



Taxi Driver

T-AIA-902-LYO_7

Tableau informatif de projet

Sujet	Le but de ce projet et d'appréhender le fonctionnement et les différentes possibilité de mise en place d'un modèle d'apprentissage par renforcement, ainsi que l'évaluation de ses performances.
Membres du groupe	Halil BAGDADI, Guillaume CORNILLE, Achille GOUTTARD, Clément GERINIERE, Julien FAURIE

Reinforcement Learning : définition

Le *Reinforcement Learning* (ou apprentissage par renforcement), est un système d'intelligence artificielle qui a pour but de d'apprendre à prendre les décisions optimales dans un environnement donné. Le principe est fondé sur un système de récompense, où l'objectif de l'algorithme est d'obtenir le meilleur score final.

Le score est déterminé à l'aide de la récompense finale, du nombre d'étape pour y parvenir et du nombre d'erreur rencontrées pour y parvenir (dans l'idéal aucune).

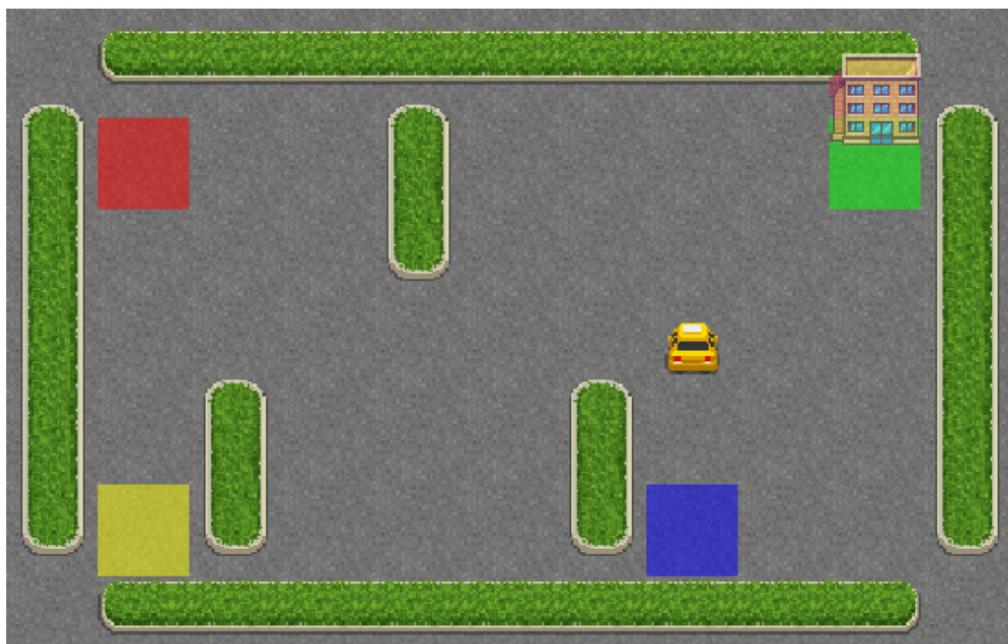
Par exemple, chaque action peut valoir -1, chaque action illégale vaudra -10, et chaque action définie comme étant l'objectif vaudra +10. Avec ces valeurs, l'algorithme cherchera à tout prix à éviter les actions illégales, et à arriver à l'objectif en un minimum d'action.

Gymnasium : l'environnement

Gymnasium est une librairie Python permettant de définir et de gérer tous les aspects de différents environnements simple appropriés aux tests des différents type d'algorithmes. La librairie propose donc plusieurs sorte d'environnements propice à l'apprentissage par la renforcement, comme la *mountain car*, le *pendulum*, le *lunar lander* ou, dans notre cas, le ***taxis-driver***.

L'environnement Gymnasium dédié au module “taxi-driver” comporte **6 actions possibles** et **500 état différents**, avec la position du client, sa destination et la position du taxi ainsi que son état (avec ou sans client).

Action: 1
Reward: -1
Episode reward: -8
Episodes: 11 / 50



Gymnasium propose un système de récompense parfaitement adapté à l'apprentissage par renforcement, qui suit les règles suivantes : **-1 point par mouvement, +20 point pour déposer un client à sa destination et -10 point par action illégale** (ex: tenter d'avancer sur un mur, récupérer ou poser un client lorsque cela n'est pas possible)

Algorithmes mis en place

Algorithme	Définition
Vanilla Q learning	Apprentissage par population d'une Q-table dans laquelle puiser les données en fonction de l'état de l'environnement à un instant t .
Deep Q learning	Mise en place de plusieurs couches de neurones dans lesquelles envoyer l'environnement à un état donné afin d'entrainer le modèle en explorant un maximum de possibilité dans l'environnement donné.
Sarsa	Acronyme de State-Action-Reward-State-Action qui est un algorithme " <i>on-policy</i> ", il utilise la politique en train d'être apprise pour mettre à jour les valeurs internes apprises.
Monte Carlo	Algorithme s'appuyant sur un échantillonnage aléatoire répété transformant ainsi des calculs déterministes complexes en problèmes d'échantillonnage aléatoires beaucoup plus simple à résoudre.

Vanilla Q learning

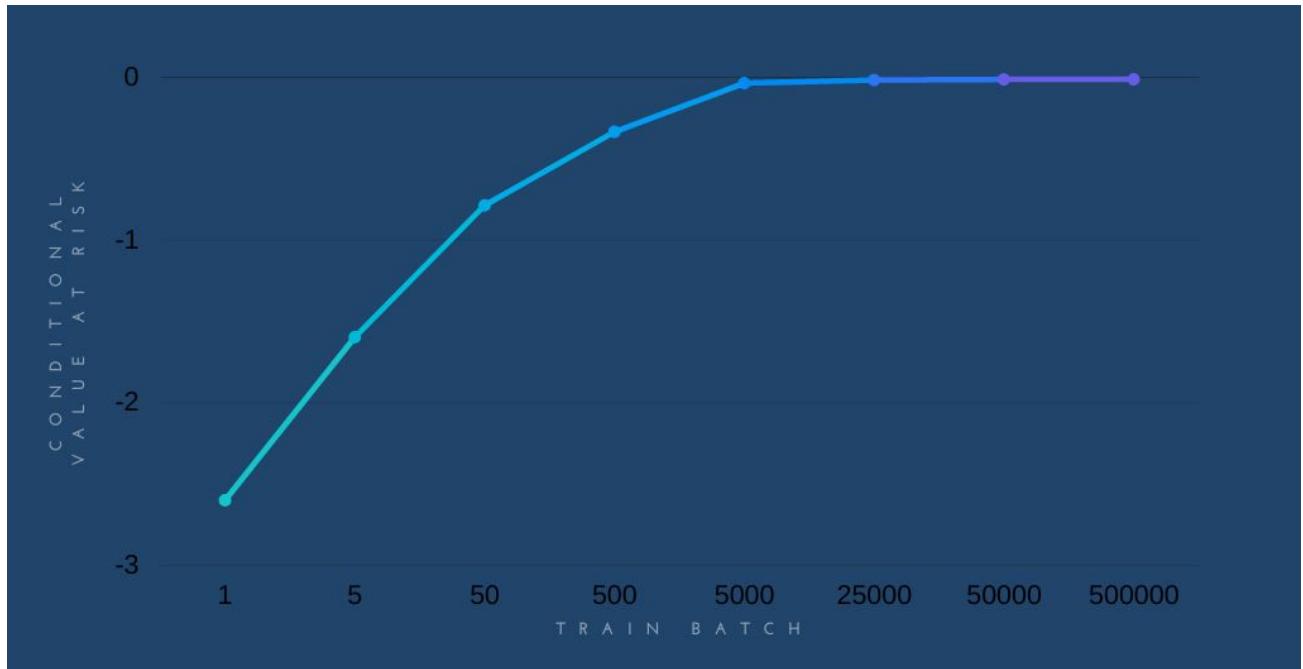
Le Q learning (ou vanilla Q learning) est un algorithme ayant pour principe de remplir un tableau d'action relatif à l'état de l'environnement à mesure que l'on entraîne le modèle (appelé **Q-table**). Les lignes de ce tableau sont constituées de **Q-value** pour chaque paire d'état - action.

Ce tableau sera alors utilisé pour prendre les décisions lors de l'utilisation. Plus le modèle sera entraîné, plus l'algorithme saura prendre la meilleure décision à un état donné.

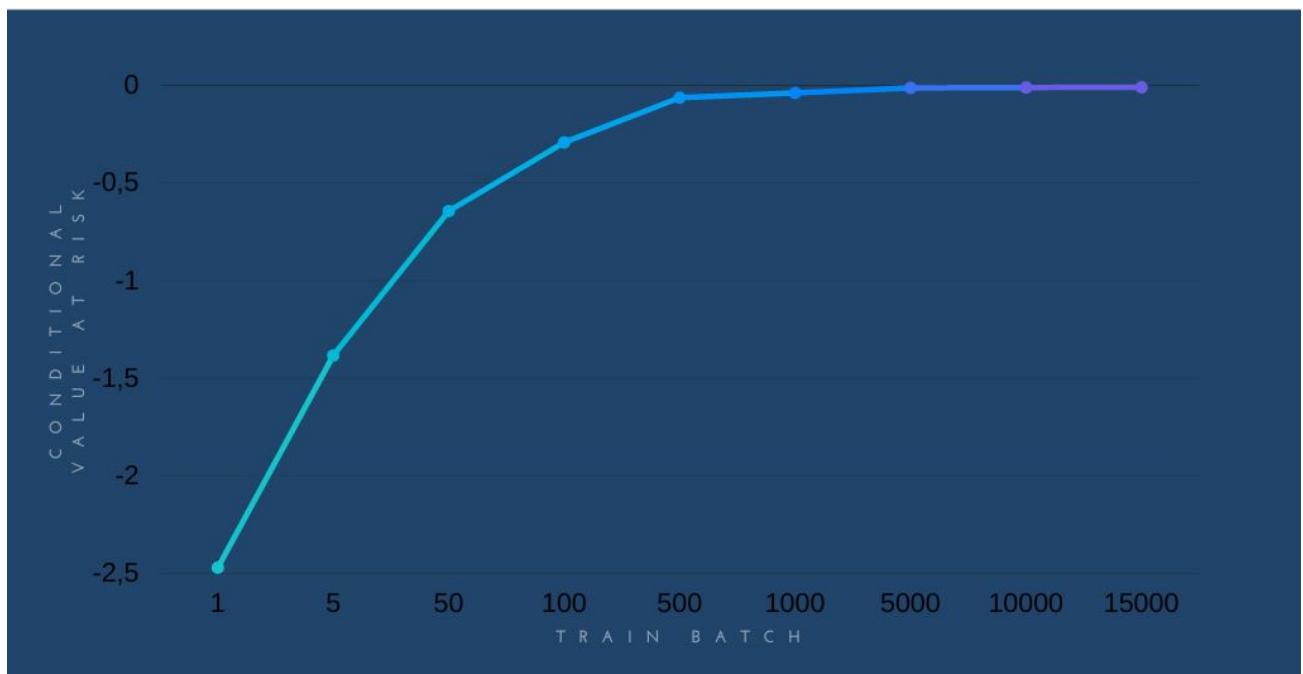
Q-table		Actions			
States		Left	Right	Up	Down
1		Q(1, Left)	Q(1, Right)	Q(1, Up)	Q(1, Down)
2		Q(2, Left)	Q(2, Right)	Q(2, Up)	Q(2, Down)
3		Q(3, Left)	Q(3, Right)	Q(3, Up)	Q(3, Down)
4		Q(4, Left)	Q(4, Right)	Q(4, Up)	Q(4, Down)
5		Q(5, Left)	Q(5, Right)	Q(5, Up)	Q(5, Down)
6		Q(6, Left)	Q(6, Right)	Q(6, Up)	Q(6, Down)
7		Q(7, Left)	Q(7, Right)	Q(7, Up)	Q(7, Down)
8		Q(8, Left)	Q(8, Right)	Q(8, Up)	Q(8, Down)
9		Q(9, Left)	Q(9, Right)	Q(9, Up)	Q(9, Down)
10		Q(10, Left)	Q(10, Right)	Q(10, Up)	Q(10, Down)
11		Q(11, Left)	Q(11, Right)	Q(11, Up)	Q(11, Down)
12		Q(12, Left)	Q(12, Right)	Q(12, Up)	Q(12, Down)
13		Q(13, Left)	Q(13, Right)	Q(13, Up)	Q(13, Down)
14		Q(14, Left)	Q(14, Right)	Q(14, Up)	Q(14, Down)
15		Q(15, Left)	Q(15, Right)	Q(15, Up)	Q(15, Down)
16		Q(16, Left)	Q(16, Right)	Q(16, Up)	Q(16, Down)

Notre algorithme a été mis en place à l'aide de la librairie **Numpy**. La librairie nous a en partie servie pour ses méthodes de gestion des **numpy array**, utiles à la création et à la gestion des **q_table**. Les méthodes **argmax & max** permettant par exemple d'extraire du numpy array la valeur de l'action la plus appropriée pour l'environnement donné.

Représentation tableau vanilla Q learning - Ceci est une représentation d'un tableau retourné par l'algorithme de vanilla Q learning une fois la phase de train terminée.



Metrics Q learning metric graph - Variables : $\alpha=0.1$; $\gamma=0.6$; $\epsilon=0.1$



Metrics Q learning metric graph [optimized] - Variables : $\alpha=0.5$; $\gamma=0.5$; $\epsilon=0.001$

Les résultats obtenus avec notre algorithme de vanilla Q learning sont très convaincant, en offrant un modèle très performant en un minimum d'itération.

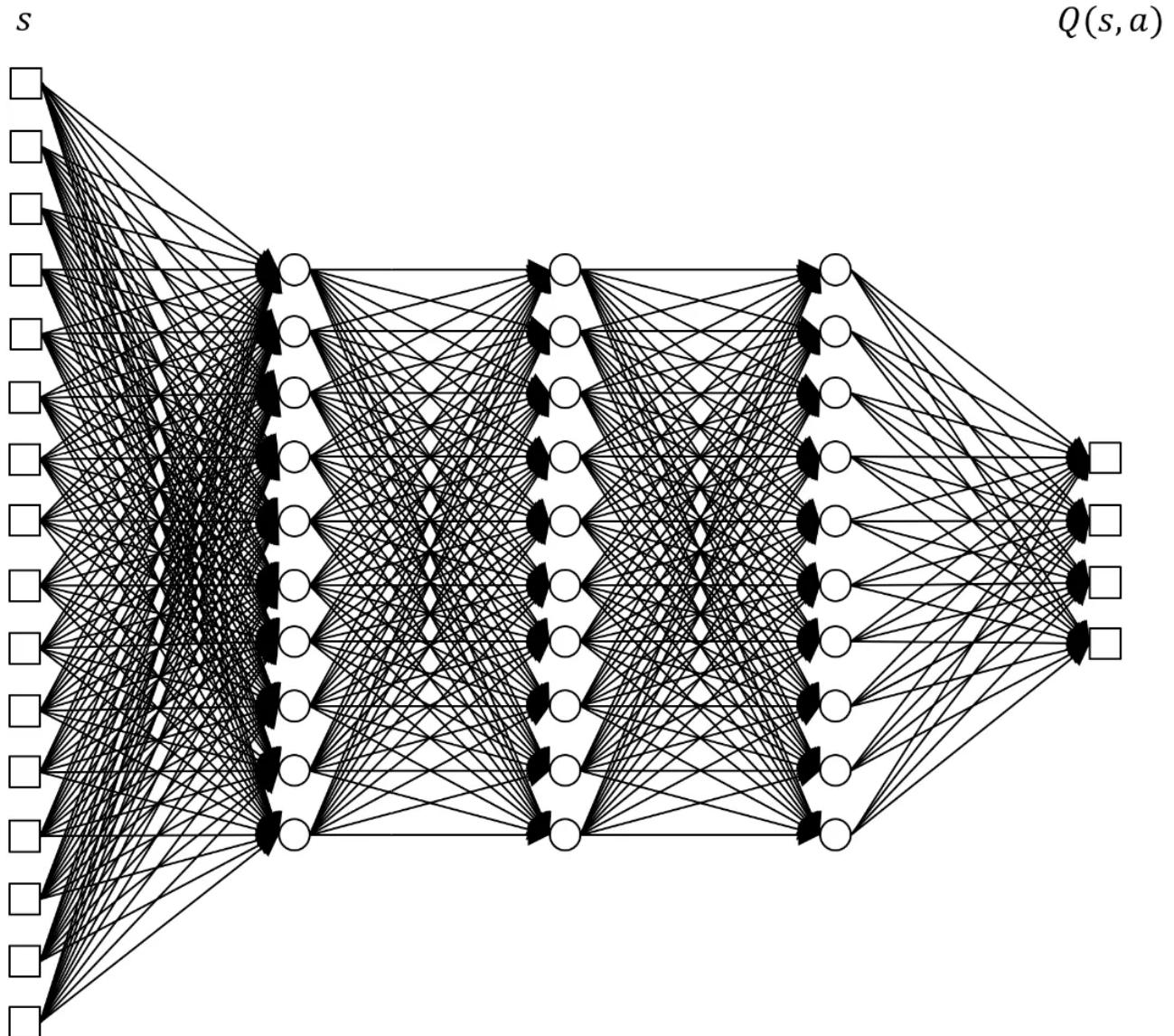
Le graphique précédent indique qu'en moyenne, l'algorithme permettra de résoudre un épisode de test de la manière la plus optimale avec **un taux d'erreur inférieur à 1%**, et ce en seulement **10 à 15 actions par épisodes**.

L'entraînement nécessaire pour obtenir ces résultats comprends seulement **2000 épisodes d'entraînements**, ce qui représente une **durée d'entraînement de généralement moins de deux secondes**, même sur une machine peu performante et sans gestion de multi-coeur du processeur.

Deep Q learning

Le deep Q learning quant à lui, fonctionne sur le principe de prendre en entrée l'état de l'environnement et de l'envoyer dans **plusieurs couches de réseau de neurones**. Le résultat sera alors sous forme de **Q-value par action**.

Ce type d'algorithme est plus pertinent dans un environnement d'une taille conséquente proposant une multitude d'actions, à l'instar du vanilla Q learning qui atteindra rapidement ses limites dans un tel environnement.



Représentation deep Q learning - Ceci est une représentation de l'utilisation d'un algorithme de deep Q learning, avec les différentes couches de neurones utilisées pour envoyer l'environnement.

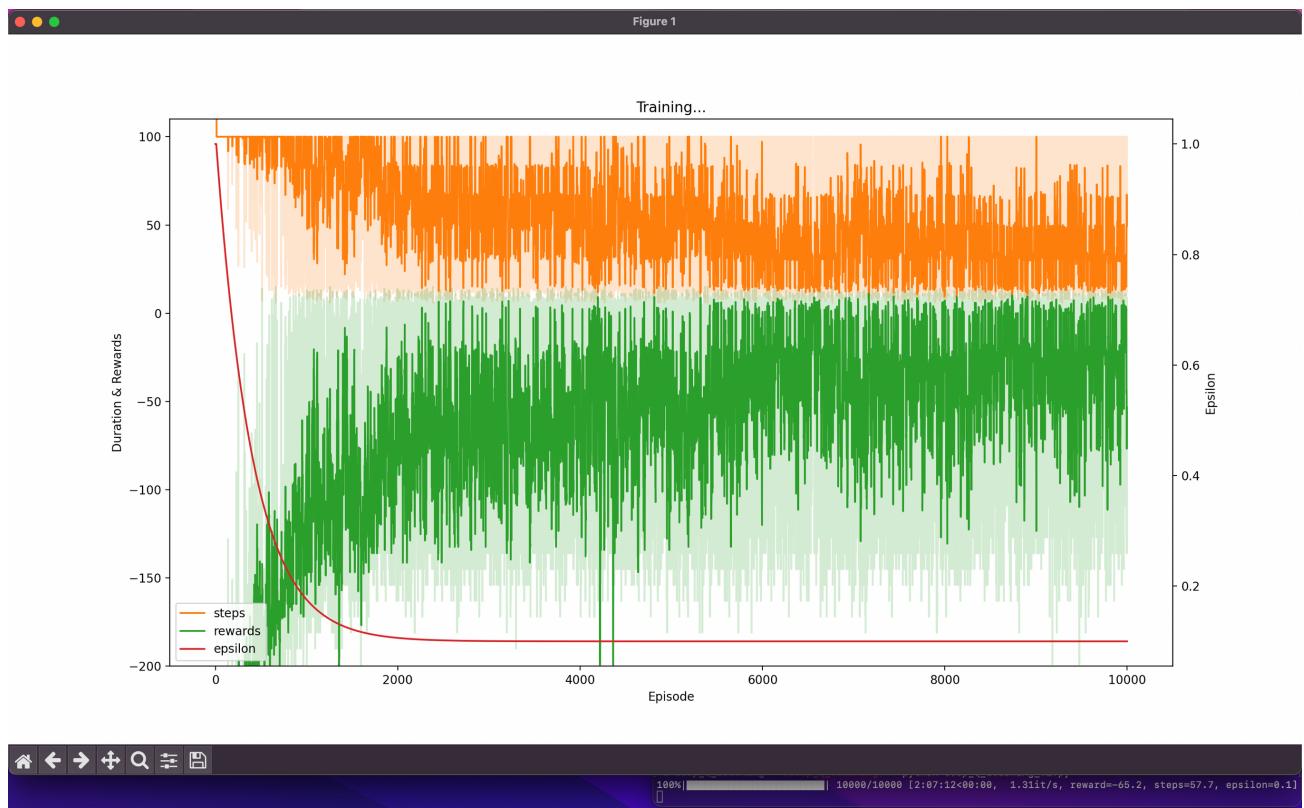
L'algorithme de deep Q learning quant à lui a été mis en place à l'aide des librairies **PyTorch** et **Numpy**.

Contrairement à l'algorithme de vanilla Q learning, il n'y a pas de besoin d'utiliser de numpy array pour la prédiction à partir du modèle entraîné. Cependant Numpy est une librairie très performante pour tout ce qui touche aux opérations mathématiques.

Dans notre cas, la librairie servira à notre algorithme pour la **génération de valeurs aléatoire** utiles à la partie exploration ou les calculs tels que **l'exponentiel et la somme cumulative** à partir de tableaux.

PyTorch a notamment permis de gérer les différentes **couches de neurones**, **l'optimizer** (**Adam** dans notre cas) ainsi que les fonction d'**activation non-linéaires** (**ReLU** – permettant à l'algorithme d'ajuster le poids de chaque paire *état-action*, plutôt que d'ajuster les valeurs associées à ces dernières, augmentant grandement la vitesse d'exécution).

Lors de notre phase de test, un tableau matplotlib nous affiche toutes les informations nécessaires à la visualisation des performances du modèle.

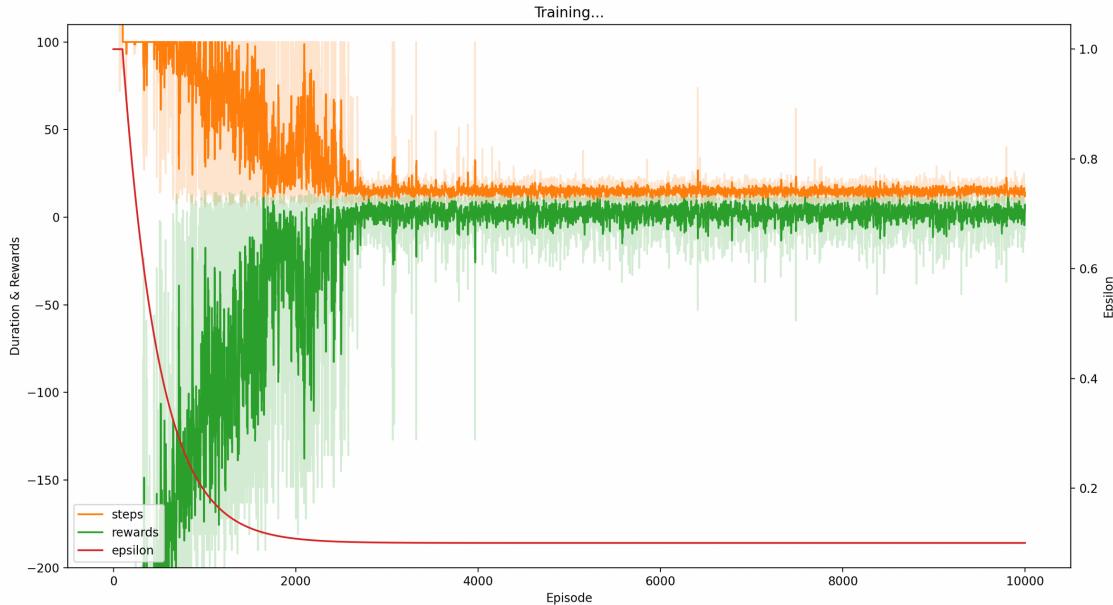


Dans le tableau précédent, nous avons entraîné notre algorithme de deep Q learning avec un **batch size de 128, 10000 épisodes** et un taux d'**apprentissage de 0.001**.

Les couches de neurones ont été construites pour obtenir une **entrée sur une couche de 4 neurones** donnant sur **2 couches de 50 neurones** avant de finalement retourner le résultat sous forme de **fichier '.pt'** (fichier de sauvegarde type de modèle PyTorch).

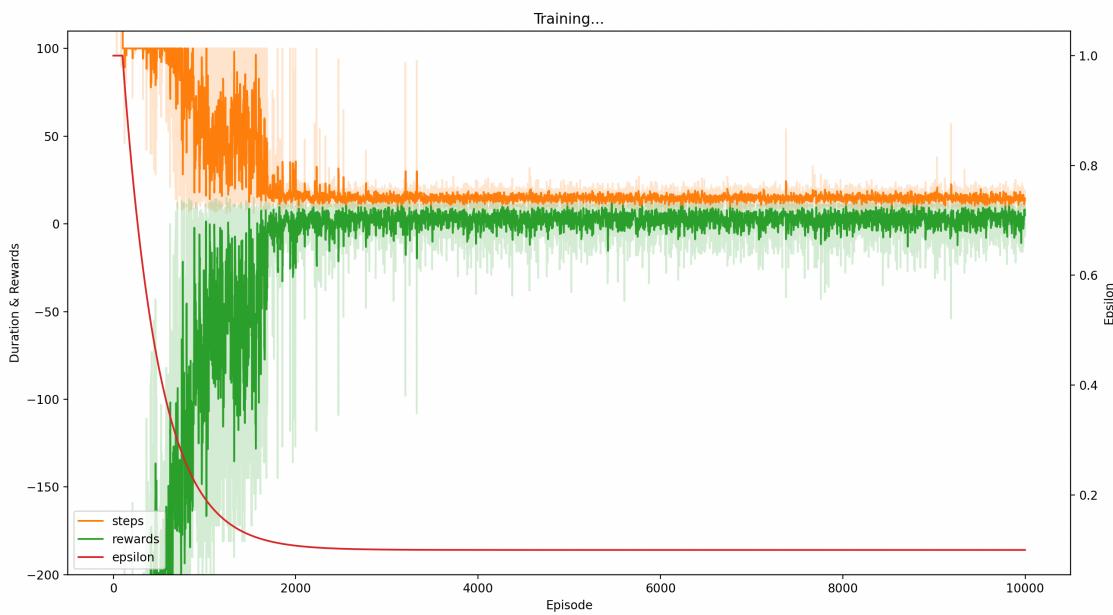
En modifiant la structure du réseau de neurone passant d'une entrée sur une couche de 4 à une **entrée sur une couche de 6 neurones**, puis en ajoutant une **troisième couche de 50 neurones**, nous pouvons voir une **nette amélioration des performances** générales, ainsi que du **temps nécessaire** pour les atteindre.

Figure 1



En se basant sur la même structure du réseau de neurone et en y augmentant le taux d'apprentissage de la phase d'entraînement, nous pouvons largement optimiser le temps nécessaire à l'algorithme pour obtenir des résultats corrects et régulier.

Figure 1



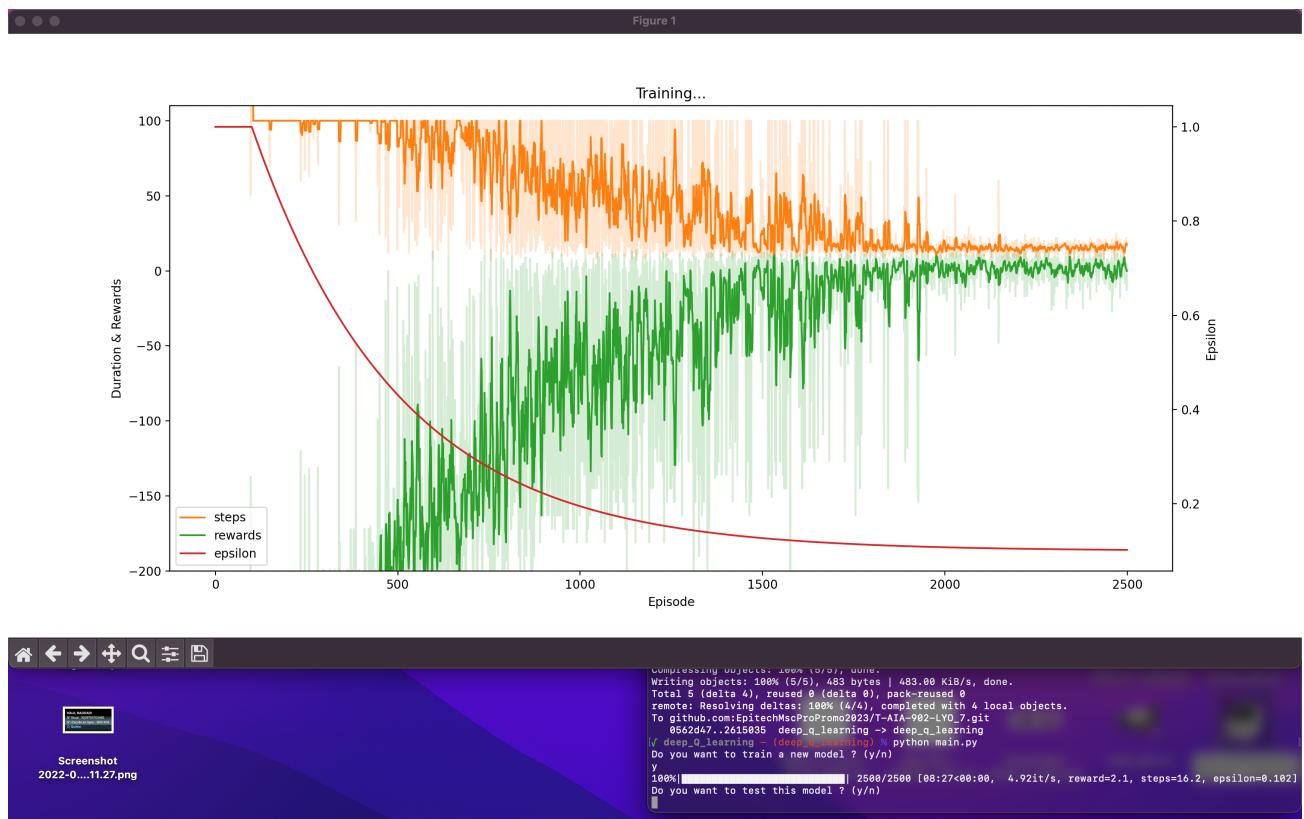
Nous pouvons observer que le modèle se stabilise sur des résultats acceptable à partir de **2500 épisodes d'entraînement**, ce qui équivaut à un peu plus de **5 minutes d'entraînement**.

Ces metrics démontrent que l'**algorithme n'est pas adapté dans notre cas**, notamment du au fait de la **très petite taille de notre environnement** et du **nombre très restreint d'actions** possibles et ce malgré un bon fonctionnement de l'algorithme.

En effet, les algorithme de deep Q learning se basant sur un réseau de neurones sont très **performant dans des environnements particulièrement grand et complet**. Il démontre cependant une faiblesse au niveau du temps d'exécution nécessaire pour obtenir des résultats convenable sur un environnement tel que celui du *taxi driver*, étant **de l'ordre de 5 min contre 2 secondes** pour le vanilla Q learning.

Notre implémentation de l'algorithme de deep Q learning **permet également de reprendre l'entraînement d'un modèle déjà entraîné** une première fois.

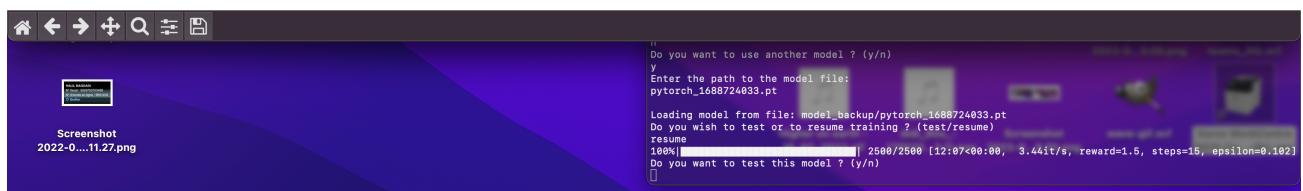
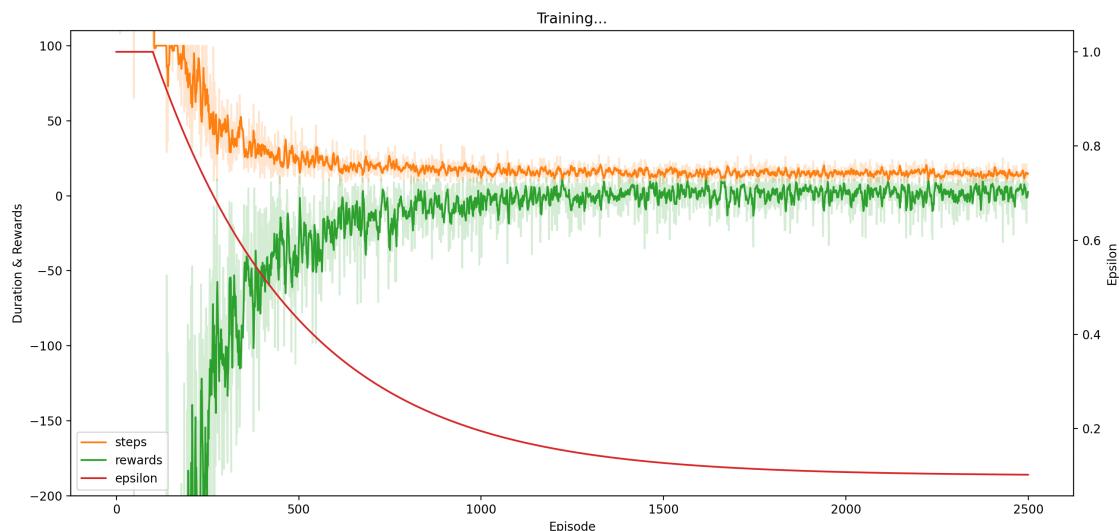
Dans l'image ci-dessous, nous pouvons voir les résultats obtenus lors d'une première session d'entraînement comprenant 2500 épisodes.



Cet entraînement affiche des résultats très convenable et réguliers **à partir de 2000 épisodes**.

Après avoir enregistré notre modèle à la suite de cet entraînement, nous l'avons de nouveau entraîné avec les mêmes paramètres, toujours sur une session de 2500 épisodes.

Figure 1



Comme nous pouvons clairement le voir sur ce deuxième graphique, l'algorithme a pu reprendre l'entraînement, en démontre les résultats bien plus rapide avec des metrics similaires **entre les 500 et les 1000 premiers épisodes**.

Dans le cas d'une application de ce genre d'algorithme sur un environnement bien plus conséquent, cette méthode pourrait permettre d'obtenir des résultats assez rapidement, tout en permettant de poursuivre l'entraînement du modèle par la suite, afin d'optimiser ses performances sans avoir à reprendre l'entraînement depuis un modèle vierge.

Sarsa

SARSA est une **méthode basée sur les valeurs**, similaire au **Q-learning**. Elle utilise donc une **Q-table** pour stocker les valeurs de chaque paire état-action. Avec les stratégies basées sur la valeur, nous formons **l'agent indirectement** en lui apprenant à **identifier les états** (ou les paires état-action) qui ont le **plus de valeur**.

Algorithme Sarsa

L'algorithme **SARSA** est un algorithme "**on-policy**", ce qui le **différencie** du **Q-Learning** (algorithme "**off-policy**"). **On-policy signifie** que pendant la formation, nous **utilisons** la **même politique** pour que l'**agent agisse** (politique d'action) **et pour mettre à jour la fonction de valeur** (politique de mise à jour). En revanche, avec l'approche hors politique, nous utilisons des politiques différentes pour l'action et la mise à jour.

Examinons maintenant l'algorithme SARSA lui-même :

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$$

Diagram illustrating the components of the SARSA update equation:

- Updated Q estimate for state-action pair**: The result of the update, shown at the top left.
- Learning Rate**: A parameter that scales the difference between the old and new Q-values.
- Discount Factor**: A parameter that discounts future rewards.
- Current Q estimate for state-action pair**: The input Q-value for the current state-action pair.
- Reward received following the action taken**: The immediate reward received after taking the action.
- Value of the next state-next action pair**: The estimated value of the next state-action pair.
- Current Q estimate for state-action pair**: The input Q-value for the next state-action pair.

- **Q** est la **fonction de valeur**, et le terme de gauche $Q(S_t, A_t)$ est la **nouvelle valeur** pour la **paire état-action** spécifique. Remarque : **S** fait référence à l'**état** et **A** à l'**action**.
- Du côté droit de l'équation, nous trouvons le **même terme $Q(S_t, A_t)$** , qui, dans ce cas, **est la valeur actuelle** pour cette **même paire état-action**.
- Pour **mettre à jour la valeur actuelle**, nous prenons la **récompense (R_{t+1})** à la suite de l'action entreprise par l'agent, nous **ajoutons** la **valeur de la paire état-action suivante $\gamma Q(S_{t+1}, A_{t+1})$ actualisée par gamma**, et nous **soustrayons** la **valeur actuelle $Q(S_t, A_t)$** .
- Ainsi, les **termes** entre crochets **produisent une valeur positive, nulle ou négative**, ce qui **entraîne** une **augmentation**, un **statu quo** ou une **diminution** de la **nouvelle valeur de $Q(S_t, A_t)$** . Il convient de noter que nous **appliquons également un taux d'apprentissage (alpha)** pour **contrôler la "taille" de chaque mise à jour**.

Comme SARSA utilise l'approche de la différence temporelle (TD), l'algorithme continuera à mettre à jour la Q-table après chaque étape jusqu'à ce que nous atteignions le nombre maximum d'itérations ou que la solution converge vers une solution optimale.

Comparaison algorithme Q-learning

Si nous examinons l'équation utilisée par l'algorithme de Q-Learning, nous constatons que la différence réside dans la manière dont il sélectionne la valeur de l'état suivant. En d'autres termes, Q-Learning prend la valeur maximale pour l'état suivant en se basant sur les valeurs existantes dans la table Q. SARSA, quant à lui, prend la valeur de la paire état suivant-action suivante, comme nous l'avons vu plus haut.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha [R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Diagram illustrating the components of the Q-Learning update equation:

- Updated Q estimate for state-action pair** (pointing to the first $Q(S_t, A_t)$)
- Learning Rate** (pointing to the α term)
- Discount Factor** (pointing to the γ term)
- Current Q estimate for state-action pair** (pointing to the second $Q(S_t, A_t)$)
- Reward received following the action taken** (pointing to the R_{t+1} term)
- Maximum value of the next state** (pointing to the $\max_a Q(S_{t+1}, a)$ term)
- Current Q estimate for state-action pair** (pointing to the final $Q(S_t, A_t)$)

Comme pour le Q-Learning, notre algorithme a été mis en place à l'aide de la librairie **Numpy** pour la gestion des **numpy array** ainsi que les méthodes disponibles. Matplotlib est utilisé pour l'analyse de l'algorithme.

Avant de commencer l'analyse de l'algorithme, nous allons devoir introduire des **concepts clés** dans l'apprentissage par renforcement qui sont l'**exploration** et l'**exploitation** et du dilemme qu'ils représentent car ce sont **eux** qui vont aider l'**agent** à mieux construire sa prise de décision.

1. L'**exploration** consiste à prendre des **actions** qui permettent à notre **agent** de **découvrir** de **nouvelles parties de l'environnement** et de **nouvelles expériences**. L'exploration est importante car elle permet à l'agent de découvrir de nouvelles options d'action et de récompense qui peuvent lui être bénéfiques à long terme.
2. L'**exploitation**, en revanche, consiste à **utiliser** les **connaissances acquises** par notre **agent** pour maximiser sa récompense à court terme. Cela signifie que l'agent prend des actions qui ont été bénéfiques dans le passé, en espérant obtenir une récompense similaire à nouveau.

Trouver le bon **équilibre** entre l'**exploration** et l'**exploitation** est un **défi essentiel** en apprentissage par renforcement.

Si l'**agent explore** de manière **excessive**, il risque de **gaspiller** du **temps** en essayant des actions qui sont inefficaces au lieu de se concentrer sur celles qui sont déjà connues pour être efficaces.

Si l'**agent se concentre** exclusivement sur l'**exploitation**, il risque de **passer à côté** d'opportunités d'apprendre de nouvelles informations et d'améliorer sa politique de prise de décision.

Evaluation de l'algorithme

Paramètres de départ :

```
# SARSA parameters
alpha = 0.5    # learning rate
gamma = 0.80   # discount factor

# Training parameters
n_episodes = 10000  # number of episodes to use for training
n_max_steps = 200    # maximum number of steps per episode

# Exploration / Exploitation parameters
start_epsilon = 1.0  # start training by selecting purely random actions
min_epsilon = 0.0    # the lowest epsilon allowed to decay to
decay_rate = 0.001   # epsilon will gradually decay so we do less exploring and more exploiting
```

Pour équilibrer l'**exploration** vs l'**exploitation**, nous allons **varier epsilon** tout au long de l'entraînement. Nous commencerons avec epsilon=1 (exploration pure) et diminuerons epsilon à chaque épisode pour passer progressivement de l'exploration pure à l'exploitation.

Quand à l'**optimisation** de l'**algorithme** lui même nous allons **changer** les valeurs **alpha** et **gamma** pour évaluer l'entraînement.

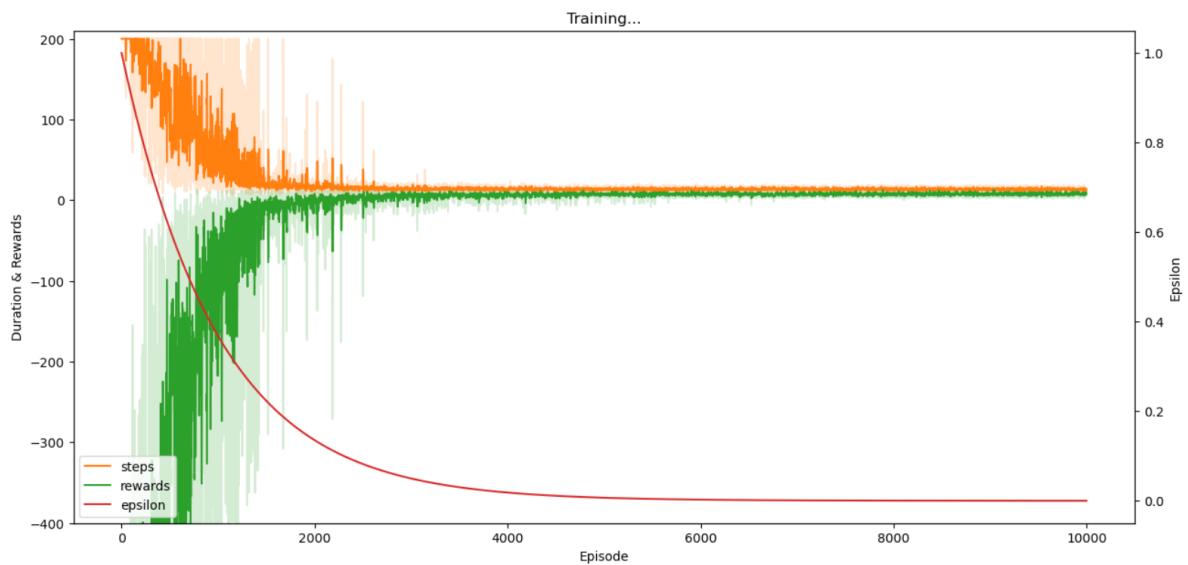
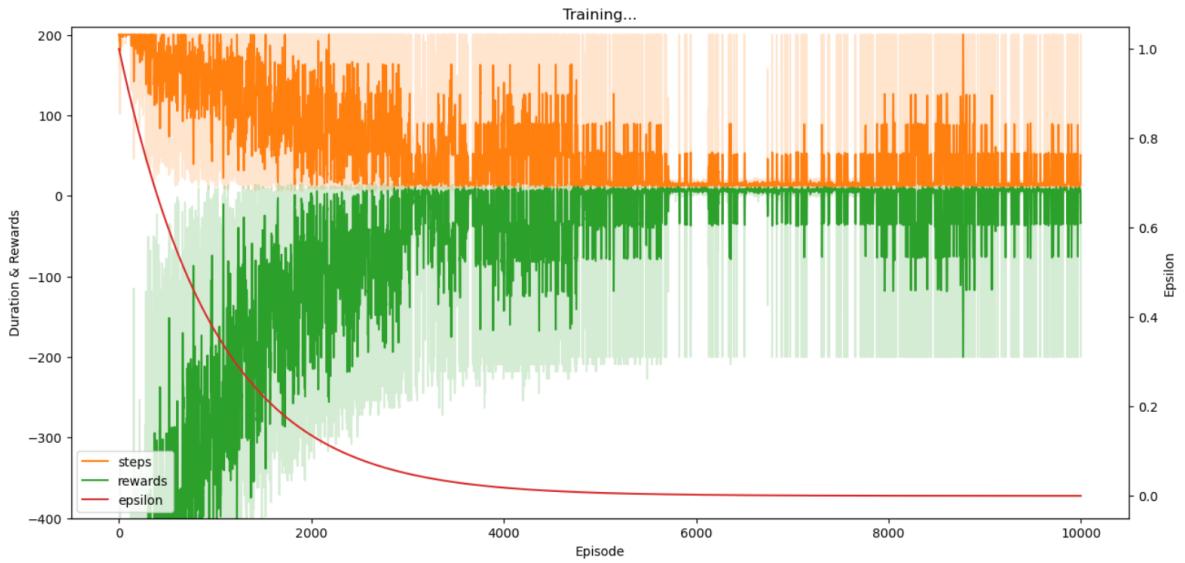
- Valeur **alpha (α)** : **Alpha** est également appelé le **taux d'apprentissage**. Il contrôle dans quelle mesure l'agent met à jour ses estimations de la valeur des états ou des actions.
- Valeur **gamma (γ)** : **Gamma** est également appelé le **facteur de remise ou le taux d'actualisation**. Il contrôle l'importance accordée aux récompenses futures par rapport aux récompenses immédiates.

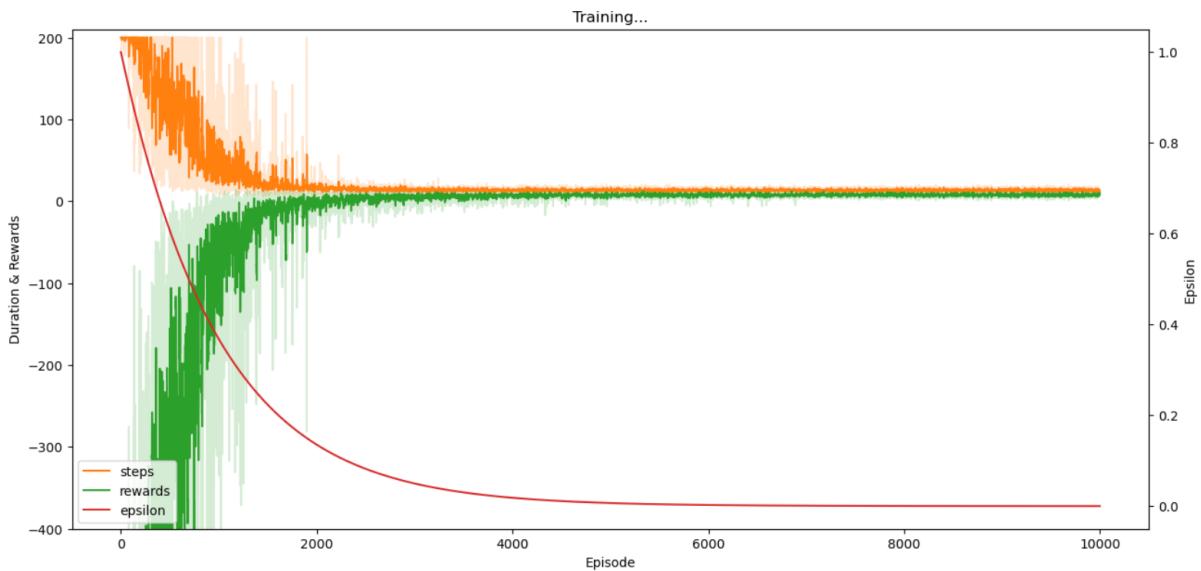
Une **valeur élevée d'alpha** signifie que les **estimations** sont **mises à jour rapidement**, ce qui peut rendre l'apprentissage **instable**. Une **valeur faible** d'alpha signifie que les **estimations** sont **mises à jour lentement**, ce qui peut entraîner un **apprentissage plus stable, mais plus lent**.

Une **valeur de gamma proche de zéro** signifie que l'agent se **concentre** uniquement sur les **récompenses immédiates, tandis qu'une valeur proche de un** signifie que l'agent prend également **en compte les récompenses futures à long terme**.

Nous avons donc réalisé un entraînement test avec un alpha à 0.5 et un gamma à 0.8 pour que l'agent prenne en compte l'importance des récompenses futures. ce test sera évalué au travers d'un **graphique** créer via la librairie **Matplotlib**.

Ce graphique va mettre en avant la mesure **d'epsilon**, du nombre de **steps** et valeurs de la **récompense** par epoch lors de l'entraînement.





Graphique 3, Paramètres : $\alpha=0.2$; $\gamma=0.98$

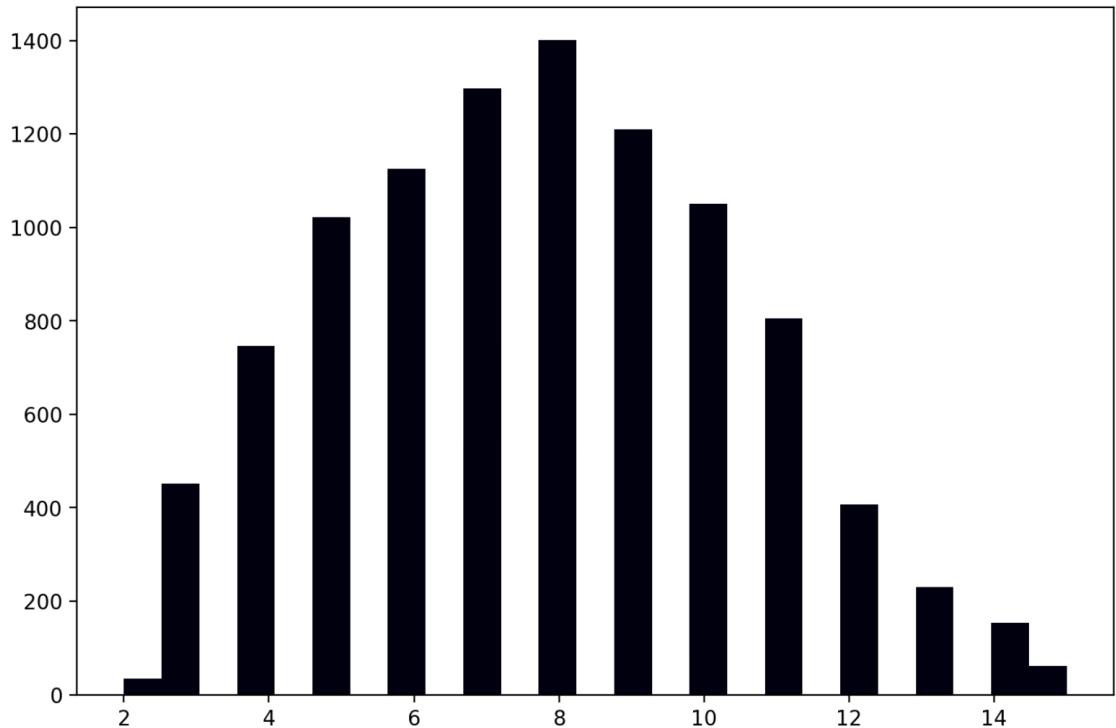
Nous observons une légère amélioration sur ce graphique avec un nombre de steps nécessaires à la réalisation d'un épisode compris entre 50 et 200 à l'épisode 0 pour se stabiliser à partir de l'épisode 2100 avec un nombre de steps allant de 0 à 25. La récompense est entre -400 et -90 à l'épisode 0 et entre -30 et 0 à partir de l'épisode 2300 puis se stabilise. La valeur d'epsilon est à 1 à l'épisode 0 et arrive à 0 entre l'épisode 4000 et 5000.

Nous avons également plusieurs mesures qui sont le nombre moyen d'action par épisode, l'écart-type ainsi que la valeur minimale et maximale de la récompense.

Average timesteps per episode: 13.2207
Mean Reward = 7.78 +/- 2.69
Min = 2.0 and Max 15.0

- La **récompense moyenne** sur **10 000 épisodes** est de **7,78 +/- 2,69**. Cela indique une certaine variation dans les récompenses qui est due au changement des emplacements d'un taxi, d'un passager et d'une dépose au début de chaque épisode. Il nous faut donc à chaque fois un nombre différent d'étapes pour accomplir la tâche, et nous ne recevons donc pas toujours la même récompense (chaque déplacement est moins 1 point).
- La **récompense maximale** étant de **15**, nous indique que nous ne pouvons pas espérer plus. Si un passager est initialisé au lieu de dépose, l'épisode se termine immédiatement (ce n'est donc pas un scénario valable et il n'y a aucun espoir d'obtenir un score de 20). Par conséquent, le meilleur cas que nous puissions espérer est celui où le passager et le taxi sont initialisés au même endroit. Si nous prenons les deux carrés de couleur les plus proches (rouge et jaune), l'agent utilisera **5 mouvements (1 pour prendre et 4 pour déplacer)** pour transporter le passager du lieu de prise en charge au lieu de dépose, ce qui nous donne la **récompense maximale possible de 15** ($20 - 5$).
- la moyenne de **13,2** action par épisode est similaire au nombre d'action du Q-learning qui est situé entre 10 et 15 actions par épisode.

Rewards distribution from evaluation



Graphique de distribution des récompenses

Le **graphique de distribution** des récompenses indique une **distribution quasi normale** des récompenses. Ainsi, bien qu'il **ne prouve pas** que nous ayons **une politique optimale**, il **suggère** que l'agent fait des **choix rationnels**.

Monte Carlo

L'algorithme de Monte Carlo s'appuie sur un **échantillonnage aléatoire répété** pour obtenir des résultats numériques. L'idée est d'**utiliser le hasard pour résoudre des problèmes** qui pourraient être déterministes en principe, mais qui sont **trop complexes à résoudre directement**.

Définition et principe du Monte Carlo

Le fonctionnement des algorithmes de Monte Carlo reposent sur plusieurs aspects clés.

Ils effectuent d'abord des expériences informatiques en utilisant **un échantillonnage aléatoire et des statistiques de probabilités** pour obtenir des résultats, à la manière du vanilla Q learning ou du Sarsa.

Au lieu d'examiner systématiquement tous les états possibles (contrairement aux algorithmes de deep Q learning), la méthode de Monte Carlo échantillonne aléatoirement parmi les états possibles sur la base de distributions de probabilités spécifiées. Cela permet d'**explorer efficacement de grands espaces de problèmes**.

L'algorithme transforme ensuite des calculs déterministes difficiles en **problèmes d'échantillonnage aléatoires beaucoup plus faciles à résoudre**. Cela repose sur la loi des grands nombres ; plus on agrège des échantillons aléatoires, plus les résultats agrégés convergent vers la distribution attendue.

Les applications courantes d'algorithme tel que celui-ci incluent l'intégration numérique, l'optimisation, les simulations de Monte Carlo et les méthodes de Monte Carlo par chaînes de Markov pour l'échantillonnage de distributions de probabilité.

Ces algorithmes suivent un **flux de travail itératif** de base. Ils commencent par définir un domaine d'entrées possibles. À partir de ces entrées, l'algorithme devra générer des entrées aléatoires selon une distribution de probabilité sur le domaine. Il effectuera ensuite un calcul déterministe sur ces dernières avant de finalement agréger les résultats sur de nombreux échantillons aléatoires pour estimer la solution réelle.

La méthode de Monte Carlo fait un **compromis entre précision et efficacité** par rapport aux algorithmes déterministes. Plus il y a d'échantillons, plus la variance de la solution est réduite, augmentant ainsi la précision.

Notre implémentation

Dans le cadre de notre projet, l'implémentation du Monte Carlo s'est déroulé en respectant ces étapes.

Les épisodes sont **générés avec une politique aléatoire**. À chaque étape, une action est donc choisie au hasard selon une distribution uniforme et ce, jusqu'à ce que l'épisode soit résolu.

L'algorithme viendra alors estimer la valeur de chaque état à un instant t en calculant la récompense obtenue à partir de chaque état visité. Dans un second temps, il est important d'estimer la valeur de l'état comme la moyenne des retours obtenus à partir de cet état.

Il nous faut ensuite **améliorer la politique greedy** par rapport aux valeurs estimées. Pour ce faire, l'algorithme choisi l'action qui mène à l'état de plus grande valeur estimée, puis réitère les étapes précédentes en utilisant cette politique améliorée.

En répétant suffisamment d'épisodes, l'algorithme viendra naturellement converger vers la solution optimale. Plus le nombre d'épisodes est important, plus l'estimation des valeurs d'états s'améliore, et de fait **la politique greedy converge vers une politique optimale**.

En résumé, on explore aléatoirement l'environnement pour estimer les valeurs, puis on exploite ces estimations pour améliorer la politique et se rapprocher de l'optimal. La répétition permet de converger vers la solution.

Comparaison avec les autres méthodes explorées

Il est à noter que le Monte Carlo se distingue du vanilla Q learning sur plusieurs points. Tout d'abord il **apprend à partir d'épisodes complets**, ce qui le rend très peu efficace pour des épisodes très longs ou pour des tâches continues sans états de fin. Le vanilla Q learning quant à lui apprend à partir des états individuels dits “de transition”.

Le Monte Carlo estime la valeur d'une politique donnée alors que le vanilla Q learning estime la valeur d'une action sur une politique optimale.

Finalement, le Monte Carlo, comme le SARSA, estime la valeur d'une politique pendant qu'il l'utilise c'est une **méthode “on-policy”**. Le vanilla Q Learning est “off-policy”, comprendre qu'il génère une politique de comportement en effectuant des actions et en évaluant une politique dite d'estimation, qui peut ne pas être liée à la politique de comportement.

Metrics

Les moyens d'obtenir des *metrics* sur les algorithmes d'apprentissage par renforcement dit "**model-free**" diffèrent des système d'intelligence artificielle basés sur des modèles.

La principale difficulté vient du fait que le modèle entraîné obtenu ne possède **pas de données labellisées** dans l'objectif comparer les résultats avec un résultat attendu pré-défini.

La mise en place d'un algorithme venant modifier les différentes valeurs d'environnement et le nombre de *batch* et d'épisodes d'entraînement serait fastidieuse et prendrait beaucoup de temps à l'exécution sans pour autant nous donner des résultats fiables.

Il est donc important de mettre en place un système de *metrics*, **basé par exemple sur le tableau de Q-values** retourné par l'algorithme d'entraînement du vanilla Q learning ou du Sarsa par exemple.

Conditional value at risk

Le **CVaR** (ou Conditional Value at Risk), est une méthode de *metrics* applicable et très répandue sur les algorithmes d'apprentissage par renforcement. Cette *metrics* est fondée sur le principe de mesure du taux de risque global de l'algorithme en se basant sur la Q-table retournée par l'algorithme d'entraînement d'algorithmes comme le **vanilla Q learning** et le **Sarsa** dans notre cas.

Son résultat sera sous forme de valeur négative, plus la valeur sera proche de 0, plus l'algorithme sera considéré comme fiable.

CONDITIONAL VALUE AT RISK WITH VANILLA Q LEARNING

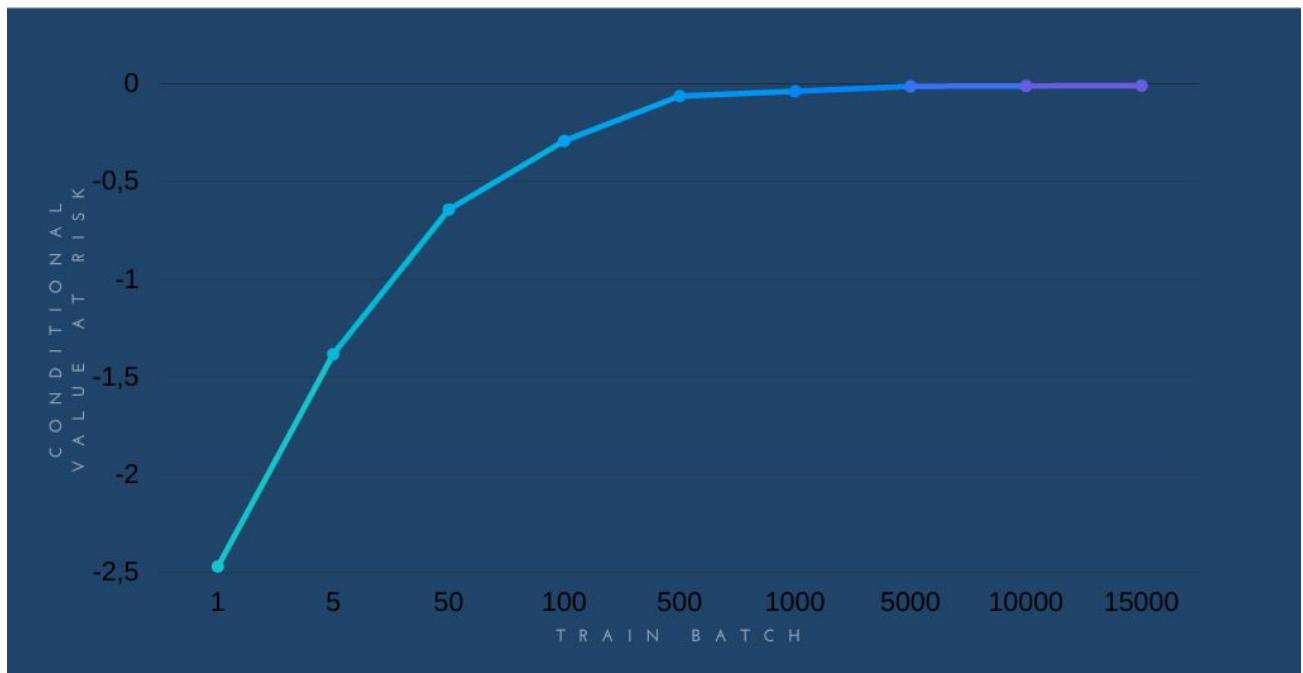


Tableau initié d'après les metrics CVaR du vanilla Q learning

Nous pouvons observer que la courbe de représentation des performances de notre algorithme de vanilla Q learning est similaire à celle représentant la courbe de la variable epsilon dans le graphique d'apprentissage de l'algorithme de deep Q learning.

Au début, celle ci va **augmenter très rapidement à mesure que l'on incrémente le nombre d'éisodes d'entraînement**, avant de finalement **se stabiliser à mesure que l'on approche d'une précision de 100%**.

Conclusion

Tout au long du projet, nous avons expérimenté une multitude d'algorithme d'apprentissage par renforcement ayant tous un fonctionnement bien propre à eux.

Dans notre cas, deux metrics principales nous permettrons de déterminer lequel ou lesquels seront le(s) plus adapté(s) pour résoudre au mieux problématique donnée.

Analyse des metrics - Fiabilité

La première *metric*, sans doute celle qui aura le plus d'impact sur le choix de notre algorithme, est la **fiabilité de l'algorithme et sa capacité à résoudre le problème de manière optimale**.

Dans l'environnement du Taxi Driver de Gymnasium, **la moyenne du nombre d'actions d'un épisode optimal varie entre 10 et 15 action** (selon le placement aléatoire du taxi, du client et de sa destination).

Viennent ensuite **les récompenses finale de chaque épisodes**, qui **oscille entre 7 et 10 points** lors d'épisodes ayant emprunté le chemin le plus optimisé et sans avoir fait d'action illégale durant leur exécution.

Ces deux informations réunies permettent de définir cette **metric de fiabilité** et de déterminer si notre algorithme revoie correctement sa fonction une fois entraîné.

Analyse des metrics - Durée d'exécution

Dans un second temps, il nous faut également se pencher sur **le temps que nos algorithmes mettront à s'entraîner** pour obtenir les résultats escomptés.

Pour se faire, nous avons déterminé le nombre d'épisodes nécessaire à chacun de nos algorithmes pour obtenir des résultats similaires et aussi proche que possible des 100% de fiabilité. Il nous suffisait alors de **comparer le temps que chacun aura mis pour s'entraîner avec ses paramètres optimaux**.

D'abord dans le cas de notre algorithme de **Deep Q Learning**, pour atteindre la meilleure précision générale avec les paramètres optimaux trouvés, il est nécessaires d'effectuer **au moins 2500 épisodes d'entraînement**, ce qui équivaut à **5 minutes en moyenne**.

Concernant notre algorithme de **Vanilla Q Learning**, pour atteindre les meilleures performances avec des **résultats supérieurs à ceux du deep Q learning**, l'algorithme devra effectuer **2000 épisodes d'entraînement**, mais dans ce cas cela représente seulement **moins de deux secondes**.

Notre algorithme de **Sarsa** quant à lui, devras effectuer entre **2100 et 2300 épisodes d'entraînement en 2,18 secondes** pour **atteindre les meilleures performances**.

D'après ces résultats, nous pouvons en conclure que **l'algorithme le plus adapté à notre problématique sera l'algorithme de vanilla Q learning**. En effet, celui propose des **résultats très satisfaisant en seulement 2 secondes d'exécution**.

Cependant, notre implémentation de **l'algorithme Sarsa est également très performantes** et offre des **résultats extrêmement proche** en un temps très similaire, de **près de 2,18 secondes**.

L'algorithme de deep Q learning quant à lui, aurait été bien plus pertinent dans un environnement plus important, proposant de meilleures performances dans ces cas là. La possibilité de reprendre l'entraînement de l'algorithme est également un énorme atout dans ces cas là, mais très peu utile dans notre cas.

En effet, l'approche du vanilla Q learning et du Sarsa, s'appuyant sur la population d'une *Q table* montrera rapidement des faiblesses dans un tel environnement, contrairement à celui du *taxis-driver* de Gymnasium.

Lexique

Exploration : Processus d'essais de nouvelles actions pour mieux comprendre leur effet sur la récompense.

Exploitation : Processus de choix d'actions connues pour être efficaces pour maximiser la récompense.

L'agent : L'agent en apprentissage par renforcement est le composant qui prend la décision de l'action à entreprendre.

Alpha : Désigne le taux d'apprentissage.

Gamma : Désigne la valeur qui contrôle l'importance accordée aux récompenses futures.