**Rapport Data Science**

**STG-6**

Sommaire

[Présentation du groupe 4](#_Toc199515956)

[Description du projet 5](#_Toc199515957)

[Description de l’environnement Gym “Taxi-v3” 6](#_Toc199515959)

[Choix des technos : 7](#_Toc199515960)

[Algorithmes testés 8](#_Toc199515963)

[Comparatif des algorithmes 9](#_Toc199515970)

[Montrer quel algorithme apprend le plus vite et avec le coût le plus bas. 9](#_Toc199515972)

[Q-Learning : 9](#_Toc199515976)

[Optimisation des hyperparamètres (sans lib AutoML) 10](#_Toc199515978)

[Algorithme retenu pour ce projet : le Q-Learning 11](#_Toc199515981)

[Application web 11](#_Toc199515983)

[Annexe : 12](#_Toc199515984)

Présentation du groupe

Florian

Torchy

Mathéo

Vitali

Hamdi

Nassri

Jeffrey

Winkler

Description du projet

**Taxi driver (Taxi-v3)**

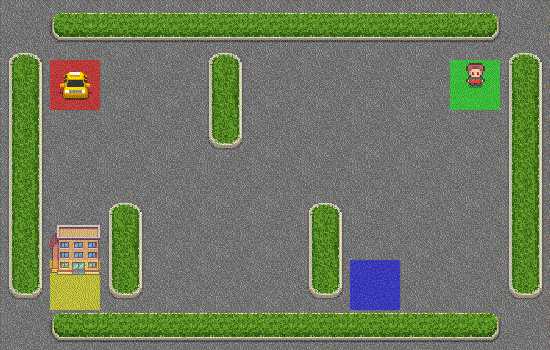
Le but du jeu est simple : contrôler un taxi sur la grille pour aller chercher un passager à son point de départ, le prendre à bord puis le conduire et le déposer à l’hôtel en effectuant le moins de déplacements possibles.

**Objectifs**

* Créer une IA capable de jouer au jeu Taxi-v3
* Choisir et implémenter deux algorithmes de RL
* Utiliser l’exploration ε-greedy
* Utiliser l’entrainement par batch (Deep-Q-Learning obligatoire)
* Optimiser les hyperparamètres sans utiliser de librairie dédiée
* Présenter une procédure d’optimisation des hyperparamètres
* Comparer les deux algorithmes implémentés
* Présenter des benchmarks détaillés des deux algorithmes (rewards, steps, time)
* Mettre en place une interface web permettant de tester les deux algorithmes

Description de l’environnement Gym “Taxi-v3”

**(Source : https://www.gymlibrary.dev/environments/toy\_text/taxi/)**



**Grille :** 5 x 5 cases nommées ligne 0-4 et colonne 0-4

**Emplacements :** Il y a 4 emplacements fixes marquée par une

case de couleur R (red), G (green), Y ( yellow), B (blue).

**Actions:** 0 = South, 1 = North, 2 = East, 3 = West, 4 = Pick-up, 5 = Drop-off

**But du jeu :**

Au début de chaque épisode, le taxi, le passager et l’hotel pop aléatoirement sur une des 4 cases citées ci-dessus. Le taxi doit aller chercher le passager sur sa case et le ramené à l’hotel en le moins de déplacement possible. Pour récupérer le passager, l’agent doit utiliser l’action 4 pick-up et pour le déposer sur une case, l’action 5 drop-off

**Récompenses :**

-1 par déplacement

-10 si pick-up ou drop-off effectué sur une mauvaise case

+20 si passager déposé sur la bonne case (fin d’épisode)

**Choix des technos :**

**Python :**

Langage de référence dans le domaine de l’IA et de l’apprentissage par renforcement.

Son principal avantage est qu’il dispose d’un catalogue de librairies plutôt riche et utiles à notre projet, comme NumPy, pandas, PyTorch et bien d’autres …

Dans le cadre de ce projet, Python nous permet :

* D’implémenter « facilement » nos algorithmes de RL (Q-Learning et Deep-Q-Learning)
* De manipuler nos données d’entrainement et de test
* D’automatiser l’optimisation des hyperparamètres avec nos propres fonctions
* De générer des benchmarks

Pour terminer, Python est simple à lire et à maintenir, ce qui a orienté notre choix vers ce langage.

**Streamlit :**

Solution populaire permettant de créer des interfaces interactives avec seulement quelques lignes de codes, pratique pour un projet centré sur l’IA (contrairement à un projet web).

Dans le cadre de notre projet, Streamlit nous permet de :

* Créer rapidement une interface web simple et intuitive
* Sélectionner un algorithme à tester
* Ajuster les hyperparamètres en temps réel
* Lancer un entrainement et afficher les résultats

Pour terminer, cette techno va de pair avec python, ce qui a orienté notre choix vers celle-ci

Algorithmes testés

**Q-learning**

Description

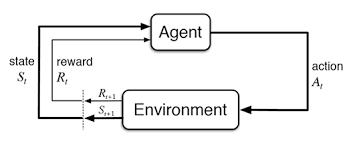
Le Q-learning est l’algorithme d’apprentissage par renforcement le plus connu car il est simple, intuitif et rapide à implémenter.

Il permet à un agent (dans notre cas, un taxi) d’apprendre par lui-même à déterminer la meilleure action à effectuer dans chaque situation grâce à un apprentissage par essais/erreurs.

Le principe repose sur une table appelée **Q-table**, qui permet à l'agent de mémoriser progressivement les meilleures décisions à prendre par la suite.

Fonctionnement de l’algorithme dans Taxi-v3

Durant l’entrainement, le taxi (l’agent) explore différentes actions possibles et ajuste progressivement ses estimations grâce aux interactions avec l’environnement.



Chaque état (state) correspond précisément à :

* La position actuelle du taxi (25 possibilités)
* La position du passager (5 possibilités)
* La destination du passager (4 possibilités)

Ce qui nous donne un total de (25 x 5 x 4) 500 états possibles.

Chaque fois que l’agent réalise une action, il obtient une récompense immédiate et passe à un nouvel état. La Q-table est mise à jour selon la formule suivante :

*Q(état, action) =* *Q(état, action) +* *α[récompense + γ x maxactionsuivante* *Q(étatsuivant, action suivante) -* *Q(état, action)]*

* **α** (learning rate) détermine la vitesse d’apprentissage
* **γ** (discount factor) représente l’importance accordée aux récompenses futures

Pour déterminer l’action à choisir durant l’entrainement, l’agent utilise la stratégie ε-greedy, combinant deux approches nommées exploration/exploitation :

* Avec une probabilité ε (exploration), il sélectionne une action aléatoire afin d’explorer de nouvelles possibilités
* Avec une probabilité 1- ε (exploitation), il sélectionne la meilleure action selon les valeurs de la Q-table

On commence avec un ε élevé pour favoriser l’exploration puis on diminue progressivement au cours de l’apprentissage pour exploiter les données stockées dans la Q-table.

**Exemple de Q-table**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Actions** | | | | | |
| **États** | 0 (Sud) | 1 (Nord) | 2 (Est) | 3 (Ouest) | 4 (Pick-up) | 5 (Drop-off) |
| 0 | -3,2 | -2,4 | -1 | -2,7 | -10 | -10 |
| 1 | -0,5 | -1,3 | -0,7 | -1 | -10 | -10 |
| 2 | -1 | -0,3 | -2 | -2,1 | 9 | -10 |
| ... | ... | ... | ... | ... | ... | ... |
| 499 | -1,2 | -1 | -1,5 | -2,3 | -10 | 20 |

Après entrainement, l’agent dispose pour chaque état d’une estimation de récompense pour chacune des 6 actions possibles, stockée dans la Q-table

Progressivement, la table converge vers la stratégie optimale pour résoudre le jeu Taxi-v3 en le moins de déplacement possible.

Deep-Q-Learning

Description :

L'algorithme **DQN (Deep Q-Network)** repose sur un **réseau de neurones** pour approximer la fonction de valeur d’action Q(s, a), remplaçant ainsi la traditionnelle Q-Table utilisée dans le Q-Learning. Cette approche, bien que plus lente à l’apprentissage au départ, devient essentielle dès que l’on travaille avec des **espaces d’états très vastes ou dynamiques**, où une table serait inefficace ou impossible à maintenir.

**Entraînement du modèle**

L'entraînement repose sur plusieurs composants clés qui permettent au réseau d’apprendre efficacement :

* **Deux réseaux de neurones** :
  + **Policy Network** : c’est le modèle que l’on entraîne activement.
  + **Target Network** : il sert de référence pour calculer les cibles de l’apprentissage. Il est mis à jour moins fréquemment pour stabiliser l'entraînement et éviter des dérives brusques.
* **Mémoire d’expérience (Replay Buffer)** : L’agent stocke ses expériences passées sous forme de transitions (état, action, récompense, nouvel état).
* **Mise à jour des poids du modèle (backpropagation)** : le réseau ajuste ces poids pour s’améliorer

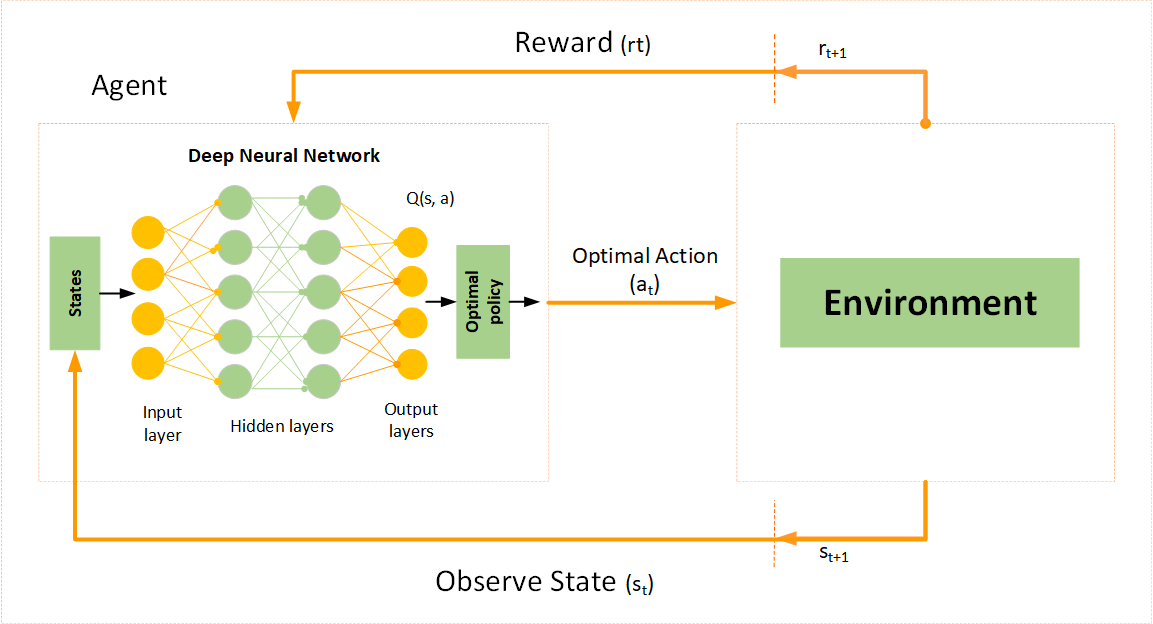
À chaque interaction avec l’environnement, l’agent enregistre ses expériences sous forme de transitions (état, action, récompense, nouvel état) dans une **mémoire tampon** appelée *replay buffer*. Plutôt que d’apprendre en temps réel sur chaque expérience, on extrait aléatoirement des lots d’expériences de cette mémoire pour entraîner le **policy network**. Cela permet de réduire les corrélations temporelles et d’améliorer la stabilité de l’apprentissage. Lors de la mise à jour, le **policy network** ajuste ses prédictions pour se rapprocher de la cible calculée à partir du **target network**. Ce dernier est mis à jour moins fréquemment afin de servir de repère stable et d’éviter que le modèle n’apprenne sur des objectifs qui changeraient en permanence. Ce mécanisme d’apprentissage par répétition et comparaison permet au réseau d’estimer de mieux en mieux les actions optimales à long terme.

**Fonctionnement de l’algorithme une fois le modèle entraîné :**

Une fois le réseau de neurones suffisamment entraîné, son fonctionnement devient simple et efficace :

1. **Observation de l’environnement** : l’état courant est transmis au réseau.
2. **Prédiction des actions** : le réseau évalue la qualité (valeur Q) de chaque action possible.
3. **Choix de l’action** : l’agent choisit l’action ayant la plus forte valeur
4. **Exécution dans l’environnement** : l’action est effectuée, et le cycle recommence.

Ce processus permet à l’agent d’agir de manière optimale en s’appuyant uniquement sur son expérience passée encapsulée dans les poids du réseau.



**Note :** Nous n’avons pas testé le brut force pour le comparatif comme demandé dans le sujet car le temps de traitement est beaucoup trop long pour terminer un épisode.

Comparatif des algorithmes

Objectif du comparatif :

Montrer quel algorithme apprend le plus vite et avec le coût le plus bas.

Le Q-Learning s’est montré très efficace avec peu d’entrainement (1000 épisodes) et très rapide (moins de 2 secondes pour l’entraîner). Sa rapidité est dû à son algorithme peu coûteux en ressource. Sa recherche d’hyperparamètres optimaux est aussi simplifier puisqu’il dispose de peu d’hyperparamètres (6 si on compte le decay et min pour epsilon).

Le DQN, de son côté s’est montré beaucoup plus lent. En moyenne, il faut environ 6 minutes pour l’entraîner et réussir à le faire converger et 5000 épisodes minimum sont nécessaires à son entraînement. Son processus d’apprentissage étant plus coûteux en ressources que le Q-Learning, cela explique pourquoi le temps passé sur chaque épisode est plus long chez le DQN.

Optimisation des hyperparamètres

Q-learning

Learning rate (α):

Sert à déterminer à quelle vitesse l’agent met à jour les valeurs de la Q-table.

* Valeur faible (0,1) l’apprentissage est lent mais plus stable.
* Valeur élevé (0,8) l’apprentissage est rapide mais instable.

Pour l’optimisation de cet hyperparamètre, nous avons décidé de commencer par une valeur de 0,4 et de la diminuer progressivement.

Discount Factor (γ):

Définit l’importance accordée aux récompenses futures par rapport aux récompenses immédiates.

* Valeur faible (0,1) l’agent privilégie les récompenses immédiates.
* Valeur forte (0,9) l’agent privilégie les récompenses futures.

Pour l’optimisation de cet hyperparamètre, nous avons décidé de commencer à 0,6 et d’augmenter la valeur progressivement.

Epsilon (ε):

Détermine à quelle fréquence l’agent teste des actions aléatoires pour explorer.

* Valeur faible (0,01) l’agent exploite les données stockées dans la Q-table.
* Valeur forte (1) l’agent explore constamment en effectuant des actions aléatoires.

Epsilon decay :

Permet de régler la vitesse à laquelle réduit l’exploration (epsilon)

* Valeur faible (0,0001), l’exploration dure plus longtemps, résultat plus pertinent
* Valeur forte (0,01), l’exploration dure moins longtemps, résultat moins pertinent

Epsilon min :

Fixe la valeur minimale d’epsilon sous laquelle on n’explorera jamais.

* Valeur faible (0,01) seulement 1% des actions restent aléatoires après la phase d’exploration
* Valeur forte (0,30) 30% des actions resteront aléatoires après la phase d’exploration

Nombre d’épisodes d’entrainement :

Plus l’agent joue d’épisodes, plus il apprend.

* Peu d’épisode (100) l’agent n’apprend pas assez, valeurs de la Q-table peu fiable
* Beaucoup d’épisodes (50 000) l’agent apprends et fiabilise les valeurs de la Q-table

Pour l’optimisation de cet hyperparamètre, nous avons décidé de commencer à 5000 et d’augmenter la valeur progressivement.

Max steps (paramètre d’environnement)

Limite le nombre de steps autorisés dans un épisode, une fois atteint, l’épisode se termine peu importe l’état de l’agent.

Pratique en début d’entrainement pour réduire le temps d’entrainement et explorer plus rapidement avec des données « fiables ».

Combinaisons d’hyperparamètres et résultats obtenus :



On a d’abord établi un test point de repère en fixant des valeurs dites « raisonnables ».

Ce premier essai nous sert de référence pour mesurer l’impact des réglages ultérieurs (test1).

Nous avons ensuite ajusté le learning rate en réduisant la valeur de 0,40 à 0,20, nous ralentissons ainsi la mise à jour de la Q-table, ce qui stabilise la courbe de récompense.

Grâce à cet apprentissage plus « lent », nous avons pu augmenter le discount factor de 0,60 à 0,95, incitant ainsi l’agent à privilégier les récompenses futures, ce qui améliore encore les résultats obtenus.

Il restait alors à affiner l’exploration. La première valeur donnée à epsilon faisait que celle-ci diminuait trop rapidement, nous avons donc ralenti cela en réduisant la valeur epsilon decay à 0,0005 afin de conserver une exploration aléatoire plus longue, permettant ainsi à l’agent de découvrir de meilleurs trajets.

Enfin, nous avons abaissé la valeur d’epsilon min à 0,01, faisant en sorte qu’après la phase d’exploration, l’agent ne sélectionne plus que 1 % d’actions aléatoires. Cela consolide durablement les gains, tout en limitant le risque de dérive.

Pour finir, nous avons simplement augmenté la durée d’entraînement. Au-delà de 10 000 épisodes, la récompense continue de progresser mais commence à plafonner autour de 15 000 épisodes. Nous avons donc retenu 12 500 épisodes comme compromis idéal entre performance et temps de calcul.

**Run final (test8) :**

α = 0,20

γ = 0,95

ε₀ = 1

ε\_decay = 0,0005

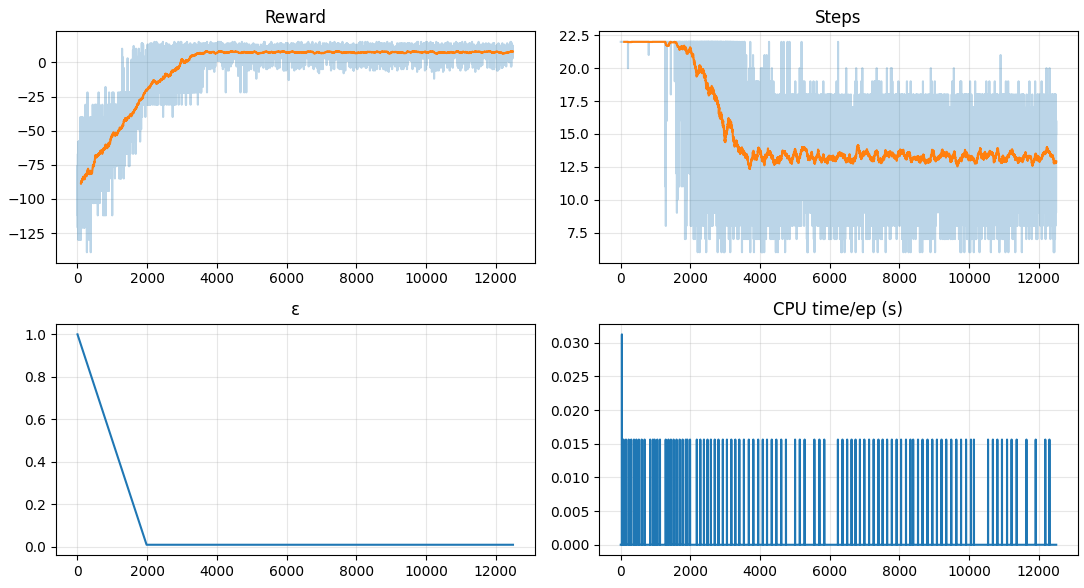
ε\_min = 0,01

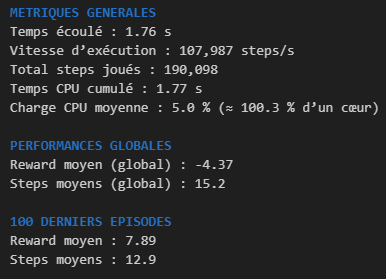
Episodes = 12 500

Max steps = 22

**Résultats :** reward moyen d’environ 7,5 et ~13 steps par mission

**Metriques :**

****



On peut voir que la valeur epsilon chute et se stabilise après 2000 épisodes, elle passe de l’exploration à l’exploitation et va peaufiner les valeurs de la Q-table par la suite en alternant exploration (99%) et exploitation (1%) grâce à epsilon min à 0,01.

A noter que le fait de mettre la valeur de max step à 22 et non 200 (valeur de base d’après nos recherches) nous fait gagner 1s sur le temps de traitement (on supprime les premiers épisodes interminables)

Deep-Q-learning (à rédiger)

Liste hyperparamètres:

* Learning rate
* Gamma
* Epsilon
* Epsilon\_decay: vitesse à laquelle epsilon va se réduire
* Epsilon\_min: la valeur minimale de l’exploration
* Memory\_size: taille de la mémoire de souvenir pour l’entraînement
* Batch\_size: taille de l’échantillon de mémoire utilisé pour entraîner le modèle (je n’ai pas fait la sécu mais ça ne peut pas être plus grand que memory\_size)
* Target\_network\_update\_frequency: fréquence à laquelle le réseau de neurones target est mis à jour (il copie juste le réseau policy)
* Nombre max d’épisodes

Algorithme retenu pour ce projet : le Q-Learning

Pourquoi ce choix ?

La grille est petite et possède un nombre limité d’états (North, South, East, West, Pick-up, Drop-off), cela tient parfaitement dans une Q-Table.

Utiliser un réseau neurones serait trop “excessif” pour ce projet

Cet algorithme permet de résoudre ce problème en seulement 16 steps (en moyenne) en peu d'épisode (**voir learning curve, à renseigner**), ce qui nous donne un faible coût de temps de calcul.

Application web

[**Repository Github STG-6**](https://github.com/Ftorchy/T-AIA-902/tree/main)

