



TME 1 à 8 RLD

Auteur

Youva ADDAD

Enseignant

Sylvain LAMPRIER

Nicolas BASKIOTIS

Table des matières

1	Problèmes de Bandits	3
1.1	Baseline :	3
1.2	UCB :	3
1.3	LinUCB :	4
2	Policy Iteration et Value Iteration :	5
2.1	Policy Iteration :	5
2.1.1	Plan 1 :	6
2.1.2	Plan 3 :	8
2.1.3	Plan 5 :	9
2.2	Value Iteration :	10
2.2.1	Plan 1 :	11
2.2.2	Plan 3 :	11
2.2.3	Plan 5 :	12
3	Q-Learning/SARSA/Dyna-Q :	12
3.1	Plan 1 :	12
3.2	Plan 5 :	13
4	DQN/Double DQN/Dueling-DQN/Noisy-DQN :	14
4.1	DQN :	15
4.1.1	Cartpole :	15
4.1.2	Lunar Lander :	16
4.2	Dueling-DQN :	16
4.2.1	Cartpole :	17
4.2.2	Lunar Lander :	17
4.3	Noisy DQN :	17
4.3.1	Cartpole :	18
4.3.2	Lunar Lander :	18
5	Policy Gradients/A2C :	18
5.1	Meilleur Optimiseur :	19
5.2	Reward To Go :	19
5.2.1	Cartpole :	19
5.2.2	Lunar Lander :	20
5.3	Generalized Advantage Estimation :	20
5.3.1	Cartpole :	20
5.3.2	Lunar Lander :	20
6	Advanced Policy Gradients/PPO :	20
6.1	PPO avec un KL pénalité Adaptative :	21
6.1.1	Cartpole :	21
6.1.2	Lunar Lander :	23
6.1.3	Compare $\bar{D}_{KL}(\theta_k \mid \theta)$ et $\bar{D}_{KL}(\theta \mid \theta_k)$:	24
6.2	PPO avec Clipped Surrogate Objective :	24
6.2.1	Cartpole :	24
6.2.2	Lunar Lander :	25
6.2.3	Comparison des Surrogate Objectives	25
7	Continuous Actions/Deep Deterministic Policy Gradient (DDPG) :	25
7.1	Pendulum :	26
7.2	Lunar Lander Continuous :	26
7.3	Mountain Car Continuous :	27

8	Continuous Actions/Soft Actor-Critic (SAC) :	27
8.1	SAC version α fixer :	28
8.1.1	Pendulum :	28
8.1.2	Lunar Lendar :	29
8.2	SAC α Adaptative :	29
8.2.1	Pendulum :	29
8.2.2	Lunar Lendar :	29
8.2.3	Montain Car :	30

1 Problèmes de Bandits

Dans le cadre de ce 1^{er} TME nous nous intéressons au problème de Bandit manchot (problème du bandit à K bras), le problème se décrit comme suit : Nous disposons de K machines, soit des variables aléatoires $X_{i,t}$ pour tout $1 \leq i \leq K$, et $t \geq 1$, où l'indice i représente une des K machines, et t représente un pas de temps, nous supposerons les rewards obtenus pour chaque action sont i.i.d. et suivent une distribution inconnue ν_i d'espérance μ_i , afin de maximiser le reward cumulé sur la période d'actions 1...T.

l'objectif de ce TME d'expérimenter les modèles UCB et LinUCB pour la sélection de publicité en ligne, nous disposons d'un fichier contenant les contextes (profils d'articles) et les taux de clic sur les publicités de 10 annonceurs pour 5000 articles. S'agissant d'un problème de bandit, donc ici les machines sont les annonceurs, une période est un article, les récompenses sont les taux de clics et le but est de maximiser le taux de clics cumulés sur les 5000 visites.

1.1 Baseline :

Afin de résoudre ce problème nous considérons 3 baselines :

- **Stratégie Random** : $\sum_{t=1}^T \text{taux}_{t,\text{annonceur}_i}$ avec annonceur_i un choix aléatoire entre annonceur_1 à annonceur_{10} .
- **Stratégie StaticBest** : let $i = \arg \max_i \text{cumsum } \text{taux}_{\text{annonceur}_i}$, le reward sera égal à $\sum_{t=1}^T \text{taux}_{t,\text{annonceur}_i}$.
- **Stratégie Optimale** : $\sum_{t=1}^T \max_{\text{taux}_i} \text{taux}_{t,\text{annonceur}_i}$

Comme nous pouvions l'attendre, la stratégie random donne le regret le plus élevé et l'optimale donne 0 regret et le maximum de reward cumulé, de ce fait nous avons borner notre reward ou il ne pourrait pas faire pire que le random et ne pourrait pas faire mieux que l'optimale.

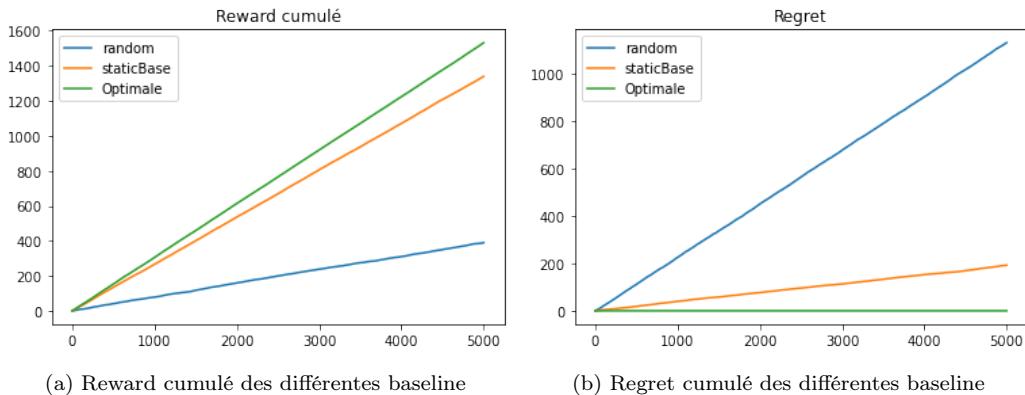


FIGURE 1 – Reward et Regret cumulé des baselines

1.2 UCB :

L'un des problèmes des méthodes gloutonne qui consiste à prendre l'argmax à chaque pas de temps est qu'il n'y a pas d'exploration, et il y'a un fort risque de rester bloqué sur un bras sous-optimal.

La sélection d'action d'UCB utilise l'incertitude dans les estimations de la valeur d'action pour équilibrer l'exploration et l'exploitation, à chaque tour, nous tirons simplement le bras qui a l'estimation de récompense empirique la plus élevée jusqu'à ce point plus un terme inversement proportionnel au nombre de fois que le bras a été joué.

$$A_t = \operatorname{argmax}_a \left(\hat{\mu}_t(a) + c \sqrt{\frac{2 \times \ln(t)}{N_t(a)}} \right) \quad (1)$$

$\hat{\mu}_t(a)$ représente l'estimation courante de l'action a au temps t et donc le terme d'exploitation.
 $c \sqrt{\frac{2 \times \ln(t)}{N_t(a)}}$ représente un terme d'exploration.

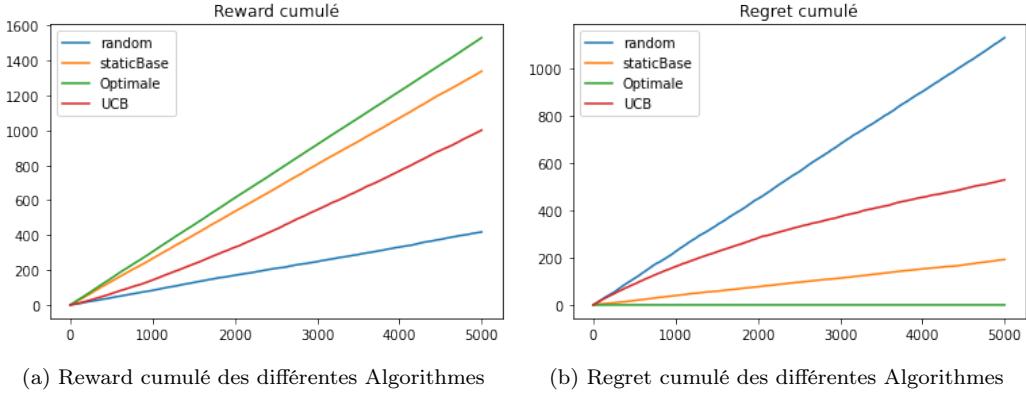


FIGURE 2 – Reward et Regret cumulé des Algorithmes

1.3 LinUCB :

L'algorithme UCB ne prend pas en compte les caractéristiques de l'utilisateur et du contenu (contexte) qui peuvent inclure les activités historiques de l'utilisateur à un niveau agrégé ainsi que les informations démographiques déclarées.

Avec LinUCB, le gain attendu d'un bras est supposé être linéaire dans son vecteur de caractéristiques de dimension de x avec un certain vecteur de coefficient θ inconnu. Une UCB doit être calculée pour chaque bras pour que l'algorithme puisse choisir un bras à chaque essai. La stratégie de choix du bras à chaque essai t est formalisée par :

$$a_t = \arg \max_{a \in \mathcal{A}_t} \left(\mathbf{x}_{t,a}^\top \hat{\boldsymbol{\theta}}_a + \alpha \sqrt{\mathbf{x}_{t,a}^\top \mathbf{A}_a^{-1} \mathbf{x}_{t,a}} \right) \quad (2)$$

Et la mise à jour ce fait de cette façon :

$$\begin{aligned} \mathbf{A}_{at} &\leftarrow \mathbf{A}_{at} + \mathbf{x}_{t,at} \mathbf{x}_{t,at}^\top \\ \mathbf{b}_{at} &\leftarrow \mathbf{b}_{at} + r_t \mathbf{x}_{t,at} \end{aligned} \quad (3)$$

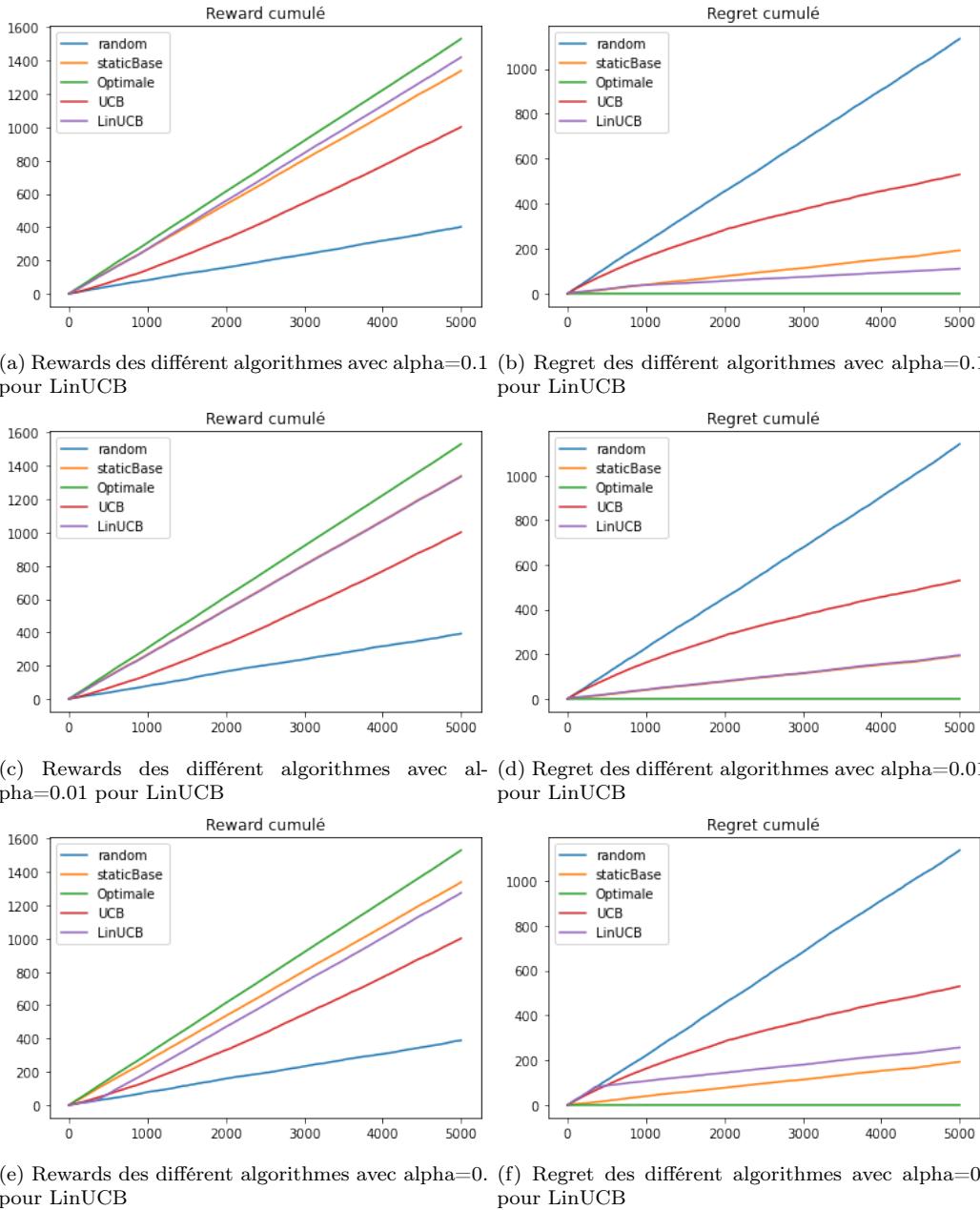


FIGURE 3 – Reward et Regret cumulé des Algorithmes avec différent alpha pour LinUCB

2 Policy Iteration et Value Iteration :

Dans ce deuxième TME nous allons expérimenté les algorithmes policy iteration et value iteration qui sont de type programmation dynamique. L'environnement est un MDP fini, ses ensembles d'état, d'action et de récompense sont aussi finis.

— **Rewards** : +1 sur la case diamant (état final), -1 sur la case feu (état final).

— **Actions** : Nord, Est, Sud, Ouest, pas de déplacement si mur.

Nous allons donc essaier de maximiser le reward cumulé dans les deux cas en utilisant la formule ci-dessus :

$$V_{i+1}^\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a, s) \sum_{s'} \mathcal{P}(s, a, s') (\mathcal{R}(s, a, s') + \gamma V_i^\pi(s')) \quad (4)$$

2.1 Policy Iteration :

L'algorithme Policy Iteration se fait en deux étapes, l'étape de l'évaluation de politique, et l'étape de l'itération de la politique.

Parce qu'un MDP fini n'a qu'un nombre fini de politiques, ce processus doit converger vers une politique opti-

male et une fonction de valeur optimale en un nombre fini d'itérations.

plan	Reward	Time
0	0.957	0.0121
1	1.988	0.0318
2	1.984	0.154
3	0.993	0.026
4	-2.001	0.4619
5	1.935	0.882
6	1.94	4.26
7	1.002	5.58
8	0.926	4.636
10	0.954	0.144

2.1.1 Plan 1 :

Nous avons testé notre algorithmes de policy iteration avec différent gamma (le discount)

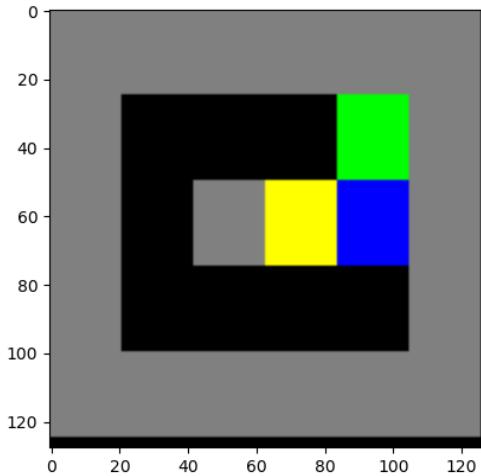


FIGURE 4 – Le plan 1 de gridworld

Dans cette expérimentation nous lancons à chaque épisode un step de policy evaluation et policy iteration, ce qui a permis de générer les différentes courbes ci-dessus.

Nous pouvons remarquer qu'avec un discount de 0. il reste sur un minimum locale égal ~ 0.5 , il regarde juste le reward immédiat. Avec un discount égal à 1. policy iteration pourrait ce retrouver à l'infini parcequ'il n'arrive pas à satisfaire la contrainte suivante : $\|V_i^{\pi_k} - V_{i-1}^{\pi_k}\| \leq \epsilon$.

Avec la valeur de 0.1 il oscille entre 1.90 et 1.85, par contre avec un 0.99 il arrive parfaitement à atteindre le reward maximum (À noter qu'il atteindrait le maximum avec 4 mouvement donc un reward maximum de 1.996).

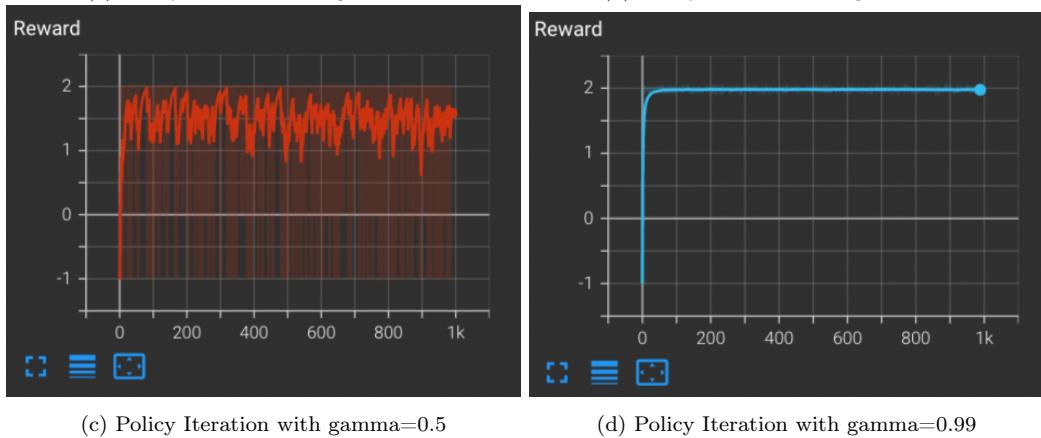
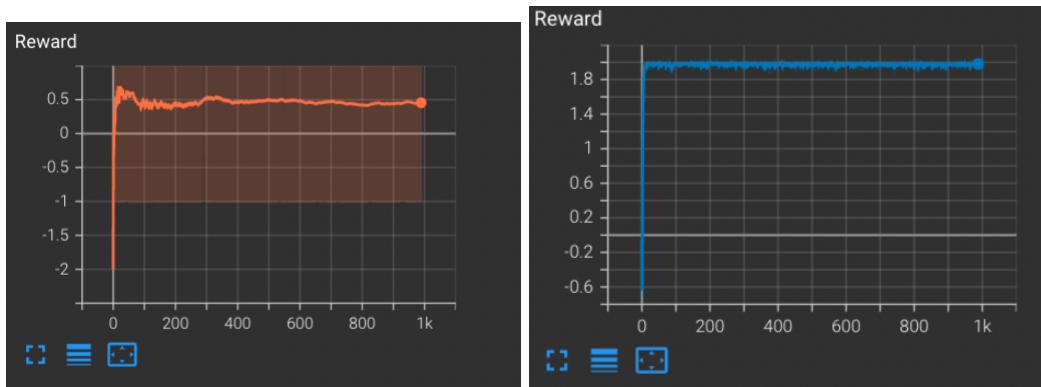
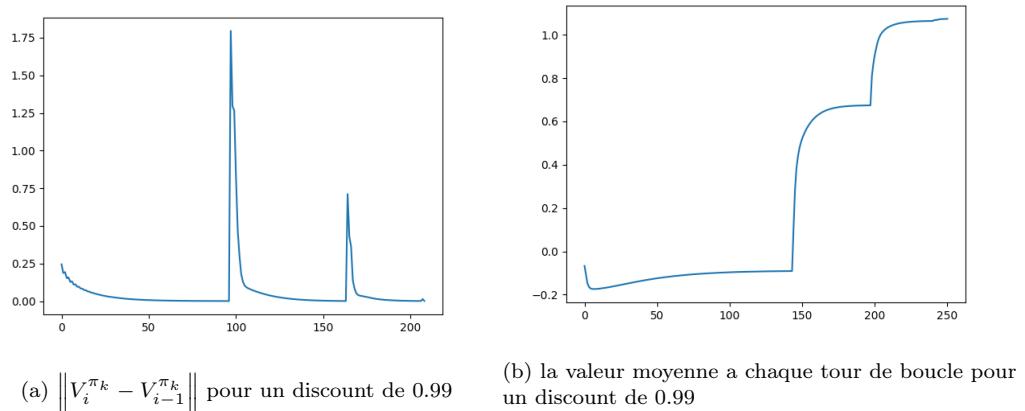


FIGURE 5 – Policy Iteration with different values of gamma



2.1.2 Plan 3 :

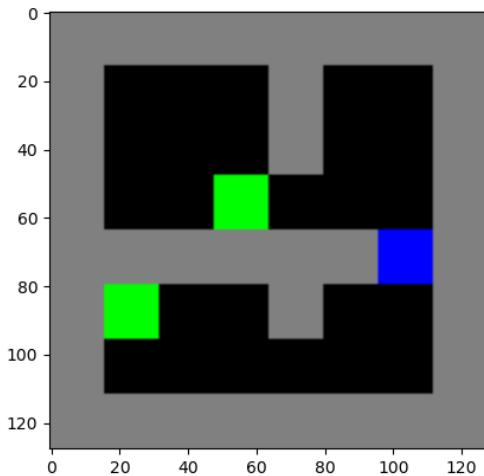


FIGURE 7 – Le plan 3 de gridworld

Nous pouvons faire les mêmes remarque que précédemment.

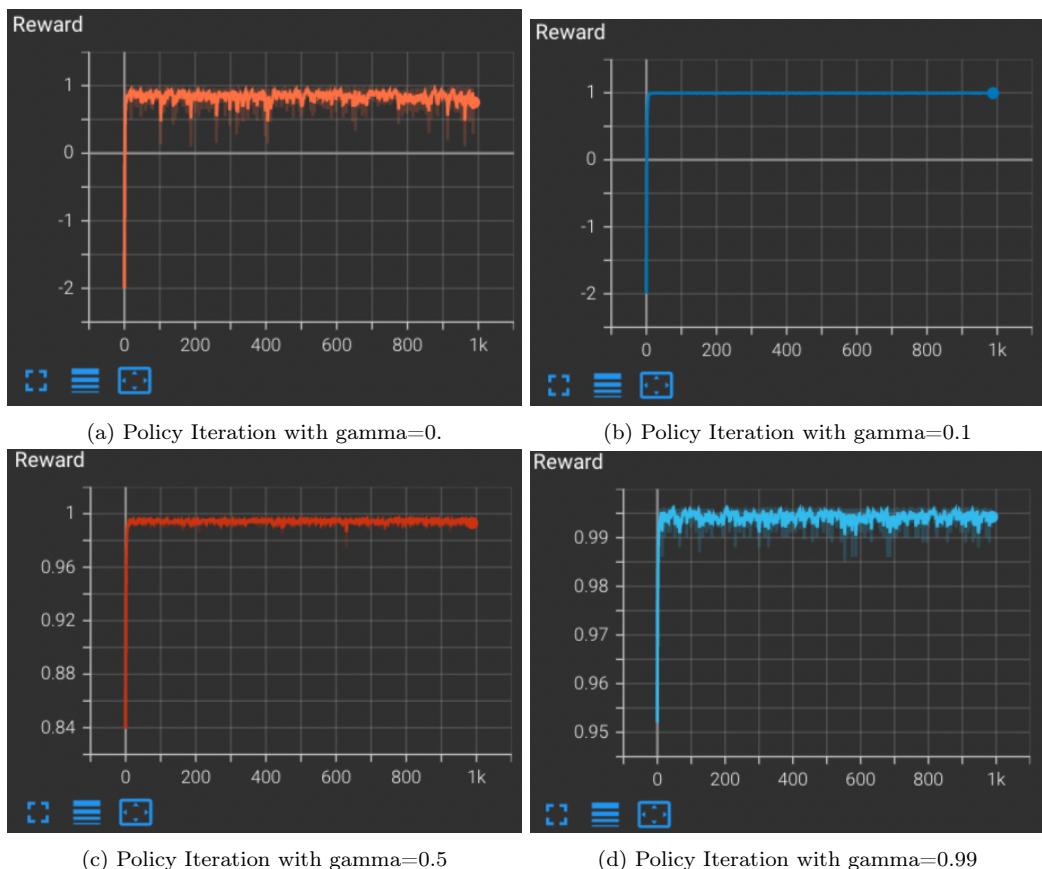
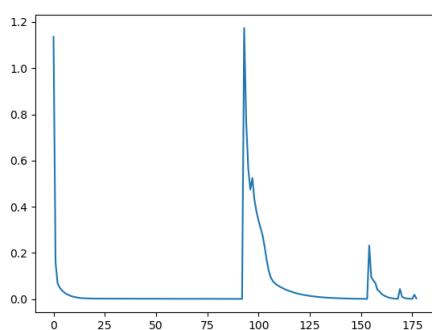
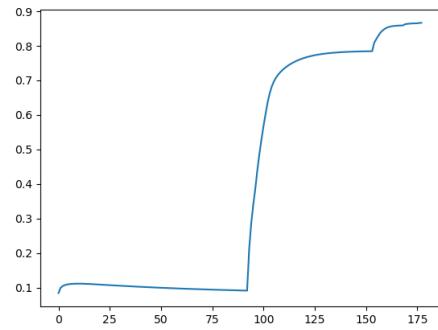


FIGURE 8 – Policy Iteration with different values of γ



(a) $\|V_i^{\pi_k} - V_{i-1}^{\pi_k}\|$ pour un discount de 0.99



(b) la valeur moyenne à chaque tour de boucle pour un discount de 0.99

2.1.3 Plan 5 :

Le plan 5 est certainement le meilleur exemple pour bien visualiser l'impact du discount, ici nous remarquons que si nous prenons le reward immédiat, le reward cumulé est au dessus du zéro, avec un discount de 0.5 il tombe sur un minimum local de valeur ~ 0.9 , mais avec un discount de 0.99 nous pouvons atteindre ~ 1.95

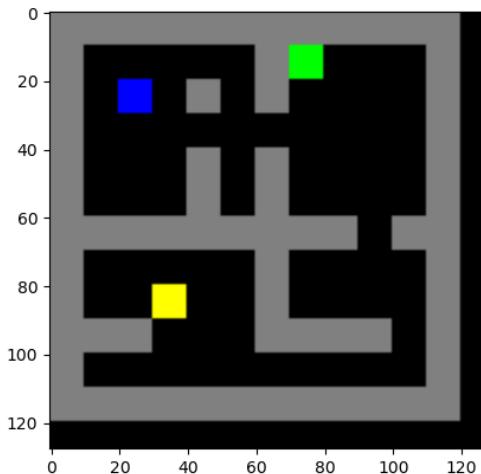


FIGURE 10 – Le plan 5 de gridworld

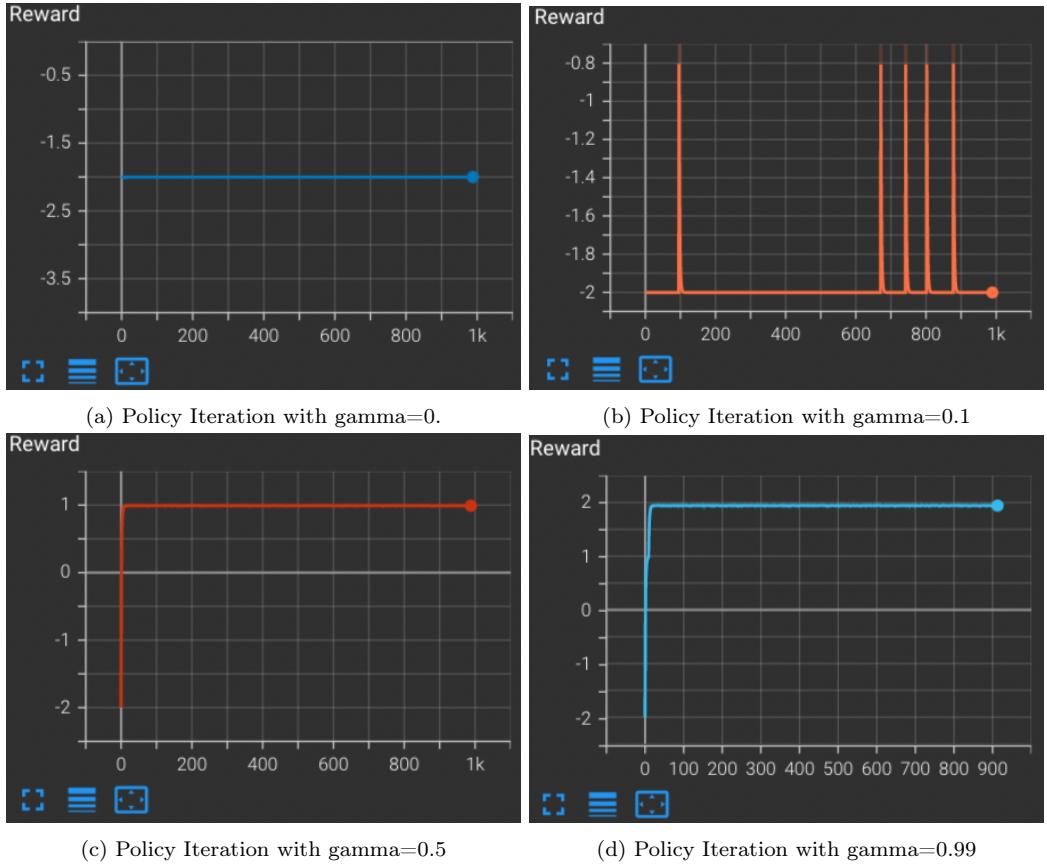
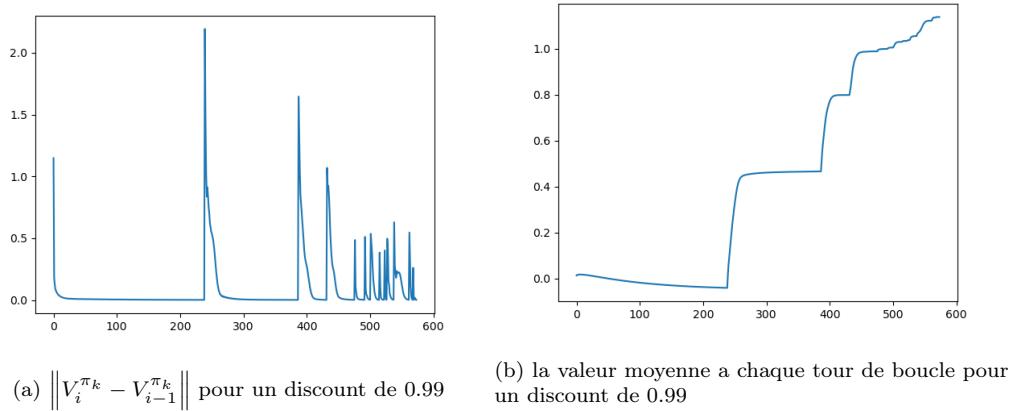


FIGURE 11 – Policy Iteration with different values of gamma



2.2 Value Iteration :

L'algorithme Policy iteration converge de manière certaine vers la politique stationnaire optimale en un temps fini mais à chaque itération, évaluation complète de la politique qui est coûteuse pour de grands graphes (nombre de passage par état intermédiaire pour arriver à l'état final), la solution value iteration nous calculons la fonction de valeur d'état optimale en mettant à jour itérativement l'estimation $V(s)$.

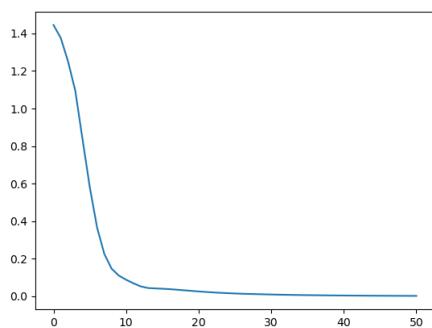
L'étape de mise à jour est très similaire à l'étape de mise à jour de l'algorithme de policy iteration. La seule différence est qu'avec value iteration nous prenons le maximum sur toutes les actions possibles, au lieu d'évaluer puis d'améliorer, l'algorithme value iteration met à jour la fonction de valeur d'état en une seule étape. Et pour les mêmes raisons que policy iteration, value iteration est garanti pour converger vers les valeurs optimales.

plan	Reward	Time
0	0.977	0.0069
1	1.972	0.015
2	1.982	0.047
3	0.995	0.0055
4	-2.001	0.255
5	1.939	0.19
6	1.94	0.74
7	1.944	2.035
8	0.938	1.28
10	0.959	0.039

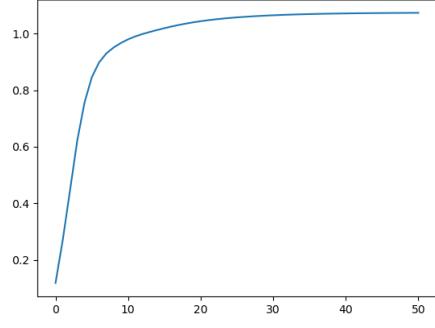
Nous pouvons remarqué que value iteration a fait aussi bon que policy iteration avec un temps plus rapide que pour policy.

2.2.1 Plan 1 :

Beaucoup plus smooth que policy iteration, et même résultat que celui-ci

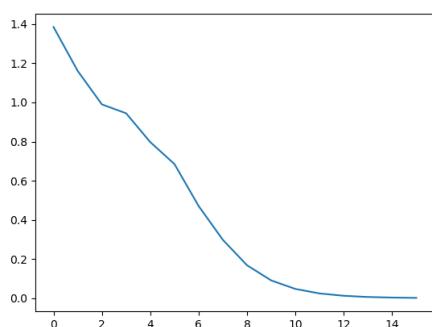


(a) $\|V_i^{\pi_k} - V_{i-1}^{\pi_k}\|$ pour un discount de 0.99

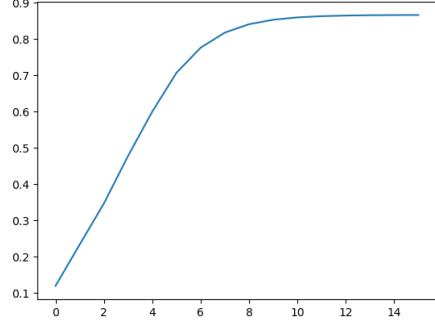


(b) la valeur moyenne à chaque tour de boucle pour un discount de 0.99

2.2.2 Plan 3 :

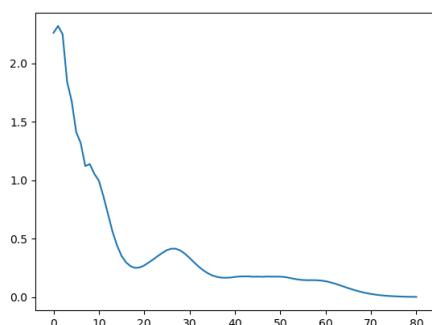


(a) $\|V_i^{\pi_k} - V_{i-1}^{\pi_k}\|$ pour un discount de 0.99

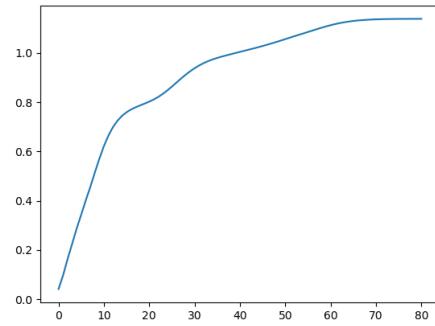


(b) la valeur moyenne à chaque tour de boucle pour un discount de 0.99

2.2.3 Plan 5 :



(a) $\|V_i^{\pi_k} - V_{i-1}^{\pi_k}\|$ pour un discount de 0.99



(b) la valeur moyenne à chaque tour de boucle pour un discount de 0.99

3 Q-Learning/SARSA/Dyna-Q :

Dans cette partie nous allons implémenté les algorithmes Q-Learning et Sarsa, qui l'un est Off-policy l'autre On-policy, et ces deux algorithmes ne nécessitent aucune connaissance du modèle.

Dans les deux méthodes, au cours de chaque épisode, à partir d'un état courant s , nous effectuons une action a de s vers un autre nouvel état s' , en observant une récompense r . L'action a est exécutée conformément à la politique ϵ -greedy actuelle (donnée par la fonction Q value actuelle), UCB ou la sélection de Boltzmann. La mise à jour est faite de la manière suivante :

- **Q-learning :** Cela se fait en choisissant action a en utilisant les politiques citer précédemment (ϵ -greedy par exemple). ce qui donne la mise à jour suivante : $Q(s, a) = Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
- **SARSA :** Cela se fait en choisissant une autre action a' en suivant la même politique citer précédemment et utilisé $r + \gamma Q(s', a')$ comme target. Donc on aura la mise à jour suivante :

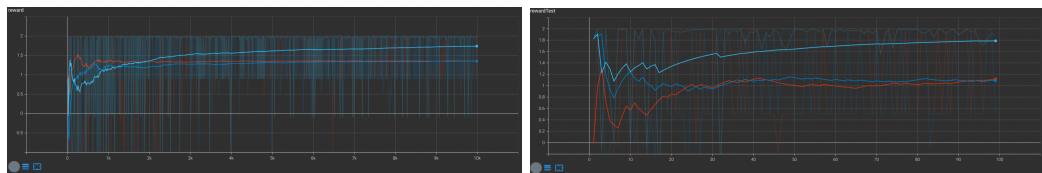
$$Q(s, a) = Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$$

En ce qui concerne l'algorithme Dyna-Q, c'est un algorithme hybrid Model-based / Value-based ou nous avons 2 étapes, l'application du Q-learning standard, mise à jour du MDP, et ensuite une étape de planning en samplant des couples d'actions déjà rencontré et réapplication du Q-learning. À noter nous avons réaliser deux versions de Dyna-Q, la première Dyna-Q fait exactement ce que nous avons mention précédemment, Dyna-Q-Modeled la mise à jour ce fait en utilisant le MDP pour la mise à jour de Q (Probabilité et Reward).

3.1 Plan 1 :

Avec $\epsilon = 0.1$, $\alpha = 0.1$, $\alpha_R = 0.5$, $\alpha_P = 0.5$ et un decay=0.999 réaliser a chaque fin de trajectoire , nous allons voir l'effet du γ sur les différents algorithmes.

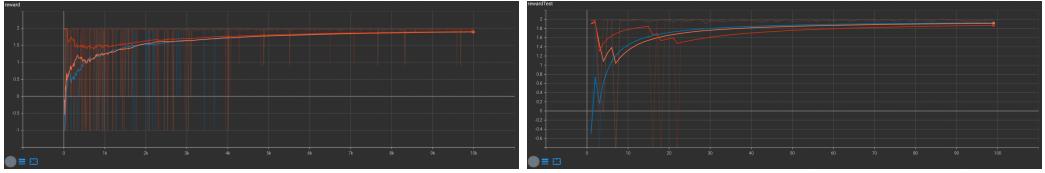
Nous remarquons un net meilleur reward pour Dyna-Q que pour les 2 autres algorithmes.



(a) Reward en train du plan 1 avec Q-Learning en blue, Sarsa en rouge et Dyna en blue ciel
(b) Reward en test du plan 1 avec Q-Learning en blue, Sarsa en rouge et Dyna en blue ciel

FIGURE 16 – Reward en test et en train avec $\gamma = 0.0$

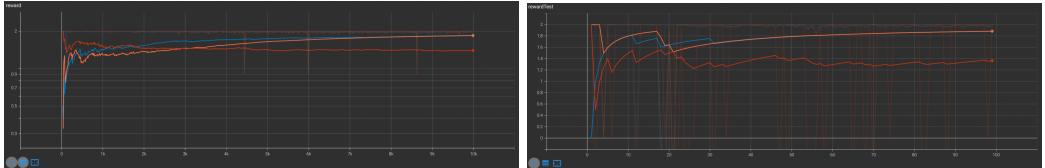
avec $\gamma = 0.1$ les algorithmes arrivent a bien arrivé a l'optimum globale avec des performances semblables pour tous, mais Dyna-Q est beaucoup plus long a la convergence.



(a) Reward en train du plan 1 avec Q-Learning en orange, Sarsa en blue et Dyna en rouge
(b) Reward en test du plan 1 avec Q-Learning en orange, Sarsa en blue et Dyna en rouge

FIGURE 17 – Reward en test et en train avec $\gamma = 0.1$

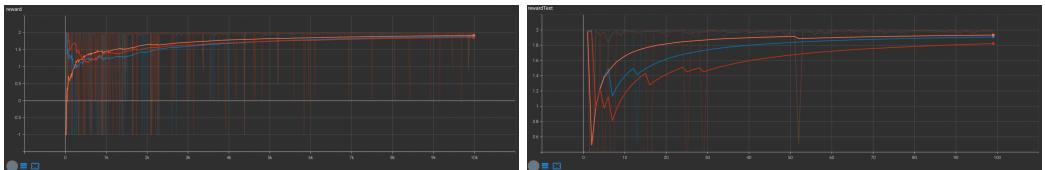
avec $\gamma = 0.5$ Dyna-Q n'atteint pas le maximum du reward sur 10K épisodes alors que Q-Learning et Sarsa ont parfaitement bien atteint l'optimum.



(a) Reward en train du plan 1 avec Q-Learning en orange, Sarsa en blue et Dyna en rouge
(b) Reward en test du plan 1 avec Q-Learning en orange, Sarsa en blue et Dyna en rouge

FIGURE 18 – Reward en test et en train avec $\gamma = 0.5$

Performance égale pour tous les algorithmes avec $\gamma = 0.99$



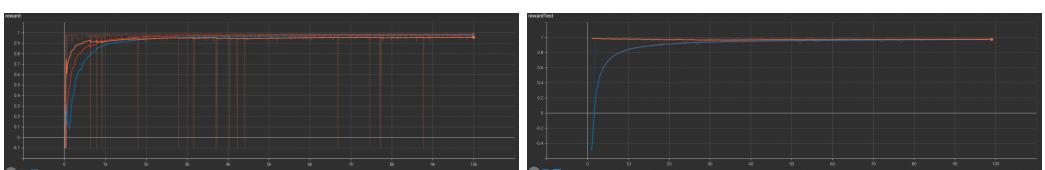
(a) Reward en train du plan 1 avec Q-Learning en orange, Sarsa en blue et Dyna en rouge
(b) Reward en test du plan 1 avec Q-Learning en orange, Sarsa en blue et Dyna en rouge

FIGURE 19 – Reward en test et en train avec $\gamma = 0.99$

3.2 Plan 5 :

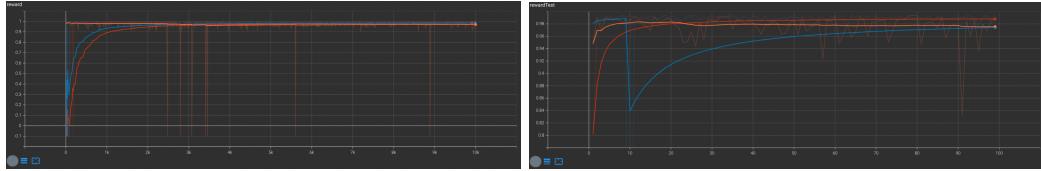
Dans ce plan 5 nous allons non pas regarder les performances en faisant varier γ mais en faisant varier ϵ , le plan 5 a besoin en effet d'une exploration élaborée, nous allons donc tester nos modèles avec des ϵ différent.

Performance égale pour tous les algorithmes avec $\epsilon = 1$. Même avec un epsilon de 1 il n'arrive pas à bien explorer, le reward maximum devrait tourner autour de 1.99, alors que là il est de ~ 1 .



(a) Reward en train du plan 5 avec Q-Learning en blue, Sarsa en rouge et Dyna en orange
(b) Reward en test du plan 5 avec Q-Learning en blue, Sarsa en rouge et Dyna en orange

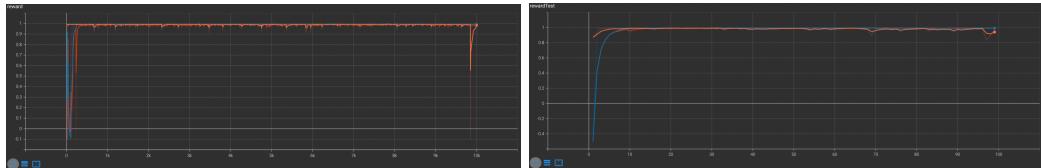
FIGURE 20 – Reward en test et en train avec $\epsilon = 1$.



(a) Reward en train du plan 5 avec Q-Learning en blue, Sarsa en rouge et Dyna en orange
(b) Reward en test du plan 5 avec Q-Learning en blue, Sarsa en rouge et Dyna en orange

FIGURE 21 – Reward en test et en train avec $\epsilon = 0.75$

Nous pouvons faire les mêmes remarques que précédemment.



(a) Reward en train du plan 5 avec Q-Learning en blue, Sarsa en rouge et Dyna en orange
(b) Reward en test du plan 5 avec Q-Learning en blue, Sarsa en rouge et Dyna en orange

FIGURE 22 – Reward en test et en train avec $\epsilon = 0.1$

Nous pouvons donc constater que sans une bonne méthode d'exploration élaborée, nous ne pouvons pas atteindre le maximum du reward, nous l'avons vu précédemment, ϵ qu'il soit égal à 0.1 ou à 1 nous n'atteignons pas le global (le reward est ~ 0.99).

4 DQN/Double DQN/Dueling-DQN/Noisy-DQN :

Q-learning est un algorithme simple mais assez puissant pour créer un aide-mémoire pour notre agent. Cela aide l'agent à déterminer exactement quelle action effectuer, l'algorithme Q-learning devient rapidement inefficace lorsqu'il s'agit d'un environnement complexe avec diverses possibilités (actions continues...), résultats, et contenant beaucoup d'état et d'actions. Heureusement, en combinant l'approche Q-Learning avec des modèles deep learning, nous pouvons surmonter ce problème, il consiste principalement à construire et à entraîner un réseau de neurones capable d'estimer, pour un état donné, les différentes valeurs Q pour chaque action.

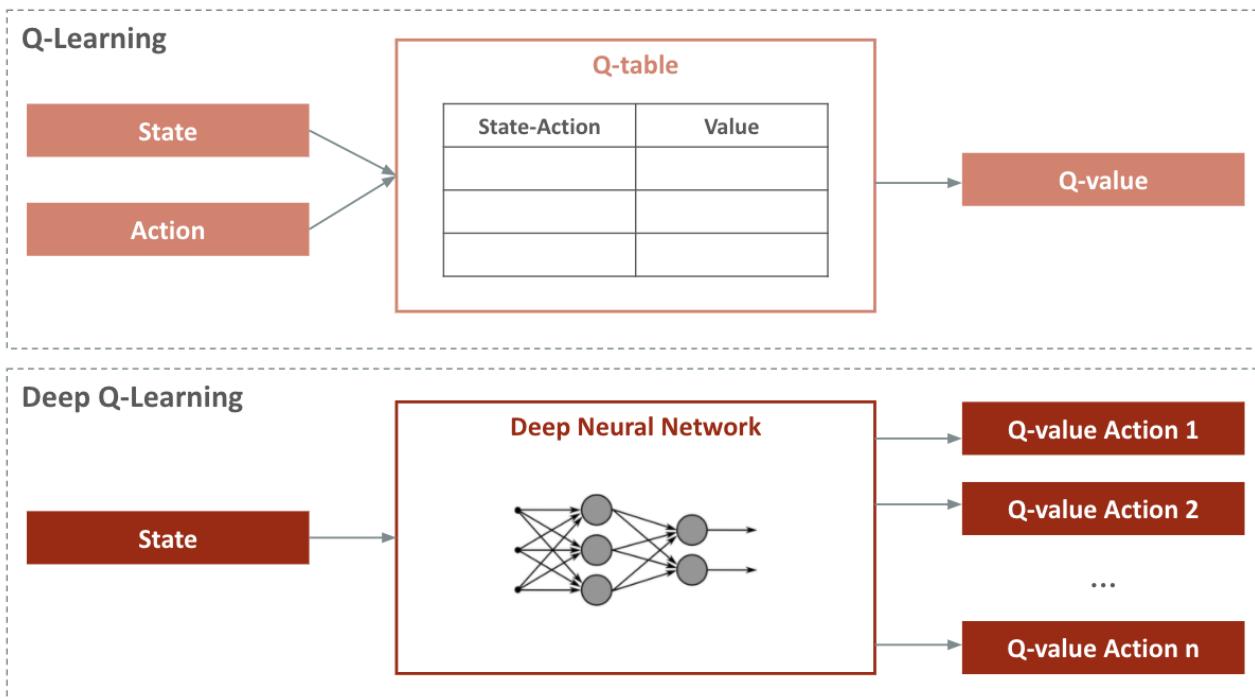


FIGURE 23 – Q-Learning vs DQN

Dans tous les modèles à base de DQN nous essayons de minimiser cette erreur :

$$\text{error} = \underbrace{\left(r_t + \gamma \cdot \max_a Q(s_{t+1}, a | \theta) - Q(s_t, a_t | \theta) \right)^2}_{\text{target}} \quad (5)$$

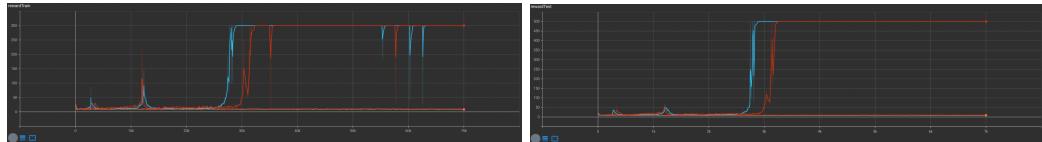
mais comme $y = r_t + \gamma \max_a Q(s_{t+1}, a | \theta)$ utilise lui-même le réseau Q , ce qui rend la tâche instable, et la rend pas vraiment supervisée, afin de pallier à ce problème le rajout d'un target network mis à jour tous les n steps est nécessaire, finalement nous aurons : $y = r_t + \gamma \max_a Q'(s_{t+1}, a | \theta')$ avec Q' est le nouvelle target network et θ' est mis à jour à θ tous les n steps. L'utilisation d'une experience Replay permet de Réduire les corrélations et permet d'accélérer l'apprentissage.

4.1 DQN :

4.1.1 Cartpole :

Target network mis à jour tous les 1000 évènements, une exploration epsilon-greedy (epsilon initial = 0.1, avec decay de 0.99999 à chaque évènement, un discount de 0.999, un pas d'apprentissage de 0.0003, un batch de 100 transitions, un replay buffer de 10000 transitions, un pas d'optimisation tous les 10 évènements, une taille d'épisode maximale de 300 évènements en apprentissage (500 en test) et selon un réseau de neurones à une couche cachée de 200 neurones (avec activation tanh sur la couche cachée), et comparaison entre Experience Replay prioritized ou non, et Target Network ou non.

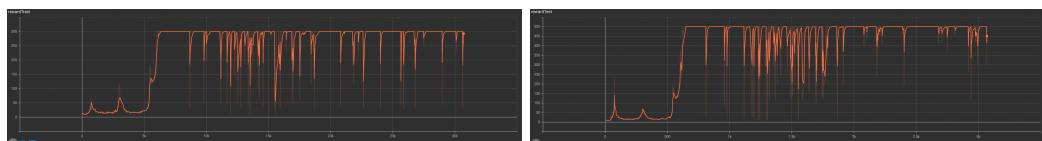
Replay	○	○	✗	✗
Target	○	✗	○	✗
Step Begin convergence	26830	NA	29770	NA
Reward	300	10	300	10



(a) Reward en train de Cartpole avec en orange sans PR ni TN, en bleu avec PR pas TN, en rouge avec PR ni TN, en bleu avec PR pas TN, en rouge avec TN pas PR, bleu ciel avec PR et TN
(b) Reward en test de Cartpole avec en orange sans PR ni TN, en bleu avec PR pas TN, en rouge avec PR ni TN, en bleu avec PR pas TN, en rouge avec TN pas PR, bleu ciel avec PR et TN

FIGURE 24 – Reward en test et en train avec $= 0.0003$

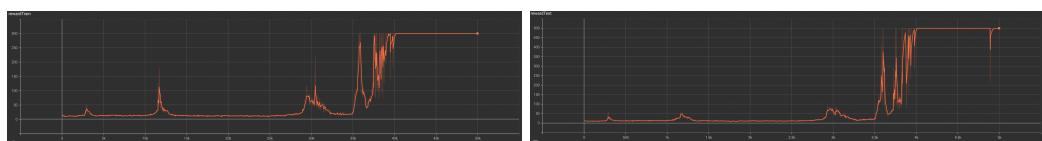
Nous pouvons remarqué qu'avec un lr trop grand le DQN devient instable même si nous attendons le reward maximum rapidement (au bout de 5000 itérations contre 27000 avec un lr=0.0003, il vaudrait mieux prendre un petit lr).



(a) Reward en train de Cartpole avec Prioritized replay et Target network et un lr=0.001
(b) Reward en test de Cartpole avec Prioritized replay et Target network et un lr=0.001

FIGURE 25 – Reward en test et en train avec $\text{explo}=0.1$ et $= 0.001$

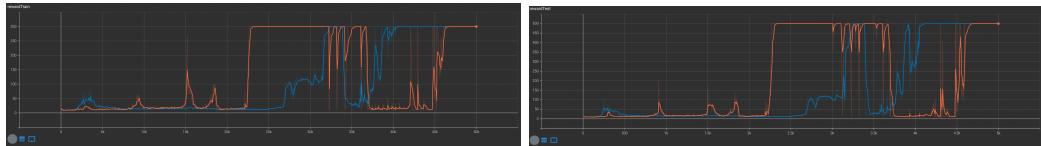
Nous remarquons qu'avec une exploration de 1. et un decay exponential de 0.9999 à chaque évènement que la convergence est beaucoup plus lente (40000 steps).



(a) Reward en train de Cartpole avec Prioritized replay et Target network
(b) Reward en test de Cartpole avec Prioritized replay et Target network

FIGURE 26 – Reward en test et en train avec $lr = 0.0003$ et $\text{explo}=1$.

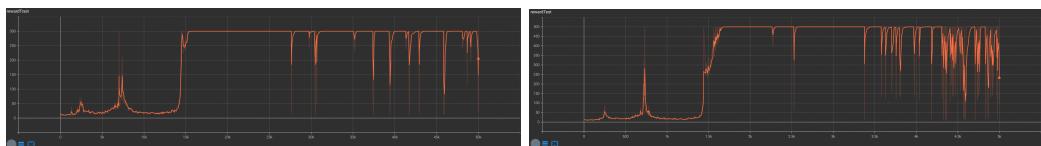
Nous pouvons remarqué que plus un buffer a une grande capacité plus il est long à converger mais après convergence il reste stable et n'oscille pas (100K), contrairement à 1K qui est beaucoup plus instable.



(a) Reward en train de Cartpole avec Prioritized replay et Target network, avec un buffer de capacité 100K en bleu et de 1K en orange
(b) Reward en test de Cartpole avec Prioritized replay et Target network, avec un buffer de capacité 100K en bleu et de 1K en orange

FIGURE 27 – Reward en test et en train avec $lr = 0.0003$, $\text{explo}=0.1$ et 100K vs 1K pour la capacité du buffer

Nous remarquons que le reward devient instable avec une mise à jour tout les 5000 steps.



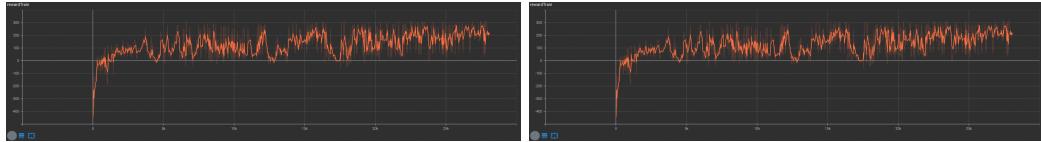
(a) Reward en train de Cartpole avec Prioritized replay et Target network, avec une mise à jour chaque 5000 steps
(b) Reward en test de Cartpole avec Prioritized replay et Target network, avec une mise à jour chaque 5000 steps

FIGURE 28 – Reward en test et en train avec $lr = 0.003$, $\text{explo}=0.1$ et une mise à jour de target chaque 5000 steps

Nous pouvons dire que les paramètres initiaux nous permettent une meilleure convergence et une meilleure stabilité.

4.1.2 Lunar Lander :

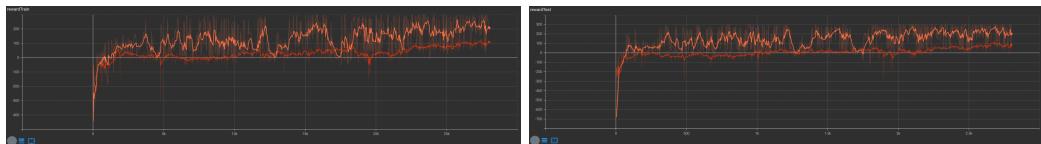
Avec les mêmes paramètres que précédemment, et un lr qui égal à 0.001 et sans decay d' ϵ , un résultat plutôt satisfaisant mais reste tout de même instable.



(a) Reward en train de Lunar avec Prioritized replay, Target network et un LR=0.001 et sans decay
(b) Reward en test de Lunar avec Prioritized replay, Target network et un LR=0.001 et sans decay

FIGURE 29 – Reward en test et en train avec $lr = 0.001$, $\text{explo}=0.1$ sans decay

Convergence un peu plus lente avec l'utilisation du decay, mais beaucoup plus stable comparé sans, avec plus d'épisode il arrivera sûrement à faire aussi bien tout en étant stable.



(a) Reward en train de Lunar avec Prioritized replay, Target network et un LR=0.001 et avec decay en rouge et sans en orange
(b) Reward en test de Lunar avec Prioritized replay, Target network et un LR=0.001 et avec decay en rouge et sans en orange

FIGURE 30 – Reward en test et en train avec $lr = 0.001$, $\text{explo}=0.1$ sans/avec decay

4.2 Dueling-DQN :

Dueling DQN divise les valeurs Q en deux parties différentes, la fonction de valeur $V(s)$ et la fonction d'avantage $A(s, a)$. La fonction de valeur $V(s)$ nous dit combien de récompense nous allons collecter de l'état s .

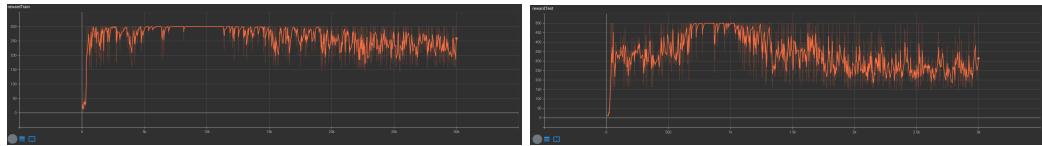
Et la fonction d'avantage $A(s, a)$ nous dit à quel point une action est meilleure par rapport aux autres actions. En combinant la valeur V et l'avantage A pour chaque action, nous pouvons obtenir les valeurs Q :

$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a' \in \mathcal{A}(s)} A(s, a') \quad (6)$$

Nous utilisons la soustraction sur la moyenne de l'avantage afin récupérer V et A de manière unique, dans le cas contraire $Q(s, a) = V(s) + A(s, a)$ n'a pas de solution unique.

4.2.1 Cartpole :

Avec le Dueling DQN nous devons avoir une schedule de decay, nous avons proposé d'utiliser un decay exponentielle en fonction du nombre d'épisode (t) $self.explo = \max(self.max_explo * np.exp(-t * self.decay), self.min_explo)$ en définissant un $explo_min$ qui est égal à 0.0001 et l' $explo_max$ qui est de 1, et donc nous décroissant l'exploration qui initialement était de 1 à chaque événement, tout en ayant gardé les mêmes hyperparamètres que précédemment.



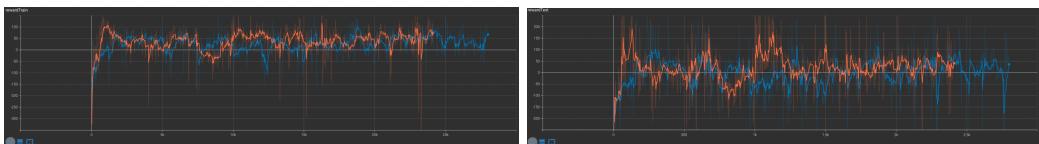
(a) Reward en train de Cartpole avec Prioritized replay, Target network et exponentielle decay
(b) Reward en test de Cartpole avec Prioritized replay, Target network et exponentielle decay

FIGURE 31 – Reward en test et en train avec $lr = 0.0003$, $explo=1$. avec exponentielle decay

4.2.2 Lunar Lander :

Nous avons testé Lunar Lander avec un LR=0.001 et un explo de 1./0.1 et un decay exponentielle de paramètre episode, donc nous décroissant l'exploration en fonction du timestep.

Nous pouvons remarqué que Lunar reste instable et avec ces paramètres le modèle ne fait pas mieux que DQN, et ne converge pas vers le reward maximum.



(a) Reward en train de Lunar avec Prioritized replay, Target network, exponentielle decay et explo égal 1. en orange et 0.1 en blue
(b) Reward en test de Lunar avec Prioritized replay, Target network, exponentielle decay et explo égal 1. en orange et 0.1 en blue

FIGURE 32 – Reward en test et en train avec $lr = 0.001$, $explo=1./0.1$ avec exponentielle decay

4.3 Noisy DQN :

Les précédent modèles utilisent $\epsilon - greedy$ pour sélectionner une actions, alternativement, Noisy DQN le remplace en ajoutant du bruit à la couche linéaire pour faciliter l'exploration, de ce fait nous utilisons pas $\epsilon - greedy$ mais nous prenons l'action maximisant Q .

Consider a linear layer of a neural network with p inputs and q outputs, represented by

$$y = wx + b,$$

where $x \in \mathbb{R}^p$ is the layer input, $w \in \mathbb{R}^{q \times p}$ the weight matrix, and $b \in \mathbb{R}^q$ the bias. The corresponding noisy linear layer is defined as:

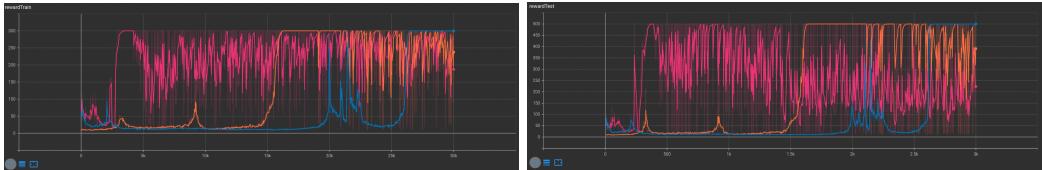
$$y \stackrel{\text{def}}{=} (\mu^w + \sigma^w \odot \varepsilon^w)x + \mu^b + \sigma^b \odot \varepsilon^b,$$

where $\mu^w + \sigma^w \odot \varepsilon^w$ and $\mu^b + \sigma^b \odot \varepsilon^b$ replace w and b in Eq. (8), respectively. The parameters $\mu^w \in \mathbb{R}^{q \times p}$, $\mu^b \in \mathbb{R}^q$, $\sigma^w \in \mathbb{R}^{q \times p}$ and $\sigma^b \in \mathbb{R}^q$ are learnable whereas $\varepsilon^w \in \mathbb{R}^{q \times p}$ and $\varepsilon^b \in \mathbb{R}^q$ are noise random variables. \square

Avec $\sigma = 0.017$ paramètre conseiller dans l'article original, et l'utilisation de la fonction héritée de module afin de spécifier que les paramètres d' ϵ ne sont pas des paramètres entrainable, nous laissant de plus les mêmes paramètres que précédemment et nous feront varier le learning rate.

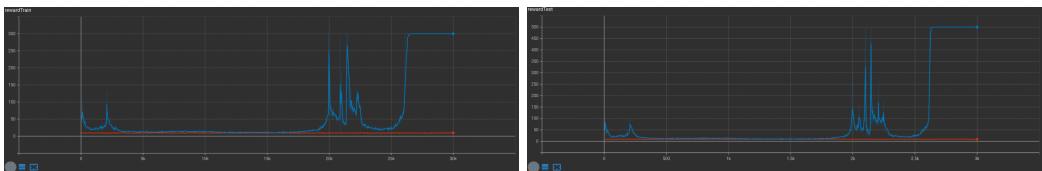
4.3.1 Cartpole :

Nous pouvons remarqué que le noisy dqn est très sensible au learning rate nous avons changé que d'un petit pas, et cela change considérablement le comportement. Mais avec 0.0003 le learning rate initiale il arrive parfaitement bien à ce converger et avec 0.001 il reste constant sur un minimum.



(a) Reward en train de Cartpole avec Prioritized replay, Target network avec un LR de 0.0003 en bleu, play, Target network avec un LR de 0.0003 en bleu, 0.0007 en rose et 0.0004 en orange
(b) Reward en test de Cartpole avec Prioritized replay, Target network avec un LR de 0.0003 en bleu, play, Target network avec un LR de 0.0003 en bleu, 0.0007 en rose et 0.0004 en orange

FIGURE 33 – Reward en test et en train avec $lr = 0.0007/0.0004/0.0003$

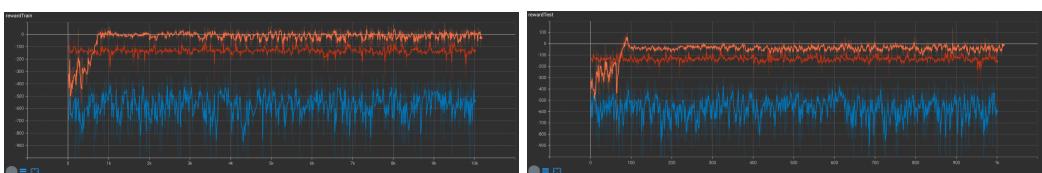


(a) Reward en train de Cartpole avec Prioritized replay, Target network avec un LR de 0.0003 en bleu play, Target network avec un LR de 0.0003 en bleu et 0.001 en rose
(b) Reward en test de Cartpole avec Prioritized replay, Target network avec un LR de 0.0003 en bleu play, Target network avec un LR de 0.0003 en bleu et 0.001 en rose

FIGURE 34 – Reward en test et en train avec $lr = 0.0003/0.001$

4.3.2 Lunar Lander :

Nous avons testé Lunar Lander avec les mêmes paramètres que précédemment, mais on faisant varier le learning rate, le reward n'est pas très bon dans tout les cas mais beaucoup mieux avec 0.0003, De façon générale pour Lunar Lander le DQN arrive à faire mieux que les autres modèles mais reste instable.



(a) Reward en train de Lunar avec Prioritized replay, Target network avec un LR de 0.0003 en orange, 0.0001 en rouge et 0.001 en bleu
(b) Reward en test de Lunar avec Prioritized replay, Target network avec un LR de 0.0003 en orange, 0.0001 en rouge et 0.001 en bleu

FIGURE 35 – Reward en test et en train avec $lr = 0.0003/0.0001/0.001$

5 Policy Gradients/A2C :

Dans les méthodes d'apprentissage par renforcement value-based model-free, la fonction de valeur d'action est représentée à l'aide d'un approximateur de fonction, tel qu'un réseau de neurones. Les méthodes Policy-Gradients s'intéresser directement à la politique, et nous essayerons de trouver les meilleurs paramètres θ . L'objectif est de s'orienter plus probablement vers les trajectoires maximisant les récompenses.

$$\pi_{\theta}(a | s) = p(a | s, \theta) \quad (7)$$

Nous essayerons donc de maximiser $J(\theta) = E \left[\sum_{t=0}^{T-1} r_{t+1} \right]$. Puisqu'il s'agit d'un problème de maximisation, nous optimisons la politique par montées de gradient successives avec la dérivée partielle de l'objectif par rapport

au paramètre de politique θ .

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} J(\theta) \quad (8)$$

Comme $J(\theta)$ correspond à une somme possiblement infinie sur l'ensemble des trajectoires, nous allons utiliser le reinforce trick :

$$\nabla_x f(x) = f(x) \frac{\nabla_x f(x)}{f(x)} = f(x) \nabla_x \log f(x) \quad (9)$$

Nous allons donc procéder a une monter de gradient avec le gradient de $J(\theta)$ comme suit :

$$\nabla_{\theta} J(\theta) = E_{\tau \sim \pi_{\theta}(\tau)} \left[R(\tau) \sum_{t=1}^{|\tau|-1} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \right] \quad (10)$$

- **Actor** : Met à jour la distribution de la politique dans la direction suggérée par le critic (comme avec les gradients de politique). Donc ils apprennent à prendre des décisions.
- **Critic** : Estime la fonction de valeur. Cela pourrait être la valeur d'action (la valeur Q) ou la valeur d'état (la valeur V). Donc ils émettent des avis sur les possibles décisions.

5.1 Meilleur Optimiseur :

Dans cette section nous allons comparer le meilleur optimiseur pour A2C, nous allons utilisé pour tous les optimiseurs le même learning rate qui est de 0.001, un gamma de 0.99, un réseau a 2 couches cachée de 30 chacun, des épisodes maximum de 500 en train et en test, et qui utilise un reward to go $\sum_{l=0}^{\infty} \gamma^l r_{t+l}$, nous avons utilisé un SGD avec un momentum de 0.9, un Adam, un RMSProp avec un alpha de 0.99. Nous avons de plus utilisé une deque afin de simuler le monte carlo Rollout, ou une optimisation est lancée a chaque fin de trajectoire (arriver a done), et nous utilisons une categorielle afin de sampler une action (dans tout cette partie).

Nous pouvons remarqué un légère avantage de RMSProp, Adam s'en sort bien aussi, mais le SGD est instable et oscille beaucoup. Par la suite nous allons préféré RMSProp comme optimiseur.

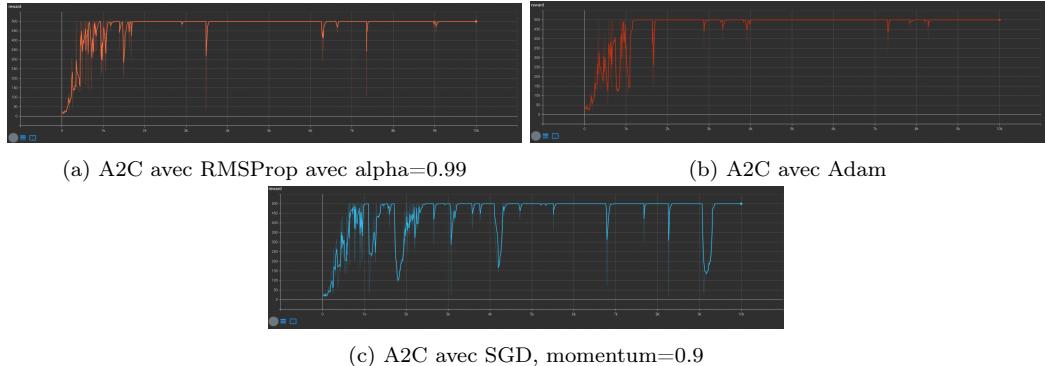


FIGURE 36 – Comparaison entre différent optimiseur sur Cartpole

5.2 Reward To Go :

Nous pouvons remaqué que plus gamma décroît plus A2C devient instable et ne converge pas vraiment vers le reward maximum.

5.2.1 Cartpole :

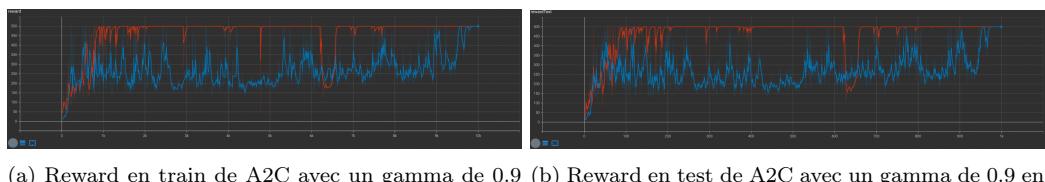


FIGURE 37 – Comparaison entre différent gamma sur Cartpole

5.2.2 Lunar Lander :

Nous avons lancé Lunar Lander avec le RMSProp, un learning rate de 0.001, avec un alpha de 0.99 pour le smoothing, un ϵ de 0.1 pour l'exploration, un γ de 0.99. Le résultat obtenu est convaincant et stable.

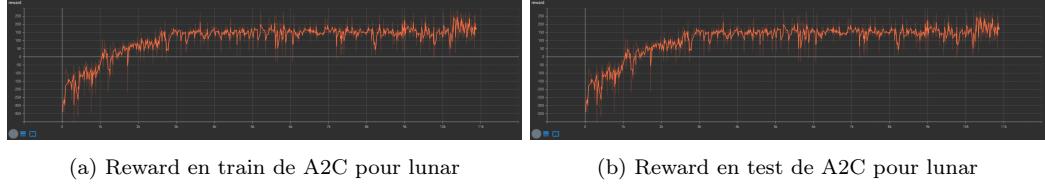


FIGURE 38 – Reward en Train et Test pour Lunar Lander

5.3 Generalized Advantage Estimation :

Nous avons donc rajouté un nouveau modèle qui cette fois utilisant une formule plus générale, qui est une méthode générique de reinforcement learning qui unifie à la fois la simulation Monte Carlo et la méthode TD zéro. En plus de γ , nous rajoutons donc un paramètre λ qui permet de regarder l'horizon.

$$\hat{A}_t^{\text{GAE}(\gamma, \lambda)} = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l}^V \quad (11)$$

$$\begin{aligned} \text{GAE}(\gamma, 0) : \quad \hat{A}_t &= \delta_t &= r_t + \gamma V(s_{t+1}) - V(s_t) \\ \text{GAE}(\gamma, 1) : \quad \hat{A}_t &= \sum_{l=0}^{\infty} \gamma^l \delta_{t+l} &= \sum_{l=0}^{\infty} \gamma^l r_{t+l} - V(s_t) \end{aligned} \quad (12)$$

5.3.1 Cartpole :

Plus λ est grand, plus nous avons un reward plus stable, et qui converge vers la valeur maximum.

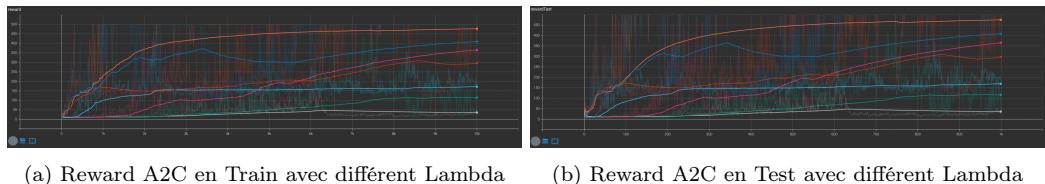


FIGURE 39 – Reward A2C avec λ égal à 0.99 en orange, 0.95 en bleu, 0.9 en rouge, 0.5 en bleu ciel, 0.25 en rouge, 0.1 en vert et 0 en gris

5.3.2 Lunar Lander :

Nous avons tester le modèle sur Lunar Lander avec le GAE, un learning rate de 0.001, RMSProp avec un smooth (alpha) de 0.99, un gamma de 0.99 et un Lambda de 0.99.

Nous pouvons remarqué la stabilisation et la convergence de Lunar, le modèle fait beaucoup mieux que le DQN, mais il reste tout de même oscillant.

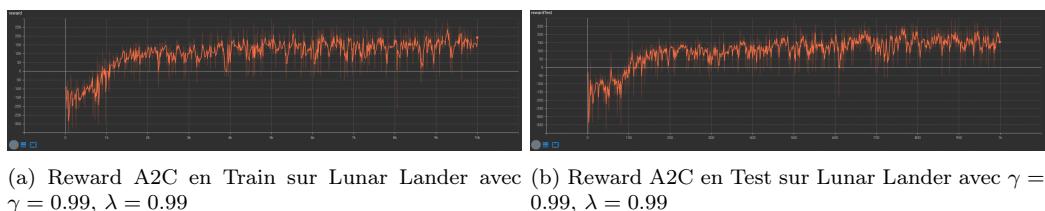


FIGURE 40 – Reward en Train et en Test sur Lunar Lander avec le RMSProp et le GAE $\lambda = 0.99$, et $\gamma = 0.99$

6 Advanced Policy Gradients/PPO :

Les policy gradient sont faiblement sample efficient, de plus si nous nous déplaçons trop vite (α trop grand), nous pouvons effectuer des mouvements catastrophiques, et si nous nous déplaçons trop lentement (α trop petit), nous risquons d'apprendre trop lentement.

Dans cette partie nous allons voir comment limiter les déplacements de la politique pour qu'elle ne varie pas au delà d'un seuil à chaque étape.

Dans TRPO une fonction objectif est maximisé avec une contrainte sur la taille de la mise à jour de la politique.

$$\begin{aligned} & \underset{\theta}{\text{maximize}} \quad \hat{E}_t \left[\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t \right] \\ & \text{subject to} \quad \hat{E}_t [\text{KL}[\pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_{\theta}(\cdot | s_t)]] \leq \delta \end{aligned} \quad (13)$$

Ce qui ce traduit par une nouvelle formule sans contrainte suivante :

$$\underset{\theta}{\text{maximize}} \hat{E}_t \left[\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t - \beta \text{KL}[\pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_{\theta}(\cdot | s_t)] \right] \quad (14)$$

Avec un coefficient β , et θ_{old} les paramètres avant la mise à jour, mais avec TRPO nous devons calculer le gradient naturel afin de résoudre l'équation (11), PPO utilise plutôt l'équation (12 KL adaptative).

Proximal Policy Optimization (PPO) est une famille de méthodes qui appliquent approximativement la contrainte KL sans calculer les gradients naturels, pour cela nous avons deux variantes :

- **PPO avec un KL pénalité Adaptative** : maximize $\hat{E}_t \left[\frac{\pi_{\theta}(a_t | s_t)}{\pi_{\theta_{\text{old}}}(a_t | s_t)} \hat{A}_t \right] - \beta \hat{E}_t [\text{KL}[\pi_{\theta_{\text{old}}}(\cdot | s_t), \pi_{\theta}(\cdot | s_t)]]$, le coefficient de pénalité β change entre les itérations pour appliquer approximativement la contrainte de divergence KL avec θ_{old} et la valeur de θ avant la mise à jour.
- **PPO avec Clipped Surrogate Objective** : $\hat{E}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right]$, où ϵ est un hyperparamètre. Si le ratio de probabilité entre la nouvelle politique et l'ancienne politique tombe en dehors de la plage $(1 - \epsilon)$ et $(1 + \epsilon)$, la fonction d'avantage sera tronquée (borne pessimiste).

6.1 PPO avec un KL pénalité Adaptative :

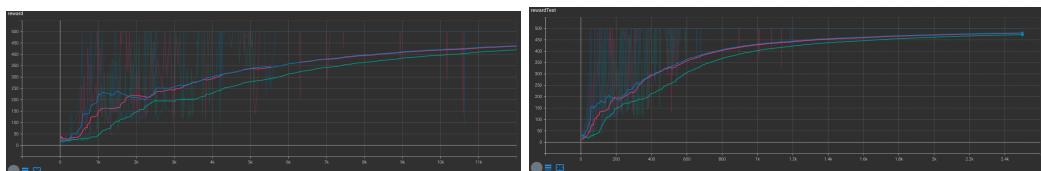
Dans cette version du PPO nous allons garder le $\lambda = 0.99$ et $\gamma = 0.99$ inchangé dans toutes les expérimentation, ainsi que maxLengthTest, maxLengthTrain qui seront à 500, un learning rate de 0.0001 pour l'actor et de 0.0003 pour le critic, un delta cible de 0.5. Nous avons aussi décidé de mettre à jour l'actor et le critic à chaque fin de trajectoire (ou nous prenons tous les exemples du buffer, fin de trajectoire veut dire soit état final soit le maxlen atteint), nous allons donc par la suite faire varier K du nombre de steps d'optimisation et le delta cible.

D'après l'article original les paramètres 1.5 et 2 sont choisis de manière heuristique, mais l'algorithme n'y est pas sensible, la valeur initiale de β est un autre hyperparamètre mais n'a pas d'importance en pratique car l'algorithme l'ajuste rapidement

Au premier passage le gradient de la KL vaut zero parce que $\pi_{\theta_{\text{old}}} = \pi_{\theta}$. L'algorithme PPO ressemblera à A2C si nous prenons K = 1 parce que la KL vaudra zéro comme mentionné précédemment, $\frac{\pi_{\theta}(a|s)}{\pi_{\theta_k}(a|s)} = 1$ et donc qui donnera la formule de maximisation d'A2C sans le terme du log.

6.1.1 Cartpole :

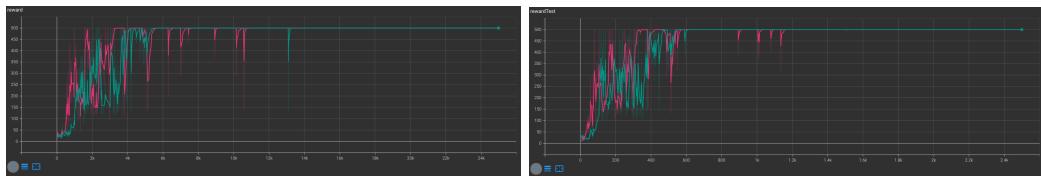
Nous pouvons remarqué que le reward moyen est presque identique pour tous, mais avec k=3 il est plus stable donc plus en augmente K plus nous devenons instable parce que nous mettons à jour à chaque fin de trajectoire.



(a) Reward en Train avec K=10 en bleu, K=5 en rose et K=3 en vert avec delta=0.5
(b) Reward en Test avec K=10 en bleu, K=5 en rose et K=3 en vert

FIGURE 41 – Reward en Train et En Test avec différent step d'optimisation k=3/5/10

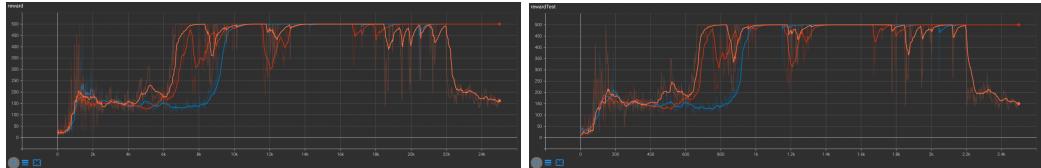
Nous pouvons voir la stabilité de plus près pour K=3 et K=5



(a) Reward en Train avec K=5 en rose et K=3 en vert avec delta=0.5 (b) Reward en Test avec K=5 en rose et K=3 en vert avec delta=0.5

FIGURE 42 – Reward En Train et en Test avec différent step d'optimisation k=3/5

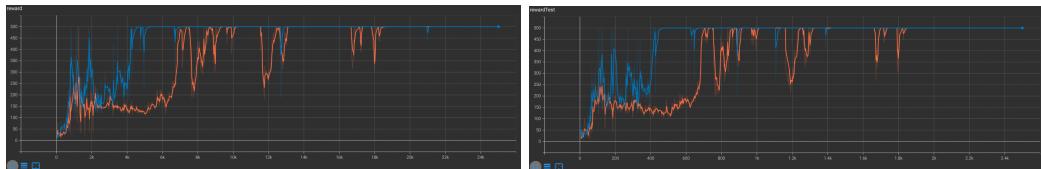
Les différents delta ont fait dans l'ensemble un bon reward et stable sauf pour le cas du 0.01 qui oscille beaucoup et n'arrive pas à ce maintenir.



(a) Reward en Train avec K=3, delta=0.01 en orange, 0.03 en bleu et 0.003 en rouge (b) Reward en Test avec K=3, delta=0.01 en orange, 0.03 en bleu et 0.003 en rouge

FIGURE 43 – Reward En Train et en Test avec différent delta=0.003/0.01/0.03

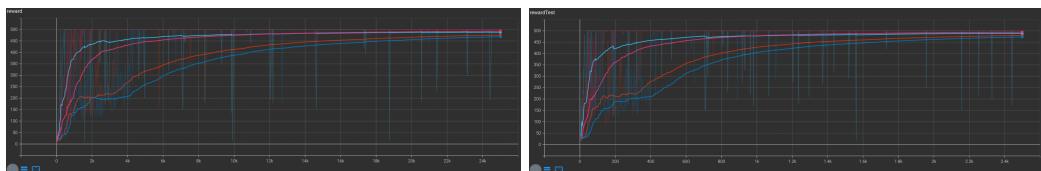
Sans surprise, la convergence est beaucoup plus rapide en rajoutant le K steps optimisation pour la critic, il est de plus encore plus stable.



(a) Reward en Train avec K=3 pour la mise à jour de l'actor, delta=0.003, K=3 en bleu pour la mise à jour de la critic et K=1 en orange (b) Reward en Test avec K=3 pour la mise à jour de l'actor, delta=0.003, K=3 en bleu pour la mise à jour de la critic et K=1 en orange

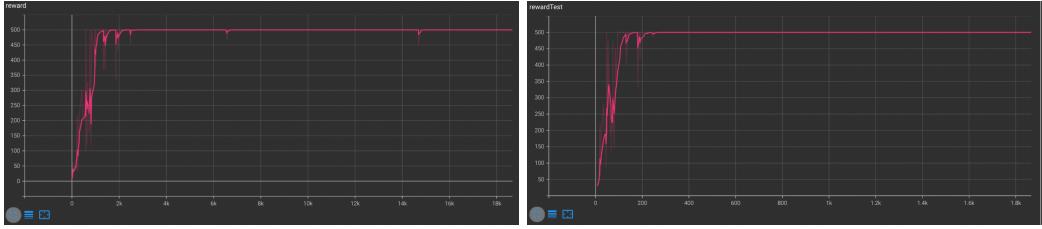
FIGURE 44 – Reward En Train et en Test avec différent steps d'optimisation pour la critic

Avec différent valeur d'optimisation K, l'algorithme converge vers une même valeur pour tous, mais plus le K est grand plus devient oscillant, et nous pouvons le remarquer sur la figure 46, avec un K=10 pour le critic et l'actor le reward devient parfaitement stable et il n'y a quasiement pas d'oscillation une fois convergence atteinte.



(a) Reward en Train avec un delta cible de 0.003, K=3 en bleu, 5 en rouge, 10 rose, 30 blue ciel le (b) Reward en Test avec un delta cible de 0.003, K=3 en bleu, 5 en rouge, 10 rose, 30 blue ciel le nombre d'optimisation de l'actor et du critic d'optimisation de l'actor et du critic

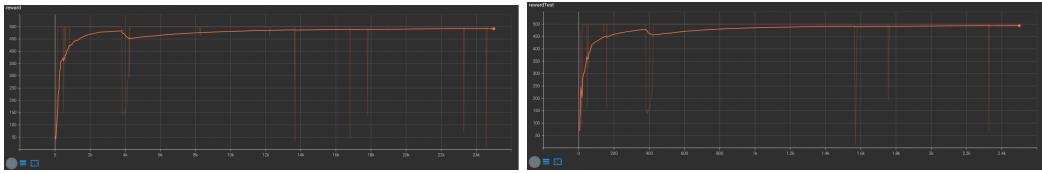
FIGURE 45 – Comparaison Reward en Train et en Test avec différent pas d'optimisation (même que l'actor et le critic)



(a) Reward en Train avec un delta cible de 0.003, (b) Reward en Test avec un delta cible de 0.003, K=10 le nombre d'optimisation de l'actor et du critic K=10 le nombre d'optimisation de l'actor et du critic

FIGURE 46 – Comparaison Reward en Train et en Test pour K=10

Avec le nombre de steps K=10, beta=1, delta=0.003, maxLengthTest=500, maxLengthTrain=500, lr_pi=0.0003, lr_v=0.001, gamma=0.99, lambda=0.97. Nous arrivons à obtenir un bon reward rapidement, donc dans l'ensemble l'algorithme PPO avec KL est très efficace, même si il a une légère sensibilité aux paramètres il arrive tout de même en moyenne à avoir un bon résultat. De plus il est plus stable que A2C et converge plus rapidement.



(a) Reward en Train avec paramètre précédent

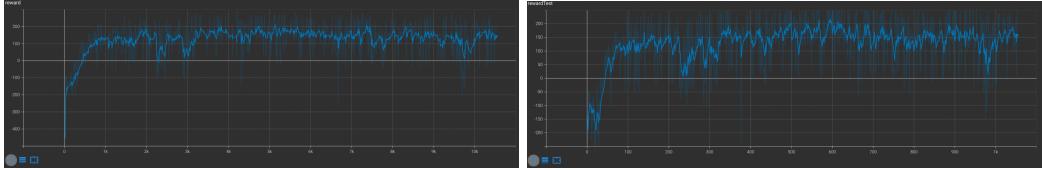
(b) Reward en Test avec paramètre précédent

FIGURE 47 – Comparaison Reward en Train et en Test avec les paramètres précédent

6.1.2 Lunar Lander :

Nous avons testé Lunar Lander avec K=10, beta=1., delta=0.003, maxLengthTest= 500 également en Train, lr_pi = 0.0003, lr_v = 0.001, gamma=0.99, lambda=0.97.

Nous pouvons dire que PPO est beaucoup plus efficace que A2C sur Lunar lunder ou il arrive à atteindre un reward de 300, reste tout de même instable du fait du problème lunaire.

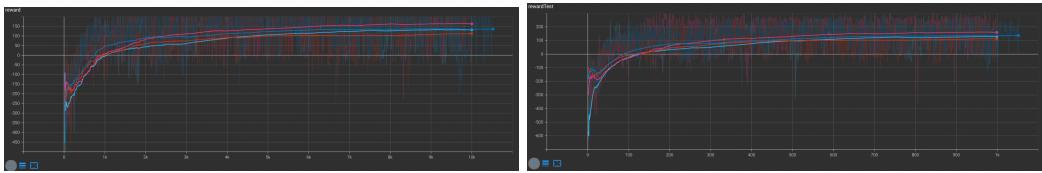


(a) Reward en Train avec paramètre précédent

(b) Reward en Test avec paramètre précédent

FIGURE 48 – Comparaison Reward en Train et en Test avec les paramètres précédent sur Lunar Lander

Nous avons expérimenté les valeurs de beta sur Lunar avec les mêmes hyperparamètres que précédemment, nous avons juste changé les betas, où $\beta \in \{0.3, 1., 3., 10.\}$. Comme nous l'avions souligné précédemment les valeurs de beta n'influe pas sur l'algorithme comme nous pouvons le voir sur la figure (49), les rewards sont quasi-identiques pour tous. PPO arrive à bien ajuster les betas, le reward est acceptable compte tenu de la tâche.



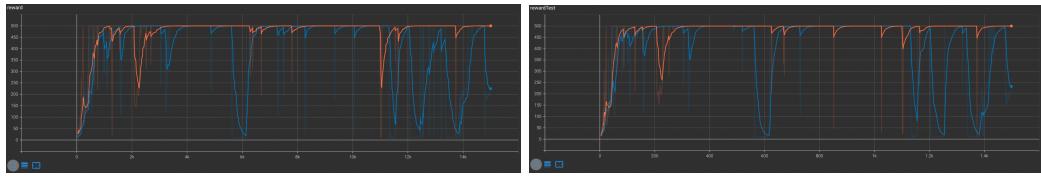
(a) Reward en Train avec paramètre précédent

(b) Reward en Test avec paramètre précédent

FIGURE 49 – Comparaison Reward en Train et en Test avec les paramètres précédent sur Lunar Lander

6.1.3 Compare $\bar{D}_{KL}(\theta_k | \theta)$ et $\bar{D}_{KL}(\theta | \theta_k)$:

Avec K=10 pour l'actor et le critic, beta=1., delta=0.003, maxLengthTest=500, maxLengthTrain=500, lr_pi=0.0003, lr_v=0.0001, gamma=0.99, lambda=0.99. L'entraînement avec $\bar{D}_{KL}(\theta | \theta_k)$ est beaucoup plus instable, il n'arrive pas à converger, il y a des oscillations brusques (de reward 500 à 10).



(a) Reward en Train avec $\bar{D}_{KL}(\theta_k | \theta)$ en orange et $\bar{D}_{KL}(\theta | \theta_k)$ en bleu
(b) Reward en Test avec $\bar{D}_{KL}(\theta_k | \theta)$ en orange et $\bar{D}_{KL}(\theta | \theta_k)$ en bleu

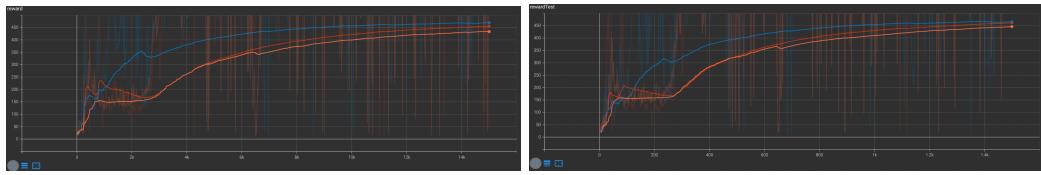
FIGURE 50 – Comparaison Reward en Train et en Test avec avec $\bar{D}_{KL}(\theta_k | \theta)$ en orange et $\bar{D}_{KL}(\theta | \theta_k)$ en blue

6.2 PPO avec Clipped Surrogate Objective :

Dans cette version du PPO nous allons fixer le maxLengthTest et le maxLengthTrain qui sont tous deux égal à 500, gamma=0.99, lambda=0.97. De plus comme précédemment nous allons mettre à jour l'actor et le critic à chaque fin de trajectoire, avec le même nombre d'optimisation.

6.2.1 Cartpole :

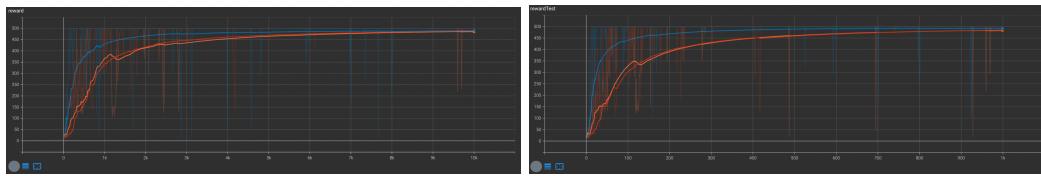
Nous allons comparer les valeurs des différents epsilon pour le clip, nous pouvons remarqué qu'avec un $\epsilon = 0.2$ le résultat est meilleur, mais dans en moyenne les résultats sont presque tous identiques.



(a) Reward en Train $\epsilon = 0.1$ en orange, 0.2 en rouge, 0.3 en bleu avec K=10
(b) Reward en Test $\epsilon = 0.1$ en orange, 0.2 en rouge, 0.3 en bleu avec K=10

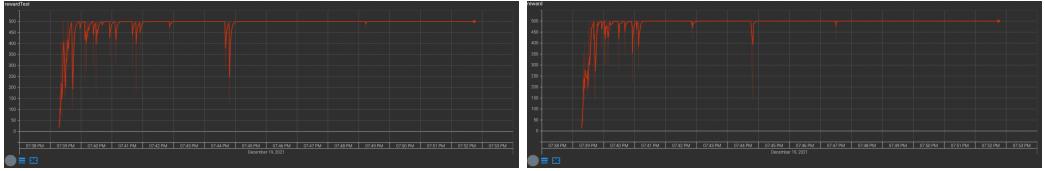
FIGURE 51 – Comparaison Reward en Train et en Test $\epsilon = 0.1$ en orange, 0.2 en rouge, 0.3 en bleu avec K=10

Avec $lr_v = 0.001$ tout en gardant les même paramètre que précédemment, nous avons exprimenter notre modél, malgré quelque oscillation de temps en temps, tous les steps k=3/5/10 arrive a avoir en moyenne presque le maximum du reward, de plus même si de temps en temps en remarque des changements brusques, mais l'algorithme converge bien vers le maximum du reward.



(a) Reward en Train $\epsilon = 0.2$ avec K=3 en rouge, K=5 en orange, K=10 en bleu, $lr_v = 0.001$
(b) Reward en Test $\epsilon = 0.2$ avec K=3 en rouge, K=5 en orange, K=10 en bleu, $lr_v = 0.001$

FIGURE 52 – Comparaison Reward en Train et en Test $\epsilon = 0.2$ avec K=3 en rouge, K=5 en orange, K=10 en bleu

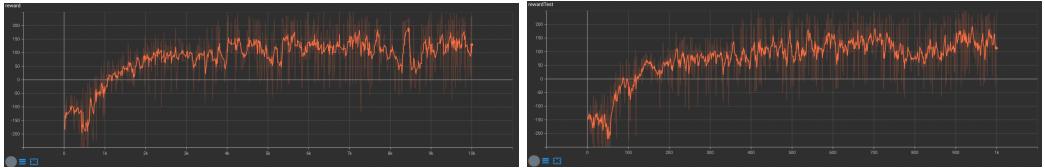


(a) Reward en Train $\epsilon = 0.2$, K=3, $lr_v = 0.001$ (b) Reward en Test $\epsilon = 0.2$, K=3, $lr_v = 0.001$

FIGURE 53 – Comparaison Reward en Train et en Test $\epsilon = 0.2$, K=3, $lr_v = 0.001$

6.2.2 Lunar Lander

Nous avons testé Lunar Lander avec les paramètres suivant : maxLengthTest=500, maxLengthTrain=500, $lr_pi = 0.0003$, $lr_v=0.001$, gamma=0.99 ,lambda : 0.97 , epsilon=0.2. Le reward obtenu avec cette tâche est semblable a celui obtenu avec PPO KL.



(a) Reward en Train $\epsilon = 0.2$, K=3, $lr_v = 0.001$ (b) Reward en Test $\epsilon = 0.2$, K=3, $lr_v = 0.001$

FIGURE 54 – Comparaison Reward en Train et en Test $\epsilon = 0.2$, K=3, $lr_v = 0.001$

6.2.3 Comparison des Surrogate Objectives

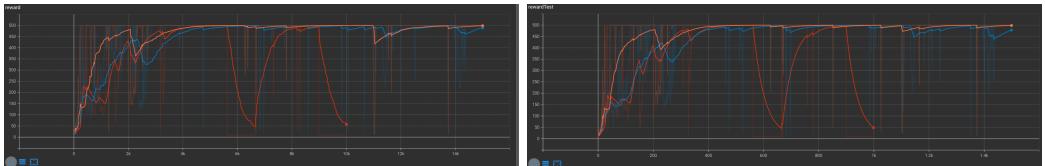
Pas de Clipping ni de Pénalité : $L_t(\theta) = r_t(\theta)\hat{A}_t$

Clipping : $L_t(\theta) = \min\left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta)\right), 1 - \epsilon, 1 + \epsilon\right)\hat{A}_t$

KL pénalité Adaptative

$L_t(\theta) = r_t(\theta)\hat{A}_t - \beta \text{KL}[\pi_{\theta_{\text{old}}}, \pi_{\theta}]$

Nous avons comparé toutes ces surrogate sur cartpole, avec un k=10 pour l'actor et le critic, $\epsilon = 2$, $\delta = 0.003$, $\gamma = 0.99$, $= 0.99$. La KL adaptative et le Clipping arrivent à faire presque les mêmes rewards, contrairement a celle n'appliquant rien qui s'effondre.



(a) Reward en Train KL adaptative en orange, Clip- (b) Reward en Test KL adaptative en orange, Clip-
ping en blue et avec rien en rouge ping en blue et avec rien en rouge

FIGURE 55 – Comparaison Reward en Train et en Test KL adaptative, Clipping, sans rien

7 Continuous Actions/Deep Deterministic Policy Gradient (DDPG) :

DQN résout les problèmes avec les espaces d'observation de grande dimension, mais il ne peut gérer que des espaces d'action discrets et de faible dimension, la solution DDPG. DDPG est un algorithme qui apprend simultanément une fonction Q et une policy. Il est off-policy et utilise l'équation de Bellman pour apprendre la fonction Q et de la fonction Q a apprendre la policy.

Le DQN classique utilise donc l'équation de Bellman afin de mettre à jour un état et une action $Q^\pi(s_t, a_t) = E_{r_t, s_{t+1} \sim E} [r(s_t, a_t) + \gamma E_{a_{t+1} \sim \pi} [Q^\pi(s_{t+1}, a_{t+1})]]$, avec s_{t+1} échantillonné par l'environnement à partir d'une distribution $E(.|s_t, a_t)$, de même pour a_{t+1} qui est échantillonner selon la politique courante.

Comme on est dans le cas déterministe nous allons utilisé la formule suivante :

$$Q^\mu(s_t, a_t) = E_{r_t, s_{t+1} \sim E} [r(s_t, a_t) + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1}))] \quad (15)$$

On considère des approximations de fonctions paramétrées par ϕ , nous allons donc essaie de minimiser la loss suivante :

$$L(\phi) = E_{s_t, a_t, r_t, s_{t+1} \sim E} \left[(Q(s_t, a_t | \phi) - y_t)^2 \right] \quad (16)$$

avec :

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1}) | \phi) \quad (17)$$

Donc en concrètement, l'algorithme maintient une fonction d'acteur paramétrée $\mu(s | \theta^\mu)$ qui spécifie la politique actuelle en mappant de manière déterministe les états à une action spécifique, $Q(s, a)$ est appris en utilisant l'équation de Bellman comme dans Q-learning, et l'acteur est mis à jour avec $E_{s_t \sim \rho^\beta} [\nabla_{\theta^\mu} Q(s, a | \phi)|_{s=s_t, a=\mu(s_t | \theta^\mu)}]$. Un défi majeur de l'apprentissage dans les espaces d'action continue est l'exploration, Un avantage de DDPG est que nous pouvons traiter le problème de l'exploration indépendamment de l'apprentissage en utilisant la formule suivante :

$$\mu'(s_t) = \mu(s_t | \theta_t^\mu) + \mathcal{N} \quad (18)$$

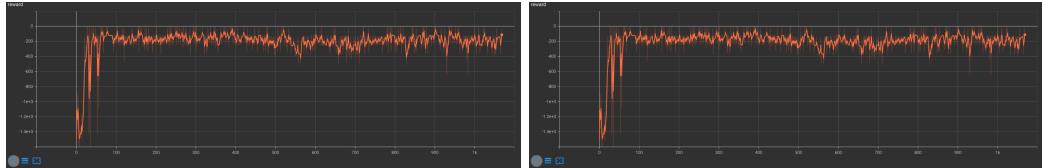
Dans ce modèle nous avons utilisé Ornstein-Uhlenbeck process pour générer le bruit d'exploration, et nous avons utilisé un soft target updates pour mettre à jour les targets, ainsi qu'un replay buffer avec lequel nous allons sampler des exemples.

Dans notre implémentation, dans le forward la dernière couche est suivi d'un tanh multiplier par high afin d'avoir des actions bornée entre la valeur high et low, de plus nous avons ajouté à nous module la possibilité de rajouté une batchnorm ou layernorm pour l'actor et le critic, le processus d'exploration de Ornstein-Uhlenbeck avec $\sigma = 0.2$ est réinitialisé à la fin de chaque trajectoire, et la mise à jour ce fait de façon soft avec un α comme paramètre. Nous allons utiliser les paramètres suivant sauf mention contraire :

`freqOptim=1000` le minimum de step avant d'optimiser, `maxLengthTest=200`, `maxLengthTrain=200`, `lr_pi = 0.001`, `lr_q = 0.003`, `gamma=0.99`, `optimStep=50` chaque optimStep nous optimisons optimStep, `use_batch_norm = false/true`, `prior=false/true`, `capacity=1000000`, `ru=0.995`, `batch_size=100`, avec activation LeakyReLU.

DDPG arrive à convergé vers une valeur négative, nous ne pouvons pas faire mieux avec pendulum

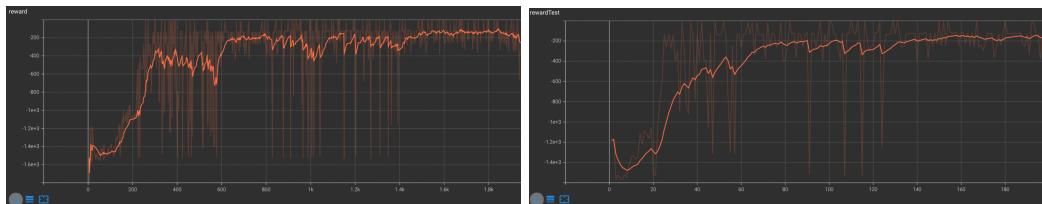
7.1 Pendulum :



(a) Reward en Train avec les paramètres précédent (b) Reward en Test avec les paramètres précédent

FIGURE 56 – Reward en Train et en Test

Nous utiliserons la batchnorm, avec un `batch_size = 1000` et une optimisation de `optimStep` chaque `freqStep` fois. Résultat un peu meilleur que précédemment mais reste instable, ici il arrive à atteindre 0 reward alors que dans l'ancien test tout était dans le négatif.

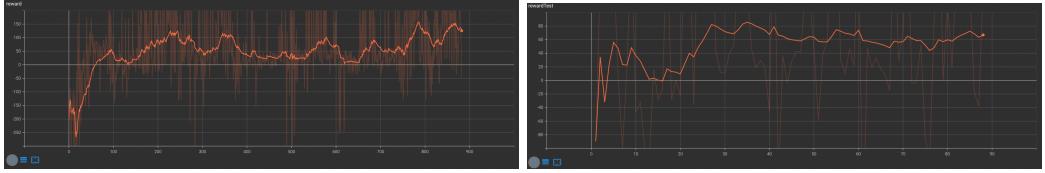


(a) Reward en Train avec les paramètres précédent et une batchnormalisation (b) Reward en Test avec les paramètres précédent et une batchnormalisation

FIGURE 57 – Reward en Train et en Test avec batchnormalisation

7.2 Lunar Lander Continuous :

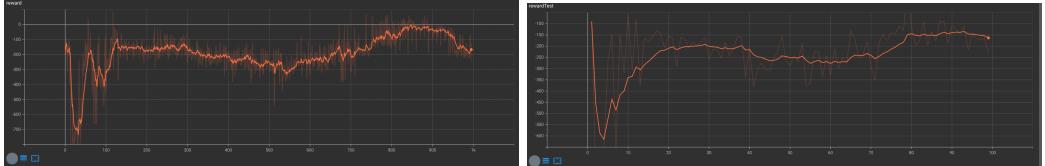
Avec les mêmes paramètres que précédemment, nous arrivons à avoir un reward positif, lunar qui était déjà difficile avec les actions continues une autre difficulté qui ce rajoute, un résultat plutôt satisfaisant sans batchnormalisation.



(a) Reward en Train avec les paramètres précédent sur Lunar
(b) Reward en Test avec les paramètres précédent sur Lunar

FIGURE 58 – Reward en Train et en Test sur Lunar

Avec la batchnormalisation il n'arrive pas vraiment à atteindre un reward positif, avec un réseau de [30,30].

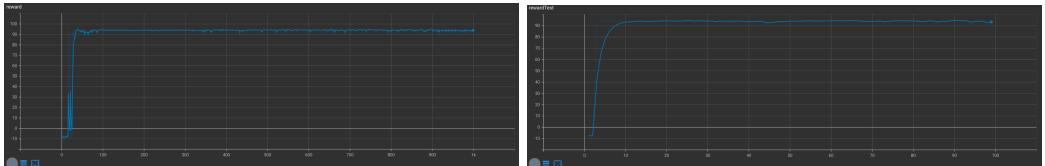


(a) Reward en Train avec les paramètres précédent sur Lunar et batchnormalisation
(b) Reward en Test avec les paramètres précédent sur Lunar et batchnormalisation

FIGURE 59 – Reward en Train et en Test sur Lunar et batchnormalisation

7.3 Mountain Car Continuous :

Avec les paramètres précédent convergence atteintes, donc ce n'est pas la peine d'essaier avec la batchnormalisation, et avec maxLenghtTest et Train=1000.



(a) Reward en Train sur mountainCar avec les paramètres précédent
(b) Reward en Test sur mountainCar avec les paramètres précédent

FIGURE 60 – Reward en Train et en Test sur mountainCar avec les paramètres précédent

8 Continuous Actions/Soft Actor-Critic (SAC) :

SAC est un algorithme qui optimise une policy stochastique off-policy. La caractéristique de SAC est qu'il utilise une fonction objectif RL modifiée, c'est à dire au lieu de maximiser la somme des reward $\sum_t E_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} [r(\mathbf{s}_t, \mathbf{a}_t)]$, SAC cherche également à maximiser l'entropie de la policy.

$$J(\pi) = \sum_{t=0}^T E_{(\mathbf{s}_t, \mathbf{a}_t) \sim \rho_\pi} [r(\mathbf{s}_t, \mathbf{a}_t) + \alpha \mathcal{H}(\pi(\cdot | \mathbf{s}_t))] \quad (19)$$

Le paramètre de température α détermine l'importance relative du terme d'entropie par rapport au reward, et contrôle ainsi la stochasticité de la politique optimale. Nous voulons maximiser l'entropie dans notre policy pour encourager explicitement l'exploration, pour encourager la policy à attribuer des probabilités égales aux actions qui ont des valeurs Q identiques ou presque égales.

SAC utilise apprend simultanément 3 réseaux, une politique π_θ et deux fonctions Q_{ϕ_1}, Q_{ϕ_2} avec la minimisation de la loss suivante :

$$L(\phi_i, \mathcal{D}) = \mathbb{E}_{(s, a, r, s', d) \sim \mathcal{D}} [(Q_{\phi_i}(s, a) - y(r, s', d))^2] \quad (20)$$

$$y(r, s', d) = r + \gamma(1-d) \left(\min_{j=1,2} Q_{\phi_{\text{targ},j}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}' | s') \right), \quad \tilde{a}' \sim \pi_\theta(\cdot | s'). \quad (21)$$

$$V^\pi(s) = \underset{a \sim \pi}{\mathbb{E}} [Q^\pi(s, a)] + \alpha H(\pi(\cdot | s)) \quad (22)$$

Nous apprenons le réseau Policy π en minimisant l'erreur suivante :

$$J_\pi(\phi) = E_{\mathbf{s}_t \sim \mathcal{D}} \left[D_{\text{KL}} \left(\pi_\phi(\cdot | \mathbf{s}_t) \| \frac{\exp(Q_\theta(\mathbf{s}_t, \cdot))}{Z_\theta(\mathbf{s}_t)} \right) \right] \quad (23)$$

la densité cible est la fonction Q, qui est représentée par un réseau de neurones et peut être différencié, et il est donc pratique d'appliquer la reparameterization trick à la place, $\mathbf{a}_t = f_\phi(\epsilon_t; \mathbf{s}_t)$

$$J_\pi(\phi) = E_{\mathbf{s}_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}} [\log \pi_\phi(f_\phi(\epsilon_t; \mathbf{s}_t) | \mathbf{s}_t) - Q_\theta(\mathbf{s}_t, f_\phi(\epsilon_t; \mathbf{s}_t))] \quad (24)$$

où π_ϕ est défini implicitement en fonction de f_ϕ , La fonction de normalisation Z est supprimée car elle ne dépend pas du paramètre ϕ , ce qui résulte de la formule suivante :

$$\hat{\nabla}_\phi J_\pi(\phi) = \nabla_\phi \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) + (\nabla_{\mathbf{a}_t} \log \pi_\phi(\mathbf{a}_t | \mathbf{s}_t) - \nabla_{\mathbf{a}_t} Q(\mathbf{s}_t, \mathbf{a}_t)) \nabla_\phi f_\phi(\epsilon_t; \mathbf{s}_t) \quad (25)$$

Mais comme nous utilisons une gaussien qui est non bornée comme distribution d'action, nous avons besoin que les actions soient bornées sur un intervalle fini, pour ce faire nous allons utilisé la squashed Gaussian policy (\tanh) pour sampler les exemples comme la formule suivante :

$$\tilde{a}_\theta(s, \xi) = \tanh(\mu_\theta(s) + \sigma_\theta(s) \odot \xi), \quad \xi \sim \mathcal{N}(0, I) \quad (26)$$

Nous appliquons donc la fonction \tanh qui a des valeurs comprises entre -1 et 1 pour garantir que les actions soient limitées à une plage finie, nous allons multiplier par le high action afin d'avoir la bornes adequat (nous faisons l'hypothèse que low=high).

En d'autre terme soit $\mathbf{u} \in R^D$ qui est une variable aléatoire et $\mu(\mathbf{u} | \mathbf{s})$ la densité correspondante (qui peut être infini). Donc nous aurons $action = \tanh(u)$, avec une densité donnée par (d'après l'article original) :

$$\pi(\mathbf{a} | \mathbf{s}) = \mu(\mathbf{u} | \mathbf{s}) \left| \det \left(\frac{d\mathbf{a}}{d\mathbf{u}} \right) \right|^{-1} \quad (27)$$

$$\log \pi(\mathbf{a} | \mathbf{s}) = \log \mu(\mathbf{u} | \mathbf{s}) - \sum_{i=1}^D \log (1 - \tanh^2(u_i)) \quad (28)$$

$$\tanh = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\tanh^2 = \frac{e^x - e^{-x}}{e^x + e^{-x}} \times \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} + e^{-2x} - 2}{e^{2x} + e^{-2x} + 2}$$

$$1 - \tanh^2 = 1 - \frac{e^{2x} + e^{-2x} - 2}{e^{2x} + e^{-2x} + 2} = \frac{(e^{2x} + e^{-2x} + 2) - e^{2x} - e^{-2x} + 2}{e^{2x} + e^{-2x} + 2} = \frac{4}{e^{2x} + e^{-2x} + 2} = \frac{4e^{-2x}}{e^{-4x} + 2e^{-2x} + 1}$$

$$(e^{-2x} + 1)^2 = e^{-4x} + 2e^{-2x} + 1 \Rightarrow 1 - \tanh^2 = \left(\frac{2e^{-x}}{e^{-2x} + 1} \right)^2$$

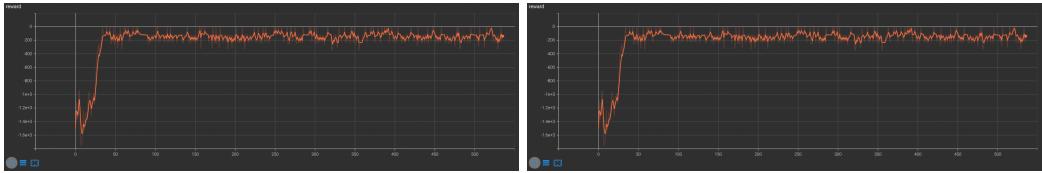
$\log(1 - \tanh^2) = 2\log(2) + -2x - 2\log(1 + e^{-2x})$, nous allons utiliser cette formule qui est plus stable que le \tanh^2 .

les paramètres sont les suivant : freqOptim=1000 le minimum de step avant d'optimiser, maxLengthTest=500, maxLengthTrain=500, lr_pi = 0.001, lr_q = 0.001, lr_alpha = 0.001, entropyTarget = 0.2, alpha=0.2, gamma=0.99, optimStep=50 chaque optimStep nous optimisons optimStep, , use_batch_norm = false/true, prior=false/true, capacity=1000000, ru=0.995, batch_size : 100, avec activation LeakyRelu.

8.1 SAC version α fixer :

8.1.1 Pendulum :

Nous testons pendulum avec les paramètres précédents, convergence atteinte (reward négative), ne fait pas mieux que DDPG

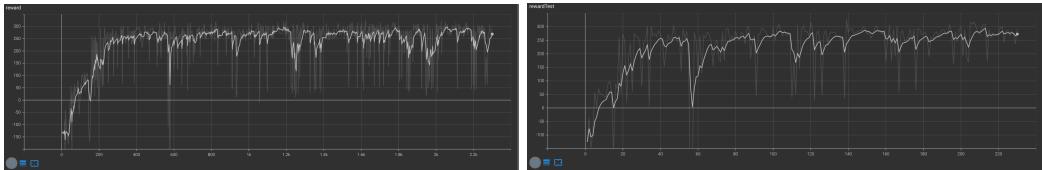


(a) Reward en Train avec les paramètres précédent (b) Reward en Test avec les paramètres précédent

FIGURE 61 – Reward en Train et en Test

8.1.2 Lunar Lundar :

Nous arrivons a atteindre une convergence pour Lunar, il arrive bien a ce stabilisé, avec les paramètres précédemment cité.



(a) Reward en Train avec les paramètres précédent (b) Reward en Test avec les paramètres précédent

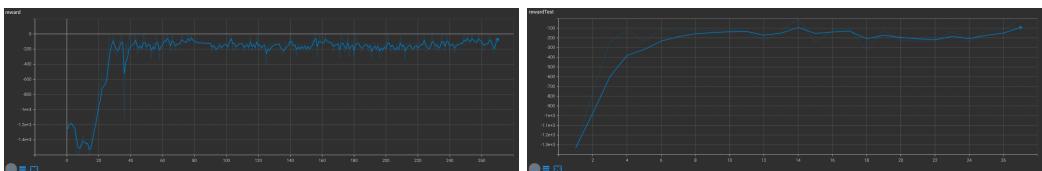
FIGURE 62 – Reward en Train et en Test

8.2 SAC α Adaptative :

Nous commencons donc avec un alpha=0 et nous ajustons le alpha avec $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$ tout les optim step et optim step fois.

8.2.1 Pendulum :

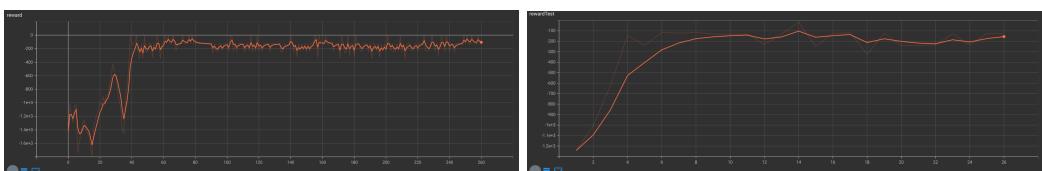
Nous testons notre modèle avec les paramètres précédent, il y a convergence rapidement mais vers une valeurs qui reste négative.



(a) Reward en Train avec les paramètres précédent sans batchnormalisation (b) Reward en Test avec les paramètres précédent sans batchnormalisation

FIGURE 63 – Reward en Train et en Test avec les paramètres précédent sans batchnormalisation

Pendulum converge rapidement, et reste stable avec une batchnormalisation, et reste identique a la version sans batchnorm.

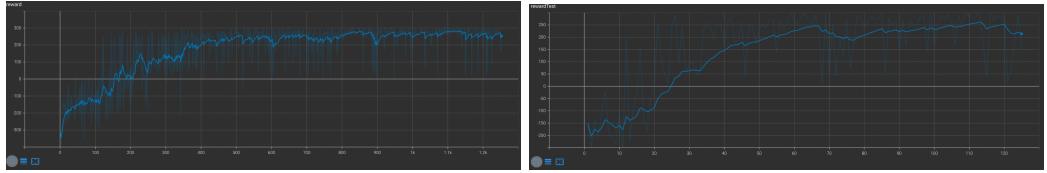


(a) Reward en Train avec les paramètres précédent et une batchnormalisation (b) Reward en Test avec les paramètres précédent et une batchnormalisation

FIGURE 64 – Reward en Train et en Test avec les paramètres précédent et une batchnormalisation

8.2.2 Lunar Lundar :

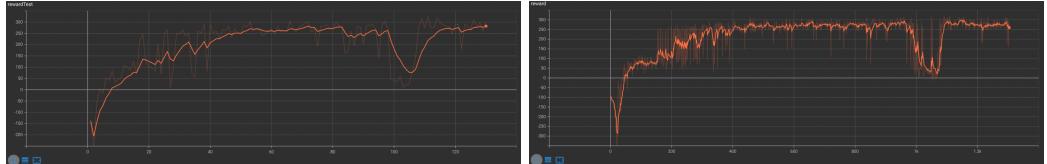
Nous arrivons a apprendre aussi bien qu'avec sac avec une temperature fixe, et il arrive bien a convergé.



(a) Reward en Train avec les paramètres précédent sans batchnormalisation
(b) Reward en Test avec les paramètres précédent sans batchnormalisation

FIGURE 65 – Reward en Train et en Test avec les paramètres précédent sans batchnormalisation

Avec le SAC adaptative et une batchnorm, Lunar arrive a convergé et reste stable, meilleur performance sur lunar obtenu.

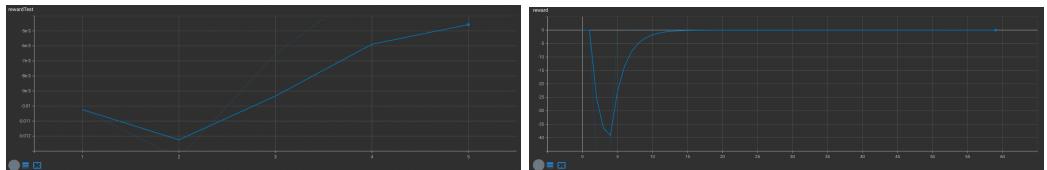


(a) Reward en Train sur Lunar Lander avec batchnorm sur les mêmes paramètres que précédemment
(b) Reward en Test sur Lunar Lander avec batchnorm sur les mêmes paramètres que précédemment

FIGURE 66 – Reward en Train et en Test avec Batchnormalisation

8.2.3 Montain Car :

Montain car arrive a converger vers zéro, mais n'arrive pas a faire mieux que le résultat trouvé avec DDPG, hyperparamétrage.



(a) Reward en Train avec les paramètres précédent avec batchnormalisation
(b) Reward en Test avec les paramètres précédent avec batchnormalisation

FIGURE 67 – Reward en Train et en Test avec les paramètres précédent avec batchnormalisation