RLD : Rapport TME Première Partie

Saadaoui Aymane & Afandi Mohammad

Décembre 2021

Table des matières

1 Bandits **2**

2 ValueIteration **5**

3 Q-Learning **8**

4 DQN **12**

5 A2C **15**

6 PPO **19**

7 DDPG **22**

8 SAC **24**

1 Bandits

Pour ce TME on étudie le problème des bandits multi-bras avec les algorithmes UCB et Lin-UCB appliqués à la sélection de publicités en ligne.

Nous disposons des taux de clics sur les publicités de 10 annonceurs pour 5000 articles, ainsi que du contexte (profil d'article). Pour chaque visite, l'objectif est de choisir la publicité d'un des 10 annonceurs permettant d'engranger le plus fort taux de clics. Il s'agit d'un problème de bandit manchot où les machines sont les annonceurs, les récompenses sont les taux de clics et le but est de maximiser le taux de clics cumulés sur les 5000 visites.

Pour pouvoir comparer l'efficacité de nos algorithmes, nous allons mettre en place 3 baselines :

- Random : on choisit un bras aléatoirement ;
- StaticBest : on choisit le bras qui possède le meilleur taux de clics cumulés depuis le début ;
- Optimale : on choisit le bras qui possède le meilleur taux de clics à chaque instant.

On notera que les deux dernières baselines sont utopiques : elles exigent de pouvoir connaître à chaque itération les taux de clics de tous les annonceurs.

Une fois cela fait nous implémentons nos algorithmes UCB, LinUCB que nous allons comparer aux baselines.

- UCB (Upper Confidence Bound) : il s'agit d'une stratégie optimiste où l'on choisit le bras qui serait le meilleur si les valeurs des bras étaient les meilleures possibles selon l'intervalle de confiance ;
- LinUCB : UCB avec cette fois la prise en compte du contexte à chaque instant.

Ci-dessous nous affichhons les rewards et la fréquence des choix des annonceurs.

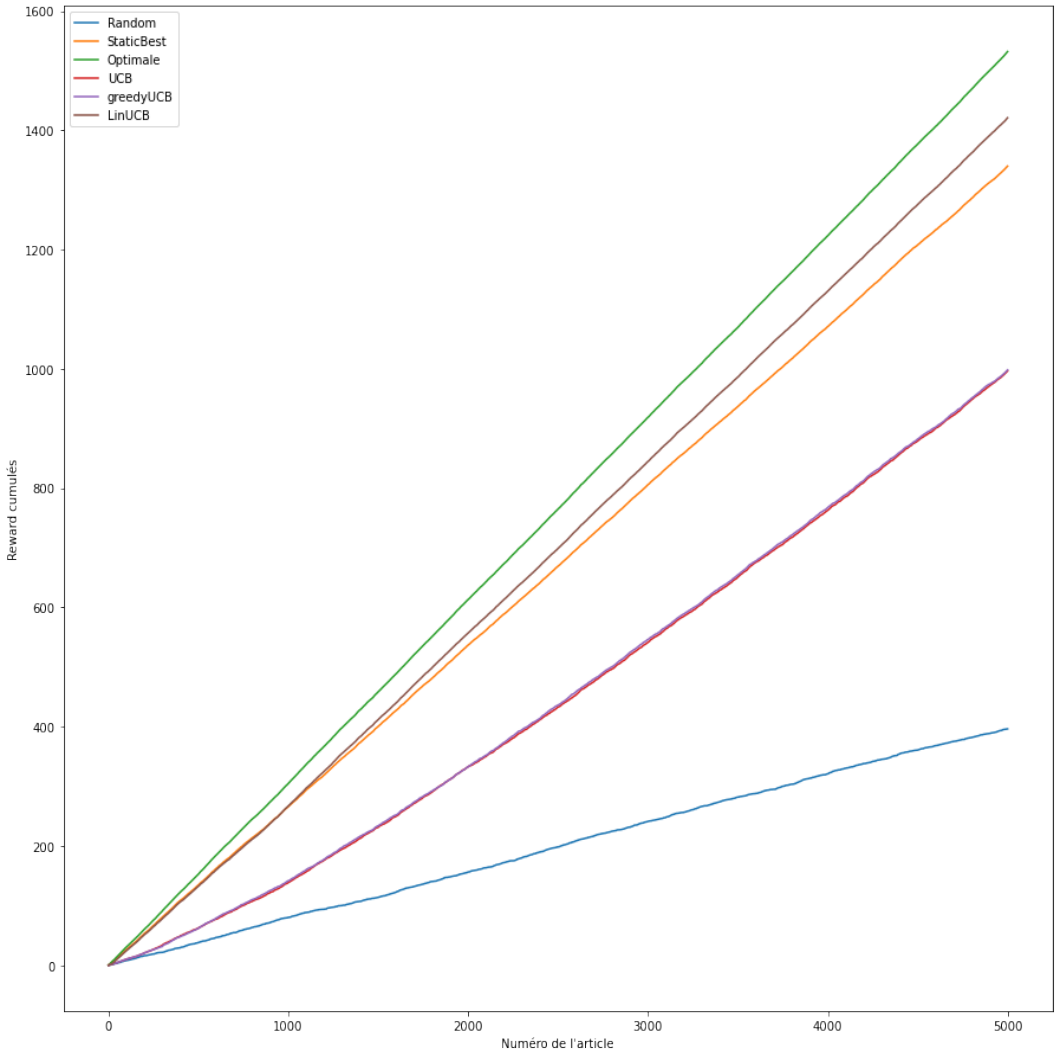


Figure 1: Rewards cumulées selon différentes stratégies au cours du temps

On constate que la stratégie Lin-UCB fait presque aussi bien que la stratégie Optimale. Il y a une nette différence avec UCB simple. La prise en compte du contexte permet donc ici de réaliser des gains non négligeables.

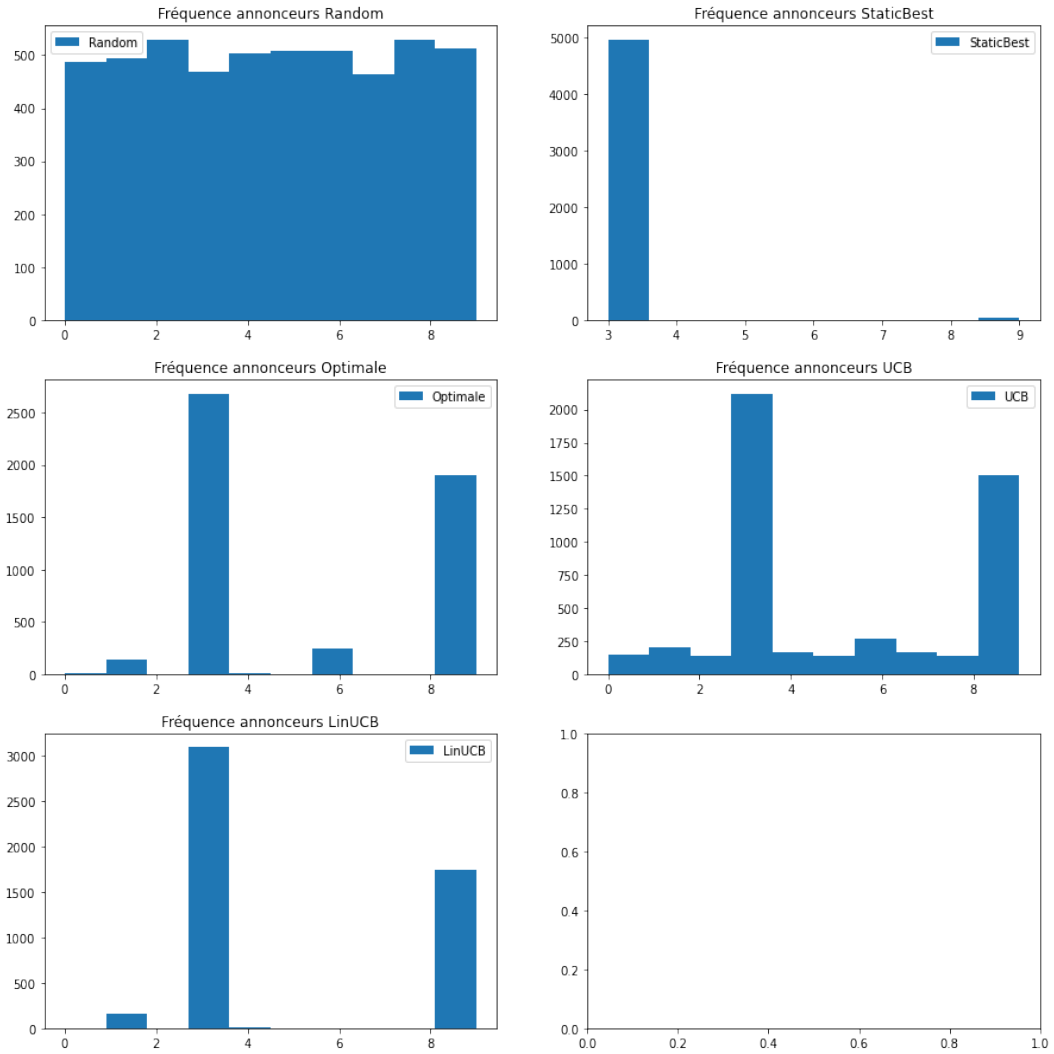


Figure 2: Fréquence des annonceurs choisis selon la stratégie

On remarque que les distribution des annonceurs choisis par les stratégies Optimale, UCB et Lin-UCB sont très similaires (principalement les annonceurs 4 et 10), tandis que pour StaticBest un seul annonceur est choisi (le 4).

2 ValueIteration

Pour ce TME nous allons étudier les algorithmes de Policy et Value Iteration sur le jeu `gridworld`. Il s'agit d'un exemple de offline solution où le MDP est totalement connu par avance.

Algorithmes

- Policy Iteration (PI) : on évalue la politique entièrement à chaque itération
- Value Iteration (VI) : on se contente à chaque itération de mettre à jour la valeur des états pour à la fin définir une politique optimale.

Remarque : on peut montrer que ces deux approches sont en fait équivalentes en s'intéressant à l'équation d'optimalité de Bellman.

Environnement

Les algorithmes sont appliqués à l'environnement `gridworld`. Ici l'agent (représenté par un carré bleu) est placé dans un labyrinthe où il doit récupérer des récompenses (carrés jaunes) puis se diriger vers la sortie (carré vert) tout en évitant les malus (carrés roses) et les pièges (carrés rouges). Le tout doit être réalisé le plus rapidement possible. Plus formellement :

- Carré jaune : $reward = 1$
- Carré rose : $reward = -1$
- Carré rouge : $reward = -1$ (final)
- Carré vert : $reward = 1$ (final)

Expériences

On remarque que l'hyperparamètre γ qui nous sert de facteur de discount a un grand impact sur la stabilité et la convergence.
Il permet d'accélérer la découverte de bonnes stratégies. S'il est proche de 1, les récompenses cumulées n'auront pas trop été modifiées et l'algorithme sera le plus précis possible ; s'il est proche de 0, plus nous avançons dans la trajectoire et plus nos récompenses vont devenir petites, entraînant donc une accélération de l'apprentissage.
Pour l'instant nous conservons $\gamma = 0.99$. Nous aurons l'opportunité de voir

son effet sur d'autres algorithmes.

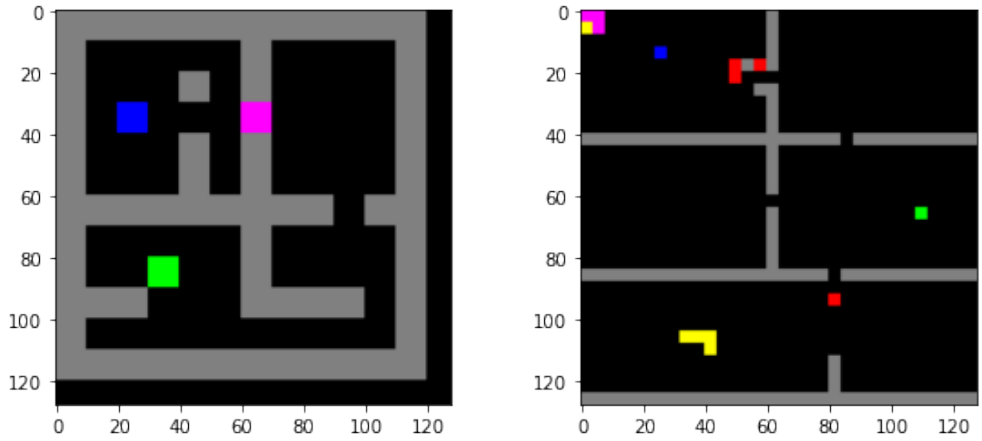
Comparons désormais ces deux méthodes sur plusieurs cartes de `gridworld`. Le temps s'exprime en ms.

Map	PI reward	VI reward	PI time	VI time
0	0.99	0.99	7	3
1	1.98	1.99	12	4
2	1.98	1.98	56	37
3	0.99	0.99	6	5
4	-2	-2	6975	102
5	1.95	1.95	140	200
6	1.94	1.93	697	807
7	2.95	2.93	40192	518
8	0.93	0.94	1244	1366
9	Nb	Nb	Nb	Nb
10	0.96	0.95	48	34

En analysant ce tableau, nous voyons qu'il est préférable d'utiliser l'algorithme de Value Iteration. En effet, pour certaines cartes l'algorithme de Policy Iteration est plus rapide de quelques ms mais sur d'autres, comme la 7, l'algorithme Value Iteration s'exécute 77 fois plus rapidement pour un score presque similaire.

Pour la carte 4 nos deux algorithmes obtiennent un score de -2 ce qui est étrange. Pour la carte 9 nous obtenons une erreur de type `RecursionError` :

```
RecursionError: maximum recursion depth exceeded while calling a
Python object.
```



(a) Carte 4 (b) Carte 9

Figure 3: Cartes 4 et 9 de `gridworld`

On constate pour la carte 4 qu'il faut nécessairement passer par la case piège non mortelle en rose pour atteindre la sortie. L'architecture de la carte nous donne au maximum un score de -0.022 pour la trajectoire optimale, ce qui est loin du -2 que nous obtenons.

Pour la carte 9 on voit qu'il y a énormément d'états à prendre en compte du fait de la difficulté et de la taille de la carte, d'où l'erreur obtenue. Le MDP de cette carte est en réalité beaucoup trop lourd à extraire.

3 Q-Learning

Pour ce TME nous étudions l'algorithme Q-Learning et ses variantes sur le jeu `gridworld`. Cette fois le MDP n'est pas connu par avance, il va donc falloir trouver un compromis entre exploration et exploitation pour établir une bonne politique. Il s'agit d'un exemple de online solution.
Pour trouver une bonne politique lorsque l'on n'a pas de connaissance sur le MDP, deux approches sont possibles :

- Approche Model-Based : recréer un MDP approximé par exploration et ensuite appliquer les algorithmes vus au TME 2 pour obtenir une bonne politique qui maximise le reward.
- Approche Model-Free : déterminer une politique en utilisant les Q-values pour maximiser le reward sans chercher à obtenir le MDP.

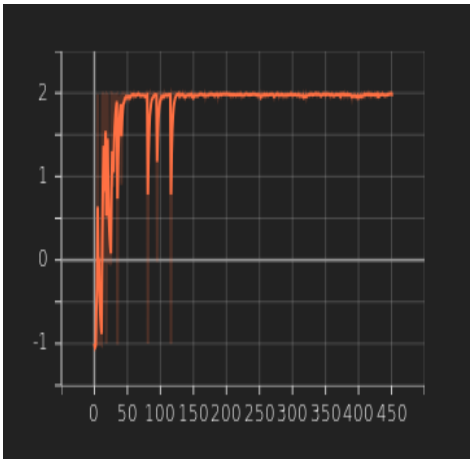
L'algorithme Q-Learning est en particulier un algorithme Model-Free, mais on verra une de ses variantes DynaQ qui combine les approches Model-Free et Model-Based.

Remarque : Q-learning est off-policy alors que Sarsa est on-policy, dans ces deux algorithmes on utilise une stratégie d'exploration (epsilon-greedy) pour échantillonner l'action à réaliser. Cependant lors du calcul du TD error l'algorithme Q-learning basique n'utilise pas cette stratégie contrairement à Sarsa.

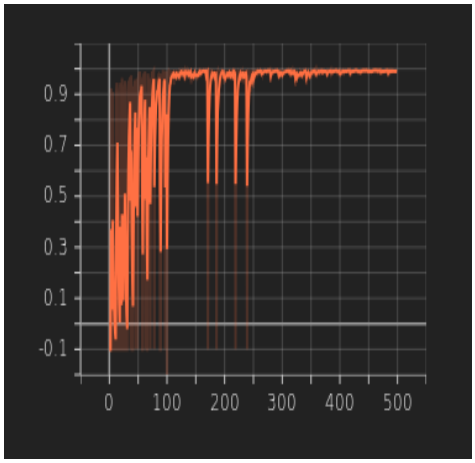
Expériences

Nous allons présenter les résultats de notre implémentation sur les cartes 1 et 5 de gridworld.

Nous utilisons pour notre algorithme : une stratégie d'exploration ϵ-greedy avec $\epsilon = 0.1$, $\gamma = 0.99$, decay $= 0.999$

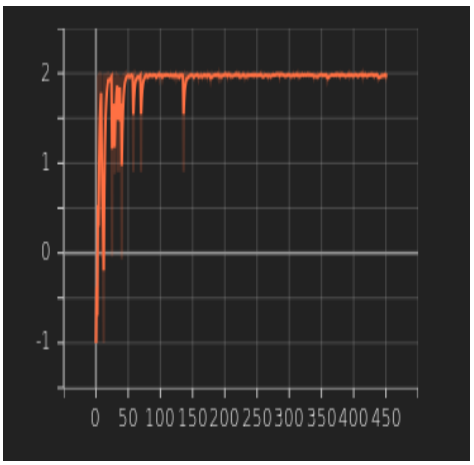


(a) Carte 1

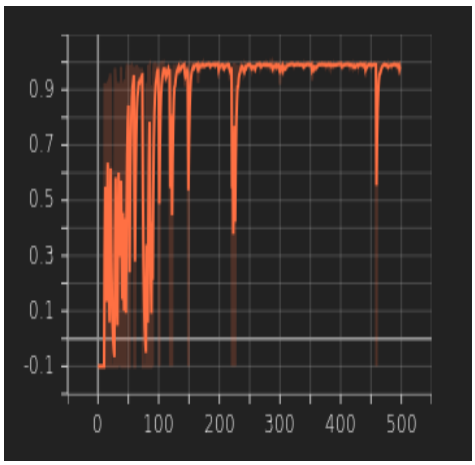


(b) Carte 5

Figure 4: Rewards de l'algorithme QLearning basique

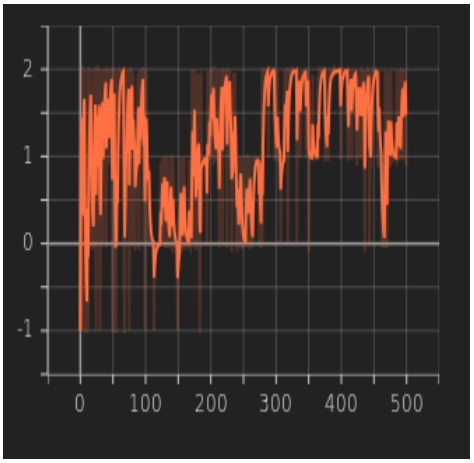


(a) Carte 1

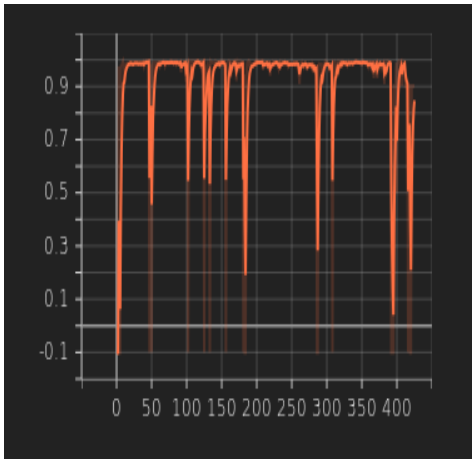


(b) Carte 5

Figure 5: Rewards de l'algorithme SARSA



(a) Carte 1



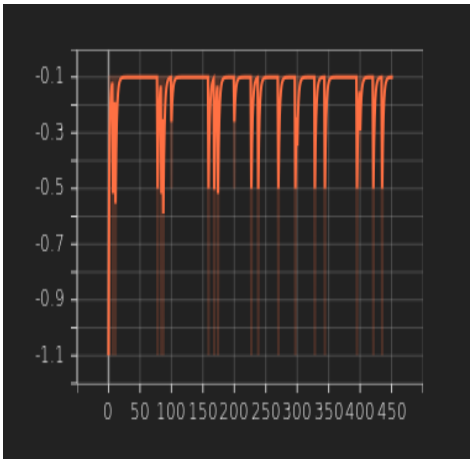
(b) Carte 5

Figure 6: Rewards de l'algorithme DynaQ

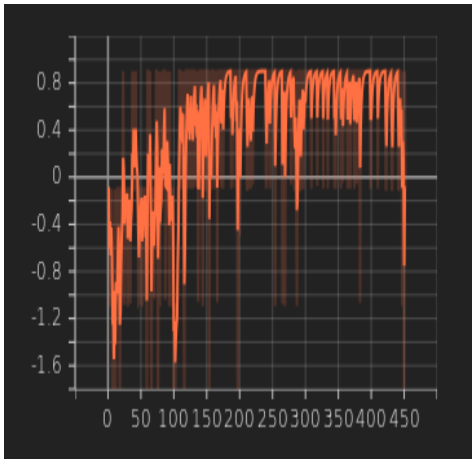
DynaQ est plus instable que les autres algorithmes et devrait mieux se débrouiller sur la carte 5 normalement car posséderait une meilleure connaissance de l'environnement grâce au MDP approximé. Peut être qu'en laissant tourner l'algorithme plus longtemps nous aurions un meilleur score mais le temps de calcul nécessaire pour DynaQ serait alors bien plus grand (27 fois lent que la version classique avec notre implémentation).

Nous avons tenté d'augmenter ϵ pour explorer un peu plus et améliorer notre score sur la carte 5 mais sans succès même avec un très grand ϵ et decay.

Nous nous intéressons finalement aux scores obtenus par SARSA pour les cartes 4 et 9 où nous avions eu des problème au TME précedent.



(a) Carte 4



(b) Carte 9

Figure 7: Rewards de SARSA pour carte 4 et 9

Cette fois-ci les résultats sont bien plus concluants.
Pour la carte 4, nous arrivons à atteindre un bien meilleur score qu'avant, passant de -2 à environ -0.1.
Pour la carte 9, le simple fait de parvenir à lancer l'algorithme est une amélioration.

4 DQN

Pour ce TME nous étudions l'algorithme Deep Q-Learning (DQN). Le principe est le même que pour l'algorithme Q-learning mais cette fois au lieu d'utiliser une version tabulaire nous utilisons un réseau de neurones qui va apprendre les Q-values.

Nous avons implémenté la version de DQN avec Target Network et Prioritized Experience Replay. Nous allons réaliser nos expériences sur les environnements `gridworld` et `cartpole` en reprenant la configuration du modèle donnée pour comparer nos résultats.

Nous employons toujours une stratégie ϵ-greedy avec un decay de 0.999 pour favoriser l'exploration.

Modèle

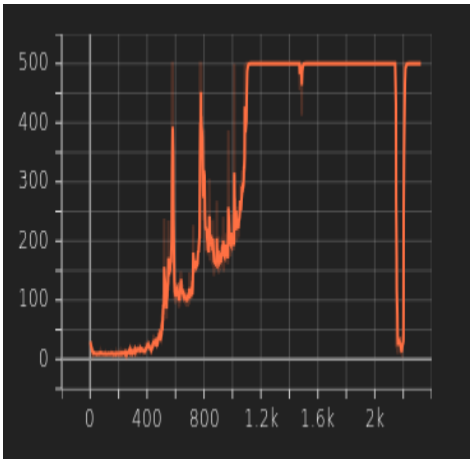
Nous utilisons pour le Q network un réseau à une couche de 200 neurones et une activation tanh avec un optimiseur utilisant l'algorithme ADAM.

Voici les hyperparamètres utilisés après quelques tests réalisés : $\epsilon = 0.05$, $\gamma = 0.99$, $\eta = 0.0001$.

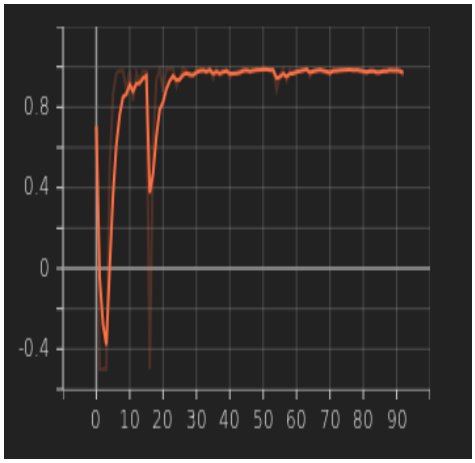
Le buffer est de taille 1000 avec batch size de 128 lors du sampling.
Le target network est mis à jour toutes les 1000 étapes avec une fréquence d'update à 1.

Expériences

Nous testons DQN sur `gridworld` pour la carte 5 et sur `cartpole`.



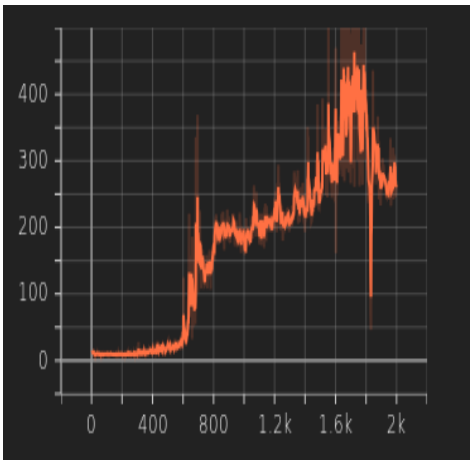
(a) CartPole



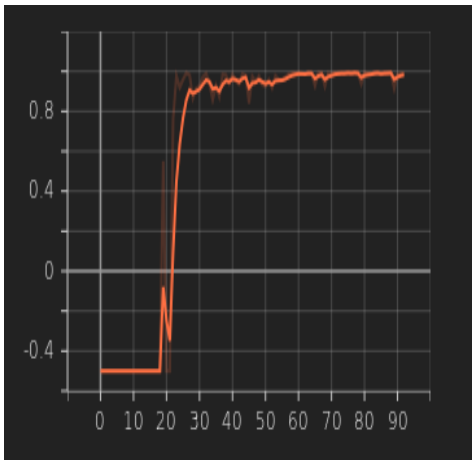
(b) Carte 5

Figure 8: Rewards de DQN avec $\gamma = 0.99$

On peut également essayer de faire diminuer l'hyperparamètre de discount pour tenter d'accélérer la vitesse d'apprentissage



(a) CartPole



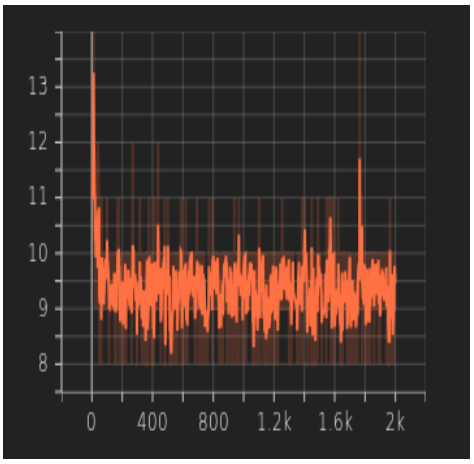
(b) Carte 5

Figure 9: Rewards de DQN avec $\gamma = 0.88$

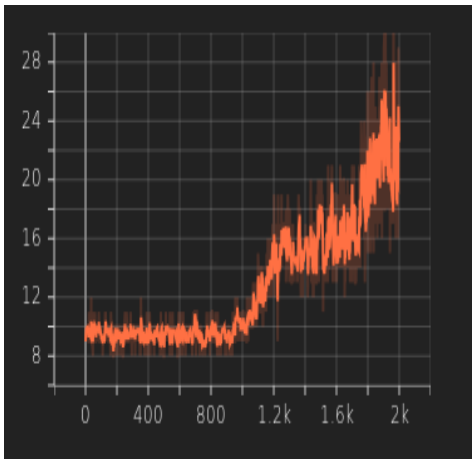
On remarque avec d'autres tests qu'il faut conserver un discount quand même relativement proche de 1 pour obtenir de très bons résultats.

Nous allons également comparer l'utilisation du target network et du Prioritized experience replay sur `cartpole`. Plus précisément, nous comparons différentes configurations de l'algorithme DQN : sans Target Network ni PER,

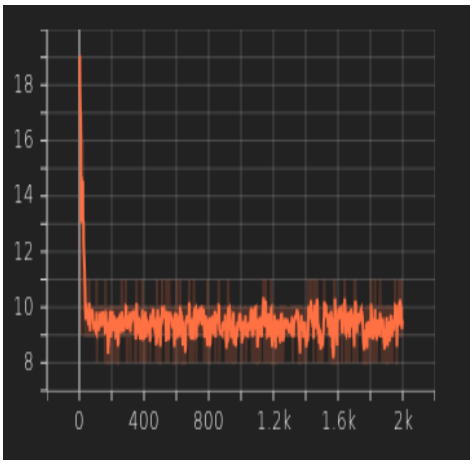
avec Target Network et sans PER, sans Target Network et avec PER, et enfin avec Target Network et PER.



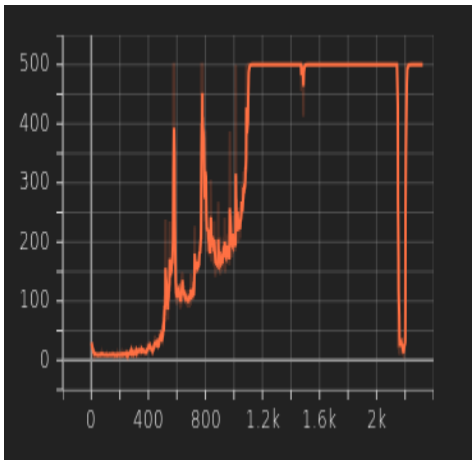
(a) Without TN/PER



(b) TN without PER



(c) PER without TN



(d) With TN/PER

Figure 10: Rewards de DQN selon TN/PER

On observe l'importance de l'utilisation d'un target network et du PER pour obtenir un score élevé rapidement avec une bonne convergence.

5 A2C

Pour ce TME nous étudions l'algorithme Advantage ActorCritic (A2C) et ses variantes sur l'environnement `cartpole`. Il s'agit d'une méthode policy gradient qui va chercher à optimiser les paramètres de l'acteur par montée de gradient.

Les méthodes actor-critic sont à la jointure des approches policy based et value based. L'algorithme A2C entraîne deux réseaux de neurones :

1. Le réseau *actor* qui correspond à la politique π
2. Le réseau *critic* dédié à l'apprentissage des valeurs des états

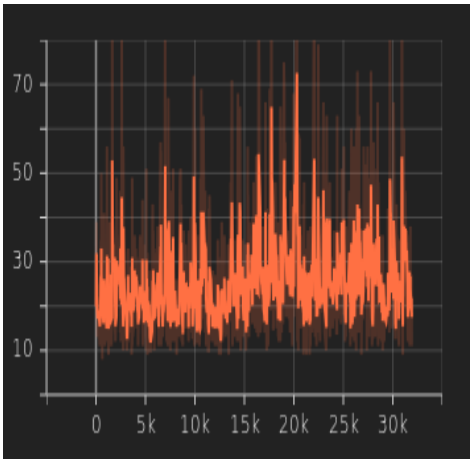
Modèle

Nous utilisons pour l'actor et le critic un réseau à 2 couches comportant 64 neurones chacune avec activation tanh. L'optimiseur est ADAM.

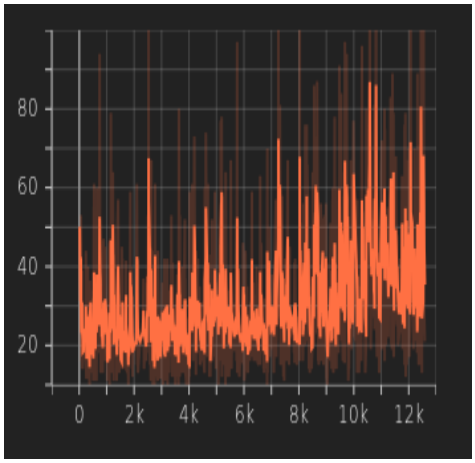
Voici les hyperparamètres utilisés : $\gamma = 0.99$,$\lambda = 0.99$, $\eta_{critic} = 0.0001$, $\eta_{actor} = 0.0003$

Expériences

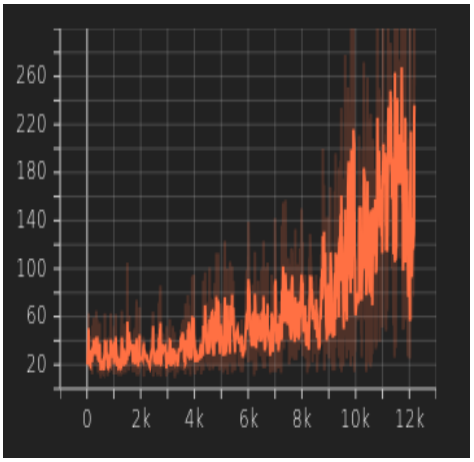
Nous allons présenter les résultats de notre implémentation sur l'environnement `cartpole`.
Nous comparons également les différents version avec GAE en modifiant le paramètre λ
Rappel : $\lambda = 0$ donne TD(0) et $\lambda = 1$ donne Monte-Carlo.



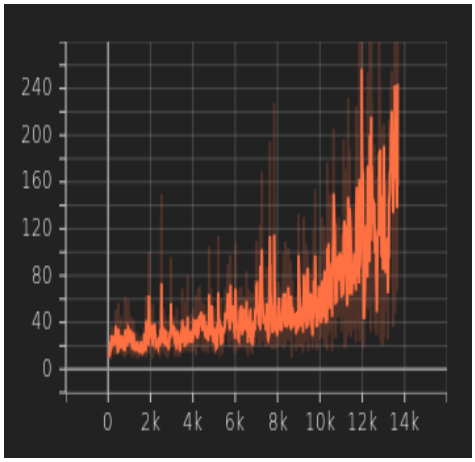
(a) $\lambda = 0$



(b) $\lambda = 0.5$

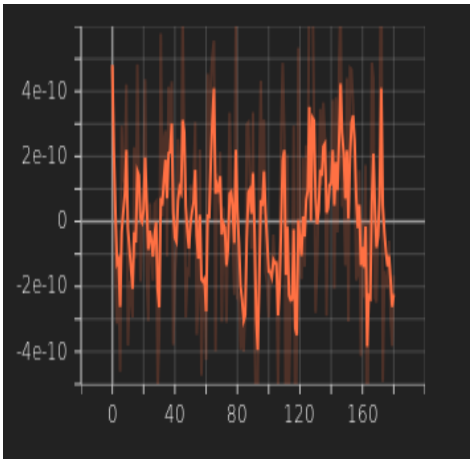


(c) $\lambda = 0.95$

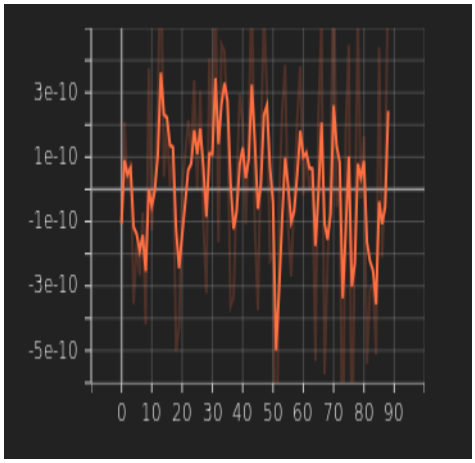


(d) $\lambda = 1$

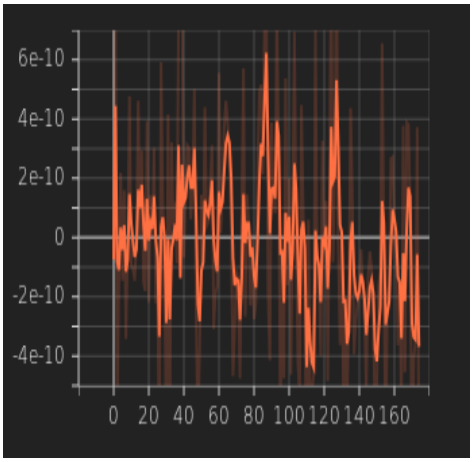
Figure 11: Rewards de A2C avec différents λ



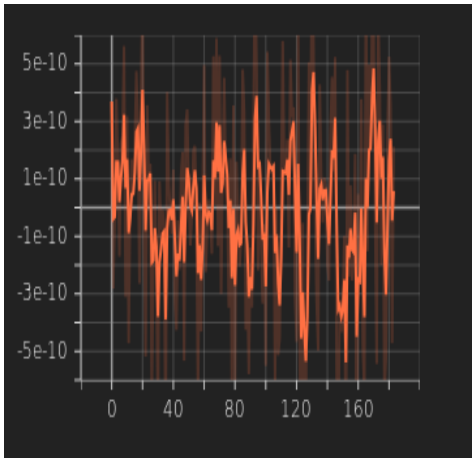
(a) $\lambda = 0$



(b) $\lambda = 0.5$



(c) $\lambda = 0.95$

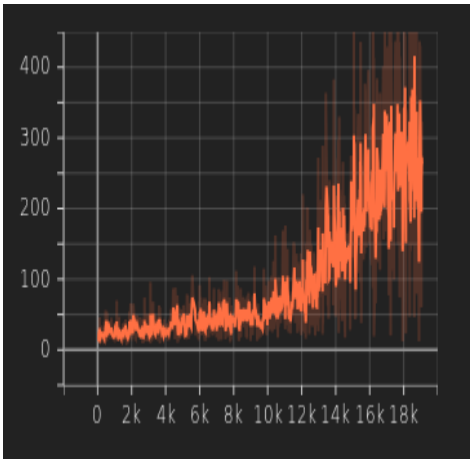


(d) $\lambda = 1$

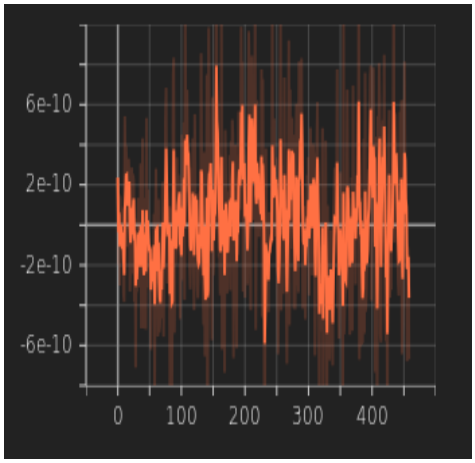
Figure 12: KL divergence de A2C avec différents λ

On remarque que pour des valeurs de λ proches de 1 nous obtenons un algorithme qui apprend contrairement à ceux utilisant un $\lambda \in [0, 0.5]$.

Cependant en analysant la variation de la politique à l'aide de la divergence KL nous constatons de grosses variations de la politique lors de l'apprentissage, peu importe le λ utilisé, ce qui pénalise l'amélioration de notre score. Cela sera corrigé avec les prochains algorithmes.



(a) Rewards



(b) KL divergence

Figure 13: A2C avec $\lambda = 0.99$

6 PPO

Pour ce TME nous étudions l'algorithme PPO et ses variantes sur l'environnement `cartpole`. Il s'agit d'une méthode policy gradient qui va chercher à optimiser les paramètres de l'acteur par montée de gradient tout en contrôlant la déviation de la politique.
Les deux variantes principales sont :

1. PPO KL : version pénalisée avec un coût KL adaptatif dont le coefficient augmente lorsque les changements de politique sont trop brutaux et diminue lorsqu'ils sont trop faibles ;

2. PPO Clipped : une version alternative sans terme KL mais dont l'objectif est tronqué afin de limiter les variations d'une trajectoire à l'autre.

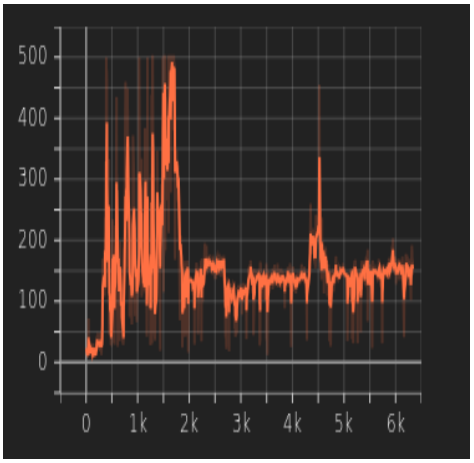
Modèle

nous utilisons pour l'actor et le critic un réseau à 2 couches comportant 64 neurones chacune avec activation tanh. L'optimiseur est ADAM.

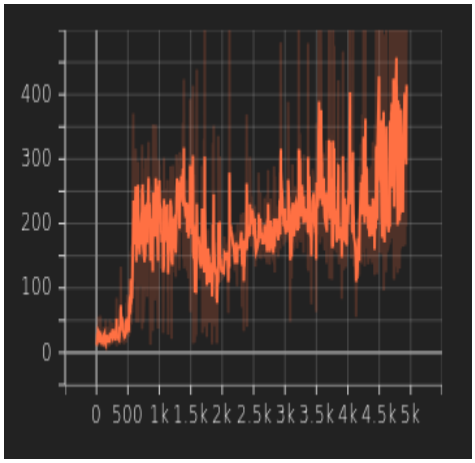
Voici les hyperparamètres utilisés : $\gamma = 0.99$,$\lambda = 0.95$, $\eta_{critic} = 0.0001$, $\eta_{actor} = 0.0003$, $K = 80$, $\epsilon = 0.2$, $\delta = 0.01$

Expériences

Nous allons présenter les résultats de notre implémentation sur l'environnement `cartpole` pour les versions KL et Clipped de PPO.

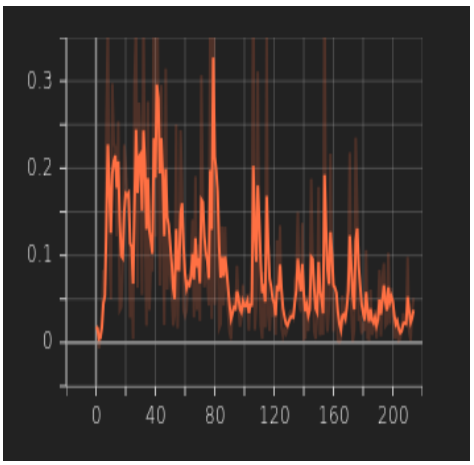


(a) Version KL

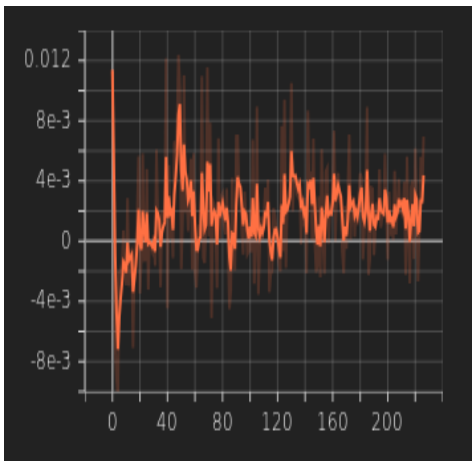


(b) Version Clipped

Figure 14: Rewards cumulés pour PPO



(a) Version KL



(b) Version Clipped

Figure 15: Divergence KL pour PPO

On remarque pour la version KL que l'algorithme décroche vers 1700 épisodes. Nous avons une préférence pour la version Clipped de PPO.

Dans les deux versions la divergence KL est tout proche de 0, c'est-à-dire que les variations de la politique ne sont plus aussi brusques que pour A2C.

Remarque : la version KL utilisée est la version *reverse*, la version *forward* n'ayant pas pu être implémentée à cause d'une erreur issue de la fonction

`KLDivLoss` de PyTorch. Bien que la divergence KL ne soit pas tout à fait symétrique, il a été montré que les deux versions ont des comportements similaires du fait des propriétés de régularisation de l'objectif dans PPO.

7 DDPG

Dans toute la suite, les algorithmes seront appliqués à des problèmes à actions continues. Les algorithmes abordés précédemment peuvent s'étendre à ce contexte.

Un tel exemple est DDPG (Deep Deterministic Policy Gradient), une extension de l'algorithme DQN vu précédemment.

Comme nous sommes dans des espaces à actions continues, il n'est plus possible de calculer les Q-values pour chaque action afin de choisir la meilleure. La fonction $Q(s, a)$ est donc supposée différentiable par rapport à l'action a. Il est dès lors posible de définir une politique $\mu(s)$ pour approximer $max_a Q(s, a)$ par $Q(s, \mu(s))$.

DDPG est un algorithme off-policy qui exploite les mécanismes d'Experience Replay et Target Network de DQN, tout en suivant les principes de DPG pour limiter la variance des trajectoires.

Modèle

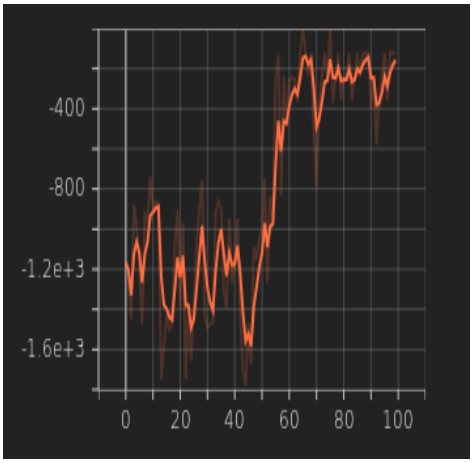
Nous utilisons pour l'actor et le critic un réseau à 2 couches comportant 128 neurones chacune avec activation ReLu sur les couches cachées et tanh sur la couche finale. L'optimiseur est ADAM.

Voici les hyperparamètres utilisés : $\gamma = 0.99$, $\rho = 0.995$, $\eta_{critic} = 0.001$, $\eta_{actor} = 0.001$, ϵ issue d'un processus de Ornstein-Uhlenbeck.

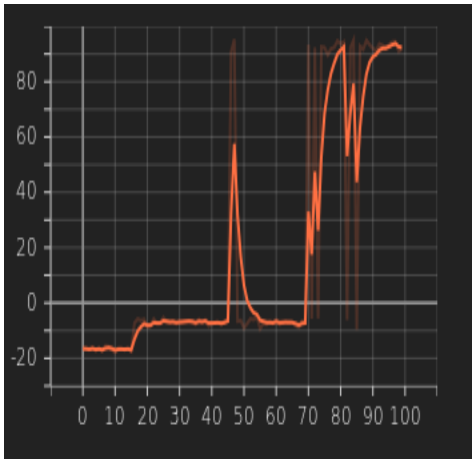
Pour le stockage nous utilisons un buffer de taille 1.000.000 et un batch size de 100 pour le sampling.

Expériences

Nous allons présenter les résultats de notre implémentation sur les environnements `Pendulum` et `MountainCar`.



(a) Pendulum



(b) MoutainCar

Figure 16: Rewards de l'algorithme DDPG

Pour `Pendulum`, l'algorithme parvient à se stabiliser au bout de 80 épisodes aux alentours d'un score de -200. Cet environnement n'étant pas résolu, nous ne disposons pas de seuil de référence. Toutefois, le score obtenu nous semble satisfaisant.
L'environnement `MountainCar` est plus compliqué, car il est facile pour un agent de s'enfermer dans un minimum local. L'exploration est cruciale dans ce cas. Nous parvenons tout de même à atteindre la barre des 80 au bout de 80 épisodes.

8 SAC

Pour ce TME nous étudions l'algorithme SAC (Soft-Actor Critic) pour les environnements à actions continues.

A l'instar de DDPG, il s'agit d'un algorithme off-policy qui exploite les mécanismes d'Experience Replay et Target Network. SAC rajoute un terme d'entropie à la fonction de reward pour encourager l'exploration. L'objectif est d'aboutir à une politique agissant de manière aussi aléatoire que possible et qui vienne à bout de la tâche.

Les deux variantes sont :

- Version à température fixe : le paramètre α est laissé constant ; plus simple à implémenter, mais potentiellement moins bonne car il est difficile de régler α pour avoir un réel contrôle sur le niveau d'entropie tout au long du processus d'apprentissage.
- Version à température adaptative : le paramètre α est le coefficient de Lagrange d'un problème d'optimisation sous contrainte dépendant d'un seuil d'entropie

Nous nous sommes concentrés sur la version à température fixe.

Modèle

Nous utilisons pour l'actor et le critic un réseau à 2 couches comportant 256 neurones chacune avec activation ReLu sur les couches cachées et tanh sur la couche finale. L'optimiseur est ADAM.

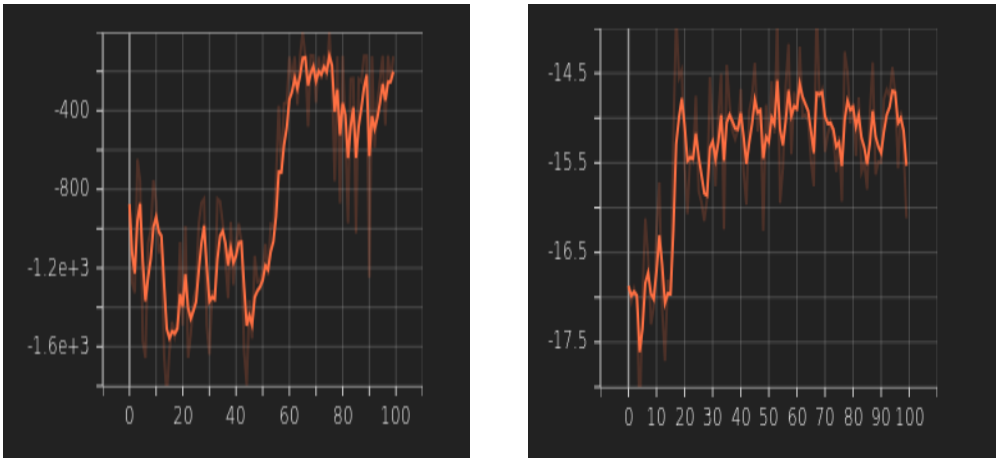
Voici les hyperparamètres utilisés : $\gamma = 0.99$, $\rho = 0.995$, $\eta_{critic} = 0.001$, $\eta_{actor} = 0.001$, $\alpha = 0.2$.

Pour le stockage nous utilisons un buffer de taille 1.000.000 et un batch size de 100 pour le sampling.

Pour créer la distribution des actions à l'état s_t qui suit une loi $N(\mu_{\phi(s_t)}, \sigma^2_{\phi(s_t)})$ nous employons deux couches linéaires prenant comme entrée la sortie intermédiaire de l'actor pour calculer $\mu_{\phi(s_t)}$ et $\sigma_{\phi(s_t)}$ que nous avons lissé avec une fonction `Softplus`. Nous pouvons alors échantillonner nos actions avec la fonction `rsample()`.

Expériences

Nous allons présenter les résultats de notre implémentation sur les environnements `Pendulum` et `MountainCar`.



(a) Pendulum (b) MountainCar

Figure 17: Rewards de l'algorithme SAC

Pour `Pendulum`, l'algorithme parvient, à l'instar de DDPG, à se stabiliser aux alentours d'un score aux alentours de -200.

Nous constatons qu'il réussit beaucoup moins bien sur `MountainCar` que DDPG. C'est un environnement compliqué où une politique agissant de manière aussi aléatoire que possible n'est pas favorisée pour obtenir un bon score. Nous suspectons que le choix du paramètre α est primordial pour obtenir de bons résultats.