
Apprentissage par renforcement
Rendu de TP n°1
TPs 1 à 8

Table des matières

1 TP1 : Problèmes de bandits	4
1.1 Baseline	4
1.1.1 Stratégie Random	4
1.1.2 Stratégie StaticBest	4
1.1.3 Stratégie Optimale	5
1.2 UCB	6
1.3 LinUCB	6
2 TP2 : Programmation dynamique	9
2.1 Policy Iteration	9
2.1.1 Etude du paramètre de discount	9
2.1.2 Etude de différentes configurations d'environnements	10
2.2 Value Iteration	11
2.2.1 Etude du paramètre de discount	11
2.2.2 Etude de différentes configurations d'environnements	12
2.3 Comparaison des deux algorithmes	13
2.3.1 Temps de calcul	13
2.3.2 comparaison des politiques résultantes	13
3 TP3 : Q-Learning	16
3.1 Plan 1	16
3.2 Plan 4	16
3.3 Plan 5	18
3.4 Plan 8	19
4 TP4 : DQN	22
4.1 Référence	22
4.2 Etude des paramètres de l'algorithme	22
4.2.1 Exploration	23
4.2.2 Amortissement de l'exploration	23
4.2.3 Paramètres de discount	24
4.2.4 Initialisation du pas d'apprentissage	24
4.2.5 Fréquence d'optimisation du réseau cible	25
4.2.6 Mini-batch	25
4.3 Comparaison des différentes versions des algorithmes	26
4.3.1 DQN simple	26
4.3.2 Target Network	26
4.3.3 Prioritized Replay	27
4.3.4 DQN target + prioritized Replay	27
4.4 Etude des autres jeux : Lunar Lander et Gridworld	28
5 TP5 : Policy Gradient	29
5.1 Cartpole	29
5.2 Lunar	30
5.3 Gridworld	30
5.4 Stratégies de mise à jour	31

6 TP6 : Advanced Policy Gradients	32
6.1 Initialisation	33
7 TP7 : Continuous actions	34
7.1 Pendulum	34
7.2 Lunar Lander	34
7.3 Continuous Mountain Car	35
8 TP8 : Soft-Actor Critic (SAC)	36
8.1 Température fixe	36
8.2 Température adaptative	37
9 Comparaison des algorithmes pour environnements à actions discrètes	38
10 Comparaison des algorithmes pour environnements à actions continues	39

Introduction

Ce rapport est un compte-rendu des travaux pratiques liés à l'UE *Reinforcement Learning and advanced Deep Learning*. L'objectif général de chaque TP est l'apprentissage de diverses tâches par un agent placé dans un environnement donné, et avec lequel il interagit. Dans ce contexte, nous ne disposons que de peu d'informations sur l'environnement permettant de définir une politique de décision pour l'agent. Le but de l'apprentissage par renforcement est donc d'apprendre à effectuer une tâche donnée à partir de la perception du monde que nous avons à disposition et en se basant sur les retours de l'environnement selon les actions choisies.

Des expérimentations des algorithmes classiques de cette branche de l'apprentissage sont présentées ici, en insistant sur les résultats quantitatifs obtenus. Plus de détails sur la théorie, et en particulier sur les pseudos-codes sont disponibles sur le site du cours d'apprentissage par renforcement de l'université de la Sorbonne, à l'adresse :<https://dac.lip6.fr/master/rld-2021-2022/>.

Le code associé à ce rendu permet de reproduire les expériences présentées. Le dossier est divisé en différentes parties, correspondants à plusieurs problèmes spécifiques. Le premier (1) consiste à explorer les différentes stratégies d'exploration/exploitation et de comprendre les compromis à faire pour maximiser ces deux quantités opposées. Il est présenté sous forme d'un notebook. La section 2 présente un premier problème d'apprentissage, lorsque le modèle du Monde dans lequel évolue l'agent est parfaitement connu. Il propose deux algorithmes pouvant être exécutés indépendamment ainsi qu'un fichier `test.py` proposant de reproduire les courbes proposées dans ce rapport. Les codes associés aux sections 3 à 6 sont regroupés dans un même dossier afin de permettre la comparaison des différents algorithmes entre eux. On s'intéresse en effet à la résolution des mêmes tâches (apprentissage de différents jeux dans des environnements à actions **discrètes** : gridworld, cartpole et lunar lander issus du package `gym`) sous différentes approches. Chaque algorithme est présenté sous forme d'une classe comprenant l'initialisation d'une instance de l'agent utilisant l'algorithme considéré, ainsi que les différentes fonctions et la boucle d'apprentissage permettant de l'exécuter. Selon les algorithmes, différentes quantités sont enregistrées dans `tensorboard` pour visualisation (dans le dossier XP). Les hyper-paramètres et les configurations nécessaires à chacun de ces algorithmes et environnements sont proposés dans le dossier `configs` et sont également associés à chaque courbe. Finalement, les deux dernières sections 7 et 8, sont structurées d'une façon similaires aux sections les précédents, mais proposent de résoudre des tâches sur des environnements à actions **continues**.

1 TP1 : Problèmes de bandits

Ce premier TP s'intéresse à l'étude de différentes stratégies afin de choisir une publicité parmi 10 annonceurs dans le but d'engrenger le plus fort taux de clics. Différentes stratégies sont proposées afin de se rapprocher le plus possible de la stratégie optimale.

1.1 Baseline

Pour commencer, 3 stratégies sont utilisées : la première consiste à tirer aléatoirement les bras. La seconde choisit toujours le même bras qui correspond au meilleur bras en moyenne. Il s'agit donc d'une stratégie non applicable en pratique puisqu'elle nécessite de connaître les récompenses avant de faire un choix. De même que la troisième, qui consiste à choisir le bras avec la meilleure récompense à chaque instant t

1.1.1 Stratégie Random

Cette première stratégie consiste à sélectionner aléatoirement le bras à utiliser à chaque pas de temps. Il s'agit de la pire stratégie à adopter, puisqu'aucun apprentissage n'est tiré des coups précédents.

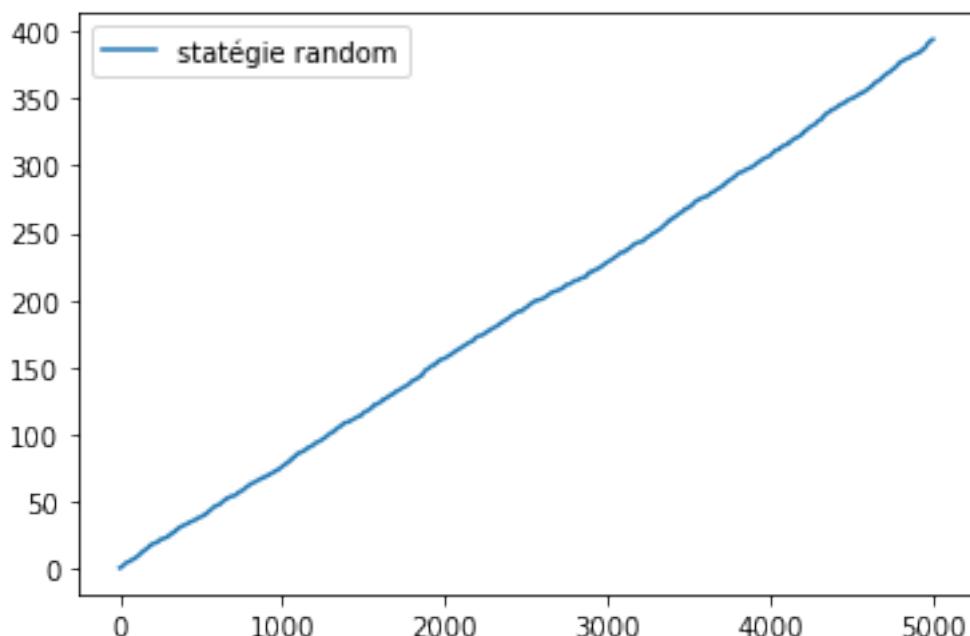


FIGURE 1 – Score cumulé de la stratégie random.

1.1.2 Stratégie StaticBest

La seconde stratégie consiste à sélectionner le bras qui maximise la somme des taux de clics sur tous les pas de temps. Il faut donc les connaître à priori et cette stratégie n'est pas utilisable en pratique.

Cette seconde stratégie est logiquement bien plus efficace que la première.

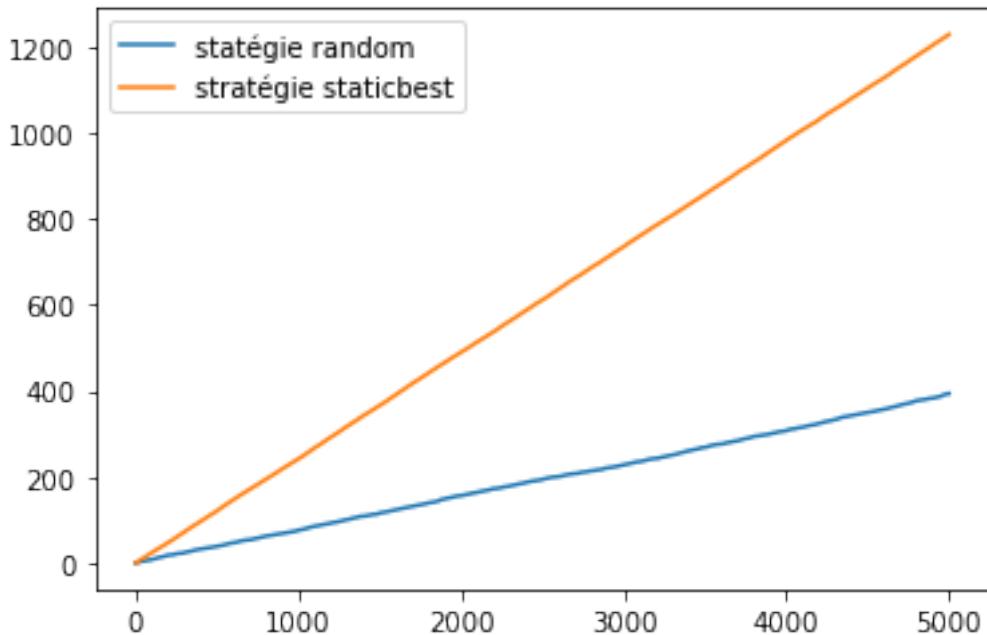


FIGURE 2 – Score cumulé de la stratégie staticbest (comparée aux précédentes).

1.1.3 Stratégie Optimale

La dernière stratégie consiste à prendre le bras qui possède la meilleure récompense à chaque itération. Encore une fois, il faut avoir accès à cette information avant d'effectuer son choix, ce qui n'est pas toujours possible en pratique.

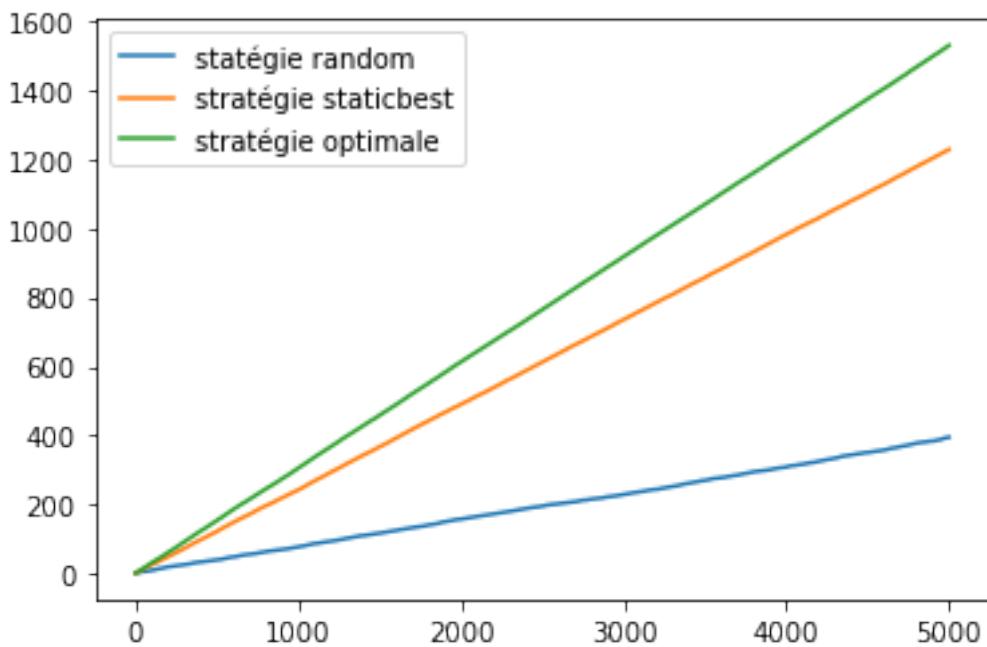


FIGURE 3 – Score cumulé de la stratégie optimale (comparée aux précédentes).

Cette troisième stratégie est encore une fois meilleure que les deux précédentes car s'agit de la meilleure solution possible avec les données.

1.2 UCB

Cette section propose une implémentation de l'algorithme UCB.

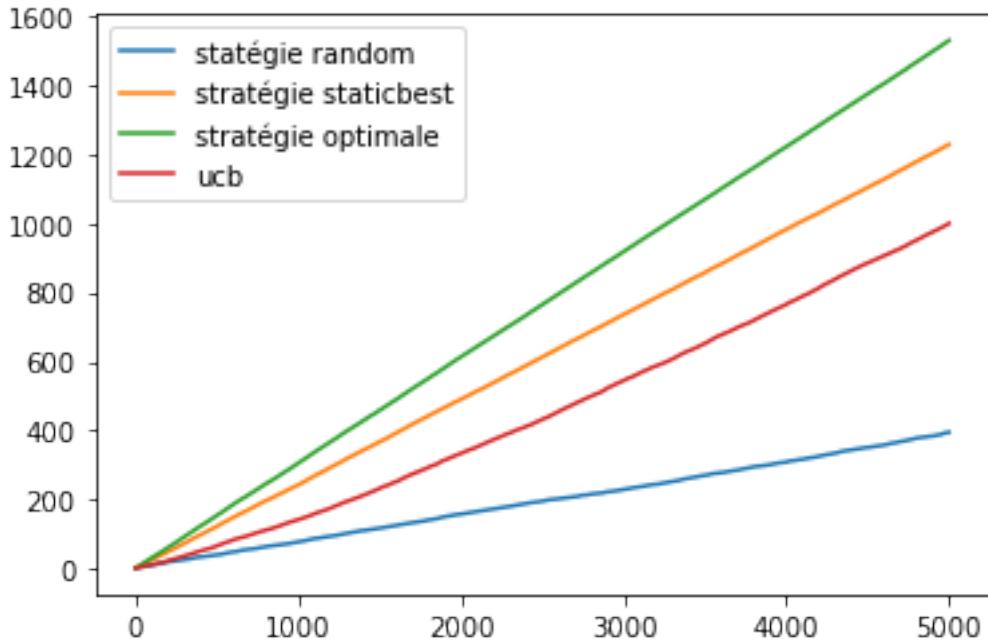


FIGURE 4 – Score cumulé de la stratégie issue de l'algorithme UCB (comparée aux précédentes).

L'algorithme UCB présente de bonnes performances, puisqu'il est bien plus efficace que la stratégie aléatoire. En revanche, il est moins bon que la stratégie staticBest.

1.3 LinUCB

Cette section propose une implémentation de l'algorithme LinUCB et basée sur le pseudo-code du cours.

LinUCB présente des performances très intéressantes. Son résultat sur le jeu de données est meilleur que les stratégies random et static best (qui utilisait le bras présentant le meilleur taux de clics moyens). En étudiant les performances de LinUCB selon la valeur de alpha, on note que ses performances peuvent se rapprocher de façon intéressante de la stratégie optimale (pour une valeur du paramètres α autour de 0.1) ([6](#) et [7](#)). Il permet de contrôler la largeur de l'intervalle de confiance. Ainsi, plus α est élevé, plus cet intervalle est large et l'agent explore plus qu'il n'exploite ses connaissances.

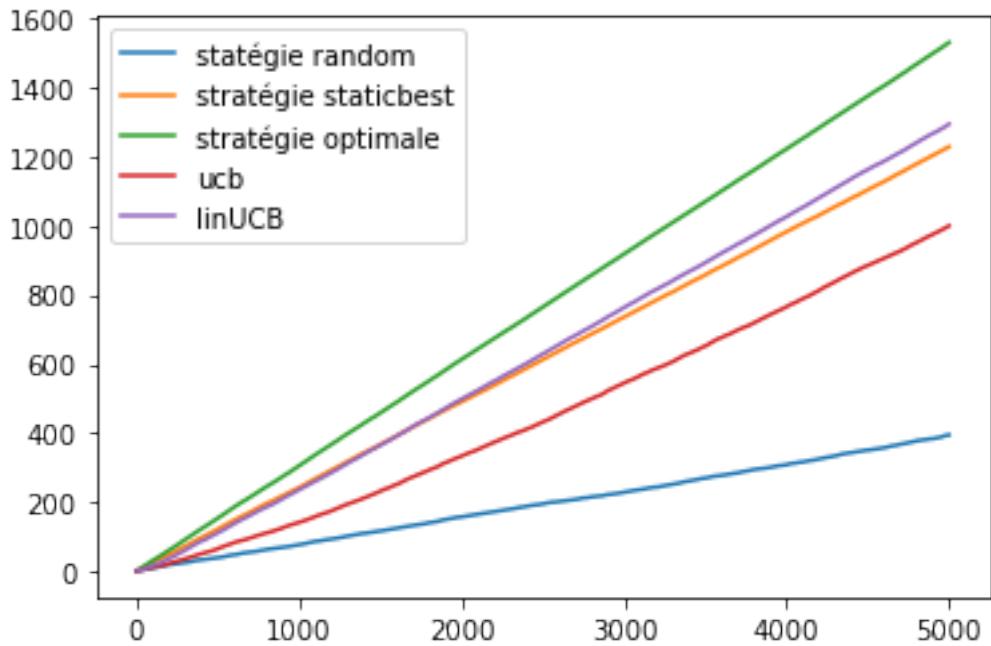


FIGURE 5 – Score cumulé de la stratégie issue de l’algorithme LinUCB (comparée aux précédentes).

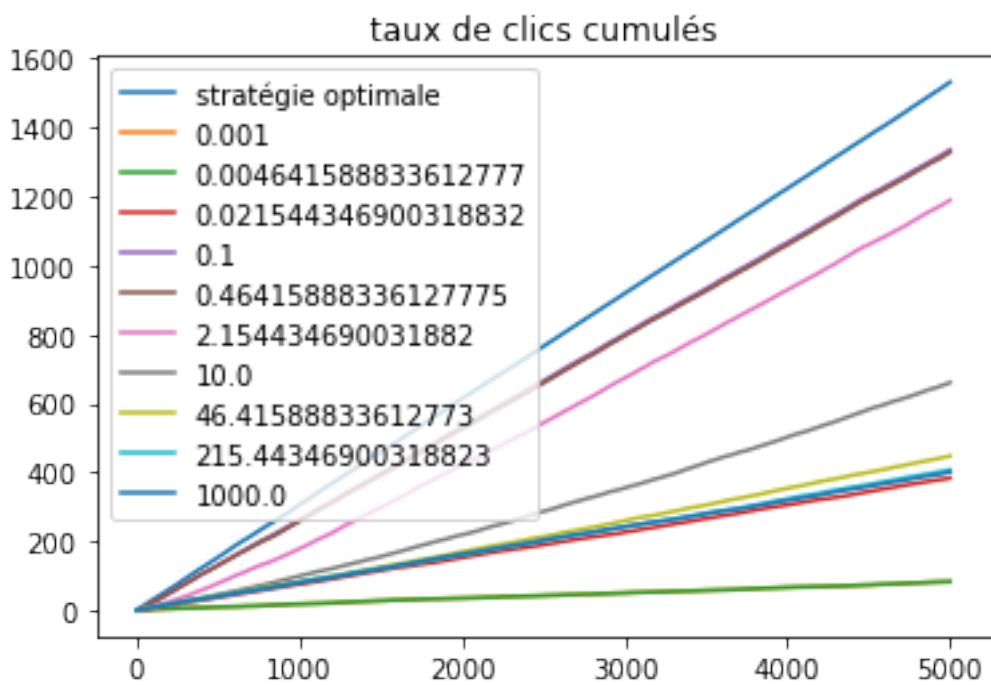


FIGURE 6 – Comparaison des scores cumulés obtenus pour différentes valeurs de α avec la stratégie optimale.

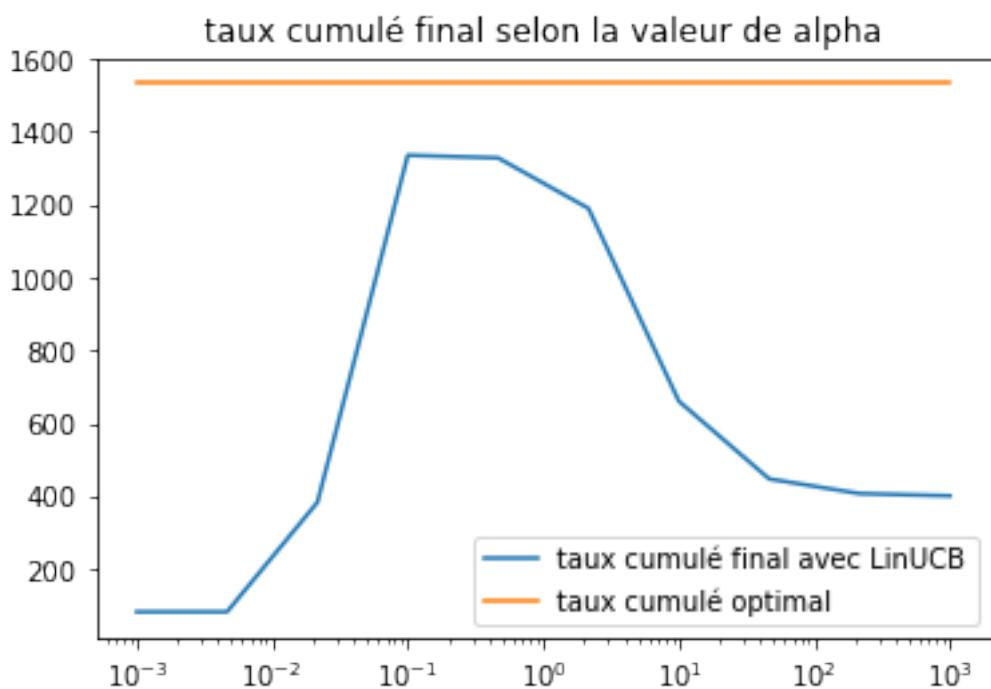


FIGURE 7 – Comparaison des score maximaux obtenus pour différentes valeurs de α avec la stratégie optimale

2 TP2 : Programmation dynamique

Ce TP a pour but d'implémenter les algorithmes Value Iteration et Policy Iteration et de les tester sur différentes cartes de l'environnement gridworld. On se place dans un contexte où le MDP est connu. Il sera donc utilisé pour calculer une politique de jeu pour résoudre le jeu en maximisant les récompenses obtenues. Le but est donc de prévoir quelles actions faire selon l'état dans lequel se trouve notre agent. On rappelle ici les significations des actions :

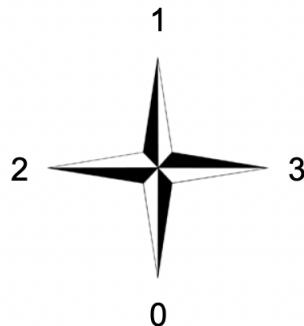


FIGURE 8 – Actions

Les expériences sont réalisées sur 10 cartes de jeu, mais nous ne présentons dans ce rapport que certains résultats choisis afin de ne pas le surcharger.

Note : La carte numérotée 9, ne fonctionnait pas, probablement car elle était trop lourde à charger sur les ordinateurs utilisés pour les calculs (une erreur de récursion apparaît lors du chargement de l'environnement).

2.1 Policy Iteration

Ce premier algorithme permet de calculer de façon itérative une politique à partir du MDP.

2.1.1 Etude du paramètre de discount

Le paramètre de discount (γ) permet de quantifier l'horizon de prise en compte des récompenses. Plus γ sera élevé, plus les récompenses à long terme auront de l'importance. L'environnement considéré est l'environnement numéroté 0 dans un soucis de lisibilité des politiques résultantes. Sur cette carte, les récompenses sont les suivantes.

- cases grises : -0.001 . La valeur est négative afin d'inciter l'agent à se diriger le plus rapidement possible vers la sortie
- case bleue : position initiale de l'agent
- case rouge : -1 . Cette case est terminale (à évite).
- case verte : $+1$. Cette case est terminale (objectif).

Les politiques observées selon différentes valeurs de γ sont présentées [10a](#) à [10f](#).

Avec ces résultats, on remarque que lorsque le paramètre de discount augmente, la politique semble se "propager" aux cases les plus éloignées (en bas à gauche) des cases

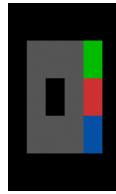


FIGURE 9 – Carte de jeu utilisée. Le point bleu correspond à notre agent au début du jeu, le point rouge à un état final avec récompense -1 et la point vert à un état final avec récompense $+1$.

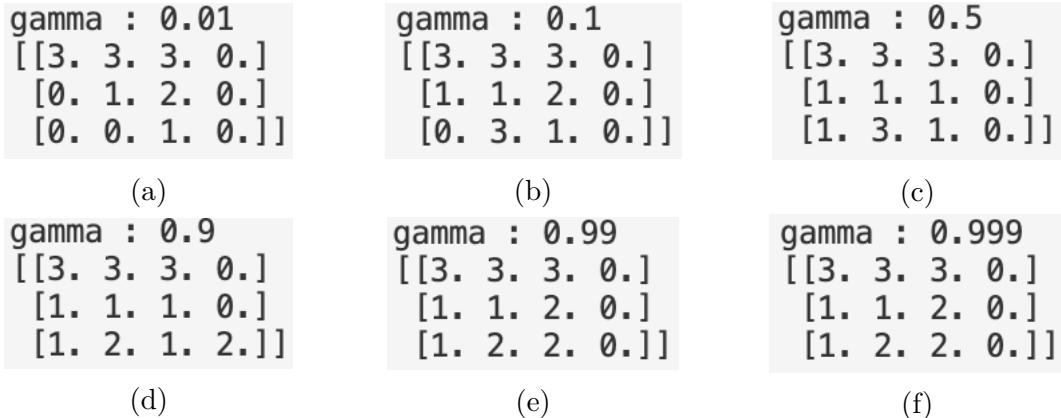


FIGURE 10 – Politique obtenue pour différentes valeurs du paramètre de discount avec Policy Iteration.

d'arrivées (en haut à droite de la carte). Entre un $\gamma = 0.5$ et un $\gamma = 0.9$, on note que la politique calculée est plus prudente. En effet, avec un paramètre de discount plus faible (0.5 par exemple), la politique de la case située à côté de l'état final rouge est de monter (au risque de tourner dans cette case perdante), alors que lorsqu'il augmente (0.9), cette politique devient plus prudente en choisissant de tourner à gauche, ce qui empêche l'agent d'aller sur la case rouge. Ce même phénomène est observé sur la case située en dessous lorsque l'on augmente encore γ . La politique devient plus prudente car l'horizon considéré est plus large. Ainsi, la possibilité de gagner a plus d'influence pour des valeurs élevées.

2.1.2 Etude de différentes configurations d'environnements

Dans cette section, nous présentons les politiques observées selon différents paramètres de l'environnement. Les paramètres utilisés par l'algorithme seront fixés à 1000 itérations maximum pour les boucles, le paramètre de discount est fixé à 0.99 et l'erreur ϵ (convergence pour le calcul de la fonction V) est choisie à 0.01. Du côté de l'environnement, ce sont les récompenses des différents types de cases qui sont modifiées. Ils sont présentés dans l'ordre suivant : [cases grises, cases vertes, cases bleues claires, cases rouges, cases roses].

La première figure (figure 11a) présente les politiques obtenues sur les configurations par défaut de l'environnement. L'agent évite à tout prix d'aller vers la case rouge qui a une récompense négative. Dans la seconde configuration (11b), ce sont les cases grises qui ont été modifiées. En effet, la pénalité apportée lorsque l'agent se déplace est plus

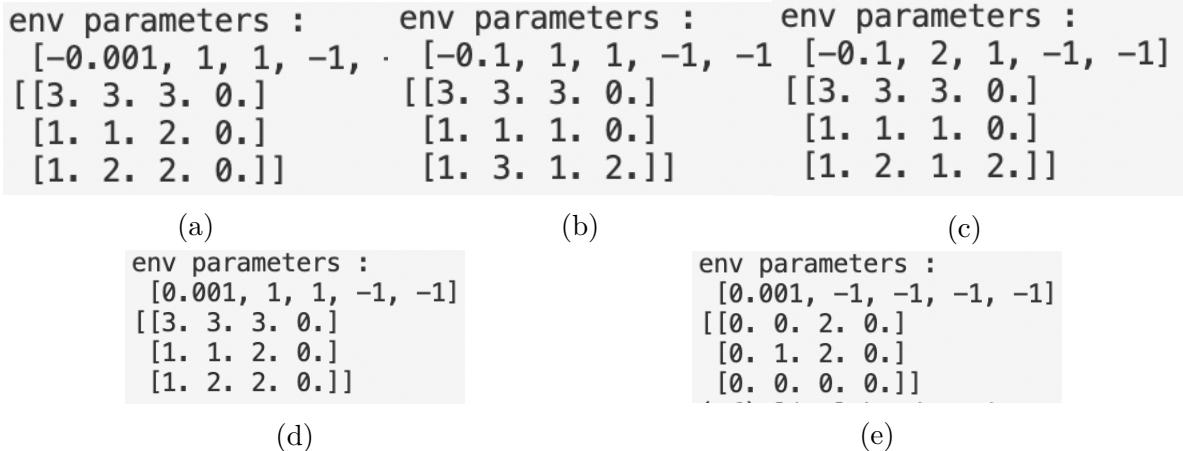


FIGURE 11 – Politique obtenue pour différentes configurations de l’environnement 0 avec Policy Iteration

élevée (elle passe de 0.01 à 0.1). Sur cet exemple, on note déjà un changement non négligeable de la politique. En effet, l’agent va prendre plus de risques pour arriver le plus rapidement possible à l’état final ayant une récompense positive. Cela modifie la politique sur les états autour de la case rouge, pour lesquels, au lieu de ne prendre aucun risque et de se diriger dans la direction strictement opposée, l’agent prend le risque d’aller vers la case perdante avec une probabilité de 0.1. La troisième configuration (11c) est similaire à la seconde, mais la récompense de la case terminale verte a été augmentée à 2. Cela ne change que très peu la politique de l’environnement précédent. Seul l’état situé sous le mur change pour avoir une politique plus prudente comme dans la première configuration. Dans les deux derniers ensembles (11d et 11e) de paramètres, les récompenses associées aux cases grises sont positives. Ainsi, l’agent n’a plus de pénalité de temps s’il reste longtemps dans l’environnement avant d’arriver à un état terminal. On note que dans le cas de la quatrième configuration, la politique obtenue est la même que celle obtenue dans l’environnement par défaut. En revanche, dans le dernier test effectué (11e), la politique est radicalement différent, puisque les récompenses des états finaux sont toutes négatives. Dans ce cas, la politique obtenue, est une politique très très prudente, dans laquelle l’agent ne se dirige jamais vers un état final. Les récompenses en jeu sont en effet positives et la stratégie optimale dans ce cas est de jouer à l’infini.

2.2 Value Iteration

L’algorithme d’itération sur la valeur permet de calculer la fonction valeur et d’extraire une politique directement à partir de cette fonction. Contrairement à l’algorithme présenté à la section précédente, il ne nécessite pas d’évaluation complète de la politique à chaque itération. Dans la section suivante, nous verrons l’impact de cette considération sur le temps de calcul.

2.2.1 Etude du paramètre de discount

Dans les figures 12a à 12f, les différentes politiques résultant de l’algorithme sont présentées.

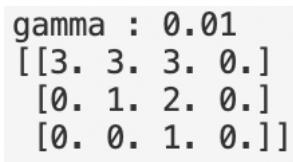
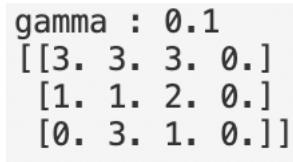
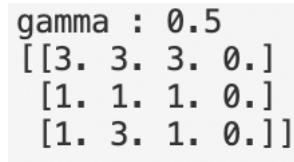
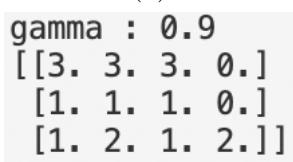
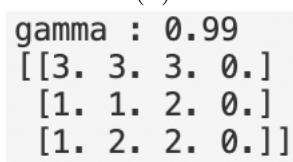
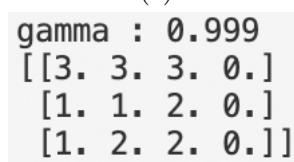
(a)  gamma : 0.01 [[3. 3. 3. 0.] [0. 1. 2. 0.] [0. 0. 1. 0.]]	(b)  gamma : 0.1 [[3. 3. 3. 0.] [1. 1. 2. 0.] [0. 3. 1. 0.]]	(c)  gamma : 0.5 [[3. 3. 3. 0.] [1. 1. 1. 0.] [1. 3. 1. 0.]]
(d)  gamma : 0.9 [[3. 3. 3. 0.] [1. 1. 1. 0.] [1. 2. 1. 2.]]	(e)  gamma : 0.99 [[3. 3. 3. 0.] [1. 1. 2. 0.] [1. 2. 2. 0.]]	(f)  gamma : 0.999 [[3. 3. 3. 0.] [1. 1. 2. 0.] [1. 2. 2. 0.]]

FIGURE 12 – Politique obtenue pour différentes valeurs du paramètre de discount avec Value Iteration

Les conclusions sont les mêmes que celles observées pour l'algorithme d'itération sur la politique : lorsque γ augmente, les récompenses se propagent aux état les plus éloignés et donc la politique est ajustée pour les prendre en compte.

2.2.2 Etude de différentes configurations d'environnements

En suivant le modèle adopté pour étudier l'algorithme d'itération sur la politique, nous présentons les politiques ici observées selon différents paramètres de l'environnement. Les paramètres utilisés par l'algorithme seront fixés à 1000 itérations maximum pour la boucle, le paramètres de discount est fixé à 0.99 et l'erreur ϵ (convergence pour le calcul de la fonction V) est choisie à 0.01. Du côté de l'environnement, ce sont les récompenses des différents types de cases qui sont modifiées. Ils sont présentés dans l'ordre suivant : [cases grises, cases vertes, cases bleues claires, cases rouges, cases roses].

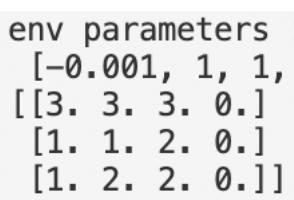
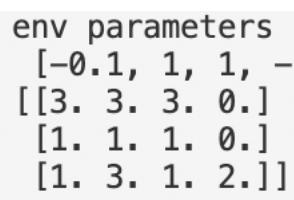
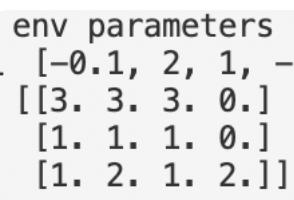
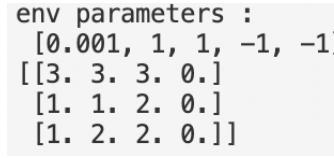
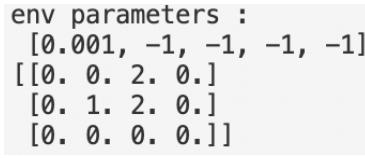
(a)  env parameters : [-0.001, 1, 1, -1, [[3. 3. 3. 0.] [1. 1. 2. 0.] [1. 2. 2. 0.]]]	(b)  env parameters : [-0.1, 1, 1, -1, [[3. 3. 3. 0.] [1. 1. 1. 0.] [1. 3. 1. 2.]]]	(c)  env parameters : [-0.1, 2, 1, -1, -1, [[3. 3. 3. 0.] [1. 1. 1. 0.] [1. 2. 1. 2.]]]
(d)  env parameters : [0.001, 1, 1, -1, -1] [[3. 3. 3. 0.] [1. 1. 2. 0.] [1. 2. 2. 0.]]	(e)  env parameters : [0.0, -1, -1, -1, -1] [[0. 0. 2. 0.] [0. 1. 2. 0.] [0. 0. 0. 0.]]	

FIGURE 13 – Politique obtenue pour différentes configurations de l'environnement 0 avec Value Iteration

Les observations et conclusions pour cet algorithme sont les mêmes que pour Policy Iteration. Les résultats sont similaires pour les configurations choisies. Lorsque les

pénalités augmentent (13a, 13b, 13c), la politique aura tendance à prendre plus de risques. Inversement, il est possible d'influencer la politique vers une stratégie extrêmement prudente (13d, 13e) en appliquant des récompenses négatives aux état finaux et en ne pénalisant pas le temps passé en jeu.

2.3 Comparaison des deux algorithmes

2.3.1 Temps de calcul

Tout d'abord, on sait que l'algorithme d'itération sur la politique est très coûteux à calculer car il nécessite une évaluation complète de la politique à chaque itération. L'algorithme d'itération sur la valeur permet de limiter cela en utilisant l'équation de Bellman pour évaluer la fonction V. De ce fait, la première comparaison réalisée entre ces deux algorithmes concerne le temps de calculs sur les différentes cartes mises à disposition (exemptée la carte 9). Les résultats sont présentés dans le graphique 14, pour lequel les paramètres utilisés sont les suivants : $\epsilon = 0.01$, nombre d'itérations = 1,000, $\gamma = 0.99$ et nombre d'itérations pour le calcul de la fonction valeur = 1,000 (policy iteration seulement).

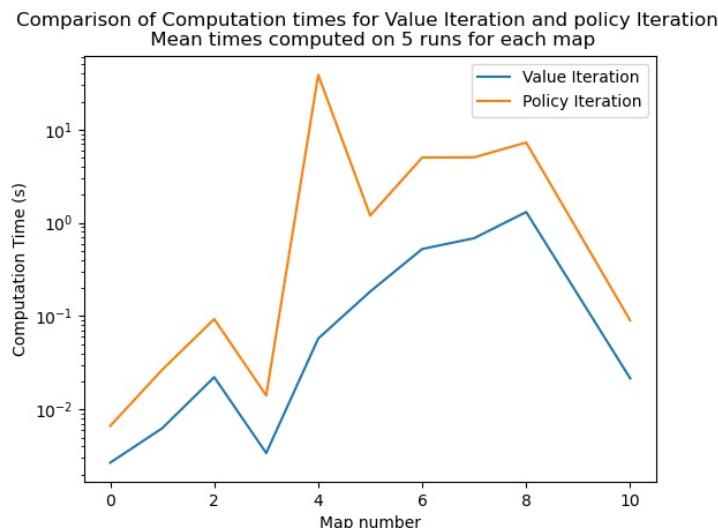
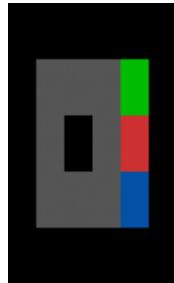


FIGURE 14 – Temps de calcul sur différentes cartes de gridworld pour les algorithmes Value Iteration et Policy Iteration

On observe effectivement que l'algorithme d'itération sur la politique est plus lent que l'algorithme d'itération sur la valeur, de part le plus grand nombre de calculs qu'il nécessite.

2.3.2 comparaison des politiques résultantes

En utilisant les mêmes paramètres que précédemment, nous allons maintenant nous intéresser à comparer les politiques obtenues par ces deux algorithmes. Pour cet exemple, les carrés vert (état final) et bleus clairs ont une récompense de +1, les carrés rouges (état final) et roses de -1 et les carrés gris de -0.001.



(a) Carte de jeu utilisée. Le point bleu correspond à notre agent au début du jeu, le point rouge à un état final avec récompense -1 et la point vert à un état final avec récompence $+1$.

```
Value iteration
[[3 3 3 0]
 [1 1 2 0]
 [1 2 2 0]]
Policy iteration
[[3 3 3 0]
 [1 1 2 0]
 [1 2 2 0]]
```

(b) Politique obtenue sur la carte de jeu 0, avec les paramètres précédents



(c) Carte de jeu utilisée. Le point bleu correspond à notre agent au début du jeu, le point rouge à un état final avec récompense -1 et la point vert à un état final avec récompence $+1$.

```
Value iteration
[[3 3 3 3 0 1 0 2 2 0]
 [3 3 0 1 0 1 0 1 2 0]
 [3 3 3 3 3 3 1 1 1 0]
 [3 3 1 1 1 1 1 1 1 0]
 [3 3 1 1 1 1 1 1 1 2]
 [1 1 1 1 1 1 1 1 1 1]
 [3 3 0 0 0 1 3 3 1 0]
 [3 3 3 0 0 1 3 3 1 1]
 [1 1 0 0 0 1 1 1 1 1]
 [3 3 3 3 3 3 3 3 3 1]]
Policy iteration
[[3 3 3 3 0 1 0 2 2 0]
 [3 3 0 1 0 1 0 1 2 0]
 [3 3 3 3 3 3 1 1 1 0]
 [3 3 1 1 1 1 1 1 1 0]
 [3 3 1 1 1 1 1 1 1 2]
 [1 1 1 1 1 1 1 1 1 1]
 [3 3 0 0 0 1 3 3 1 0]
 [3 3 3 0 0 1 3 3 1 1]
 [1 1 0 0 0 1 1 1 1 1]
 [3 3 3 3 3 3 3 3 3 1]]
```

(d) Politique obtenue sur la carte de jeu 5, avec les paramètres précédents

FIGURE 15 – Politique obtenue sur deux environnements pour les algorithmes d’itérations sur la politique et d’itération sur la valeur. Les politiques obtenues sont identiques.

En suivant cet encodage des actions, on obtient les politiques suivantes sur les cartes numérotées 0 et 5 par exemple.

Les deux algorithmes calculent également une politique identique pour les autres cartes proposées.

Conclusion

Ce TP focalisé sur les algorithme de prévision de la politique lorsque le MDP est connu, aura présenté deux algorithmes permettant de résoudre ce problème. Bien que

les solutions obtenues soient très similaire, la version d’itération sur les valeurs possède un avantage computationnel appréciable. On note également, qu’il est possible d’influencer la politique résultante en ajustant les différents paramètres (du monde mais également de l’algorithme). Cela permet d’adapter la politique au problème à résoudre, puisque dans certains contextes il sera préférable d’avoir des agents très fiables et prudents, quand dans d’autres situations, la rapidité de résolution sera privilégiée.

3 TP3 : Q-Learning

Dans ce TP, on s'intéresse à l'implémentation de la méthode de renforcement basée sur la fonction $Q : (a, s) \mapsto \mathbb{E}[R_t | s_t = s, a_t = a]$ la plus élémentaire : Q-learning. L'idée est d'actualiser progressivement notre connaissance du MDP à l'aide de la fonction Q à partir d'une politique initiale aléatoire. On agit en entraînement de façon ε -greedy afin de garantir une convergence pas trop rapide. En test on agit en suivant la politique définie par $\pi(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q(s, a)$. Comme dans le TP2, on utilise l'environnement Gridworld.

3.1 Plan 1

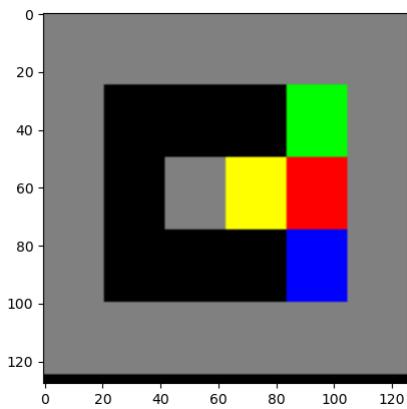


FIGURE 16 – Gridworld plan 1

Sur cette carte (et dans les suivantes), le carré jaune correspond à un reward de $+1$ sans être un état terminal. C'est donc un bonus qu'il est souvent nécessaire d'obtenir pour optimiser les récompenses (quand la pénalité en reward à chaque pas de temps est suffisamment petite). Le carré vert est un état terminal de reward $+1$ et le carré rouge un état terminal de reward -1 .

L'algorithme Q-learning a réussi sans problèmes à trouver la trajectoire optimale sur le plan 1, comme l'attestent les rewards ci-dessous. Le nombre d'états étant particulièrement réduit, il est facile de récupérer "par hasard", le bonus jaune et donc de l'inclure dans toutes les futures trajectoires.

En modifiant le paramètre de discount γ (en le baissant de 0.99 à 0.5), on rend l'apprentissage plus lent et plus difficile. On observe en particulier une augmentation de la variance des rewards due à la plus forte dépendance de la trajectoire aux premiers parcouru par l'agent. En effet le discount étant faible, très rapidement les états rencontrés n'auront plus d'impact sur le reward final de la trajectoire.

3.2 Plan 4

Dans ce plan apparaît un carré rose, qui est état non terminal de reward -1 . C'est un malus que l'on cherche à éviter quand c'est possible. Cependant, puisque nous connaissons le MDP (contrairement à l'algorithme) nous voyons que sur le plan 4, il

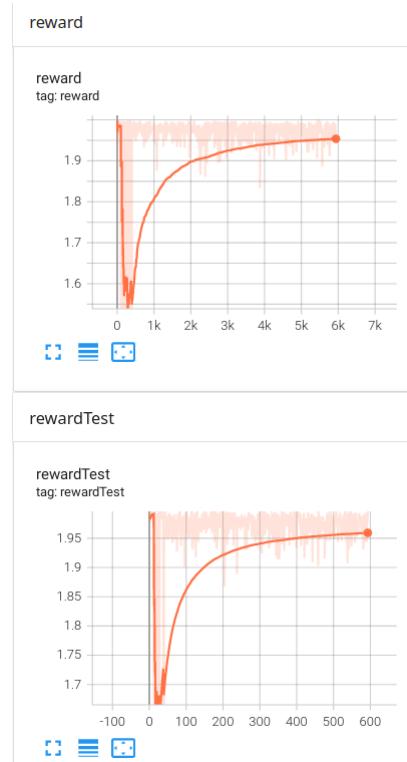


FIGURE 17 – Rewards de Q-learning sur le plan 1 en entraînement (en haut) et en test (en bas).

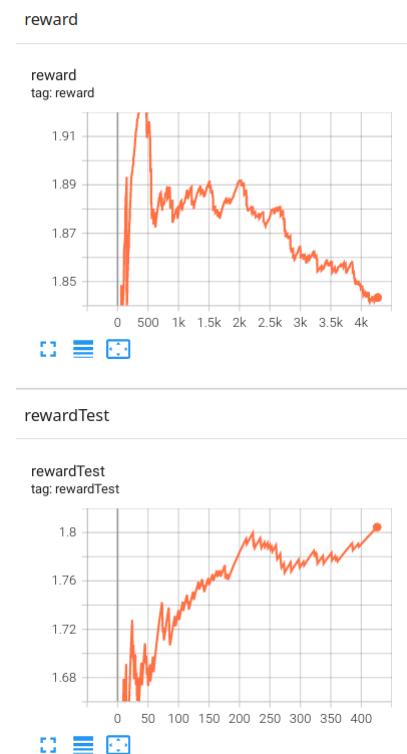


FIGURE 18 – Rewards de Q-learning sur le plan 1 et un discount de 0.5 en entraînement (en haut) et en test (en bas).

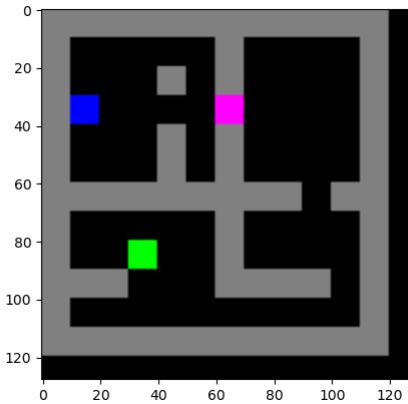


FIGURE 19 – Gridworld plan 4

est obligatoire de récupérer le malus rose pour atteindre l'état vert terminal et donc maximiser la récompense.

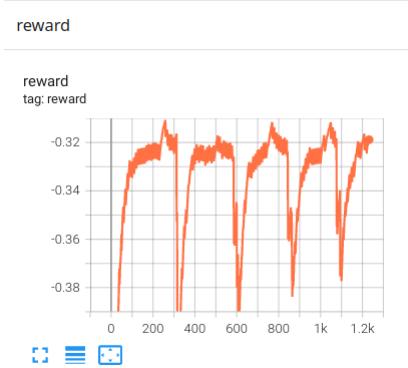


FIGURE 20 – Rewards de Q-learning sur le plan 4 en entraînement (en haut) et en test (en bas).

Malheureusement Q-learning peine à atteindre l'état terminal vert (aucune trajectoire ne termine naturellement, en rencontrant un état terminal) en récupérant le malus rose, car d'une part l'état vert est trop loin de l'état initial pour être trouvé par exploration hasardeuse sur peu d'itérations, d'autre part la présence du malus rose sur toutes les trajectoires menant à l'état vert, condamne toutes ces trajectoires en diminuant la valeur des $Q(a, s)$ tels que $s' = \text{malus rose}$. Sur la courbe des rewards, les pics descendants correspondent aux trajectoires de test, plus longues. Il faut en effet considérablement augmenter le nombre d'epoch en entraînement afin d'atteindre l'état final intéressant. Une fois ce dernier atteint, les trajectoires sont rapidement mises à jour vers la politique intéressante (21a et 21b).

3.3 Plan 5

Sur ce plan, Q-learning va trouver facilement une solution, mais sous optimale, car le hasard n'est pas suffisant pour explorer aussi profondément que la position du carré jaune le MDP. De plus, assez rapidement, l'exploration est limitée car les actions menant au carré vert sont favorisées.

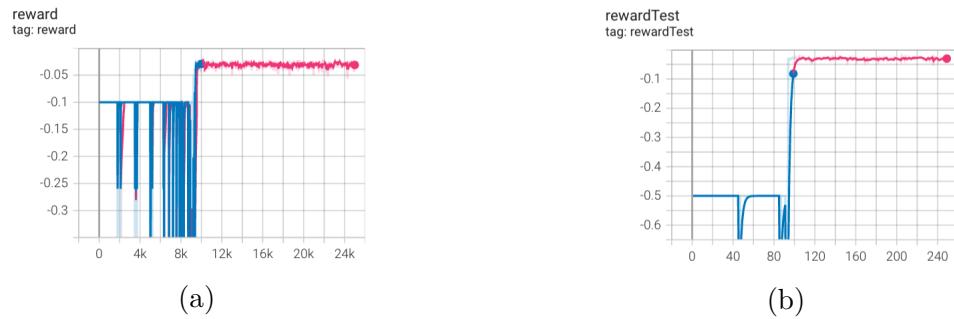


FIGURE 21 – Récompenses en entraînement et en test pour Q-learning sur le plan4

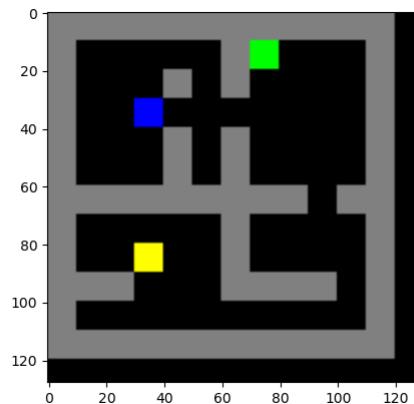


FIGURE 22 – Gridworld plan 5

3.4 Plan 8

Dans ce plan, le nombre d'état est bien plus grand que dans les plans précédents mais ne présente pas d'autres difficultés majeures pour l'apprentissage. Cependant, nous sommes contraints d'augmenter substantiellement la longueur maximale des trajectoires car autrement on n'atteint jamais l'état terminal vert en explorant au hasard. Nous sommes face à un cas de récompense parcimonieuse, pour espérer apprendre, il faut croiser au moins une fois l'état terminal vert.

Sur les rewards à trajectoires courtes on observe qu'on ne rencontre jamais l'état vert terminal, cela n'est pas surprenant comme expliqué au dessus. Cependant, les rewards pour les trajectoires longues sont plus intéressants : on observe d'abord une stagnation, jusqu'à environ 1000 trajectoires. Cela correspond à la phase d'exploration durant laquelle on n'a pas encore rencontré l'état vert. Puisque l'on se repose uniquement sur le hasard pour le trouver, au vu de la taille du MDP, il est normal qu'il faille un certain nombre de trajectoires avant de le rencontrer. Mais dès que l'on a rencontré l'état vert une fois, Q-learning apprend rapidement et fini par déterminer la trajectoire optimale.

Un autre possibilité pour aider l'algorithme à trouver la stratégie optimale est d'augmenter de façon significative le nombre d'itération d'entraînement. A force d'exploration, Qlearning finit par trouver l'état final et met rapidement à jour sa politique d'exploration.

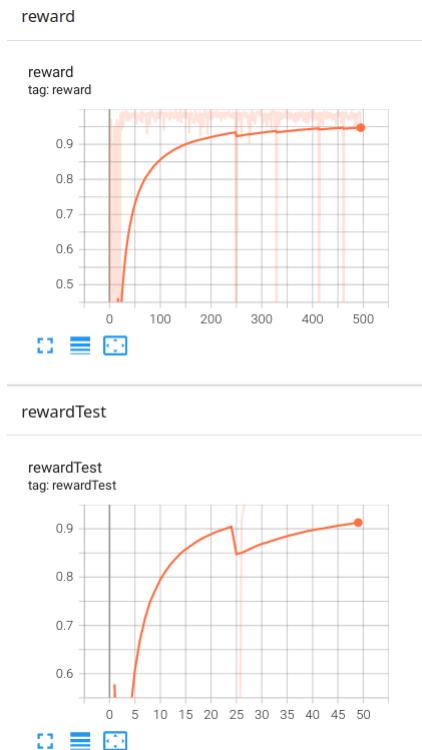


FIGURE 23 – Rewards de Q-learning sur le plan 5 en entraînement (en haut) et en test (en bas).

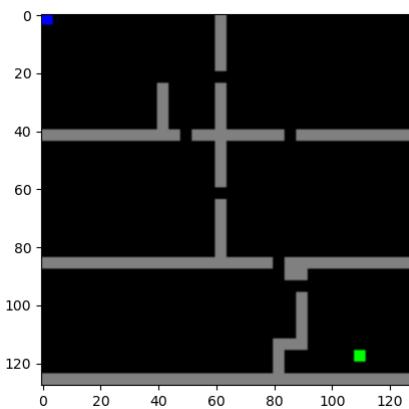


FIGURE 24 – Gridworld plan 8

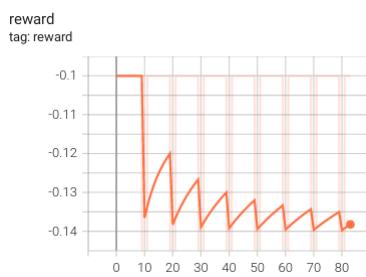


FIGURE 25 – Rewards de Q-learning sur le plan 8 avec des trajectoires de longueurs maximales courtes en entraînement (en haut) et en test (en bas).

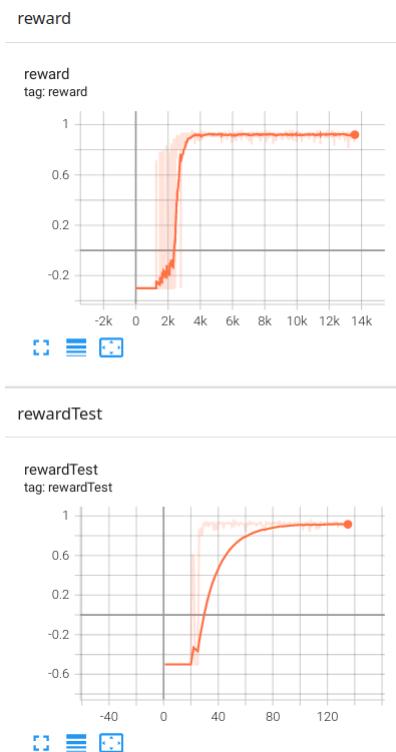


FIGURE 26 – Rewards de Q-learning sur le plan 8 avec des trajectoires de longueurs maximales longues en entraînement (en haut) et en test (en bas).

4 TP4 : DQN

Ce TP propose d'implémenter l'algorithme DQN. Cet algorithme propose de modéliser la fonction Q (utilisée pour extraire une politique de jeu) par un réseau de neurones qui apprendra à partir de données générées par l'agent lorsqu'il joue (les rewards obtenues selon l'action choisie dans un état donné).

4.1 Référence

Dans un premier temps, il est important de trouver des hyperparamètres satisfaisants pour notre algorithme. Nous retiendrons la configuration suivante sur l'environnement cartpole pour l'algorithme qui servira "de référence" dans les sections suivantes.

- Une stratégie d'exploration ϵ -greedy avec un seuil d'exploration de 0.1, qui décroît avec un facteur 0.9 à chaque itération.
- Un paramètre de discount de 0.9999.
- Un pas d'apprentissage de 0.003 pour l'optimiseur Adam.
- L'utilisation d'un réseau cible que l'on optimise toutes les 10 étapes .
- Une mémoire "prioritized" de taille 10000 dont on extrait des mini-batches de taille 100.
- Un réseau Q avec une couche cachée avec 200 neurones.

Les résultats de DQN avec cet ensemble de paramètres sont présentés figure 27a à 27c.

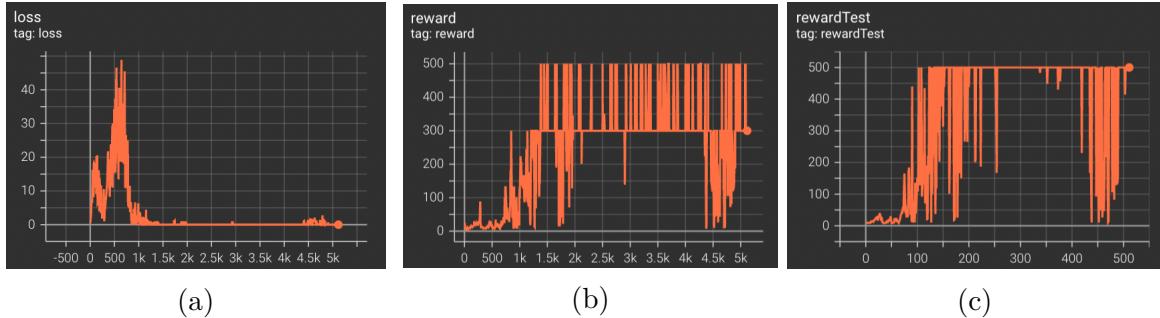


FIGURE 27 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues avec les paramètres précisés ci-dessus.

Ces graphiques nous montrent que l'agent apprend plutôt rapidement à atteindre le niveau de récompense maximal (à peu près 1500 epochs) et met ensuite à peu près 500 epochs supplémentaires à se stabiliser. Notons également l'instabilité observée autour de l'epoch 4500. L'agent a brusquement changé de politique (probablement dû à un coup aléatoire de la stratégie d'exploration). Ce changement est également visible sur la loss.

4.2 Etude des paramètres de l'algorithme

Plusieurs tests ont été effectués, afin d'évaluer l'influence de tous les paramètres. Dans les sections dédiées aux TP suivants, toutes ces expériences ne seront pas systématiquement refaites afin de se focaliser sur les spécificités des algorithmes.

4.2.1 Exploration

Pour ce test, le paramètre d'exploration a été fixé à 0.05. Le taux d'exploration initial est donc de 5% et va rapidement diminuer.

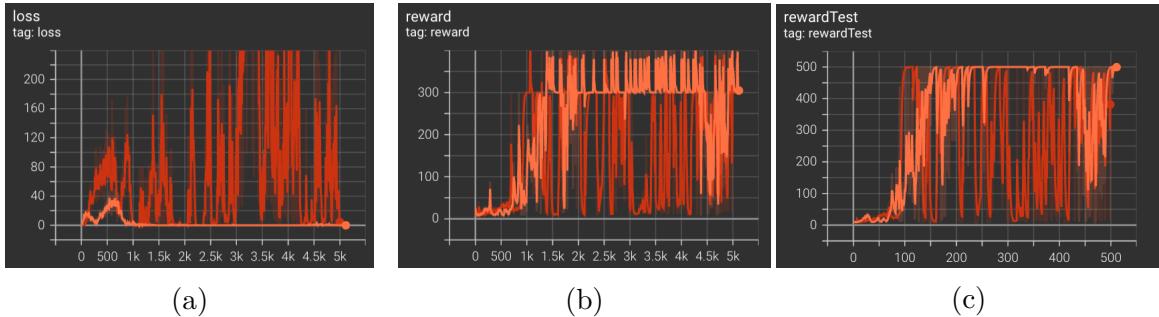


FIGURE 28 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues avec un coefficient d'exploration initial de 0.05. La courbe orange correspond à notre référence et la courbe rouge à l'entraînement modifié.

On observe une grande instabilité sur toutes les courbes (28a à 28c). Bien que l'agent réussisse à apprendre et à atteindre les récompenses maximales (légèrement avant la référence), on observe une grande variance sur les récompenses obtenues, mais également sur la loss, qui peine à se stabiliser.

4.2.2 Amortissement de l'exploration

Pour ce test, le paramètre qui amortit l'exploration a été abaissé à 0.8 (29a à 29c). Cela a pour conséquence que l'agent diminue beaucoup plus vite son exploration. Les résultats sont présentés (29a à 29c).

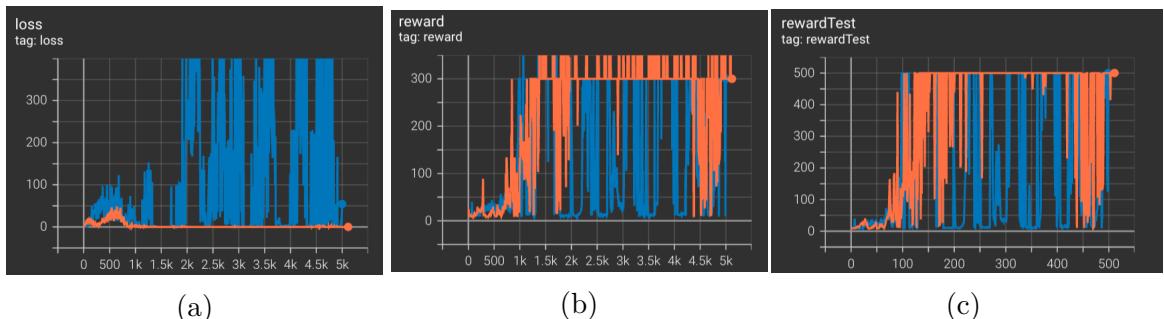


FIGURE 29 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues avec un decay d'exploration de 0.8. La courbe orange est la référence et la courbe bleue l'entraînement avec le paramètre modifié.

Les conclusions sont les mêmes que pour le paramètre d'exploration développé à la section précédente. En effet, le decay modélise également la curiosité de l'agent envers son monde, et il est donc naturel d'observer les mêmes phénomènes lorsqu'il est diminué.

4.2.3 Paramètres de discount

Pour ce test, le paramètre γ a été abaissé à 0.9 (0.9999 pour la référence). Les trajectoires lointaines sont donc moins d'importance dans le calcul.

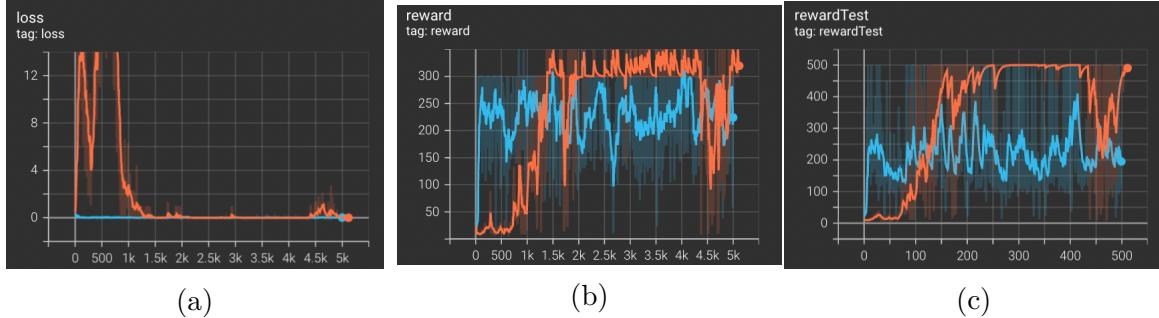


FIGURE 30 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues avec un coefficient γ de 0.9. La courbe orange correspond à notre référence et la courbe bleue à l’entraînement modifié.

Le paramètre γ modélise l’influence des trajectoires futures sur la prédiction. Autrement dit, il pondère l’espérance des gain dans le calcul des observation à une itération donnée. Lorsque ce paramètre diminue, on accorde moins d’importance à ces quantités. On note alors que l’algorithme converge (la convergence est assez rapide), mais montre des difficultés à atteindre les récompenses maximales (30c et 30b) (qui nécessite de jouer pendant longtemps). La récompense atteinte est d’environ 200.

4.2.4 Initialisation du pas d’apprentissage

Pour ce test, le pas d’apprentissage a été légèrement modifié à 0.001 (0.003 pour la référence). Il sert à l’initialisation de l’algorithme ADAM utilisé ici.

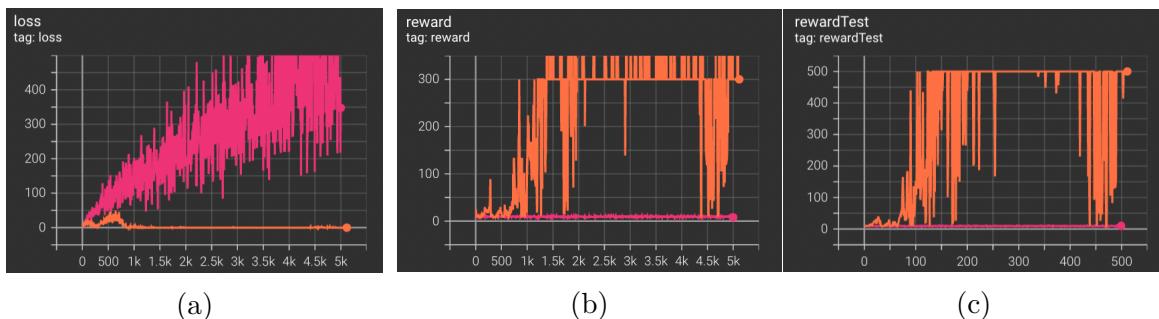


FIGURE 31 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues avec un learning rate initial de 0.001. La courbe orange correspond à notre référence et la courbe rose à l’entraînement modifié.

L’infime modification apportée au pas d’apprentissage initiale ne permet pas à l’algorithme de converger. On note que dès le début, la loss (31c) croît de façon très instable. Les récompenses restent minimales car l’agent ne réussit pas à apprendre une quelconque stratégie pertinente (il ne réussit pas à faire baisser la loss).

4.2.5 Fréquence d'optimisation du réseau cible

Pour ce test, le réseau cible n'est mis à jour que toutes les 30 itérations (au lieu de 10).

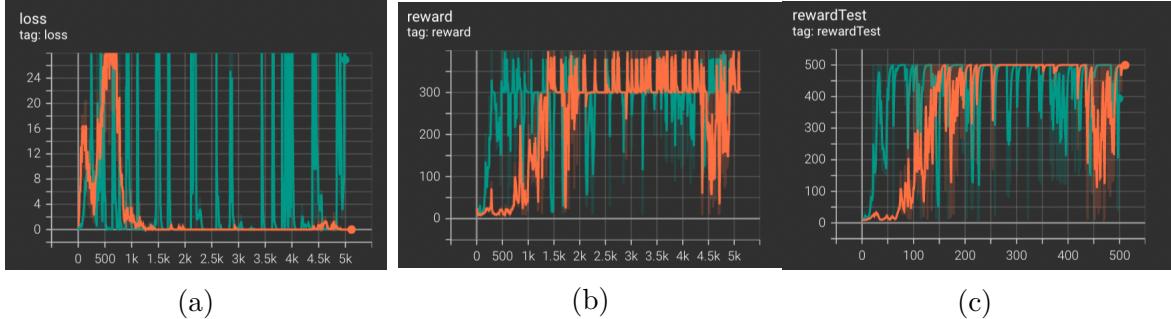


FIGURE 32 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues en mettant à jour le réseau cible toutes les 30 epochs. La courbe orange correspond à notre référence et la courbe verte à l'entraînement modifié.

Diminuer la fréquence d'optimisation du réseau cible montre encore une fois la grande instabilité de l'algorithme, que ce soit sur la loss ou sur les valeurs de récompenses. En revanche, l'algorithme converge plus rapidement vers la stratégie lui permettant d'atteindre les récompenses maximales ([32a](#) à [32c](#)).

4.2.6 Mini-batch

Pour ce test, la taille des mini-batchs est augmentée à 500.

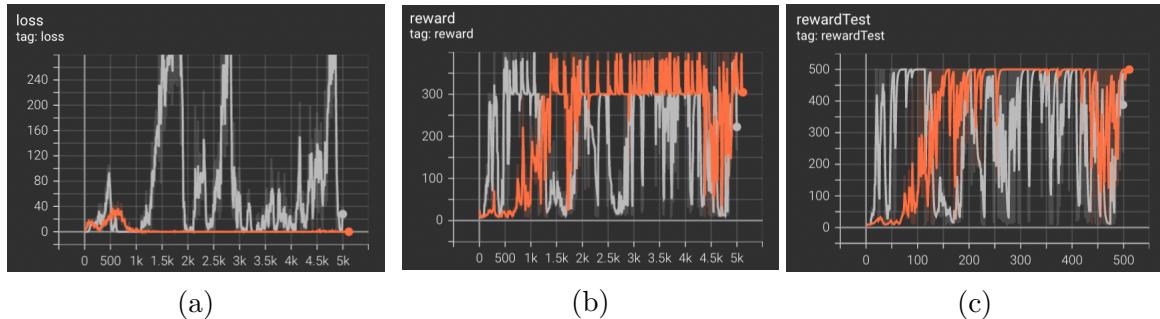


FIGURE 33 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues en utilisant des mini-batchs de taille 500. La courbe orange correspond à notre référence et la courbe grise à l'entraînement modifié.

Premièrement, le temps d'entraînement en utilisant des batchs de cette taille est considérablement augmenté. Ils permettent d'atteindre rapidement un bon score. En revanche, encore une fois, cette modification entraîne une grande instabilité de l'algorithme lors de l'apprentissage.

4.3 Comparaison des différentes versions des algorithmes

4.3.1 DQN simple

Cette version de DQN est la version la plus simple, c'est à dire qu'elle n'a pas de réseau cible, ni de prioritized replay buffer, c'est-à-dire que la mémoire utilisée sélectionne de façon aléatoire les transitions du mini-batch sans ajouter de pondération supplémentaire.

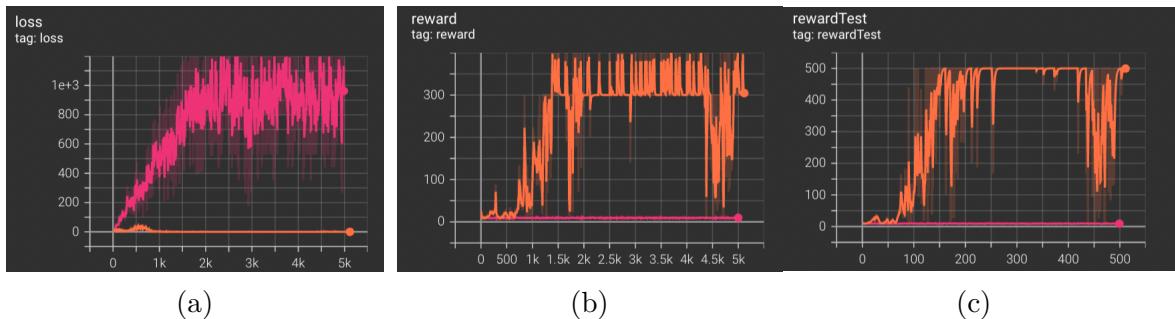


FIGURE 34 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues sur le DQN simple. La courbe orange correspond à notre référence et la courbe rose à l'entraînement modifié.

Cette version de l'algorithme obtient de très mauvais résultats. En effet, la loss augmente et se stabilise, mais l'agent n'apprend rien de pertinent lui permettant d'améliorer ses résultats sur le jeu.

4.3.2 Target Network

Dans cette version, le réseau cible est ajouté. Il permet de stabiliser les valeurs cibles pour l'apprentissage. Car en effet, dans le cadre de l'apprentissage par renforcement, les valeurs cibles du réseau sont modifiées à chaque itération puisqu'elles sont générées à partir de ce même réseau (qui est mis à jour à chaque appel de la fonction `learn`). L'ajout d'un réseau cible, qui sera mis à jour moins régulièrement et qui sera utilisé pour générer les valeurs cibles du réseau apprenant, permet de stabiliser ces données et donc d'améliorer l'apprentissage.

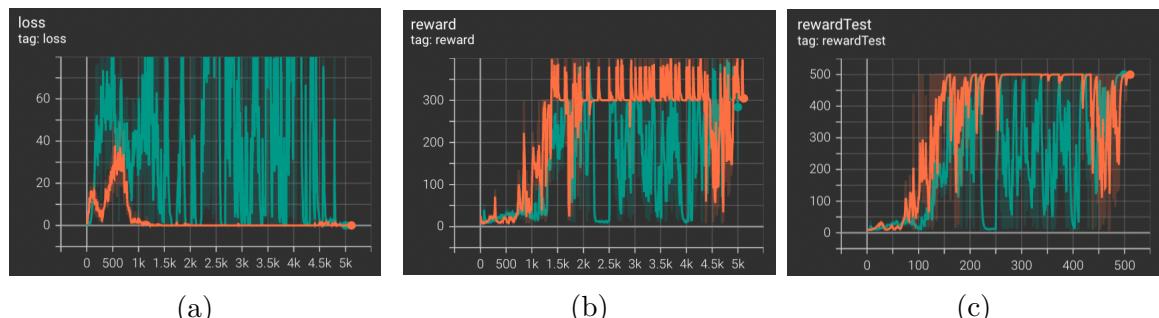


FIGURE 35 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues en utilisant un réseau target. La courbe orange correspond à notre référence et la courbe verte à l'entraînement modifié.

On observe sur ces courbes une nette amélioration par rapport à la version proposée à la section précédente. En effet l'apprentissage est plus performant car les récompenses atteintes par l'algorithme sont largement plus élevées et la loss (35a) n'augmente pas. En revanche, on note une grande instabilité des courbes, l'algorithme semble avoir des difficultés à trouver une stratégie performante.

4.3.3 Prioritized Replay

Dans cette section, nous présentons le DQN avec prioritized replay buffer. Il s'agit de gérer la mémoire des trajectoires de façon plus intelligente en prenant en compte leur écart à la valeur cible au moment de les échantillonner. Cela permet d'avoir des batch d'exemples plus pertinents.

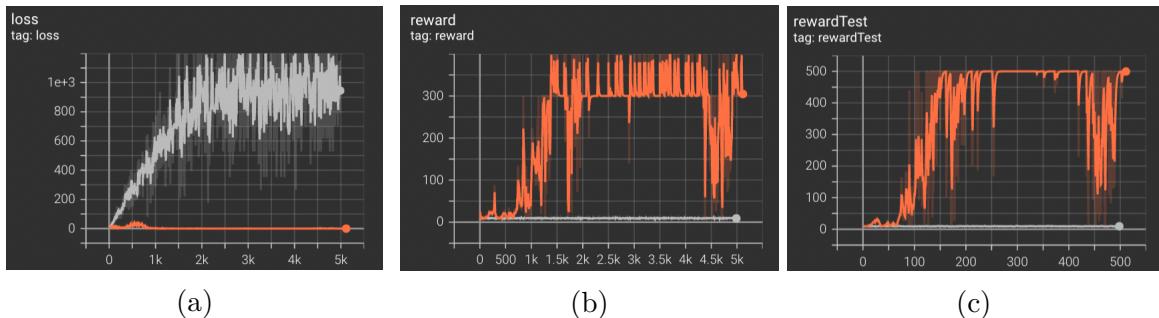


FIGURE 36 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues en utilisant des mini-batchs de taille 500. La courbe orange correspond à notre référence et la courbe grise à l'entraînement modifié.

Cette amélioration seule n'apporte pas de nouveaux éléments par rapport à la version simple du DQN.

4.3.4 DQN target + prioritized Replay

Cette version est la version qui a été choisie comme référence pour les tests, et produit les meilleurs résultats en terme de stabilité de l'algorithme. Elle utilise à la fois un réseau cible et une mémoire améliorée. Elle utilise l'ensemble de paramètres démontrant les meilleures performances en termes de stabilité de l'algorithme, mais également de récompenses obtenues.

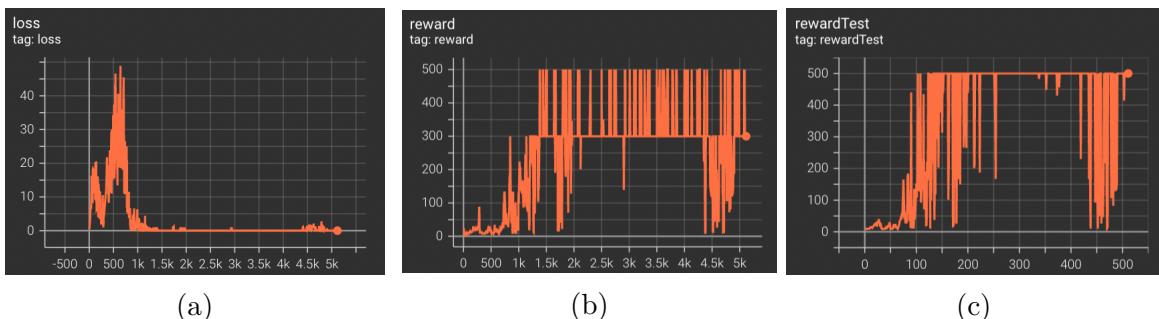


FIGURE 37 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues pour la version la plus intéressante de nos tests.

4.4 Etude des autres jeux : Lunar Lander et Gridworld

Ces deux nouveaux jeux nécessitent d'adapter les hyper paramètres de notre algorithme afin de les voir converger en un temps raisonnable. Sur la configuration utilisée pour la référence, les résultats obtenus sont présentés figures 38a à 38f.

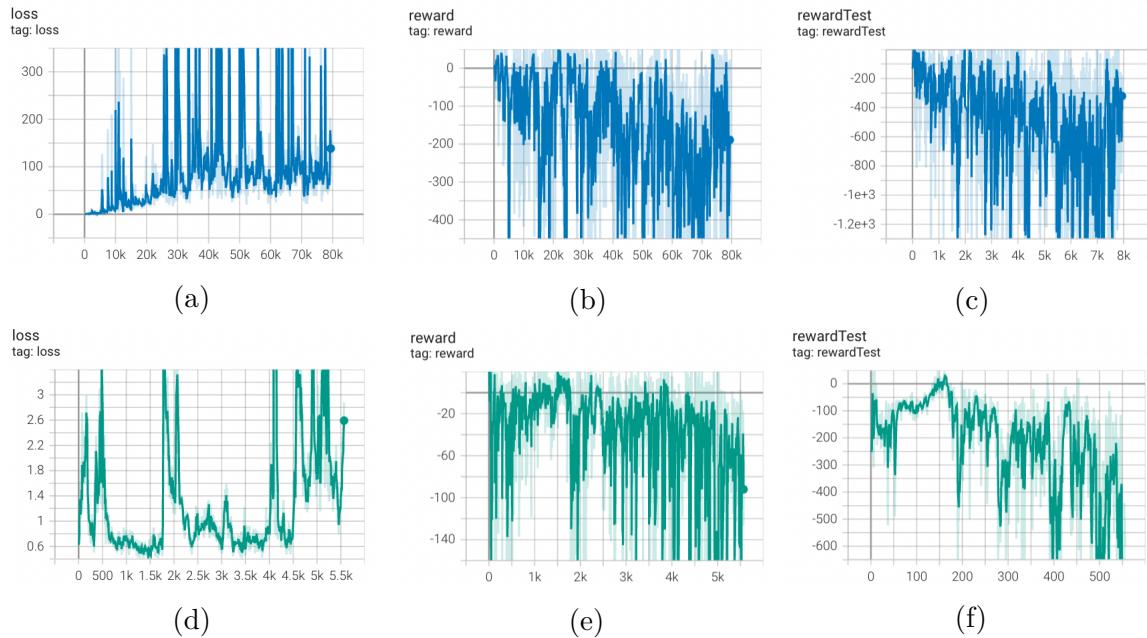


FIGURE 38 – De gauche à droite : Loss, récompenses en entraînement et en test obtenues pour l'environnement lunar lander et notre algorithme DQN. De haut en bas : les paramètres utilisés pour notre référence (haut) et une modification du pas d'apprentissage (0.001 au lieu de 0.003) (bas).

Sur ces courbes, on note une amélioration en modifiant le pas d'apprentissage. En effet sur les figures 38d à 38e, les loss diminuent et les rewards augmentent dans un premier temps, ce qui témoigne d'un certain apprentissage de notre agent (ce qui n'était pas le cas sur la première courbe). En revanche, l'algorithme est très instable et "perd" cet apprentissage rapidement. En ajustant les paramètres précisément à cette tâche, il serait possible d'apprendre une politique intéressante pour résoudre le jeu.

Le jeu gridworld est plus lent à tourner avec DQN et observe des difficultés à apprendre une politique de jeu optimale avec les paramètres de références.

5 TP5 : Policy Gradient

Ce TP a pour objectif d’expérimenter les approches de renforcement Policy Gradients et en particulier l’algorithme Actor-Critic (A2C). Les courbes présentées dans les figures suivantes (39a à 42b) ont été générées en utilisant le jeu de paramètres suivant :

- Un paramètre de discount de 0.99.
- Un pas d’apprentissage de 0.001 pour l’optimiseur Adam pour les réseaux de l’acteur et la critique.
- Une mémoire de taille 1000 dont on extrait une trajectoire complète.
- Un réseau critique (V) avec deux couches cachées de 30 neurones et une activation tanh (pas d’activation finale).
- Un réseau acteur (politique) avec deux couches cachées de 30 neurones, une activation tanh et une activation finale *SoftMax*.

5.1 Cartpole

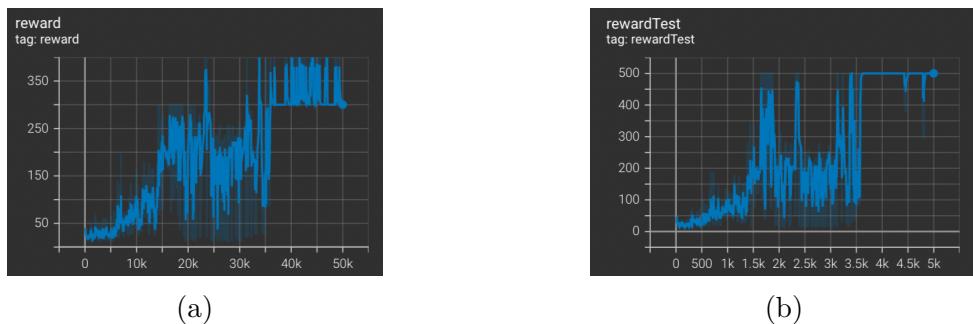


FIGURE 39 – De gauche à droite : Récompenses en entraînement et en test obtenues avec notre implémentation de A2C sur l’environnement Cartpole.

Cartpole est un environnement très simple et à partir d’environ 35k trajectoires, on atteint un plateau maximal pour les rewards. Ce qui correspond à des trajectoires de longueurs maximales (300 pas de temps). L’atteinte de ce plateau est rassurante dans la mesure où les rewards ne diminuent plus jamais, A2C a appris *définitivement* à résoudre le problème Cartpole de manière optimale. Sur les figures ci dessus, on observe les résultats pour plusieurs expériences menées indépendamment. Pour chaque expérience, les rewards sont croissants et finissent par atteindre le plateau, cela renforce notre confiance sur la robustesse d’A2C.

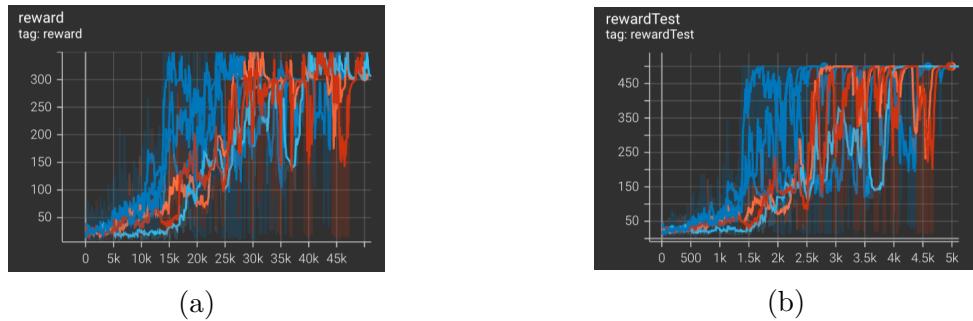


FIGURE 40 – De gauche à droite : Récompenses en entraînement et en test obtenues avec notre implémentation de A2C sur l'environnement Cartpole sur plusieurs expériences.

5.2 Lunar



FIGURE 41 – De gauche à droite : Récompenses en entraînement et en test obtenues avec notre implémentation de A2C sur l'environnement Lunar Lander.

Les rewards obtenus croissent avec le nombre de trajectoires. Cependant ils ne semblent pas encore bien stabilisés, cela est peut-être dû à la complexité de l'environnement Lunar. Nous n'avons que 50k trajectoires et nous avons vu que Cartpole seulement en nécessitait de l'ordre de 35k pour converger.

5.3 Gridworld



FIGURE 42 – De gauche à droite : Récompenses en entraînement et en test obtenues avec notre implémentation de A2C sur la carte 1 de Gridworld.

Le reward maximal théorique pour la carte 1 de Gridworld est proche de 2 (le reward de la case jaune et celui la case verte finale), nous peinons à l'atteindre en entraînement malgré 50k trajectoires de rewards croissant bien que nous nous en approchions plus en test (plus important). Encore une fois, Gridworld est plus complexe que Cartpole, il n'est pas surprenant que l'on ai pas encore trouvé la trajectoire optimale en 50k trajectoires, au vu des résultats précédents.

5.4 Stratégies de mise à jour

Dans notre algorithme, nous considérons une version Rollout Monte-Carlo, c'est-à-dire que ce sont les rewards cumulées sur une trajectoire qui sont utilisées pour mettre à jour les réseaux V et procéder au calcul des avantages. Plusieurs autres possibilités existent : utiliser des version TD(0) ou TD(λ). Dans un soucis d'implémentation, nous n'utilisons qu'une seule trajectoire complète par batch, ce qui est source de grande variabilité dans les rewards obtenues. C'est également un point qui pourrait être sujet à amélioration.

6 TP6 : Advanced Policy Gradients

Dans ce TP, on s'intéressera à l'algorithme PPO (avec KL divergence). Les hyperparamètres utilisés dans un premier temps sont les suivants :

- Un paramètre de discount de 0.999.
- Un pas d'apprentissage de 0.01 pour l'optimiseur Adam pour les réseaux de l'acteur et la critique.
- Une mémoire de taille 1000 dont on extrait une trajectoire complète.
- Un réseau critique (V) avec deux couches cachées de 30 neurones et une activation tanh (pas d'activation finale).
- Un réseau acteur (politique) avec deux couches cachées de 30 neurones, une activation tanh et une activation finale *SoftMax*.
- 10 optimisations à chaque itération de l'algorithme. Une cible KL de 0.1

Les résultats obtenus sur les environnements Cartpole et Lunar Lander sont proposés dans les figures 43a à 44b.

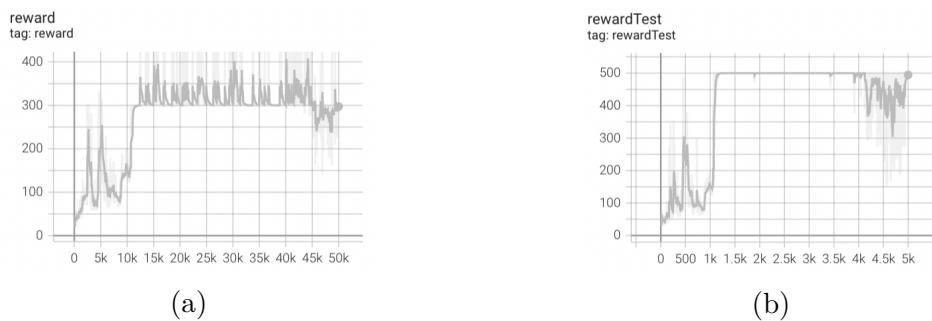


FIGURE 43 – De gauche à droite : récompenses en entraînement et en test obtenues avec notre implémentation de PPO avec KL sur l'environnement Cartpole.

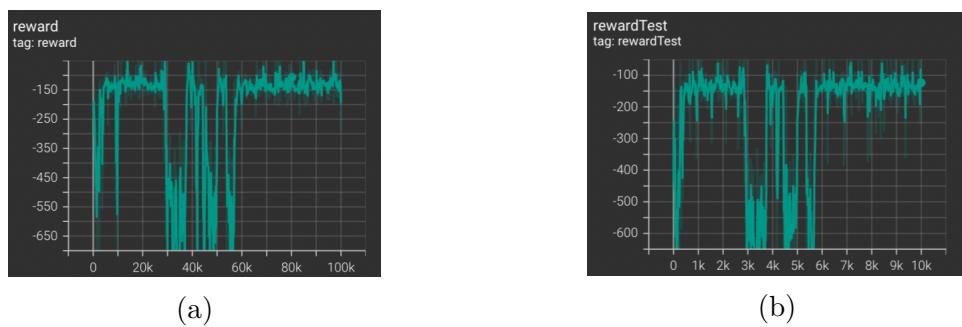


FIGURE 44 – De gauche à droite : récompenses en entraînement et en test obtenues avec notre implémentation de PPO avec KL sur l'environnement Lunar Lander.

Sur l'environnement Cartpole, on note que l'algorithme semble avoir très bien appris à résoudre le problème. Les courbes de récompenses sont très stables, à l'exception près de l'epoch 45000, où un coup aléatoire a probablement perturbé la politique alors en cours. Il faut attendre que l'agent ré apprenne et stabilise à nouveau sa politique. En revanche, l'environnement lunar lander est plus compliqué pour notre algorithme. En

effet, bien qu'il réussisse à apprendre une certaine stratégie de jeu, cette dernière est sous-optimale pour l'environnement. De plus, on observe également sur cet environnement de brusques changements de politiques, se matérialisant par des récompenses plus faibles. Note : Les paramètres utilisés ici sont optimisés pour l'environnement Cartpole. Une étude plus approfondie dédiée à Lunar permettrait probablement d'améliorer grandement les performances et la stabilité de PPO.

6.1 Initialisation

Bien que PPO propose des résultats intéressants sur Cartpole, il présente néanmoins une grande sensibilité aux premiers coups joués et à l'initialisation. En effet, selon les premières trajectoires utilisées, il peut montrer de grandes difficultés pour apprendre une quelconque politique de jeu, comme présenté figures 45a et 45b.



FIGURE 45 – De gauche à droite : récompenses en entraînement et en test obtenues avec notre implémentation de PPO avec KL sur l'environnement Cartpole pour deux initialisations de trajectoires différentes.

Ces courbes présentent deux jeux joués dans les mêmes conditions (mêmes hyper paramètres). On remarque que la courbe rose, réussit à apprendre une politique de jeu, et à l'améliorer jusqu'à atteindre la politique souhaitée, optimale. A l'inverse, sur la seconde courbe (bleue claire), on observe clairement qu'aucun apprentissage n'a été retiré de l'entraînement effectué.

7 TP7 : Continuous actions

Dans cette section, les environnements considérés seront à actions continues (et non discrètes comme précédemment). Nous nous intéresserons à l'algorithme DDPG qui a une structure proche de celle du DQN de la section 4. Les hyper-paramètres suivants permettent d'obtenir les courbes 46a à 48c.

- Un paramètre de discount de 0.95.
- Un pas d'apprentissage de 0.001 pour l'optimiseur Adam pour le réseau Q et 0.003 pour le réseau de la politique.
- Une mémoire de taille 10000 dont on extrait un batch de taille 100. Les rewards sont divisées par 1000.
- Un réseau pour la fonction Q avec deux couches cachées de 10 neurones et une activation *leakyRelu* (pas d'activation finale).
- Un réseau acteur (politique) avec deux couches cachées de 10 neurones, une activation *leakyRelu* et une activation finale tanh.
- Deux réseaux cibles pour la politique et la fonction Q, mis à jour toutes les 10 étapes.
- Un bruit Orn-Uhlen de paramètres $\mu = 0$ et $\sigma = 0.2$.

7.1 Pendulum

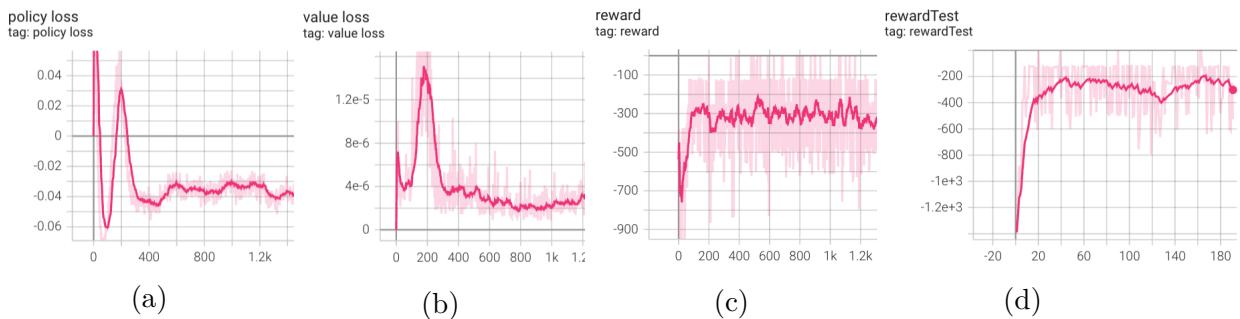


FIGURE 46 – De gauche à droite : Loss pour l'acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de DDPG sur l'environnement à actions continues Pendulum.

DDPG converge assez rapidement (seulement 200 trajectoires) vers une solution optimale pour l'environnement Pendulum.

7.2 Lunar Lander

Notons que l'environnement Lunar Lander se distingue ici de celui considéré dans le TP5 par exemple, car l'espace des actions possible est désormais continu. Dans ce TP Lunar Lander est donc plus complexe à apprendre que dans les précédents.

DDPG converge en seulement une centaine de trajectoires vers une politique, qui n'est pas optimale. Nous obtenions des rewards valant 100 dans le TP5 avec A2C, contre 0 ici. Obtenir des rewards plus faibles était prévisible étant donné la complexité accrue de l'environnement.

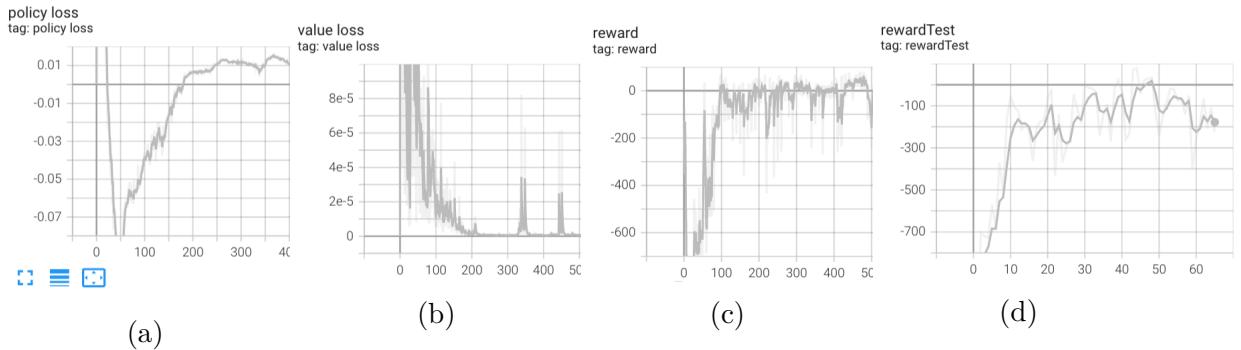


FIGURE 47 – De gauche à droite : Loss pour l’acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de DDPG sur l’environnement à actions continues Lunar Lander.

7.3 Continuous Mountain Car

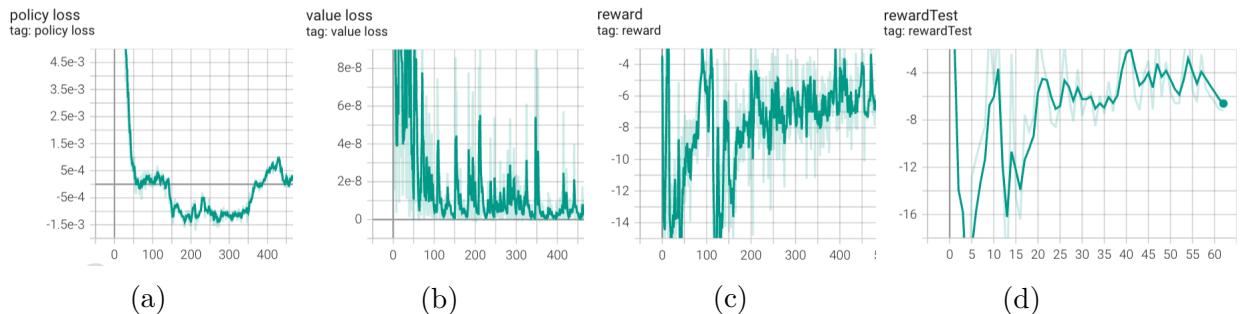


FIGURE 48 – De gauche à droite : Loss pour l’acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de DDPG sur l’environnement à actions continues Continuous Mountain Car.

Cet environnement de jeu est difficile pour l’agent car il s’agit d’un environnement à récompenses parcimonieuses. En effet, à moins d’atteindre le sommet de la montagne et de gagner le jeu grâce à des coups aléatoires, l’algorithme ne dispose que de très peu de retour de son environnement sur les actions qu’il produit.

8 TP8 : Soft-Actor Critic (SAC)

Ce dernier TP propose d'étudier l'algorithme SAC (Soft-Actor Critic) pour environnements continus, en considérant dans un premier temps une température fixe, puis, une température adaptative. Les paramètres suivants ont été utilisés pour produire les résultats suivants :

- Un paramètre de discount de 0.95.
- Un pas d'apprentissage de 0.001 pour l'optimiseur Adam pour le réseau Q et 0.01 pour le réseau de la politique.
- Une mémoire de taille 10000 dont on extrait un batch de taille 100. Les rewards sont divisées par 1000.
- Deux réseaux Q_1 et Q_2 pour la fonction Q avec deux couches cachées de 30 neurones et une activation *leakyRelu* (pas d'activation finale).
- Un réseau acteur (politique) avec deux couches cachées de 30 neurones, une activation *leakyRelu* (pas d'activation finale).
- Deux réseaux cibles pour les fonctions Q, mis à jour toutes les 10 étapes.
- Une optimisation "soft" des réseaux Q avec un paramètre $\tau = 0.1$.
- Un bruit gaussien et un paramètre $\alpha = 0.01$.

Les tests sont effectués sur l'environnement à actions continues Pendulum.

8.1 Température fixe

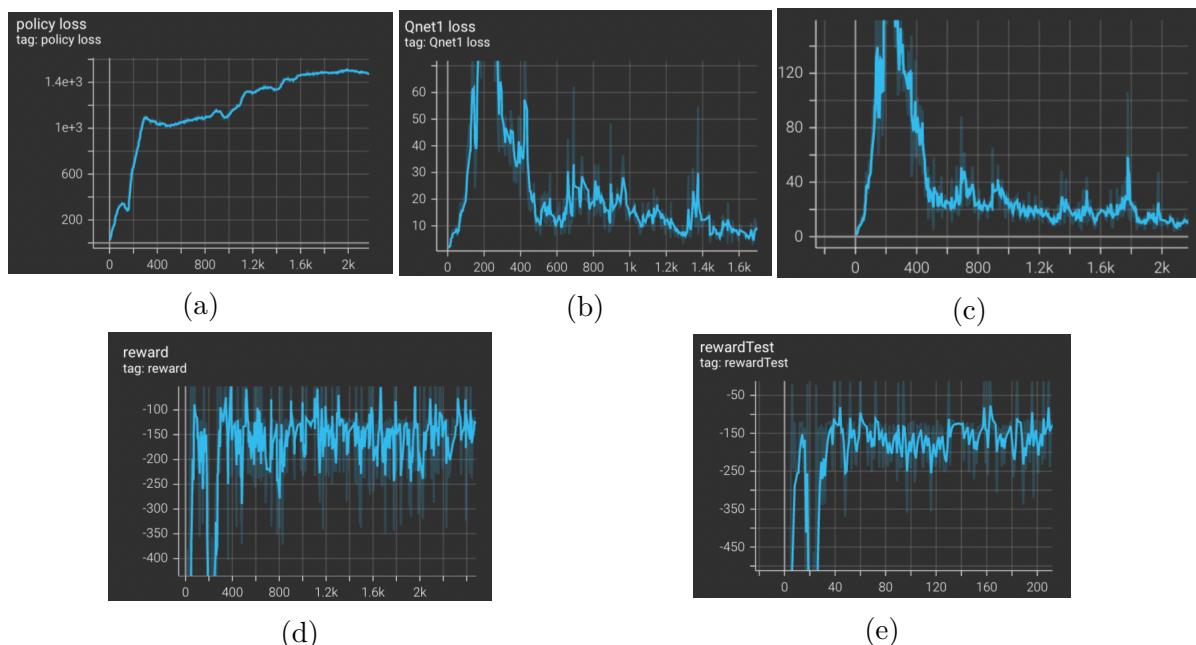


FIGURE 49 – De gauche à droite : Loss pour l'acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de SAC (à température fixe) sur l'environnement à actions continues Pendulum.

Sur cet environnement, l'implémentation de SAC (à température fixe) converge assez rapidement (environ 400 trajectoires) vers une solution optimale pour l'environnement Pendulum.

8.2 Température adaptative

Les résultats présentés figures 50a et 50b utilisent les paramètres de SAC et une initialisation de α à 0.01, un pas d'apprentissage pour α de 0.001 et une entropie visée de 0.2.

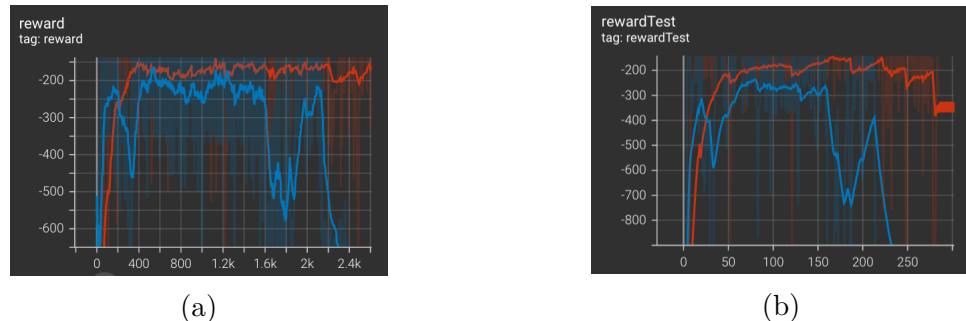


FIGURE 50 – De gauche à droite : récompenses en entraînement et en test obtenues avec notre implémentation de SAC sur l'environnement à actions continues Pendulum.

Sur cet environnement, l'implémentation de SAC (à température adaptative) montre des difficultés à maintenir la politique trouvée. Une recherche plus fine des paramètres optimaux pour cet algorithme pourrait permettre d'obtenir des résultats plus parlant.

9 Comparaison des algorithmes pour environnements à actions discrètes

Dans cette section, une comparaison des différents algorithmes pour les environnements à actions discrètes est proposée. Les paramètres utilisés sont les paramètres optimaux proposés dans les sections 4 à 6.

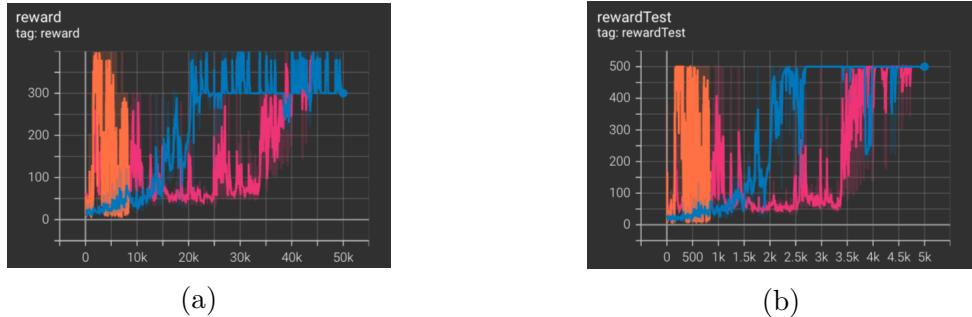


FIGURE 51 – De gauche à droite : récompenses en entraînement et en test obtenues sur l’environnement Cartpole pour les 3 algorithmes étudiés : DQN (orange), A2C (bleu foncé), PPO (rose).

Tout d’abord, l’algorithme DQN semble être le plus rapide à atteindre les récompenses maximales. En revanche, il est relativement instable sur l’entraînement effectué.

Ensuite, les algorithmes A2C et PPO, utilisant tous deux le gradient de la politique, convergent tous les deux vers une politique optimale. La version A2C (20000 itérations) est plus rapide que PPO (45000 itérations) à converger sur notre exemple de comparaison. En revanche, sur les figures 43a et 43b, on remarque que la convergence était intervenue bien plus tôt (< 15000 itérations). On a encore ici une illustration de la sensibilité de nos algorithmes à l’initialisation. PPO présente également un stabilité intéressante sur les stratégies optimales. Une recherche plus précise des paramètres optimaux permettrait également d’améliorer ces résultats.

10 Comparaison des algorithmes pour environnements à actions continues

Dans cette section, une comparaison des différents algorithmes pour les environnements à actions continues est proposée. Les paramètres utilisés sont les paramètres proposés dans les section 7 à 8.

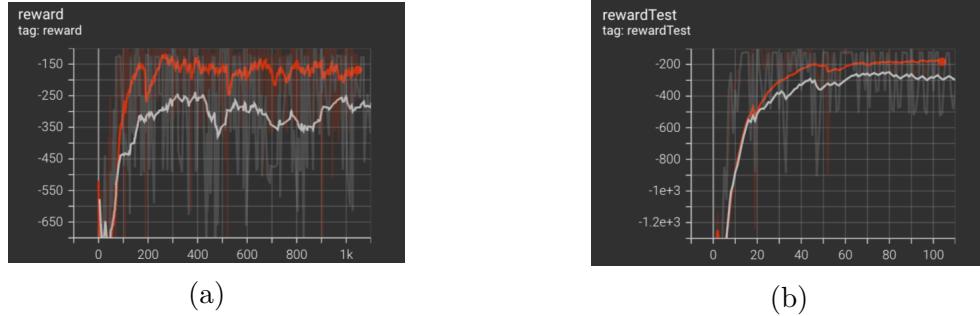


FIGURE 52 – De gauche à droite : récompenses en entraînement et en test obtenues sur l’environnement Pendulum pour les 2 algorithmes étudiés : DDPG (gris) et SAC (rouge)

Ces résultats illustrent les caractéristiques des algorithmes DDPG et SAC. En effet, DDPG montre une convergence plus rapide (200 itérations) mais vers des récompenses plus faibles (-300 en entraînement) que SAC (-200 en entraînement), qui est un peu plus lent à converger (300 itérations).

Conclusion

Dans ce rapport, plusieurs algorithmes de l'état de l'art de l'apprentissage par renforcement sont étudiés. Dans un premier temps, il s'agit des stratégies d'exploration et d'algorithmes spécifiques au cas où le MDP est connu. Ensuite, deux cas sont étudiés, les environnements à actions discrètes (Q-learning, DQN, A2C, PPO) et continues (DDPG et SAC).

En revanche, tous les algorithmes proposés n'ont pas pu être implémentés (différentes variantes de DQN ou encore les traces d'éligibilité par exemple), ou bien n'ont pas convergés (PPO clipped notamment). En effet, la recherche des bons hyper-paramètres permettant de faire apprendre à l'agent une politique de jeu intéressante est ardue et longue car elle nécessite de tester de nombreuses combinaisons d'hyper-paramètres et d'effectuer des entraînements, parfois très lents. Les codes de ces algorithmes sont néanmoins joints à ce rapport. Il s'agit d'ailleurs de la principale difficulté rencontrée dans ces TME car lorsqu'un agent ne réussit pas à apprendre, il est difficile de savoir s'il n'apprend pas car l'implémentation n'est pas correcte ou alors si ce sont les hyper-paramètres qui ne sont pas bons.