

Rapport Deep Reinforcement Learning

BLOCH Alexandre

BANDZOUZI NTADI Thomas

EL HARRAR Ethan

5IABD1 Nicolas VIDAL

Lien Github : <https://github.com/alexblch/Projet-DRL-2.git>

**Table des matières :**

**INTRODUCTION**

1. **Présentation des algos de renforcements**
2. **Le Lucky number**

**A. Le jeu avec interface graphique**

**B. Le random vs random**

**C. L'environnement**

1. **Analyse des entraînements**

**A. TD Learning**

**B. REINFORCE et PPO A2C**

**C. MCTS et RANDOM ROLLOUT**

1. **Synthèse**

**CONCLUSION**

**INTRODUCTION**

L’apprentissage par renforcement (Reinforcement Learning, RL) est une branche du machine learning où un agent apprend à interagir avec son environnement pour maximiser une récompense cumulative. Contrairement à l’apprentissage supervisé, dans le RL, les données ne sont pas fournies sous forme d’exemples étiquetés. L'agent doit explorer les actions possibles et en déduire les meilleures stratégies en fonction des retours (récompenses ou pénalités) reçus de l’environnement. Ce cadre est particulièrement adapté à des contextes dynamiques et incertains, comme les jeux vidéo, la robotique ou encore la gestion des stocks, où les décisions successives ont un impact cumulatif sur les résultats futurs.

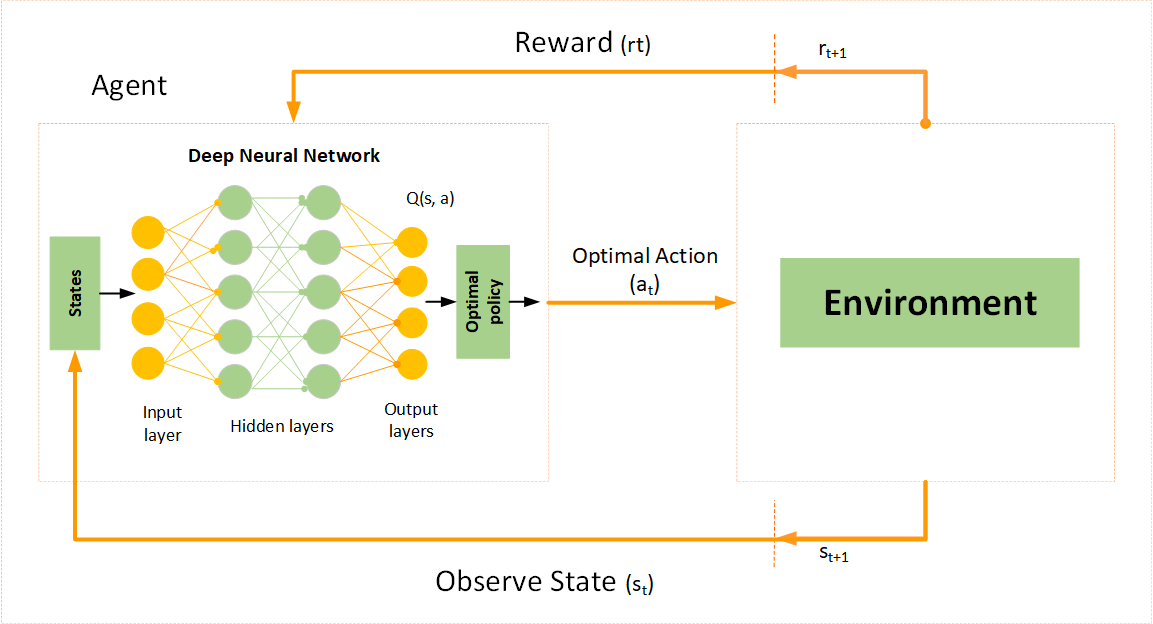
Cependant, le RL seul peut être limité dans des environnements complexes où une exploration aveugle devient coûteuse en termes de temps et de ressources. C’est ici que l’apprentissage supervisé entre en jeu : il peut être utilisé pour guider l’agent en lui apprenant des modèles ou des comportements optimaux à partir de données existantes. Par exemple, un réseau de neurones formé en supervision peut prédire les valeurs ou probabilités d’actions optimales, réduisant ainsi l’espace d’exploration de l’agent RL. Cette combinaison hybride, souvent appelée deep reinforcement learning, permet de tirer parti de la puissance des deux approches pour atteindre des performances plus rapides et plus fiables dans des environnements complexes.

Ainsi, de nouveaux algorithmes puissants ont émergé, combinant les principes du reinforcement learning avec les capacités des réseaux de neurones profonds. Parmi eux, **les méthodes TD Learning (Temporal Difference)**, telles que **DQN (Deep Q-Network)** et **Double DQN**, permettent d'estimer les valeurs des actions tout en réduisant les biais d'estimation associés aux approches traditionnelles. Ces algorithmes exploitent des réseaux de neurones pour approximativement modéliser les fonctions de valeur dans des environnements complexes. De leur côté, les algorithmes basés sur **REINFORCE** et **PPO (Proximal Policy Optimization)** se concentrent sur l’optimisation des politiques directement, offrant une approche **on-policy** où les mises à jour se font à partir de données générées par la politique actuelle de l'agent.

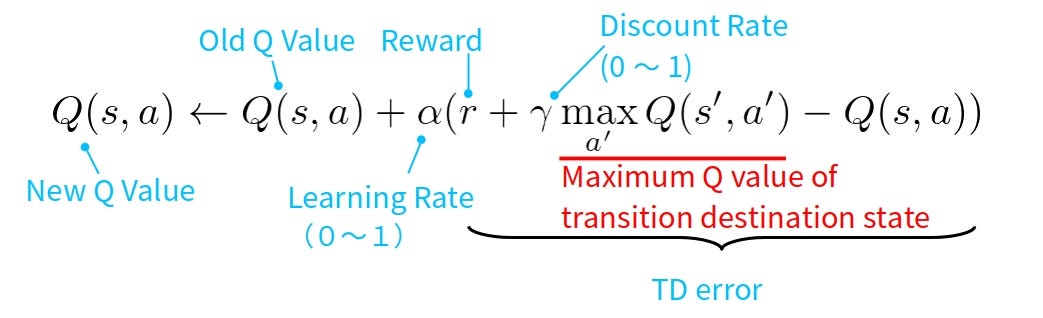
Ces avancées ont culminé avec des approches comme les arbres de recherche de Monte-Carlo (MCTS), intégrés dans des algorithmes tels qu’**AlphaZero** et **MuZero**. AlphaZero a marqué une révolution en combinant MCTS avec des réseaux de neurones pour exceller dans des jeux comme les échecs ou le Go, sans connaissances préalables spécifiques aux règles. MuZero, encore plus sophistiqué, a été conçu pour apprendre les dynamiques de l’environnement de manière implicite, ce qui le rend adaptable à des contextes variés. Ces innovations montrent à quel point le deep reinforcement learning a transformé l’apprentissage des agents intelligents, rendant possible une performance de haut niveau dans des environnements de plus en plus complexes et variés.

1. **Présentation des algos de renforcements**

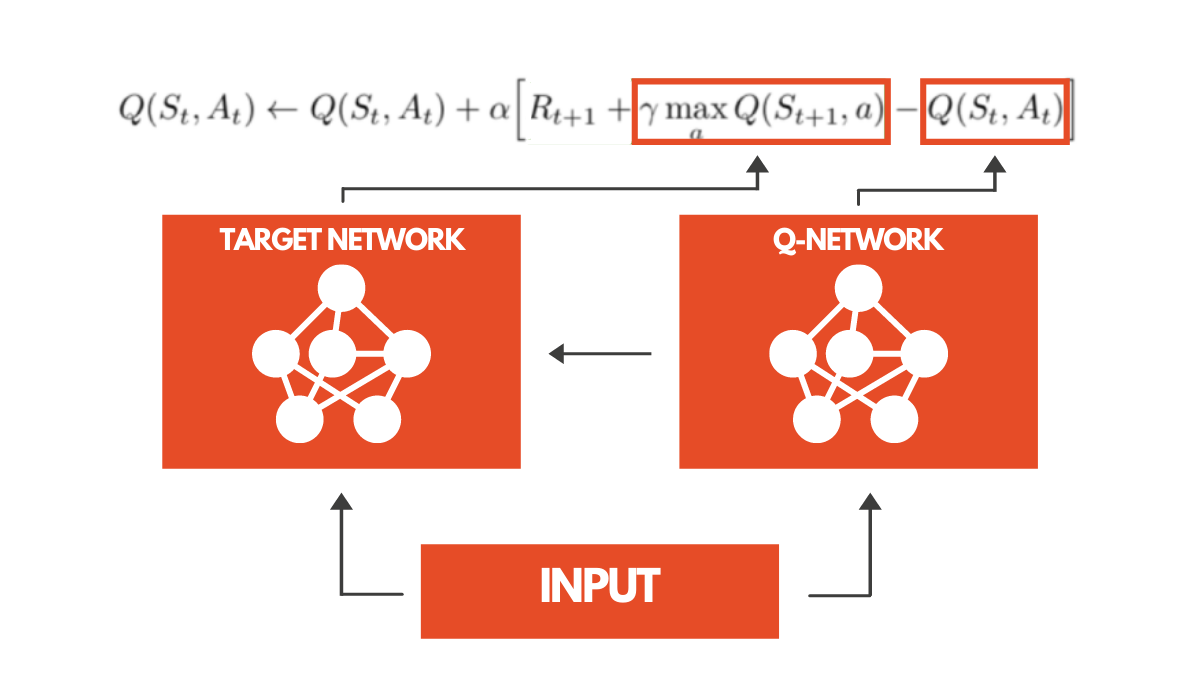
Le Deep Q-Network (DQN) est un algorithme combine l’apprentissage par renforcement avec les réseaux de neurones pour résoudre des problèmes où l’espace d’état est complexe et de grande dimension. Il s’appuie sur l’approche classique du Q-learning, mais remplace la table de Q-valeurs par un réseau de neurones qui approxime la fonction de Q-valeur Q(s,a), laquelle prédit la valeur d’une action a dans un état s.

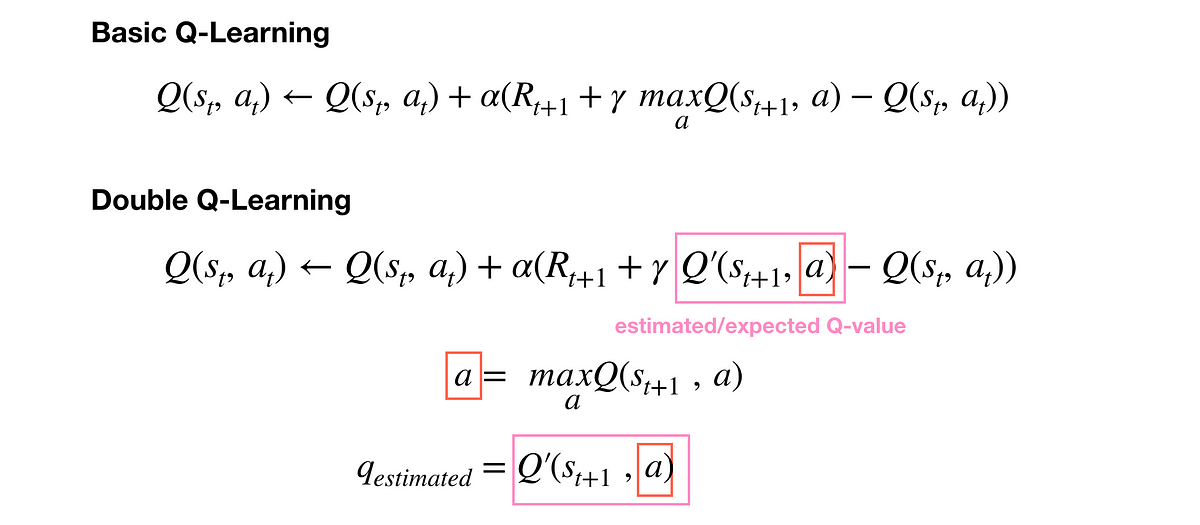


Le dqn va donc prendre l’état et retourner Q(s,ai) pour n actions allant de 0 à n – 1 contrairement au Q learning qui utilisait la formule suivante :

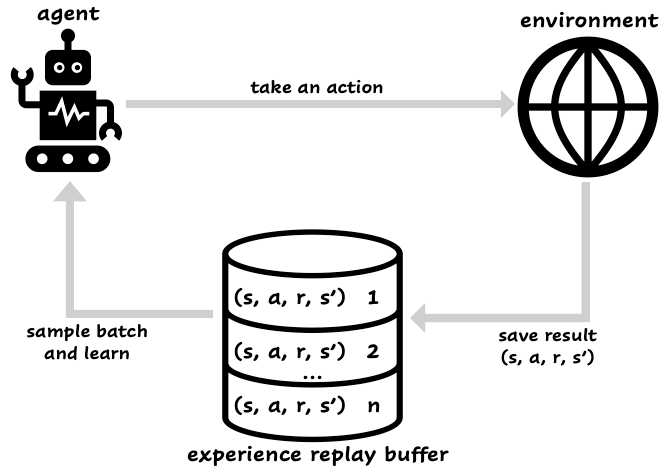


Le **Double Deep Q-Network (Double DQN)** est une extension du DQN qui repose sur l'utilisation de deux réseaux de neurones distincts pour améliorer la précision des estimations de Q-valeurs. Contrairement au DQN classique, où un seul réseau de neurones est utilisé à la fois pour sélectionner et évaluer les actions, le Double DQN divise ces responsabilités entre deux réseaux, réduisant ainsi le biais de surestimation des Q-valeurs.





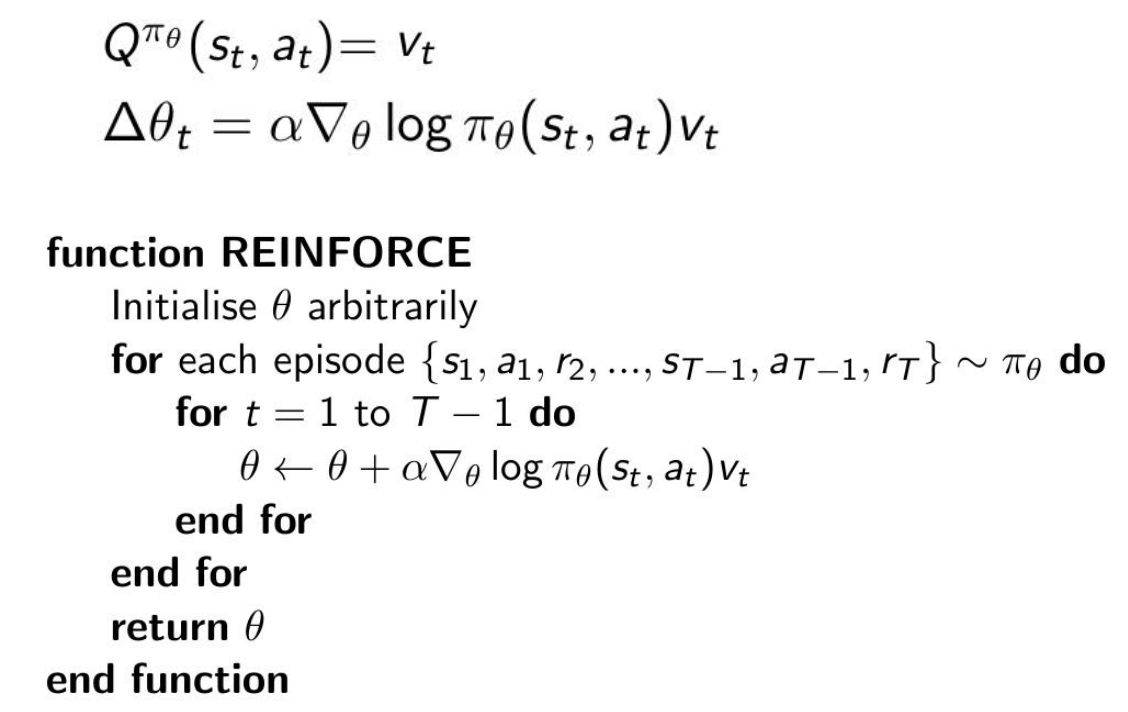
L’**Experience Replay** est une technique clé utilisée dans les algorithmes de deep reinforcement learning, comme le DQN, pour améliorer l’efficacité et la stabilité de l’apprentissage. Plutôt que d'utiliser directement les transitions (s,a,r,s′)(s, a, r, s')(s,a,r,s′) générées par l’agent à chaque étape, celles-ci sont stockées dans une mémoire tampon (replay buffer). Lors de l’apprentissage, des échantillons sont prélevés aléatoirement dans cette mémoire pour entraîner le réseau de neurones.



Le **Prioritized Experience Replay** (PER) est une extension de l’Experience Replay classique, conçue pour améliorer encore l’efficacité de l’apprentissage en donnant la priorité aux transitions les plus importantes. Plutôt que de tirer les étiquettes uniforme, le PER attribue une probabilité d'échantillonnage à chaque transition en fonction de sa **priorité**, qui reflète son importance pour l’apprentissage.

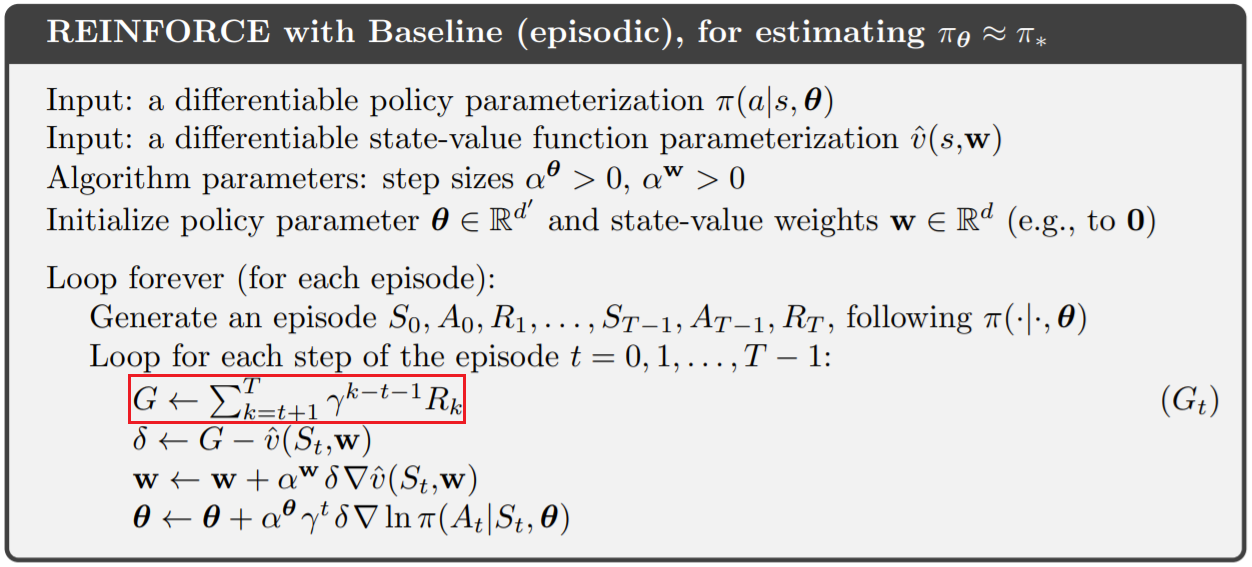
L’algorithme **REINFORCE** est une méthode d’apprentissage par renforcement qui appartient à la famille des policy gradients. Son objectif principal est d’apprendre directement une politique, c’est-à-dire une stratégie qui indique à l’agent quelle action effectuer dans un état donné, afin de maximiser la récompense totale qu’il peut obtenir sur le long terme. Contrairement à d’autres algorithmes qui estiment une fonction de valeur pour choisir les actions, REINFORCE se concentre uniquement sur l’optimisation de la politique.

Le fonctionnement repose sur la collecte d’expériences où l’agent interagit avec l’environnement en suivant une politique paramétrée, généralement représentée par un réseau de neurones. À la fin de chaque épisode, l’algorithme évalue les actions prises en fonction des récompenses obtenues et ajuste les paramètres de la politique pour augmenter la probabilité de choisir des actions ayant conduit à de bonnes récompenses. L’apprentissage repose sur une estimation des retours liés à chaque action, ce qui peut rendre le processus parfois instable à cause de la variance élevée des mises à jour.



Le **REINFORCE avec baseline** est une amélioration de l’algorithme REINFORCE visant à réduire la variance des mises à jour tout en conservant un biais nul dans l’apprentissage. Cette méthode introduit une **baseline**, une valeur de référence qui est soustraite à la récompense avant de mettre à jour les paramètres de la politique. La baseline agit comme un point de comparaison : seules les actions qui ont généré des récompenses supérieures à cette référence augmentent significativement la probabilité d’être sélectionnées, tandis que celles inférieures sont moins renforcées.

En pratique, cette baseline est souvent approximée par une fonction de valeur, qui estime la récompense attendue dans un état donné. Cela permet de mieux contextualiser les performances d’une action en tenant compte de ce qui est considéré comme "normal" pour un état donné, réduisant ainsi les fluctuations inutiles dans les mises à jour. Cette amélioration rend l’apprentissage plus stable et rapide, sans modifier les principes fondamentaux de l’algorithme REINFORCE.

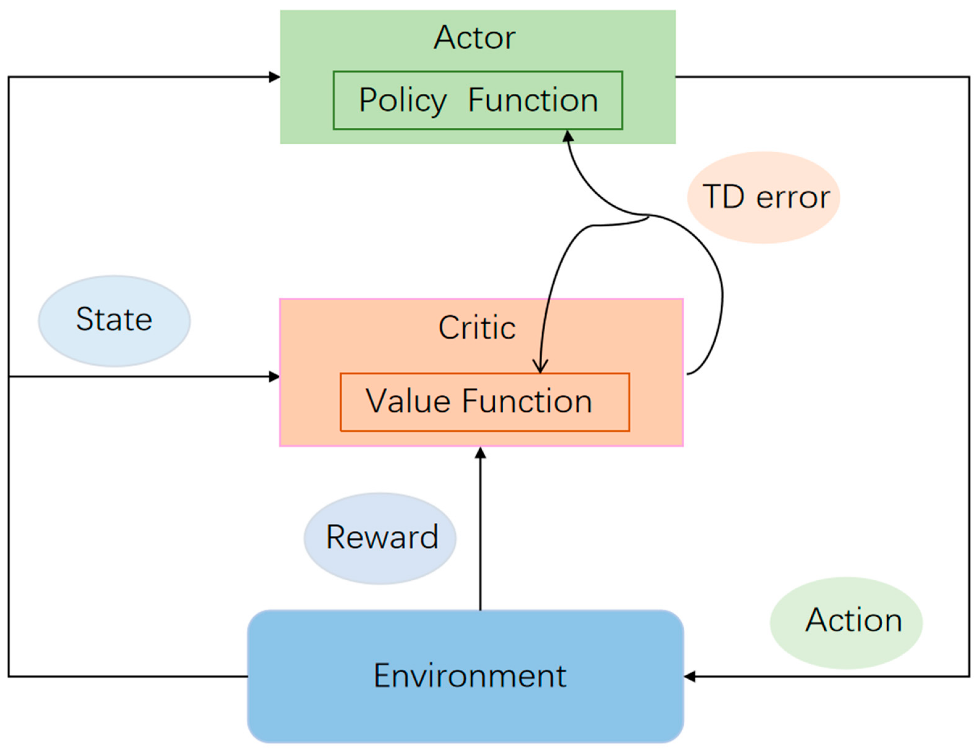


Le **REINFORCE avec critic** va encore plus loin que le REINFORCE avec baseline en introduisant un **critic**, une fonction qui estime directement la valeur de chaque état ou l'avantage d'une action dans un état donné. Contrairement à la baseline qui est simplement une valeur de référence, le critic joue un rôle actif en fournissant une estimation plus précise de l’impact de chaque action sur la récompense future.

Cette approche, souvent appelée **Actor-Critic**, sépare le modèle en deux parties : l'**actor**, qui décide des actions en suivant une politique, et le **critic**, qui évalue la qualité des décisions prises par l'actor. Grâce à cette évaluation, l'actor peut ajuster ses paramètres de manière plus fine en fonction des retours du critic, réduisant encore davantage la variance des mises à jour par rapport à REINFORCE standard. Le critic améliore également l’efficacité dans les environnements où les retours de récompense sont plus complexes ou différés. Cette structure combinée permet un apprentissage plus rapide et plus stable dans des environnements dynamiques.

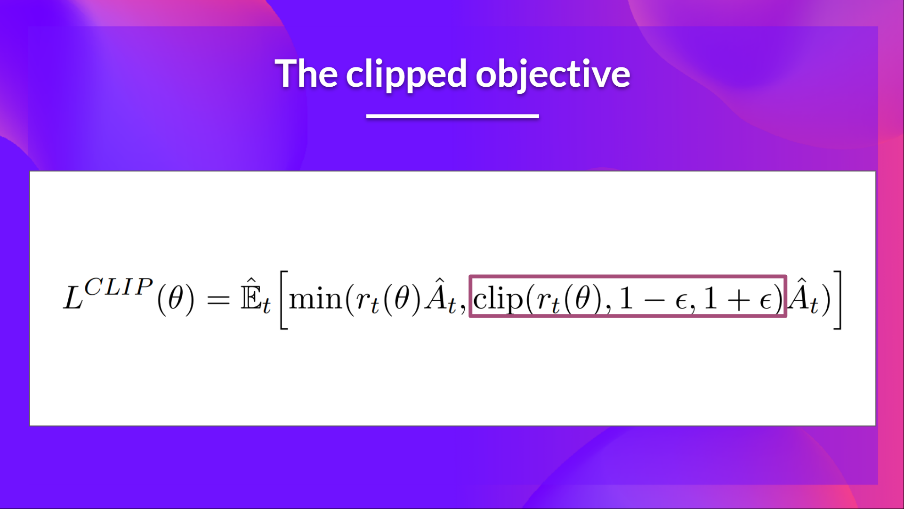
L’actor et le critic interagissent en utilisant des mises à jour par **ascension** et **descente de gradient stochastique**, respectivement, pour optimiser leurs fonctions respectives.

* **Ascension de gradient pour l'actor** : L'objectif de l'actor est d'améliorer sa politique en maximisant les récompenses futures. Il utilise l'estimation fournie par le critic (généralement sous forme d'avantage ou de différence temporelle) pour ajuster ses paramètres de manière à augmenter la probabilité des actions qui mènent à de bonnes récompenses. Cela correspond à une ascension de gradient stochastique, où l'actor se dirige vers des politiques plus optimales.
* **Descente de gradient pour le critic** : Le critic, quant à lui, cherche à minimiser l’erreur entre la valeur estimée et la valeur réelle des récompenses observées. Cette minimisation est effectuée via une descente de gradient stochastique sur la fonction de perte associée à la fonction de valeur (par exemple, une perte quadratique). Cette étape assure que l’évaluation des actions par le critic reste fiable et cohérente au fil de l’apprentissage.



Le **Proximal Policy Optimization (PPO)** est une extension des méthodes Actor-Critic, comme l'Advantage Actor-Critic (A2C), qui se distingue par sa stabilité et son efficacité dans l’apprentissage. Par rapport à l’A2C, où l’actor met à jour sa politique en utilisant directement les gradients calculés à partir des retours du critic, le PPO introduit une contrainte supplémentaire pour éviter que les mises à jour de la politique ne soient trop brusques, ce qui pourrait dégrader les performances.

Dans PPO, une **fonction d'objectif "clippée"** est utilisée pour limiter les modifications apportées à la politique, garantissant que les probabilités des actions ne changent pas de manière excessive entre deux mises à jour. Cela empêche l’actor d’effectuer des ajustements trop importants, même si le critic le suggère, rendant ainsi l’apprentissage plus stable. Contrairement à l’A2C, où l’actor et le critic travaillent de manière plus directe, le PPO équilibre mieux la vitesse d’apprentissage et la précision en contrôlant explicitement la manière dont la politique est mise à jour. Cette amélioration rend PPO particulièrement efficace dans des environnements complexes où les politiques doivent être ajustées progressivement.

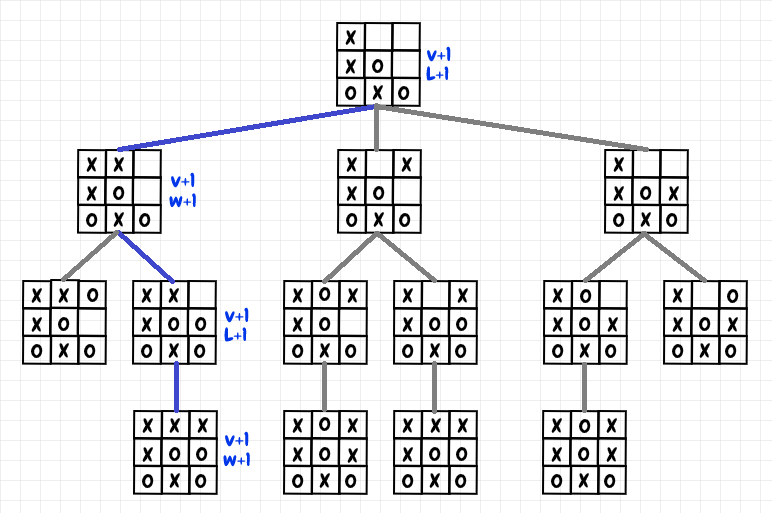


Le **random rollout** est une méthode utilisée dans les algorithmes d’apprentissage par renforcement, en particulier dans les approches basées sur la recherche, comme les **arbres de recherche Monte-Carlo (MCTS)**. Il s'agit d'une technique où, à partir d'un état donné, l'agent exécute une séquence d'actions choisies aléatoirement jusqu'à atteindre une condition d'arrêt, comme la fin de l'épisode ou une profondeur maximale. L'objectif est d'estimer la valeur de cet état en observant les récompenses obtenues lors de ces simulations.

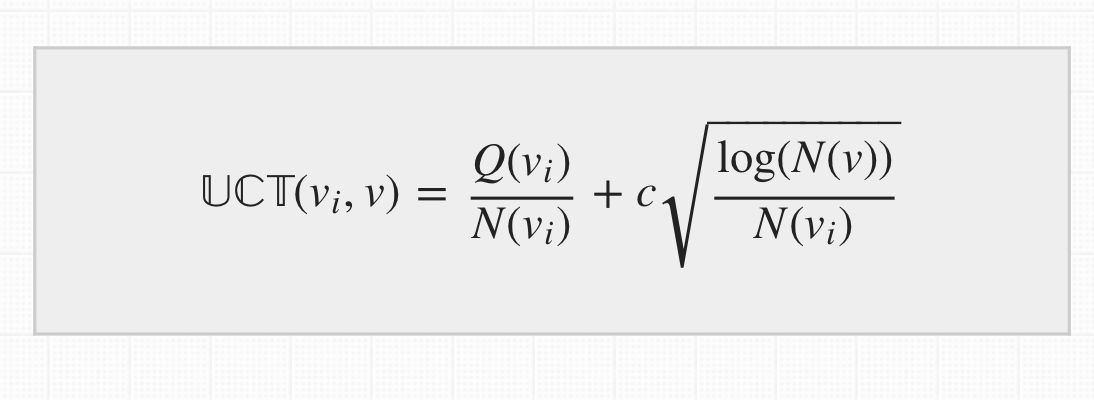
L'algorithme **Monte Carlo Tree Search (MCTS)** est une méthode de recherche basée sur la simulation, largement utilisée pour prendre des décisions dans des environnements avec un espace d'actions complexe. Il explore un arbre de décisions en simulant des trajectoires possibles et en évaluant les résultats obtenus, permettant ainsi de sélectionner l’action optimale. Il existe deux variantes principales de MCTS, différenciées par la manière dont elles estiment la qualité des états ou actions : **l'approche avec random rollout** et celle utilisant des **réseaux de neurones**.

Dans cette version, MCTS s'appuie sur des **simulations aléatoires** pour évaluer les nœuds de l’arbre. Lorsqu'un nœud est atteint, des random rollouts sont exécutés à partir de cet état : l'agent choisit des actions aléatoires jusqu’à une condition d’arrêt (fin d’épisode ou profondeur maximale). Les récompenses obtenues lors de ces rollouts sont ensuite agrégées pour estimer la valeur de l'état ou de l'action associée au nœud. Bien que cette méthode soit simple et efficace dans certains contextes, elle souffre de limitations, notamment une précision réduite dans des environnements où des trajectoires aléatoires sont peu représentatives de la qualité réelle des décisions.

Dans cette variante plus sophistiquée, MCTS utilise un **réseau de neurones** pour prédire directement la valeur des nœuds et/ou la probabilité des actions à chaque étape. Le réseau est généralement entraîné en parallèle, en utilisant des données collectées lors des simulations. Par exemple, dans des algorithmes comme Alpha Go ou Alpha Zéro, le réseau est utilisé pour estimer la valeur d'un état (fonction de valeur) et pour guider la sélection des actions (politique). Cela permet de remplacer les random rollouts par des prédictions plus informées, ce qui accélère la recherche et améliore la précision des évaluations. Cette approche est particulièrement puissante dans des environnements complexes, mais elle nécessite des ressources computationnelles importantes pour entraîner et exécuter le réseau.



L’équilibre entre l’exploration et l’exploitation dans l’algorithme **MCTS avec réseau de neurones** est principalement déterminé par l’**UCT (Upper Confidence Bound for Trees)**. L’UCT est une formule qui guide la sélection des nœuds dans l’arbre de recherche en tenant compte à la fois de la qualité d’une action (exploitation) et de son degré d’exploration (le nombre de fois où cette action a été visitée). Cet équilibre est crucial pour optimiser la recherche.



**AlphaZero** est un algorithme d’apprentissage par renforcement développé par DeepMind, qui combine le **Monte Carlo Tree Search (MCTS)** et les réseaux de neurones pour atteindre des performances de haut niveau dans des jeux comme les échecs, le shogi et le go. Contrairement à ses prédécesseurs, il apprend à jouer sans connaissance humaine, uniquement à partir des règles du jeu, en s’appuyant sur l’auto-entraînement.

AlphaZero utilise le **MCTS** pour explorer les actions possibles à partir d’un état donné, tout en intégrant un réseau de neurones pour guider cette exploration. Le MCTS génère des simulations basées sur les actions, et ces simulations sont utilisées pour affiner la politique (stratégie) de jeu et la fonction de valeur (qualité d’un état). Ces deux éléments sont appris simultanément par le réseau.

Le dernier algorithme, le Muzero, développé aussi par Google DeepMind, combine le **Monte Carlo Tree Search (MCTS)** et les réseaux de neurones, tout en introduisant une capacité unique : il apprend à modéliser les dynamiques de l’environnement sans connaissance explicite des règles. Contrairement à AlphaZero, qui nécessite une connaissance préalable des transitions et des récompenses associées aux jeux, MuZero apprend à prédire ces informations à partir de l’expérience.

1. **Le Lucky number**

Nous avons choisi d’utiliser le jeu **Lucky Number** parmi les jeux proposés, car il représente un compromis idéal entre simplicité et complexité. Contrairement à certains jeux trop élémentaires, il demande à l’agent de mobiliser une certaine logique mathématique pour organiser les nombres de manière optimale. En même temps, il n’est pas excessivement difficile, ce qui permet de tester et d’implémenter efficacement des algorithmes de reinforcement learning tout en maintenant une courbe d’apprentissage raisonnable. Cette caractéristique en fait un terrain d’expérimentation pertinent pour évaluer les performances des agents dans un cadre à la fois stimulant et accessible.

Dans le jeu **Lucky Number**, l’objectif est de remplir une grille en plaçant des nombres tirés aléatoirement tout en respectant une contrainte stricte : les nombres doivent être disposés dans un ordre croissant, à la fois horizontalement (ligne par ligne) et verticalement (colonne par colonne). À chaque tour, un nombre est proposé à l’agent, qui doit choisir soit de le placer dans une tuile disponible de la grille, soit de le **rejeter dans une pile dédiée**. Cependant, les nombres rejetés ne peuvent plus être utilisés ultérieurement, ce qui rend chaque décision cruciale pour l’organisation future de la grille.

En comparaison, une stratégie **aléatoire** (random) place les nombres proposés dans les tuiles libres ou les rejette sans considérer les conséquences à long terme. Cette approche aboutit souvent à des grilles invalides ou incomplètes, car aucune anticipation n’est réalisée pour maximiser les opportunités futures. Ce contraste met en évidence la complexité et la logique nécessaires dans Lucky Number : un agent optimisé doit non seulement planifier ses placements pour maintenir la validité de la grille, mais aussi gérer judicieusement les rejets, équilibrant ainsi risque et opportunité pour maximiser ses chances de succès.

1. **Le jeu avec interface Graphique**

A l’ade de Tkinter, pour créer des jeux ou des applications simples en python, nous avons réussi à pouvoir jouer contre un agent random au lucky number tout en voyant les deux grilles avec au début les grilles initialisés avec les diagonales :

Une image contenant texte, capture d’écran, diagramme, carré

Description générée automatiquement

En cas de victoire, le jeu affiche une petite fenêtre marquée victoire :

Une image contenant texte, capture d’écran, diagramme, carré

Description générée automatiquement

1. **Lucky number random vs random**

Dans l'environnement **Lucky Number**, une fois les états et actions correctement définis, une première simulation a été réalisée où deux agents aléatoires s'affrontaient. Ces agents utilisent une stratégie entièrement basée sur des décisions aléatoires. Ils choisissent, à chaque tour, entre piocher une tuile, prendre une tuile du cache partagé, ou placer une tuile sur leur grille, sans aucune planification ni raisonnement stratégique, le reward est de 0 s’il se trompe de décision, -1 s’il perd la partie, 1 s’il la gagner exemple ici avec une défaite :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Cette configuration de **Random vs Random** permet de tester la robustesse de l'environnement et de s'assurer que les règles du jeu sont respectées même dans un cadre non optimisé. Par ailleurs, ce type de simulation offre une base de comparaison pour évaluer les performances futures d'agents plus sophistiqués. En effet, les résultats obtenus avec des agents aléatoires servent de point de référence pour mesurer l’impact d’une logique ou d’un apprentissage sur les stratégies de placement.

Nous obtenons ainsi une moyenne de 200 parties par secondes, avec un taux de victoires entre les 2 random quasiment égaux, montrant ainsi que les random se valent entre eux, cela permet aussi de tester la vitesse de notre environnement qu’on aura prochainement, voici ici un affichage qu’on a sur la console :

Sur 10000 parties :

IA 1 a gagné 4977 fois

IA 2 a gagné 5023 fois

1. **L’environnement**

Dans l’environnement **Lucky Number** pour une grille de taille 4×44 \times 44×4, il y a un total de **54 états** possibles et **38 actions disponibles** pour l'agent. Ces 54 états sont déterminés par plusieurs composantes : la grille de l’agent et celle de l’adversaire, chacune comportant 16 cases, le cache partagé contenant jusqu’à 20 tuiles, la tuile courante qui peut prendre 21 valeurs (incluant l’état vide), et enfin le tour actuel qui peut être celui de l’agent ou de l’adversaire. En ce qui concerne les actions, l’agent peut choisir parmi 38 possibilités, incluant 1 action pour piocher une tuile, 20 actions pour prendre une tuile spécifique du cache, 1 action pour ajouter la tuile courante au cache partagé, et 16 actions pour placer une tuile sur une des cases de sa grille. Cette configuration offre une base structurée et diversifiée pour entraîner des agents et tester différentes stratégies.

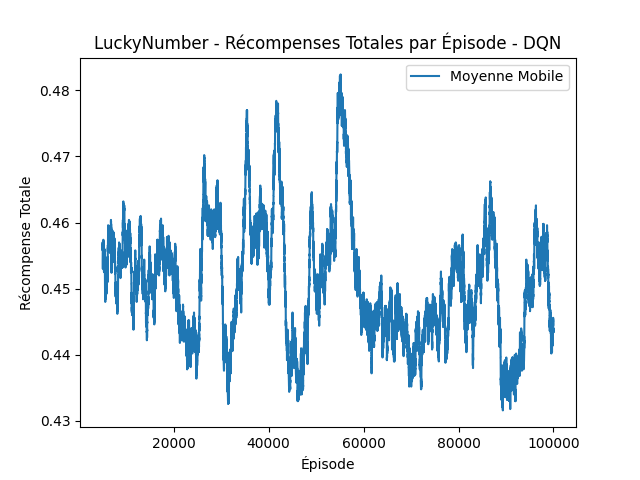
Nous avons conçu et codé l’environnement **Lucky Number** de manière à ce qu’il soit compatible avec tous les algorithmes d’apprentissage par renforcement, qu’il s’agisse de méthodes basées sur des politiques (comme PPO ou REINFORCE) ou sur des valeurs (comme DQN). Cette flexibilité permet d’explorer et de tester une large gamme de stratégies d’entraînement. En parallèle, une attention particulière a été portée à l’optimisation des calculs et à la gestion des états, afin de garantir des performances minimales adéquates, même dans des simulations complexes ou avec des agents multiples. L’objectif était de fournir un environnement polyvalent et robuste, capable de répondre aux exigences de divers algorithmes tout en offrant une exécution fluide et efficace.

Une image contenant texte, capture d’écran

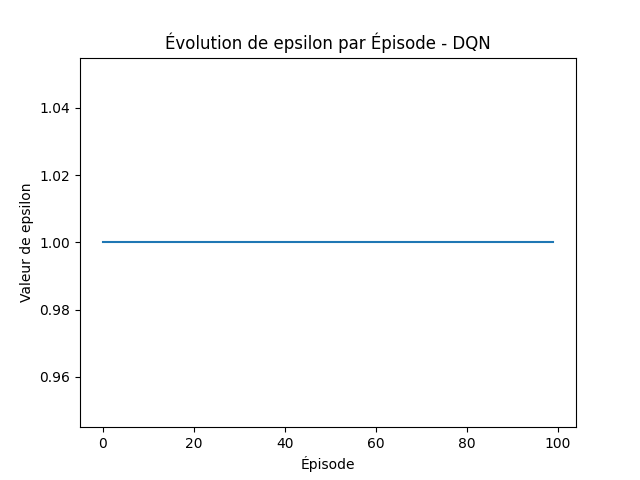
Description générée automatiquement

1. **Analyse des entraînements**

**DQN**



Le graphique montre les récompenses totales par épisode obtenues par l'agent DQN dans l’environnement LuckyNumber. On observe des fluctuations importantes dues à l’exploration active de l’agent, particulirement au début de l’entraînement. La moyenne des récompenses converge progressivement autour de 0.45-0.47, indiquant que l’agent atteint un plateau de performance. Les pics notables (environ 40 000 et 60 000 épisodes) reflètent des périodes où l’agent découvre des stratégies plus efficaces. Cependant, l’absence d’amélioration continue suggère un possible besoin d’ajustement des hyperparamètres ou de la structure du modèle pour une meilleure convergence, étant donné qu’epsilon est toujours à 1 ici car pas de méthode learn, nous pouvons dire que c’est du random.



L’epsilon reste constant à 1, signifiant que l’agent effectue uniquement des actions aléatoires. Cela empêche la transition vers l’exploitation des stratégies optimales apprises.

Une image contenant texte, capture d’écran, diagramme, Tracé

Description générée automatiquement

Dans ce graphique, les récompenses fluctuent initialement, mais deviennent constantes et négatives après l'épisode 40. Cela montre que l’agent n’apprend pas de stratégie efficace et converge vers un comportement inefficace, probablement dû à une exploration insuffisante ou des paramètres mal ajustés.

Une image contenant texte, capture d’écran, affichage, ligne

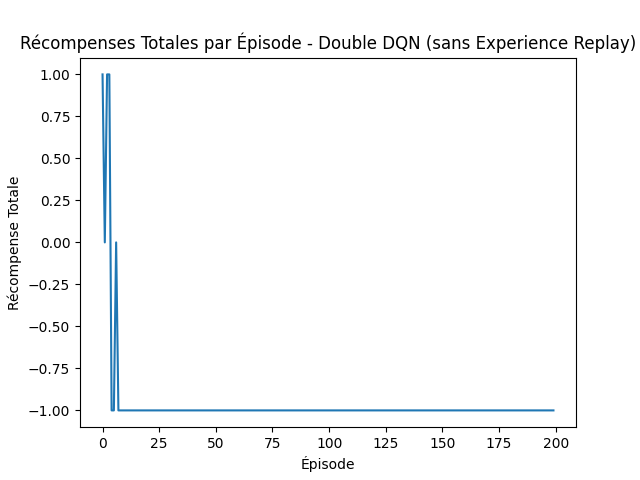
Description générée automatiquement

L’epsilon diminue très rapidement avant l’épisode 20 et reste proche de 0 par la suite. Cela signifie que l’agent passe trop tôt en mode exploitation, sans explorer suffisamment l’environnement, ce qui limite son apprentissage.

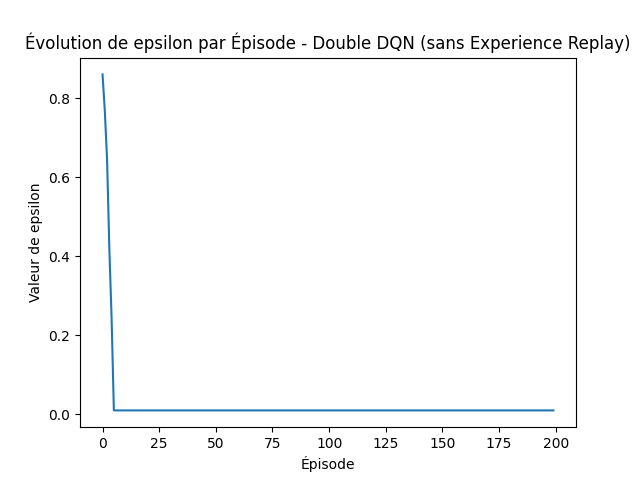
Le problème principal est une exploration insuffisante causée par une réduction trop rapide de l’epsilon. Pour améliorer les performances, il faut ralentir cette diminution et ajuster les récompenses ou les hyperparamètres pour permettre un apprentissage plus efficace.

**DoubleDQN**

Le Double Deep Q-Network (Double DQN) améliore le DQN classique en évitant de surévaluer certaines actions, ce qui rend l’apprentissage plus stable et précis. Il utilise deux réseaux neuronaux : un pour choisir les actions (réseau principal) et un autre pour estimer leur valeur (réseau cible). Cette méthode aide l’agent à maximiser les récompenses en apprenant à prendre de meilleures décisions dans son environnement, grâce à une combinaison d’exploration (epsilon-greedy) et d’un réseau neuronal profond pour traiter des situations complexes.



Le graphique montre une chute rapide des récompenses totales vers une valeur stable de -1. Cela indique que l'agent n'a pas réussi à apprendre une politique efficace, probablement en raison de l'absence d'expérience replay, qui est essentielle pour diversifier les données d'entraînement et éviter le surajustement aux transitions récentes.

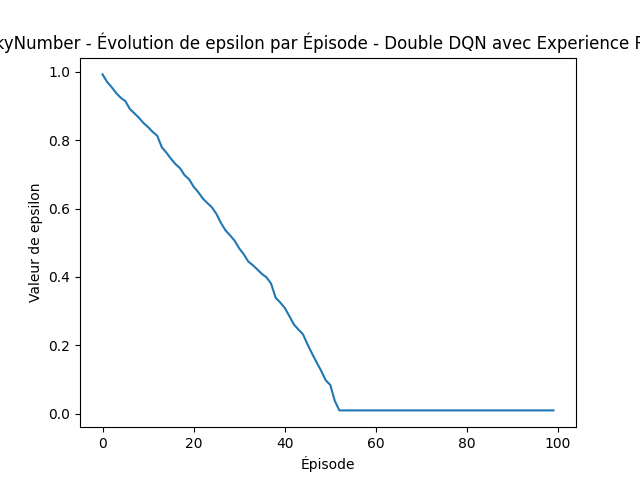


L'epsilon décroît rapidement, passant de 1.0 à une valeur proche de 0 en moins de 50 épisodes. Cela signifie que l'agent réduit très tôt son exploration, se concentrant sur l'exploitation alors qu'il n'a pas encore suffisamment exploré l'environnement. Cette décroissance rapide peut expliquer le manque d'amélioration dans les performances.

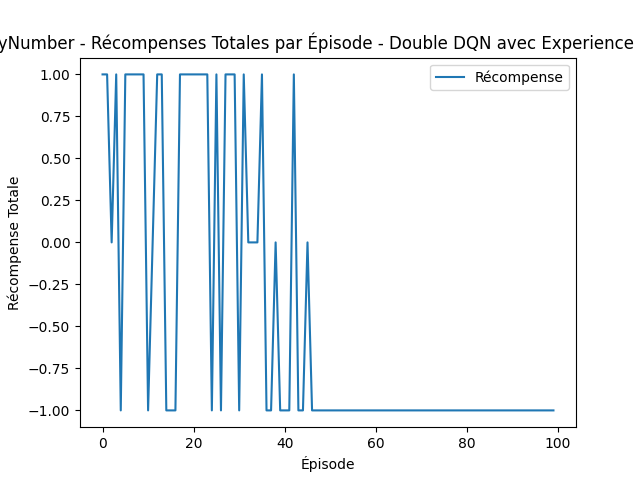
Ces résultats mettent en évidence deux limitations : (1) l’absence d’expérience replay empêche une exploitation efficace des données passées, et (2) la décroissance rapide d’epsilon réduit l’exploration prématurément. Cela conduit à un apprentissage sous-optimal et une politique non performante.

**Double DQN Avec Experience replay**

Le Double Deep Q-Network avec Experience Replay améliore la stabilité et l'efficacité de l'apprentissage grâce à une mémoire qui enregistre les transitions (état, action, récompense, nouvel état, état final). En s'entraînant à partir d'échantillons aléatoires issus de cette mémoire, le réseau réduit les corrélations entre les données, ce qui favorise une meilleure généralisation. L'utilisation de deux réseaux distincts (principal et cible) permet aussi de limiter les biais optimistes dans l'évaluation des valeurs Q.



La valeur d’epsilon décroît graduellement de 1 à 0 sur environ 50 épisodes, suivant une politique programmée. Cela montre que l'agent commence par explorer massivement, puis se concentre de plus en plus sur l'exploitation des politiques apprises. Cette transition bien contrôlée favorise un équilibre efficace entre exploration et exploitation.



Les récompenses oscillent fortement dans les premiers épisodes, avec une alternance entre des performances positives et négatives. À partir d’environ 40 épisodes, les récompenses se stabilisent autour de -1. Cette stagnation peut indiquer que l’agent apprend une politique sous-optimale ou que l’environnement est particulièrement complexe. Les oscillations initiales sont le reflet de l'exploration intensive avec un epsilon élevé.

Le Double DQN avec Experience Replay introduit une meilleure gestion des données et un apprentissage plus stable. Cependant, malgré un mécanisme d’apprentissage optimisé, l’agent semble plafonner à des performances limitées. Pour améliorer les résultats, des ajustements de l’architecture du modèle, des hyperparamètres ou une exploration plus prolongée pourraient être nécessaires.

**Double DQN avec prioritized Expérience Replay**

Le Double Deep Q-Network avec Prioritized Experience Replay (PER) rend l’apprentissage plus efficace en mettant l’accent sur les expériences les plus importantes. Au lieu de choisir les transitions au hasard, le PER leur donne une priorité basée sur leur erreur TD (différence entre la valeur estimée et la cible). Cela permet à l’agent d’apprendre plus rapidement en se concentrant sur les situations où il a le plus à gagner. En combinaison avec le Double DQN, cette approche limite les biais et accélère les progrès de l’agent.

Une image contenant texte, capture d’écran, ligne, Tracé

Description générée automatiquement

Le graphique montre des oscillations fréquentes entre des récompenses positives et négatives au début de l'entraînement, ce qui reflète une phase d'exploration active de l'agent.

Après environ 40 épisodes, les récompenses se stabilisent autour de -1, indiquant que l'agent n'a pas encore appris une politique performante pour cet environnement. Cette stagnation peut provenir d'une complexité élevée de l'environnement ou d'une exploration insuffisante.

Une image contenant texte, diagramme, ligne, capture d’écran

Description générée automatiquement

L'epsilon décroît progressivement de 1.0 à 0.0 en environ 50 épisodes, suivant une stratégie contrôlée. Cela montre une transition entre exploration (au début) et exploitation (plus tard), mais une réduction trop rapide peut limiter l'apprentissage dans les environnements complexes.

Le Double DQN avec Prioritized Experience Replay améliore l’apprentissage en introduisant des priorités, mais la stagnation des performances de l’agent montre qu’il reste des ajustements à faire. Prolonger la phase d’exploration en réduisant plus lentement le taux epsilon et revoir les hyperparamètres de priorisation pourraient renforcer la stabilité et l’efficacité de l’apprentissage. Malgré ces limites, cette méthode reste une avancée importante par rapport aux approches classiques.

**Reinforce**

L’algorithme **REINFORCE**, testé sur l’environnement **Lucky Numbers**, a permis d’obtenir des résultats significatifs en démontrant la capacité de cet algorithme à apprendre une politique optimale dans un environnement à contraintes mathématiques. Cet agent a été implémenté avec une structure robuste, intégrant un modèle de politique neuronale configuré pour s’adapter à la taille des états et des actions spécifiques à l’environnement.

On obtient donc les résultats suivants :

Une image contenant texte, capture d’écran, ligne, Tracé

Description générée automatiquement

Concernant le REINFORCE, il est probable que l’algorithme aurait pu atteindre une politique optimale si le nombre d’itérations avait été plus élevé. Le processus d’apprentissage par REINFORCE, basé sur des mises à jour à partir de retours épisodiques, nécessite souvent un grand nombre d’épisodes pour converger, en particulier dans des environnements complexes avec de multiples contraintes. Une augmentation du nombre d’itérations pourrait ainsi permettre de mieux explorer l’espace des actions et d’affiner les choix stratégiques de l’agent mais aussi une modification des hyperparamètres, fondamentaux pour le processus de l’algorithme de renforcement.

**Reinforce Baseline**

**Une image contenant texte, capture d’écran, ligne, diagramme

Description générée automatiquement**

Pour le reinforce Baseline, l’algorithme n’a clairement pas réussi à réellement apprendre le lucky numbers, nous pourrions dire que le nombre d’itérations est peut-être manquants, et la variation d’hyperparamètres aurait pu changer la donne.

**Reinforce With Learn Critic et PPO**

**Une image contenant texte, capture d’écran, ligne, diagramme

Description générée automatiquementUne image contenant texte, capture d’écran, diagramme, ligne

Description générée automatiquement**

Les algorithmes **REINFORCE** et **PPO (style A2C)** testés sur l’environnement **Lucky Numbers** n’ont pas donné de résultats concluants. Les deux agents ont perdu l’intégralité de leurs parties, et les courbes de loss obtenues étaient biaisées, suggérant des difficultés dans leur processus d’apprentissage. Ces performances insatisfaisantes peuvent être attribuées à plusieurs facteurs : un nombre insuffisant d’itérations pour permettre une convergence efficace, un manque de variation dans les hyperparamètres (comme le taux d’apprentissage, le facteur d’escompte ou la taille des mini-lots), et potentiellement des erreurs dans l’implémentation des algorithmes ou de leur interaction avec l’environnement. Ces observations mettent en évidence la nécessité d’un réglage plus approfondi et de vérifications supplémentaires pour améliorer l’efficacité des modèles dans cet environnement.

**Mcts avec Réseaux de neurones**

**Une image contenant texte, ligne, Tracé, diagramme

Description générée automatiquement**

Pour le **mcts avec réseaux de neurones**, pour raison de performance seulement 10 épisodes ont été réalisés, redant ainsi l’analyse compliquée, nous pouvons conjecturer que le mcts avec réseaux de neurones aurait fini par trouver une politique optimale avec plus d’entraînement, montrant ainsi qu’il peut être performant dans des environnements complexes.

1. **Synthèse et analyse critique**

Ainsi, notre étude des algorithmes de renforcement sur l’environnement **Lucky Numbers** ne reflète pas pleinement le potentiel de ces méthodes. Les résultats obtenus, bien qu'intéressants, sont limités par plusieurs facteurs qui auraient pu être optimisés. Tout d'abord, le nombre d'itérations réalisées était insuffisant pour permettre aux algorithmes d'explorer pleinement l’espace des actions et d’apprendre des stratégies efficaces. Ensuite, une variation plus large des hyperparamètres, tels que le taux d’apprentissage, le facteur d’escompte, ou encore la structure des réseaux neuronaux, aurait pu permettre de mieux adapter les modèles aux spécificités de l’environnement. Enfin, une exploration plus approfondie des configurations et une validation rigoureuse de l’implémentation des algorithmes auraient également pu contribuer à des résultats plus concluants. Ces éléments soulignent que les performances observées ne doivent pas être interprétées comme représentatives du véritable potentiel des algorithmes de renforcement, mais plutôt comme une première étape vers une exploration plus complète.

Malgré un setup GPU performant, conçu pour accélérer les calculs et optimiser les entraînements, les algorithmes testés sur l’environnement **Lucky Numbers** ont consommé une quantité significative de ressources. Cette consommation élevée s'explique par la nature même des algorithmes de renforcement, qui nécessitent des simulations répétées et intensives pour explorer l’espace des actions, ainsi que par la complexité combinatoire de l’environnement. Les calculs impliquant des réseaux neuronaux pour la politique et la fonction de valeur, combinés à la gestion des états et des transitions, ont accentué cette demande en ressources. Cela illustre que, même avec une infrastructure adaptée, l’apprentissage par renforcement peut être très gourmand, notamment dans des environnements où les itérations doivent être nombreuses pour espérer une convergence efficace.

**Conclusion**

En conclusion, les algorithmes de renforcement continuent de démontrer leur puissance et leur efficacité dans des contextes variés, grâce notamment à l’intégration de l’apprentissage supervisé qui améliore la précision et l’adaptabilité des modèles. Cependant, cette montée en puissance s’accompagne d’une consommation de ressources de plus en plus importante, posant des défis en termes de coût énergétique et d’impact environnemental.

Ces méthodes trouvent aujourd’hui des applications prometteuses dans des domaines complexes et exigeants. Par exemple, dans les voitures autonomes, le renforcement permet d’entraîner des agents à prendre des décisions en temps réel tout en minimisant les risques. Dans la finance, notamment pour la gestion de portefeuille ou la prévision des marchés boursiers, ces algorithmes aident à optimiser les stratégies face à des environnements dynamiques et incertains. Ces avancées soulignent le potentiel illimité des algorithmes de renforcement pour résoudre des problèmes de plus en plus sophistiqués.

À l’avenir, un défi majeur sera de développer des algorithmes de renforcement plus durables, capables d’atteindre des performances élevées tout en réduisant leur empreinte écologique. Cela pourrait passer par des architectures plus efficaces, des techniques d’entraînement moins gourmandes en données et en calcul, ou encore des approches hybrides qui s’inspirent de la biologie et des mécanismes naturels. Cette réflexion ouvre la voie vers un apprentissage par renforcement non seulement intelligent, mais également responsable, en harmonie avec les besoins de notre environnement.