T-AIA-902 : Apprentissage par renforcement

Hugo ALLEGAERT, Florian SORIN, Sarah PETANEC et Oren COHEN, 19 juin 2022

Objectif

L'objectif du projet est de résoudre le jeu **Taxi-v3** de la librairie *Gym* en utilisant un algorithme d'apprentissage épisodique optimisé. Les règles du jeu sont simples : lorsque l'épisode commence, le taxi démarre sur une case aléatoire et le passager se trouve aussi à un endroit aléatoire. Le taxi se rend à l'emplacement du passager, prend le passager, le conduit jusqu'à sa destination (un autre des quatre emplacements spécifiés), puis dépose le passager. Une fois le passager déposé, l'épisode se termine. La carte est une grille de 25 cases, il y a 5 positions possibles pour le passager et 4 emplacements de destination, ce qui donne 500 observations possibles au total. Le score final est calculé comme suit :

- - 1 point par étape (sauf si une autre récompense est déclenchée).
- + 20 points pour déposer le passager à sa destination.
- - 10 points si une action dites de « ramassage » ou de « dépôt » est exécutée illégalement.

Le score maximal d'une partie sera donc compris entre 7 et 15 en fonction de l'emplacement de départ du taxi, du passager et de la destination.

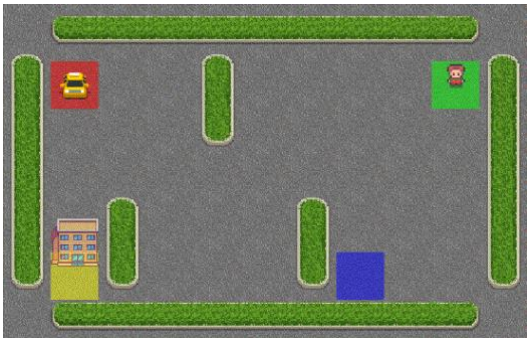


Figure 1 : représentation visuelle d'une partie de Taxi-v3

Brute force

La première méthode que nous avons utilisée afin de résoudre le jeu et avoir une base de comparaison pour nos prochains algorithmes a été la manière dite « Brute force ». Afin de finir le jeu l'agent exécute une action aléatoire parmi les 6 actions proposées par l'environnement, et ce, jusqu'à temps qu'il finisse le jeu.

Après l'avoir fait jouer 10 000 parties voici les résultats obtenus :

En moyenne il faut à l'agent 196 étapes (à noter que le jeu se finit automatiquement après 200 étapes) et reçoit un score de -772 points pour 64 pénalités. Le meilleur score qu'il ait réussi à obtenir est de 14 points, pour 7 étapes et 0 pénalités.

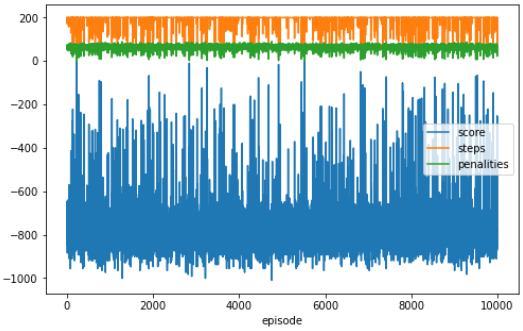


Figure 2 : scores (bleu), étapes (orange) et pénalités (vert) obtenus à chaque épisode (axe x)

	score	steps	penalities
mean	-772.1707	196.7545	64.0413
max	14.0000	200.0000	90.0000
min	-1010.0000	7.0000	0.0000

Figure 3 : Moyenne, maximum et minimum des scores, étapes et pénalités obtenus par l'agent lors de ses 10000 parties.

Q-Learning

Par la suite, afin d'avoir une IA réellement capable d'apprentissage et pouvant résoudre le jeu de façon stable et non pas aléatoirement comme précédemment, nous avons choisi d'implémenter l'algorithme dit de « Q-Learning ». Avec cette algorithme, l'agent utilise les récompenses renvoyées par l'environnement à chaque action afin de choisir la meilleure action à utiliser dans un état donné. Pour ce faire, il dresse un tableau avec pour chaque paires état-action une valeur Q correspondant à l'avantage de choisir cette action dans l'état donné. Une valeur de Q plus élevée, implique de meilleures chances d'obtenir de plus grandes récompenses. Au fur et à mesure que l'agent explore l'environnement les valeurs de Q sont mises à jour selon l'équation de Bellman suivante :

$$Q(state,action) \leftarrow (1-\alpha)Q(state,action)+\alpha(reward+\gamma \max Q(next\ state,all\ actions))$$

On peut constater la présence de 2 hyperparamètres au sein de cette fonction :

- α (Learning rate) : valeur utilisée pour déterminer à quelle vitesse nous allons abandonner les valeurs de Q précédemment enregistré, plus la valeur de α est élevée, plus les valeurs de Q seront remplacées rapidement. Dans notre cas nous utilisons α = 0.1.
- γ (gamma) : valeur comprise entre 1 et 0, utilisée pour définir l'importance que l'on accorde aux récompenses futures. Plus la valeur est proche de 0 plus l'agent donnera d'importance au récompenses immédiate et vis-versa. Dans notre cas, nous utilisons γ = 0.99.

Avec ces paramètres donnés, on peut constater que l'IA prend environ 2500 épisodes pour déterminer les valeurs de Q optimales, ensuite, ses résultats cesseront de s'améliorer (Figure 4) et nous pouvons passer à la phase d'évaluation.

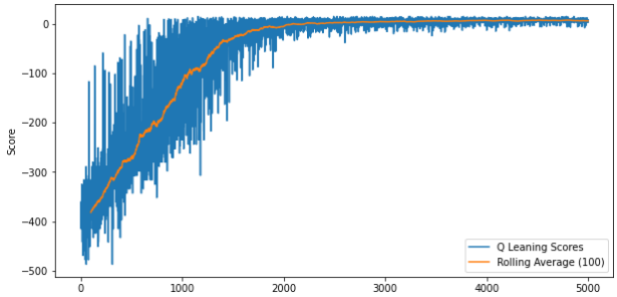


Figure 4 : Courbe d'apprentissage de l'algorithme Q-Learning, l'axe x représente le nombre d'épisode joué et l'axe y le score obtenu par l'agent pour l'épisode x.

Évaluation

Afin d'évaluer notre IA et de comparer ses performances, nous lui avons fait passer le même test que pour le « Brute force ». Ainsi après 10 000 parties jouées, notre agent achève un score moyen de 7, pour 13 étapes et 0.06 pénalités. Il lui faut au maximum 22 étapes afin de terminer une partie, ce qui signifie qu'elle n'échoue jamais à résoudre le jeu, dont la limite est fixée à 200 étapes, alors que précédemment la méthode « Brute force » résolvait le jeu seulement 4.46 % du temps. On peut cependant remarquer qu'il arrive toujours à notre agent de recevoir des pénalités (6.64 % du temps), prouvant ainsi que des optimisations sont possibles.

	score	steps	penalities
mean	7.0502	13.3288	0.069
max	15.0000	22.0000	3.000
min	-23.0000	6.0000	0.000

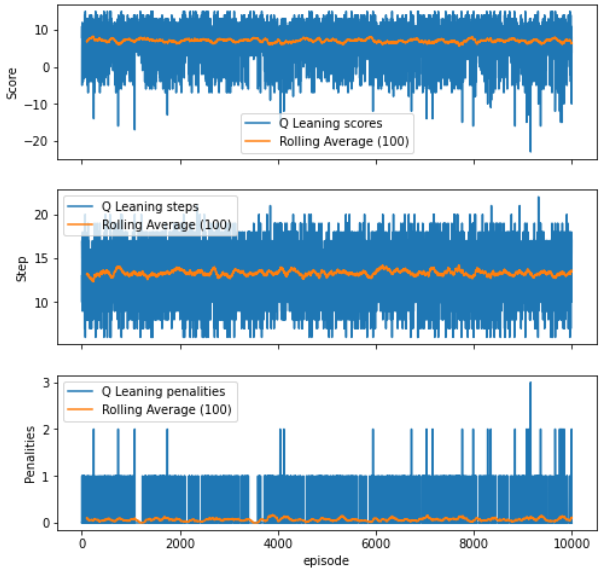


Figure 5 : tableau statistiques et résultats obtenus par l'agent durant les 10 000 parties d'évaluation (axe x).

Deep Q-Learning

Aux vues des résultats encourageants du Q-Learning mais pas parfait pour résoudre notre problème, nous avons choisi d'implémenter un algorithme dit de « Deep Q-Learning ». Car effectivement si le Q-Learning semble simple et robuste, si l'environnement dispose d'un grand nombre d'états et d'actions, alors, le tableau stockant les valeurs de Q auras une taille bien trop élevée, créant ainsi des problèmes de mémoires et de temps d'exploration. Pour pallier ce problème, le Deep Q-Learning (DQN) utilise un réseau de neurones qui se concentre sur l'apprentissage d'un approximateur de fonction, pour prédire les valeurs de Q qui satisferont l'équation récursive de Bellman. Ainsi en passant un état donné du jeu à notre algorithme, celui-ci nous renverra une valeur de Q approximative pour chaque action possible.

Pour entraîner un agent avec notre implémentation du DQN, voici la liste des hyperparamètres configurable et leurs effets sur l'apprentissage :

- ***Learning rate*** (lr) : comme précédemment pour le Q-Learning, le lr est la valeur utilisée pour déterminer à quelle vitesse on abandonne les valeurs de Q précédemment apprise. Ici, pour optimiser l'apprentissage, nous avons ajouté la possibilité de choisir une valeur de départ (lr) ainsi qu'une valeur minimale (lr_min) pour le Learning rate, qui sera atteinte en décroissant de façon exponentielle après un nombre d'épisodes choisit par l'utilisateur (lr_decay).

Figure 6 : résultat d'un entrainement avec un lr de 0.1 (valeur élevée). Le fait de ne pas suffisamment prendre en compte les valeurs de Q précédemment calculées ne lui permet pas d'apprendre de ses erreurs et tend à rendre son comportement aléatoire.

- ***Gamma*** : valeur comprise entre 1 et 0, utilisée pour définir l'importance que l'on accorde aux récompenses futures. Dans notre cas si la valeur de gamma est trop faible, l'agent cherchera à maximiser ses récompenses immédiates sans prendre en compte les potentielles récompenses futures.

Figure 7 : résultat d'un entrainement avec une valeur de gamma à 0.10 (faible). On peut constater que l'agent maximise bien ses récompenses immédiates en ne commettant plus d'action illégale qui ferait baisser son score. En revanche le fait de ne pas prendre en compte la possibilité de finir le jeu dans un état futur qui lui ferait gagner davantage de récompenses l'empêche de prendre des décisions correctes qui lui permettrait de résoudre l'environnement.

- ***Memory size*** : lors de son entraînement, l'agent stocke dans sa mémoire les coups qu'il a joué ainsi que la récompense obtenue et l'état du jeu avant et après l'action. Afin de ne pas fausser la distribution des données, le DQN échantillonne des petits lots de cette mémoire qu'il passera au réseau de neurone pour son apprentissage.

Figure 7 : résultat d'un entrainement avec une mémoire de 1000 (valeur faible). On constate que l'agent ne réussit pas à résoudre l'environnement, le fait d'avoir une mémoire trop basse l'empêche de se rappeler des actions et des états qui lui on permit de finir le jeu, bloquant ainsi son entraînement.

- ***Batch size*** : valeur correspondant à la taille des lots venant de la mémoire que le DQN passera au réseau de neurones. Une valeur trop faible aura le mêmes effets qu'une valeur trop basse de la taille de la mémoire. Mais une valeur trop élevée aura pour conséquence de fortement ralentir le temps d'apprentissage (temps réel, du fait du nombre de calculs plus élevé requis) sans pour autant améliorer les performances.

- ***Epsilon*** : dans l'objectif de correctement explorer l'environnement durant l'entraînement, comme pour le Q-Learning, nous utilisons la stratégie d'exploration « Epsilon-Greedy ». Pour ce faire, l'agent choisit une action aléatoire de probabilité epsilon et exploite la meilleure action connue de probabilité 1 - epsilon. De la même manière que pour le Learning rate, notre implémentation du DQN offre la possibilité de choisir la valeur de départ d'epsilon (eps_start), ainsi que sa valeur minimale (eps_min), qui sera atteinte en décroissant de façon exponentielle après un nombre d'épisodes choisit par l'utilisateur (eps_decay).

Figure 8 : résultat d'un entraînement avec une valeur d'epsilon de 0.1 atteinte en 50 épisodes. Après 50 épisodes joués, l'agent n'explore plus son environnement, comme ici durant les 50 premiers épisodes il n'a jamais réussi à résoudre l'environnement, il lui sera impossible de le faire par la suite du fait de ses connaissances de l'environnement qui sont trop limités. D'où l'importance d'une phase d'exploration suffisamment longue.

- ***Fonction de perte*** (loss function) : nous avons choisi de laisser à l'utilisateur le choix de la loss function à utiliser par le réseau de neurones durant l'entraînement, étant donné que le choix de celle-ci dépend du set de données. Il y a 3 choix possibles : Mean Squared Error (mse), Mean Absolute Error (mae) et Huber loss (huber). Pour simplifier, vous voudrez utiliser la loss function Huber chaque fois que vous sentirez que vous avez besoin d'un équilibre entre donner un peu de poids aux valeurs aberrantes, mais pas trop. Pour les cas où les valeurs aberrantes sont très importantes pour vous, utilisez la MSE. Et enfin pour les cas où vous ne vous souciez pas du tout des valeurs aberrantes, utilisez la MAE.

Figure 9 : résultat d'un entrainement en utilisant chacune des 3 fonctions de loss (de haut en bas MSE, HUBER, MAE). On peut constater que l'entrainement utilisant la fonction de perte MSE obtient de bien meilleurs résultats, cela est dut au fait que dans notre set de données les valeurs aberrantes correspondent à la résolution de l'environnement ou aux pénalités. Il est donc capital de donner de l'importance à ses valeurs.

- ***Fréquence de mise à jour du réseau cible*** (target network update) : une optimisation permettant de conduire à une plus grande stabilité et efficacité dans le processus d'apprentissage est l'utilisation de deux réseaux de neurones. Effectivement en utilisant un deuxième réseau cible (target network) non entraîné pour évaluer les valeurs de Q, nous nous assurons que les valeurs de Q cibles sont stables pendant une période donnée. Cependant, nous savons que les valeurs cibles ne seront pas les meilleures au début de l'apprentissage (car aléatoires), c'est pourquoi le réseau cible doit être mis à jour périodiquement avec les valeurs du réseau principal. C'est ce que la valeur du paramètre target_update représente, en nombre d'épisodes.

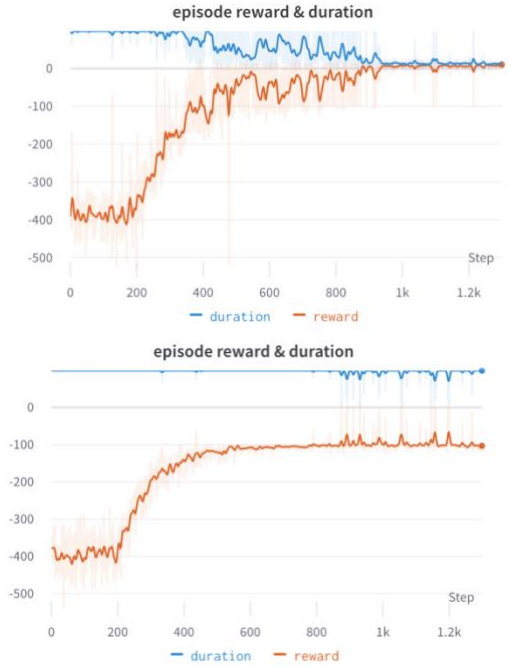
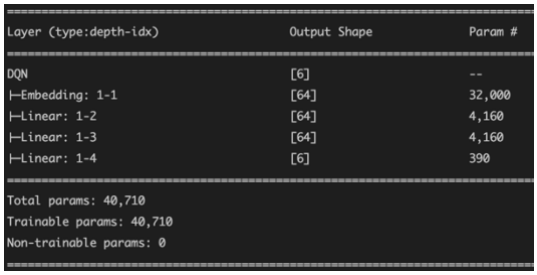


Figure 10 : de haut en bas, résultat d'un entrainement avec une mise à jour du target network à chaque épisode, et avec une mise à jour tous les 100 épisodes joués. On peut constater l'instabilité de l'entrainement lorsque la mise à jour du target network est trop fréquente, cela correspond approximativement à un entraînement utilisant un unique réseau de neurones. Si l'entrainement est instable, cela n'empêche pas pour autant l'IA de converger et de résoudre l'environnement, alors que au contraire si la mise à jour n'est pas assez fréquente les valeurs cible de Q étant trop fausser ne permettrons pas à l'IA de converger.

Note sur le model

Pour notre implémentation du DQN nous avons choisi une structure simple pour nos réseaux de neurones qui consiste en 4 couches. La première couche dites d'« Embedding » transforme l'entier d'entrée (état de l'environnement, compris entre 0 et 500) en un vecteur à 4 coordonnées. Elle est suivie par 2 couches dense cachées utilisant la fonction d'activation ReLu, pour finir sur une dernière couche dense à 6 sorties, correspondant aux 6 actions de l'environnement. Pour les couches cachées, à la suite de tests, nous avons choisi de leurs attribuer 64 neurones en entrées et en sortie.



Layer (type:depth-idx)	Output Shape	Param #
DQN	[6]	--
├─Embedding: 1-1	[64]	32,000
├─Linear: 1-2	[64]	4,160
├─Linear: 1-3	[64]	4,160
├─Linear: 1-4	[6]	390

Total params: 40,710
Trainable params: 40,710
Non-trainable params: 0

Figure 11 : architecture du modèle

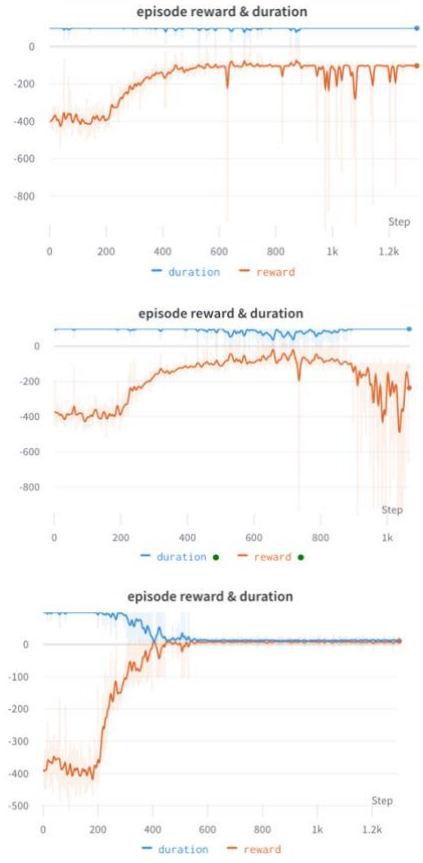


Figure 12 : de haut en bas, résultat d'un entraînement avec : des couches cachées composées de 10 entrées et 10 sorties, composées de 400 entrées et sorties et finalement avec des couches cachées de 64 entrées et sorties.

Optimisation des hyperparamètres

Dans le but d'accélérer le temps d'entrainement d'un agent tout en maximisant ses performances finales, nous avons cherché à optimiser les différentes valeurs des hyperparamètres cité précédemment. Pour nous aider dans cette tâche fastidieuse, nous avons utilisé la plateforme Weight&Biases et leur outil « Sweep ». Le Sweep permet, en définissant une plage de valeurs pour les différent hyperparamètres, de lancer un nombre donné d'entraînements utilisant une configuration aléatoirement piochée dans ces plages. Une fois tous les entraînements finis, dans notre cas 50, le logiciel est capable de dresser un tableau donnant une approximation de l'importance du paramètre dans l'obtention d'un meilleur résultat ainsi que la corrélation de sa valeur, plus la corrélation est élevée plus le paramètre doit avoir une valeur élevée et vis-versa (Figure 13).

À la suite du Sweep et de nouveaux tests avec les résultats obtenu, nous sommes arrivés à la conclusion que les valeurs les plus optimisées pour le DQN sur ces environnements sont les suivantes :

Hyperparamètre	Valeur optimale
Fréquence mise à jour du target network	5
Learning rate (départ)	0.01
Learning rate minimun	0.001
Learning rate decay (épisodes)	50
gamma	0.99
Loss function	mse
Taille de la mémoire	50 000
Taille des lots	128
Epsilon (départ)	1
Epsilon minimum	0.01
Espilon decay (épisodes)	350

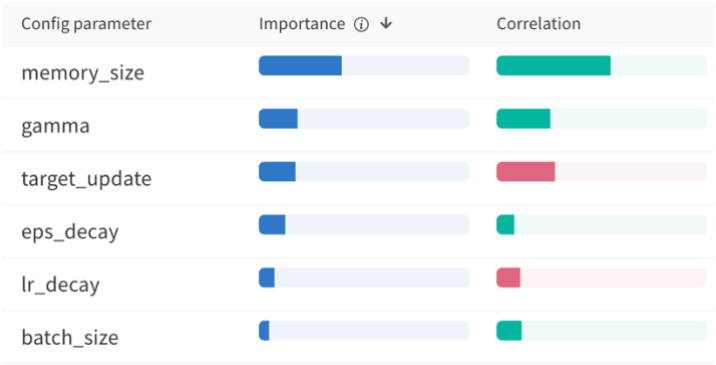


Figure 13 : Approximation de l'importance et corrélation des hyperparamètres d'après Weight&Biases. On peut y interpréter qu'il est important d'avoir une mémoire très élevée. Qu'une valeur élevée de gamma et de la taille des lots est aussi importante. Alors qu'une valeur faible pour la mise à jour du target network et de la décrémentation du learning rate est conseillée.

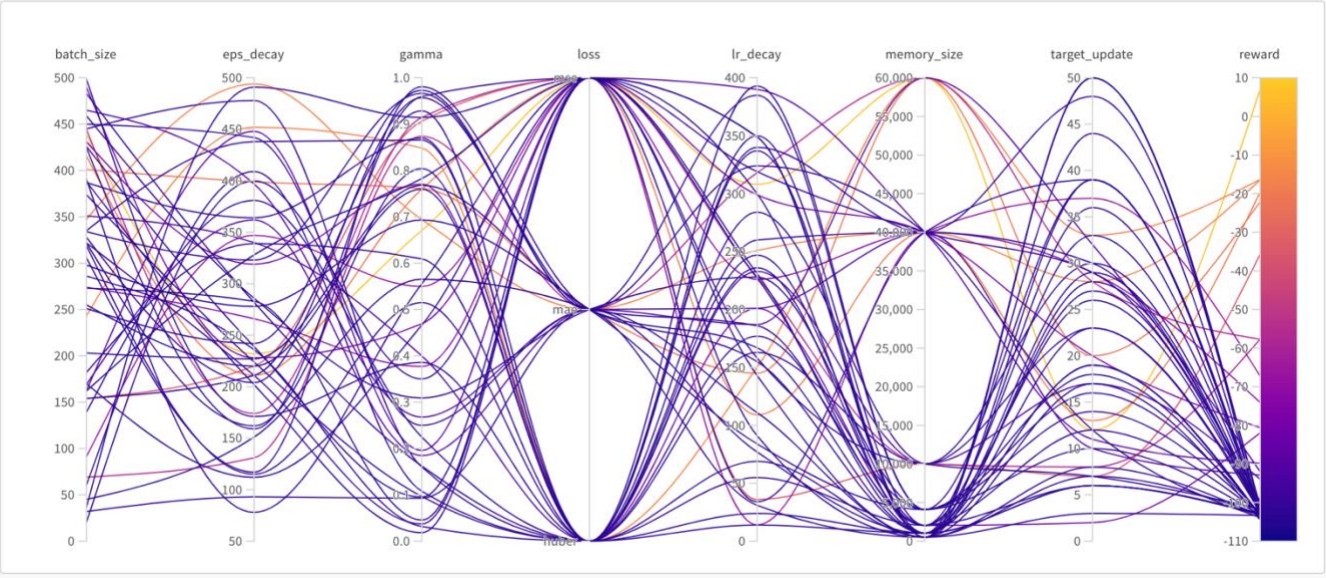


Figure 14 : résumé des 50 entraînements effectués par le Sweep. Chaque courbe représente un entrainement avec ses valeurs d'hyperparamètres, aléatoirement sélectionnés parmi les différentes plages de valeurs, et finissant sur le score moyen obtenu après 1300 épisodes.

Avec les paramètres précédemment définis, il faut approximativement 200 épisodes (les 200 premiers épisodes non affichés sur le graphique sont joués de façon aléatoire et servent à remplir la mémoire de données, aucune mise à jour des poids des réseaux de neurones n'est faite durant cette périodes) à l'agent pour résoudre l'environnement de manière constante et 200 épisodes de plus pour finaliser son apprentissage et avoir la meilleure approximation des valeurs de Q. Un total de 400 épisodes suffit donc à notre DQN pour converger et finir son apprentissage alors qu'il en fallait approximativement 2500 au Q-Learning.

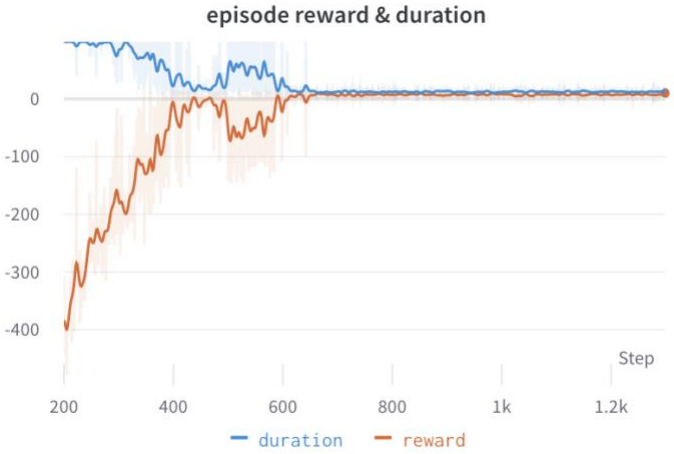


Figure 15 : courbe d'apprentissage du DQN avec les hyperparamètres optimisés.

Évaluation

Une fois de plus, pour évaluer notre algorithme nous l'avons fait jouer à 10 000 parties. Nous obtenons un score moyen de 8 pour 12.9 étapes et 0 pénalités, soit 1 points de score moyen supérieur au Q-Learning pour 1 étapes moyenne en moins. De plus, on remarque que notre IA n'engendre plus aucunes pénalités en exécutant des actions interdites là où le Q-Learning avez encore tendance à en commettre. On peut donc en conclure que notre première optimisation est un succés.

	score	steps	penalities
mean	8.025	12.975	0.0
max	15.000	18.000	0.0
min	3.000	6.000	0.0

Figure 16 : Moyenne, maximum et minimum des scores, étapes et pénalités obtenus par l'agent lors de ses 10 000 parties.

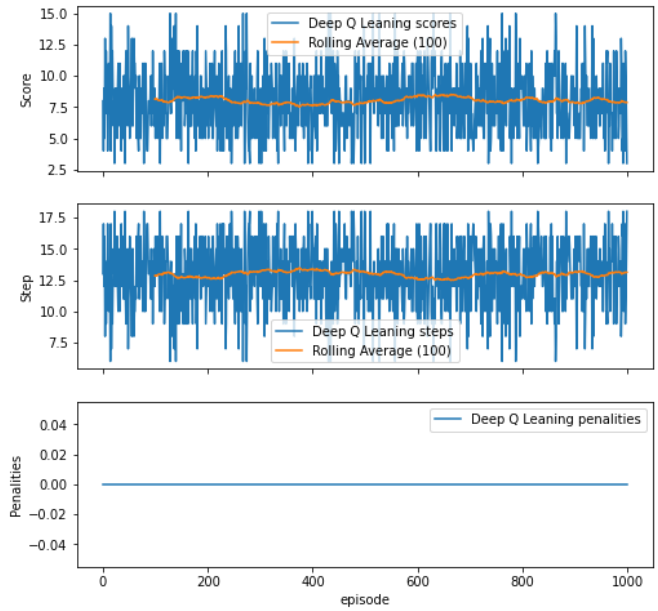


Figure 17 : résultat obtenu par l'agent durant les 10 000 parties d'évaluation (axe x).

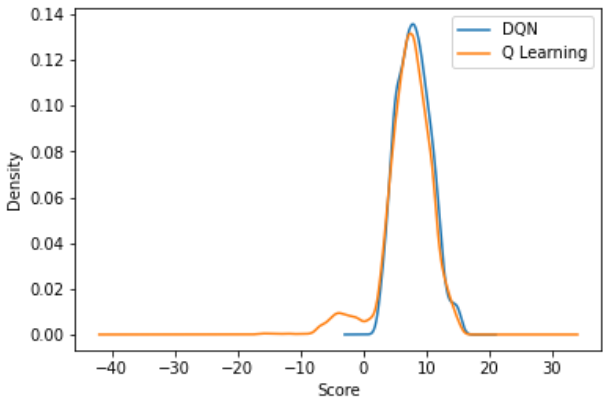


Figure 18 : distribution des scores obtenus par le Q-Learning et le DQN sur 10 000 parties joués. On peut constater que le DQN obtiens une légèrement plus importante pour les scores positifs et presque aucun score négatif, montrant ainsi que le DQN tend à résoudre le jeu de manière plus fréquentes et avec de meilleurs scores que le Q-Learning.

Double Deep Q-Learning

Le DQN est connue pour avoir tendance à surestimer les valeurs de Q, qui dans certains cas peut fausser le choix de la meilleure action à prendre dans un état donné. Une optimisation simple mais efficace au DQN ce nomme le « Double Deep Q-Learning » (DDQN) et a pour objectif de réduire le bruit présent dans le calcul des valeurs de Q. Le DDQN se base sur le postula suivant : par exemple prenons une balance, lorsque l'on pèse une personne celle-ci nous donne son poids avec une

marge d'erreur de plus ou moins 1 kg. Si après d'avoir pesé 100 personnes on choisit la personne avec le poids le plus élevé Q, alors il y a des chances que ce ne soit pas réellement la personne la plus lourde à cause de marge d'erreur de la balance. Pour corriger cette marge d'erreur on pèse à nouveau la personne avec une balance similaire et on garde cette valeur de Q, qui aura probablement été corrigé de sa variance. Pour reproduire ce comportement, le DDQN calcule une première fois les valeurs de Q en utilisant le réseau de neurones principale, sélectionne la valeur de Q maximal, et re calcul cette valeur à l'aide du target network. Et c'est avec cette seconde valeur de Q que la loss sera calculer et que les poids du réseau de neurones principale seront mis à jour.

Les modifications du code permettant de transformer le DQN en DDQN étant minime, nous avons décidé d'ajouter un paramètre « ddqn » à notre class DQN, qui, si égale à « true », utilisera la manière de calculer les valeurs de Q du DDQN.

Optimisation des hyperparamètres

Une nouvelles fois pour obtenir les valeurs optimales des hyperparamètres pour le DDQN nous avons utilisé un Sweep. Les résultats sont sensiblement semblables que pour le DQN a l'exception que la fréquence de mise à jour du target network doit être légèrement réduite et la taille des lots légèrement augmenter.

Voici les valeurs optimales que l'on a déterminé pour le DDQN :

Hyperparamètre	Valeur optimale
Fréquence mise à jour du target network	4
Learning rate (départ)	0.01
Learning rate minimun	0.001
Learning rate decay (épisodes)	50
gamma	0.99
Loss function	mse
Taille de la mémoire	50 000
Taille des lots	252
Epsilon (départ)	1
Epsilon minimum	0.01
Espilon decay (épisodes)	350

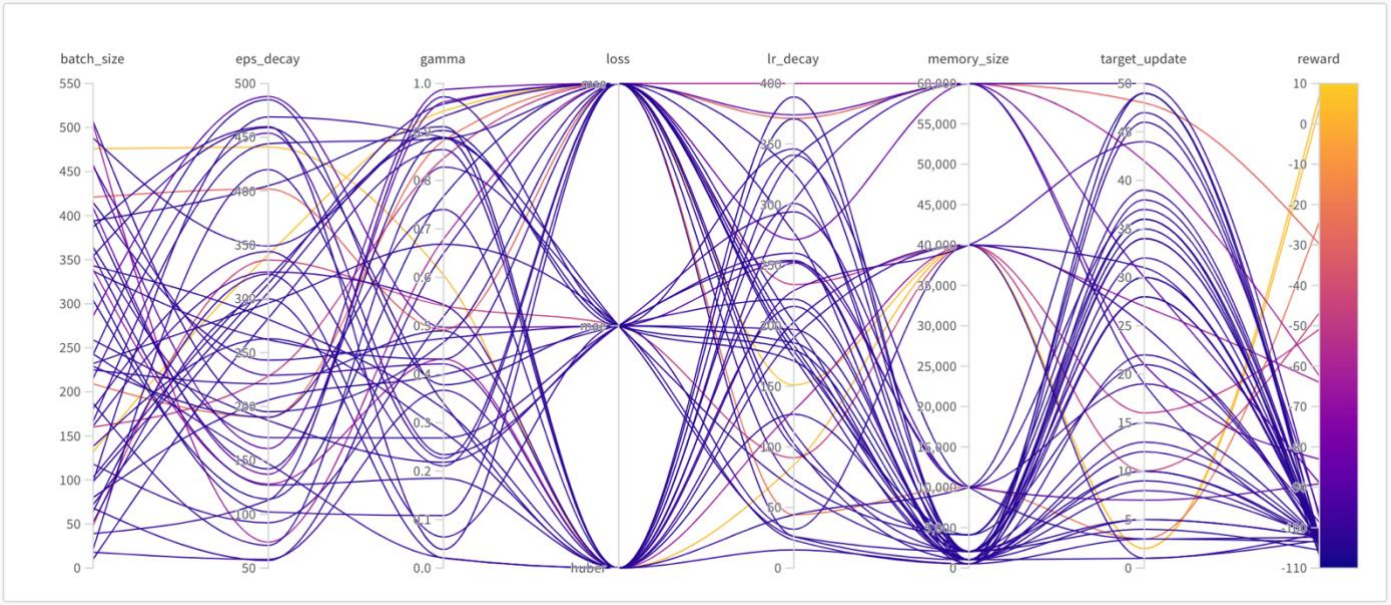


Figure 19 : résultats du Sweep pour le DDQN

Évaluation

Le DQN ayant déjà les scores maximums pour la résolution de l'environnement et sans commettre d'erreur, comparer les résultats du DDQN et du DQN après 10 000 parties d'évaluation ne montre pas de résultat probant.

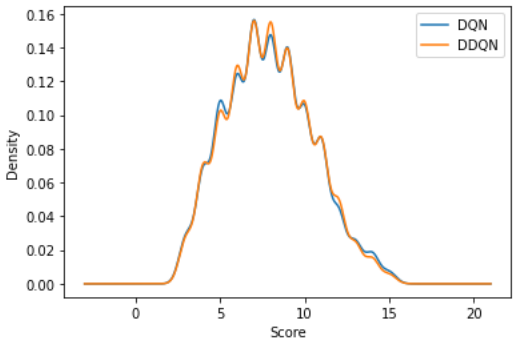


Figure 20 : distribution des scores obtenus par le DQN et DDQN lors de 10 000 parties d'évaluation jouées.

Néanmoins, il est intéressant de comparer leurs courbes d'apprentissages (Figure 22). Effectivement en les entraînent chacun sur le même nombre d'épisodes et utilisant leurs valeurs d'hyperparamètres les plus optimal, nous pouvons constater des différences. Alors qu'il faut au DDQN environ 250 épisodes afin d'estimer les bonnes valeurs de Q et de stabiliser son score au score maximal de l'environnement, le DQN a un apprentissage légèrement plus instable avec en moyenne un besoin de 100 épisodes supplémentaires pour stabiliser ses valeurs de Q. On peut donc en conclure que même si les résultats finaux à la suite de l'apprentissage sont les mêmes pour les deux algorithmes sur cet environnement, la vitesse accélérées et la meilleure stabilité d'apprentissage du DDQN en font bien une amélioration du DQN.

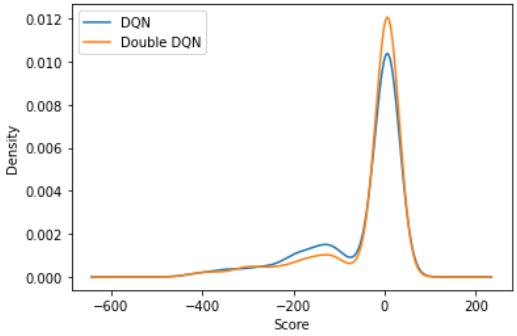


Figure 21 : distribution des scores obtenus par le DQN et DDQN lors leurs 1500 parties d'entrainement. On constate que le DDQN tend à gagner plus fréquemment et avec des scores plus élevés.

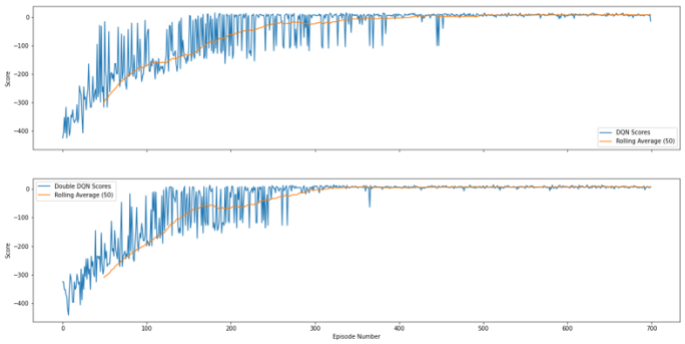


Figure 22 : courbe d'apprentissage du DQN (haut) et DDQN (bas) sur 700 épisodes (axe x).

Proximal Policy Optimization

L'algorithme « Proximal Policy Optimization » (PPO) a été introduit par l'équipe OpenAI en 2017 et est rapidement devenu l'une des méthodes d'apprentissage par renforcement les plus populaires. Il fonctionne en collectent un petit lot d'expériences en interagissant avec l'environnement et utilise ce lot pour mettre à jour sa politique de prise de décision. Une fois la stratégie mise à jour avec ce lot, les expériences sont supprimées et un nouveau lot est collecté avec la stratégie récemment mise à jour. C'est la raison pour laquelle il s'agit d'une approche « On-Policy » où les échantillons d'expérience collectés ne sont utiles qu'une seule fois pour mettre à jour la politique actuelle, contrairement au DQN et DDQN vue précédemment qui été donc « Off-Policy ». Le principal objectif du PPO est de s'assurer qu'une nouvelle mise à jour de la politique de prise de décision ne la modifie pas trop par rapport à la politique précédente (algorithme du gradient). Cela conduit à moins de variance durant l'entrainement au prix néanmoins d'un certain biais, mais assure un entrainement plus fluide et garantit également que l'agent ne s'engage pas sur une voie irrécupérable consistant à prendre des actions insensées.

Nous avons donc essayé d'implémenter notre version du PPO afin de résoudre l'environnement Taxi-v3, mais malheureusement il semble que dû à un manque d'exploration ou / et de la manière dont les récompenses et pénalités sont distribuées dans le jeu, notre agent ne parvient pas à résoudre l'environnement. Il apprend néanmoins très rapidement à ne plus faire d'actions illégales, maximisant ainsi son score à -100.

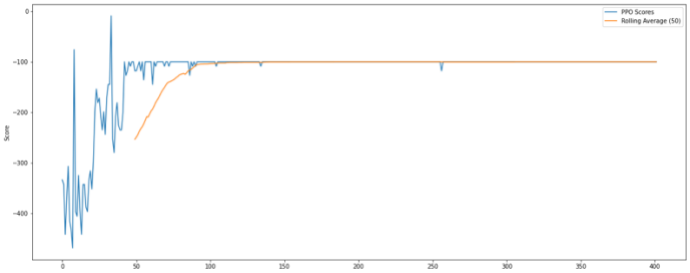


Figure 22 : courbe d'apprentissage du PPO. On constate qu'il faut un peu moins d'une 100 d'épisodes à l'agent pour stabiliser son score et ne plus faire d'action illégale même s'il ne finit pas le jeu pour autant.

Conclusion

Pour conclure, l'algorithme d'apprentissage épisodique optimisé que nous avons réussis à implémenter et qui a obtenu les meilleurs résultats est le DDQN. Avec environ le besoin de 300 parties d'entrainement pour résoudre le jeu de la manière la plus optimale, soit un score moyen de 8 pour 12.9 étapes et 0 pénalités.

Références

Jason Brownlee, What is gradient in machin learning ? 14 Avril 2021. *https://machinelearningmastery.com/gradient-in-machine-learning/*

Wanshun Wong, What is gradient Clipping ? 3 Mars 2020. *https://towardsdatascience.com/what-is-gradient-clipping-b8e815cdfb48*

Shreyas Gite, Getting started with Q-Learning, 4 Avril 2017. *https://towardsdatascience.com/practical-reinforcement-learning-02-getting-started-with-q-learning-582f63e4acd9*

George Seif, The 3 most common loss function for Machine Leaning Regression, 21 Mais 2019. https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3

Jeremy Jordan, Setting the learning rate of your neural network, 1 Mars 2018. *https://www.jeremyjordan.me/nn-learning-rate/*

Kowshik Chilamkurthy, Off-Policy vs On-policy Reinforcement Learning, 7 Novembre 2020. https://kowshikchilamkurthy.medium.com/off-policy-vs-on-policy-vs-offline-reinforcement-learning-demystified-f7f87e275b48

David R. Pugh, Improving the DQN algorithm using Double Q-Learning, 11 Avril 2020. https://davidrpugh.github.io/stochastic-expatriate-descent/pytorch/deep-reinforcement-learning/deep-q-networks/2020/04/11/double-dqn.html

Vikash Kumar Das, Difference between Q and Deep Q-Learning, 19 Octobre 2020. https://www.globaltechcouncil.org/reinforcement-learning/reinforcement-learning-difference-between-q-and-deep-q-learning/