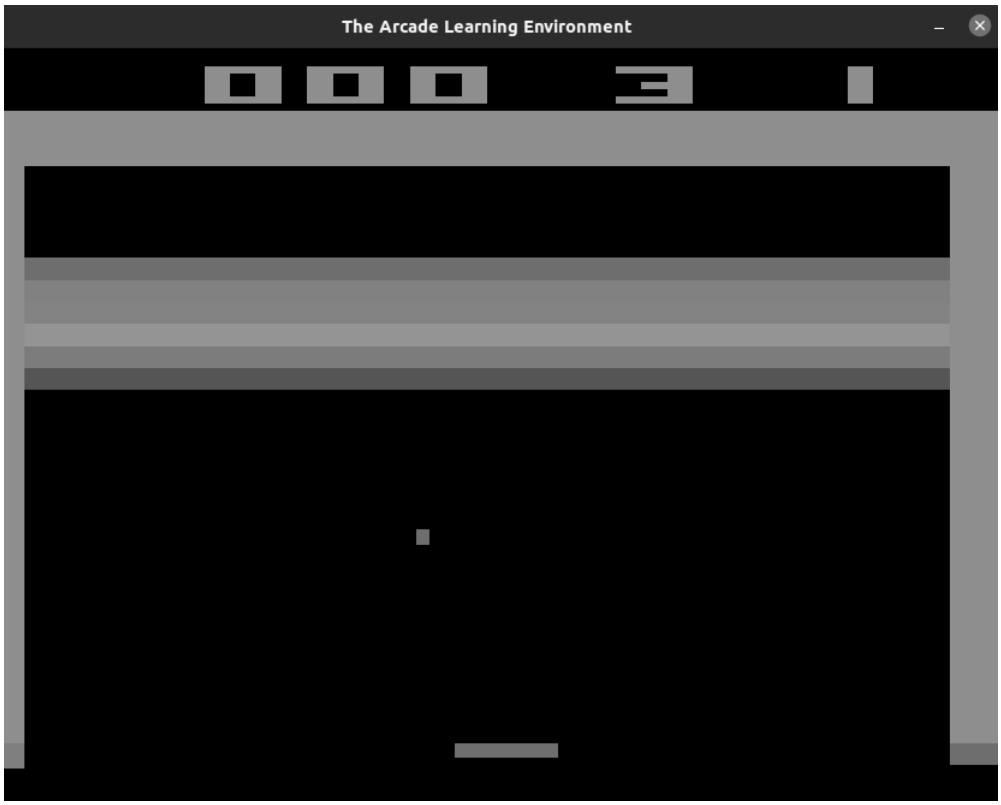

FIGURE 36 – Image du jeu breakout en niveau de gris

On applique les noyaux de convolution D_x et D_y suivants :

$$D_y = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \tag{10}$$

$$D_x = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \tag{11}$$

D_y correspond à une dérivée selon l'axe y, D_x à une dérivation selon l'axe x.
L'image $DIdy = I * D_y$ figure **38** obtenue après convolution par le noyau D_y fait apparaître les contours horizontaux. En effet, le résultat de la convolution est non nul lorsqu'il y a une forte variation d'intensité verticale.
Avec le même raisonnement, $DIdx = I * D_x$ figure **37** fait apparaître les contours verticaux.

Dans le cas d'un réseaux de convolution, certains couches du réseaux effectuent des convolutions avec ce type de noyaux, et le réseau doit ensuite interpréter le résultat

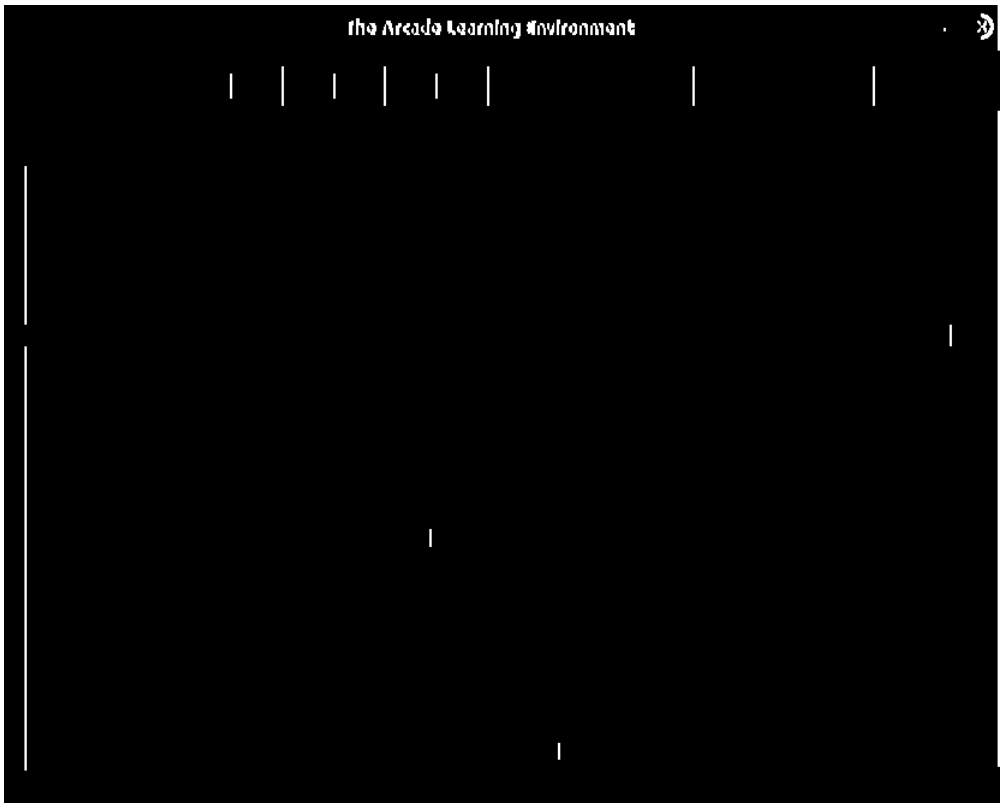

FIGURE 37 – Dérivée selon l'axe x : I*Dx

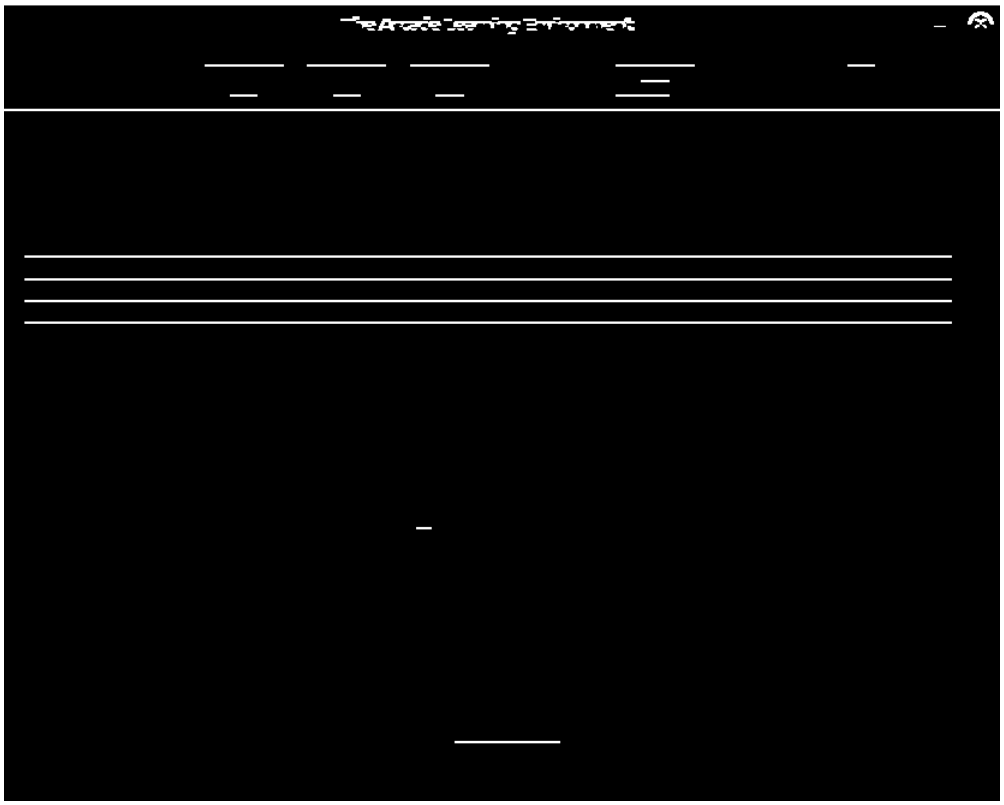

FIGURE 38 – Dérivée selon l'axe y, I*Dy

de ces couches pour prendre une décision. Les convolutions sont effectuées sur les 3 matrices correspondant aux 3 couleurs (rouge-vert-bleu) de l'image, et le réseau prend une décision avec l'ensemble de ces résultats.

VII.4 Théorèmes mathématiques

Cette partie présente des théorèmes mathématiques utilisés dans la suite du rapport.

VII.4.a Théorème de Radon-Nikodym-Lebesgues [6]

Soit λ, μ deux mesures σ-finies sur (Ω, F). Alors il existe un unique couple de mesures (μ_1, μ_2) σ-finies positives tel que :

— $\mu = \mu_1 + \mu_2$
— $\mu_1 << \lambda$
— $\mu_2 \perp \mu_1$

Il existe une unique fonction h mesurable positive telle que pour tout $F_1 \in F$:

$$\mu_1(F_1) = \int_{F_1} h d\nu \tag{12}$$

On dit que μ possède une densité h par rapport à ν. On note :

$$h = \frac{d\mu}{d\nu} \tag{13}$$

Corollaire : Soient μ, ν deux mesures σ-finies positives sur (Ω, F). Il y a équivalence entre :

— $\mu \ll \nu$
— μ possède une densité par rapport à ν

VII.4.b Inégalité de Jensen

Soit f une fonction convexe et $(x_1, ..., x_n)$ un n-uplets de réels positifs appartenant à l'intervalle de définition de f, et $(\lambda_1, ..., \lambda_n)$ un n-uplet de réels positifs tels que :

$$\sum_{i=1}^{N} \lambda_i = 1 \tag{14}$$

Alors :

$$f(\sum_{i=1}^{N} \lambda_i x_i) \leq \sum_{i=1}^{N} \lambda_i f(x_i) \tag{15}$$

VII.5 Théorie de l'information

La théorie de l'information est une branche des mathématiques appliquées qui permet de quantifier la quantité d'information présente dans un signal. Dans le cas du Machine Learning, la théorie de l'information est utilisée pour caractériser des lois probabilistes et les comparer entre elles. Dans le cas des probabilités, la réalisation d'un événement peu probable contient plus d'information que la réalisation d'un événement presque sûr.

VII.5.a Information d'un événement

L'information d'un événement décroît donc avec sa probabilité. Intuitivement, on souhaite que les 3 propriétés suivantes soient vérifiées par l'information :

— Les événements probables doivent avoir peu d'information, les événements presque sûrs ne doivent pas contenir d'information.

— Les événements incertains doivent contenir plus d'information.

— L'information est additive pour des événements indépendants.

L'information d'un événement est définie par :

$$I(x) = -logP(x) \tag{16}$$

On vérifie que I vérifie bien les trois propriétés voulues.
Lorsque le logartihme est dans la base usuelle e, l'unité de l'information est le *nat*.

VII.5.b Information d'une loi de probabilité : entropie de Shannon

Pour caractériser l'information du loi de probabilité, on calcule l'espérance de l'information de la loi :

$$H(x) = \mathbb{E}_{x\sim P}[I(x)] = -\mathbb{E}_{x\sim P}[\log(P(x))] \tag{17}$$

H est appelé **entropie de Shannon**.
Les propriétés de l'information d'un événement se généralisent à l'entropie : elle est faible si une loi est déterministe, et élevée si la loi est proche d'une loi uniforme.

VII.5.c Comparaison de lois de probabilités : divergence KL

Pour mesurer la différence entre deux lois de probabilité s'appliquant à la même variable aléatoire X, on peut faire la différence de leur entropie de Shannon, aussi appelée divergence **Kullback-Leibler** (KL) :

$$D_{KL}(P||Q) = \mathbb{E}_x[\log(\frac{P(x)}{Q(x)})] = \mathbb{E}_{x\sim P}[\log P(x) - \log Q(x)] \tag{18}$$

Dans le cas de lois de probabilités discrètes, la divergence KL s'écrit directement :

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)} \tag{19}$$

Dans le cas de distributions P et Q continues, on utilise des intégrales :

$$D_{KL}(P||Q) = \int_{-\infty}^{+\infty} p(x) \log \frac{p(x)}{q(x)} dx \tag{20}$$

Dans le cas général, on considère P, Q deux mesures σ-finies sur (Ω, F), absolument continues par rapport à une mesure μ. C'est possible car sur R, B(R), on peut considérer la mesure absolument continue par rapport à la mesure de Lebesgues de la décomposition de Lebesgues de P et Q. Par le corollaire du théorème de Radon-Nikodym-Lebesgues, il existe des densités p, q telles que $dP = pd\mu$ et $dQ = qd\mu$.
On définit alors :

$$D_{KL}(P||Q) = \int_F p \log \frac{p}{q} d\mu \tag{21}$$

D_{KL} existe si P est absolument continue par rapport à Q. Dans ce cas, $\frac{p}{q} = \frac{dP}{dQ}$ est la dérivée de Radon-Nikodym-Lebesgues de P par rapport à Q, et la divergence KL s'écrit :

$$D_{KL}(P||Q) = \int_F \log \frac{dP}{dQ} dP = \int_F \frac{dP}{dQ} \log \frac{dP}{dQ} dQ \tag{22}$$

On reconnaît la différence des entropies de P et Q.

Propriété : La divergence KL n'est pas symétrique. Il en résulte que ce n'est pas une distance. Elle ne vérifie par non plus l'inégalité triangulaire.

Propriété (positivité) : La divergence KL est positive.
Démonstration (cas dicret) :
Dans le cas discret, par définition :

$$D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)} = -\sum_i P(i) \log \frac{Q(i)}{P(i)} \tag{23}$$

Le logarithme étant concave, $-log$ est convexe, et par l'inégalité de Jensen :

$$\sum_i P(i) \log \frac{Q(i)}{P(i)} \leq \log(\sum_i P(i) \frac{Q(i)}{P(i)}) = \log(\sum_i Q(i)) = \log(1) = 0 \tag{24}$$

VII.5.d Estimateur de maximum vraisemblance

Considérons X un vecteur aléatoire sur (Ω, $\mathbb{F}$, $\mathbb{P}$) un espace probabilisé, à valeurs dans ($\mathbb{R}^d$, $\mathbb{B}(\mathbb{R}^d)$). On suppose que X suit une certaine loi $\mathbb{P}_X$, non précisée. Soit $\mathbb{X} = (X^{(1)}, ..., X^{(n)})$ n réalisations indépendantes de X.
Soit $\mathbb{P}_{model}(\mathbf{x}, \theta)$ une famille paramétrée par θ de lois de probabilités, i.e. $\mathbb{P}_{model}(\mathbf{x}, \theta)$ approxime la vraie probabilité de l'événement $[X \in \mathbf{x}]$ qui est $\mathbb{P}_X(\mathbf{x})$, pour $\mathbf{x} \in \mathbb{B}(\mathbb{R}^d)$.
On définit l'estimateur de maximum de vraisemblance par :

$$\theta_{ML} = \underset{\theta}{argmax} \quad P_{model}(\mathbb{X}; \theta) \tag{25}$$

Les n réalisations étant indépendantes :

$$\theta_{ML} = \underset{\theta}{argmax} \prod_{i=1}^{n} P_{model}(X^{(i)}; \theta) \tag{26}$$

Le calcul de sommes est mieux réalisé par les ordinateurs que le calcul de produits, notamment à cause des erreurs sur les flottants. On préfère donc prendre le logarithme dans le terme de droite de l'expression précédente. Le logarithme étant strictement croissant, le terme de gauche est conservé :

$$\theta_{ML} = \underset{\theta}{argmax} \sum_{i=1}^{n} \log(P_{model}(X^{(i)}; \theta)) \tag{27}$$

argmax étant homogène de degré 0, on peut diviser le terme de droite par n. En considérant une variable aléatoire **x** qui suit une loi uniforme sur les données empiriques $\tilde{P}_{data}$, on obtient l'expression sous la forme d'une espérance :

$$\theta_{ML} = \underset{\theta}{argmax}\, \mathbb{E}_{x \sim \tilde{P}_{data}} \log(P_{model}(X^{(i)}; \theta)) \tag{28}$$

Maximiser θ dans (22) revient en fait à minimiser la dissimilarité entre la distribution empirique $\tilde{P}_{data}$, qui est celle du *training set*, et la distribution du modèle paramétrique. Cette dissimilarité se mesure avec la divergence KL, introduite dans la partie précédente :

$$D_{KL}(\tilde{P}_{data} || P_{model}) = \mathbb{E}_{x \sim \tilde{P}_{data}}[\log \tilde{P}_{data}(\mathbf{x}) - \log P_{model}(\mathbf{x})] \tag{29}$$

Le terme de gauche dans la différence dépend du processus de génération de données, et pas du modèle, donc la minimisation de la divergence KL est la minimisation de :

$$-\mathbb{E}_{x \sim \tilde{P}_{data}}[log P_{model}(\mathbf{x})] \tag{30}$$

qui n'est autre que la maximisation de (22).
La divergence KL a une valeur minimale de 0, comme démontré dans la partie précédente. C'est une propriété utile pour sa minimisation.

VII.6 Equation de Bellman démonstration

$$V_{\Pi}(s) = \mathbb{E}_{\Pi}[R_t | s_t = s]$$

$$Q_{\Pi}(s, a) = \mathbb{E}_{\Pi}[R_t | s_t = s, a_t = a]$$

Les relation liant Q_{Π} et V_{Π} s'obtiennent facilement en utilisant la formule de Bayes pour les espérances conditionnelles (rappel $\Pi(s, a)$ est la probabilité de jouer a, étant dans l'état s) :

$$V_\Pi(s) = \sum_{a \in A} \Pi(s,a) Q_\Pi(s,a)$$

$$\begin{aligned} Q_\Pi(s,a) &= \mathbb{E}_\Pi[R_t|s,a] \\ &= \mathbb{E}_\Pi[r_{t+1} + \gamma R_{t+1}|s_t = s, a_t = a] \\ &= \sum_{s' \in S} P(s'|s,a)(\mathbb{E}[r_{t+1}|s,a,s'] + \gamma V_\Pi(s')) \end{aligned} \tag{31}$$

VII.6.a Rétropropagation et descente du gradient

Un réseau de neurones ne commence jamais avec les poids optimaux capables de résoudre efficacement un problème. Les poids variables sont initialisées la plus part du temps aléatoirement. Il existe différentes méthodes d'apprentissage, la plus repondue étant la descente du gradient stochastique.

Le principe de base de l'ajustement des poids est le suivant :

- Pour chaque échantillon d'entraînement
  - Récupérer la sortie du réseau
  - Récupérer la sortie réelle (ou attendue)
  - Ajuster les poids en effectuant une rétropropagation en comparant sortie du réseau et la sortie attendue

Le réseau s'entraîne sur un jeu de données étiquetées, c'est à dire des entrées et la sortie attendue. Ensuite l'erreur entre la réponse donné et la sortie attendue est mesurée grâce à une fonction erreur (ou perte). Une fonction perte est une fonction qui prend en entrée deux vecteurs de même dimension et retourne un réel, cette fonction doit être minimale lorsque les deux vecteurs sont égaux.

Par exemple en appelant X l'entrée, Y la sortie, Y_r la sortie attendue, et W les poids du réseau, une fonction erreur naturelle est :

$$MSE(Y(X,W), Y_r) = ||Y(X,W) - Y_r||^2$$

Les poids sont alors ajusté grace au gradient selon les poids de la fonction perte.

$$W \leftarrow W - \alpha \overrightarrow{\nabla}_W MSE$$

Après un nombre d'entraînement équilibré (entraîner suffisamment sans surentraîner), on peut espérer trouver un minimum local satisfaisant et des prédictions pertinentes sur des jeux de données jamais vus auparavant.

La notion de réseau de neurones et de descente du gradient est vaste et sera approfondit section VII.2.

VII.7 Implémentation puissance4 : diagrammes explicatifs

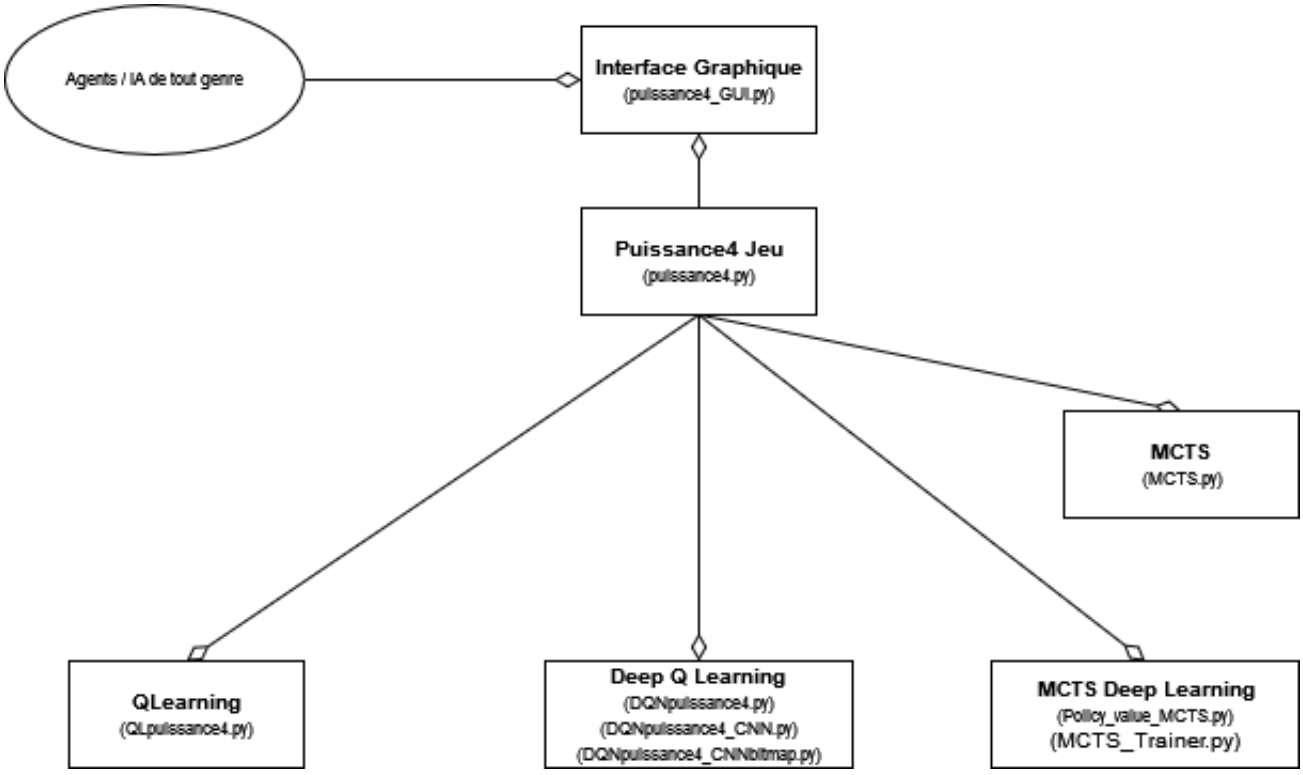

FIGURE 39 – Diagramme Général de des classes pour le puissance4

Game

+ h,w : int (taille du jeu)
+board : int array array (matrice (h,w) de 0,1 ou -1)
+turn : -1 ou 1 (tour)

#JOUER UN COUP/ SIMULER UN COUP#

+play_move(move : int) : w in {False,None,0,1,-1}
Joue le coup move et renvoie le gagnant
L'état du jeu est mis à jour.
(None = partie en cours
False = coup invalide)
+simulate_move(move : int, board : int array array) : gagnant et nouveau board
Analogue à play_move, sans modifier l'état du jeu

#CONVERTION ETAT VERS BOARD et BOARD VERS ETAT#
+get_state(board : int array array) : différentes méthodes pour récupérer l'état du jeu sous différentes formes
(Aplati, bitmap, hash)
+convert_state_to_board(?state) : board : int array

différentes méthodes pour convertir différents types d'état en board
#IA MINMAX#
+minmax(*args) : best_move : int, best_score : int
IA minmax implémentée.
Peut être utilisé sans arguments, la fonction récupère alors l'état du jeu

FIGURE 40 – Détails sur la classe Game du script puissance4.py (utilisée par tous les agents/IAs)

VII.8 Recherche arborescante Monte Carlo

Noeud MCTS :

— Etat du noeud

— Parents = Etat menant au noeud

— Fils = Etats accessibles depuis le noeud

— Valeur : initialement 0

— Visites : initialement 0

1 Entrée : un état du jeu Initialisation : création d'un Noeud* contenant la valeur de l'état considéré
2 **pour** $i = 1, 2, ..., N_{itrations}$ **faire**
3 | Sélection : on part de la racine, et on sélectionne le fils qui maximise un terme spécifique, jusqu'à un noeud qui peut être développé (expanded).
4 | Expansion : On choisit un coup aléatoire, on le joue, puis on attribue une valeur au nouvel état.
5 | Simulation : lorsqu'on ne sait pas attribuer de valeur à l'état, on simule une partie à partir de l'état, et sa valeur sera le résultat de la partie
6 | Retropropagation : On incrémente le nombre de visite et la valeur de ce nouvel état ainsi que tous les parents.
7 **fin**
8 Choix du coup à jouer parmis les fils de la racine :

— Le plus visité

— La valeur la plus élevée

— La valeur moyenne la plus élevée

— Maximisant une quantité autre (ex : LCB)

Algorithme 5 : Algorithme MCTS

L'étape de simulation est utilisée lorsqu'il n'est pas possible d'attribuer une valeur à un état. Dans le MCTS adapté à des réseaux de neurones, cette étape est évitée, la valeur de l'état est simplement donné par le Value Network.

VII.9 Résultats puissance 4

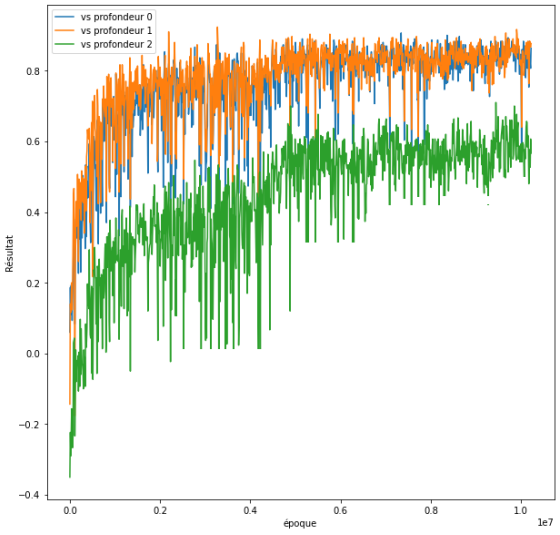

FIGURE 41 – Résultats bruts pour le QL 4x4 pour l'agent X

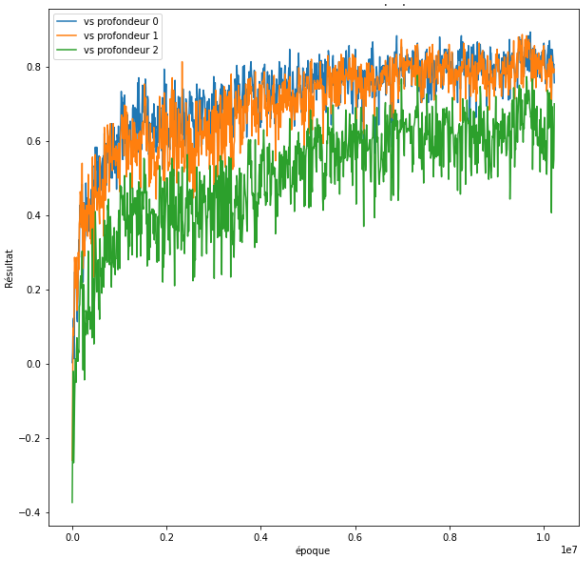

FIGURE 42 – Résultats bruts pour le QL 4x4 pour l'agent O

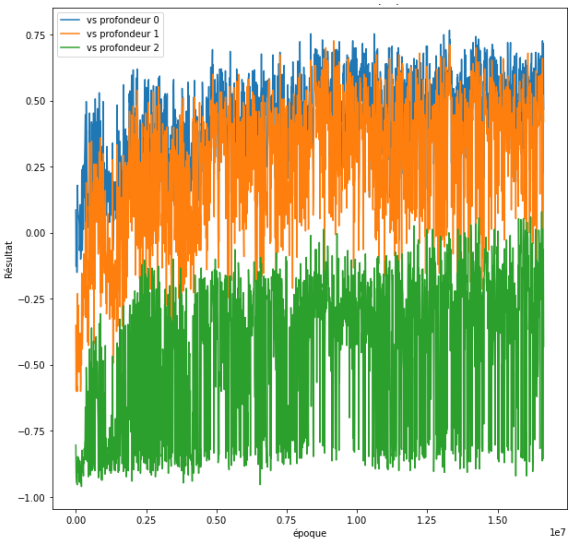

FIGURE 43 – Résultats bruts pour le QL 6x7 pour l'agent X

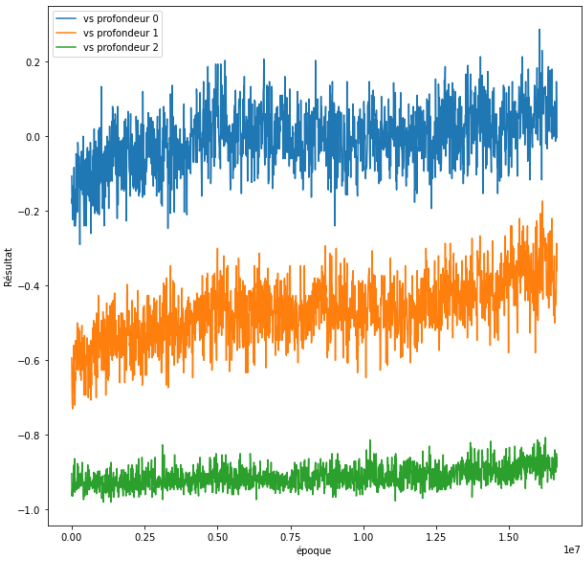

FIGURE 44 – Résultats bruts pour le QL 6x7 pour l'agent O

VII.10 Références, mesure, Puissance 4

Les tableaux suivants présentent le résultat après 1000 parties entre deux IAs Min-Max de profondeur entre 0 et 6 (ou 0 et 5). Chaque case représente : le taux de victoire,

le taux de défaite, et le résultats (victoires - défaites) pour le joueur 1 (rouge).

Prof	0	1	2	3	4	5	6
0	28 %, 24 % 4	46 %, 19 % 26	46 %, 4 % 41	47 %, 5 % 41	50 %, 4 % 46	50 %, 4 % 46	47 %, 3 % 44
1	26 %, 32 % -6	39 %, 25 % 14	41 %, 8 % 33	45 %, 7 % 37	47 %, 5 % 41	46 %, 5 % 41	49 %, 3 % 46
2	4 %, 40 % -36	7 %, 37 % -30	8 %, 12 % -3	14 %, 12 % 2	13 %, 8 % 5	22 %, 9 % 12	18 %, 7 % 11
3	5 %, 43 % -38	6 %, 41 % -35	9 %, 16 % -7	13 %, 17 % -3	15 %, 14 % 0	19 %, 12 % 7	20 %, 8 % 11
4	2 %, 45 % -42	4 %, 42 % -38	6 %, 18 % -12	7 %, 19 % -12	6 %, 14 % -8	15 %, 12 % 3	15 %, 7 % 7
5	3 %, 50 % -46	4 %, 47 % -42	5 %, 26 % -20	9 %, 25 % -16	6 %, 22 % -16	15 %, 21 % -6	14 %, 12 % 2
6	1 %, 50 % -48	1 %, 47 % -46	1 %, 28 % -26	3 %, 27 % -24	3 %, 21 % -17	5 %, 26 % -21	6%, 13% -6

TABLE 1 – Résultats MinMax pour un puissance4 de taille 4x4

Prof	0	1	2	3	4	5
0	54 %, 44 % 10	80 %, 19 % 60	96 %, 3 % 92	95 %, 4 % 90	99 %, 1 % 98	99 %, 1 % 98
1	32 %, 67 % -34	60 %, 39 % 21	91 %, 8 % 82	93 %, 7 % 86	98 %, 2 % 96	99 %, 1 % 98
2	8 %, 91 % -83	13 %, 86 % -73	49 %, 39 % 9	69 %, 26 % 42	76 %, 16 % 60	86 %, 8 % 78
3	6 %, 93 % -87	8 %, 91 % -83	32 %, 64 % -32	53 %, 44 % 8	63 %, 31 % 32	83 %, 12 % 71
4	4 %, 95 % -90	3 %, 96 % -93	17 %, 73 % -55	28 %, 67 % -39	41 %, 43 % -2	61 %, 32 % 28
5	2 %, 97 % -95	3 %, 96 % -92	17 %, 76 % -59	24 %, 71 % -47	34 %, 55 % -21	49 %, 45 % 4

TABLE 2 – Résultats MinMax pour un puissance4 de taille 6x7

VII.11 Utilisation de l'interface graphique puissance4 pour tester les agents

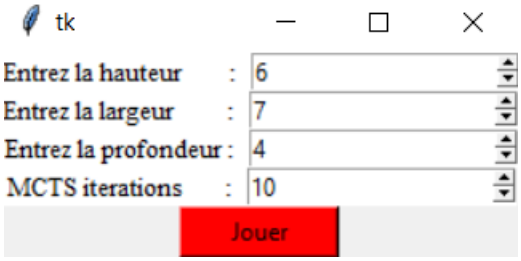

FIGURE 45 – Première fenêtre de l'interface graphique : menu initial

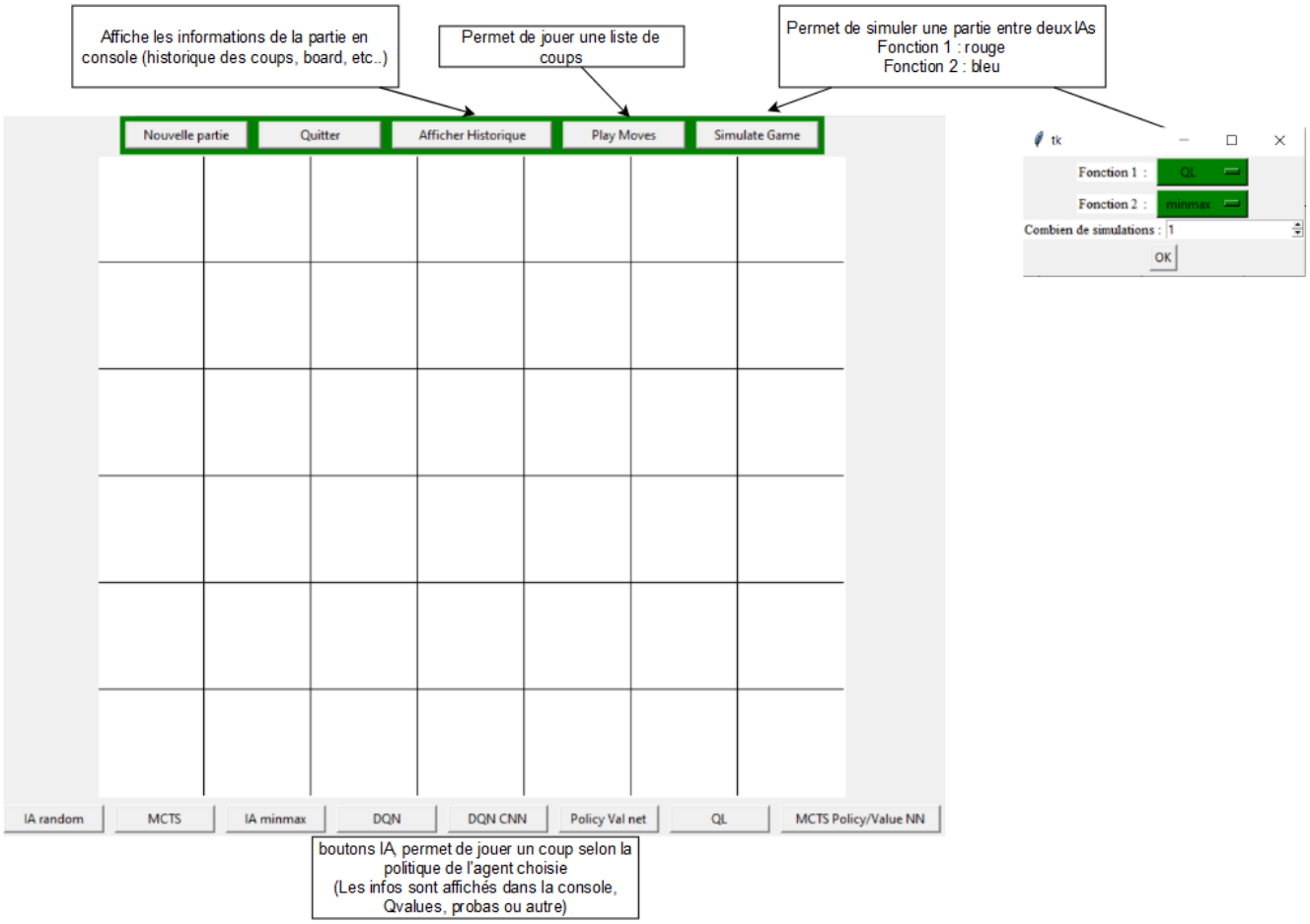

FIGURE 46 – Utilisation de l'interface graphique

```
Qs for CNN net (player 1) :  [5.542696  5.5035706 4.562918  7.6469307
5.6270833 5.272086  5.395219 ]
```

FIGURE 47 – Exemple d'affichage en console lors de l'utilisation du bouton "DQN CNN"

Références

[1] Dimitri P. BERTSEKAS. *Dynamic programming and optimal control. volume 1*. eng. Fourth edition. Belmont, Mass : Athena Scientific, 2017. ISBN : 978-1-886529-43-4.

[2] *Cartpole Game*. URL : https://jeffjar.me/cartpole.html (visité le 05/04/2022).

[3] *collections — Container datatypes — Python 3.10.4 documentation*. URL : https://docs.python.org/3/library/collections.html (visité le 29/03/2022).

[4] *FIG. 13 : Common activation functions in artificial neural networks...* en. URL : https://www.researchgate.net/figure/Common-activation-functions-in-artificial-neural-networks-NNs-that-introduce_fig7_341310767 (visité le 29/03/2022).

[5] Ian GOODFELLOW, Yoshua BENGIO et Aaron COURVILLE. *Deep learning*. Adaptive computation and machine learning. Cambridge, Massachusetts : The MIT Press, 2016. ISBN : 978-0-262-03561-3.

[6] *Kullback–Leibler divergence*. en. Page Version ID : 1073690484. Fév. 2022. URL : https://en.wikipedia.org/w/index.php?title=Kullback%E2%80%93Leibler_divergence&oldid=1073690484 (visité le 29/03/2022).

[7] Micheal LANHAM. *Hands-on reinforcement learning for games : implementing self-learning agents in games using artificial intelligence techniques*. English. OCLC : 1140142977. 2020. ISBN : 978-1-83921-493-6.

[8] Marlos C. MACHADO et al. "Revisiting the Arcade Learning Environment : Evaluation Protocols and Open Problems for General Agents". en. In : *Journal of Artificial Intelligence Research* 61 (mars 2018), p. 523-562. ISSN : 1076-9757. DOI : 10.1613/jair.5699. URL : https://jair.org/index.php/jair/article/view/11182 (visité le 29/03/2022).

[9] Stuart J. RUSSELL et Peter NORVIG. *Artificial intelligence : a modern approach*. Fourth edition. Pearson series in artificial intelligence. Hoboken : Pearson, 2021. ISBN : 978-0-13-461099-3.

[10] David. SILVER, Thomas HUBERT et Julian SCHRITTWIESER. *A general reinforcement learning algorithm that masters chess, shogi and Go through self-play*. 1st edition. Deepmind, 2018. URL : https://storage.googleapis.com/deepmind-media/DeepMind.com/Blog/alphazero-shedding-new-light-on-chess-shogi-and-go/alphazero_preprint.pdf.

[11] Richard S. SUTTON et Andrew G. BARTO. *Reinforcement Learning : An Introduction*. Second edition. Bradford, 2015.