Étude des performances des algorithmes DQN et DDPG

Forest Thomas and **Patterson Jérémie**
Polytechnique Montréal
{thomas-2.forest, jeremie.patterson}@polymtl.ca

Abstract

Dans ce rapport, nous étudions et comparons les performances des algorithmes de Deep Reinforcement Learning DQN (Deep Q-Network) et DDPG (Deep Deterministic Policy Gradient) dans différents environnements Gymnasium, en fonction de la nature discrète ou continue de l'espace d'actions. Nous présentons une synthèse du fonctionnement interne de chacun de ces algorithmes, notamment l'utilisation du replay buffer, des réseaux cibles et des stratégies d'exploration adaptées. Des expérimentations sur les environnements *MountainCar*, *LunarLander* et *CarRacing* mettent en évidence les forces et limites de chaque approche : DQN se montre plus stable et rapide à entraîner sur des espaces d'actions discrets, tandis que DDPG est mieux adapté aux environnements à actions continues mais se révèle plus sensible aux choix d'hyperparamètres. Nos résultats confirment l'importance de choisir l'algorithme en fonction de la structure de l'espace d'actions et soulignent le potentiel de DDPG pour les tâches complexes nécessitant un contrôle continu.

1 Introduction

L'apprentissage profond en apprentissage par renforcement a connu une avancée majeure en 2013 avec l'introduction des Deep Q-Networks [1] (DQN), permettant ainsi de résoudre de nombreux jeux Atari à espace d'actions discret. Néanmoins, ces environnements de jeux restent relativement simples et peu représentatifs de la complexité des applications réelles. Les DQN reposent sur l'estimation de la fonction Q, qui associe à chaque paire (état, action) une valeur représentant la récompense future attendue. Pour estimer la politique optimale, le réseau doit alors itérer sur toutes les actions possibles, ce qui suppose que le nombre d'actions soit connu et restreint. Cependant, des environnements plus complexes ne possèdent pas cette caractéristique et ont des espaces d'actions continus, impossibles à estimer efficacement avec un tel réseau. Cette limitation ouvre la voie à des recherches sur des méthodes plus adaptées aux environnements avec espace d'action continu. Ainsi, l'algorithme Deep Deterministic Policy Gradient [2] (DDPG), paru en 2015, vient proposer une solution à ce problème. DDPG propose une approche Actor-Critic permettant de donner tout de suite la meilleure action au lieu d'itérer sur les actions possibles. Cela est possible grâce au réseau Actor, qui est entraîné pour retourner une seule action, et au modèle Critic, qui permet de guider l'amélioration de celle-ci à l'entraînement pour évaluer à quel point l'action proposée est bonne.

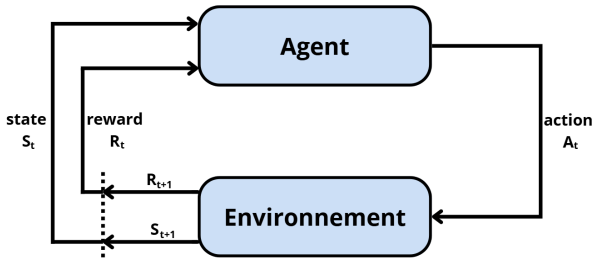

Figure 1: Boucle agent-environnement en apprentissage par renforcement

Dans ce papier, nous analyserons et comparerons ces deux algorithmes d'apprentissage par renforcement sur des environnements de type gymnasium [3] adaptés aux spécificités de chaque algorithme. Ainsi, l'algorithme DQN sera testé sur des environnements avec actions discrètes. L'algorithme DDPG sera testé plutôt sur des environnements à base d'actions continues. Afin de pouvoir les comparer convenablement, nous regarderons principalement ceux qui supportent les deux types d'actions comme MountainCar, LunarLander et CarRacing.

2 Background

Considérons un problème classique d'apprentissage par renforcement dans un environnement $\mathcal{E}$, modélisé comme un processus de décision de Markov (MDP) entièrement observable. À chaque *step* t, un agent se trouve dans un état $s_t \in \mathcal{S}$, choisit une action $a_t \in \mathcal{A}$ selon une politique $\pi(a|s)$, reçoit une récompense $r_{t+1} \in \mathbb{R}$, et l'environnement transite vers un nouvel état s_{t+1} selon une dynamique probabiliste $p(s_{t+1}|s_t, a_t)$ tel qu'illustré en Figure 1.

L'objectif de l'agent est de maximiser la récompense future

attendue à partir de t :

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

avec $\gamma \in [0, 1]$ un facteur donnant plus ou moins d'importance aux récompenses futures.

Les deux algorithmes ici étudiés utilisent la fonction action-valeur (fonction Q) qui représente la valeur maximale de la récompense que l'agent pourrait attendre en suivant une certaine politique π après avoir pris une action a dans un état s :

$$Q^\pi(s_t, a_t) = \mathbb{E}_\pi[G_t | s_t = s, a_t = a]$$

Comme de nombreux problèmes en apprentissage par renforcement, l'estimation de cette fonction repose sur l'équation de Bellman :

$$Q^\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r(s_t, a_t) + \gamma \, \mathbb{E}_{a_{t+1} \sim \pi}[Q^\pi(s', a')]]$$

Les méthodes dites *value-based*, comme DQN [1], s'appuient directement sur cette équation pour approximer la fonction optimale $Q^*(s, a) = \max_\pi Q^\pi(s, a)$ à l'aide de réseaux de neurones, ici paramétrés par θ. L'apprentissage se fait en minimisant l'erreur quadratique entre la sortie du réseau et une cible :

$$y_t = r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta),$$

$$L(\theta) = \mathbb{E}\left[\left(y_t - Q(s_t, a_t; \theta) \right)^2 \right],$$

où θ sont les paramètres du réseau.

DDPG utilise une approche dite *policy-based*, utilisant un acteur et un critique. L'acteur permet d'approximer une politique déterministe $\mu(s)$ qui maximise la fonction Q : $\mu(s) = \arg\max_a Q(s, a)$. Le critique approxime la fonction $Q(s, a)$ elle-même. Notons alors θ^Q les paramètres de cette fonction d'approximation. Similairement à l'approche *value-based* de DQN, on a alors:

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1}); \theta^Q),$$

$$L(\theta^Q) = \mathbb{E}\left[\left(y_t - Q(s_t, a_t; \theta_t) \right)^2 \right].$$

3 Algorithmes

es changements majeurs apportés par l'algorithme DQN tels que l'utilisation d'un *replay buffer* ainsi qu'un *réseau cible* permettant de stabiliser l'apprentissage en calculant la cible y_t ont été réutilisés pour DDPG.
Le replay buffer permet de pallier le fait que les expériences récupérées séquentiellement dans un environnement par un agent ne sont pas indépendantes. Ainsi, on sauvegarde les états et les transitions à chaque étape dans un buffer de taille finie. On utilise ces transitions sous forme de batch choisi uniformément pour mettre à jour les réseaux d'apprentissage. Les réseaux cibles sont initialisés avec les mêmes paramètres que les réseaux d'apprentissage respectifs. La mise à jour de ces réseaux cibles se fait ensuite de manière progressive en suivant l'équation: $\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$, où $\tau \ll 1$.

De par la nature différente des espaces d'actions gérés par les deux algorithmes, la manière d'explorer l'environnement n'est pas la même. En effet, pour DQN, l'utilisation d'un algorithme ϵ-greedy choisissant la meilleure action selon $Q(s, a)$ avec une probabilité $1-\epsilon$ ou une action aléatoire avec une probabilité ϵ est suffisante. Cet algorithme d'exploration permet, pour un espace d'actions discret, de garantir que l'exploration de l'environnement soit suffisante, puisque toutes les actions finiront par être sélectionnées. Dans le cas d'un espace d'actions continu (DDPG), le nombre d'actions est infini, l'échantillonnage d'une action aléatoire n'est alors ni efficace ni significatif et l'approche ϵ-greedy n'est plus valide. Il faut explorer de manière structurée et locale autour de la politique actuelle $\mu(s)$. Pour cela, DDPG ajoute un bruit à la sortie de la politique déterministe tel que :

$$a_t = \mu(s_t | \theta^\mu) + \mathcal{N}_t,$$

où $\mathcal{N}_t$ est un bruit temporellement corrélé tel que le processus d'Ornstein-Uhlenbeck [5].

Le papier présentant DQN [1] s'intéresse particulièrement aux environnements de type Atari qui sont des environnements à haute dimension. Dans ces environnements, l'observation à chaque étape est la matrice de pixels constituant l'image. De manière à ce que notre réseau puisse interpréter ces images, il est nécessaire d'adopter une architecture reposant en partie sur un réseau convolutionnel. Ces environnements restent néanmoins assez complexes à l'apprentissage étant donné notre puissance de calcul. Ainsi, nous avons créé deux versions pour notre réseau DQN : une relativement simple ne nécessitant qu'un MLP de quelques couches cachées pour les environnements à faible dimension et une version que l'on appellera DQN-CNN pour les environnements ayant une image comme observation. Nous avons suivi le même principe pour l'algorithme DDPG, qui comporte donc 2 types d'acteurs et 2 types de critics afin de supporter les différents types d'environnements. À noter que pour les environnements à haute dimension, un pré-traitement des images est nécessaire pour simplifier l'apprentissage des réseaux. Ce pré-traitement consiste notamment à redimensionner l'image et à la transformer en niveau de gris pour supprimer une dimension. De plus, pour ces environnements, une image n'est souvent pas suffisante pour capter la dynamique à un certain instant (direction d'un objet, vélocité...). Pour pallier ce problème, une convention souvent utilisée est d'empiler les 4 dernières frames pour en faire l'input courant du réseau.

4 Résultats

Nos expérimentations ont montré que l'algorithme DDPG était plus sensible aux hyperparamètres comparé à DQN. Nous avons eu besoin de plusieurs essais pour arriver à trouver les valeurs des hyperparamètres qui guident au mieux le modèle vers une récompense maximale. L'algorithme DQN s'exécutait aussi plus rapidement ce qui facilitait les recherches d'hyperparamètres. D'ailleurs, comme mentionné plus tôt les deux approches ont été testées dans les mêmes environnements, mais nous avons aussi fait en sorte qu'elles soient testées sur un même nombre d'étapes pour que les

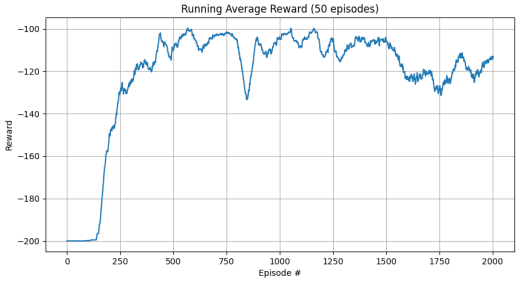

Figure 2: Évolution de la récompense obtenue avec l'algorithme DQN pour l'environnement MountainCar

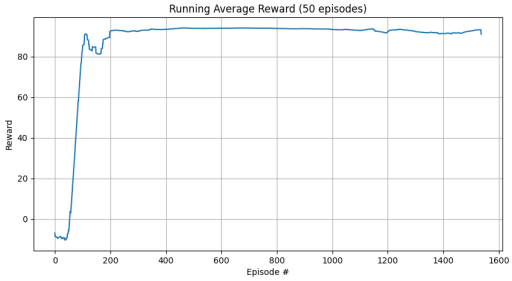

Figure 3: Évolution de la récompense obtenue avec l'algorithme DDPG pour l'environnement MountainCar

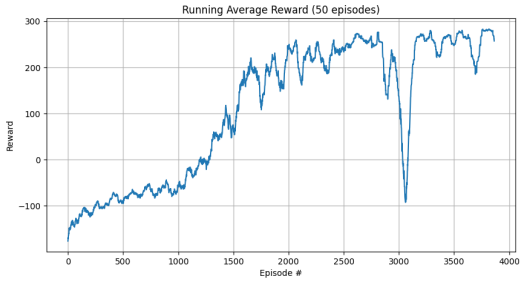

Figure 4: Évolution de la récompense obtenue avec l'algorithme DQN pour l'environnement LunarLander

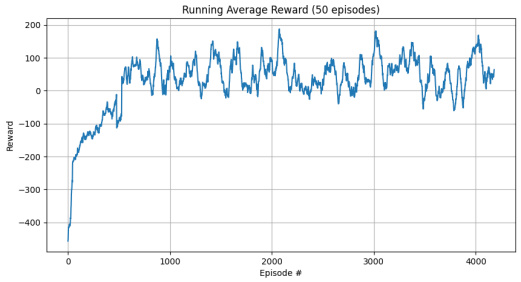

Figure 5: Évolution de la récompense obtenue avec l'algorithme DDPG pour l'environnement LunarLander

comparaisons soient pertinentes.

Dans le cas de MountainCar, il est possible de voir dans les Figures 2 et 3 que pour DDPG, un plus petit nombre d'épisodes résulte de ce nombre d'étapes fixes. Il faut savoir qu'un épisode est une série d'actions jusqu'à ce que la mission de l'environnement soit complétée, que le modèle ait produit une action illégale ou alors que le temps alloué soit écoulé. La façon d'attribuer les récompenses varie entre l'environnement discret ou continu alors cela explique le comportement différent. Pour l'environnement continu avec DDPG, il y a des pénalités pour les actions pour les actions avec une force trop importante d'où la difficulté à atteindre le sommet parfois. À l'inverse, pour DQN, c'est plus facile puisque la seule pénalité est le temps écoulé (avec un temps maximum plus bas). Bref, pour cette raison il est difficile de comprarer les algorithmes sur l'environnement MountainCar, mais cela montre que la modélisation de la fonction de récompense joue un rôle très important sur l'apprentissage du modèle.

Dans le cas de LunarLander, l'attribution de récompense est la même pour les deux algorithmes parce que l'environnement supporte directement les deux types d'actions. C'est donc possible de bien comparer avec les Figures 4 et 5 qui montrent l'évolution de la récompense obtenue pour chacune des approche. Il est possible de voir que DDPG converge plus rapidement vers de meilleurs résultats en dépassant 100 points avant 800 épisodes alors que DQN y arrive vers 1500 épisodes. Par contre, DQN finit par arriver à une meilleure récompense maximale. Cela pourrait s'expliquer par le fait que l'exploration en dehors de la politique est limité quand DDPG arrive à obtenir de bonnes valeurs de récompense. Cela expliquerait aussi pourquoi DDPG reste aussi plus stable contrairement à DQN qui rencontre une grosse baisse vers 3000 épisodes. Il faut aussi noter qu'avec la même configuration de récompense, le million d'étapes produit un nombre similaire d'épisodes pour les deux.

Dans le cas de RacingCar, le comportement de la courbe pour l'algorithme DQN ressemble beaucoup à celui sur l'environnement LunarLander, mais nous avons été obligés de réduire le nombre d'étapes dans l'analyse à cause de la puissance de calcul limité à notre disposion. C'est pourquoi la Figure 6 ne montre pas de stabilisation même si on note une claire augmentation de la récompense. Il faut mentionner que cet environnement est plus complexe et que le traitement d'images nécessaire par les modèles dans celui-ci est couteux en temps. C'est le même constat pour DDPG, mais ça devenait vraiment compliqué d'avoir des résultats intéressants à comparer étant donné qu'il y a deux fois plus de réseaux à entraîner (actor et critic). Nous avons quand même réussi à exécuter une vingtaine d'épisodes qui ont confirmé que notre implémentation était fonctionnelle.

Pour obtenir de meilleurs résultats efficacement, nous aurions pu mieux organiser la recherche d'hyperparamètres en sauvegardant les résultats avec un service en ligne comme Weight and Biases ou en organisant mieux notre architecture

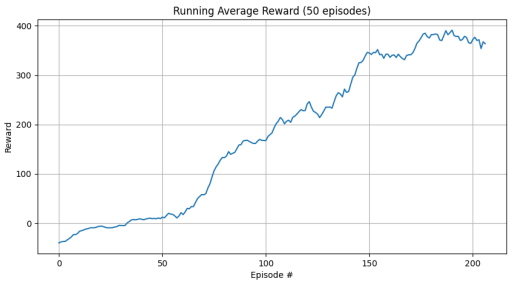

Figure 6: Évolution de la récompense obtenue avec l'algorithme DQN pour l'environnement CarRacing

afin de séparer la gestion des hyperparamètres des différents environnements.

5 Conclusion

Dans ce projet, nous avons constaté l'importance que joue la valeur des hyperparamètres et la définition de la fonction de récompense sur l'apprentissage par renforcement d'un modèle. Nous avons remarqué que selon l'environnement (ou la tâche à accomplir) les bonnes valeurs pour les hyperparamètres pouvaient drastiquement changer. Nous avons aussi noté que l'algorithme DQN prenait moins de temps d'exécution, mais que lorsque la puissance de calcul était suffisante, DDPG converge plus rapidement vers des bonnes performances. L'apprentissage par renforcement nous semble vraiment intéressant pour faire en sorte qu'un modèle apprennent les lois physiques d'un environnement sans explicitement lui indiquer quelle action entreprendre. L'algorithme DDPG semble être une réelle amélioration de l'algorithme DQN puisqu'il permet d'effectuer l'apprentissage sur des espaces d'actions continues et donc gérer des environnements plus complexes.

References

[1] Mnih, V. et al. (2015). *Human-level control through deep reinforcement learning*. arXiv:1312.5602.

[2] Lillicrap, T. P. et al. (2015). *Continuous control with deep reinforcement learning*. arXiv:1509.02971.

[3] Towers, M. et al. (2024). *Gymnasium: A Standard Interface for Reinforcement Learning Environments*. arXiv:2407.17032.

[4] Sutton, R. S., Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT press.

[5] Uhlenbeck, G. E., Ornstein, L. S. (1930). *On the theory of the Brownian motion*. Physical Review, 36(5):823.

Annexe - Détails des expériences

Dans cette section, nous détaillons les expériences réalisées. Nous avons pu comparer les algorithmes sur trois environnements différents, *MountainCar*, *LunarLander* ainsi que *CarRacing*, tous les trois issus de la librairie gymnasium [3]. Les deux premiers sont en faible dimension alors que le dernier se base sur des images en tant qu'observations. Le cas du *MountainCar* est un peu particulier car les versions discrètes et continues de l'environnement ne possèdent pas la même fonction de récompense, donc leur comparaison n'est pas vraiment pertinente. Pour chacune des autres expériences, nous avons utilisé des paramètres similaires. L'environnement LunarLander a été exécuté pour 10^6 steps alors que les deux autres 250k steps. Pour les deux algorithmes DQN, les hyperparamètres étaient les mêmes : $batch\_size = 64, \gamma = 0.99, lr = 1e^{-4}, \tau = 0.005$. Son fonctionnement est détaillé dans l'Algorithme 1. Pour l'algorithme DDPG de l'environnement *LunarLander*, les hyperparamètres utilisés étaient également les mêmes, excepté le système d'exploration basé sur un processus d'Ornstein-Uhlenbeck avec $\sigma = 0.2$, ainsi que le taux d'update qui a été réduit à 0.001 pour une meilleure stabilité. Enfin, pour l'algorithme DDPG sur le *CarRacing* (images), les hyperparamètres étaient très compliqués à fine-tuner et l'apprentissage était très sensible. En effet, contrairement à la version discrète où l'agent ne prend qu'une seule action par étape, dans sa version continue, l'agent prend une série d'actions à valeurs continues à chaque étape (degré de volant, frein, accélérateur). Or, une mauvaise combinaison de ces 3 actions peut très vite faire dégénérer son comportement. Ainsi, pour cet environnement, nous avons décidé d'augmenter le *batch size* à 256 et de réduire le taux d'apprentissage des réseaux à $1e^{-5}$. Comme expliqué dans les sections précédentes, nous avons également du réduire le nombre de steps pour cet environnement. Le fonctionnement de l'algorithme DDPG est détaillé dans l'Algorithme 2.

Algorithm 1 Deep Q-learning algorithm

Initialize replay buffer $\mathcal{D}$ de taile N
Initialize Q with random weights θ^Q
Initialize target network Q' with weights θ^Q
for episode = 1 to M **do**
    Receive initial state $s_1 = \{x_1\}$
    if observation type is an image **then**
        Preprocess state s_1
    end if
    for $t = 1$ to T **do**
        With probability ε select a random action a_t
        Otherwise select $a_t = \arg\max_a Q^*(s_t, a; \theta)$
        Execute action a_t and observe reward r_t and new state s_{t+1}
        if observation type is an image **then**
            Preprocess state s_t
        end if
        Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
        Set $s_{t+1} = s_t$
        Sample random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{D}$
        Set $y_i = r_i + \gamma \max_{a'} Q'(s_{i+1}, a'|\theta^{Q'})$
        Update the network by minimizing the loss $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the target network: $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
    end for
end for

Algorithm 2 Deep Deterministic Policy Gradient algorithm

Initialize replay buffer $\mathcal{D}$ de taile N
Initialize critic Q and actor μ with random weights θ^Q and θ^μ
Initialize target networks Q' and μ' with weights θ^Q and θ^μ
for episode = 1 to M **do**
    Intialize a random process $\mathcal{N}$ for action exploration
    Receive initial state $s_1 = \{x_1\}$
    if observation type is an image **then**
        Preprocess state s_1
    end if
    for $t = 1$ to T **do**
        Select an action a_t using the actor network μ
        Add noise $\mathcal{N}_t$ to the selected action a_t
        Execute action a_t and observe reward r_t and new state s_{t+1}
        if observation type is an image **then**
            Preprocess state s_t
        end if
        Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
        Set $s_{t+1} = s_t$
        Sample random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from $\mathcal{D}$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update the critic network by minimizing the loss $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor network by using the sampled policy gradient
        Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

    end for
end for