IAR Deep RL Lab

Arthur Esquerre-Pourtère

Ce document est mon rapport du TME de deep reinforcement learning de l'UE IAR, dans le cadre du M2 spécialité ANDROIDE à Sorbonne Université.

Ce TME consiste à utiliser des algorithmes d'apprentissage par renforcement profond dans un environnement de la librairie gym.

Avec ce document se trouvent les politiques obtenues après entraînement avec les algorithmes DDPG et SAC. Si l'on souhaite évaluer ces politiques il faut les mettre dans le dossier "src/data/policies" et lancer le fichier "evaluator.py". L'agent le plus performant d'après mes tests est celui obtenu avec l'algorithme DDPG. Dans le dossier src on retrouve tout le code que j'ai utilisé, les politiques DDPG et SAC sont dans les fichiers "src/policies/ddpg.py" et "src/policies/sac.py", on peut les entrainer en lançant les fichiers "src/main_ddpg.py" ou "src/main_sac.py".Il faut installer auparavant la librairie machin pour faire ces entraînements.

1 Environnement

L'environnement que je vais donc utiliser pour ce TME vient de la librairie python gym et s'appelle "Pendulum-v0". Le but de cet environnement est de faire tenir droit (vers le haut) un pendulum, en appliquant une force dans un sens ou dans l'autre et en essayant de minimiser cette force. À chaque étape, on peut donc effectuer une action continue d'une valeur entre -2 et 2 qui correspond à la force que l'on applique et à sa direction. L'état est décrit par trois valeurs : le cosinus de l'angle, le sinus, ainsi que la vitesse angulaire. La reward vaut entre -16.2736044 et 0. Celle-ci pénalise le fait d'être loin de l'état où le pendulum pointe vers le haut et pénalise aussi l'effort (la force) que l'on utilise. On peut voir une image de cet enviromment, dans le cas où le pendulum penche vers la gauche sur la figure [1].

Figure 1: L'environnement pendulum

2 Policy gradient

J'ai commencé par utiliser le code fourni au départ pour le TME. J'ai donc utilisé les algorithmes de type policy gradient. Plus particulièrement, j'ai commencé très simplement avec une politique de type bernouilli. Il s'agit d'une politique stochastique binaire qui permet d'utiliser seulement 2 actions différentes (des actions discrètes donc). À chaque étape on a donc une probabilité de prendre une action ou l'autre. Cette probabilité dépend de notre politique et est définie par un réseau de neurones, qui prend en entrée l'état et donne en sortie cette probabilité. L'apprentissage se fait en interagissant avec l'environnement et consiste donc à apprendre les poids de notre réseau de neurones via une descente de gradient.

J'ai fait cet apprentissage sur 500 épisodes et j'ai utilisé les 3 variantes présentes dans le code, le cas où l'on prend en compte la somme des rewards, le cas où l'on prend la somme des rewards en appliquant un facteur gamma qui réduit la valeur des rewards que l'on rencontre plus tard et la somme des rewards normalisée en appliquant un facteur gamma et où l'on soustrait la valeur moyenne.

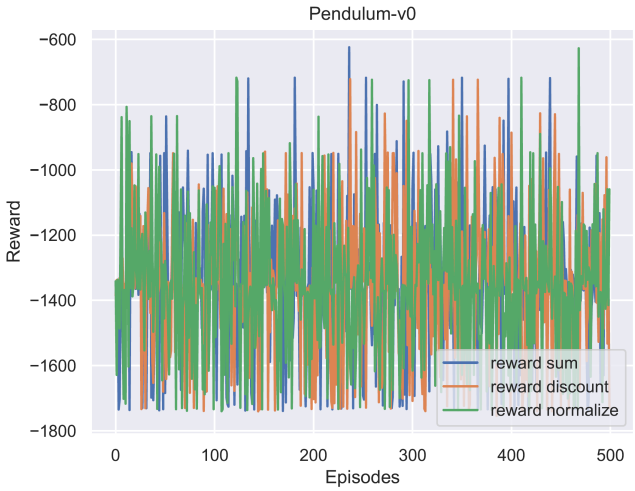

Figure 2: Évolution de la reward obtenue pendant les 500 épisodes d'apprentissage avec la politique bernouilli

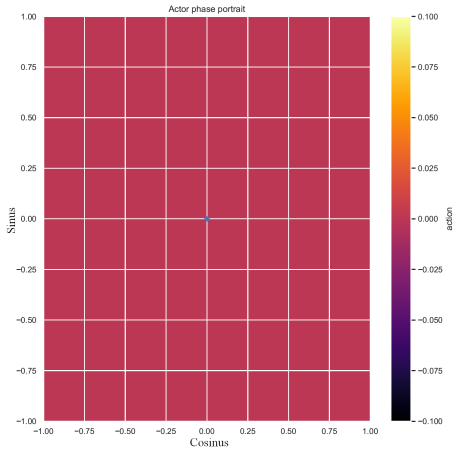

Figure 3: Action en fonction de l'état, après apprentissage avec la politique bernouilli (normalized reward)

Comme on le voit dans la figure [2], la reward obtenue n'a pas augmenté au cours de l'apprentissage. On observe cela pour les 3 variantes de la reward. Cela montre donc que l'agent n'a pas réussi à apprendre d'une manière efficace.

On peut aussi voir sur la figure [3] la politique apprise après apprentissage. Ce graphe affiche pour chaque état l'action qu'on a la plus grande probabilité d'effectuer, attention toutefois, dans le cas du pendulum, tout les états représentés ne sont pas possibles car les valeurs de sinus et cosinus dont dépendantes. Les seules valeurs à prendre en comptes sont celles qui sont situées sur un cercle centré en (0,0) et de rayon 1. On voit clairement qu'il y a un problème, en effet l'agent a appris à effectuer toujours la même action quelque soit état.

J'ai donc choisi ensuite d'utiliser un autre algorithme de type policy gradient. En effet, la politique de type

bernouilli nous permettait seulement d'avoir des actions discrètes, toutefois le problème du pendulum est un problème où les actions sont continues. Il serait donc pertinent d'avoir un algorithme capable de gérer cela. J'ai donc utilisé une politique de type gaussienne (ou normale). Cette fois-ci, la politique est toujours stochastique mais les probabilités des actions sont définies de manière continues via une gaussienne. Le but de l'apprentissage est donc d'apprendre la valeur "mu" du centre de cette gaussienne pour chaque action. De plus la valeur de la variance varie aussi puisqu'elle est apprise par l'algorithme.

J'espérais via cette expérience, en laissant une plus grande marge de manoeuvre pour les actions, que l'algorithme arriverait à apprendre plus facilement. En effet peut-être que les actions discrètes n'étaient tout simplement pas suffisantes pour faire tenir le pendulum et que l'algorithme n'arrivait donc pas à apprendre efficacement.

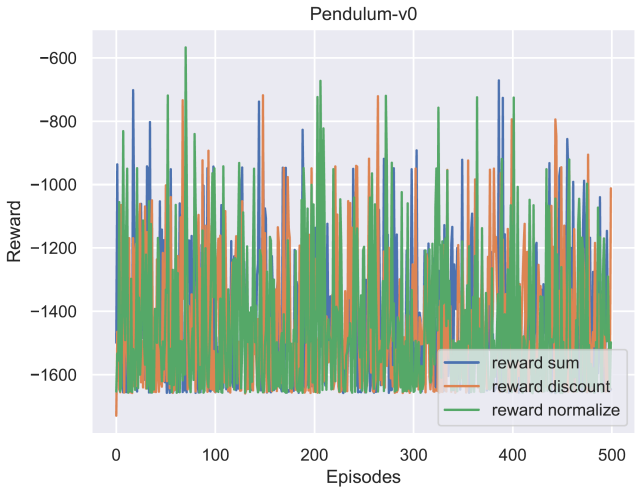

Figure 4: Évolution de la reward obtenue pendant les 500 épisodes d'apprentissage avec la politique gaussian

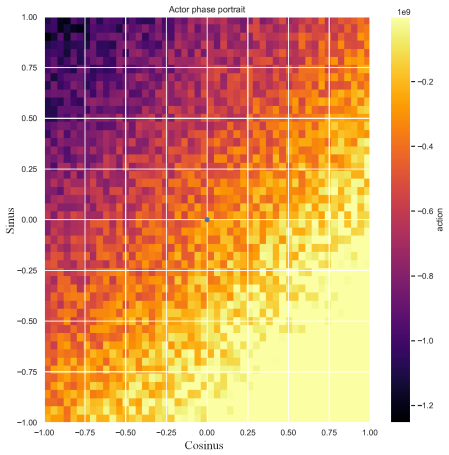

Figure 5: Action en fonction de l'état, après apprentissage avec la politique gaussian (normalized reward)

Toutefois, comme on le voit une fois de plus dans la figure [4], la reward obtenue n'augmente pas non plus au cours de l'apprentissage.
On peut toutefois remarquer un phénomène intéressant. Au premier coup d'oeil on peut avoir l'impression que l'algorithme effectue des actions différentes selon les états sur la figure [5]. Mais on peut remarquer

en regarder l'échelle de valeur que la plupart des états prennent des valeurs extrêmement grandes pour les actions...alors que le pendulum accepte seulement des valeurs entre -2 et 2. Les différences que l'on observe ne sont donc pas significatives et correspondent probablement pour la plupart à la même action. Ceci risque de poser des problèmes pour l'apprentissage et on pourrait donc vouloir empêcher l'algorithme de donner de telles valeurs aux actions.

Afin de résoudre cela j'ai choisi d'utiliser une politique de type squashed gaussian. Ceci consiste à garder le même algorithme qu'avant, mais en faisant passer les valeurs des actions dans une fonction tangente hyperbolique. Ainsi, on s'assure que les valeurs des actions restent dans les bornes fixées par l'environnement.

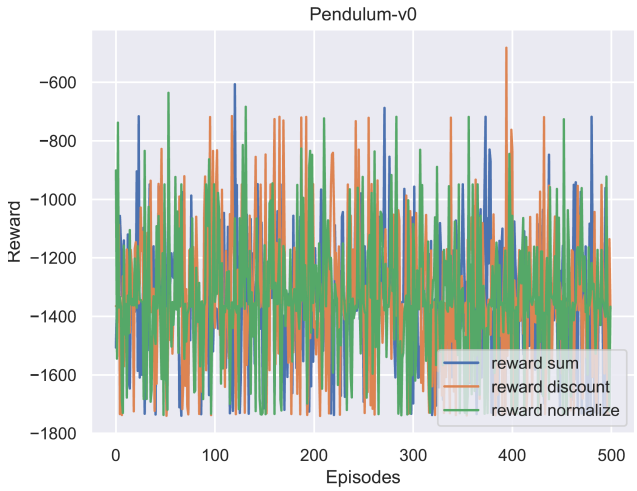

Figure 6: Évolution de la reward obtenue pendant les 500 épisodes d'apprentissage avec la politique squashed gaussian

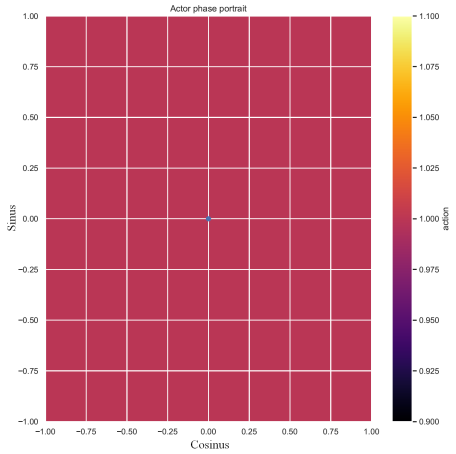

Figure 7: Action en fonction de l'état, après apprentissage avec la politique squashed gaussian (normalized reward)

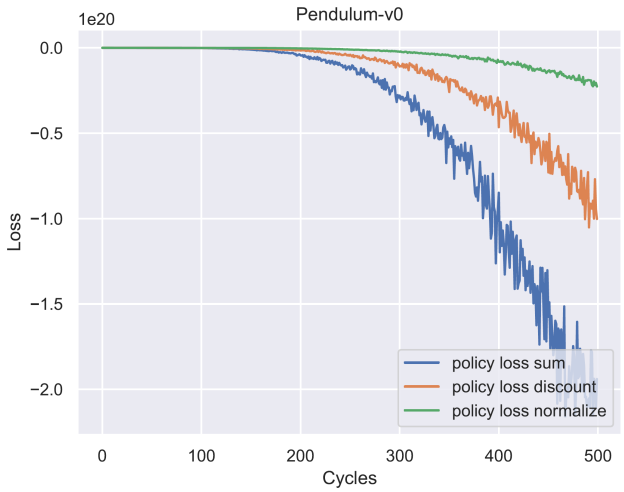

Figure 8: Évolution de la policy loss pendant les 500 épisodes d'apprentissage avec la politique squashed gaussian

Comme on le voit sur la figure [7] , ceci a permis d'empêcher les valeurs des actions d'exploser. Toutefois on rencontre à nouveau le problème qu'on avait auparavant : on obtient toujours la même action quelque soit l'état.
On voit aussi sur la figure [8] que la valeur de la policy loss prend une valeur négative extrêmement grande (ordre de grandeur de 10^{20}). Il faut aussi essayer de résoudre ce problème.

De plus, on observe sur la figure [6] que l'algorithme n'arrive toujours pas à faire augmenter la reward au cours de l'entraînement et n'arrive donc pas à apprendre.

Une hypothèse de l'origine de problème est que l'algorithme apprendrait mieux en ayant des valeurs de reward positives et plus proches de 0. J'ai donc choisi de redimensionner les valeurs des rewards. Pour cela, j'applique sur chaque reward la formule suivante : $newReward = (16.2736044 + reward)/20$. J'ai mis cette formule en commentaire dans le fichier "src/wrappers/pendulum_wrapper.py".
J'espérais ainsi qu'avec de telles valeurs le réseau de neurones apprendrait beaucoup mieux.

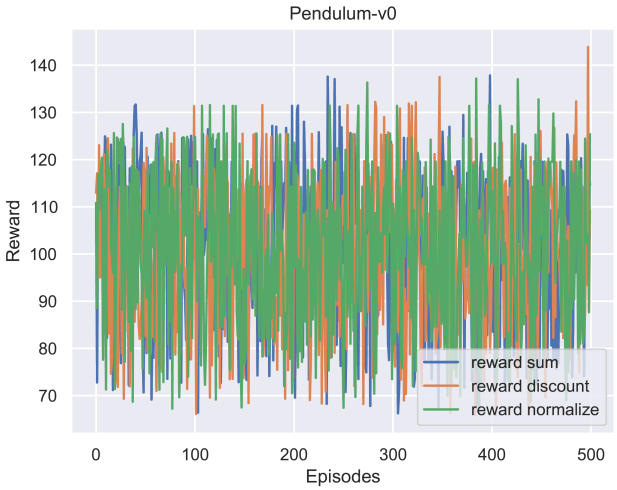

Figure 9: Évolution de la reward obtenue pendant les 500 épisodes d'apprentissage avec la politique squashed gaussian, en renormalisant la reward

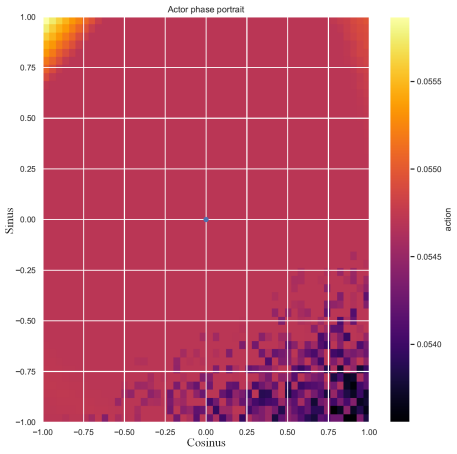

Figure 10: Action en fonction de l'état, après apprentissage avec la politique squashed gaussian (normalized reward), en renormalisant la reward

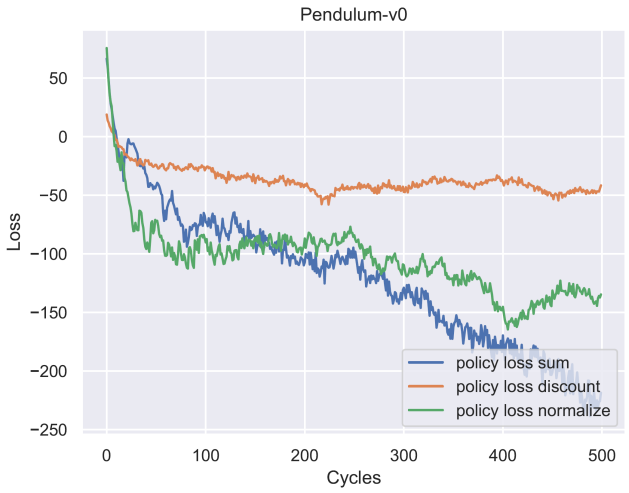

Figure 11: Évolution de la policy loss pendant les 500 épisodes d'apprentissage avec la politique squashed gaussian, en renormalisant la reward

Malheureusement, comme on le voit sur la figure [9] , cela n'a pas été suffisant et l'algorithme n'arrivait toujours pas à apprendre. Toutefois, on peut observer un résultat encourageant, on voit en effet sur la figure [11] que la valeur de la policy loss n'explosait plus et ne prenait plus des valeurs démesurées. La valeur de la policy loss a donc bien diminué comme espéré mais cela n'a pas suffit pour faire bien fonctionner l'algorithme.

On observe aussi sur la figure [10] que l'algorithme effectue toujours des actions ayant des valeurs entre 0.0535 et 0.0560. Cela semble donc logique que l'algorithme ne soit pas capable de faire tenir droit le pendulum.

Ensuite, j'ai essayé d'adapter le learning rate, car un learning rate trop grand pourrait empêcher l'algorithme de converger vers une bonne solution et qu'un learning rate trop petit pourrait rendre l'algorithme trop lent à converger. J'ai essayé entre autres des learning rate de 0.1, 0.01 ou encore 0.001.

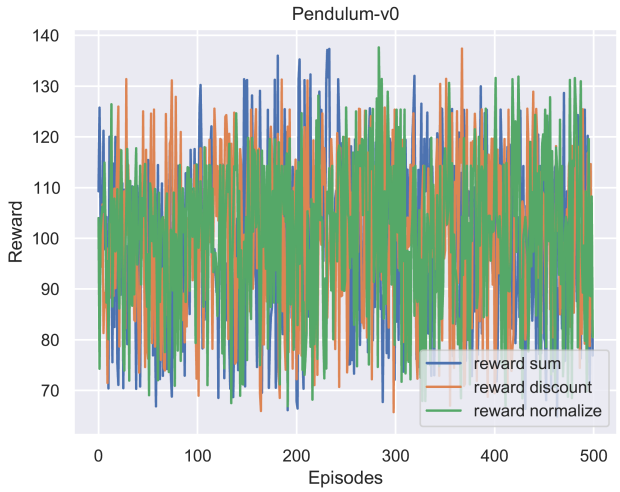

Figure 12: Évolution de la reward obtenue pendant les 500 épisodes d'apprentissage avec la politique squashed gaussian, en renormalisant la reward et avec un learning rate de 0.1

Malheureusement cela n'a pas fonctionné non plus. J'ai mis un exemple de la courbe des reward obtenues dans la figure [12] dans le cas où l'on prend un learning rate de 0.1

Enfin, j'ai fait l'hypothèse que cela pouvait être dû à un manque d'exploration et que ce manque pouvait être dû au fait que l'on effectuait aussi la descente de gradient sur l'écart type de la gaussienne. Afin de tester mon hypothèse, j'ai effectué plusieurs tests en fixant la valeur de l'écart type de la gaussienne et en observant les résultats.

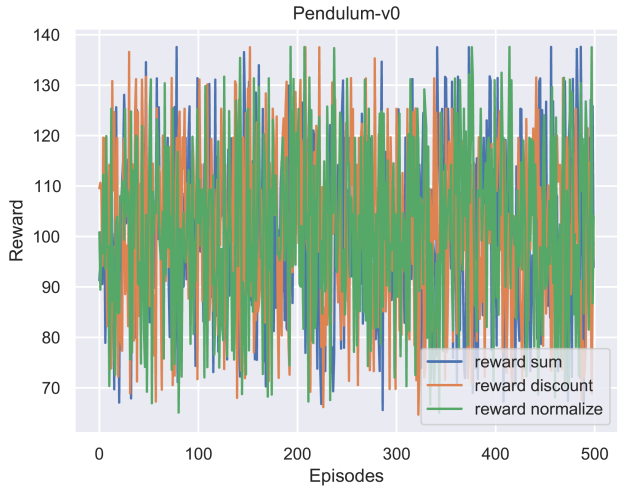

Figure 13: Évolution de la reward obtenue pendant les 500 épisodes d'apprentissage avec la politique squashed gaussian, en renormalisant la reward et avec un écart-type fixe de 0.5

J'ai donc testé plusieurs valeurs d'écart type : 0.1, 0.5 et 1.5.
Toutefois J'ai obtenu à chaque fois des résultats similaires, l'algorithme n'arrivait pas à apprendre correctement. On peut voir sur la figure [13] l'évolution de la reward dans le cas d'un écart type fixe de 0,5. On voit clairement que cette reward n'augmente pas au cours de l'apprentissage.

3 DDPG

N'ayant pas réussi à obtenir de résultats satisfaisants en essayant d'ajuster l'algorithme de policy gradient et en faisant varier les hyper paramètres, j'ai donc choisi d'utiliser d'autres types d'algorithme de deep reinforcement learning. J'ai commencé par l'algorithme déterministe DDPG. L'algorithme DDPG est off policy et utilise un replay buffer, ce qui le rend assez sample efficient, autrement dit, il utilise efficacement toutes les expériences qu'il a eu avec l'environnement pendulum.
De plus, cet algorithme utilise la méthode actor-critic, il y a donc 2 réseaux de neurones :

- Un réseau actor, qui prend en entrée l'état et donne en sortie une action à effectuer
- Un réseau critic, qui prend en entrée l'état et l'action déterminée par l'actor et donne en sortie la Q-Value du couple état action

L'apprentissage se fait en apprenant les valeurs optimales pour des poids du critic via une descente de gradient, en essayant de minimiser l'erreur pour la QValue (en utilisant la technique de la temporal difference). De même, on apprend aussi les poids du réseau actor, une fois de plus par descente de gradient, en utilisant cette fois le gradient de la QValue (donnée par le critic) en fonction de l'action donnée par l'actor.

Plutôt que d'implémenter par moi-même cet algorithme, j'ai utilisé une librairie, accessible sur github via ce lien : https://github.com/iffiX/machin

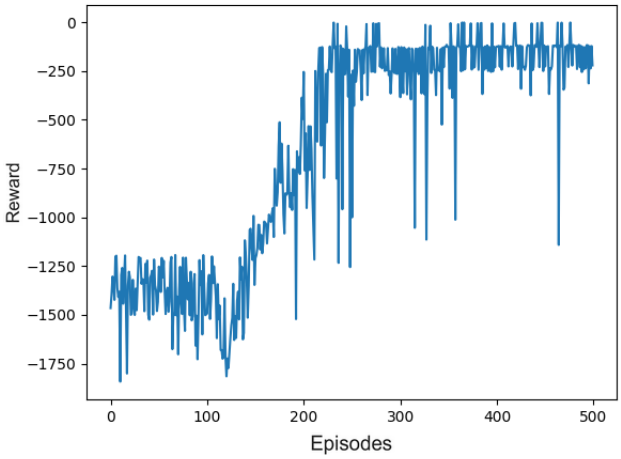

Figure 14: Évolution de la reward obtenue pendant les 500 épisodes d'apprentissage avec l'algorithme DDPG

Sur la figure [14], on voit la reward obtenue au cours de l'entraînement de l'algorithme. On voit clairement que cette reward augmente avec le temps, ce qui semble montrer que, contrairement aux expériences précédentes, l'apprentissage a bien fonctionné.
J'ai ensuite utilisé la classe evaluator, pour faire l'évaluation finale du résultat obtenu. Cette évaluation est faite en répétant l'expérience 900 fois, afin d'avoir des résultats peu bruités. J'ai obtenu un score de -146, avec un écart-type de 85.

4 SAC

J'ai ensuite choisi d'utiliser l'algorithme d'état de l'art "SAC" en espérant pouvoir obtenir de meilleurs résultats.
On peut séparer les algorithmes de deep reinforcement learning en 2 catégories : les off-policy, qui sont assez instables et mais utilisent efficacement toutes les interactions que l'agent a eu avec l'environnement et les

on-policy qui sont plus stables, mais risquent d'avoir besoin de plus d'interactions avec l'environnement. Le but de l'algorithme SAC est de mélanger ces 2 approches afin de combiner leurs avantages. L'algorithme utilise donc un replay buffer comme DDPG, mais il s'agit cette fois d'un algorithme stochastique. En effet, l'algorithme utilise une méthode appelée "entropy regularization" afin de choisir ses actions de manières stochastique et donc de favoriser l'exploration.

J'ai donc choisi d'utiliser cet algorithme en espérant que la stabilité apportée par SAC comparée à DDPG me permettrait d'obtenir de meilleurs résultats.

Cette fois aussi j'ai utilisé la librairie machin pour l'implémentation, accessible via ce lien :
https://github.com/iffiX/machin

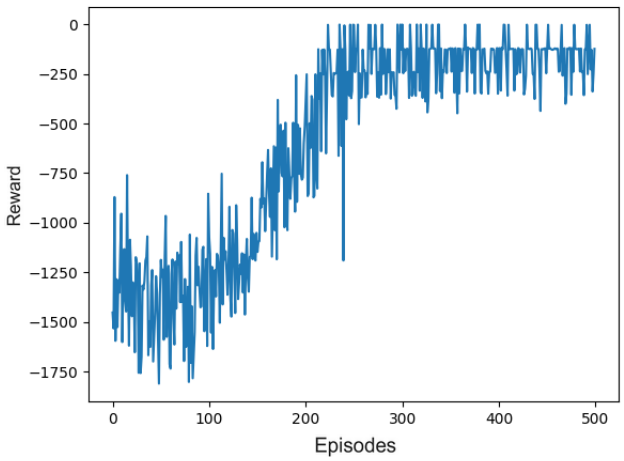

Figure 15: Évolution de la reward obtenue pendant les 500 épisodes d'apprentissage avec l'algorithme SAC

Sur la figure [15], on voit que la reward obtenue au cours de l'entraînement de l'algorithme a augmenté au cours de l'apprentissage. On voit clairement que cette reward augmente jusque l'épisode 250 environ et n'augmentent plus ensuite. Cette fois l'algorithme a donc réussi à converger vers une solution satisfaisante en environ 250 épisodes.
Cette fois aussi j'ai utilisé la classe evaluator et j'ai obtenu un score moyen de -153.4 avec un écart-type de 94. L'agent obtenu via apprentissage par l'algorithme est donc légèrement moins performant que celui que j'ai obtenu via DDPG. Toutefois, en faisant tourner plusieurs fois l'evaluateur, les résultats que j'obtenais variaient légèrement. La différence obtenue entre l'évaluation de l'algorithme DDPG et l'évaluation de l'algorithme SAC n'est donc peut-être pas très significative.

5 "Bonus" : MountainCar

J'ai utilisé le temps restant pour m'intéresser à l'environnement MountainCar-v0. J'ai juste eu le temps de commencer à m'y intéresser mais je n'ai pas pu pousser la réflexion aussi loin que pour le pendulum.

Le problème de l'environnement MountainCar est qu'il est très difficile pour l'agent de trouver une bonne reward. Cette reward est trop clairsemée. Beaucoup de méthodes habituelles ne fonctionnent donc pas.

Par curiosité, j'ai donc essayé de m'inspirer d'une méthode de type novelty search, afin de l'adapter à l'apprentissage par renforcement.

J'ai donc utilisé l'algorithme DQN, mais j'ai modifié la reward. Cette fois l'agent était récompensé en fonction de la distance entre sa position, lors de l'état final, avec les positions obtenues lors des autres épisodes (J'ai considéré les 20 plus proches voisins). J'espérais que cela permettrait d'encourager l'exploration et que l'agent arriverait à résoudre l'environnement.

Le code de ce que j'ai fait se trouve dans les fichiers "src/policies/dqn.py" et "src/main_dqn.py".

J'ai encore utilisé la librairie machin pour l'implémentation de DQN : https://github.com/iffiX/machin

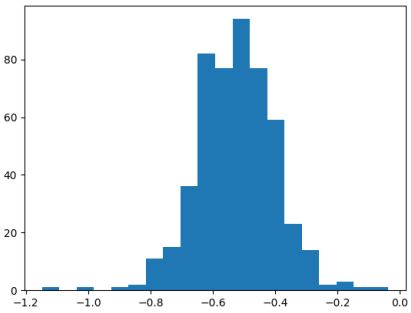

Figure 16: Histogramme des positions à la fin des épisodes

On voit sur la figure [16] que cela n'a pas bien fonctionné. Les agents sont globalement restés bloqués dans la cuvette et n'ont pas beaucoup exploré.

On pourrait peut-être obtenir de meilleures performances en prenant aussi en compte la vélocité pour évaluer la nouveauté. On pourrait aussi faire varier le nombre de "voisins" qu'on utilise pour évaluer la nouveauté.

De plus, il serait intéressant de tester la version originale de novelty search, avec des algorithmes génétiques, sur cet environnement.