Deep Deterministic Policy Gradient

Sébastien Labine [1] , **William Sevigny**[1] , **Philippe Kavalec**[2]
[1]Génie Logiciel, Polytechnique Montréal
[2]Génie Informatique, Polytechnique Montréal
sebastien.labine@polymtl.ca, william.sevigny@polymtl.ca, philippe.kavalec@polymtl.ca

Abstract

Deep Deterministic Policy Gradient est un algorithme d'apprentissage par renforcement profond applicable à des environnements avec un espace d'action continue. Nous implémenterons cet algorithme dans l'objectif de résoudre un environnement OpenAI. Malgré nos résultats non concluants, nous présentons une excellente synthèse de l'algorithme et de la motivation derrière celui-ci.

1 Introduction

L'intelligence artificielle permet de trouver des solutions à des tâches complexes et difficiles à résoudre avec des algorithmes standards. Selon le problème, différents types d'algorithmes tirant profit de l'intelligence artificielle peuvent être utilisés afin d'optimiser la solution. L'apprentissage par renforcement est la technique permettant de résoudre des tâches demandant une interaction avec un environnement sur la base d'entrées sensorielles de haute dimension.

Popularisée en 2015 dans l'article *Human-level control through deep reinforcement learning*, l'apprentissage par renforcement profond est l'évolution de cette technique. Le *deep reinforcement learning* combine la puissance des réseaux de neurones aux techniques d'apprentissage par renforcement classique. Cette approche a repopularisé le domaine de l'apprentissage par renforcement alors qu'elle multiplie les possibilités d'application de la technologie. Elle est possible grâce à la puissance de computation qui nous est maintenant disponible.

Dans cet article, nous vous présenterons le *Deep Deterministic Policy Gradient* ou *DDPG* [1], un algorithme d'apprentissage par renforcement. L'algorithme DDPG se caractérise par l'application de l'apprentissage par renforcement sur des environnements avec un espace d'action continue. Nous commencerons par vous présenter les travaux antérieurs, puis nous présenterons l'approche théorique formant la base du sujet. Enfin, nous présenterons et discuterons de nos expériences et de nos résultats pour conclure avec une analyse critique de l'approche utilisée pour comprendre cet algorithme.

2 Travaux antérieurs

2.1 Q-Learning

L'algorithme DDPG se base initialement sur le *Q-Learning* où un agent vise à apprendre la fonction Q ou *Q- Function* pour trouver la politique optimale dans un environnement. Cet algorithme excelle dans des environnements limités ayant un espace d'états de faible dimension et où chaque état est défini par une petite quantité de paramètres. Lorsque confronté à un environnement avec une grande quantité d'états et de transitions (actions), il devient difficile pour un algorithme de *Q-learning* d'établir un lien entre sa situation actuelle dans l'environnement et le chemin vers une meilleure récompense.

2.2 Deep Reinforcement Learning

L'apprentissage par renforcement profond regroupe les algorithmes généralisant la technique de *Q learning* sur des espaces d'actions complexes. Ces algorithmes modélisent le lien entre la situation de l'agent dans l'environnement et l'action à poser grâce à des réseaux de neurones profonds. Cette technique est inspirée des mécanismes d'apprentissages retrouvés chez les humains et les animaux.

Deep Q-Learning

Un premier algorithme d'apprentissage par renforcement profond est le *Deep Q Learning* [2]. Le *Deep Q Learning* vise à approximer la fonction Q pour inférer la meilleure politique. Cet algorithme modélise la fonction Q de l'apprentissage par renforcement par un réseau de neurones profond appelé *Deep Q network* ou *DQN*. L'algorithme *DQN* est performant pour des espaces d'action discrets de par sa capacité à évaluer la valeur de chaque action. Cependant, l'algorithme se prête difficilement à dans des espaces d'action continue puisqu'il y a une infinité d'actions possibles à chaque état.

Policy Gradient

Un second algorithme est le *Policy Gradient* [3]. L'algorithme ici vise à apprendre la politique optimale à partir d'un échantillon d'expériences passées et un réseau de neurones. Le réseau de neurones profond est entraîné afin de prioriser les actions menant aux meilleures récompenses jusqu'à l'atteinte d'une convergence. Cette approche est priorisée lorsqu'il est difficile de modéliser la fonction Q associée aux récompenses d'un environnement. Cet algorithme est cependant considéré comme étant moins stable alors que

ses performances dépendent fortement de la paramétrisation de la politique.

2.3 Acteur - Critique

Enfin, le modèle d'apprentissage par renforcement profond Acteur - Critique est une combinaison des deux approches précédentes. L'acteur est une implémentation du *Policy Gradient* et utilise les résultats du critique, une implémentation du *Deep Q learning*, pour apprendre la politique optimale.

2.4 Deep Deterministic Policy Gradient

En combinant le modèle Acteur - Critique à des environnements continus, nous obtenons l'algorithme *Deep Deterministic Policy Gradient*, sujet de cet article de recherche. *DDPG* permet cette combinaison en représentant l'espace d'action comme une fonction dérivable.

3 Approche théorique

L'algorithme *Deep Deterministic Policy Gradient* ou DDPG est un algorithme d'apprentissage profond par renforcement dans des espaces d'action continus. L'algorithme peut être vu comme une implémentation du *Deep Q-learning* pour des espaces d'action continus. DDPG utilise une architecture Acteur - Critique, où l'acteur vise à apprendre la politique optimale, et le Critique la *Q-Function*.

3.1 Mise en situation

Imaginez un individu sans aucune expérience voulant apprendre à jouer au golf. Cet individu est l'acteur. Il compte apprendre la bonne technique en suivant les commentaires d'un second individu, lui aussi sans aucune expérience. Ce second individu est le critique.

L'acteur joue et tire des balles à répétition sur le même terrain pour s'entraîner. Après un moment, le critique examine des reprises vidéo d'une sélection de tirs, et donne des conseils au joueur. Pour un coup en particulier, le critique pourrait dire: "La balle n'est pas allée en ligne droite, c'est bien. Dans cette situation, essaie de continuer à sauter avant de frapper la balle." Ce n'est pas un bon conseil, et l'acteur ne le sait pas, le critique non plus. L'acteur suit le conseil, et son jeu ne s'améliore pas.

Il est ainsi essentiel que le critique apprenne à bien évaluer les coups faits par l'acteur dans leur contexte. L'acteur apprend quels coups faire selon la situation en suivant les conseils du critique, qui lui-même apprend en temps réel à bien évaluer les coups de l'acteur. Le critique doit apprendre ce que constitue un bon coup , étant donné que l'acteur ne sera aussi bon au golf que les conseils qui lui sont donnés.

Dans l'algorithme DDPG, L'Acteur et le Critique sont modélisés par des réseaux de neurones. L'apprentissage se fait par descente du gradient sur des expériences antérieures, comme les reprises vidéo des tirs dans cette situation.

3.2 Mémoire d'expériences

Deep Deterministic Policy Gradient est un algorithme hors politique, c'est-à-dire que l'apprentissage est fait avec des échantillons d'expériences antérieures. Ces expériences sont contenues dans une mémoire d'expériences ou *Replay Buffer*. À chaque itération, on ajoute à la mémoire une entrée contenant l'état actuel, l'action choisie, la récompense et le prochain état.

$$ReplayBuffer < obs, act, reward, obs_{next} > \quad (1)$$

Les réseaux Acteur et Critique sont entraînés sur des *mini-batchs* d'expérience. Dans le cadre de l'algorithme, on ne commence l'apprentissage qu'au moment où la mémoire d'expérience contient autant ou plus d'observation que la taille d'une *mini-batch.*

3.3 Acteur

L'acteur est un paramétrage de la politique par un réseau de neurones. Dans le modèle DDPG, la politique apprise est déterministe, c'est-à-dire que l'acteur associe directement une action pour chaque état, et non une distribution de probabilité des actions possibles.

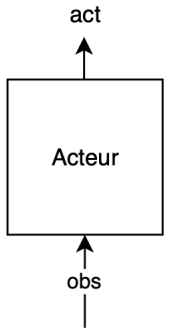

Figure 1: Entrée et sortie de l'acteur [4]

Dans un espace d'action discret fini, il est possible d'identifier l'action optimale en évaluant la Q-value de chaque action, et choisir la meilleure.

$$act^*(obs) = argmax_a Q^*(obs, act) \quad (2)$$

Cette technique est inefficace dans un espace continu, alors qu'il y a une infinité d'actions possibles. DDPG approche ce problème de la manière suivante : comme l'espace d'action est continu, la fonction Q est dérivable en fonction de l'action choisie. L'action optimale est ainsi identifiée par la résolution d'un problème de maximisation par descente du gradient et l'acteur utilise l'approximation de la fonction Q modélisée par le critique pour apprendre la politique optimale.

$$act^*(obs) = maxQ^*(obs, Actor(obs)) \quad (3)$$

Comme mentionné plus haut, l'apprentissage est hors politique, car il est fait avec des expériences passées. L'acteur met à jour la distribution de probabilité des actions possibles pour privilégier les actions menant à une plus grande récompense

attendue selon l'état actuel. On fait une descente du gradient pour minimiser -Q issue du réseau critique, maximisant ainsi la qualité des actions choisies.

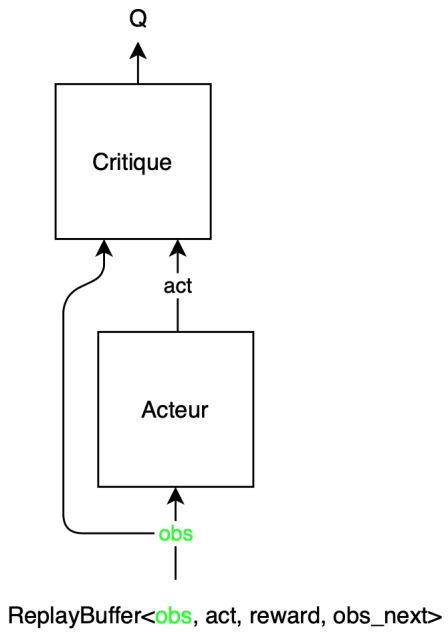

Figure 2: Processus d'apprentissage du réseau acteur [4]

3.4 Critique

Le critique est un paramétrage de la fonction Q par un réseau de neurones. Il utilise l'algorithme hors politique (l'acteur) et une estimation de la valeur Q avec l'équation de Bellman pour apprendre la fonction Q.

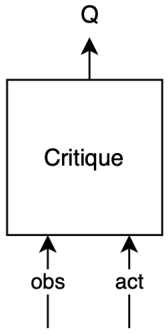

Figure 3: Entrées et sortie du critique [4]

Comme dans l'approche classique du Q-learning, la valeur Q mise à jour est obtenue par l'équation de Bellman. Le calcul nécessite la valeur Q du prochain état ou Qnext.

$$Q^*(obs, act) = r(obs, act) + \gamma * maxQ^*(obs', act') \quad (4)$$

Acteur et Critque cible

La valeur Q* est calculée avec les réseaux de neurones *Target Acteur* et *Target Critique*. Les réseaux de neurones *target* ou cible sont des copies des réseaux acteur et critique. Ils diffèrent cependant dans leur technique de mise à jour. On peut voir ces réseaux comme des versions antérieures. À chaque itération, leurs paramètres sont mis à jour avec une portion des apprentissages de l'Acteur et du Critique.

$$\theta' = \tau\theta + (1 - \tau)\theta' \quad (5)$$

Étant donné que l'équation 4 utilisée pour entraîner le Critique nécessite des valeurs calculées par le réseau lui-même, les réseaux cibles servent à stabiliser l'apprentissage et éviter la divergence de l'algorithme.

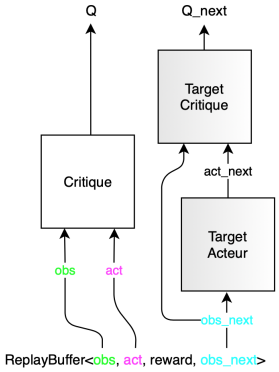

Figure 4: Processus d'apprentissage du réseau critique [4]

Une fois les valeurs connues, on fait une descente du gradient pour minimiser l'erreur entre la valeur Q trouvée par le critique et la valeur Q mise à jour. Cette optimisation apprend au critique à bien évaluer les actions prises par l'acteur.

3.5 Exploration et exploitation

Le problème d'exploration et exploitation dans un espace d'action continu est adressé par l'ajout de bruit aux actions prises par l'acteur. L'implémentation de l'algorithme DDPG suggère l'utilisation d'un bruit provenant du processus Ornstein-Uhlenbeck. Le processus dépend du temps et a originalement été introduit comme une modélisation de la vélocité d'une particule sous un mouvement Brownian.

$$Actor'(obs) = Actor(obs) + N \quad (6)$$

L'ajout du bruit aux actions permet d'explorer l'environnement, de familiariser l'agent à un échantillon de bonne et mauvaise action, et ultimement d'apprendre une meilleure politique optimale.

4 Expériences

4.1 Environnement de test

Afin de tester notre algorithme DDPG, nous avons utilisé une plateforme connue dans le milieu de l'intelligence artificielle,

la plateforme OpenAI Gym. Cette plateforme offre des environnements conçus afin de développer, tester et comparer des algorithmes d'apprentissage par renforcement. Les environnements récompensent l'agent pour chaque action effectuée.

L'environnement utilisé afin de tester notre implémentation de l'algorithme DDPG est l'environnement *LunarLander Continuous V2*, ayant pour but de faire atterrir une navette spatiale sur une plateforme d'atterrissage. Une expérience débute avec la navette spatiale à un endroit fixe, mais avec un vecteur de direction différent à chaque nouvel épisode. L'espace d'action comporte 2 différentes entrées continues ayant un domaine variant de [-1, 1]: une pour le moteur principal sous la navette et une pour les moteurs sur les côtés de celle-ci. Pour résoudre ce problème, l'agent doit obtenir une récompense moyenne de 200 ou plus sur 100 épisodes. Les détails concernant le système de récompenses se retrouvent dans la documentation officielle du Gym OpenAI.

4.2 Configuration

Dans le cadre de nos expériences, nous avons développé deux implémentations de l'algorithme basées sur TensorFlow. Notre première implémentation utilise la librairie Keras pour construire les réseaux de neurones ainsi que les hyper paramètres recommandés dans l'article de référence. [1]

Nos expérimentations avec cette première configuration ont conclu que notre modèle ne réussissait pas à apprendre, ce qui nous à obligé de modifier notre base de code pour se fier vers une implémentation fonctionnelle. En rétrospective, il nous est clair que le manque de succès de cette première implémentation est le résultat d'une mauvaise configuration des librairies utilisées. À ce stade du projet, nous n'étions pas tout à fait familiers avec les différentes versions de Tensorflow et cette implémentation utilisait un mélange des versions 1.0 et 2.0 de la librairie.

À ce stade, nous avons donc recherché une implémentation fonctionnelle sur laquelle se baser. Notre seconde implémentation utilise cette fois tflearn pour entraîner les réseaux de neurones, ainsi que la version 1.15 de Tensorflow pour calculer les gradients et mettre à jour l'algorithme. Le code sur laquelle notre implémentation est basée est disponible la page github[1] suivante.

Notre configuration finale est constituée d'un réseau de neurones avec une couche cachée de 400 neurones et une couche cachée de 200 neurones pour le réseau acteur et pour le réseau critique.

Vous pouvez consulter le code sur le répertoire github [2] de l'équipe.

Pour chaque test, l'algorithme a été entraîné sur 2000 épisodes. Comme l'entraînement prend plusieurs heures, nous avons roulé l'algorithme sur plusieurs ordinateurs en parallèle pour augmenter notre capacité à tester.

5 Présentation des résultats

Le graphique suivant présente les résultats obtenus par notre algorithme sur l'environnement *LunarLander Continuous V2* pour différents learning rates. Chaque entrée représente la moyenne des 100 épisodes précédents.

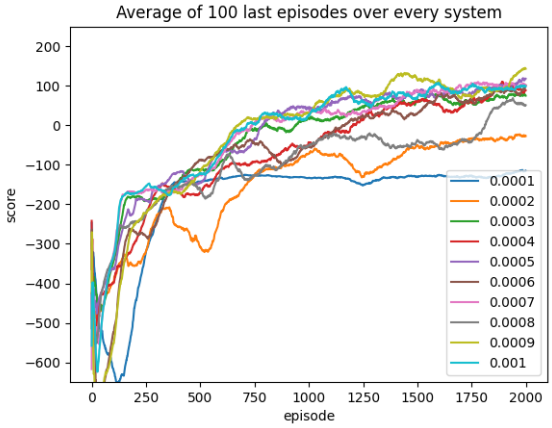

Figure 5: Comparaison de la récompense moyenne des 100 derniers épisodes sur différents taux d'apprentisage avec le code de l'équipe

À titre de comparaison, le graphique suivant présente les résultats obtenus par l'algorithme de référence dans le même environnement. Encore une fois, une entrée représente la moyenne des 100 épisodes précédents.

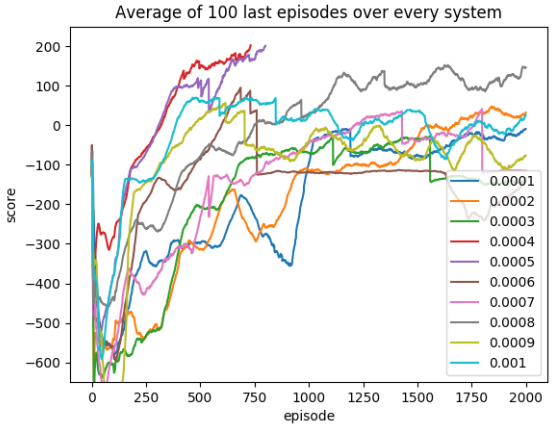

Figure 6: Comparaison de la récompense moyenne des 100 derniers épisodes sur différents taux d'apprentisage exécutés pour le code de référence

Nous avons rapporté les meilleurs résultats des deux algorithmes sur le graphique suivant. Notre algorithme utilise ici un taux d'apprentissage de 0.0009 et l'algorithme de référence un taux d'apprentissage de 0.0004.

[1] https://github.com/shivaverma/OpenAIGym [5]

[2] https://github.com/WSVGNY/DDPG-Project [6]

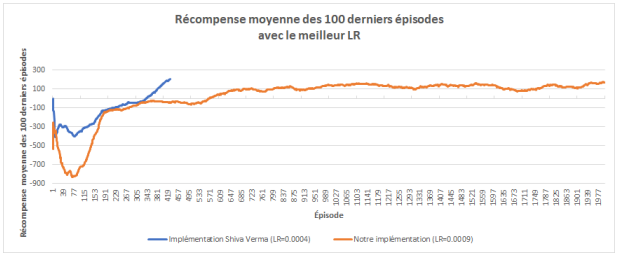

Figure 7: Comparaison avec le code de référence de la récompense moyenne des 100 derniers épisodes sous les meilleures conditions

Il est à noter que chaque séquence d'entraînement prend en moyenne de 5 à 7 heures pour compléter.

Note - plus de résultats de nos expériences sont disponibles sur notre google drive.[3].

6 Discussion sur les résultats

Comme la figure 5 et la figure 6 le démontrent très bien, le taux d'apprentissage joue un rôle très important dans le processus d'apprentissage et sur le succès de l'algorithme. On peut inférer de ce graphique que les hyperparamètres ont une très grande influence sur les résultats finaux.

La comparaison des performances présentées par la figure 7 nous permet de comprendre assez rapidement que notre implémentation ne performe pas à la hauteur de l'algorithme de référence. Notre code se basant sur l'implémentation de Shiva Verna, celui-ci devrait avoir des performances similaires. En regard avec les résultats obtenus, nous observons que Shiva Verna résout le problème en moins de 500 épisodes, tandis qu'après 2000 épisodes, nous n'avons toujours pas résout l'environnement.

Les résultats inférieurs pourraient ainsi être attribués directement à la configuration initiale de notre architecture. Nous avons fait plusieurs essais afin de déterminer un taux d'apprentissage qui améliore les performances. Dans cette optique, il aurait été intéressant de procéder à une telle recherche pour les autres hyperparamètres. Par exemple, changer l'initialisation des poids proposés par l'article de référence pour les réseaux acteur et critique. Une autre variable qui aurait aussi pu être explorée est de faire varier la taille des couches cachées des réseaux de neurones.

7 Analyse critique

Avant de présenter notre méthodologie d'apprentissage, il est important de noter que ce projet est pour tous les membres de l'équipe un premier contact avec l'apprentissage par renforcement profond. De ce fait, nous sommes loin d'être des experts, et nous l'avons réalisé assez tôt dans notre démarche. Malgré notre expérience limitée, nous avons tous un grand intérêt pour la technologie. Nous avons alors défini notre méthodologie d'apprentissage afin que nous puissions réussir à comprendre en profondeur et implémenter notre version de cet algorithme.

[3]https://drive.google.com/drive/folders/13wwhQHtoQ5zTsM2ul1RbaJ-PPKIBK4c5?usp=sharing

Premiers contacts

Pour commencer, nous avons lu l'article de recherche Continuous Control with deep reinforcement learning afin de nous familiariser avec le concept du Deep Deterministic Policy Gradient. Sans nécessairement tout comprendre, cette première lecture nous a permis d'acquérir des connaissances initiales sur l'essence de l'algorithme. Cette lecture a aussi été très utile pour identifier les points nécessitant plus de précision. À ce stade, nous ne comprenons pas très bien les rôles de l'acteur et du critique, ainsi que le fonctionnement de la boucle d'entraînement.

Nous avons ensuite cherché à comprendre le rôle de chaque partie du DDPG. À ce titre, nous avons consulté des références [4] qui vulgarisent des concepts d'apprentissage par renforcement afin de pouvoir bien dresser un portrait de l'algorithme. Des vidéos YouTube et plusieurs articles sur des forums comme Towards Data Science ont été très utiles. Nous avons, par la suite, organisé une téléconférence assurant une compréhension égale du sujet par tous les membres de l'équipe. Cet appel a été bénéfique puisque certains d'entre nous avaient compris certaines subtilités de l'algorithme que d'autres n'avaient pas si bien comprises. Enfin, nous avons conçu une modélisation visuelle de l'algorithme afin de nous préparer à l'implémentation.

Implementation

L'implémentation a été l'étape la plus difficile du projet. Dans un premier temps, nous avons monté un portfolio d'exemples d'implémentation de l'algorithme. Ces exemples nous ont permis d'associer sans difficulté des classes et du code aux modules identifiés à la première étape de notre démarche.

En revanche, nous avons rencontré plusieurs problèmes lors du développement de notre implémentation. Le plus difficile a été la situation des librairies de machine learning, en particulier la librairie Tensorflow et les changements majeurs entre les versions 1.0 et 2.0. Nous avons beaucoup travaillé pour implémenter l'algorithme avec TF 2.0, mais sans succès. La librairie est très différente des versions antérieures et plus complexes d'utilisation pour certaines applications. En somme, la meilleure solution a été de porter notre implémentation vers Tensorflow 1.15.

8 Conclusion

En conclusion, l'objectif de ce projet était d'expérimenter avec l'algorithme Deep Deterministic Policy Gradient. Nous avons commencé par une revue des travaux antérieurs dans le domaine de l'apprentissage par renforcement, suivi d'un résumé de l'approche théorique formant la base de cette recherche. Nous avons ensuite présenté et analysé les résultats de notre implémentation, pour enfin développer une analyse critique sur l'approche utilisée pour se familiariser avec le sujet étudié. Malgré nos résultats non concluants, nous sommes très fiers de notre projet. Nous avons bien maîtrisé le fonctionnement et la motivation derrière l'algorithme Deep Deterministic Policy Gradient. Le projet a aussi été une bonne introduction aux technologies d'apprentissage par renforcement.

Pour aller plus loin, nous avons brièvement testé notre implémentation sur l'environnement Mountain Car Contin-

uous, sans trop de succès. De ce fait, il aurait été intéressant de tester plus en profondeur notre algorithme sur d'autres environnements afin de voir sa capacité à se généraliser pour différents problèmes.

References

[1] Alexander Pritzel Nicolas Heess Tom Erez Yuval Tassa David Silver Daan Wierstra Timothy P. Lillicrap, Jonathan J. Hunt. Continuous control with deep reinforcement learning. https://arxiv.org/pdf/1509.02971.pdf, 2015. [Disponible: Accédé le 15 Avril 2020].

[2] David Silver Andrei A. Rusu Joel Veness Marc G. Bellemare Alex Graves Martin Riedmiller Andreas K. Fidjeland Georg Ostrovski Stig Petersen Charles Beattie Amir Sadik Ioannis Antonoglou Helen King Dharshan Kumaran Daan Wierstra Shane Legg Demis Hassabis Volodymyr Mnih, Koray Kavukcuoglu. Human-level control through deep reinforcement learning. https://www.nature.com/articles/nature14236, 2015. [Disponible: Accédé le 26 Avril 2020].

[3] Satinder Singh Yishay Mansour Richard S. Sutton, David McAllester. Policy gradient methods for reinforcement learning with function approximation. https://homes.cs.washington.edu/~todorov/courses/amath579/reading/PolicyGradient.pdf, 2000. [Disponible: Accédé le 22 Avril 2020].

[4] Aylwin Wei. Reinforcement learning - "ddpg" explained. https://www.youtube.com/watch?v=oydExwuuUCw, 2020. [Disponible: Accédé le 26 Avril 2020].

[5] Shiva Verma. Solving openai gym problems. https://github.com/shivaverma/OpenAIGym, 2020. [Disponible: Accédé le 18 Avril 2020].

[6] Philippe Kavalec Sébastien Labine, William Sevigny. Ddpg-project. https://github.com/WSVGNY/DDPG-Project, 2020. [Disponible: Accédé le 26 Avril 2020].