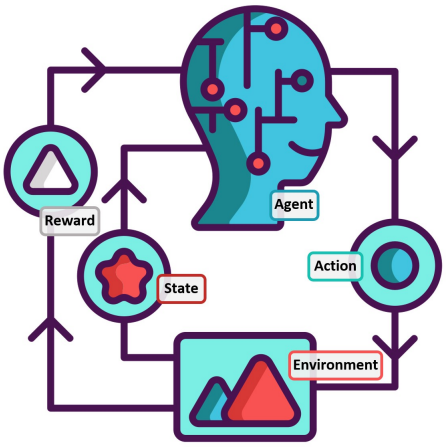
L'apprentissage Q profond dans le cadre de l'apprentissage par renforcement

CARPENTIER Lucas COURTOIS Thibault
SYSMER 3A



Responsable : M. Léandri

27 octobre 2024

Introduction

En 2016, DeepMind a révolutionné le domaine de l'intelligence artificielle lorsque son programme AlphaGo a battu Lee Sedol, champion du monde de Go. Cet exploit marquant a mis en lumière la puissance de l'apprentissage par renforcement profond, un sous-domaine clé du Reinforcement Learning (RL). Ce mémoire s'attache à explorer cette technologie, notamment l'apprentissage Q profond, qui constitue l'une des avancées majeures du RL. Le choix de traiter cette composante du RL se justifie non seulement par son impact novateur, mais aussi par ses contributions significatives à l'amélioration des performances des agents intelligents. L'objectif est de permettre au lecteur de comprendre le fonctionnement des réseaux Q profonds. Pour ce faire, nous commencerons par présenter et expliquer les bases de l'apprentissage par renforcement. Nous aborderons ensuite l'apprentissage Q et expliquerons pourquoi les chercheurs se sont tournés vers l'apprentissage Q profond. Enfin, nous conclurons avec un cas pratique de mise en place d'un Deep-Q-Network en utilisant le framework OpenAI Gym.

Table des matières

1 Théorie générale de l'apprentissage par renforcement (RL)

1.1 Définition de l'apprentissage par renforcement

Le Reinforcement Learning (RL), ou Apprentissage par Renforcement, est une méthode d'apprentissage automatique où un agent apprend à interagir avec un environnement afin de maximiser une récompense cumulative. L'objectif principal est de permettre à l'agent de réagir correctement à toute situation. Pour cela, l'agent explore l'espace dans lequel il évolue et y entreprend des actions. Si ces actions sont jugées favorables, il reçoit une récompense, tandis que les actions conduisant à des comportements indésirables sont pénalisées. L'objectif est alors qu'il cumule le plus possible de récompenses avec le moins de pénalités.

Dans certains contextes, le Reinforcement Learning (RL) ne se limite pas à un seul agent interagissant avec son environnement, mais peut impliquer plusieurs agents évoluant simultanément dans un même espace. Ces environnements multi-agents introduisent une nouvelle complexité, car chaque agent cherche à maximiser sa propre récompense tout en étant influencé par les actions des autres. Dans ces cas, le problème peut être vu à travers la théorie des jeux. Chaque agent doit adapter ses décisions en fonction des actions des autres agents et ce dans une dynamique d'interaction stratégique similaire à ce qui est étudié en théorie des jeux. Par exemple, chaque agent peut adopter des stratégies compétitives ou coopératives, et les décisions de chaque agent affectent directement les récompenses des autres. Cette interaction crée une situation où le succès d'un agent dépend de sa capacité à anticiper et à réagir aux actions des autres, ce qui est typique des scénarios de théorie des jeux comme le dilemme du prisonnier ou la négociation de Nash.

Dans le cadre de notre étude, nous nous concentrerons uniquement sur les cas mono-agents. Cependant, nous invitons grandement le lecteur à s'intéresser à ces autres perspectives qui sont toutes très intéressantes.

1.2 Concepts fondamentaux du RL

Le Reinforcement Learning repose sur plusieurs concepts clés qui définissent la manière dont un agent va interagir avec son environnement afin d'apprendre à maximiser une récompense. Ces composants fondamentaux sont essentiels pour comprendre le fonctionnement global des algorithmes d'apprentissage par renforcement. Ils définissent en quelques sortes un cahier des charges à suivre afin de créer un model d'apprentissage par renforcement. Ces concepts sont les suivants :

— **L'environnement** : Il s'agit d'un monde, qui peut être simulé ou bien réel, dans lequel l'agent évolue. L'environnement définit en quelques sortes les règles du jeu, y compris les actions disponibles pour l'agent, les transitions d'état et la fonction de récompense. Il est la source des informations nécessaires à l'agent pour apprendre et s'améliorer. Ces environnements sont diverses et peuvent prendre des formes très différentes. Dans la dernière section de ce mémoire, quelques environnements seront présentés grâce au framework Open AI Gym.

— **Agent** : L'agent est l'entité au coeur de l'apprentissage. Il prend des décisions et interagit avec son environnement en choisissant des actions à réaliser. Ces actions peuvent lui rapporter des récompenses ou bien des pénalités. Son but est d'apprendre une stratégie optimale pour maximiser ces récompenses tout en minimisant le nombre de pénalité reçu.

— **État** : L'état représente la situation actuelle de l'agent dans l'environnement. C'est une description de l'environnement à un instant donné, et il sert de base à la prise de décision. Chaque changement dans l'environnement modifie l'état perçu par l'agent. L'état est très important car ce sont les seules informations que l'on dispose et que l'on peut alors exploiter.

— **Action** : À chaque étape, l'agent doit choisir une action parmi un ensemble d'actions possibles. L'ensemble des actions formant l'espace d'action, cet espace est très souvent discret pour éviter un comportement imprévu de l'agent. Il peut naviguer dans cet espace pour explorer différentes options et découvrir les meilleures décisions à prendre ainsi que les pires.

— **Récompense** : La récompense est le retour immédiat que l'agent reçoit après avoir effectué une action. Elle peut être positive ou négative, et l'objectif de l'agent est de maximiser la somme des récompenses. Plusieurs stratégies sont alors possibles. Certaines mettent beaucoup en valeur les récompenses pouvant être obtenus immédiatement tandis que d'autres maximisent les récompenses à plus long terme. Ce point sera abordé plus tard dans ce mémoire.

— **Politique** : La politique est la stratégie suivie par l'agent pour choisir ses actions en fonction de l'état actuel. Elle peut être déterministe, c'est-à-dire qu'elle attribue une action précise à chaque état, ou stochastique, où une distribution de probabilités est associée à chaque action possible. L'apprentissage de la politique optimale est au cœur du RL.

Vous l'avez vu, ces différents concepts sont tous variables. Cela signifie que de nombreuses variantes du Reinforcement Learning sont possibles, en fonction de la manière dont l'agent apprend à choisir ses actions. Dans les formes les plus simples, l'agent peut suivre une politique déterminée à l'avance ou une politique basée sur des règles simples. Ces politiques reposent souvent sur des heuristiques ou des algorithmes pré construits qui dictent la meilleure action à entreprendre dans un état donné.

Cependant, lorsque l'environnement devient plus complexe, ces politiques simples atteignent rapidement leurs limites. C'est alors que l'apprentissage par réseau de neurones entre en jeu. En utilisant des réseaux de neurones, l'agent est capable d'apprendre une politique complexe de manière autonome en interagissant avec l'environnement et en ajustant ses décisions en fonction des récompenses qu'il reçoit. Cela permet à l'agent de traiter des espaces d'état et d'action bien plus vastes et de s'adapter à des environnements plus imprévisibles.

Dans la section suivante, nous examinerons comment les réseaux de neurones peuvent être utilisés pour développer des politiques optimales en Reinforcement Learning, et pourquoi cette approche a révolutionné le domaine.

1.3 Politiques basées sur les réseaux de neurones

Les politiques apprises par des réseaux de neurones sont des approches puissantes pour prendre des décisions dans des environnements complexes et dynamiques. Contrairement aux méthodes traditionnelles qui reposent sur des tables de valeurs ou des formules simples, les réseaux de neurones permettent de traiter une large gamme d'états et d'actions et ce en même temps. Cela est crucial dans des environnements à haute dimension ou dans lesquels la complexité est grande. Les réseaux de neurones offrent une capacité de généralisation qui permet d'apprendre des politiques non linéaires. Grâce à cette flexibilité, ils s'adaptent aux variations de l'environnement et gèrent efficacement l'incertitude. Cela les rend particulièrement utiles pour modéliser des situations où les méthodes classiques échouent.

1.3.1 Modèle de politique

Dans cette approche, un réseau de neurones sert à approximer la politique, c'est-à-dire la fonction qui détermine l'action à entreprendre en fonction de l'état actuel de l'environnement. Le modèle peut être résumé comme suit :

— **Entrée :** L'état actuel de l'agent dans son environnement. Cela est souvent représenté sous forme d'un vecteur de caractéristiques.

— **Sortie :** Une action spécifique ou une distribution de probabilités sur les actions possibles que l'agent peut choisir dans cet état.

Cependant, comme avec toutes utilisations de réseaux de neurones, il persiste toujours le même problème : l'entraînement. Dans notre cas nous ne disposons pas d'échantillons de données annotées qui permettraient à l'agent de connaître son environnement sans même y avoir encore évolué. En effet, l'agent est en quelque sorte lâché dans un monde ou il ne connaît pas les conséquences de ses actions. Il va alors falloir qu'il explore son environnement afin de comprendre son fonctionnement.

1.3.2 Méthodes basées sur les gradients de politique

Les méthodes dites de gradient de politique (*policy gradient*) optimisent directement la politique en ajustant les paramètres du réseau de neurones pour maximiser la récompense totale attendue. L'idée est que l'agent va adapter ses décisions en fonction des retours qu'il obtient de l'environnement :

— Au début de l'entraînement, les actions sont sélectionnées de manière aléatoire, mais à mesure que l'agent apprend, les actions les plus efficaces deviennent de plus en plus probables.

— L'algorithme ajuste progressivement les poids du réseau de neurones pour favoriser les actions conduisant à des récompenses élevées et réduire la probabilité des actions moins bénéfiques.

— L'apprentissage repose sur la récompense cumulative de chaque épisode, ce qui permet d'affiner la politique pour maximiser cette récompense à long terme.

Cette capacité à optimiser directement la politique rend les réseaux de neurones particulièrement efficaces dans des environnements où il est difficile de calculer explicitement les solutions optimales avec des approches traditionnelles.

1.3.3 Exploration vs Exploitation

Un des défis majeurs dans l'apprentissage par renforcement est le dilemme entre exploration et exploitation. L'agent doit choisir entre exploiter les connaissances acquises jusqu'à présent pour maximiser sa récompense immédiate (exploitation), ou bien explorer de nouvelles actions qui pourraient potentiellement conduire à de meilleures récompenses à long terme (exploration). Les réseaux de neurones, lorsqu'ils sont utilisés pour modéliser des politiques, doivent également naviguer entre ces deux approches. Initialement, l'agent explore beaucoup pour comprendre l'environnement, mais à mesure qu'il apprend, il doit trouver un équilibre en exploitant ce qu'il sait déjà tout en continuant à explorer de nouvelles stratégies. Trouver cet équilibre est crucial pour converger vers une politique optimale.

Les concepts abordés, tels que l'agent, l'environnement, la politique, ainsi que le dilemme exploration-exploitation, s'inscrivent tous dans un cadre théorique plus large : celui des processus de décision Markoviens (MDP). Ce modèle mathématique est couramment utilisé pour formaliser les problèmes d'apprentissage par renforcement. Dans un MDP, chaque décision que prend l'agent dépend de l'état actuel de l'environnement et de la politique suivie. La décision influence l'état futur à travers une fonction de transition probabiliste.

1.4 Processus de décisions Markoviens

Les chaînes de Markov sont des processus stochastiques sans mémoire qui évoluent d'un état à un autre de manière aléatoire, selon une probabilité de transition qui dépend uniquement de l'état actuel. Cependant, dans un processus de décision Markovien un agent peut choisir parmi plusieurs actions et les probabilités de transition entre les états dépendent alors des actions choisies. De plus, certaines transitions génèrent des récompenses soient positives ou bien au contraire négatives. L'objectif de l'agent comme dit précédemment est de maximiser les récompenses cumulées au fil du temps en suivant une politique optimale. Voici un exemple de schéma d'une situation dans laquelle on pourrait y établir un processus de décisions Markovien :

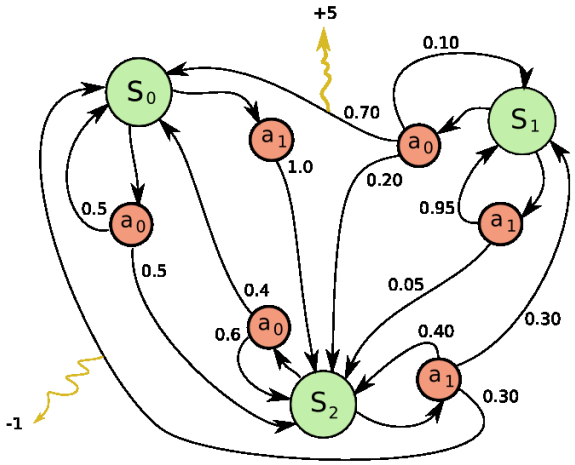


FIGURE 1 – Processus décisionnels de Markov [0]

Dans ce cas nous avons alors 3 états (S_0, S_1, S_2) et 2 actions (a_0, a_1) disponibles à chaque état. Chaque transition entre les états est associée à une probabilité (indiquée à côté des flèches) et certaines transitions entraînent des récompenses positives ou négatives (indiquées par $+5$ et -1).

— Les états : Les états disponibles sont S_0, S_1, et S_2. L'agent se trouve dans un état maximum et ce à chaque moment.

— Les actions : L'agent peut choisir entre deux actions possibles dans chaque état : a_0 et a_1. Par exemple, dans l'état S_0, l'agent peut prendre l'action a_0 ou a_1. Chaque action emmène l'agent vers un autre état.

— Les probabilités de transition : Les probabilités sont représentées par des valeurs sur les flèches reliant les états. Par exemple, à partir de l'état S_0, si l'agent choisit l'action a_0, il a 50% de chance de rester dans S_0 et 50% de chance de se déplacer vers S_2. Si l'agent choisit a_1, il a 100% de chance d'aller à S_2. Une fois dans S_2, si il prend l'action a_1 il aura 40% de chance de rester sur ce même état, 30% d'aller en S_0 et enfin 30% de chance d'aller en S_1.

— Les récompenses : Certaines transitions sont associées à des récompenses positives ou négatives. Par exemple, l'agent peut recevoir une récompense de $+5$ en arrivant à l'état S_1 et après avoir pris l'action a_0, ou une pénalité de -1 en prenant l'action a_1 depuis l'état S_2.

1.4.1 Équation d'optimalité de Bellman

Le mathématicien Richard Bellman a formulé une équation permettant de calculer la valeur d'état optimale $V^*(s)$, qui représente la somme des récompenses futures attendues en suivant une politique optimale à partir de l'état s. Voici son équation :

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V^*(s')\right]$$

Définition des termes :

— $T(s, a, s')$: probabilité de transition de l'état s à l'état s' après l'action a.

— $R(s, a, s')$: récompense obtenue pour la transition de s à s' avec l'action a.

— γ : facteur de rabais qui pondère l'importance des récompenses futures.

1.4.2 Algorithme d'itération sur la valeur

L'algorithme d'itération sur la valeur permet d'estimer progressivement les valeurs d'état optimales en mettant à jour les estimations à chaque itération, jusqu'à convergence du résultat. La convergence ici est en réalité le moment ou la somme des récompenses est maximisée. A ce moment précis le comportement de l'agent est considéré optimal.

Voici l'équation relient l'état $k+1$ à l'état k.

$$V_{k+1}(s) = \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V_k(s')\right]$$

1.4.3 Valeurs Q et itération sur la valeur Q

L'algorithme d'itération sur la valeur Q permet de calculer la valeur Q optimale pour chaque couple état-action (s, a). La valeur $Q^*(s, a)$ représente la somme des récompenses futures pondérées après avoir effectué l'action a à partir de l'état s, en supposant que l'agent agit de manière optimale par la suite [1].

$$Q^*(s, a) = \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \max_{a'} Q^*(s', a')\right]$$

Une fois que les valeurs $Q^*(s, a)$ sont connues, la politique optimale $\pi^*(s)$ est définie comme l'action qui maximise $Q^*(s, a)$ pour chaque état s :

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

1.5 Processus d'apprentissage par Différence Temporelle

Les processus de décision Markoviens offrent un cadre théorique puissant pour modéliser la prise de décision dans des environnements complexes et dynamiques. Cependant, l'un des défis majeurs de cette approche est que l'agent ne connaît pas initialement l'environnement dans lequel il évolue. Il ne dispose donc pas des informations nécessaires, comme les probabilités de transition ou les récompenses associées aux différentes actions, pour déterminer directement une politique optimale. Dans ces conditions, l'agent doit explorer l'environnement pour découvrir comment ses actions influencent les transitions entre états et les récompenses qu'il reçoit. C'est ici que les méthodes d'apprentissage viennent compléter les MDP. L'apprentissage par différence temporelle permet à l'agent de mettre à jour ses estimations de valeurs et de politiques au fur et à mesure qu'il interagit avec l'environnement, sans avoir besoin d'un modèle global.

1.6 Algorithme d'apprentissage par Temporal Difference

Les problèmes d'apprentissage par renforcement avec des actions discrètes peuvent souvent être modélisés à l'aide des processus de décision Markoviens. Cependant, l'agent n'a initialement aucune idée des probabilités des transitions (il ne connaît pas $T(s, a, s')$) et ne sait pas quelles seront les récompenses (il ne connaît pas $R(s, a, s')$). Il doit tester au moins une fois chaque état et chaque transition pour connaître les récompenses, et il doit le faire à plusieurs reprises s'il veut avoir une estimation raisonnable des probabilités des transitions.

L'algorithme d'apprentissage par différence temporelle est très proche de l'algorithme d'itération sur la valeur, mais il prend en compte le fait que l'agent n'a qu'une connaissance partielle du MDP. En général, on suppose que l'agent connaît initialement uniquement les états et les actions possibles rien de plus. Il se sert d'une politique d'exploration, par exemple une politique purement aléatoire, pour explorer le MDP. Au fur et à mesure de sa progression l'algorithme actualise les estimations des valeurs d'état en fonction des transitions et des récompenses observées. Il aura ainsi en connaissance toutes les cartes du jeu après un grand nombre d'étape.

$$V_k(s) \leftarrow V_k(s) + \alpha \left[r + \gamma V_k(s') - V_k(s) \right]$$

ou, de façon équivalente :

$$V_k(s) \leftarrow V_k(s) + \alpha \delta_k(s, r, s')$$

avec

$$\delta_k(s, r, s') = r + \gamma V_k(s') - V_k(s)$$

- α : le taux d'apprentissage.

La cible TD est donnée par :

$$r + \gamma V_k(s')$$

L'erreur TD est notée :

$$\delta_k(s, r, s')$$

On peut établir une analogie avec la méthode de descente du gradient, où l'algorithme ajuste progressivement les valeurs d'état pour minimiser l'erreur entre la valeur actuelle et la valeur estimée. C'est ce mécanisme d'ajustement progressif qui permet à l'agent d'apprendre efficacement en interagissant avec son environnement. Vous l'avez probablement compris, l'apprentissage Q profond (Deep Q-Learning) tire son nom du paramètre Q que nous venons d'explorer. Toutefois, avant d'introduire l'apprentissage Q profond, il est essentiel de comprendre les bases de l'apprentissage Q classique et d'examiner les limites qui ont motivé les chercheurs de DeepMind à intégrer des réseaux de neurones pour améliorer les performances.

2 Apprentissage Q : Algorithme de base

2.1 Introduction au Q-Learning

L'algorithme du Q-Learning est une méthode d'apprentissage par renforcement qui adapte l'algorithme d'itération sur la valeur Q pour des situations où l'agent ne connaît ni les probabilités de transition ni les récompenses à l'avance. Cet algorithme permet à l'agent d'apprendre par l'expérience et en essayant différentes actions de manière aléatoire au départ. Il ajuste au fur et à mesure progressivement ses estimations des valeurs Q en fonction des résultats observés.

Au plus l'agent explore son environnement, au plus il améliore ses estimations des valeurs Q pour chaque paire état-action. Une fois qu'il possède des estimations suffisamment précises, il peut adopter une politique optimale en fonction du besoin de la situation.

2.2 Algorithme d'apprentissage Q

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

L'algorithme met à jour la valeur Q pour chaque paire état-action (s,a) en calculant une moyenne pondérée des récompenses immédiates r que l'agent reçoit en effectuant l'action a dans l'état s. Il y ajoute également la somme des récompenses futures qu'il anticipe. Puisque l'objectif est d'optimiser les actions futures, la somme des récompenses futures est estimée en prenant le maximum des valeurs Q pour les actions possibles dans l'état suivant s'.

2.3 Apprentissage Q et Convergence

L'algorithme d'apprentissage Q permet de converger vers les valeurs Q optimales, mais il nécessite un grand nombre d'itérations et l'ajustement de plusieurs hyperparamètres. Contrairement à l'itération sur la valeur Q, qui converge rapidement sous certaines conditions, l'apprentissage Q prend généralement plus de temps pour atteindre une solution optimale. Cette différence est principalement due au fait que, dans l'apprentissage Q, l'agent ne dispose pas au départ des informations sur les probabilités de transition ou les récompenses associées à chaque action. Cela complexifie la recherche d'une politique optimale, car l'agent doit explorer et estimer ces informations au fur et à mesure de ses interactions avec l'environnement.

2.4 Politiques d'exploration

1. **Politique gloutonne (greedy algorithm)** : Cette politique consiste à toujours choisir l'action qui va maximiser la valeur Q pour l'état actuel en choisissant la meilleure action a effectuer. Cela signifie que l'agent exploite ses connaissances actuelles sans prendre de risque en explorant de nouvelles actions. Cela peut parfois porter préjudice aux résultats car l'agent restera en quelques sortes dans sa zone de confort et n'ira pas tenter de nouvelles possibilités qui pourraient s'avérer plus fructueuses.

$$\pi(s) = \arg\max_{a} Q(s,a)$$

2. **Politique epsilon-gloutonne (epsilon-greedy algorithm)** : La politique epsilon-gloutonne introduit un compromis entre l'exploration et l'exploitation. Avec une probabilité ϵ, l'agent choisit une action aléatoire (exploration), tandis qu'avec une probabilité $1 - \epsilon$, il choisit l'action optimale (exploitation). Cela permet de régler le problème cité précédemment.

— Avec une probabilité ϵ, choisir une action aléatoire :

$$alpha = \text{random}(a)$$

— Avec une probabilité $1 - \epsilon$, choisir l'action optimale :

$$alpha = \arg\max_a Q(s, a)$$

3. **Politique stochastique** : Dans une politique stochastique, l'agent choisit une action selon une distribution de probabilité sur les actions possibles. Cela permet une certaine variabilité dans les choix d'actions et de laisser place à un comportement plus souple de l'agent.

$$\pi(a|s) = P(a|s)$$

Une autre manière d'améliorer l'exploration consiste à inclure une fonction d'exploration dans l'algorithme, qui incite l'agent à tester des actions moins fréquemment choisies. Cela se fait en ajoutant un bonus aux estimations de la valeur Q pour les actions moins explorées :

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') + f(Q, N(s', a'))\right)$$

Dans cette équation :
- $N(s', a')$ représente le nombre de fois où l'action a' a été choisie dans l'état s'.
- $f(Q, N)$ est une fonction d'exploration, comme $f(Q, N) = \frac{\kappa}{1+N}$, où κ contrôle l'incitation à explorer des actions peu fréquentes.

Cette approche permet à l'agent d'équilibrer encore mieux l'exploration des actions peu connues tout en affinant ses estimations des valeurs Q.

2.5 Limites de l'apprentissage Q classique

Bien que l'algorithme d'apprentissage Q soit efficace pour des problèmes relativement simples, il rencontre des difficultés lorsque l'espace d'état ou d'action devient très vaste. En effet, chaque paire état-action nécessite de stocker une valeur Q spécifique, ce qui peut devenir inenvisageable pour des environnements complexes avec des milliers, voire des millions d'états et d'actions possibles. De plus, l'apprentissage Q classique repose sur une table de valeurs Q qui devient rapidement trop grande à manipuler ou à mettre à jour efficacement dans ces scénarios.

C'est là qu'interviennent les réseaux de neurones Q profonds. Au lieu de stocker directement une table de valeurs Q, les DQN approximent ces valeurs à l'aide d'un réseau de neurones. Ce réseau prend en entrée une représentation de l'état et estime les valeurs Q pour chaque action possible. Cela permet de généraliser sur de vastes espaces d'état, en capturant des structures complexes au sein des données d'entrée tout cela sans avoir besoin de connaître explicitement chaque état ou transition.

3 Deep Q-Learning (DQN)

3.1 Définition

Les réseaux Q profonds constituent une extension puissante de l'apprentissage Q traditionnel. Introduits par DeepMind dans les années 2010, ces réseaux de neurones permettent de surmonter les limitations de l'apprentissage Q classique en approximant les valeurs Q à l'aide de réseaux de neurones profonds. Cela leur permet d'exceller dans des environnements complexes, comme les jeux Atari, où ils ont montré des performances remarquables [2].

La force principale des DQN réside dans leur capacité à gérer des environnements où le nombre d'états et d'actions est astronomique. Par exemple, dans des jeux vidéo ou des simulations complexes, l'espace d'état est souvent trop vaste pour être traité par une table de valeurs Q traditionnelle. Les réseaux Q profonds permettent de généraliser l'apprentissage à travers ces grands espaces, en utilisant un réseau de neurones qui ajuste ses poids au fil du temps, en apprenant à partir d'exemples rencontrés lors de l'interaction avec l'environnement. En revanche, bien que l'apprentissage Q profond soit une alternative tout à fait envisageable dans de nombreux scénario, les mêmes problèmes qui se posent avec les réseaux de neurones se posent également ici. Il est alors nécessaire de bien avoir en tête toutes les problématiques qui peuvent se poser.

3.2 Entraînement des DQN

L'entraînement d'un DQN consiste à ajuster les poids du réseau de manière à ce que la valeur Q estimée $Q(s, a)$ se rapproche de la **valeur Q cible**, calculée en utilisant l'équation de Bellman :

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

La propagation de l'erreur est une technique clé permettant d'ajuster les poids d'un réseau de neurones en fonction de l'erreur commise à la sortie. Cela permet de choisir les poids qui minimisent l'erreur de prédiction, voici un résumé du fonctionnement :

1. **Propagation avant :** Le réseau génère une sortie en passant les données d'entrée à travers les couches cachées jusqu'à la sortie finale. Chaque neurone applique une fonction d'activation sur la somme pondérée des entrées et transmet ce résultat aux neurones suivants.
2. **Calcul de l'erreur :** L'erreur est mesurée en comparant la sortie prédite avec la sortie réelle via une fonction de perte (comme l'entropie croisée pour la classification, l'erreur quadratique moyenne pour la régression).
3. **Rétropropagation :** L'erreur est propagée en sens inverse soit de la couche de sortie vers les couches cachées en calculant les gradients des poids à l'aide de la règle de la chaîne. Ces gradients indiquent comment ajuster les poids pour réduire l'erreur de prédiction.
4. **Mise à jour des poids :** Les poids sont ajustés en utilisant la descente de gradient selon la formule :
$$w \leftarrow w - \alpha \frac{\partial L}{\partial w}$$
où α est le taux d'apprentissage, et $\frac{\partial L}{\partial w}$ est le gradient de la perte par rapport au poids.

Le DQN est entraîné pour minimiser l'écart entre la valeur estimée et la valeur cible. Ce processus utilise des algorithmes de descente de gradient standard. Cette approche permet au DQN d'apprendre progressivement une politique efficace à partir des interactions avec l'environnement, même lorsque l'espace d'état est vaste et complexe.

4 Améliorations des Deep-Q-Network

Comme pour les autres types de réseaux de neurones, de nombreuses variantes des DQN existent et permettent de réduire l'erreur et le temps de calcul. En voici une liste non exhaustive.

4.1 Cibles de la valeur Q Fixées

Dans l'algorithme standard de l'apprentissage Q profond, le modèle utilisé pour prédire les valeurs Q est également celui qu'on utilise pour définir ses propres cibles. Cette approche peut entraîner une rétroaction instable qui peut potentiellement engendrer des erreurs ou des oscillations dans l'entraînement du réseau. Nous pourrions prendre comme analogie le chien qui essaye de courir après sa queue en pensant attraper un autre chien... Pour remédier à ce problème, DeepMind a proposé en 2013 une solution consistant à utiliser deux réseaux de neurones Q distincts.

— **Modèle en ligne :** Ce modèle est mis à jour à chaque étape d'apprentissage. Il est responsable de choisir les actions de l'agent en fonction des valeurs Q prédictives. C'est alors le réseau principal.

— **Modèle cible :** Contrairement au modèle en ligne, le modèle cible est utilisé uniquement pour définir les cibles des valeurs de Q. Il s'agit d'une copie du modèle en ligne, mais il n'est mis à jour qu'à des intervalles réguliers, ce qui permet de fixer des cibles stables et de réduire les fluctuations durant l'apprentissage. Cela permet de réduire grandement le taux de sur-apprentissage.

4.2 Double Deep Q Network

En 2015, les chercheurs de DeepMind ont perfectionné leur algorithme DQN pour en améliorer les performances et stabiliser le processus d'apprentissage. Cette nouvelle version, connue sous le nom de *Double Deep-Q-Network* (DQN double) a permis de corriger certains qui étaient présents dans l'algorithme. L'amélioration repose sur une observation qui a été cruciale : le réseau cible tend à surestimer les valeurs Q. En théorie, si toutes les actions sont équivalentes, les valeurs Q prédites par le modèle cible devraient être cohérentes sauf que dans la pratique, les approximations faites par le modèle peuvent être légèrement surestimé et ce causé par le hasard. Le modèle cible choisit systématiquement la valeur Q la plus élevée, ce qui peut conduire à une estimation exagérée par rapport à la valeur Q moyenne. Cela fausse alors la valeur réelle car nous nous retrouvons avec des valeurs de Q plus élevées que la moyenne ce qui n'est pas souhaitable.

Cette dissociation entre la sélection d'action réalisée par le modèle en ligne et l'estimation des valeurs Q réalisée par le modèle cible permet de mieux gérer le biais de surestimation. En effet, en séparant ces deux tâches l'algorithme s'assure que les actions ne sont pas systématiquement choisies sur la base de valeurs Q surestimées. Cela améliore la stabilité de l'apprentissage et permet à l'agent de faire des choix plus fiables à long terme.

4.3 Dueling DQN

Les Dueling Deep Q-Network sont une extension de l'algorithme DQN qui améliore la précision des estimations des valeurs Q et ce en particulier dans les environnements où certaines actions n'ont que peu d'influence sur les récompenses immédiates. L'idée clé derrière est de séparer l'estimation de la valeur d'un état s et celle de l'avantage d'une action a. Le réseaux de neurones est divisé en deux branches :

— Une branche estime la valeur d'état $V(s)$, qui mesure combien un état s est bon indépendamment des actions possibles.

— L'autre branche estime l'avantage de chaque action $A(s, a)$, qui mesure l'impact relatif de chaque action a dans l'état s.

Les deux branches sont ensuite combinées pour obtenir une estimation des valeurs Q. Cela permet d'obtenir des estimations plus stables et plus précises dans des environnements complexes. La combinaison se fait généralement via l'équation suivante :

$$Q(s, a) = V(s) + \left(A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a') \right)$$

où $\mathcal{A}$ est l'ensemble des actions possibles.

4.4 Prioritized Experience Replay

Le Prioritized Experience Replay est une amélioration du mécanisme classique de replay buffer dans les réseau Q profond. Dans un DQN classique, les expériences sont stockées dans une mémoire et sont ensuite échantillonnées de manière aléatoire pour l'apprentissage. Cependant, toutes les expériences n'ont pas la même importance pour l'apprentissage. Certaines se révèlent très fructifiantes et d'autre peuvent l'être moins. Pour palier ce problème, le but est de prioriser les expériences bénéfiques, ce sont celles qui ont un TD-error (erreur temporelle) élevé, c'est-à-dire celles où il y a une plus grande différence entre la valeur Q actuelle et la cible de la mise à jour. Cela permet à l'agent de se concentrer davantage sur les transitions qui contiennent des informations critiques pour améliorer sa politique que sur celle qui ne lui apportent pas beaucoup d'informations.

Le degré de priorité peut être ajusté avec un hyper-paramètre α qui contrôle dans quelle mesure la probabilité d'échantillonnage dépend de l'erreur TD. La formule pour la probabilité d'échantillonnage d'une transition i est donnée par :

$$P(i) = \frac{|_i|^\alpha}{\sum_k |_k|^\alpha}$$

où $_i$ est la priorité de transition pour i i. Si $\alpha = 1$ alors $p_i^\alpha = 1$.

De plus, pour compenser le biais introduit par cet échantillonnage, une correction par importance-sampling est appliquée pendant l'apprentissage et elle-même ajustée par un hyperparamètre β, afin de garantir une convergence stable et non biaisée :

$$w(i) = \left(\frac{1}{N \cdot P(i)} \right)^\beta$$

où N est le nombre total d'expériences dans le buffer. Cette approche permet à l'algorithme d'apprendre plus rapidement en se concentrant sur les transitions les plus pertinentes. Cela permet dans de nombreux cas de réduire le temps de calcul et d'améliorer les performances du model.

5 Application avec OpenAi Gym

Cette section a pour but de présenter une implémentation de réseaux Q et Q profonds. Cela permettra alors au lecteur de voir une application réelle de la théorie discutée précédemment. Le but est donc de comprendre comment implémenter ces techniques dans un environnement. Plusieurs tutoriels ont été suivis afin de proposer un contenu clair et précis.

5.1 Environnement FrozenLake avec un algorithme Q

Cette première application porte sur l'environnement FrozenLake de OpenAI Gym. Cet environnement présente un problème de prise de décision intéressant, où un agent doit naviguer sur une grille glacée pour atteindre un objectif tout en évitant de tomber dans des trous. De plus, le sol étant glacé, les déplacements de l'agent sont alors soumis à un comportement non désiré aléatoire (donc non solvable par des algorithmes classiques de recherche de chemin comme A*).
Cette situation est facilement extrapolable à une situation de planification de chemin dans un environnement discrétisé pour un robot en milieu ouvert dans le cas non glissant.

5.1.1 Description

L'environnement initial représente une grille (4x4) où l'agent (lutin) doit se déplacer pour atteindre une case finale (cadeau). Chaque case de la grille peut être l'une des suivantes :

- **S - start** : Le point de départ de l'agent.
- **F - frozen** : Une case de glace solide sur laquelle l'agent peut se déplacer sans danger.
- **H - hole** : Un trou dans la glace où l'agent tombe, provoquant un échec de l'épisode.
- **G - goal** : La case de l'objectif que l'agent doit atteindre pour gagner.

Voici un exemple de ce à quoi l'environnement pourrait ressembler dans le cas d'un $(4 * 4)$:

FIGURE 2 – Environnement FrozenLake pour n=4

L'agent peut effectuer 4 actions possibles :

- **0** : Aller vers la gauche.
- **1** : Aller vers le bas.
- **2** : Aller vers la droite.
- **3** : Aller vers le haut.

L'état de l'agent correspond à la case sur laquelle il se trouve. La case de départ (start) est systématiquement dans le coin supérieur gauche de la carte (0), la dernière case est dans le coin inférieur droit (15). Dans le cas d'une carte de taille 4x4, il y a donc 16 états possibles, de 0 à 15,

correspondant à l'ensemble des positions que peut prendre l'agent.

La matrice actions/états ou Q-table, qui sera recalculée à chaque itération de l'apprentissage pour trouver la politique optimale, sera donc composée de n x n lignes (nombre d'états correspondant à la taille de la carte) et 4 colonnes (taille du champ d'action).

TABLE 1 – Q-table pour FrozenLake 4x4

État	0	1	2	3
0	$q_{0,0}$	$q_{0,1}$	$q_{0,2}$	$q_{0,3}$
1	$q_{1,0}$	$q_{1,1}$	$q_{1,2}$	$q_{1,3}$
2	$q_{2,0}$	$q_{2,1}$	$q_{2,2}$	$q_{2,3}$
⋮	⋮	⋮	⋮	⋮
15	$q_{15,0}$	$q_{15,1}$	$q_{15,2}$	$q_{15,3}$

5.1.2 Configuration de l'environnement

utiliser l'apprentissage Q-Learning avec une politique ϵ-greedy pour résoudre le problème et atteindre la case G le maximum de fois possible. Dans un premier temps, le glissement aléatoire sera désactivé (`slippery=False`) ; la situation sera donc déterministe et un apprentissage devrait permettre d'obtenir la récompense systématiquement. Puis, dans un second temps, nous introduirons le glissement aléatoire.

Pour complexifier la situation nous générerons systématiquement une carte de taille 8x8. De cette façon, les prises de décisions basés sur l'exploration et l'exploitation auront un rôle plus important et il faudra implémenter une méthode pour utiliser correctement chacune des ces phases d'apprentissage.

5.1.3 Implémentation du Q-Learning epsilon-greedy

Pour équilibrer les phases d'exploitation et d'exploration, une méthode couramment utilisée est d'introduire un "decay-rate" ou taux de décroissance. L'idée est de partir d'un paramètre epsilon à 1, ce qui équivaut à une exploration maximale pour interagir avec le maximum de situations différentes possible. Puis, à chaque apprentissage, nous allons décrémenter la valeur d'epsilon par le taux de décroissance pour exploiter de plus en plus les apprentissages précédents.
Le nombre d'itérations de l'apprentissage est donc directement lié au facteur de décroissance. L'objectif étant d'arriver à une politique greedy, il faut qu'epsilon atteigne 0. Dans notre cas, le taux de décroissance sera de 0.0001, ce qui équivaut à un nombre minimal d'itérations de 10000.

Pour trouver les valeurs optimales des hyper-paramètres α (taux d'apprentissage) et γ (facteur d'actualisation), une approche courante est d'utiliser une recherche par grille (grid search) combinée à une validation croisée. Cette méthode consiste à définir une plage de valeurs possibles pour chaque paramètre (par exemple, α et γ de 0.1 à 0.9 par pas de 0.1) et à tester toutes les combinaisons. Pour chaque combinaison, on effectue plusieurs exécutions de l'algorithme et on évalue la performance moyenne (par exemple, le nombre moyen d'épisodes pour atteindre l'objectif). La combinaison qui donne les meilleurs résultats est alors choisie. Cette approche peut être coûteuse en temps de calcul, mais elle permet d'explorer systématiquement l'espace des paramètres et de trouver une configuration robuste pour le problème donné.

Le code ne détaille pas la rechercher des paramètres α et γ. Ils seront égaux tous les deux à 0.9 pendant l'apprentissage. Une fois arrivé à une politique 100% greedy, α est passé à 0.01 pour éviter le sur-apprentissage sur des situations qui ne sont plus que de l'exploitation des apprentissages précédents.

Dans les deux cas étudiés (avec ou sans glissement), les hyper-paramètres, le taux de décroissance ainsi que le nombre d'itérations de l'apprentissage seront les mêmes. Une itération d'apprentissage est terminé quand l'agent tombe dans un trou ou récupère la récompense, ou quand celui-ci effectue 200 actions sans trouver la récompense.

5.2 Transcription de la formule de Q-Learning epsilon-greedy en python

Pour rappelle, voici la formule du Q-Learning :

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

Ce qui dans le code python équivaut à :

```
q[state,action] = q[state,action] + learning_rate_a * ( reward + discount_factor_g *
np.max(q[new_state,:]) - q[state,action])
```

Avec :

- $Q(s,a)$, correspondant ici à la Q-table qui est recalculée à chaque itération de l'apprentissage. Elle est initialisée dans le code par un tableau numpy de dimension 64x4 (nous travaillons sur une carte de dimension 8x8) : `q = np.zeros((env.observation_space.n, env.action_space.n))`
- `learning_rate_a` et `discount_factor_g` correspondant aux hyper-paramètres α et γ
- Reward, state et action sont des variables qui prennent les valeurs des méthodes sample et step du framework Gymnasium.

Dans le code une condition a été introduite avec un booléen `is_training` et permet d'appliquer la Q-Learning pendant l'entrainement ou utiliser directement la dernière Q-table calculé après l'entrainement pour tester l'apprentissage final.

5.3 Analyses des résultats et comparaisons

5.3.1 Analyse de l'apprentissage

Pour analyser l'apprentissage, nous utilisons un graphique qui compte le nombre de succès (récupération de la récompense par l'agent) par tranche de 100 itérations d'apprentissage. Une politique optimale correspond donc, dans un cas déterministe, à 100 récompenses pour 100 essais. Ce graphe va nous permettre de voir l'évolution du nombre de récompenses obtenues par tranche de 100 au fur et à mesure de l'apprentissage, ainsi que la valeur seuil, dans les cas avec et sans glissements.

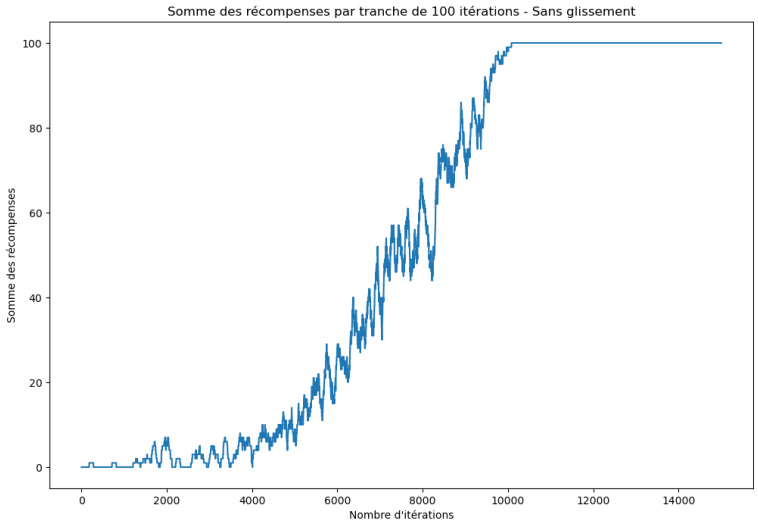


FIGURE 3 – Somme des récompenses par tranche de 100 itérations – Sans glissement

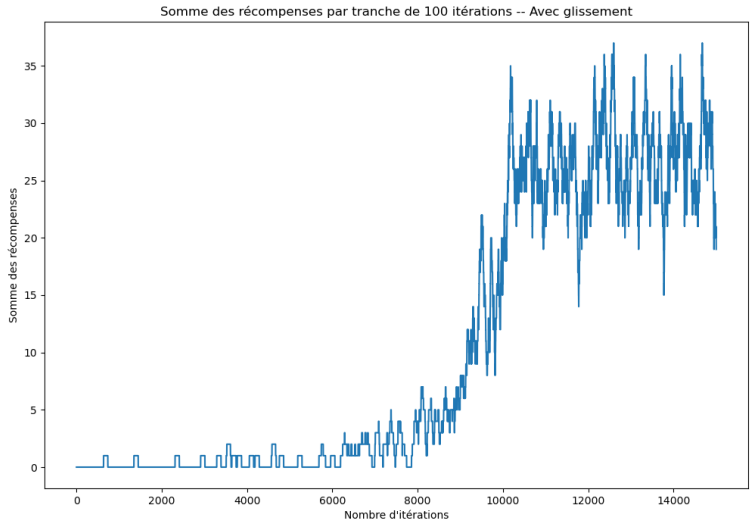


FIGURE 4 – Somme des récompenses par tranche de 100 itérations – Avec glissement

Les figures 4 et 3 illustrent l'évolution de l'apprentissage dans les environnements avec et sans glissement, respectivement.

Dans le cas sans glissement (Figure 3), nous observons une progression exponentielle qui ce stabilise parfaitement au seuil maximal de 100 récompense pour 100 itérations aux alentours de 10000 itérations (passage à une politique totalement greedy et arrêt de l'apprentissage). Cette courbe reflète un apprentissage efficace dans un environnement déterministe, où l'agent peut rapidement identifier et exploiter la stratégie optimale.

En revanche, dans le cas avec glissement (Figure 4), l'apprentissage présente une progression plus lente, irrégulière mais toujours avec un caractère exponentiel. Les performances fluctuent considérablement, même après 15 000 itérations, avec un maximum d'environ 35 récompenses par tranche de 100 itérations. Cette différence marquée met en évidence la difficulté accrue de l'apprentissage dans un environnement stochastique, où les actions de l'agent n'ont pas toujours les effets escomptés.

La comparaison de ces deux graphiques souligne l'impact significatif du glissement sur le processus d'apprentissage :

- Sans glissement, l'agent converge rapidement vers une politique optimale et maintient des performances stables.
- Avec glissement, l'agent doit constamment s'adapter à l'incertitude de l'environnement, ce qui se traduit par des performances plus faibles et plus variables.

Les figures 5 et 6 illustrent les performances de l'agent après la phase d'entraînement, respectivement avec et sans glissement et confirment l'analyse précédente.

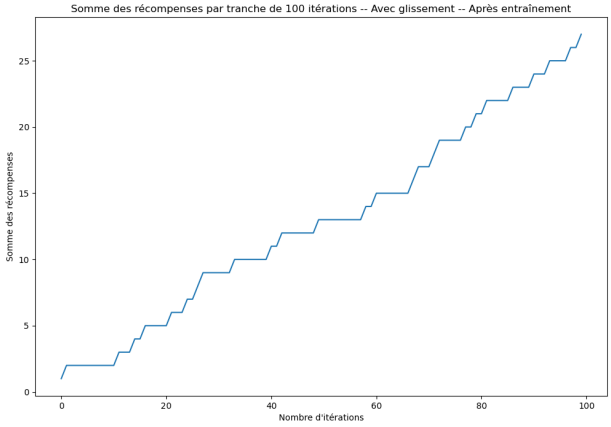


FIGURE 5 – Somme des récompenses par tranche de 100 itérations – Avec glissement – Après entraînement

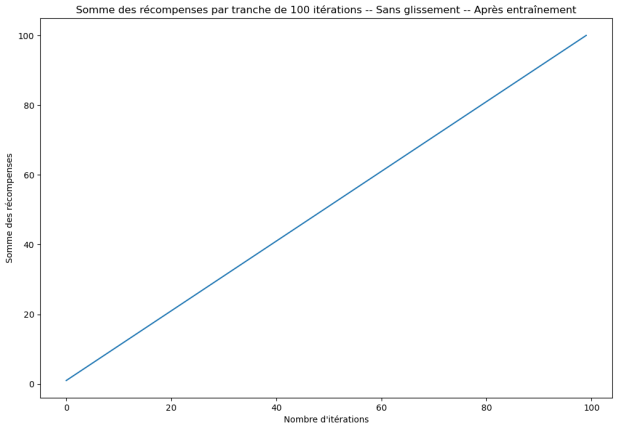


FIGURE 6 – Somme des récompenses par tranche de 100 itérations – Sans glissement – Après entraînement

5.3.2 Analyse du nombre d'actions par épisode

Pour approfondir notre compréhension du comportement de l'agent, nous avons également analysé le nombre d'actions effectuées par épisode dans les environnements avec et sans glissement.

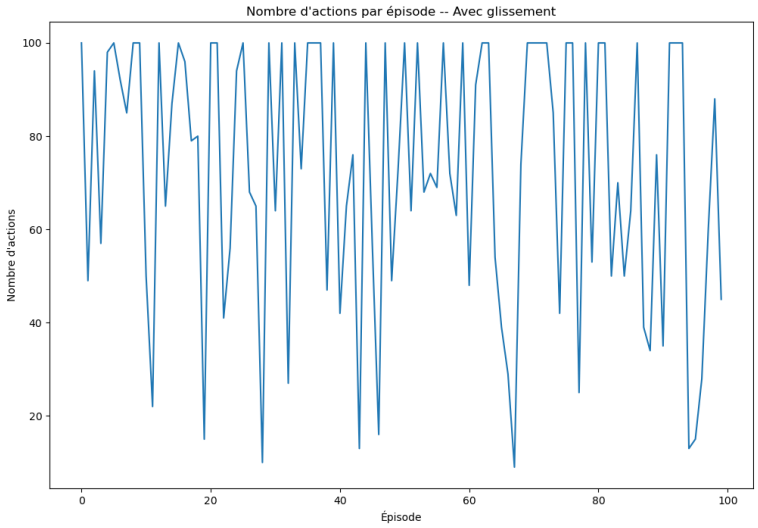


FIGURE 7 – Nombre d'actions par épisode – Avec glissement

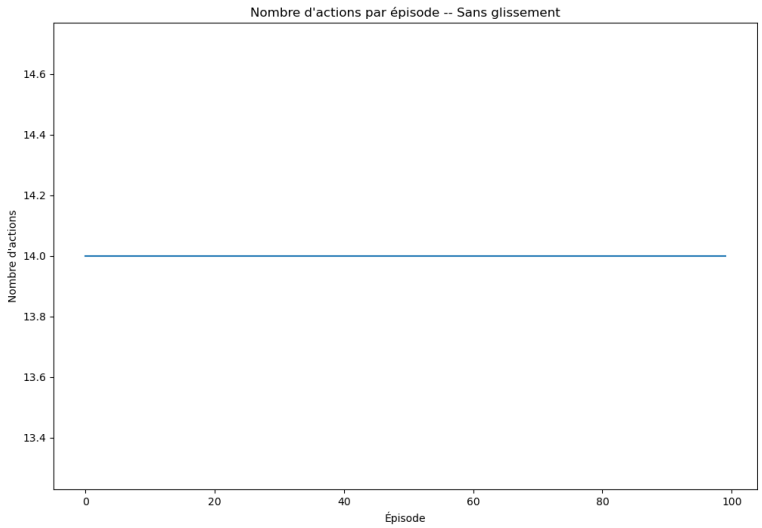


FIGURE 8 – Nombre d'actions par épisode – Sans glissement

La figure 7 illustre le nombre d'actions par épisode dans l'environnement avec glissement. On observe une grande variabilité, avec un nombre d'actions oscillant entre environ 10 et 100 par épisode. Cette forte variation reflète la nature stochastique de l'environnement :

- Les épisodes avec peu d'actions (10-20) suggèrent que l'agent est tombé dans un trou ou a eu beaucoup de chance.
- Les épisodes avec beaucoup d'actions (80-100) indiquent des situations où l'agent a du mal à naviguer efficacement, probablement en raison de glissements défavorables mais semble avoir trouvé la récompense.
- La distribution irrégulière des pics montre que l'agent doit constamment s'adapter à des situations imprévisibles.

En contraste, la figure 8 montre une ligne parfaitement droite à 14 actions par épisode dans l'environnement sans glissement. Cette constance remarquable indique que :

- L'agent a trouvé un chemin optimal et déterministe pour atteindre l'objectif.
- Ce chemin est répété de manière identique à chaque épisode, démontrant une politique parfaitement stable et prévisible.
- L'absence de variation confirme la nature déterministe de l'environnement sans glissement.

5.3.3 Analyse comparative des Q-tables

L'examen des Q-tables nous permet d'approfondir notre compréhension des stratégies apprises par l'agent dans les environnements avec et sans glissement.

Environnement sans glissement Voici un extrait de la Q-table pour l'environnement sans glissement, montrant les 8 premiers états :

TABLE 2 – Extrait de la Q-table sans glissement

État	Gauche	Bas	Droite	Haut
0	0.2288	0.2542	0.2542	0.2288
1	0.2288	0.2824	0.2824	0.2542
2	0.2542	0.3138	0.3138	0.2824
3	0.2824	0.3487	0.3487	0.3138
4	0.3138	0.3874	0.3874	0.3487
5	0.3487	0.4305	0.4305	0.3874
6	0.3874	0.4783	0.4783	0.4305
7	0.4305	0.5314	0.4783	0.4783

Environnement avec glissement Voici un extrait similaire pour l'environnement avec glissement :

TABLE 3 – Extrait de la Q-table avec glissement

État	Gauche	Bas	Droite	Haut
0	1.09e-04	1.15e-04	1.69e-03	1.04e-04
1	1.11e-04	1.36e-04	7.59e-04	1.47e-04
2	1.48e-04	8.83e-04	1.62e-04	1.59e-04
3	2.82e-04	2.43e-04	1.76e-03	2.24e-04
4	4.08e-04	3.85e-04	2.82e-03	4.07e-04
5	6.63e-04	6.13e-04	5.79e-03	6.09e-04
6	1.30e-03	1.14e-03	9.86e-03	1.23e-03
7	8.43e-03	1.29e-03	1.30e-03	1.26e-03

Interprétation En comparant ces deux tables, nous pouvons faire plusieurs observations importantes :

1. **Ordre de grandeur** : Les valeurs dans la Q-table sans glissement sont nettement plus élevées (de l'ordre de 0.1 à 0.5) que celles avec glissement (de l'ordre de 10^{-4} à 10^{-3}).
2. **Différenciation des actions** : Dans le cas sans glissement, on observe souvent une action clairement préférée (valeur plus élevée) pour chaque état. Par exemple, pour l'état 7, l'action "Bas" a la valeur la plus élevée (0.5314). En revanche, dans le cas avec glissement, les différences entre les actions sont moins marquées.
3. **Progression des valeurs** : Sans glissement, on observe une augmentation générale des valeurs Q à mesure qu'on s'approche de l'état 7 (dernière case de la première ligne), ce qui suggère une politique claire pour atteindre l'objectif. Avec glissement, cette progression est moins évidente.
4. **Stabilité de l'apprentissage** : Les valeurs plus cohérentes et structurées dans le cas sans glissement indiquent un apprentissage plus stable et déterministe. Les valeurs plus variables et généralement plus faibles avec glissement reflètent l'incertitude introduite par l'environnement stochastique.

5.3.4 Conclusion de l'analyse

Cette analyse approfondie des performances de l'agent Q-learning dans les environnements FrozenLake avec et sans glissement met en lumière plusieurs aspects cruciaux de l'apprentissage par renforcement :

1. **Impact de la stochasticité** : L'introduction du glissement transforme radicalement la nature du problème, passant d'un environnement déterministe à un environnement stochastique. Cela se traduit par une diminution significative des performances et une augmentation de la variabilité des résultats.
2. **Adaptabilité de l'algorithme** : Bien que les performances soient moins impressionnantes dans l'environnement avec glissement, l'agent parvient néanmoins à apprendre et à s'améliorer, démontrant la robustesse de l'algorithme Q-learning face à l'incertitude.
3. **Stratégies d'apprentissage** : L'analyse des Q-tables révèle des différences fondamentales dans les stratégies apprises. Dans l'environnement déterministe, l'agent développe une politique claire et confiante, tandis que dans l'environnement stochastique, il adopte une approche plus nuancée et adaptative.
4. **Complexité de l'apprentissage** : Le nombre d'actions par épisode et l'évolution des récompenses soulignent la complexité accrue de l'apprentissage dans un environnement stochastique, nécessitant plus d'itérations et aboutissant à des performances moins stables.

Ces observations soulignent l'importance de considérer la nature de l'environnement lors de la conception et de l'évaluation des algorithmes d'apprentissage par renforcement. Elles mettent également en évidence le besoin de développer des approches plus sophistiquées pour traiter efficacement les environnements stochastiques, qui sont plus représentatifs des défis du monde réel.

5.4 Environnement CartPole-v1 avec un algorithme Q profond

L'environnement CartPole-v1 est un environnement classique de contrôle utilisé pour tester des algorithmes d'apprentissage par renforcement. Le problème consiste à contrôler un chariot auquel est attaché un pendule. Le but de l'agent est de maintenir le pendule à la verticale en appliquant des forces sur le chariot.

L'agent peut choisir entre deux actions :

— **Action 0** : Pousser le chariot vers la gauche.

— **Action 1** : Pousser le chariot vers la droite.

L'observation est un tableau contenant les quatre variables suivantes :

— **Position du chariot** : $[-4.8, 4.8]$

— **Vitesse du chariot** : $(-\infty, \infty)$

— **Angle du pendule** : $[-0.418 \text{ rad}, 0.418 \text{ rad}]$, soit environ $[-24°, 24°]$

— **Vitesse angulaire du pendule** : $(-\infty, \infty)$

Remarque : Si l'angle du pendule dépasse $\pm 12°$ ou que le chariot se déplace au-delà de ± 2.4 unités de sa position d'origine, l'épisode se termine.

L'agent reçoit une récompense de $+1$ à chaque étape réussie, y compris à l'étape de terminaison. L'objectif est de maximiser le nombre d'étapes pendant lesquelles le pendule reste debout.

L'épisode se termine lorsque :

— L'angle du pendule dépasse $\pm 12°$.

— La position du chariot dépasse ± 2.4.

— L'épisode dure plus de 500 étapes.

5.4.1 Création de l'environnement CartPole-v1

```
env = gym.make("CartPole-v1")
```

La fonction `gym.make("CartPole-v1")` crée l'environnement `CartPole-v1`

```
input_shape = [4]
output_shape = 2
```

L'entrée du réseau est un vecteur de dimension 4, où chaque valeur représente une caractéristique de l'état :

— Position du chariot,

— Vitesse du chariot,

— Angle du pendule,

— Vitesse angulaire du pendule.

La sortie du réseau est un vecteur de dimension 2 correspondant aux actions possibles :

— 0 : Appliquer une force vers la gauche,

— 1 : Appliquer une force vers la droite.

5.5 Création du modèle de réseau de neurones

```
model = keras.models.Sequential([
    keras.layers.Dense(32, activation='relu', input_shape = input_shape),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dense(output_shape)
])
```

Le modèle est créé avec `keras` de manière séquentielle :

- `keras.layers.Dense(32, activation='relu', input_shape = input_shape)` : Première couche dense avec 32 neurones et une fonction d'activation ReLU. Elle prend une entrée de dimension 4 (l'état de `CartPole-v1`).
- `keras.layers.Dense(32, activation='relu')` : Deuxième couche dense avec 32 neurones et la même fonction d'activation.
- `keras.layers.Dense(output_shape)` : Dernière couche avec 2 neurones de sortie (correspondant aux deux actions possibles). Aucune fonction d'activation n'est nécessaire, car il s'agit de la prédiction des valeurs Q.

5.5.1 Fonctions pour l'implémentation de l'algorithme DQN

Fonction `epsilon_greedy()` :
Cette fonction permet à l'agent de choisir une action selon une politique *epsilon-greedy*. L'agent choisit une action aléatoire avec une probabilité ϵ, et avec une probabilité $1 - \epsilon$, il sélectionne l'action avec la valeur $Q(s, a)$ la plus élevée, prédite par le modèle.

Fonction `sample_experiences()` :
Cette fonction permet de prélever un échantillon d'expériences à partir du *replay buffer*. Elle retourne les états, actions, récompenses, états suivants, ainsi que les indicateurs d'achèvement ou de troncature des épisodes pour un lot d'expériences.

Fonction `play_one_step()` :
Cette fonction permet à l'agent d'effectuer une action dans l'environnement et de stocker l'expérience correspondante dans le *replay buffer*. L'agent choisit l'action en utilisant la stratégie *epsilon-greedy* puis il interagit avec l'environnement et enfin retourne les informations associées à cette interaction.

Fonction `training_one_step()` :
Cette fonction implémente une étape d'apprentissage pour ajuster les poids du modèle. Elle utilise un échantillon d'expériences prélevé à partir du *replay buffer*, et met à jour les poids du modèle en fonction de l'erreur entre les valeurs Q prédites et les cibles de valeurs Q (basées sur l'équation de Bellman).

Boucle principale d'entraînement :
La boucle d'entraînement s'exécute sur plusieurs épisodes (`for episode in range(600)`). Pour chaque épisode, l'agent interagit avec l'environnement jusqu'à ce que l'épisode soit terminé. Les récompenses et le nombre d'étapes sont enregistrés, et les poids du modèle sont mis à jour à des intervalles réguliers (tous les 50 épisodes).

5.5.2 Analyse des résultats

Dans cette section, nous présentons les résultats obtenus en fonction des différents paramètres choisis. Pour ce premier entraînement, nous avons opté pour un paramètre d'exploration $\epsilon = 0.1$. Cette valeur relativement élevée encourage l'agent à explorer activement tout au long de l'entraînement. L'entraînement s'étend sur 2000 époques, une durée raisonnable pour observer des améliorations progressives dans la performance de l'agent. Nous nous attendons à ce que l'agent montre une augmentation significative de ses gains au fur et à mesure de l'entraînement, à mesure qu'il équilibre exploration et exploitation.

Voici un graphe montrant l'évolution de epsilon suivi de la répartition des gains en fonction de l'époque :

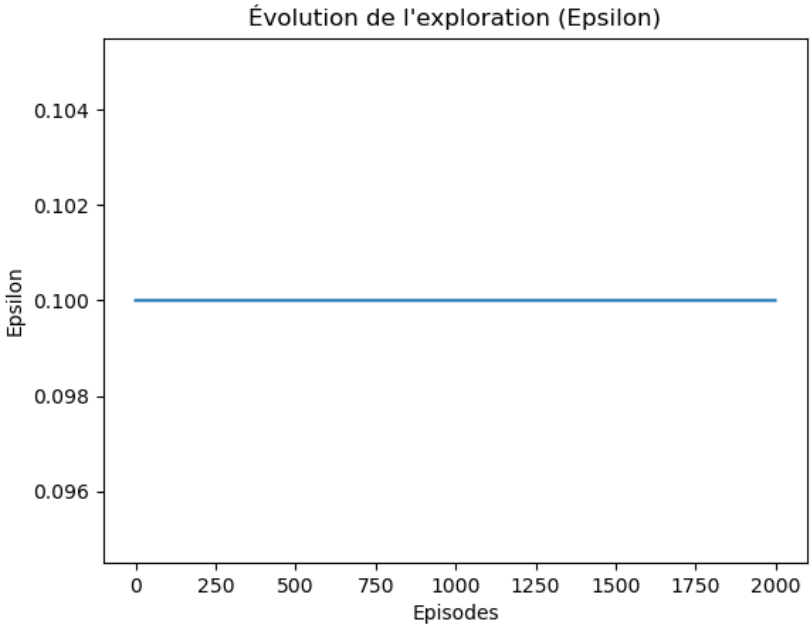


FIGURE 9 – Évolution de epsilon sur 2000 époques

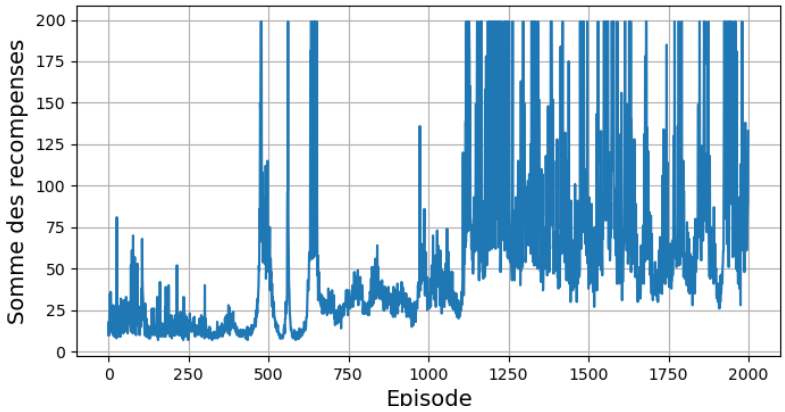


FIGURE 10 – Récompenses obtenues en fonction de l'époque

Les deux graphiques ci-dessus illustrent deux aspects clés du processus d'apprentissage de l'agent. Ce premier graphique montre l'évolution du taux d'exploration ϵ au cours de l'entraînement. La valeur d'ϵ est restée constante à 0.1 tout au long des 2000 épisodes. Cela indique que l'agent a maintenu une probabilité fixe de 10% pour explorer une action aléatoire à chaque étape. Cette stratégie encourage une exploration continue de l'environnement, ce qui peut être utile dans des situations où l'agent n'a pas encore complètement exploré l'espace d'état. Toutefois, cela peut également ralentir la convergence de l'agent vers une politique optimale, car il continue d'explorer même lorsqu'il aurait pu se concentrer davantage sur l'exploitation des actions déjà connues pour offrir de bonnes récompenses.

Le deuxième graphique montre la somme des récompenses obtenues par l'agent à chaque épisode. Il y a une grande variabilité dans les récompenses pendant les 1000 premiers épisodes, avec des valeurs relativement faibles et irrégulières. Cependant, à partir de l'épisode 1000, on observe une augmentation plus régulière des récompenses, avec des pics fréquents. Cela indique que l'agent commence à apprendre et à mieux naviguer dans l'environnement et améliore ainsi ses performances au fil du temps. Toutefois, les fluctuations présentes montrent que l'agent n'a pas encore complètement stabilisé ses gains, probablement en raison de l'exploration persistante ($\epsilon = 0.1$).

En comparant les deux graphiques, il est évident que malgré un taux d'exploration constant, l'agent parvient tout de même à apprendre et à améliorer ses performances. Cependant, la stabilisation des récompenses pourrait être améliorée en diminuant progressivement ϵ, permettant à l'agent de privilégier l'exploitation de ses connaissances plutôt que de continuer à explorer des actions aléatoires. Voyons alors le résultat pour un ϵ doté d'un facteur de diminution.

Voici les résultats obtenus :

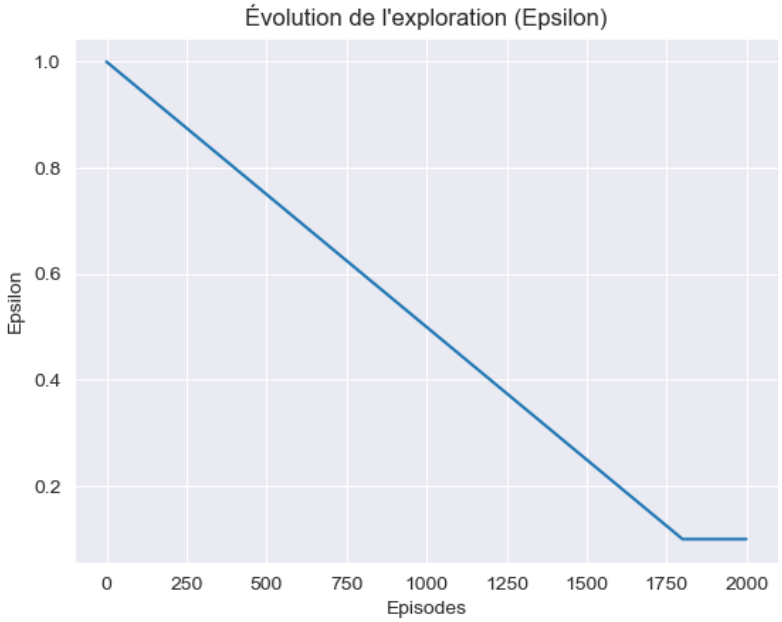


FIGURE 11 – Évolution de epsilon sur 2000 époques

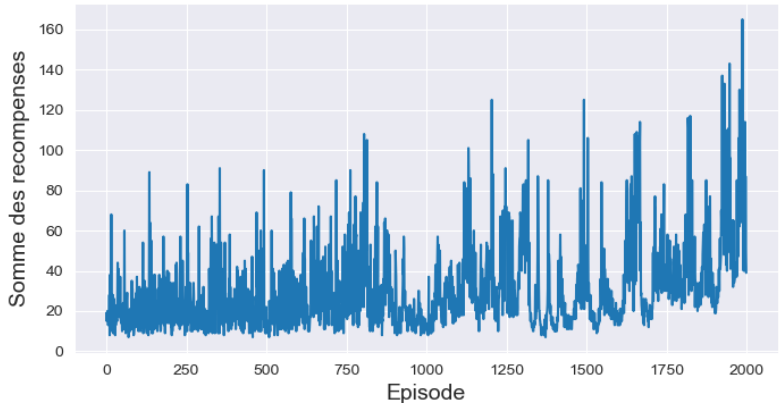


FIGURE 12 – Récompenses obtenues en fonction de l'époque

Le premier graphique montre une décroissance linéaire de ϵ qui passe de 1.0 à 0.01. Cela reflète une stratégie classique en apprentissage par renforcement : au début, l'agent explore beaucoup pour découvrir son environnement, puis il se concentre progressivement sur l'exploitation des connaissances acquises. Au début de l'entraînement, l'agent fait principalement des choix aléatoires (exploration), tandis qu'à la fin, il privilégie les actions optimales avec une probabilité très élevée (exploitation).

Le deuxième graphique montre une amélioration progressive des performances de l'agent. Au début, les récompenses sont faibles et varient fortement, ce qui est typique lors des phases d'exploration intensive où l'agent n'a pas encore identifié les actions optimales. À mesure que l'entraînement progresse, les récompenses augmentent de manière plus constante, avec des pics fréquents après 1000 épisodes. Ces pics montrent que l'agent a réussi à identifier des séquences d'actions optimales

qui conduisent à de meilleures performances. Cependant, une certaine variabilité persiste, surtout vers la fin, probablement en raison de la faible valeur de ϵ, ce qui continue de permettre un peu d'exploration malgré l'acquisition d'une bonne stratégie.

Les résultats ci-dessous montrent les récompenses moyennes obtenues (le nombre d'étapes durant lesquelles le bâton reste en équilibre) sur les 2000 épisodes d'entraînement, ainsi que des moyennes distinctes pour les 1000 premiers et 1000 derniers épisodes.

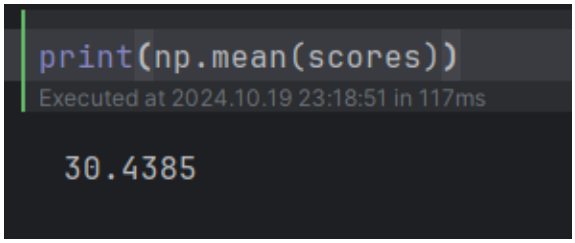


FIGURE 13 – Récompense moyenne sur les 2000 épisodes

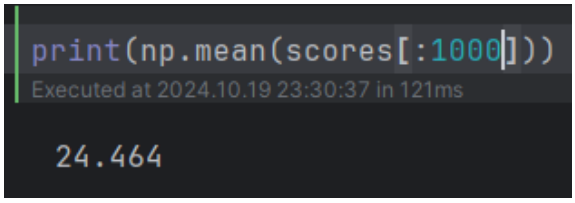


FIGURE 14 – Récompense moyenne sur les 1000 premiers épisodes

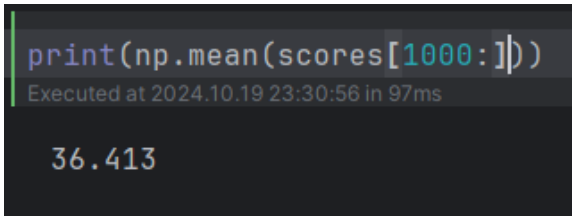


FIGURE 15 – Récompense moyenne sur les 1000 derniers épisodes

— Sur l'ensemble des 2000 épisodes, la récompense moyenne globale est de 30.44. Ce résultat indique que l'agent a réussi, en moyenne, à maintenir l'équilibre du bâton pendant environ 30 étapes avant la fin de chaque épisode.

— Pour les 1000 premiers épisodes, la récompense moyenne est plus faible, à 24.46. Cela montre qu'au début de l'entraînement, l'agent avait plus de difficultés à trouver des stratégies efficaces pour maintenir l'équilibre du bâton.

— En revanche, pour les 1000 derniers épisodes, la récompense moyenne s'élève à 36.41 donc cela indique une nette amélioration des performances de l'agent au fil du temps. Cette augmentation montre que l'agent a progressivement appris à mieux maintenir l'équilibre du bâton et à optimiser ses actions dans son environnement.

Ces résultats montrent l'importance du paramètre ϵ et sa gestion progressive pour permettre à l'agent d'apprendre efficacement. L'agent explore intensivement au début puis exploite de plus

en plus ses connaissances, ce qui se traduit par des performances croissantes au fur et à mesure de l'entraînement. Il faut alors trouver le juste milieu entre exploration et exploitation. Cependant, il aurait été intéressant de prolonger l'entraînement pour observer si l'agent parvient à stabiliser ses gains et à maximiser ses performances sur une plus longue période.

Conclusion

Les avancées en apprentissage par renforcement profond, illustrées par des succès tels qu'AlphaGo, ont marqué un tournant majeur dans le domaine de l'intelligence artificielle. Ce mémoire a eu pour objectif d'explorer le Deep Q-Learning, en présentant ses mécanismes et ses défis. Des notions fondamentales, notamment les processus de décision markoviens et l'équilibre entre exploration et exploitation, ont été abordées. À travers des expérimentations sur des environnements interactifs, il a été démontré que les réseaux Q profonds constituent une solution efficace pour traiter des espaces d'état vastes et incertains. Ces techniques sont un pilier essentiel pour l'amélioration des agents intelligents et ouvrent la voie à des applications encore plus sophistiquées dans des domaines variés.

Annexes

Annexe A : Frozen-Lake 8x8

```
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import pickle


def run(episodes, is_training=True, render=False):
    env = gym.make('FrozenLake-v1', map_name="8x8", is_slippery=True,
    render_mode='human' if render else None)

    if (is_training):
        q = np.zeros((env.observation_space.n, env.action_space.n))
    else:
        with open('frozen_lake8x8.pkl', 'rb') as f:
            q = pickle.load(f)

    learning_rate_a = 0.9
    discount_factor_g = 0.9
    epsilon = 1
    epsilon_decay_rate = 0.0001
    rng = np.random.default_rng()

    rewards_per_episode = np.zeros(episodes)
    actions_per_episode = np.zeros(episodes)
    # Nouveau tableau pour stocker le nombre d'actions

    for i in range(episodes):
        state = env.reset()[0]
        terminated = False
        truncated = False
        action_count = 0  # Compteur d'actions pour cet épisode

        while not terminated and not truncated:
            if is_training and rng.random() < epsilon:
                action = env.action_space.sample()
            else:
                action = np.argmax(q[state, :])

            new_state, reward, terminated, truncated, _ = env.step(action)
            action_count += 1  # Incrémente le compteur d'actions
```

```
            if is_training:
                q[state, action] = q[state, action] + learning_rate_a * (
                        reward + discount_factor_g * np.max(q[new_state, :]) -
                        q[state, action]
                )

            state = new_state

        epsilon = max(epsilon - epsilon_decay_rate, 0)
        if epsilon == 0:
            learning_rate_a = 0.0001

        if reward == 1:
            rewards_per_episode[i] = 1

        actions_per_episode[i] = action_count  # Enregistre le nombre d'actions pour cet épisode

    env.close()

    # Calcul et affichage du nombre moyen d'actions
    avg_actions = np.mean(actions_per_episode)
    print(f"Nombre moyen d'actions par épisode : {avg_actions:.2f}")

    # Graphique des récompenses
    sum_rewards = np.zeros(episodes)
    for t in range(episodes):
        sum_rewards[t] = np.sum(rewards_per_episode[max(0, t - 99):(t + 1)])

    plt.figure(figsize=(12, 8))
    plt.plot(sum_rewards)
    plt.title('Somme des récompenses par tranche de 100 itérations -- Sans glissement -- Après ent
    plt.xlabel("Nombre d'itérations")
    plt.ylabel('Somme des récompenses')
    plt.savefig('frozen_lake8x8--sans_glissement -- apres_entrainement.png')
    plt.close()

    # Graphique du nombre d'actions par épisode
    plt.figure(figsize=(12, 8))
    plt.plot(actions_per_episode)
    plt.title("Nombre d'actions par épisode -- Avec glissement")
    plt.xlabel('Épisode')
    plt.ylabel("Nombre d'actions")
    plt.savefig('frozen_lake8x8--nombre_actions_par_episode -- Avec_glissement.png')
    plt.close()
```

```
    if is_training:
        with open("frozen_lake8x8.pkl", "wb") as f:
            pickle.dump(q, f)


if __name__ == '__main__':
    run(15000, is_training=True, render=False)
```

Annexe B : CartPole-v1

```
def politique_epsilon_greedy(etat, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(2)
    else:
        print(type(etat))
        valeurs_Q = modele.predict(etat[np.newaxis])
        print(valeurs_Q.shape)
        return np.argmax(valeurs_Q[0])

from collections import deque

memoire_rejeu = deque(maxlen=200)

def echantillon_experiences(taille_lot):
    indices = np.random.randint(len(memoire_rejeu), size=taille_lot)
    lot = [memoire_rejeu[index] for index in indices]
    etats, actions, recompenses, prochains_etats, termines, tronques = [
        np.array([experience[champ_index] for experience in lot])
        for champ_index in range(6)
    ]
    return etats, actions, recompenses, prochains_etats, termines, tronques

def jouer_une_etape(env, etat, epsilon):
    action = politique_epsilon_greedy(etat, epsilon)
    prochain_etat, recompense, termine, tronque, info = env.step(action)
    memoire_rejeu.append((etat, action, recompense, prochain_etat, termine, tronque))
    return prochain_etat, recompense, termine, tronque, info

taille_lot = 32
facteur_escompte = 0.95
optimiseur = keras.optimizers.Adam(learning_rate=0.001)
fonction_perte = keras.losses.mean_squared_error

modele_cible = tf.keras.models.clone_model(modele)
modele_cible.set_weights(modele.get_weights())

recompenses = []
```

```
meilleur_score = 0

def entrainement_une_etape(taille_lot):
    exp = echantillon_experiences(taille_lot)
    etats, actions, recompenses, prochains_etats, termines, tronques = exp
    prochaines_valeurs_q = modele_cible.predict(prochains_etats, verbose=0)
    max_q = np.max(prochaines_valeurs_q, axis=1)
    cible_q = (recompenses + (1 - termines) * facteur_escompte * max_q)
    masque = tf.one_hot(actions, output_shape)

    with tf.GradientTape() as bande:
        toutes_valeurs_q = modele.predict(etats)
        valeurs_Q = tf.reduce_sum(toutes_valeurs_q * masque, axis=1, keepdims=True)
        perte = tf.reduce_mean(fonction_perte(cible_q, valeurs_Q))
    gradients = bande.gradient(perte, modele.trainable_variables)
    optimiseur.apply_gradients(zip(gradients, modele.trainable_variables))

for episode in range(600):
    obs, info = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, recompense, termine, tronque, info = jouer_une_etape(env, obs, epsilon)
        if termine or tronque:
            break
    recompenses.append(step)
    if step >= meilleur_score:
        meilleurs_poids = modele.get_weights()
        meilleur_score = step

    if episode % 50:
        modele_cible.set_weights(modele.get_weights())

modele.set_weights(meilleurs_poids)
```

Bibliographie

Zhang, S., Whiteson, S. (2016). Exploring Deep Reinforcement Learning with Multi-Q Learning. *ResearchGate*. https://www.researchgate.net/publication/310429350_Exploring_Deep_Reinforcement_Learning_with_Multi_Q-Learning

Jaël Gareau, *Processus de décision Markovien (MDP)*, 2024. Disponible en ligne : https://www.jaelgareau.com/fr/project/mdp/. Consulté le 10 octobre 2024.

DeepMind. *Agent57 : Outperforming the Human Atari Benchmark*. 2020. Consulté le 11 octobre 2024. URL : https://deepmind.google/discover/blog/agent57-outperforming-the-human-atari-benchmark/.