A Memory Efficient Deep Reinforcement Learning Approach For Snake Game Autonomous Agents

Reinforcement Learning

Mention Mathématiques et Informatique
Parcours HPDA

18/10/2024

Rédigé par :
CLEMENCEAU Maxime
maxime.clemenceau@cy-tech.fr
HONOR Julien
julien.honor@cy-tech.fr

PAULY Alexandre
alexandre.pauly@cy-tech.fr
SABADIE Laura
laura.sabadie@cy-tech.fr

Abstract :

Cet article présente une méthode d'apprentissage par renforcement profond modifiée pour jouer au jeu du Snake en utilisant moins de mémoire et de ressources de calcul tout en atteignant de bonnes performances. Ce rapport propose deux approches, une utilisant le prétraitement d'images, un réseau neuronal convolutif léger et un tampon de rejeu d'expérience plus petit pour réduire les exigences de mémoire par rapport aux méthodes traditionnelles. La seconde utilise un modèle plus simple sans couches convolutionnelles et directement accès sur les états du jeu.

Contents

1 Introduction

Le Reinforcement Learning (RL) est une technique qui entraîne les logiciels à prendre des décisions en vue d'obtenir les meilleurs résultats. Elle imite le processus d'apprentissage par tâtonnements employé par les êtres humains pour atteindre leurs objectifs. Les algorithmes de RL utilisent un paradigme de récompense et de punition lorsqu'ils traitent les données. Ils apprennent du retour d'information de chaque action et découvrent par eux-mêmes les chemins de traitement optimaux pour atteindre les résultats finaux [https://aws.amazon.com/fr/what-is/reinforcement-learning/]. Bien que les premiers algorithmes de renforcement aient démontré leur efficacité dans des contextes simples, ils n'ont pas réussi à se généraliser pour des entrées de grande dimension, limitant ainsi leur applicabilité pratique.

Récemment, les méthodes basées sur des réseaux de neurones convolutifs (CNN) ont permis de contrôler des environnements visuels en utilisant des images, mais ces approches sont très gourmandes en mémoire, ce qui les rend inadaptées à l'utilisation sur des appareils mobiles ou des robots de milieu de gamme. Une des solutions à ce problème repose sur l'utilisation du replay buffer, qui améliore l'efficacité des données et stabilise l'apprentissage des modèles. Cependant, la taille optimale de ce buffer reste un problème non résolu.

2 Objectif de l'étude

Dans cette étude [1], Md. Rafat Rahman Tushar et Shahnewaz Siddique, chercheurs au département de génie électrique et informatique de la North South University à Dhaka (Bangladesh), proposent une approche modifiée pour le prétraitement des images et l'optimisation de la taille du **replay buffer**. L'objectif est de réduire la consommation de mémoire des algorithmes d'apprentissage par renforcement profond (DRL), tout en maintenant des performances élevées. Ce modèle a été testé dans un environnement simulé à l'aide du jeu classique **Snake**, et les résultats obtenus sont comparables à d'autres méthodes, tout en minimisant l'utilisation de mémoire, ce qui le rend applicable sur des dispositifs mobiles.

L'objectif de cette implémentation est de concevoir une version optimisée de l'algorithme **Deep Q-learning** qui améliore l'utilisation de la mémoire tout en maintenant des performances élevées, sans recourir aux CNN. En utilisant un réseau de neurones entièrement connecté (linear layers), cette approche cherche à réduire la complexité et la consommation de mémoire par rapport aux méthodes traditionnelles, qui nécessitent des traitements d'images intensifs. L'idée est de simplifier le modèle tout en maximisant l'efficacité en mémoire, en tirant parti de la structure du réseau pour apprendre efficacement les relations entre les états et les actions du jeu. L'objectif final est d'atteindre des performances comparables à celles des méthodes basées sur des CNN, tout en ayant une empreinte mémoire significativement réduite, rendant cette solution adaptée aux environnements à ressources limitées.

3 Le Jeu Snake : Règles et Objectifs

Snake [2] est un jeu emblématique, né dans les années 1970 et popularisé dans les années 1990 grâce à sa pré installation sur les téléphones Nokia. Sa simplicité et son accessibilité en ont fait un jeu addictif.

Le gameplay de base consiste à contrôler la direction du serpent (droite, gauche ou tout droit) pour manger des pommes qui apparaissent aléatoirement. Chaque pomme mangée fait grandir le serpent, rendant sa manœuvrabilité plus difficile et augmentant les risques de collision avec sa propre queue ou les bords de l'écran. Le joueur gagne lorsque le serpent remplit l'écran et perd en cas de collision.

4 Modélisation

L'objectif est de réduire l'utilisation de la mémoire pendant l'entraînement tout en atteignant les meilleures performances possibles. La mémoire de rejeu prend une quantité considérable de mémoire, comme décrit plus tard. L'étude menée par Md. Rafat Rahman Tushar et Shahnewaz Siddique [1] tente alors d'optimiser l'efficacité de cette mémoire en réduisant l'exigence massive de la mémoire tampon de rejeu avec le prétraitement d'image et la taille de la mémoire tampon. Pour cela, la taille de la mémoire tampon est soigneusement choisie afin que l'agent ait les informations nécessaires pour bien s'entraîner et atteigne un score modéré. Pour ce faire, une légère variation de l'algorithme d'apprentissage par renforcement profond est utilisée pour atteindre cet objectif.

Mise en place du jeu

Comme évoqué précédemment, l'application de cette méthode cible le jeu connu de tous, Snake. Dans ce cadre, il a été implémenté à l'aide du module 'pygame' de Python avec les conditions suivantes :

- Un écran de jeu divisé en une grille de 12×12
- Une résolution du jeu réglée sur 252×252
- Une taille initiale du serpent de 3.

Logiquement, le contrôleur à quatre entrées pour naviguer, soit les flèches directionnelles haut, bas, gauche et droite. A savoir qu'en réalité il n'est possible d'aller qu'à droite, à gauche ou tout droit, il est impossible de revenir sur soi. Dans ce cas, deux implémentations possibles : cela entraîne une défaite ou alors l'action est nulle.

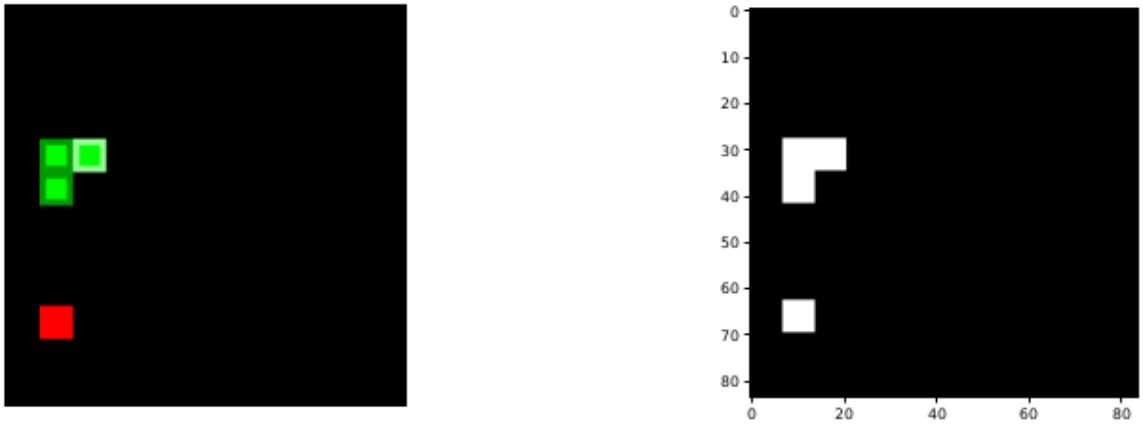
Apprentissage par renforcement profond

La méthode utilisée par les deux chercheurs est un algorithme de Deep Q-Learning (DQN) avec un réseau de neurones à deux couches entièrement connectées. L'utilisation d'un algorithme DQN est inévitable dans un contexte d'apprentissage par renforcement, par contre, le choix du réseaux de neurones est primordial pour optimiser la mémoire de rejeu. L'article utilise un CNN pour traiter des images issues du jeu Snake et approximer la fonction de valeur d'action. Les images dont il est question sont en réalité l'état actuel du jeu qui est perçu comme un tableau. Ces données sont alors envoyées au modèle afin que l'agent puisse effectuer le meilleur choix.

Prétraitement des images

L'agent obtient les valeurs RVB dans le format de tableau 3D à partir des environnements de jeux. Ce tableau RVB est converti en niveaux de gris afin de diminuer l'impact sur les performances, économisant ainsi trois fois la mémoire. Également, les données sont redimensionnéées en niveaux de gris 84×84 pixels pour limiter les coûts computationnels.

Niveaux de gris qui seront par la suite binarisés (prennent les valeurs 0 ou 1) pour une réduction de mémoire plus importante.



(a) Avant prétraitement **(b)** Après prétraitement

Figure 2: Données visuelles avant et après prétraitement

Fonction de perte et optimiseur

Afin d'améliorer la performance du modèle, cette implémentation calcule les valeurs Q et utilise une fonction de perte basée sur la fonction de Bellman pour ajuster les poids du réseau de neurones. Quant à l'optimiseur, Adam a été utilisé pour une convergence plus rapide du modèle. En effet, Adam permet d'ajuster les paramètres d'un réseau neuronal en temps réel pour améliorer sa précision et sa vitesse [3].

Mémoire tampon de rejeu d'expérience

Le but principal de la mémoire tampon de rejeu est de briser la corrélation temporelle des expériences vécues par l'agent, tout en augmentant l'efficacité de l'entraînement. L'agent, dans un environnement, suit une séquence de décisions (actions) qui sont corrélées entre elles. Si l'on entraîne directement le modèle à partir de ces expériences consécutives, le modèle peut sur-apprendre des séquences spécifiques et devenir instable ou converger vers une solution sous-optimale.

Ici, une mémoire tampon de rejeu de taille 50 000 est utilisée, nécessitant moins de mémoire (seulement 5%) que [6] et [7], qui utilisent une mémoire tampon de rejeu de taille 1 000 000. La mémoire tampon de rejeu stocke les données dans l'ordre FIFO (premier entré, premier sorti) afin que la mémoire tampon ne contient que les dernières données.

Q-Learning

Le Q-learning [5] est un algorithme populaire utilisé dans l'apprentissage par renforcement. Il s'agit d'un type d'algorithme sans modèle, ce qui signifie qu'il n'a pas besoin d'un modèle de l'environnement pour fonctionner. Au lieu de cela, il utilise une table Q pour suivre les récompenses pour différentes actions dans différents états.

L'agent utilise cet apprentissage pour décider du chemin à prendre. Il le fait en gardant une trace des récompenses pour différentes actions dans différents états. L'agent utilise ensuite ces informations pour prendre des décisions sur le chemin à suivre. Au fil du temps, l'agent met à jour sa table Q et comprend mieux quelles actions conduisent aux récompenses les plus élevées.

$$\text{New } Q(S,A) = Q(S,A) + \alpha \left[R(S,A) + \gamma \cdot \max(Q(S',A')) - Q(S,A)\right] \tag{1}$$

- $Q(S,A)$: La Q-value actuelle pour l'état **S** et l'action **A**.
- α : Le taux d'apprentissage, qui détermine dans quelle mesure les nouvelles informations remplacent les anciennes.
- $R(S,A)$: La récompense immédiate obtenue après avoir pris l'action **A** dans l'état **S**.
- γ : Le facteur de discount ou taux d'actualisation, qui mesure l'importance des récompenses futures par rapport aux récompenses immédiates.
- $max(Q(S',A'))$: La récompense future maximale attendue pour le nouvel état **S'**.

Cette équation de Bellman permet à l'agent d'ajuster ses décisions en fonction des récompenses qu'il reçoit et de la valeur attendue des futures actions.

Il faut savoir que le Q-Learning est un algorithme puissant et il est largement utilisé dans de nombreuses applications telles que la robotique, l'IA de jeu et les voitures autonomes.L'un des principaux avantages de ce type d'apprentissage est qu'il peut être utilisé dans des environnements où la dynamique est inconnue ou non stationnaire.

Voici comment cet algorithme fonctionne dans ce contexte. Les états sont représentés par le symbole S. Ici, un état correspond à l'ensemble des informations importantes à un moment donné, comme :

- La position de la tête du serpent.
- La position de la pomme (la nourriture).
- La présence de dangers autour du serpent, comme le mur ou le corps du serpent lui-même.

Ces informations permettent à l'agent de savoir ce qui se passe dans l'environnement à chaque instant et de décider de la meilleure action à entreprendre.

Les actions, notées A, sont les choix que le serpent peut faire à chaque étape. Dans notre cas, le serpent a trois actions possibles : Aller à droite, à gauche ou continuer tout droit. Ces actions permettent à l'agent de naviguer dans l'environnement et de tenter de maximiser son score en mangeant des pommes et en évitant les obstacles.

L'environnement change à chaque fois que le serpent effectue une action :

- Si le serpent mange une pomme, son score augmente et un nouvel état est atteint.
- Si le serpent se prend un mur ou se mord lui-même, la partie se termine.
- Sinon, le serpent continue de se déplacer dans la direction choisie.

L'objectif du Q-learning est d'apprendre à l'agent à maximiser ses récompenses au fil du temps. Cela signifie que l'agent doit prendre des décisions qui l'aideront à atteindre le meilleur résultat possible (manger le plus de pommes tout en évitant de mourir).

Mécanisme de récompense

Le système de récompense est basé comme suit :

- Manger une pomme : +1
- Collision : -1
- Déplacement sans manger de pomme ou avoir de collision : -0.1

Ce système de récompense incite l'agent à réduire le nombre de déplacements inutiles pour maximiser le fait de manger des pommes. De plus, bien que cela ne soit pas précisé dans l'article, nous supposons que la balance entre le fait de manger une pomme et d'avoir une collision est équilibrée pour éviter d'inciter le modèle à favoriser une action plutôt qu'une autre.

Analyse des résultats

Comme évoqué plus haut, la réduction de dimensionnalité des images (donc de l'état de jeu) représente la première stratégie de préservation de la mémoire. Pour cela, nous avons **Table 1** pour nous confirmer ce résultat. En effet, binariser les images après réduction de dimension des données d'entrées du CNN en 84 x 84 aura permis de préserver 99.7% de la mémoire, là où transformer en niveaux de gris n'en préserve de 67%.

Table 1: Mémoire sauvegardée

	RVB	**Niveaux de gris**	**Binaire**
Utilisation de la mémoire	1267.71	420.57	2.628
Mémoire sauvegardée v-à-v du RVB	0%	67%	99.7%
Mémoire sauvegardée v-à-v du niveaux de gris	-	0%	99.4%

Au-delà de la mémoire préservée, il reste intéressant de s'intéresser aux performances du modèle car l'objectif est d'optimiser, voire minimiser, l'utilisation de la mémoire tout en préservant la performance. Pour cela, **Table 2** montre une comparaison entre la moyenne et le record de plusieurs agents, notamment par un humain, le modèle de [7] et celui de [1]. Les modèles proposés par les IA restent bien meilleurs en moyenne, mais [1] surpasse [7] de peu, mais reste tout de même plus performant.

Table 2: Comparaison des performances de différents agents

Performance	**Score**
Moyenne humaine	1.98
Moyenne refined DQN	9.04
Moyenne du modèle	9.53
Record humain	15
Record refined DQN [7]	17
Record du modèle	20

5 Expérimentation

L'expérimentation que nous avons menée se base sur un code open source [4]. Elle propose une version moins complexe et plus légère que l'approche de l'article [1], en mettant l'accent sur l'efficacité de la mémoire et la simplicité de l'implémentation, tout en cherchant à rester compétitif en termes de performances sans utiliser de CNN.

L'objectif de cette expérimentation est d'évaluer les performances d'un modèle utilisant un réseau de neurones entièrement connecté (nommé **Linear_QNet**) pour jouer au jeu Snake, tout en visant à obtenir des résultats au moins similaires à ceux obtenus avec un CNN. Voici les détails de notre approche expérimentale ainsi que les ajustements testés pour améliorer les performances de l'agent.

Modèle

Notre méthode repose sur un modèle de réseau de neurones entièrement connecté (Linear_QNet), dépourvu de couches convolutionnelles, ce qui le rend plus simple et mieux adapté aux données tabulaires ou aux états numériques du jeu. Contrairement aux modèles basés sur les CNN, ce modèle n'effectue pas de traitement d'image ni de conversion en données binaires, simplifiant ainsi considérablement les calculs.

Les données d'entrée du modèle proviennent directement des caractéristiques numériques du jeu Snake, telles que la position de la tête du serpent, les directions possibles, et la position relative de la nourriture. L'absence de traitement d'image permet de réduire les coûts computationnels et d'améliorer l'efficacité en mémoire.

Après plusieurs expérimentations, nous avons fixé la taille de notre mémoire tampon (**replay buffer**) à une valeur maximale de 50 000 expériences ($MAX\_MEMORY = 50\ 000$). Cette approche permet d'améliorer la stabilité de l'apprentissage en échantillonnant aléatoirement les expériences passées. L'entraînement repose sur un seul réseau Q, avec une optimisation classique utilisant l'optimiseur **Adam** et une fonction de perte de type **Mean Squared Error** (MSE).

6 Résultats et Discussions

Tentatives d'améliorations

Pour commencer, nous avons tenté de modifier les inputs, c'est-à-dire les paramètres d'entrée. Nous avons constaté que le serpent mourait toujours de la même façon : lorsqu'il rencontrait un danger en face, sans danger immédiat à droite ni à gauche, il se dirigeait systématiquement vers la pomme. Cependant, il pouvait rencontrer un danger plus loin dans cette direction. Nous avons donc décidé de supprimer les paramètres indiquant la présence de dangers sur les cases adjacentes et de les remplacer par la distance au danger le plus proche.

Malheureusement, cela a considérablement réduit la performance de l'intelligence artificielle (IA), qui n'apprenait plus rien. Au bout d'une quarantaine d'itérations, le serpent se mettait à tourner sur lui-même pendant des centaines de tours. Pour éviter ce comportement, nous avons modifié les récompenses afin de pénaliser le serpent à chaque pas, espérant ainsi l'empêcher de tourner en rond.

Cependant, cette modification n'a eu aucun effet : le serpent continuait à tourner en rond sans chercher les récompenses, minimisant ainsi sa récompense, ce qui est contre-intuitif.

Nous avons ensuite ajouté davantage de pommes pour augmenter les chances que l'IA en trouve par hasard, dans l'espoir qu'elle comprenne progressivement ce qu'il fallait faire. Mais cela n'a rien changé : le serpent restait focalisé sur la minimisation de sa récompense.

Face à ce problème, nous avons envisagé que le modèle n'était peut-être pas assez complexe. Nous avons donc augmenté le nombre de couches cachées et changé la fonction d'activation, passant de ReLU à la sigmoïde. Malheureusement, cela n'a eu aucun impact.

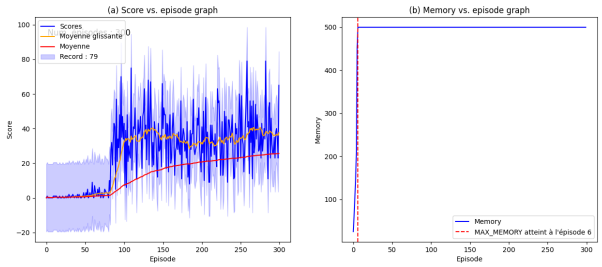
Enfin, dans une dernière tentative, nous avons réduit toutes les valeurs d'inputs entre 0 et 1, et inversé la distance pour que les valeurs soient plus élevées lorsque le danger était proche, le rendant ainsi plus menaçant.

Pour autant, aucune de ces améliorations n'a porté ses fruits, même après avoir ajusté le learning rate.

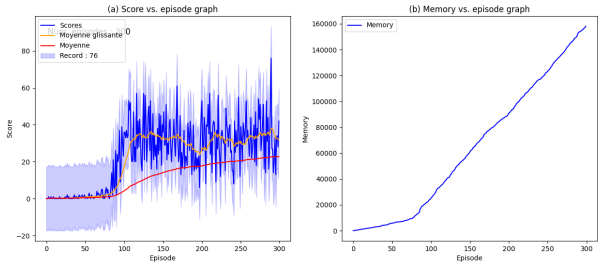
Dans le cadre de notre expérimentation, nous avons fait varier la taille de la mémoire tampon (MAX_MERORY) entre 500 et 300 000 pour observer son impact sur les performances de l'agent. L'objectif était de déterminer si une augmentation ou une réduction significative de la taille de la mémoire affecterait le processus d'apprentissage et les performances globales du modèle.

Cependant, après avoir testé plusieurs configurations de taille de mémoire dans cette

fourchette, nous n'avons observé aucune différence notable en termes de performance ou de score final obtenu par l'agent. Que la mémoire soit relativement grande (300 000) ou plus petite (500), les résultats étaient comparables. Cela suggère que dans ce contexte particulier du jeu Snake, la taille de la mémoire n'a pas d'impact majeur sur la capacité de l'agent à apprendre et à jouer efficacement.



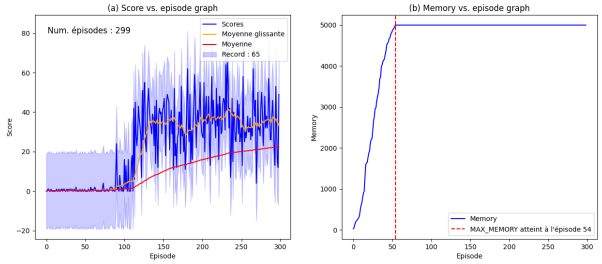
(a) Mémoire tampon = 500



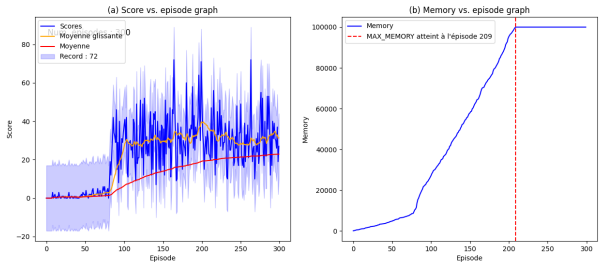
(b) Mémoire tampon = 300 000

Figure 3: Résultats pour différentes valeurs de mémoire tampon

Cette observation peut s'expliquer par le fait que l'environnement du jeu Snake est relativement simple et que l'agent n'a pas besoin d'une mémoire très étendue pour apprendre des comportements optimaux. Il est possible qu'au-delà d'une certaine taille, l'ajout de plus de données d'expérience n'apporte pas d'informations supplémentaires significatives pour améliorer l'apprentissage.



(a) Mémoire tampon = 5 000



(b) Mémoire tampon = 100 000

Figure 4: Résultats pour différentes valeurs de mémoire tampon

Egalement, nous avons modifié le code pour empêcher l'agent de faire demi-tour instantanément. En d'autres termes, l'agent ne peut pas appuyer sur la direction opposée (par exemple, appuyer sur "gauche" lorsqu'il se dirige vers la droite). Cette modification a permis d'améliorer les résultats, car le modèle n'a plus besoin d'apprendre à éviter ce cas d'erreur évident, simplifiant ainsi la tâche d'apprentissage.

Propositions d'améliorations

Nous pensons qu'il existe une solution plus efficace que celle initialement proposée, et nos suggestions répondaient en partie à ces attentes. Toutefois, nous n'avons pas eu le temps de tester suffisamment les hyperparamètres et les fonctions d'activation pour trouver la solution optimale. Nous regrettons de ne pas avoir exploré certaines pistes supplémentaires.

Par exemple, nous aurions pu expérimenter avec la taille et la forme du tableau pour que le serpent s'entraîne sur des terrains variés, et qu'il soit plus souvent confronté à des situations complexes. De plus, nous avons peut-être vu trop grand dès le départ ; il aurait été intéressant de conserver le système de vision à un bloc de distance mais de l'améliorer en ajoutant quatre paramètres offrant une vision à deux blocs de distance, voire plus.

7 Conclusion

Dans ce projet, nous avons exploré des approches optimisées de l'apprentissage par renforcement profond pour la création d'agents autonomes jouant au jeu du Snake tout en réduisant l'utilisation de la mémoire. Nos résultats montrent qu'il est possible d'atteindre des performances comparables à celles des méthodes plus gourmandes en ressources, en particulier celles basées sur des CNN, en adoptant des stratégies plus légères telles que la réduction de la taille du buffer de rejeu et l'utilisation d'un réseau de neurones entièrement connecté.

La réduction de la complexité des modèles et l'optimisation des ressources mémorielles sont des facteurs cruciaux, notamment dans des environnements à ressources limitées tels que les dispositifs mobiles. Bien que certaines tentatives d'amélioration, telles que la modification des entrées et des récompenses, n'aient pas abouti aux résultats escomptés, elles ont ouvert la voie à d'autres pistes d'exploration, telles que l'optimisation des hyper-paramètres et l'élargissement de la vision de l'agent.

En somme, ce travail prouve qu'une réduction significative de l'utilisation de la mémoire est possible sans sacrifier les performances. Toutefois, des recherches futures sont nécessaires pour affiner ces méthodes, en particulier en ajustant davantage les paramètres du modèle et en testant sur des environnements plus variés.

References

[1] A Memory Efficient Deep Reinforcement Learning Approach For Snake Game Autonomous Agents. (s. d.), https://arxiv.org/abs/2301.11977.

[2] Contributeurs aux projets Wikimedia. (2006, 23 octobre). Snake (genre de jeu vidéo) - Wikipédia. Wikipédia, l'encyclopédie libre, https://fr.wikipedia.org/wiki/Snake_(genre_de_jeu_vidéo).

[3] What is the Adam Optimizer and How is It Used in Machine Learning. (s. d.). Artificial Intelligence +, https://www.aiplusinfo.com/blog/what-is-the-adam-optimizer-and-how-is-it-used-in-machine-learning/.

[4] Vizuara. (2024, 24 juillet). Teach Neural Network to play snake game | Reinforcement Learning | Reward maximization | AI gameplay [Vidéo]. YouTube, https://www.youtube.com/watch?v=03zDwIzRPQI.

[5] Mash'Al, Y. (2023, 17 janvier). What is Q-learning ? Medium, https://medium.com/@YazanMashal/what-is-q-learning-d29401f4eaf3.

[6] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, ser. AAAI'16. AAAI Press, 2016, p. 2094–2100.

[7] Z. Wei, D. Wang, M. Zhang, A.-H. Tan, C. Miao, and Y. Zhou, "Autonomous agents in snake game via deep reinforcement learning," in 2018 IEEE International Conference on Agents (ICA), 2018, pp. 20–25.