

T-AIA-902

Delivery

Groupe N°1

1. Introduction	3
2. Explication de la démarche	5
3. Métriques	6
4. Algorithmes	10
4.1. Brute Force	10
4.1.1. Présentation	10
4.1.2. Intérêt dans le projet	10
4.2. Monte Carlo	11
4.2.1. Présentation	11
4.2.2. Résultats avec les paramètres optimaux	11
4.2.3. Conclusion	12
4.3. SARSA	13
4.3.1. Présentation	13
4.3.2. Résultats avec les paramètres optimaux	13
4.3.3. Nombre d'éisodes	15
4.3.3.1. Paramètre bas (1 000 épisodes)	15
4.3.3.2. Paramètre élevé (50 000 épisodes)	16
4.3.4. Gamma (paramètre bas (0,85))	17
4.3.5. Epsilon Greedy (paramètre bas (0,1))	18
4.3.6. Decay Epsilon Greedy (paramètre haut (0,1))	19
4.3.7. Conclusion	19
4.4. Q-Learning	21
4.4.1. Présentation	21
4.4.2. Intérêt dans le projet	21
4.4.3. Résultats avec les paramètres optimaux	22
4.4.4. Nombre d'éisodes (paramètre bas (1 000 épisodes))	23
4.4.5. Epsilon (paramètre bas (0,1))	24
4.4.6. Decay Epsilon Greedy (paramètre haut (0,1))	25
4.4.7. Gamma (paramètre bas (0,85))	26
4.4.8. Conclusion	26
4.5. Value Iteration	27
4.5.1. Présentation de l'algorithme	27
4.5.2. Expérimentation	27
4.5.3. Conclusion	29
4.6. Deep Q-Learning	30
4.7. Temps d'entraînement	31
5. Conclusion	34

1. Introduction

Vous retrouverez à l'intérieur de ce document les différents résultats obtenus ainsi que les réflexions en lien avec le projet Taxi Driver.

Ce projet vise à optimiser les déplacements et les décisions d'un chauffeur de taxi en utilisant diverses méthodes d'apprentissage par renforcement.

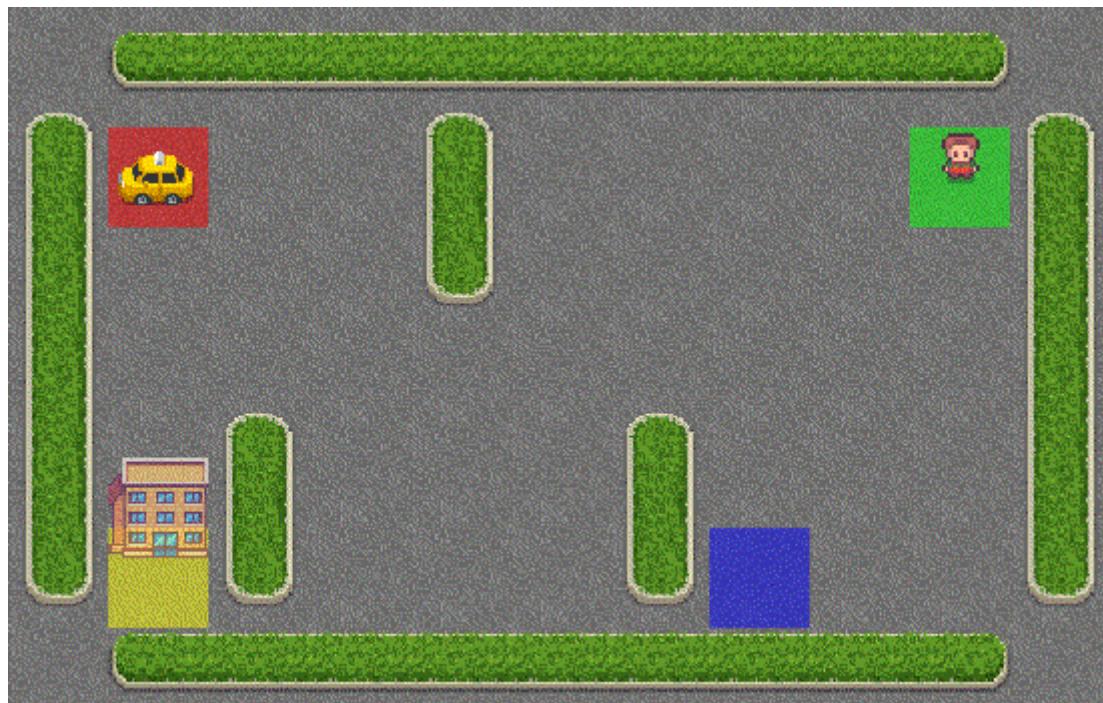
Nous utilisons dans ce projet l'API Gym qui est une bibliothèque en Python conçue pour faciliter la création, la simulation et l'évaluation d'environnements pour l'apprentissage par renforcement (RL, Reinforcement Learning).

Elle est développée par OpenAI et elle propose une interface standardisée pour interagir avec divers environnements, permettant de tester et de comparer facilement différents algorithmes de RL.

Gym offre une large gamme d'environnements pré-construits et permet de créer des environnements personnalisés.

Elle simplifie le processus d'entraînement des agents en fournissant des outils pour surveiller et évaluer leurs performances.

L'environnement étudié dans ce projet est celui du Taxi de Gym, qui ressemble à ceci :



Le but est de contrôler un taxi pour récupérer un passager à un endroit donné et le déposer à une destination spécifiée **en effectuant le trajet le plus court possible.**

L'environnement est représenté par une grille de taille fixe. Le taxi peut se trouver dans l'une des cases de manière aléatoire. Il y a quatre emplacements possibles pour les passagers et les destinations, marqués par les lettres "R", "G", "B" et "Y" (Red, Green, Blue, Yellow). L'état complet est défini par la position du taxi, la position du passager et la destination de ce dernier.

L'agent dispose d'un nombre limité d'actions dans son environnement ce qui nous permettra par la suite de jouer dessus pour améliorer nos modèles.

- **0** : Se déplacer vers le sud
- **1** : Se déplacer vers le nord
- **2** : Se déplacer vers l'est
- **3** : Se déplacer vers l'ouest
- **4** : Prendre un passager
- **5** : Déposer un passager

Selon l'action choisi, l'agent sera récompensé différemment:

- l'agent se déplace: récompense de -1
- l'agent réalise une mauvaise action (mauvais dépôt, mauvaise récupération du passager): récompense de -10
- l'agent gagne la partie (récupère le passager et le dépose au bon endroit): récompense de 20

L'objectif de ce document est de lister les algorithmes utilisés tout au long du projet, de décrire la démarche méthodologique mise en place, et de présenter les résultats obtenus ainsi que les conclusions tirées de ces études.

Nous explorerons en détail les différentes techniques d'apprentissage, leurs implémentations, les défis rencontrés, et les solutions apportées.

Ce document a pour but de fournir une compréhension approfondie des approches adoptées.

2. Explication de la démarche

Dans le cadre de nos recherches, plusieurs algorithmes se sont révélés efficaces dans le traitement de notre problématique, à savoir optimiser le trajet d'un taxi dans un environnement donné. Ces algorithmes utilisent des notions diverses pour calculer progressivement une **politique** permettant à un **agent** d'évoluer dans un environnement et de le résoudre en prenant des décisions selon ce qu'il a pu mémoriser.

Afin de tester tous ces algorithmes sur une base la plus proche possible, nous les avons implémentés selon une base commune. Pour expérimenter l'impact des différents paramètres relatifs à l'apprentissage par renforcement sur l'entraînement, nous avons modifié unitairement les valeurs d'entrée de ces paramètres. L'idée est de faire le moins de changement entre les entraînements pour avoir des comparaisons les plus intéressantes possible à la fin. Nous avons également comparé la performance des algorithmes avec des paramètres équivalents.

Pour définir des paramètres initiaux, nous nous sommes basés sur ce qui existait déjà dans le domaine de l'apprentissage par renforcement. Ces paramètres se sont souvent révélés optimaux.

Le domaine de l'apprentissage renforcé n'étant pas nouveau, de nombreuses études sur le sujet traitent déjà des raisons pour lesquelles utiliser certains paramètres et comment les définir. Notre travail a plutôt consisté à modifier ces valeurs pour voir l'impact que cela peut avoir sur les modèles.

Exemple de paramètres optimaux pour Q-Learning et SARSA: *Épisodes : 10 000, Learning Rate : 0,85, Gamma : 0,99, Epsilon : 1, Min Epsilon : 0,001, Max Steps: 100*

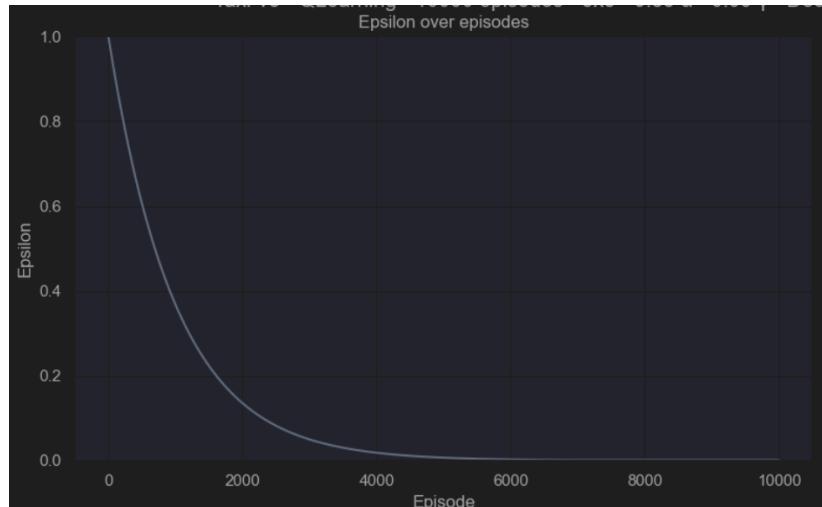
Dans la suite de ce document, nous allons traiter des différents algorithmes en modifiant les paramètres de ceux-ci tout en étudiant les résultats que cela apporte. Pour rendre l'étude des modèles plus intéressante, nous allons afficher uniquement les paramètres qui ont eu le plus d'impact dans le traitement de chaque algorithme.

Les paramètres principaux qui ont été modifiés sont :

- Le nombre d'épisode,
- Le learning rate
- Gamma
- Epsilon
- Theta pour l'algorithme de Value Iteration

3. Métriques

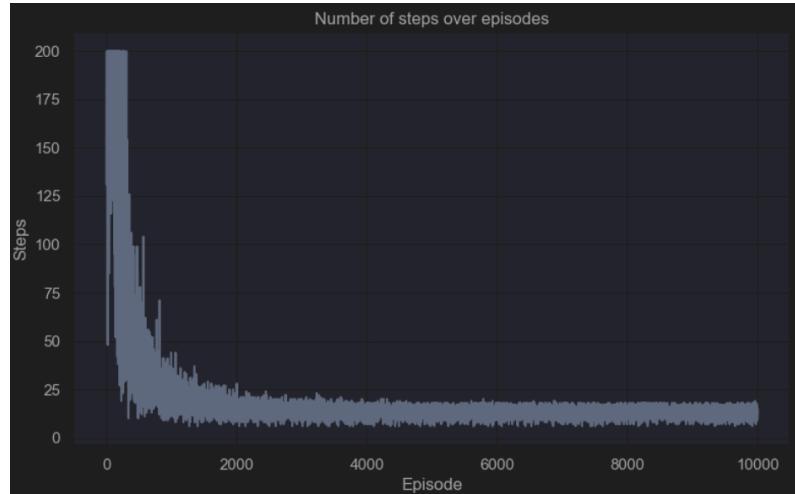
Pour comprendre ce document, nous allons étudier différents graphiques qui seront produits lors de chaque entraînement.



Evolution de la valeur epsilon au cours des épisodes (ici, la valeur décroît dans le cadre de l'utilisation d'une politique “decayed epsilon greedy”)

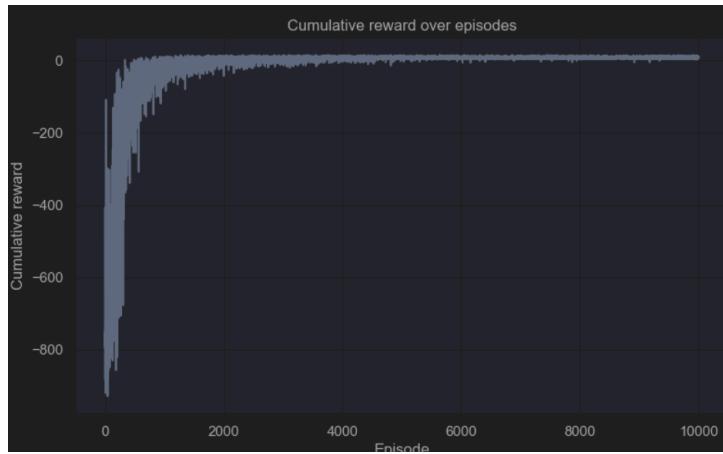
Le premier graphique représente l'évolution de la valeur **epsilon** tout au long de l'entraînement. Cette courbe représente la part d'apprentissage et d'exploration de l'agent pendant l'entraînement. C'est-à-dire que celui-ci va avoir un taux d'apprentissage proportionnel à la valeur d'epsilon. L'utilisation d'epsilon dépend du choix de la politique d'apprentissage. L'epsilon est utilisé dans les politiques “epsilon greedy” ou “decayed epsilon greedy”.

Dans le cas d'un “epsilon greedy”, la valeur d'epsilon est fixe et la proportion d'exploration et d'apprentissage n'évoluera pas. Dans le cas d'un “decayed epsilon greedy”, il faut comprendre que plus le nombre d'épisodes avance, plus la valeur d'epsilon décroît. L'agent va limiter son exploration de nouveaux chemins et pour se concentrer sur le renforcement des trajets déjà explorés.



Evolution du nombre de pas réalisés par l'agent au cours des épisodes

Celui-ci représente le nombre de pas nécessaire à l'agent (entité qui applique des choix dans un environnement) pour réussir à atteindre son objectif au cours de l'entraînement. Dans ce cas, on peut voir que l'agent a besoin de 1 500 pas avant de se stabiliser. On comprend à travers ce graphique que l'algorithme converge et semble avoir trouvé la solution optimale pour résoudre l'environnement.



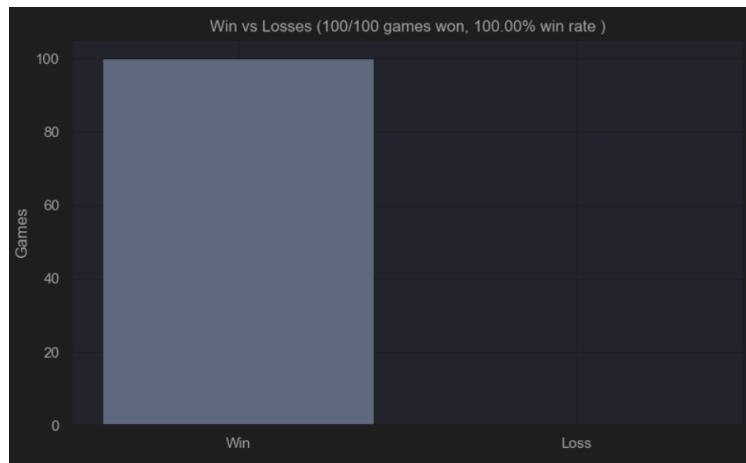
Evolution du nombre de récompenses obtenues par l'agent au cours des épisodes

Le graphique de récompense montre l'évolution de celle-ci au fil des épisodes. La récompense est le paramètre qui intéresse le plus notre modèle, il est celui qui définit si l'action que prend notre agent est bonne ou non.

On utilise ce système de récompense en fonction des actions réalisées, si l'agent fait une bonne action on lui donne une récompense positive et inversement dans le cas échéant.

Le modèle cherchera donc à obtenir une récompense globale la plus proche de 0 ce qui améliorera la qualité de notre modèle. On observe que ce graphique évolue parallèlement à celui de l'évolution du nombre de pas au cours des épisodes.

Plus notre agent est rapide, plus il obtient des récompenses; plus il obtient de récompenses et plus il tendra à réaliser les actions qui l'ont récompensé pour remporter une partie, à savoir dans notre cas, être le plus rapide à conduire un client vers sa destination.

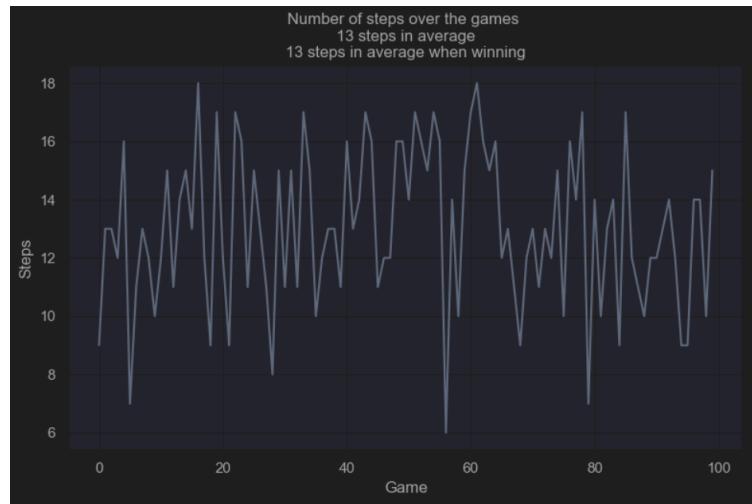


Ratio de parties gagnées et perdues sur 100 parties réalisées après l'entraînement (ici, l'agent a remporté toutes les parties)

Le graphique des victoires est le moyen de déterminer si notre modèle fonctionne.

Lorsqu'un modèle a terminé son entraînement, on va simuler 100 parties pour voir dans combien de cas l'agent arrive à terminer sa partie. Une partie se termine lorsque l'agent arrive à prendre une personne et l'amener au bon endroit ou lorsqu'il réalise plus de déplacement que ce qu'on lui octroie (par défaut, 200 actions par épisode / partie).

Dans le cas où un modèle atteint 100% de réussite, on peut affirmer que notre modèle est fonctionnel. Cela ne signifie pas pour autant qu'il est le plus rapide.



Evolution et moyenne du nombre de pas effectués par l'agent sur 100 parties réalisées (ici, la moyenne de pas est de 13)

Enfin, le dernier graphique permet de suivre l'évolution du nombre de pas moyen que fait l'agent pour réussir une partie.

Le nombre de pas n'est pas constant puisque l'agent évolue dans un **environnement aléatoire (random seed)**. Cela signifie que la position des objectifs (point de ramassage et point de dépôt) n'est pas la même entre deux parties et que le chemin à parcourir peut être plus ou moins long.

4. Algorithmes

4.1. Brute Force

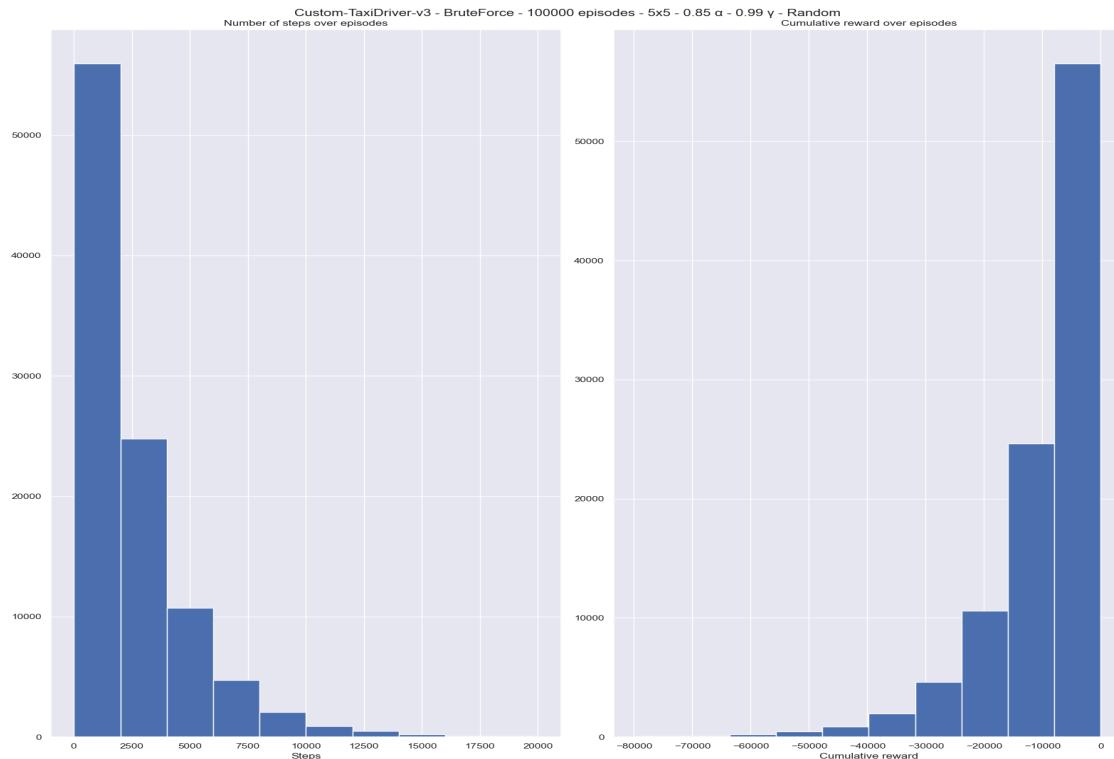
4.1.1. Présentation

L'algorithme de brute force est une méthode simple mais efficace pour résoudre certains problèmes en essayant toutes les possibilités jusqu'à trouver la solution correcte. Dans le cadre d'un choix aléatoire d'actions, l'algorithme de brute force explore toutes les options disponibles et sélectionne une action de manière aléatoire.

4.1.2. Intérêt dans le projet

L'algorithme de brute force offre une méthode simple, impartiale et robuste pour effectuer des choix aléatoires parmi une liste d'actions disponibles. Ses avantages en termes de simplicité d'implémentation, d'exploration exhaustive et d'impartialité en font un outil précieux dans les premières phases de développement et d'exploration de notre projet. En outre, il fournit une base de référence utile pour l'évaluation et l'amélioration de méthodes plus sophistiquées.

Voici la répartition du nombre de pas (*steps*) ainsi que celle du nombre de récompenses (*reward*) pour 100 000 épisodes.



4.2. Monte Carlo

4.2.1. Présentation

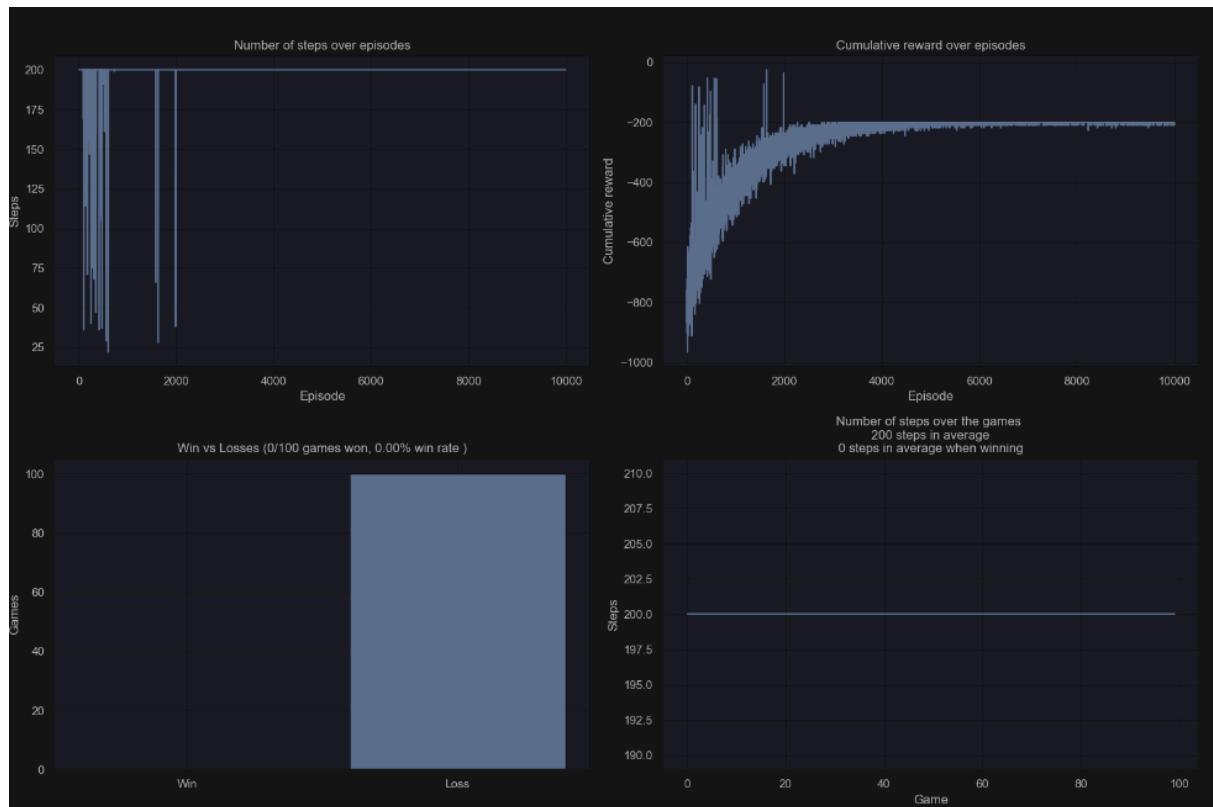
L'apprentissage Monte-Carlo est une méthode en apprentissage par renforcement où l'agent apprend en simulant des épisodes complets.

Plutôt que d'estimer les valeurs des états-actions à chaque pas comme le Q-learning, Monte-Carlo calcule la valeur **en moyennant les récompenses réelles reçues à la fin des épisodes.**

Cela permet à l'agent de prendre des décisions basées sur les récompenses totales accumulées après avoir exploré différentes séquences d'actions dans l'environnement.

Le modèle Monte-Carlo est le premier algorithme “intelligent” que nous avons appliqué au projet.

4.2.2. Résultats avec les paramètres optimaux



Cet algorithme est un peu particulier en comparaison des autres algorithmes vus plus bas. C'est-à-dire que celui-ci attend la fin des épisodes pour mettre à jour ces valeurs. De ce fait nous nous sommes retrouvés face à un conflit à savoir qu'à partir de 1700 épisodes, le modèle a arrêté d'essayer d'apprendre à découvrir son environnement.

Notre réflexion nous a amenée à penser que le modèle entraîné est persuadé qu'il est plus intéressant d'un point de vue récompense de réaliser une même action en boucle (par exemple se déplacer continuellement sur un côté quitte à rester collé à un mur).

Cela expliquerait pourquoi la récompense atteint -200 à partir de 5 000 épisodes puis reste tel quel. L'agent ne prend pas le risque de jouer et de perdre pour apprendre; il préfère minimiser les pertes.

Nous avons essayé de modifier d'autres paramètres mais les résultats n'ont rien apporté d'intéressant à détailler dans ce document.

4.2.3. Conclusion

L'algorithme de Monte-Carlo se révèle être une approche intéressante et méthodique en apprentissage par renforcement, basée sur l'évaluation des performances à la fin des épisodes plutôt qu'à chaque pas.

Il nous offre une première approche des algorithmes de renforcement et est un bon premier point d'entrée pour comprendre le fonctionnement global de l'apprentissage par épisode.

En intégrant cette approche dans notre projet, nous avons pu observer une capacité de l'agent à prendre des décisions stratégiques basées sur des expériences concrètes, ouvrant ainsi la voie à des développements plus sophistiqués que ce soit avec l'algorithme SARSA ou encore le Q-Learning expliqué ci-dessous.

4.3. SARSA

4.3.1. Présentation

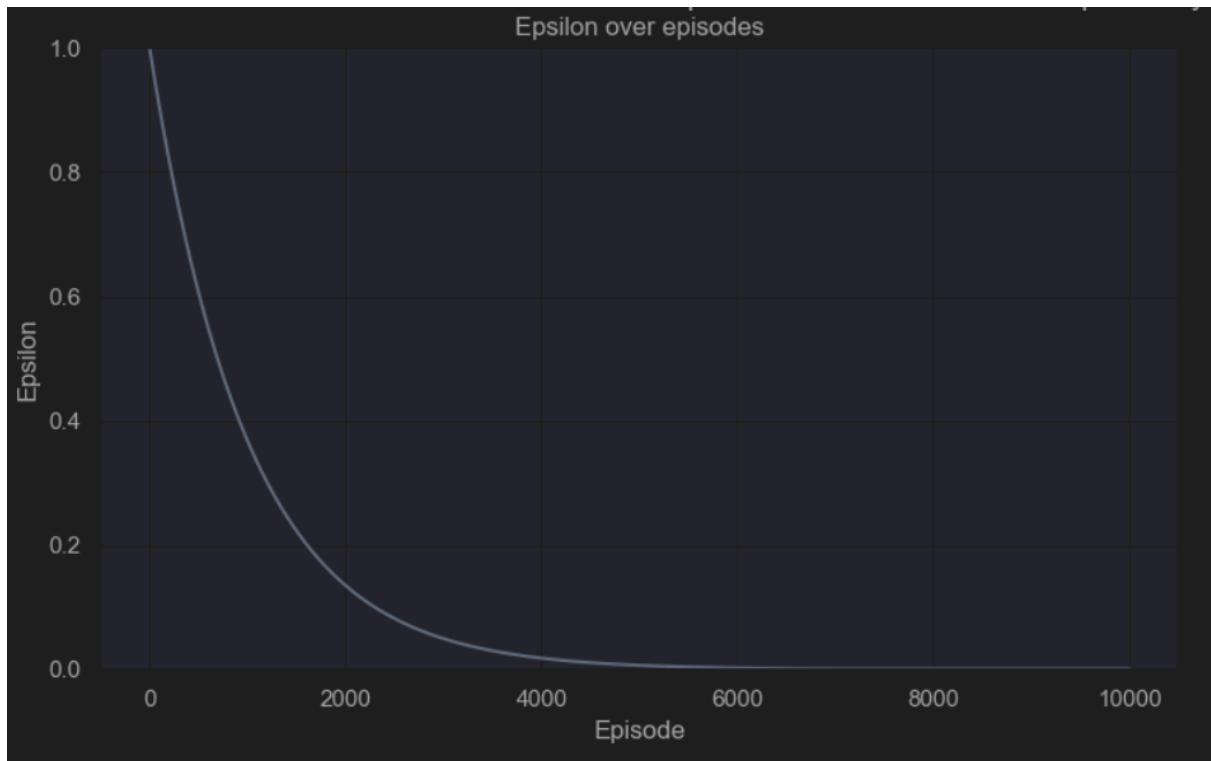
L'algorithme SARSA (State-Action-Reward-State-Action) est une méthode d'apprentissage par renforcement où un agent apprend à prendre des décisions en interagissant avec son environnement.

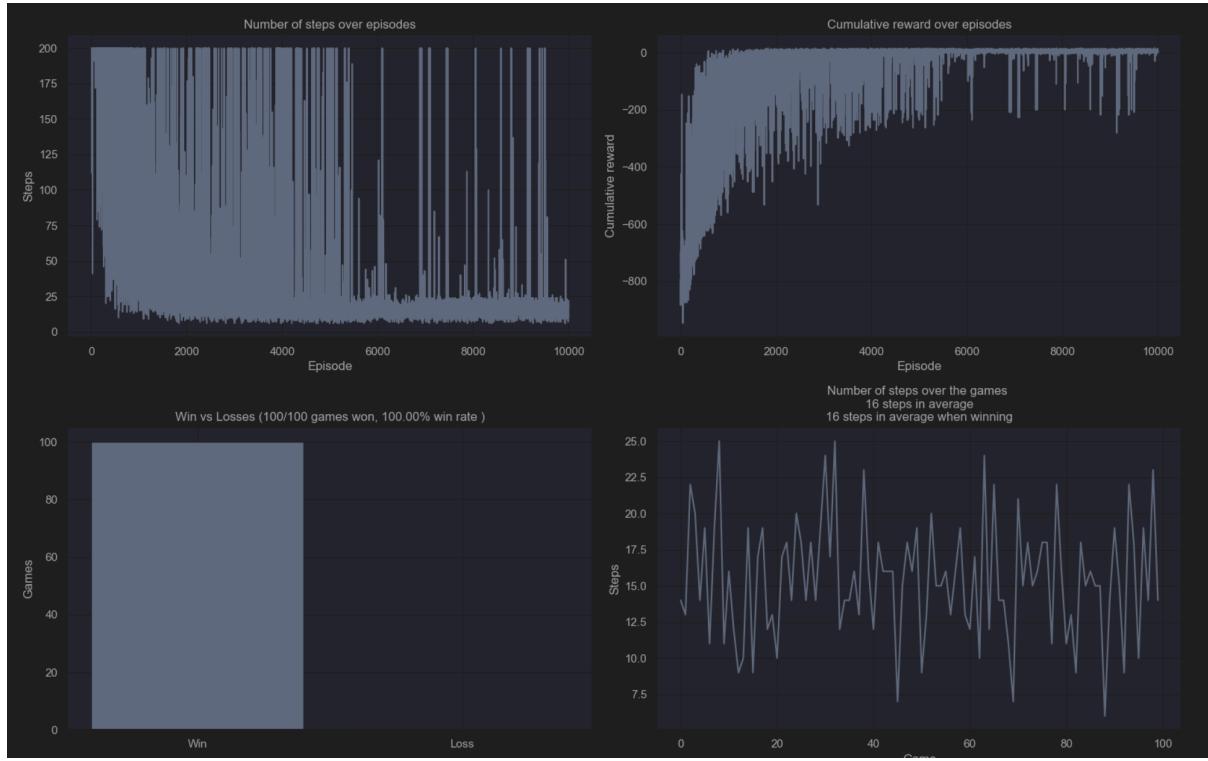
L'agent suit une politique et met à jour sa stratégie en utilisant une séquence d'états et d'actions. À chaque étape, il observe un état, choisit une action, reçoit une récompense, observe le nouvel état, puis choisit une nouvelle action.

La valeur de chaque paire état-action est mise à jour en fonction de l'action réellement choisie. SARSA est ainsi appelé car il utilise la séquence (État, Action, Récompense, État suivant, Action suivante).

L'algorithme SARSA est intéressant à utiliser car il a un fonctionnement qui se rapproche de ce que pourrait nous proposer le Q-learning, à savoir l'utilisation de son état et de son environnement pour améliorer les résultats obtenus pendant tout le traitement.

4.3.2. Résultats avec les paramètres optimaux





Avec les paramètres optimaux on observe que le modèle termine son apprentissage aux alentours de 4 000 épisodes.

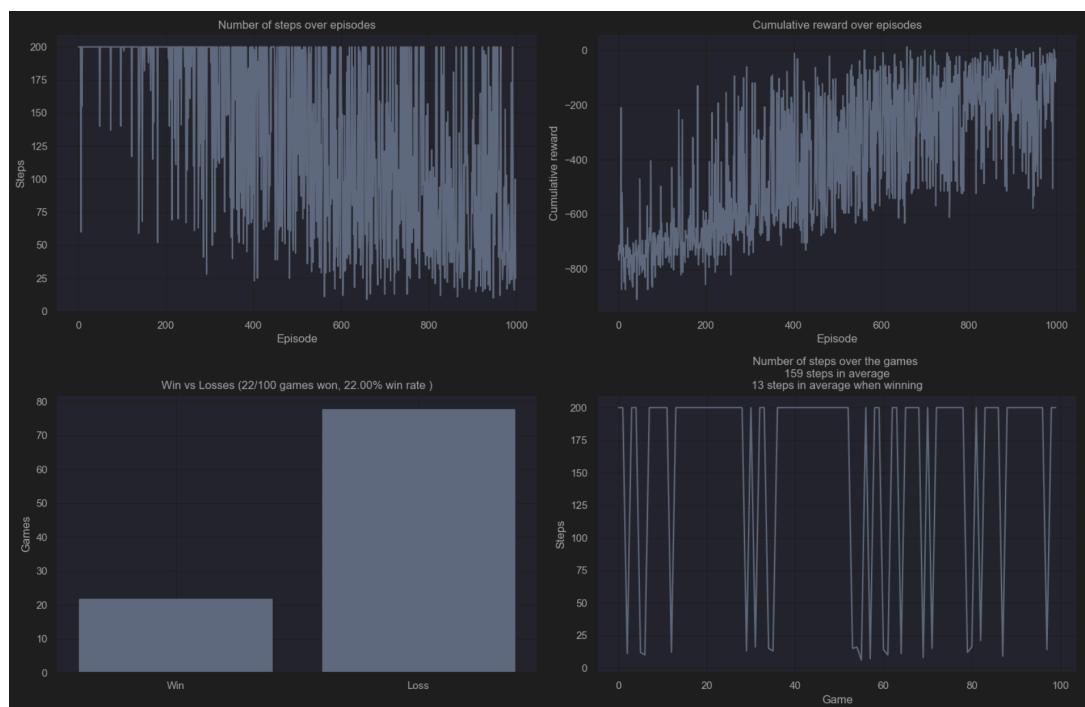
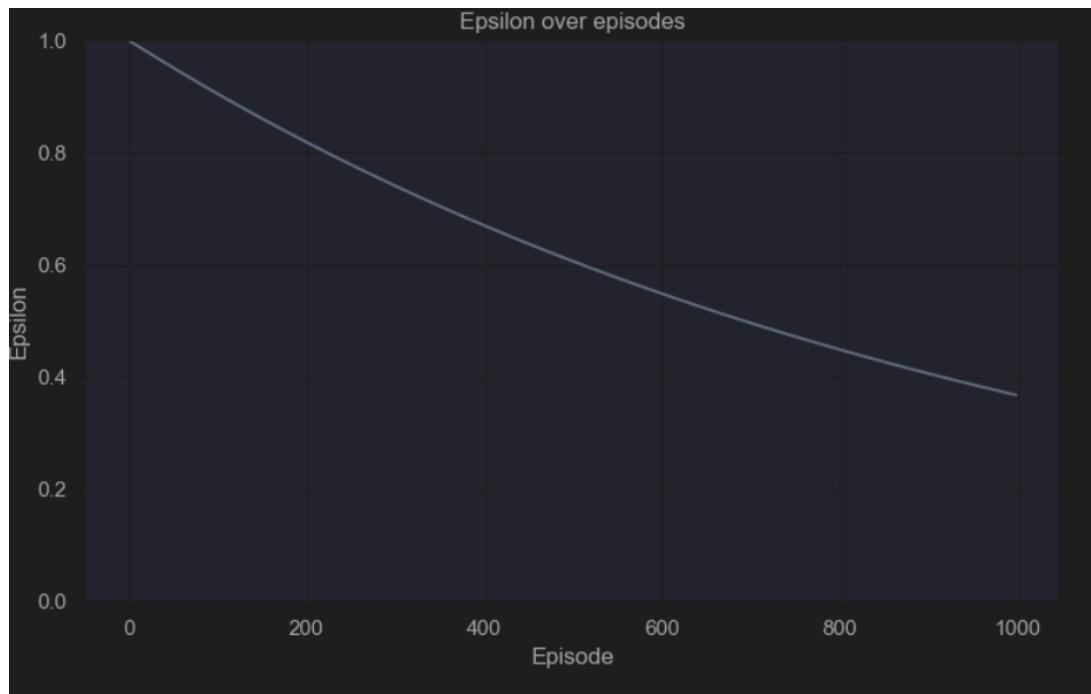
Pour ce qui est des résultats une fois l'entraînement terminé, on simule 100 parties et on observe les résultats. À la fin de ce traitement on obtient un taux de réussite de 100%. C'est-à-dire que l'agent à réussi à traiter les passagers correctement dans tous les essais.

Il obtient ce résultat en se déplaçant en moyenne 16 fois par épisode.

On observe aussi que la récompense tend à se rapprocher de 0 à partir de 6 000 épisodes même si on observe encore des variations lors de certains épisodes.

4.3.3. Nombre d'épisodes

4.3.3.1. Paramètre bas (1 000 épisodes)



Avec les paramètres Bas on observe que le modèle n'a pas eu le temps de terminer son apprentissage. Cela a un impact important sur les résultats suivants.

Pour ce qui est des résultats une fois l'entraînement terminé, on obtient un taux de réussite de 22%. Ce résultat est principalement dû au manque de temps d'apprentissage du modèle qui n'a pas encore réussi à s'adapter complètement à son environnement.

Il obtient ce résultat en se déplaçant en moyenne 159 fois lors d'une tentative lambda et 13 fois lorsqu'il arrive à remporter une partie.

On observe aussi que la récompense commence à se rapprocher de 0 mais reste encore bien trop élevée.

4.3.3.2. Paramètre élevé (50 000 épisodes)

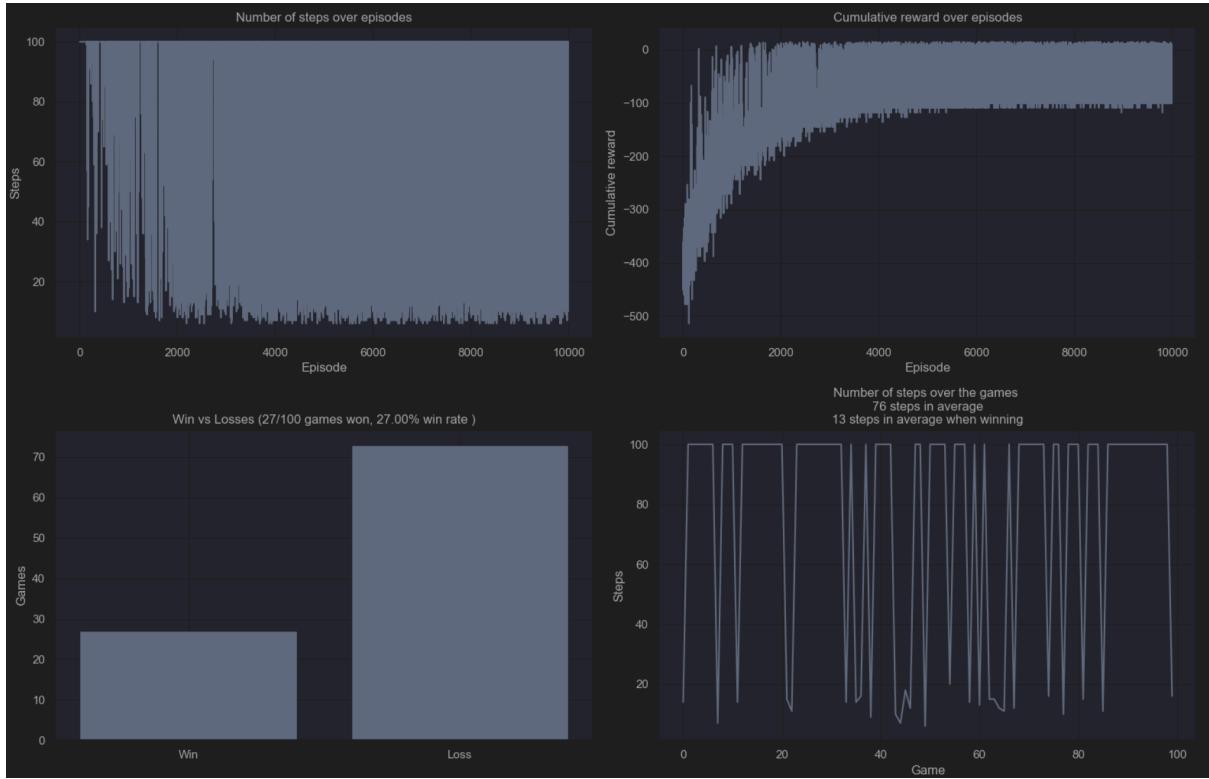


Avec les paramètres Haut on obtient un résultat qui n'est pas très différent du modèle optimal.

Le modèle obtient alors toujours 100% de réussite et le nombre de pas pour y arriver est légèrement plus faible que dans le modèle optimal à savoir 14 au lieu de 16.

On observe aussi que la récompense suit la dynamique du modèle optimal à savoir être proche de 0 au bout de 6 000 épisodes mais fluctue pendant la suite du traitement.

4.3.4. Gamma (paramètre bas (0,85))



La valeur de Gamma impacte fortement et rapidement les résultats du traitement. En utilisant un gamma de 0,85 on obtient les observations suivants :

Le nombre de pas a du mal à se stabiliser, on varie entre 0 et 200 pas tout le long du traitement.

Le modèle a aussi un taux de réussite bien plus faible à la fin de l'entraînement soit 27 % avec une moyenne de pas de 13 lors des réussite mais de 76 dans le cas échéant.

La courbe de récompense se stabilise à -150 aux alentours des 3 000 épisodes.

4.3.5. Epsilon Greedy (paramètre bas (0,1))



On observe dans ces graphiques que malgré l'utilisation d'un epsilon plus faible, on arrive quand même à des résultats satisfaisants pour le modèle.

Cela s'explique car SARSA met à jour les valeurs d'action-état en suivant la politique actuellement utilisée pour prendre des décisions.

Cela rend l'apprentissage plus stable même avec une exploration modérée.

L'agent adapte progressivement sa politique en fonction des récompenses immédiates et futures, ce qui permet une convergence même avec des valeurs de epsilon plus faibles.

4.3.6. Decay Epsilon Greedy (paramètre haut (0,1))



En réduisant trop rapidement la valeur de epsilon, le modèle n'exploré pas assez l'environnement ce qui peut avoir comme effet de piéger notre modèle dans une politique non-optimale.

La conséquence est que le modèle ne s'équilibre pas assez entre exploration et apprentissage et ne donne donc pas de résultats optimaux à la fin.

Dans notre cas, on voit que le modèle a perdu en taux de victoire et se retrouve avec quelques épisodes qui ne se terminent pas correctement.

4.3.7. Conclusion

Le modèle SARSA a été testé dans notre projet pour comparer ses performances à celles d'autres algorithmes tels que le Q-learning, Monte Carlo et Deep Q-learning.

Les résultats obtenus montrent que SARSA, en mettant à jour les valeurs état-action basées sur l'action réellement choisie, offre une approche différente des algorithmes vu par la suite.

Contrairement au Q-learning, SARSA se concentre sur la politique actuelle de l'agent, ce qui peut fournir des perspectives uniques dans des environnements dynamiques.

Comparé au Monte Carlo, SARSA permet des mises à jour plus fréquentes, et bien qu'il ne rivalise pas avec les capacités du **Deep Q-learning** pour des tâches complexes, il reste pertinent pour des environnements plus simples.

4.4. Q-Learning

4.4.1. Présentation

Le Q-learning est une méthode d'apprentissage par renforcement qui permet à un agent d'apprendre à prendre des décisions optimales dans un environnement en utilisant des récompenses pour guider son apprentissage.

Il fonctionne en estimant une valeur appelée Q-valeur pour chaque paire état-action, représentant la récompense attendue à long terme en partant de cet état et en choisissant cette action.

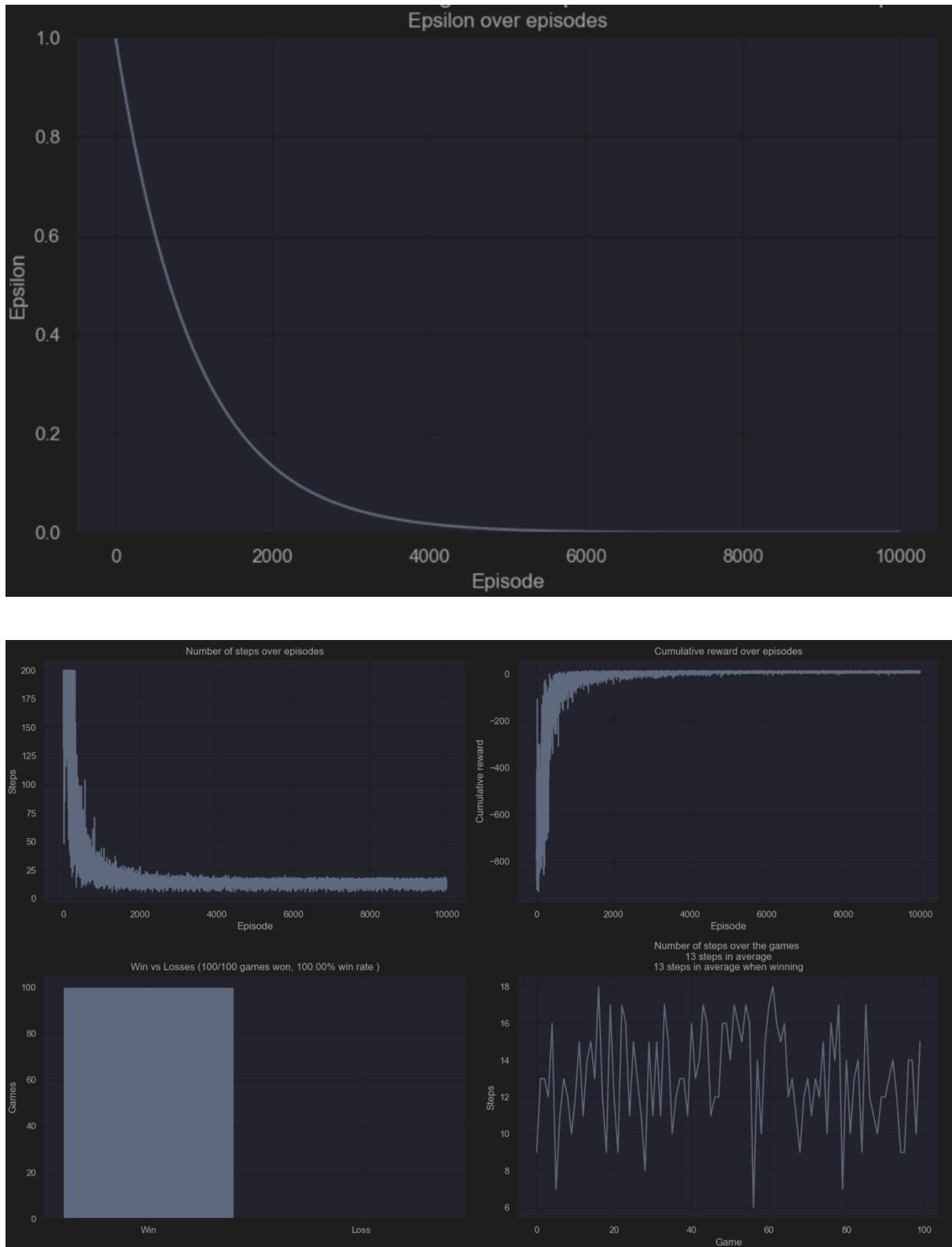
L'agent ajuste progressivement ses estimations de Q-valeurs en fonction des récompenses réelles obtenues, utilisant une règle d'apprentissage itérative qui maximise la valeur cumulée des récompenses au fil du temps.

4.4.2. Intérêt dans le projet

Le Q-Learning est l'algorithme qui se rapproche le plus de notre objectif sur le projet à savoir mettre en place du Deep-Q-Learning.

Il possède une vitesse d'apprentissage rapide et offre des résultats très intéressants pour la suite de nos recherches.

4.4.3. Résultats avec les paramètres optimaux



Avec les paramètres optimaux on observe que le modèle termine son apprentissage aux alentours de 1 000 épisodes.

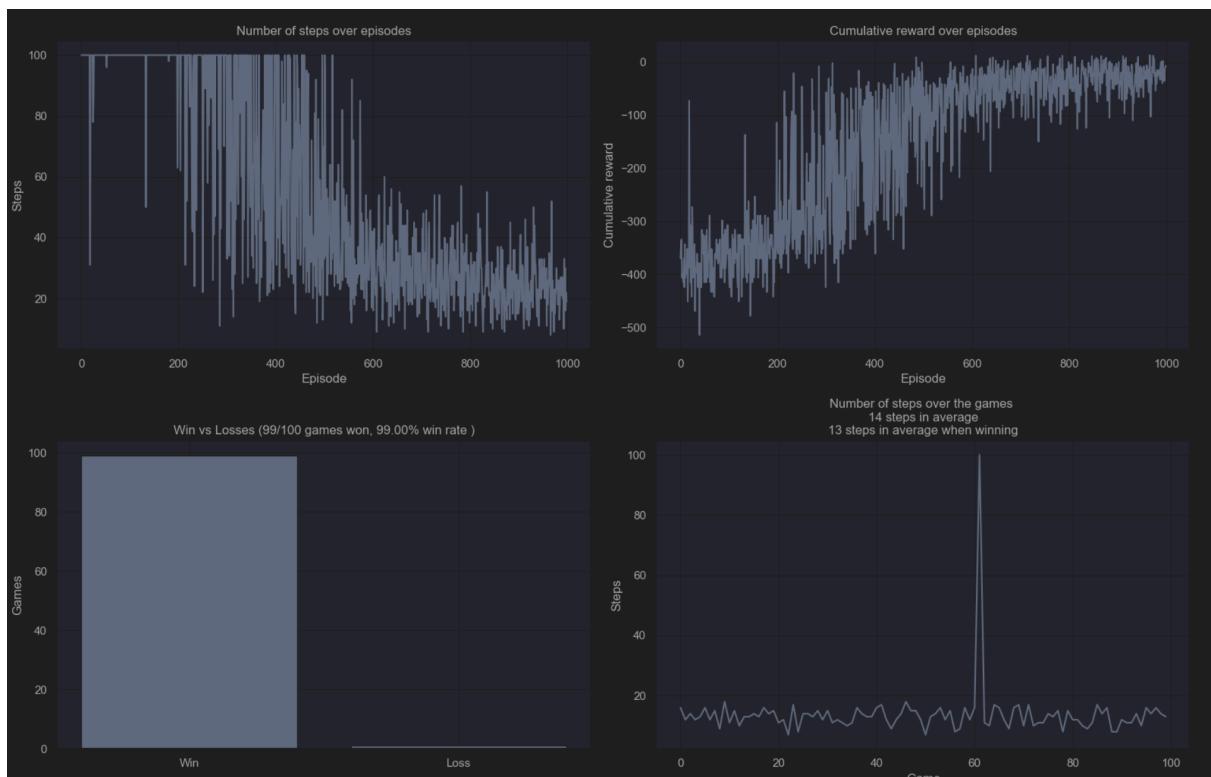
Pour ce qui est des résultats une fois l'entraînement terminé, on simule 100 parties et on observe les résultats. À la fin de ce traitement on obtient un taux de réussite de 100%. C'est-à-dire que l'agent à réussi à traiter les passagers correctement dans tous les essais.

Il obtient ce résultat en se déplaçant en moyenne 13 fois par épisode.

On observe aussi que la récompense tend à se rapprocher de 0 à partir de 1 500 épisodes et reste très stable sur le reste des épisodes.

Le nombre de pas nécessaires à valider un épisode est aussi atteint très rapidement. Au bout de 1 500 épisodes, il devient stable.

4.4.4. Nombre d'épisodes (paramètre bas (1 000 épisodes))



Avec les paramètres Bas on observe déjà une grande amélioration par rapport au SARSA.

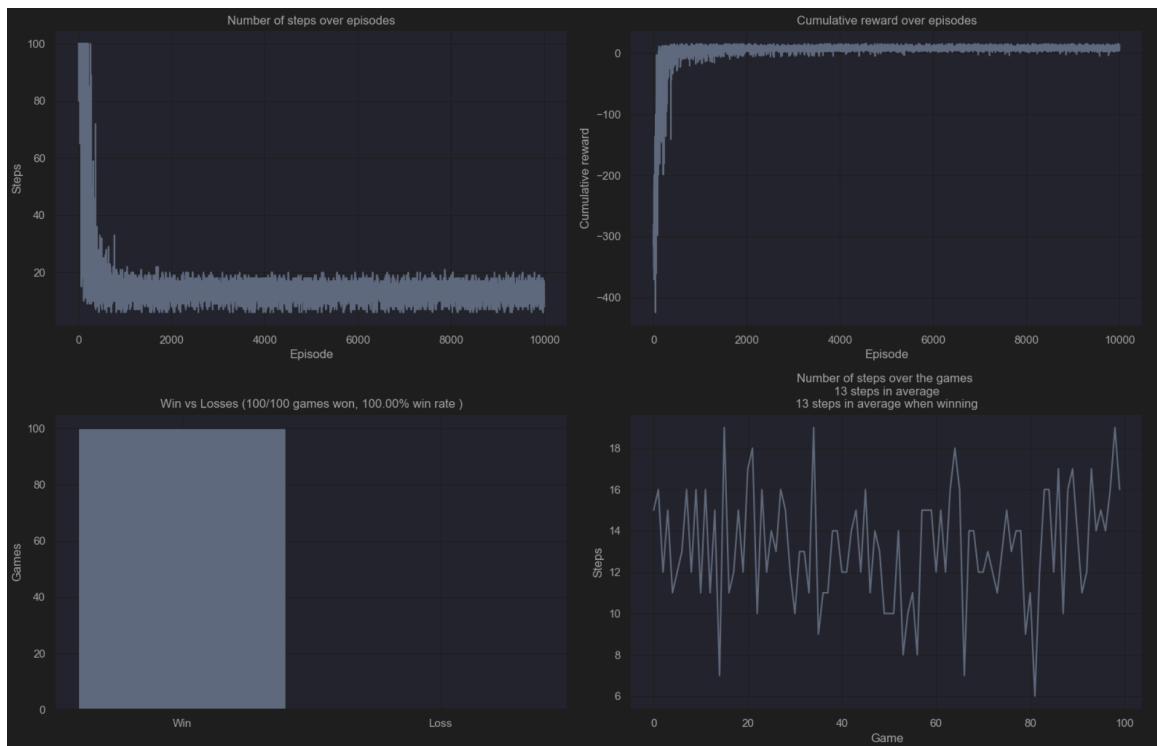
En termes de résultats, même si l'apprentissage n'est pas terminé on arrive déjà à un taux de réussite de 99%.

Le nombre de pas pour y arriver est aussi déjà stable à 13 pas en moyenne.

La récompense n'est pas encore stable mais est déjà proche de son objectif de 0.

4.4.5. Epsilon (paramètre bas (0,1))

Sachant que le Q-learning apprend plus rapidement que le SARSA, il est intéressant de tester l'algorithme en lui donnant un pourcentage faible pour vérifier l'impact sur le traitement.



Malgré le peu d'apprentissage dont dispose le modèle, on arrive quand même à des résultats très similaires aux paramètres optimaux.

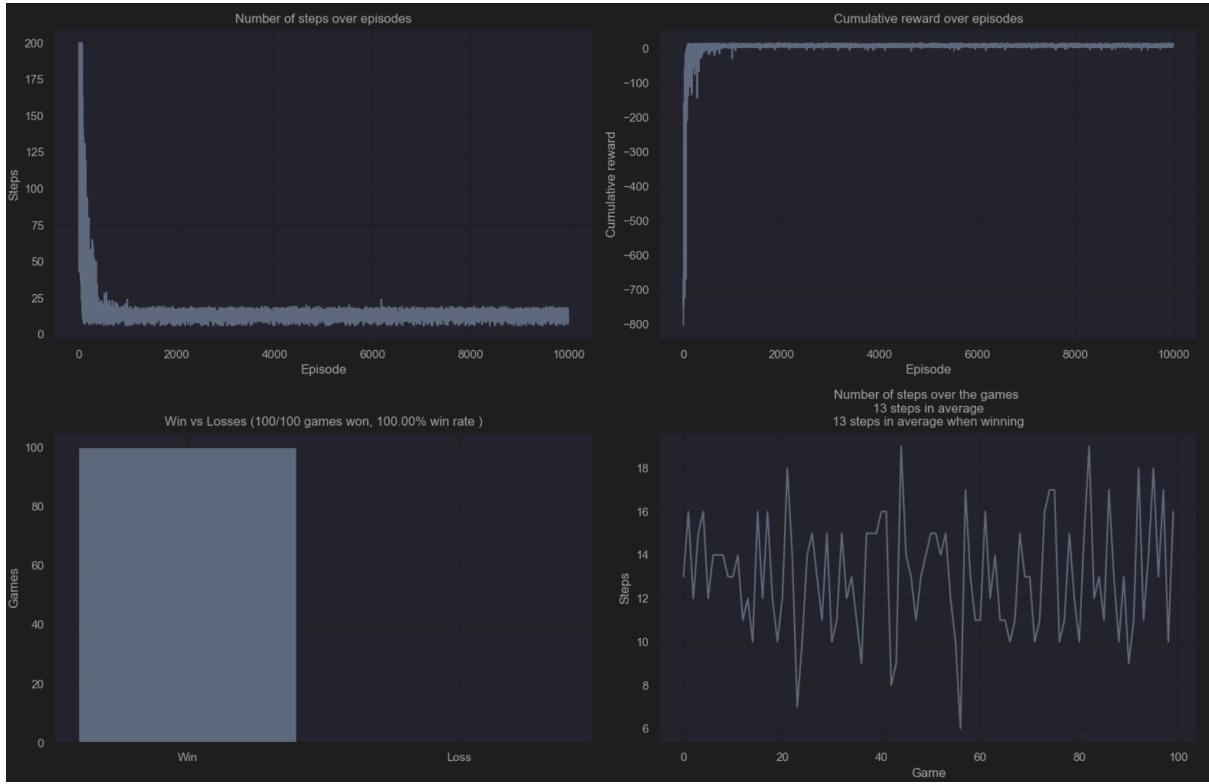
La récompense atteint cependant 0 de manière plus rapide, aux alentours des 700 épisodes.

En modifiant la valeur de Epsilon, à savoir le nombre d'épisodes d'exploration dans l'environnement, on se rend compte que les résultats ne sont pas trop impactés dans ce modèle.

Cela s'explique car l'environnement possède un nombre très limité d'actions, il converge rapidement vers une politique optimale et donc des résultats intéressants.

La structure simple et les récompenses claires de l'environnement facilitent cette convergence.

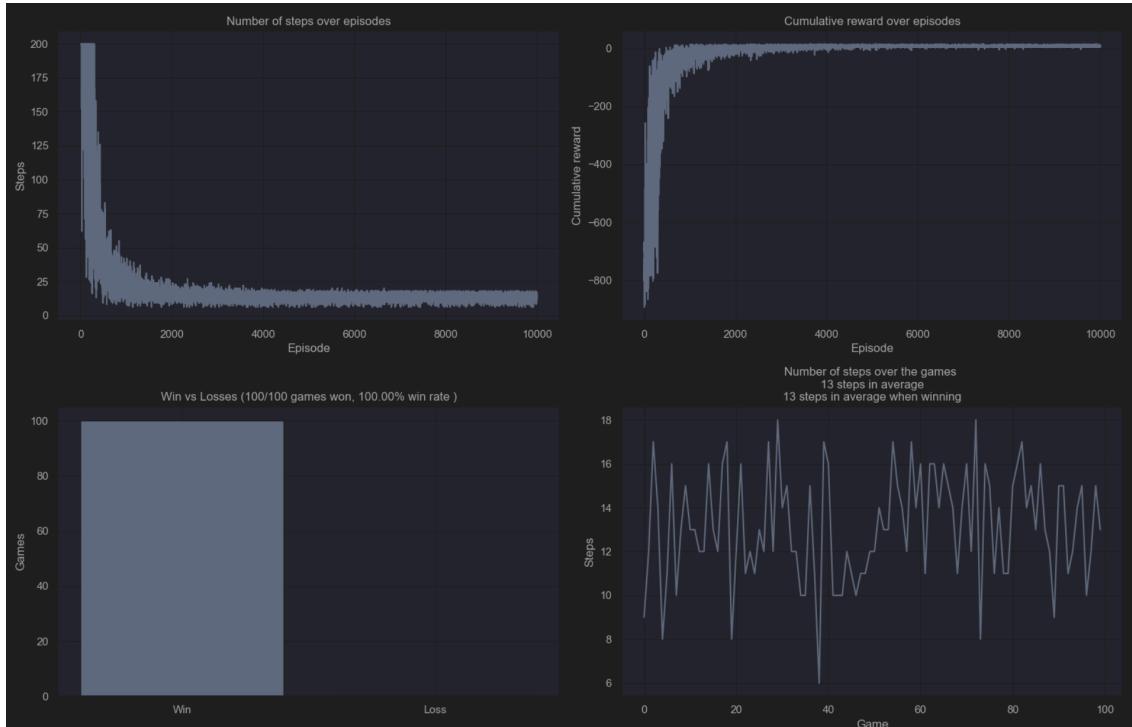
4.4.6. Decay Epsilon Greedy (paramètre haut (0,1))



Comme pour le modèle SARSA, en réduisant trop rapidement la valeur de epsilon, le modèle n'explore pas assez l'environnement ce qui peut avoir comme effet de piéger notre modèle dans une politique non-optimale.

Dans notre cas, comme l'environnement taxi driver n'est pas très complexe et que le modèle Q-Learning converge rapidement vers une politique optimale, les résultats n'ont pas été affectés par la vitesse de passage de l'exploration à l'apprentissage.

4.4.7. Gamma (paramètre bas (0,85))



Le paramètre de gamma à, sur ce modèle, très peu d'impact en comparaison du SARSA.

En utilisant un Gamma de 0,85 on obtient les mêmes résultats qu'avec les paramètres optimaux.

4.4.8. Conclusion

Au vu des résultats obtenus et des graphiques présentés, le Q-learning se révèle être une méthode robuste et efficace pour résoudre des problèmes d'apprentissage par renforcement.

Les courbes montrent clairement comment l'agent apprend progressivement à maximiser ses récompenses au fil du temps, grâce à l'ajustement des Q-valeurs en fonction des expériences acquises.

Cette approche offre non seulement une vitesse d'apprentissage rapide, mais elle démontre également sa capacité à converger vers des politiques optimales, garantissant ainsi des performances prometteuses dans des environnements complexes.

Ces résultats confirment que le Q-learning constitue une fondation solide pour explorer des variantes plus avancées comme le **Deep-Q-Learning**.

Ils montrent aussi que le Q-Learning semble être une approche plus solide et sereine que le SARSA, car l'apprentissage de l'agent converge plus rapidement vers la solution optimale, où la récompense obtenue est maximisée de façon claire.

4.5. Value Iteration

4.5.1. Présentation de l'algorithme

L'algorithme de Value Iteration est une méthode classique d'apprentissage par renforcement qui utilise une approche dynamique pour trouver la politique optimale en maximisant la fonction de valeur d'état. Cet algorithme est particulièrement efficace dans des environnements avec un espace d'état et d'action défini, où l'objectif est de calculer la politique optimale en itérant sur la fonction de valeur jusqu'à ce que la convergence soit atteinte.

Contrairement aux autres méthodes d'apprentissage par renforcement qui nécessitent des épisodes et des étapes d'exploration spécifiques, Value Iteration ne repose pas sur une politique de base pour explorer l'espace d'état. De plus, l'itération sur les valeurs d'état se fait sans la nécessité de simuler des épisodes ou des étapes, rendant l'algorithme indépendant des interactions séquentielles typiques nécessaires dans des environnements dynamiques.

Une fois la fonction de valeur suffisamment stabilisée (convergence atteinte), la politique optimale peut être déduite en choisissant pour chaque état l'action qui maximise la valeur d'état suivante selon la fonction de valeur mise à jour.

Cette méthode directe d'évaluation et d'amélioration de la politique à chaque itération permet à Value Iteration d'atteindre rapidement une solution optimale, particulièrement dans les environnements où la dynamique est entièrement connue et modélisée.

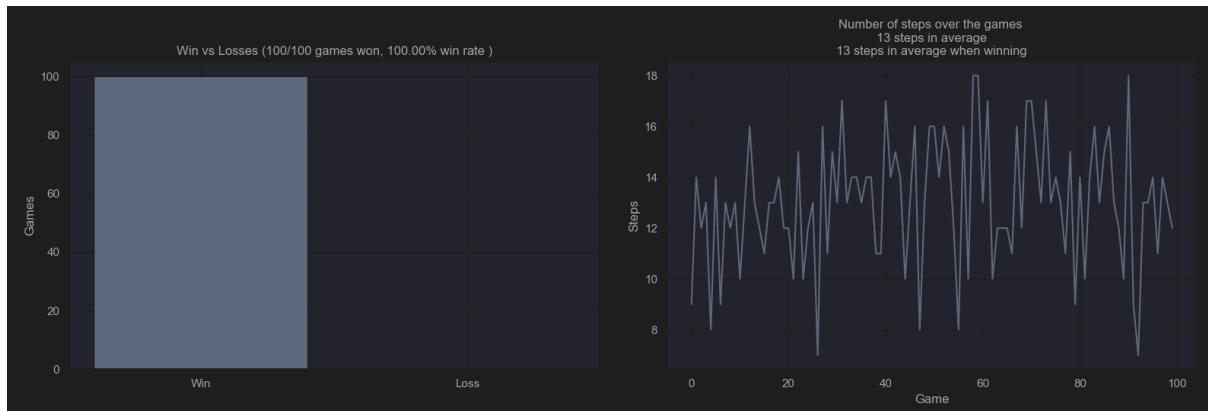
4.5.2. Expérimentation

Optimisation avec theta

Cette expérimentation visait à identifier le seuil de convergence optimal tout en maximisant les performances de l'algorithme. Nous avons commencé nos tests avec un theta initial de 0.01, augmentant progressivement cette valeur afin d'observer l'impact sur le temps d'entraînement et sur les performances.

Theta à 0.01

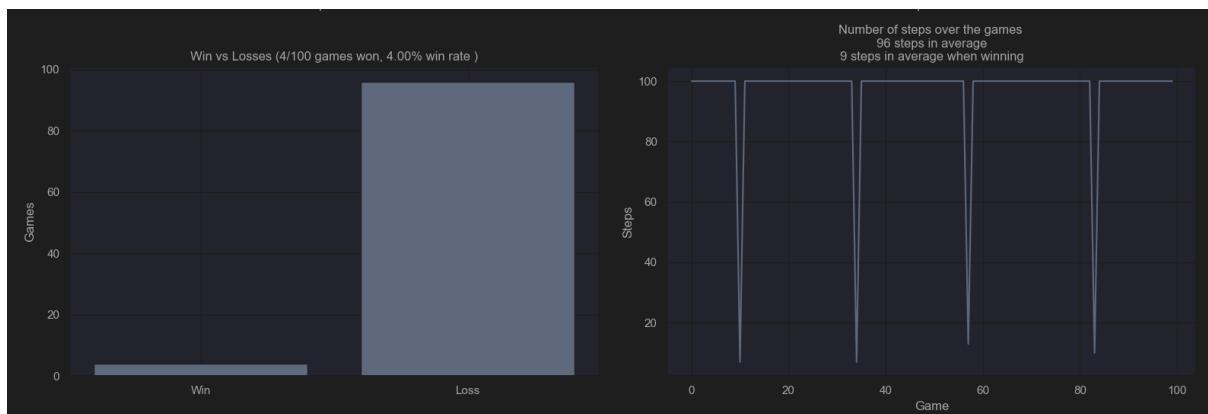
Execution time: 13.27 seconds



Progressivement, nous avons augmenté la valeur de theta. Tant que les performances restaient stables, nous avons poursuivi cette augmentation jusqu'à atteindre un seuil critique à 20. À ce niveau, bien que le temps d'entraînement fût presque nul, les performances ont dramatiquement chuté, indiquant une perte de précision dans la convergence.

Theta à 20:

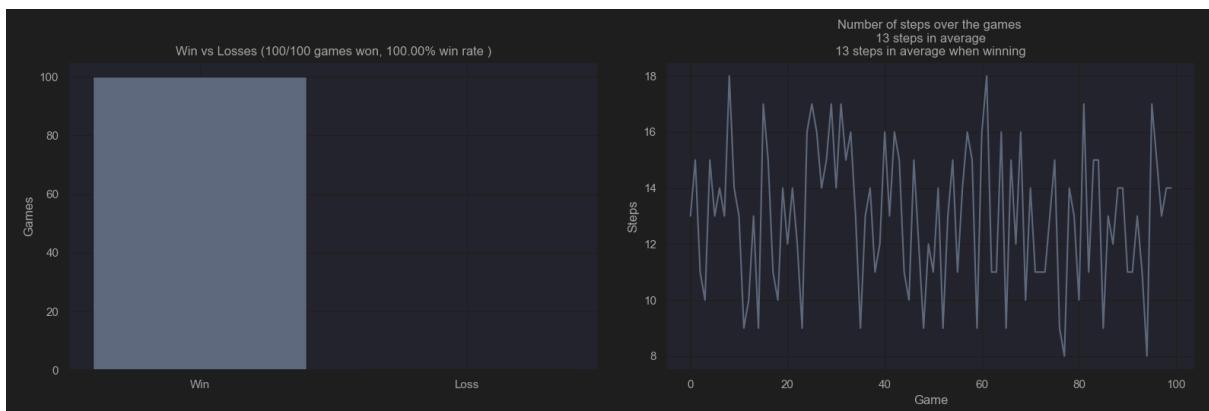
Execution time: 0.07 seconds



Finalement, en fixant theta à 15, nous avons trouvé un équilibre idéal entre l'efficacité de l'entraînement et la qualité des performances. Ce compromis nous a permis de réduire considérablement les temps de calcul tout en maintenant une précision acceptable des résultats.

Theta à 15 :

Execution time: 0.65 seconds



Le paramètre theta se révèle ainsi essentiel pour calibrer la précision de la convergence de Value Iteration. Son ajustement permet de moduler l'équilibre entre rapidité de convergence et exactitude de la politique dérivée, offrant ainsi une méthode efficace pour optimiser les performances sans compromettre la qualité globale de la solution..

Taille de la map

Nous avons souhaité explorer les limites de cet algorithme en termes de temps d'entraînement face à des problématiques plus complexes, notamment en augmentant la taille de la carte. Cependant, dans cet environnement de jeu spécifique, la dimension de la carte n'est pas configurable, ce qui nous a empêché d'effectuer ce test.

4.5.3. Conclusion

L'algorithme de Value Iteration s'est avéré être une méthode extrêmement efficace pour résoudre des problèmes dans des environnements d'apprentissage par renforcement avec des espaces d'état et d'action bien définis. Son approche, qui ne nécessite pas de simulations d'épisodes ou d'étapes d'exploration séquentielles, le rend particulièrement adapté pour des applications où la dynamique de l'environnement est entièrement connue et modélisable.

Au cours de nos expérimentations, nous avons ajusté le paramètre theta pour trouver un équilibre optimal entre la rapidité de convergence et la précision de la politique optimale. En commençant avec un theta à 0.01 et en augmentant progressivement sa valeur, nous avons exploré les limites de l'efficacité opérationnelle de l'algorithme. L'augmentation continue de theta a permis de réduire significativement le temps d'entraînement jusqu'à un seuil à 20, où

les performances ont commencé à se dégrader. Un équilibre a été trouvé avec un theta fixé à 15, permettant une convergence rapide sans compromettre de manière substantielle les résultats attendus, illustrant ainsi la capacité de l'algorithme à fournir des solutions rapides et précises dans des scénarios contrôlés.

Nous avons également tenté d'étendre nos tests à des scénarios plus complexes en augmentant la taille de la carte. Cependant, la nature non configurable de la dimension de la carte dans cet environnement spécifique a limité notre capacité à tester l'algorithme dans des conditions variées. Cette contrainte met en lumière les limitations pratiques de l'environnement de test et suggère un besoin d'explorer d'autres plateformes ou simulations où la taille de l'environnement peut être ajustée pour mieux comprendre l'évolutivité et la robustesse de Value Iteration.

4.6. Deep Q-Learning

4.7. Temps d'entraînement

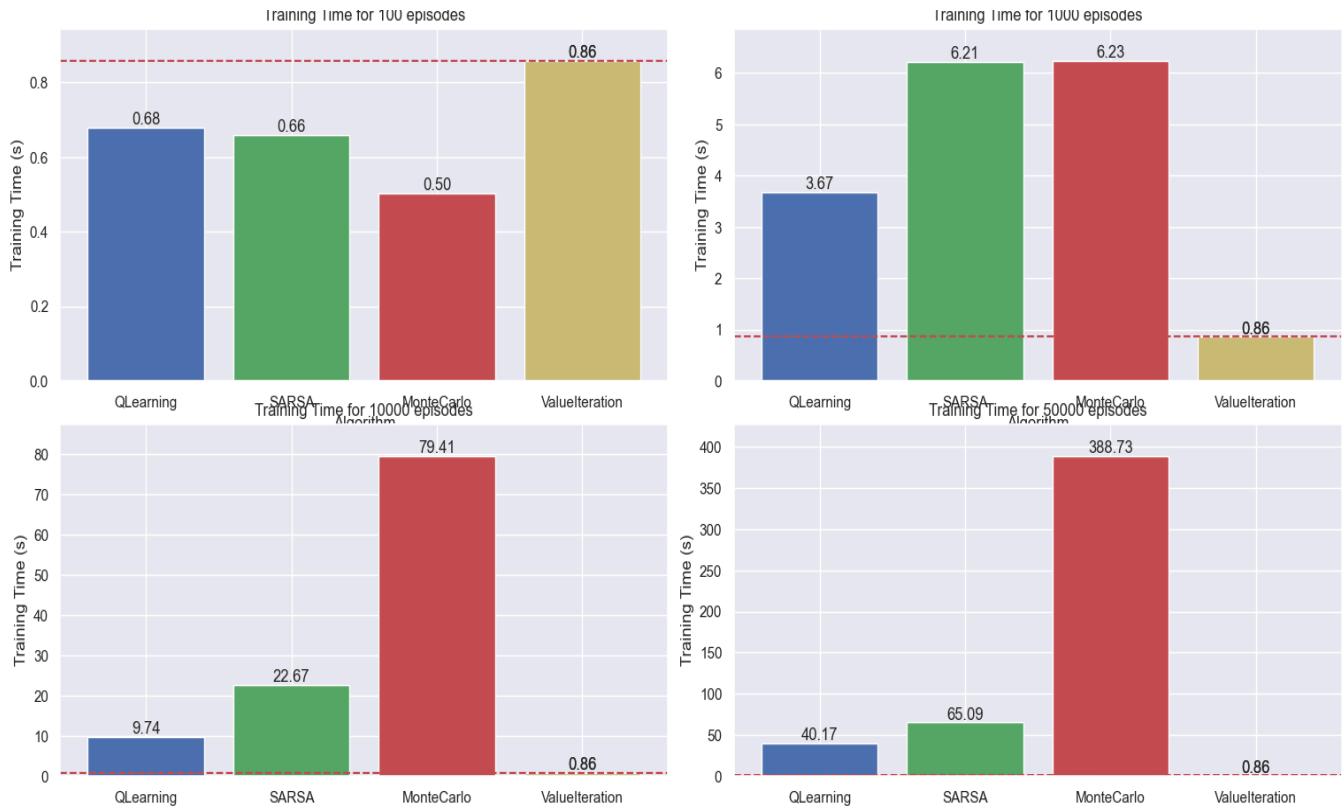
Évaluer la performance d'un algorithme ne se limite pas uniquement à mesurer la qualité de ses décisions ou la maximisation des récompenses obtenues. Il est également crucial de mesurer le temps d'entraînement nécessaire pour atteindre ces performances optimales. Cette évaluation temporelle permet de comprendre l'efficacité opérationnelle de l'algorithme, en particulier dans des contextes où les ressources computationnelles sont limitées ou coûteuses.

Pour évaluer de manière comparative le temps d'entraînement des algorithmes de reinforcement learning, nous avons mesuré les performances de **Value Iteration**, **Monte-Carlo**, **SARSA** et **Q-Learning**. Nous avons volontairement exclu l'approche **Brute Force**, en raison de son temps d'entraînement imprévisible et souvent prohibitif. De même, nous avons omis le **Deep Q-Learning** de notre comparaison, car bien qu'il soit plus performant dans des environnements complexes, il exige des ressources matérielles considérablement supérieures, notamment des GPU, rendant le temps d'entraînement difficilement comparable à celui des autres algorithmes évalués.

Nous avons donc mesuré le temps d'entraînement sur quatre algorithmes avec un nombre croissant d'épisodes à jouer, 100, 1 000, 10 000 puis 50 000. Monte-Carlo, SARSA et Q-Learning utilisent la politique du "Decayed Epsilon Greedy"; Value Iteration utilise la politique "Max". Les résultats sont décrits dans le graphique ci-dessous.

A noter que le nombre d'épisodes n'a pas d'impact sur l'algorithme Value Iteration, étant donné que son principe de fonctionnement ne se base pas sur l'expérience de l'environnement mais sur une approche itérative basée sur la connaissance de ce dernier.

Son temps d'entraînement servira donc de valeur de référence au fil de l'augmentation du nombre d'épisodes, puisqu'il n'évolue pas selon le nombre d'épisodes **mais selon la taille des états et des actions possibles dans un environnement**.



Temps d'entraînement d'un agent avec les algorithmes Q-Learning, SARSA, Monte-Carlo et Value Iteration sur 100, 1 000, 10 000 et 50 000 épisodes et 200 pas maximum

A l'issue de cette expérimentation, plusieurs observations sont à relever. La valeur référence établie par **Value Iteration** est d'environ 1 seconde.

Pour 100 épisodes, les autres algorithmes ont un temps d'entraînement inférieur (environ 0.5s). Cependant, 100 épisodes ne suffisent pas à converger comme nous avions pu l'observer lors de la variation du nombre d'épisodes pour ces algorithmes hormis le Value Iteration qui ne fonctionne pas par épisode et qui n'a besoin que d'une seule passe pour être performant.

A partir de 1 000 épisodes, les algorithmes SARSA et Q-Learning présentent des résultats intéressants et semblent converger. Monte-Carlo sera inefficace peu importe le nombre d'épisodes. Concernant le temps d'entraînement, ce dernier avoisine les 3,5 secondes, avec une tendance qui semble se dessiner. En effet, **SARSA** semble prendre légèrement plus de temps que le **Q-Learning**. Une observation qui se concrétise pour 10 000 et 50 000 épisodes, où l'écart se creuse (8,5 secondes puis 18 secondes d'écart).

Comparativement au temps d'entraînement du **Value Iteration**, le temps d'entraînement du Q-Learning et du SARSA sont largement au-dessus dès 1 000 épisodes. Cette observation est logique puisque le temps d'entraînement pour ces deux algorithmes augmente selon le

nombre d'épisodes, ce qui n'est pas le cas du Value Iteration. **Pour un environnement plus complexe (plus d'états ou plus d'actions), le temps d'entraînement du Value Iteration exploserait et une approche exploratoire serait sûrement plus performante.**

Enfin, le Monte-Carlo ne convergent pas sur notre environnement, il est logique que son temps d'entraînement surpassé ses concurrents. Avec cet algorithme, l'agent tend à ne pas prendre de risque et simplement faire des pas jusqu'au maximum autorisé par l'environnement pour limiter sa perte de récompense, plutôt que de chercher la voie qui le récompenserait en gagnant une partie.

Ainsi, l'algorithme du Monte-Carlo présente la valeur de temps d'entraînement maximal obtenable par variation du nombre d'épisodes. On remarque donc que le Q-Learning et le SARSA arrivent largement à optimiser les trajets de l'agent au fur et à mesure de l'entraînement, contrairement au Monte-Carlo: sur 50 000 épisodes, on gagne 3 minute et 30 secondes d'entraînement voire plus entre le Monte-Carlo et le Q-Learning et SARSA.

5. Conclusion

Les algorithmes d'apprentissage par renforcement présentent chacun des forces et des faiblesses distinctes.

L'environnement Taxi Driver est un exemple simplifié, avec un espace d'état relativement petit et des règles de transition claires. **Il ne représente pas la complexité des problèmes réels où ces algorithmes d'apprentissage par renforcement seraient appliqués.**

- **Brute Force** peut trouver des solutions optimales ici, mais serait impraticable pour des environnements plus complexes.
- **Monte Carlo** pourrait fonctionner efficacement, mais la lenteur de la convergence reste un problème.
- **SARSA** et **Q-Learning** montrent des performances adéquates dans cet environnement simpliste, mais leur véritable efficacité et robustesse doivent être validées dans des environnements plus réalistes et dynamiques. Dans notre cas, le Q-Learning semble plus performant que le SARSA dans la prise de décision lors de l'entraînement. Le Q-Learning semble converger plus sereinement et rapidement et maximise la récompense, là où le SARSA peut encore hésiter.
- **Deep Q-Learning** surpassé généralement les autres algorithmes dans des environnements complexes et de grande échelle, mais son avantage est moins évident dans des environnements simplistes comme Taxi Driver.
- **Value Iteration** est le plus efficace en termes de temps d'entraînement dans cet environnement spécifique.

Bien que ces algorithmes présentent des forces et des faiblesses distinctes, ils sont tous surpassés par **Deep Q-Learning** lorsqu'il s'agit de résoudre des problèmes d'apprentissage par renforcement à grande échelle et complexes.

Cependant, la performance observée dans l'environnement simplifié Taxi Driver n'est pas nécessairement indicative de leur efficacité dans des situations réelles plus complexes.

Il est donc crucial de tester ces algorithmes dans des environnements plus représentatifs de problèmes concrets pour évaluer pleinement leur performance et leur applicabilité.