



## RLADL : APPRENTISSAGE PAR RENFORCEMENT ET DEEP LEARNING AVANCÉ

---

### Rapport de TMEs

---

*Groupe :*

Hakim CHEKIROU  
Aymen MERROUCHE

*Enseignant :*

Sylvain LAMPRIER

# Table des matières

<b>1 Problèmes de Bandits</b>	<b>4</b>
1.1 Les données . . . . .	4
1.2 Baselines . . . . .	4
1.2.1 Stratégie Aléatoire . . . . .	4
1.2.2 Stratégie Optimale . . . . .	4
1.2.3 Stratégie Static-Best . . . . .	5
1.3 Stratégie . . . . .	5
1.3.1 Greedy . . . . .	5
1.3.2 Stratégie $\epsilon$ -Greedy . . . . .	5
1.3.3 Greedy optimiste . . . . .	6
1.3.4 Upper Confidence Bound UCB . . . . .	6
1.3.5 Stratégie linUCB . . . . .	7
1.4 Comparaison de toutes les stratégies . . . . .	7
1.5 Conclusion . . . . .	8
<b>2 TME 2. Programmation Dynamique</b>	<b>9</b>
2.1 Le MDP . . . . .	9
2.2 Vitesse de convergence . . . . .	9
2.3 Comparaison des politiques obtenues . . . . .	10
2.4 Effet du paramètre discount . . . . .	11
2.5 Étude de cas : Impact des récompenses sur la politique . . . . .	12
2.5.1 Carte 4 et 7 : Agent qui boucle . . . . .	12
2.5.2 Carte 6 : pièce éloignée de l'objectif . . . . .	14
2.5.3 Carte 11 : objectif éloigné . . . . .	15
2.6 Conclusion . . . . .	16
<b>3 TME 3. Q-Learning</b>	<b>17</b>
3.1 Introduction : . . . . .	17
3.1.1 Q-Learning vs SARSA vs Dyna-Q : performances . . . . .	17
3.2 Q-learning vs SARSA vs Dyna-Q : temps de calcul . . . . .	20
3.2.1 Comparaison des politiques d'exploration sur Q-learning : . . . . .	20
<b>4 TME 4. Deep Q network</b>	<b>22</b>
4.1 Introduction . . . . .	22
4.2 DQN . . . . .	22
4.3 Target Network . . . . .	25
4.4 Expérience Replay . . . . .	25
4.5 Prioretized Experience Replay . . . . .	26
4.6 Conclusion . . . . .	27

<b>5 TME 5. Policy Gradients</b>	<b>28</b>
5.1 Introduction . . . . .	28
5.2 Algorithme Batch Actor Critic . . . . .	28
5.3 Version avec TD(0) . . . . .	28
5.3.1 Cartpool . . . . .	28
5.3.2 Lunar Lander . . . . .	29
5.3.3 Grid world . . . . .	29
5.4 Version avec Rollout MonteCarlo . . . . .	30
5.5 Conclusion . . . . .	31
<b>6 TME 6-7. Advanced Policy Gradients</b>	<b>32</b>
6.1 Introduction . . . . .	32
6.2 Algorithme PPO Adaptative KL . . . . .	32
6.2.1 Cartpool . . . . .	32
6.2.2 Lunar Lander . . . . .	32
6.3 KL reversed . . . . .	33
6.3.1 Cartpool . . . . .	33
6.4 PPO Clipped objective . . . . .	34
6.4.1 Cartpool . . . . .	34
6.4.2 Lunar Lander . . . . .	34
6.5 Comparaison . . . . .	35
6.5.1 Cartpool . . . . .	35
6.5.2 Lunar Lander . . . . .	36
6.6 Bonus : Entropie . . . . .	36
6.7 Conclusion . . . . .	37
<b>7 TME 8. Continuous Actions</b>	<b>38</b>
7.1 Introduction . . . . .	38
7.2 Pendulum . . . . .	38
7.3 LunarLander continu . . . . .	38
7.4 Mountain Car . . . . .	39
7.5 Conclusion . . . . .	40
<b>8 TME 9. Generative Adversarial Networks</b>	<b>41</b>
8.1 Introduction . . . . .	41
8.2 Résultats : . . . . .	41
8.2.1 Architecture : . . . . .	41
8.2.2 Hyper-paramètres : . . . . .	43
8.3 Évolution de l'erreur du discriminateur et celle du générateur durant l'entraînement : . . . . .	43
8.4 Évolution de la classification des données réelles et des données générées par le discriminateur : . . . . .	44
8.5 Image générées au long de l'apprentissage : . . . . .	44

8.6 GANs sur le dataset MNIST : . . . . .	45
8.6.1 Architecture : . . . . .	45
8.6.2 Hyper-paramètres : . . . . .	46
8.7 Évolution de l'erreur du discriminateur et celle du générateur durant l'entraînement : . . . . .	46
8.8 Évolution de la classification des données réelles et des données générées par le discriminateur : . . . . .	47
8.9 Image générées au long de l'apprentissage : . . . . .	47
<b>9 TME 10. Variational Auto-Encoders</b>	<b>50</b>
9.1 Introduction . . . . .	50
9.2 Résultats : . . . . .	50
9.3 Architecture : . . . . .	50
9.4 Hyper-paramètres : . . . . .	51
9.5 Évolution de la perte durant l'apprentissage : . . . . .	51
9.6 Effet de la dimension de l'espace latent sur la reconstruction : . . . . .	53
9.7 Analyse de l'espace latent : . . . . .	55
<b>10 TME 11. Multi-Agents RL</b>	<b>58</b>
10.1 Introduction . . . . .	58
10.2 Cooperative navigation : Simple spread . . . . .	58
10.3 Physical déception : simple adversary . . . . .	59
10.4 Predator Prey : Simple Tag . . . . .	59
10.5 Conclusion . . . . .	60
<b>11 TME 12-13. Imitation Learning</b>	<b>61</b>
11.1 Introduction . . . . .	61
11.2 Behavioral Cloning : . . . . .	61
11.3 Generative Adversarial Imitation Learning : . . . . .	62
11.3.1 Expérimentations : . . . . .	62
<b>12 TME 14. Curriculum RL</b>	<b>63</b>
12.1 Introduction . . . . .	63
12.2 DQN avec but : . . . . .	63
12.2.1 Expérimentations : . . . . .	63
12.3 HER : Hindsight Experience Replay . . . . .	64
<b>13 Projet</b>	<b>66</b>
13.1 Introduction . . . . .	66
13.2 Baseline : Random . . . . .	66
13.3 DQN . . . . .	66
13.3.1 DQN vs Random. . . . .	67
13.3.2 DQN vs Statistical. . . . .	67
13.3.3 DQN vs Reactionnary. . . . .	68
13.4 Conclusion . . . . .	68

# 1 Problèmes de Bandits

Pour formaliser le dilemme de l'exploration/exploitation, on prend l'exemple d'un problème de type bandits manchots sous la forme de sélection de publicités. Le problème du choix des annonces à afficher à un utilisateur peut être modélisé par un problème de bandits manchots contextuel, c'est à dire avec des informations sur chaque utilisateur. On a réalisé une campagne d'expériences sur une panoplie de stratégies, avec différentes configurations pour enfin les comparer entre eux pour déterminer la meilleure stratégie.

Avant de commencer les tests, quelques définitions s'imposent :

**Gain** : On considère comme gain en choisissant une annonce, le taux de clics associé.

**Gain optimal** : Le gain optimal pour un article donné est le plus haut taux de clics.

**Regret** : Le regret pour une itération donnée est la différence entre le gain optimal et le gain.

## 1.1 Les données

On dispose d'un fichier décrivant pour 5000 articles, leurs profils (contextes) et le taux de clic sur les publicités de 10 annonceurs. Le but est d'arriver à une stratégie qui maximise le taux de clics.

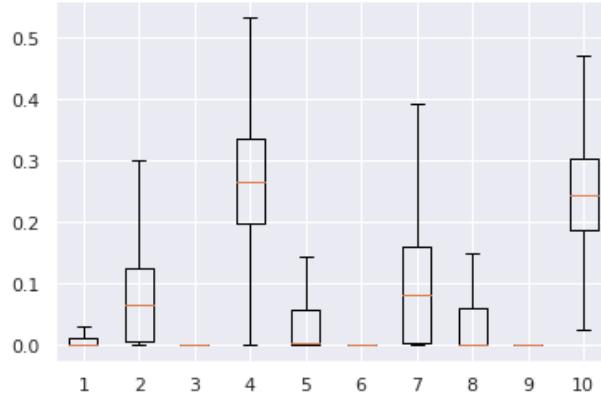


FIGURE 1: Distribution des taux de clics par annonceur

Comme on peut le voir, les taux de clics ne sont pas uniformément distribués.

## 1.2 Baselines

### 1.2.1 Stratégie Aléatoire

Pour chaque article, on tire un annonceur de façon uniforme. C'est la stratégie la moins performante qu'il faudra dépasser.

### 1.2.2 Stratégie Optimale

A chaque itération, on choisit l'annonceur qui a le meilleur taux de clics à cette itération. En temps normal, nous n'avons pas accès à cette information, il s'agit donc d'une borne supérieure qu'on ne pourra jamais dépasser.

### 1.2.3 Stratégie Static-Best

A chaque itération, on choisit l'annonceur avec le meilleur taux de clics cumulés sur toutes les annonces. On ne dispose pas non plus de cette information, c'est une baseline, mais inférieur à la stratégie optimale.

## 1.3 Stratégie

### 1.3.1 Greedy

La stratégie Greedy ou gloutonne se contente d'exploiter à chaque itération les expériences passées. On choisit l'annonceur avec le meilleur taux de clics moyens sur les expériences passées. On l'initialise pendant  $k$  itérations en faisant une sélection aléatoire des annonceurs, c'est la partie exploration. Après ça on se contente d'exploiter les expériences précédentes.

sur la figure suivante, on a affiché les résultats pour différentes valeurs de  $k$ .

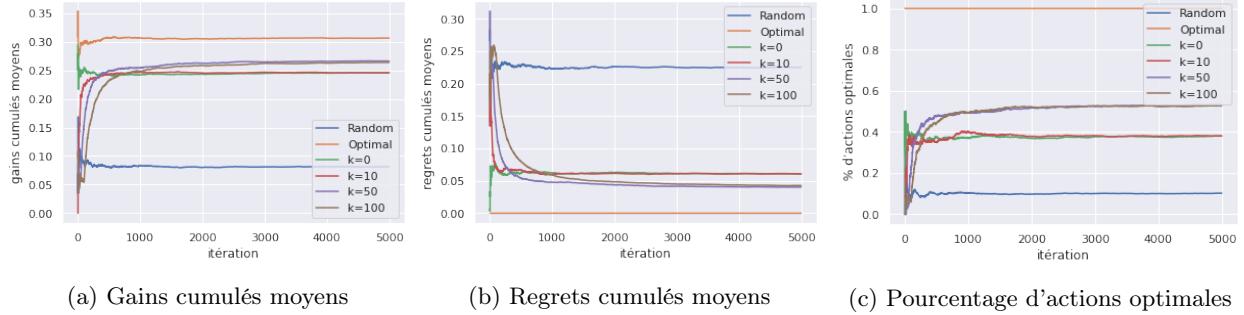


FIGURE 2: Résultats pour la stratégie Greedy sur plusieurs valeurs de  $k$

**Analyse :** Plus  $k$  est élevé, mieux est l'estimation qui est exploitée. Pour  $k$  entre 0 et 10, nous n'arrivons pas à déterminer la meilleure stratégie. Quand on explore pendant 50 itérations, c'est suffisant à atteindre la meilleure stratégie, explorer plus ne fait que ralentir l'algorithme. Si on choisit cet algorithme, il faudra s'assurer que l'estimation est assez bonne pour pouvoir l'exploiter.

### 1.3.2 Stratégie $\epsilon$ -Greedy

Dans cette stratégie, l'agent explore (tirage uniforme) avec une probabilité  $\epsilon$  et exploite la meilleure annonce avec une probabilité de  $1 - \epsilon$ .

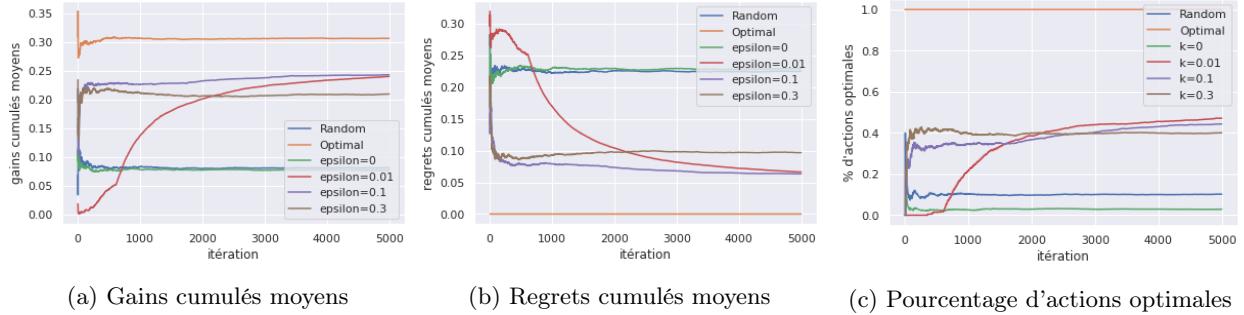


FIGURE 3: Résultats de  $\epsilon$ -greedy pour différentes valeur de  $\epsilon$

**Analyse :**  $\epsilon$  Détermine l'exploration de la méthode, pour  $\epsilon = 0.1$ , l'algorithme arrive à déterminer le meilleur levier, pour une valeur de 0.3, on continue à explorer alors que la meilleure annonce a été déterminé

ce qui fait que les résultats sont moins bons. Pour une valeur de 0.001 l'algorithme met du temps à trouver la meilleure solution car on explore rarement.

### 1.3.3 Greedy optimiste

Une manière de faire en sorte que l'algorithme greedy explore plus est de l'initialiser avec des valeurs élevées. Ici il n'y a pas de phase d'exploration, mais on initialise les estimations des gains par 5, c'est une valeur supérieure à tous gains qu'on pourrait avoir, cela force l'algorithme à continuer l'exploration. Quand un levier est sélectionné, la gains moyen de ce levier diminue, ce qui force l'algorithme à sélectionner un autre levier au tour suivant. L'algorithme se stabilise une fois avoir vu assez d'exemple.

Dans la figure ci-dessous nous avons comparé  $\epsilon - greedy$  à l'algorithme greedy optimiste.

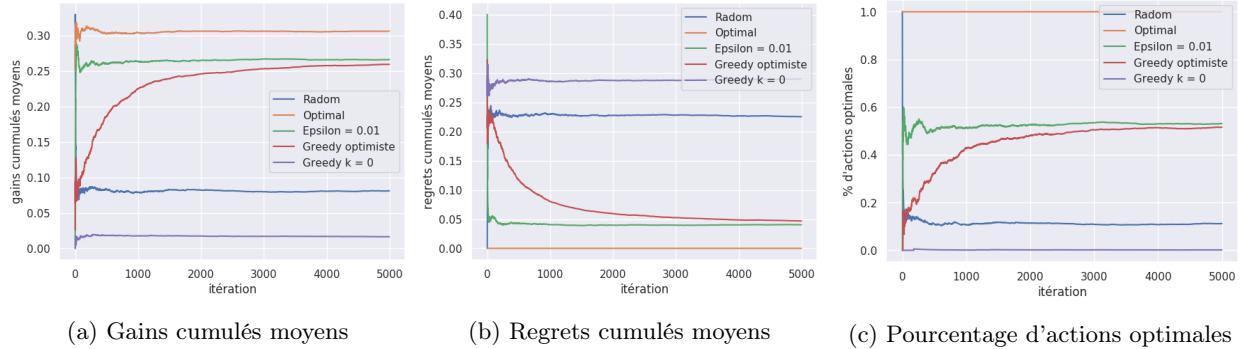


FIGURE 4: Comparaison entre Greedy-optimiste et  $\epsilon - greedy$

**Analyse :** On remarque que la convergence est plus lente que pour  $\epsilon - greedy$  mais ils atteignent le même résultats après 5000 itérations.

### 1.3.4 Upper Confidence Bound UCB

Pour la stratégie UCB, on choisit le levier  $j$  qui maximise la quantité suivante :

$$\hat{x}_j + c \sqrt{\frac{\ln n}{n_j}}$$

où  $\hat{x}_j$  est le gain moyen du levier  $j$ ,  $n$  le nombre d'essais,  $n_j$  le nombre de fois que le levier  $j$  a été joué et  $c$  est un facteur d'exploration, il est souvent mis à  $\sqrt{2}$ .

Dans la figure suivante, on a exécuté la méthode UCB avec différentes valeurs de  $c$ .

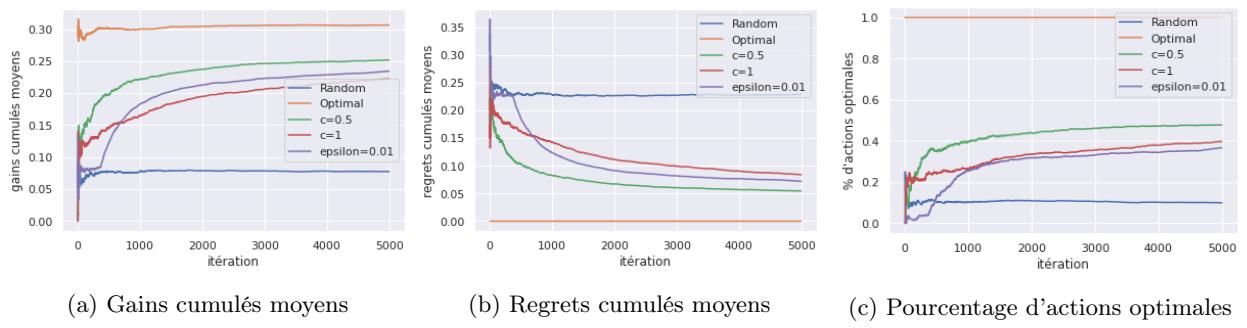


FIGURE 5: Algorithme UCB pour différentes valeurs de  $c$

**Analyse :** L'algorithme atteint de meilleur résultats que  $\epsilon$ -greedy et une valeur de  $c$  moins élevée accélère la convergence.

### 1.3.5 Stratégie linUCB

Dans la méthode LinUCB avec modèle linéaire disjoints, on suppose que l'espérance du gain d'une annonce est linéaire en son contexte  $x_t$  avec comme coefficients le vecteur  $\theta^*$ , le modèle est appelé disjoints car on suppose que les paramètres de chaque gains sont différents. De cette hypothèse on dérive la stratégie suivante :

$$a_t \in \operatorname{argmax}_{a \in \mathcal{A}_t} (x_{t,a}^T \hat{\theta}_a + \alpha \sqrt{x_{t,a}^T A_a^{-1} x_{t,a}})$$

avec  $\mathcal{A}_t$  l'ensemble des leviers à l'itération  $t$ ,  $x_{t,a}$  le vecteur contexte à l'itération  $t$  avec l'action  $a$  (dans notre cas toutes les annonces ont le même contexte pour un même utilisateur),  $A_a = D_a^T D_a + I_d$  et  $D_a$  la matrice des contextes observés pour le bras  $a$  et  $\alpha = 1 + \sqrt{\ln(2/\delta)/2}$  pour tout  $\delta > 0$

On a implémenté l'algorithme puis testé en faisant varier  $\alpha$  face à une méthode classique  $\epsilon$ -greedy.

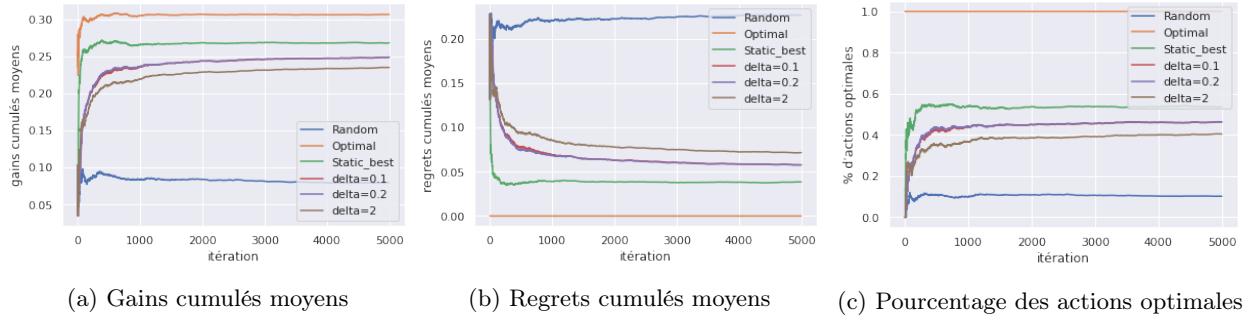


FIGURE 6: LinUCB pour différentes valeurs de  $\alpha$

**Analyse :** On remarque que LinUCB converge rapidement pour des valeurs de  $\delta$  moins élevées, mais il n'y a pas de différences entre  $\delta = 0.1$  et  $\delta = 0.2$ .

## 1.4 Comparaison de toutes les stratégies

Dans ce dernier test, nous comparons toutes les stratégies avec les meilleurs paramètres trouvés dans les tests précédents. Dans la méthode LinUCB,  $\delta = 0.2$ , pour UCB,  $c = 0.5$ , pour  $\epsilon$ -greedy,  $\epsilon = 0.3$ .

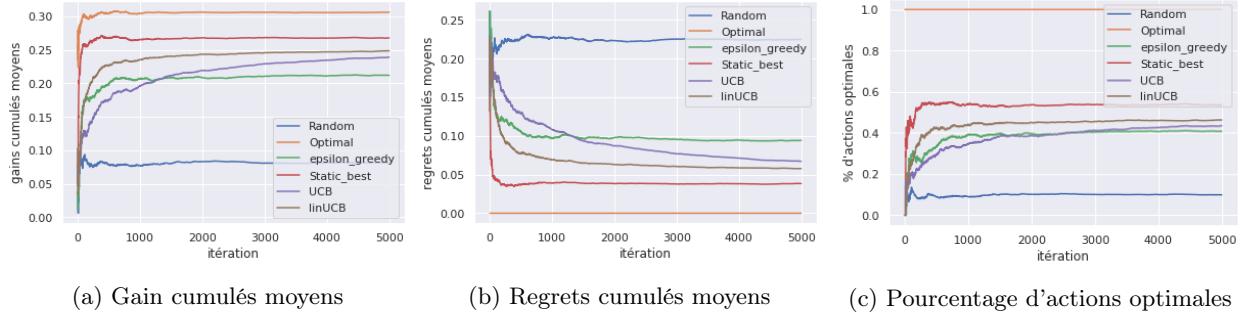


FIGURE 7: Comparaison de toutes les méthodes

LinUcb arrive à de meilleurs résultats suivit de UCB mais qui converge lentement et  $\epsilon$ -greedy qui converge rapidement mais vers un gains inférieur.

## 1.5 Conclusion

A travers ce tme, on a exploré les algorithmes de résolution de problèmes de type bandit manchots, on a procédé au réglage des paramètres et effectuer une campagne d'expériences.

## 2 TME 2. Programmation Dynamique

Ce TME a pour objectif d'expérimenter les algorithmes de programmation dynamique sur un MDP classique de type GridWorld. Dans les cas où l'on connaît le MDP, deux algorithmes existent :

- **Policy Iteration** : Évaluation de la politique jusqu'à convergence puis mise à jour de la politique.
- **Value Iteration** : Recherche itérative de la politique optimale.

Nous avons implémenté les algorithmes d'itération de la valeur et d'itération de la politique. Nous avons analysé les deux algorithmes du point de vue de la performance, de la convergence en temps et en nombre d'itération pour trouver la politique, et nous avons comparé les politiques obtenues à l'aide des deux méthodes. Nous avons ensuite fait plusieurs expériences sur différentes cartes pour voir comment la répartition des récompenses affectait les politiques. Nous avons aussi analysé l'impact du paramètre de discount  $\gamma$  sur la politique suivie par l'agent. Nous avons implémenté une méthode pour visualiser la valeur et la politique directement sur la carte.

### 2.1 Le MDP

Nous utilisons un MDP de type GridWorld. L'agent en question peut effectuer 4 actions dans un monde discret. Il peut aller vers le haut, le bas, à gauche et à droite, sachant que son action ne réalisera qu'avec une probabilité de 0.8, 20% du temps, il sera dévié vers les directions orthogonales à la direction choisie, par exemple si l'agent décide d'aller en haut, il se peut qu'il aille à gauche ou à droite avec une probabilité de 0.2.

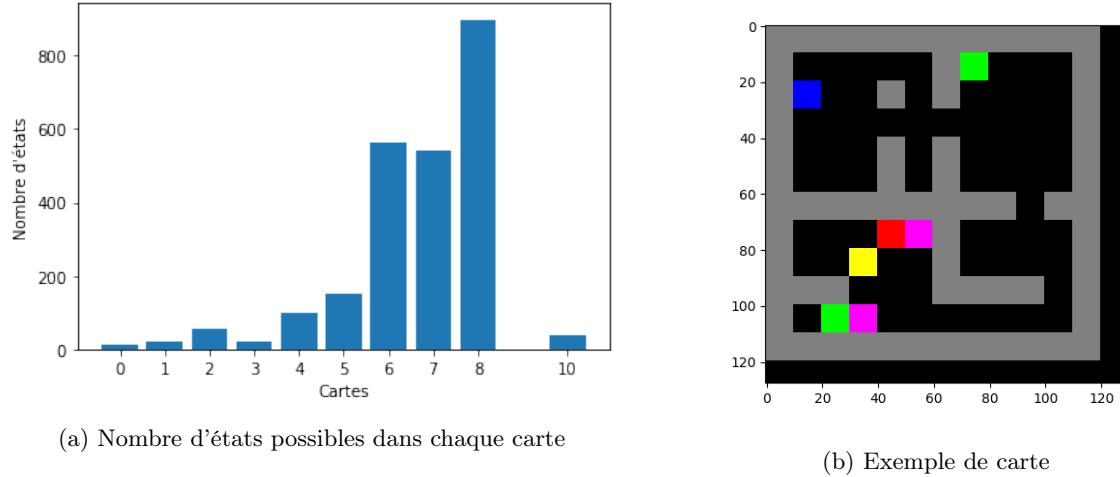


FIGURE 8: Caractéristiques des cartes

Les cases terminales sont les cases vertes "partie gagnée" et les cases rouges "partie perdue". Il existe deux cases ramassable par l'agent, les jaunes "pièce" et les roses.

### 2.2 Vitesse de convergence

Dans cette partie, nous comparons la temps et le nombre d'itérations nécessaire aux deux algorithmes pour déterminer la politique optimale. Nous effectuons les tests sur chaque carte donnée (sauf la 9, le nombre de cas possibles est trop important donc le MDP est trop lourd à charger). Nous fixons  $\gamma = 0.99$  et  $\epsilon = 10^{-3}$ .

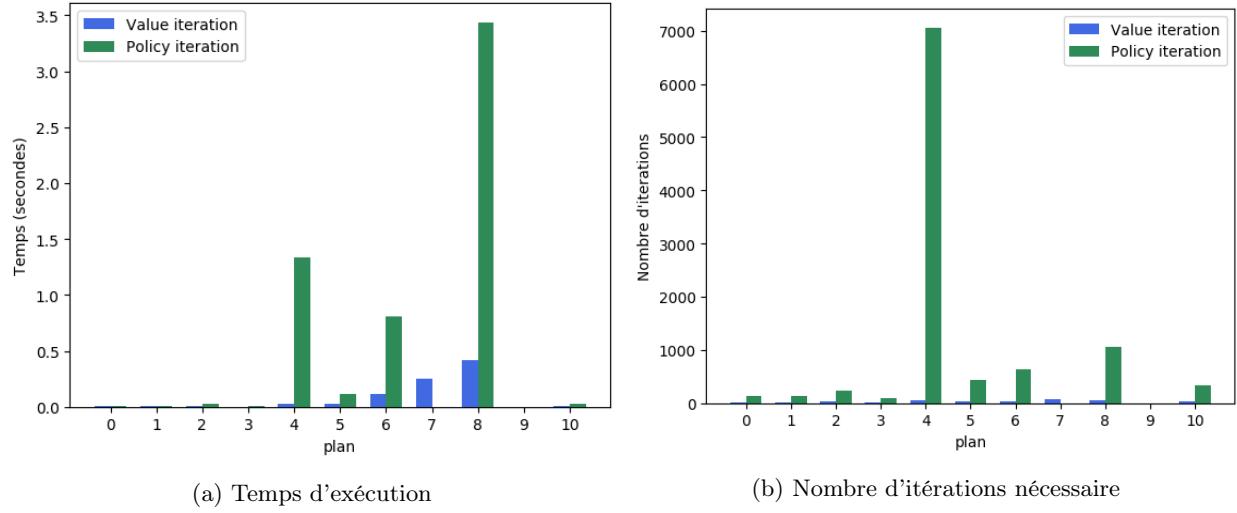


FIGURE 9: Analyse de la convergence des deux algorithmes

L'algorithme d'itération de la politique est plus lent et nécessite bien plus d'itérations que l'algorithme d'itération de la valeur, ce n'est pas surprenant vu que l'itération de la valeur requiert moins d'opérations.

### 2.3 Comparaison des politiques obtenues

Dans cette partie, nous voulons tester si les politiques résultantes des deux méthodes sont les mêmes, nous lançons donc des tests sur la carte 6 sur 10000 épisodes en gardant les paramètres précédents. Nous analysons le gain total obtenu par l'agent à chaque épisode.

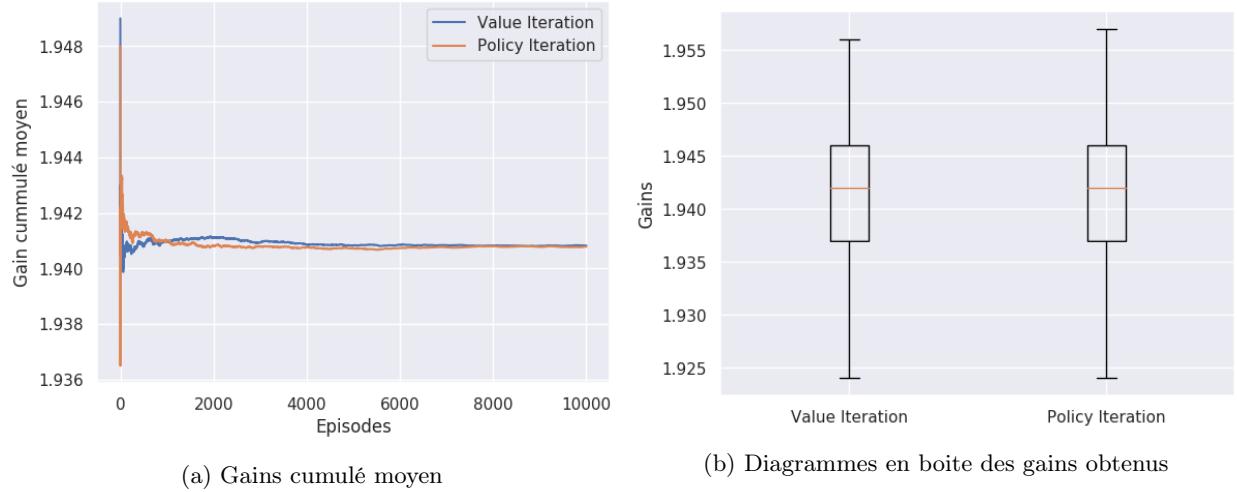


FIGURE 10: Comparaison des gains

Comme on peut le voir, il n'y a pas de distinctions en terme de gains obtenus par les deux politiques. On peut aussi tracer le nombre d'actions effectuées à chaque épisode.

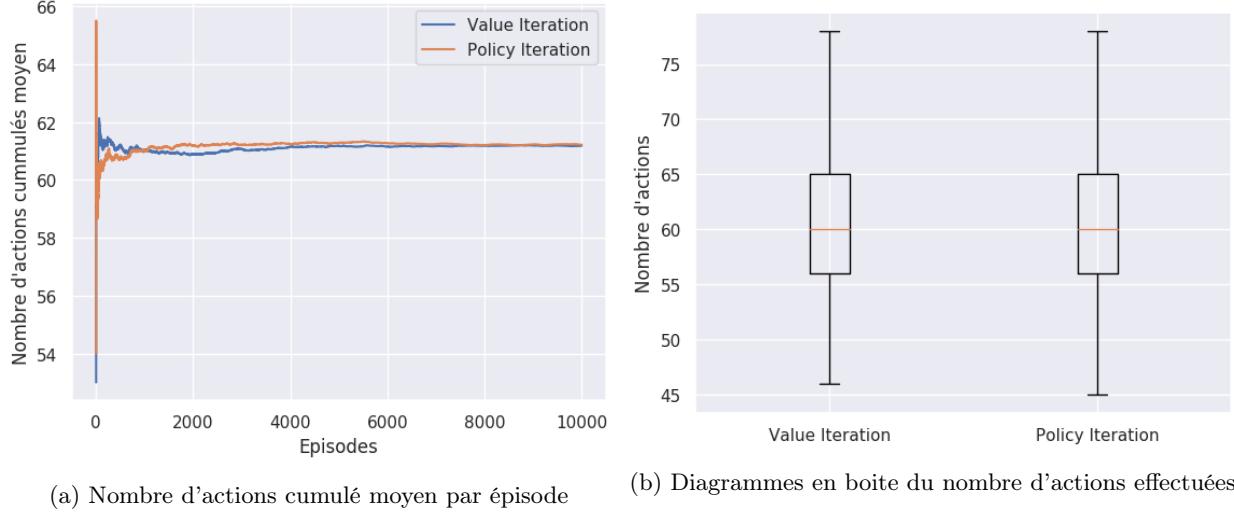


FIGURE 11: Comparaison du nombre d'actions effectuées

Comme précédemment, que l'agent suive la politique obtenue par itération de la valeur ou par itération de la politique, il effectue le même nombre d'actions.

## 2.4 Effet du paramètre discount

Pour analyser l'effet du paramètre de discount, nous avons exécuté l'algorithme d'itération de la valeur en faisant varier  $\gamma$  de 0.8 à 1 puis en traçant les gains moyens et le nombre d'actions moyen effectuées par l'agent sur 1000 épisodes pour chaque valeur de  $\gamma$ . On garde les mêmes valeurs de  $\epsilon$  et la même répartition des gains.

Sur la carte 0, on observe les résultats suivant :

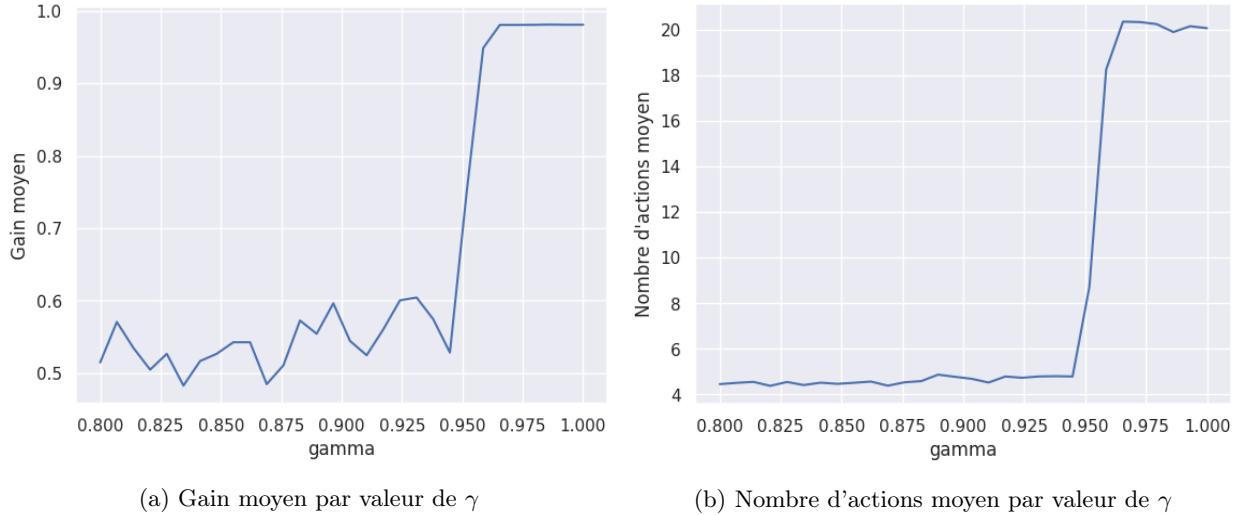


FIGURE 12: Effet de gamma sur la carte 0

Les gains moyens sont maximisés pour des valeurs de  $\gamma$  supérieures à 0.97. Quant au nombre d'actions effectuées, il augmente pour les valeurs de  $\gamma > 0.97$ , cela est du au fait que sur cette carte, le meilleur chemin est le plus long.

En faisant les mêmes tests mais sur la carte 8 où l'état but est éloigné de la position initiale de l'agent, nous avons obtenu les résultats ci-dessous :

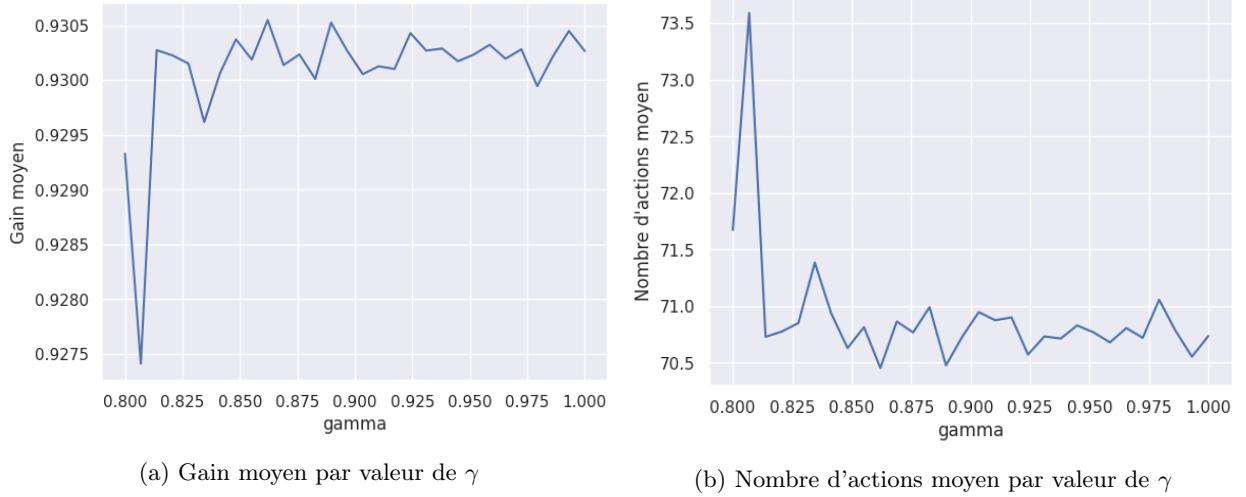


FIGURE 13: Effet de  $\gamma$  sur la carte 0

Pour cette carte, Les gains moyens sont maximisés dès  $\gamma > 0.81$ , mais il fluctuent plus que pour l'exemple précédent, c'est du au fait qu'il y a plus d'états dans cette carte. Quant au nombre d'actions, il diminue pour  $\gamma > 0.81$ , le meilleur chemin étant l'état le plus court.

## 2.5 Étude de cas : Impact des récompenses sur la politique

Dans cette partie, nous nous intéressons à l'impact de la répartition des récompenses sur la politique choisie. Nous avons tracé pour chaque cas de figure la politique sur une carte, ainsi que les gains et les nombre d'actions par épisode.

### 2.5.1 Carte 4 et 7 : Agent qui boucle

Sur la **carte 4**, il y a une case rose avec une récompense de -1 qui se trouve sur le seul chemin menant à la case verte finale. Avec la configuration par défauts, nous avons les résultats de la figure suivante.

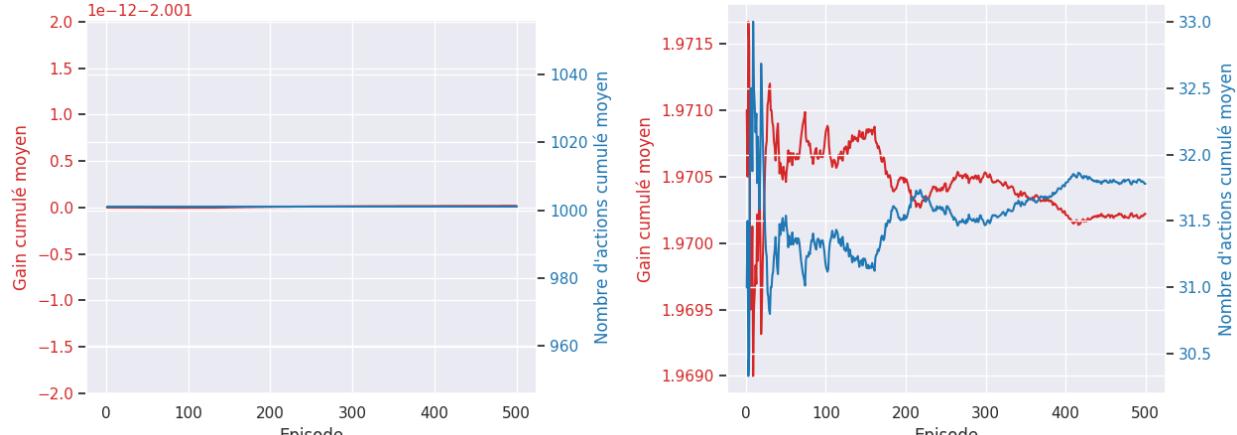


FIGURE 14: Comportement de l'agent avec différentes configuration sur la carte 4

Avec la configuration originale, l'agent tourne en rond et ne franchie jamais la case rose, la partie est arrêtée au bout de 1001 épisodes et le gain est de  $0 = -0.001 * 1000$ . En changeant la récompense attribuée à la case rose en  $R(\text{rose}) = 1$ , on incite l'agent à franchir cette case et les parties se terminent en 1001 épisodes.

Sur la **carte 7**, il y a un passage, avec 3 cases rouges, 3 pièces et une case rose qu'on est obligé d'emprunter pour aller vers la case verte. La carte est représentée dans la figure ci-dessous.

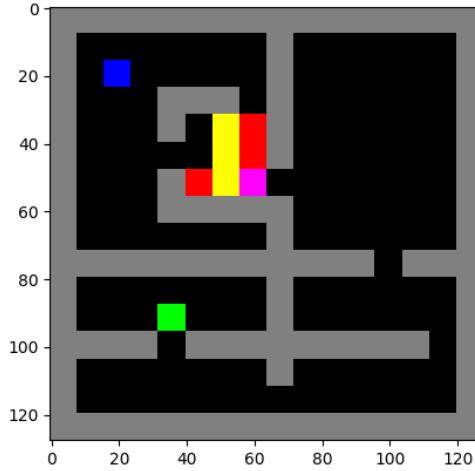


FIGURE 15: carte 7

Comme on peut le voir sur le graphique suivant, le nombre d'actions par partie avoisine les 700 actions, cela montre que l'agent bloque sur l'endroit où se trouve les pièces et ne franchit pas la case rose pour terminer la partie.

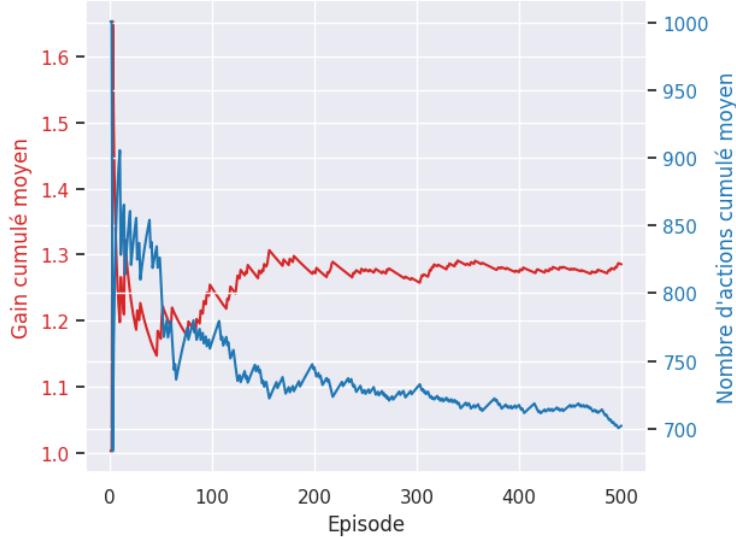


FIGURE 16: Résultats sur la carte 7 avec configuration par défauts

Une manière de régler ce problème est de changer les récompenses par case en : case vide : -0.001, case verte : 1, pièce : 0.5, case rouge : -1, case rose : -0.5. On diminue les récompenses associées aux pièces afin d'inciter l'agent à se diriger vers la case verte et on augmente la récompense de la case rose à -0.5 afin de l'inciter à dépasser cette case pour se diriger vers la case verte.

On peut voir que de cette façon, l'agent ne tourne plus en rond et le jeu se termine plus tôt avec 50 actions par épisode et un gain cumulé moyen de 4.

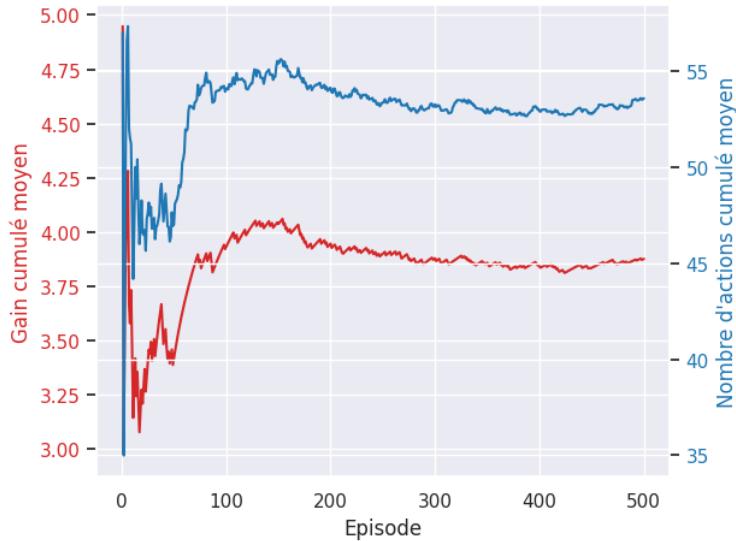
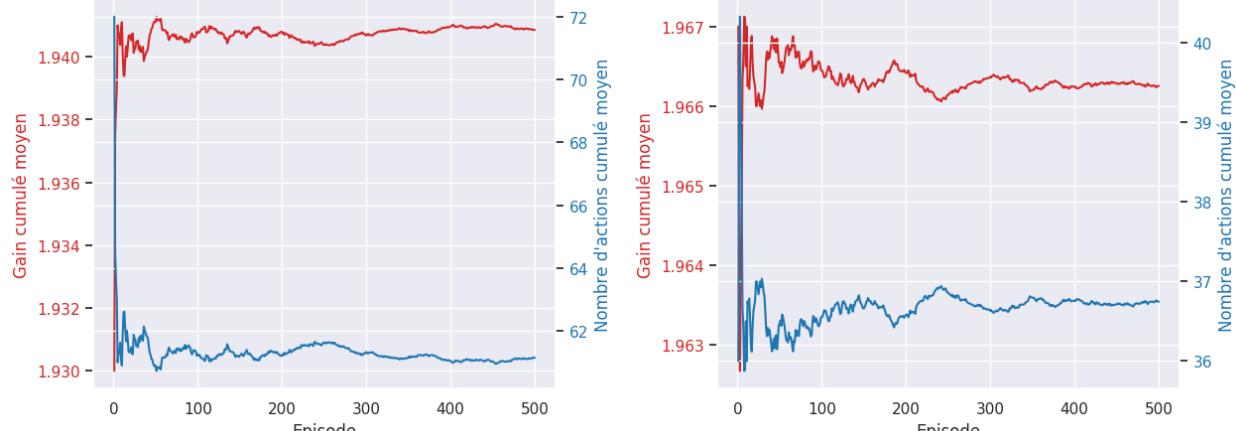


FIGURE 17: Résultats sur la carte 7 avec la nouvelle configuration

### 2.5.2 Carte 6 : pièce éloignée de l'objectif

Sur la carte 6, en gardant la configuration par défaut, l'agent récupère la pièce puis remonte vers la case verte supérieure. En modifiant la récompense liée à la case rose,  $R(rose) = 0$ , l'agent passe par la case verte immédiatement adjacente à la pièce. On réduit le nombre d'actions par épisode de 60 à 30.



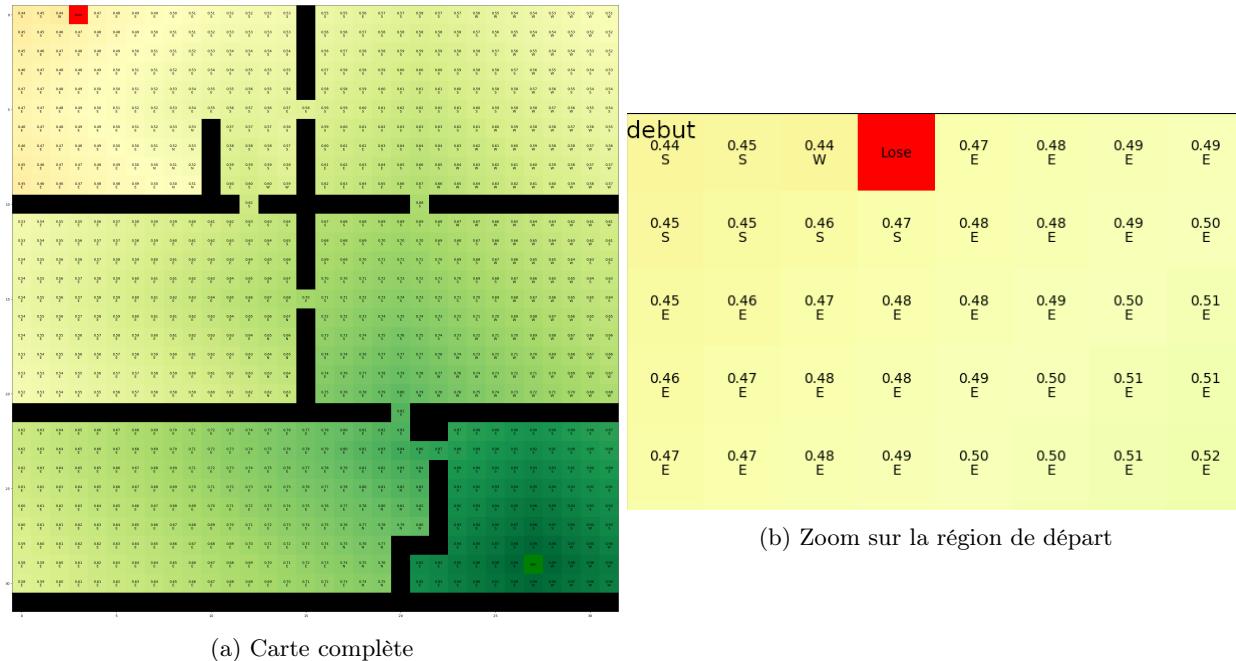
(a) Résultat avec la configuration par défaut

(b) Résultat avec la nouvelle configuration

FIGURE 18: Comportement de l'agent avec différentes configuration sur la carte 6

### 2.5.3 Carte 11 : objectif éloigné

Nous introduisons une nouvelle carte en reprenant la carte 8 mais en rajoutant une case rouge proche de l'état initial de l'agent. Nous nous intéressons à l'impact de la récompense liée aux cases vides. Avec la configuration par défaut, l'agent suit la politique représentée ci-dessous.

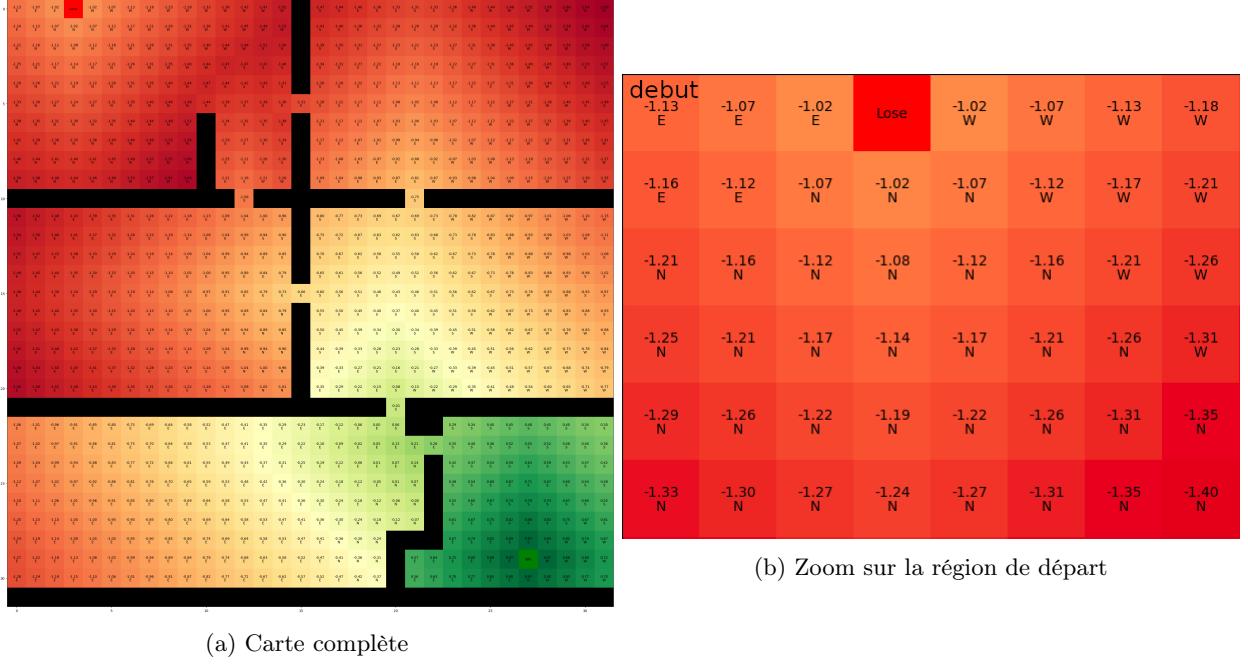


(a) Carte complète

(b) Zoom sur la région de départ

FIGURE 19: Politique de l'agent sur la carte 11 avec la configuration d'origine

L'agent évite la case rouge et se dirige vers la case verte. Maintenant si on diminue la récompense de la case vide à  $R(vide) = -0.05$ , les déplacements sont plus coûteux, et on obtient la politique suivante :



(a) Carte complète

(b) Zoom sur la région de départ

FIGURE 20: Politique de l'agent sur la carte 11 avec la nouvelle configuration

L'agent préfère maintenant abréger la partie en se dirigeant directement vers la case rouge, car la case verte étant éloignée, les déplacement successifs ne valent pas la récompense de 1. L'agent serait encore plus sensible à la récompense liée aux déplacements si la carte était plus grande et la case verte encore plus éloignée.

## 2.6 Conclusion

A travers ce tme, nous avons exploré les algorithmes d'itération de la valeur et de la politique, nous avons comparé les deux méthodes en vitesse de convergence et en terme de politique obtenue. Puis, nous avons examiné l'effet du paramètre de discount. Ensuite, nous avons discuté de l'importance de la configuration des récompenses sur le comportement de l'agent en testant sur plusieurs cas de figure.

## 3 TME 3. Q-Learning

### 3.1 Introduction :

Dans ce TME contrairement au précédent, avec les algorithmes de planification de trajectoire où on supposait connaître le *MDP (offline)*, on se place dans le cas où l'on ne connaît ni  $\mathcal{P}$  ni  $\mathcal{R}$ . Ce faisant, l'agent doit interagir avec l'environnement pour apprendre une politique d'action (*online*). Une première approche *Model Based* serait d'estimer le *MDP* ( $\mathcal{P}$  et  $\mathcal{R}$ ) en explorant l'environnement pour ensuite utiliser les algorithmes de planification (value iteration et policy iteration) dans le modèle appris. Cette méthode est prometteuse si l'on est capable d'apprendre parfaitement le MDP. Cependant, pour la plupart des problèmes, estimer le MDP est plus difficile que de trouver une politique optimale. D'autres méthodes, appelées *Model Free*, cherchent à trouver une politique optimale sans estimer le *MDP*. Il existe aussi des méthodes "hybrides" à la jonction entre les méthodes *Model Based* et *Model Free*. Dans ce qui va suivre, nous allons explorer plusieurs algorithmes :

- L'algorithme **Q-learning** : un algorithme Model Free, on implémente la version  $TD(0)$ . De plus, il s'agit d'un algorithme *off-policy*, c'est-à-dire que la mise à jour de  $Q$  dans l'itération de l'algorithme ne dépend pas de la politique  $\pi$ . La politique est uniquement utilisée pour améliorer la qualité des samples. Ce qui permet, par exemple, de se resserrer des observations précédentes avec une nouvelle politique (Garder en mémoire les quadruplets observés : *replay Buffer*).
- L'algorithme **SARSA** : un algorithme Model Free. Cependant, ce dernier, contrairement à Q-learning est *on-policy*, c'est-à-dire que la mise à jour de  $Q$  dans l'itération de cet algorithme dépend de la politique actuelle (qui sert à interagir avec l'environnement).
- L'algorithme **Dyna-Q** : un algorithme hybride (*Model Base / Model Free*), tout en interagissant avec l'environnement avec une stratégie d'exploration, on estime le MDP.

Ainsi que différentes stratégies d'explorations :

- Greedy.
- $\epsilon$ -Greedy.
- Sélection de Boltzman.
- UCB-1

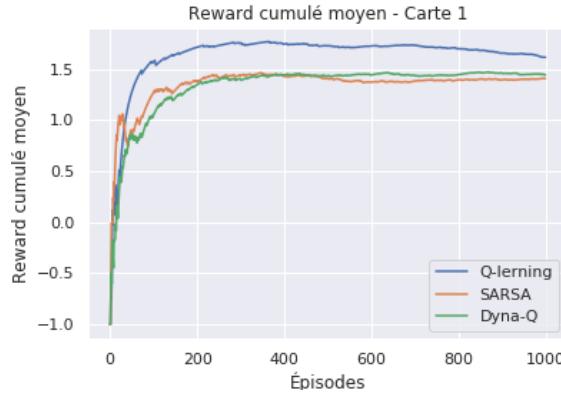
#### 3.1.1 Q-Learning vs SARSA vs Dyna-Q : performances

Dans cette section, on s'intéresse aux performances des différents algorithmes sus-mentionnés, sur quatre cartes : la carte 1, la carte 3, la carte 6 et la carte 7. Pour les trois algorithmes on utilise une stratégie  $\epsilon$ -Greedy.

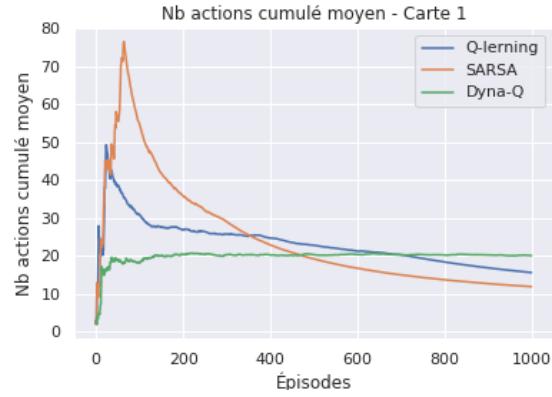
#### Hyper-paramètres :

- Paramètre de mise à jour de  $Q$  :  $\alpha = 0.1$ .
- Facteur de discount  $\gamma = 0.95$ .
- Paramètre de mise à jour de  $P$  et de  $R$  (pour les méthodes *Model Based* où hybrides : Dyna-Q) :  $\alpha_R = \alpha_P = 0.01$ .
- Nombre de couples états-actions échantillonnes dans Dyna-Q,  $k = 20$
- Paramètre d'exploration d' $\epsilon$ -Greedy,  $\epsilon = 0.01$ .
- Nombre d'épisodes : 1000.

Pour chaque carte, on trace le reward cumulé moyen ainsi que le nombre d'actions cumulé moyen de chaque algorithme :



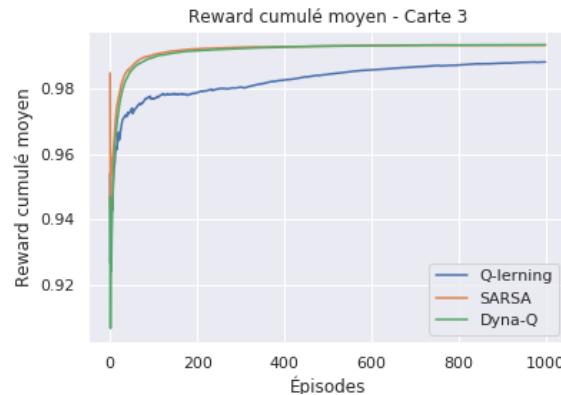
(a) Reward cumulé moyen par épisode - carte 1.



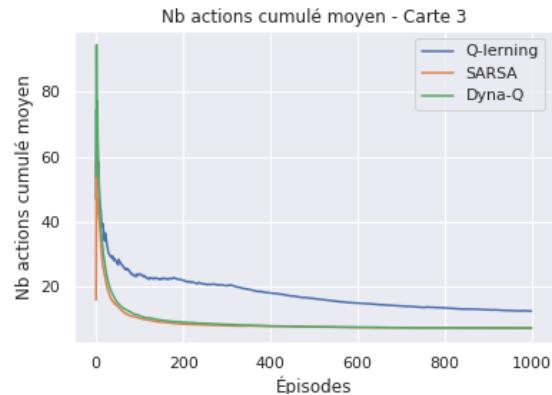
(b) Nombre d'actions par épisode cumulé moyen - carte 1.

FIGURE 21: Comparaison des performances de SARSA, Q-Learning et Dyna-Q - carte 1.

**Analyse - carte 1.** Sur la première carte assez simple, SARSA et Dyna-Q convergent au même moment et vers des rewards similaires. Q-learning achève les meilleures performances et est le plus rapide à converger. Concernant le nombre d'actions moyen, les trois algorithmes convergent vers des épisodes de même taille à peu près (Dyna-Q > Q-learning > SARSA). On remarque un pic dans la taille des épisodes au début de l'apprentissage pour Q-learning et SARSA ce qui est dû à la phase d'exploration, ou l'agent n'a pas assez d'informations pour faire des trajectoires optimales. Ce pic, n'est pas observé avec Dyna-Q, on attribue cette différence à l'estimation du MDP dans ce dernier, qui permet d'exploiter au mieux le peu de transitions explorées.



(a) Reward cumulé moyen par épisode - carte 3.



(b) Nombre d'actions par épisode cumulé moyen - carte 3.

FIGURE 22: Comparaison des performances de SARSA, Q-Learning et Dyna-Q - carte 3.

**Analyse - carte 3.** Sur la carte 3, SARSA et Dyna-Q convergent vers des performances et des trajectoires de tailles identiques. Q-learning achève des performances moindres et avec des trajectoires plus grandes. On attribue cette différence à l'exploration qui se fait dans Q-learning, qui est atténuée par l'estimation du MDP pour Dyna-Q et la mise à jour on-policy de SARSA.

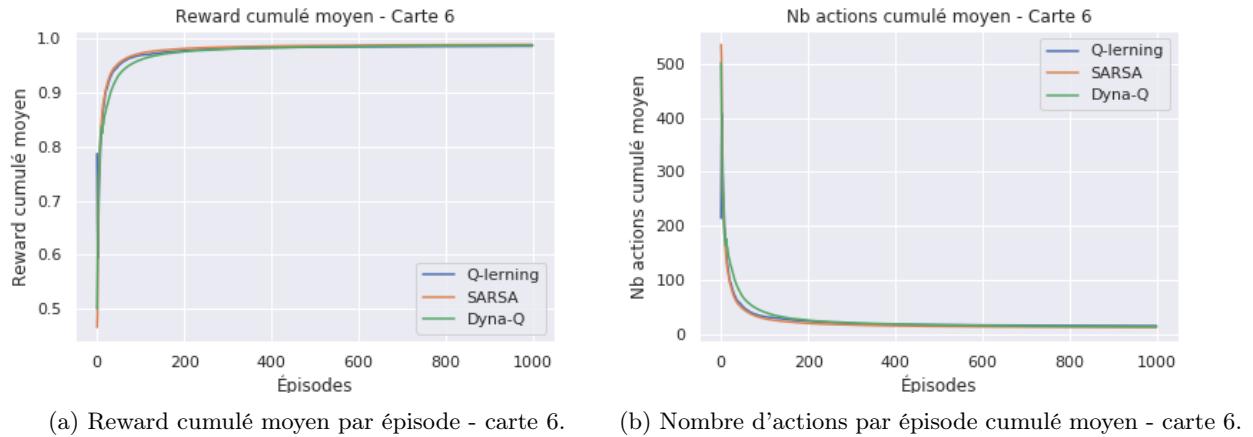


FIGURE 23: Comparaison des performances de SARSA, Q-Learning et Dyna-Q - carte 6.

**Analyse - carte 6.** Sur la carte 6, les comportement des trois algorithmes est identique.

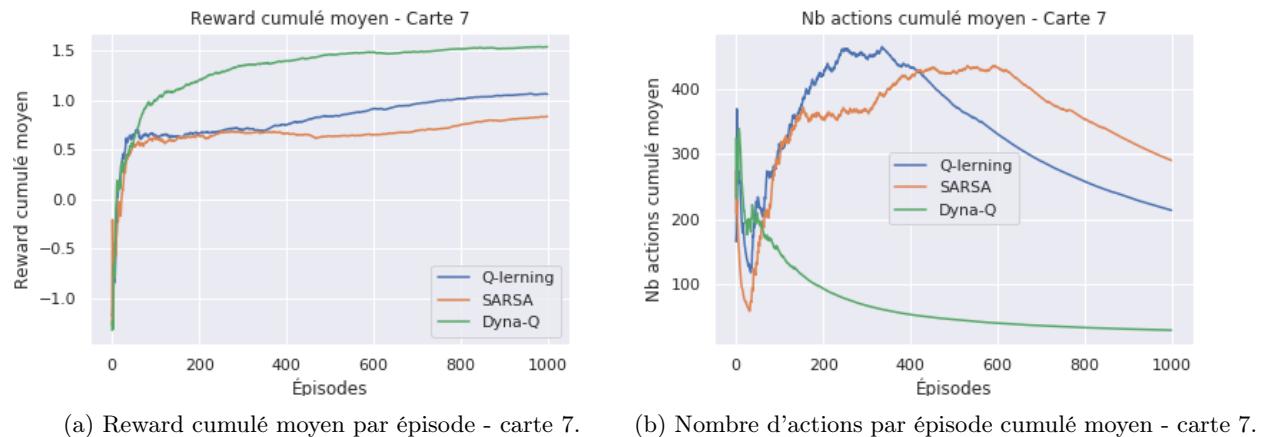


FIGURE 24: Comparaison des performances de SARSA, Q-Learning et Dyna-Q - carte 7.

**Analyse - carte 7.** Sur la carte 7 assez difficile, il y a un passage, avec 3 cases rouges, 3 pièces et une case rose qu'on est obligé d'emprunter pour aller vers la case verte. La carte est représentée dans la figure ci-dessous.

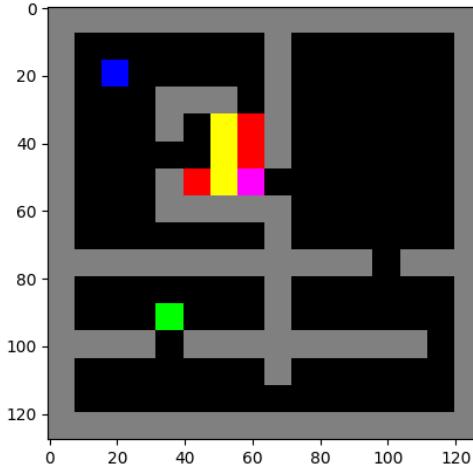


FIGURE 25: carte 7

L'algorithme le plus performant est Dyna-Q, et ce, en vitesse de convergence, en reward et en taille des trajectoires. Q-learning comme SARSA, achève des performances beaucoup moindres ( $Q\text{-learning} > \text{SARSA}$ ). On attribue cette supériorité de Dyna-Q à la complexité de la carte qui nécessite une certaine estimation du MDP pour pouvoir y naviguer optimalement.

### 3.2 Q-learning vs SARSA vs Dyna-Q : temps de calcul

En utilisant les mêmes hyper-paramètres que dans la section précédente, on enregistre le temps de calcul de chaque algorithme sur chacune des cartes (temps pour accomplir 1000 épisodes). Les résultats sont illustrés sur le tableau suivant :

TABLE 1: Temps de calcul de SARSA, Q-Learning et Dyna-Q pour différentes cartes

<i>Algorithme</i>	Carte 1	Carte 3	Carte 6	Carte 7
<i>Q – learning</i>	0.337ms	0.325ms	0.450ms	6.320ms
<i>SARSA</i>	0.257ms	0.202ms	0.353ms	8.108ms
<i>Dyna – Q</i>	16.009ms	7.433ms	28.578ms	342.502ms

Les temps de calcul des différents algorithmes sont dépendants de deux facteurs : la taille des trajectoires de chaque épisode qui dépend des performances de l'algorithme et la complexité de l'algorithme lui-même. Le temps de calculs sur les différentes cartes est comparable entre Q-learning et SARSA. Dyna-Q quant à lui prend énormément plus de temps (non-négligeable  $\approx 30$ ), ceci est dû à la complexité de l'algorithme lui-même (opérations de mise à jour du MDP, opérations de mise à jour de  $Q$  basées sur le MDP). Sur la carte 7 par exemple, il faut faire un compromis, performances et rapidité.

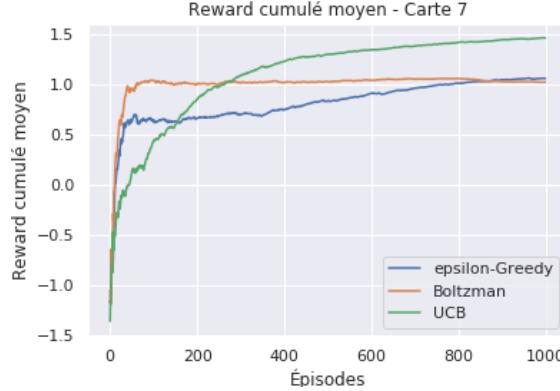
#### 3.2.1 Comparaison des politiques d'exploration sur Q-learning :

Dans cette section on s'intéresse à l'effet de la politique d'exploration sur les performances de Q-learning. En particulier on testera sur la carte 7, trois stratégies différentes :  $\epsilon$ -greedy, UCB-1 et la sélection de Blotzman.

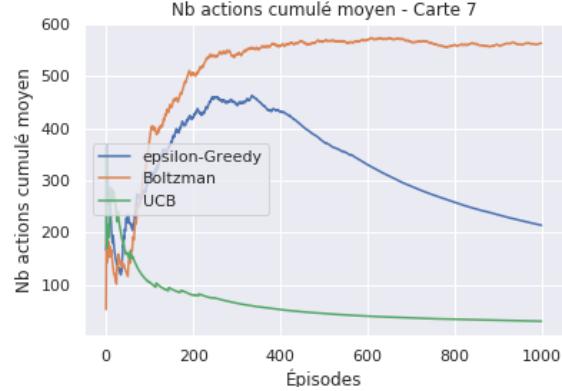
### Hyper-paramètres.

- Facteur d'exploration d' $\epsilon$ -greedy :  $\epsilon = 0.01$ .
- Température dans la sélection de Boltzman :  $T = 1e - 3$ .

Les résultats des expériences sont illustrés sur les figures suivantes :



(a) Reward cumulé moyen par épisode - carte 7.



(b) Nombre d'actions par épisode cumulé moyen - carte 7.

FIGURE 26: Comparaison des performances de Q-learning pour plusieurs stratégies d'exploration - carte 7.

**Analyse.** Sur la carte 7, Boltzman et epsilon-greedy convergent vers des performances égales (Boltzman converge beaucoup plus rapidement qu'epsilon-greedy). Quant aux tailles de trajectoires Boltzman, converge vers des trajectoires de tailles anormalement grandes, une raison de ce comportement peut être la température qui est trop petite ( $T$  grand  $\rightarrow$  distribution uniforme  $\rightarrow$  aléatoire et  $T$  petit  $\rightarrow$  distribution piquée  $\rightarrow$  greedy) et donc une stratégie trop greedy. UCB est la stratégie la plus performante en reward comme en taille de trajectoires.

On analyse aussi le temps que prend l'agent pour effectuer 1000 épisodes. Les résultats sont illustrés dans le tableau suivant :

TABLE 2: Temps de calcul de Q-Learning sur la carte 7 pour différentes stratégies d'exploration

Algorithme – Carte	$\epsilon$ -greedy	Boltzman	UCB-1
<i>Q – learning – Carte 7</i>	6.320ms	18.249ms	1.085ms

**Analyse.** Puisque les trois stratégies d'exploration ont des complexités de calculs très comparables, le temps d'exécution n'est proportionnel qu'à la longueur des trajectoires de l'agent au cours des épisodes d'apprentissage. Les temps d'exécution dans le tableau reflètent exactement la nombre d'actions cumulé moyen sur la courbe précédente (Boltzman >  $\epsilon$ -greedy > UCB-1).

## 4 TME 4. Deep Q network

### 4.1 Introduction

Le but de cette section est d'explorer l'algorithme deep Q network et ses variantes à travers 3 simulations : *CartPool*, *GridWorld* et *LunarLander*.

### 4.2 DQN

Nous commençons par l'algorithme Deep Q Network de base, donné en figure 27.

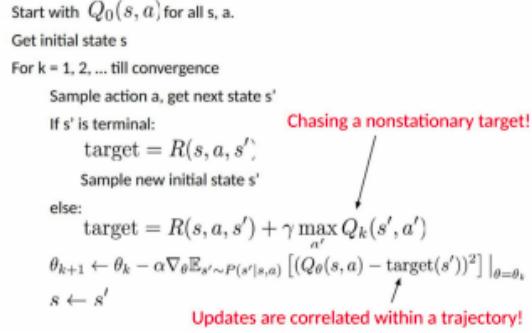


FIGURE 27: Algorithme DQN

Nous testons cet algorithme sur les 3 environnements séparément,

**CartPool.** Cartpol est un jeu où l'on cherche à stabiliser une barre verticale sur un chariot roulant, en fonction de ses mouvements. Deux actions possibles à chaque itération : gauche ou droite. Le jeu est perdu si la barre verticale s'incline de plus de 15 % (elle tombe alors inévitablement) ou si le chariot sort de l'écran. Le reward cumulé correspond au nombre de pas de temps sans terminaison du jeu (maximum 500 pas de temps). Les observations sont des vecteurs de 4 réels donnant la position du chariot, sa vitesse, l'inclinaison de la barre et sa vitesse de rotation. C'est a priori l'environnement le plus simple.

**DQN appliqué sur Cartpool.** Nous testons plusieurs paramétrages de DQN sur cartpool. Les résultats sont présentés sur la figure 28.

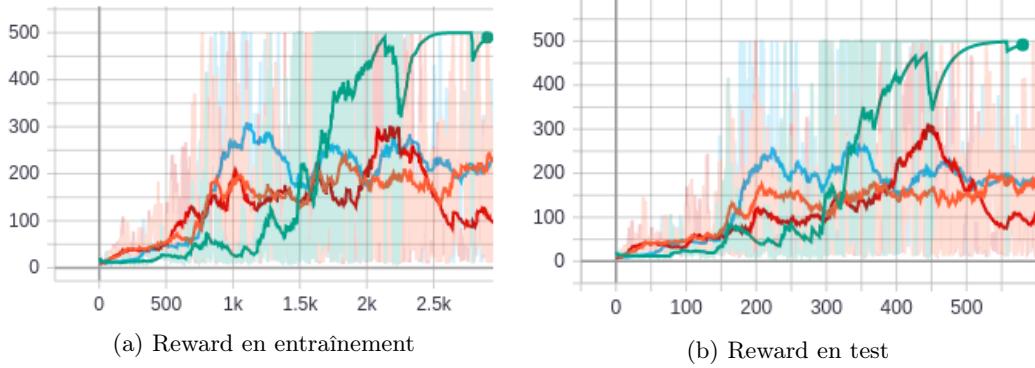


FIGURE 28: Reward par itération pour différents paramétrages. **Vert** :  $\epsilon$ -greedy avec *weight-decay* entraîné avec *ADAM* sur un réseau de 200 neurones. **Orange** :  $\epsilon$ -greedy de 0.3 ,SGD,  $Q$  de 100 neurones. **Rouge** : réseau de deux couches de 100 neurones avec *SGD*. **Bleu** : une seule couche de 30 neurones avec *SGD*.

**Analyse.** Le paramètre qui a le plus d’impact est l’algorithme d’optimisation. En utilisant *ADAM*, DQN arrive à converger vers la valeur maximale 500 en ~2000 itérations. On utilisera par la suite ces paramètres pour *CartPool*.

**GridWorld.** Cet environnement a été introduit dans la section 2. Pour tester DQN, on se fixe à la carte 0. Dans la figure 29, on teste différents paramétrages sur la version de base de DQN.

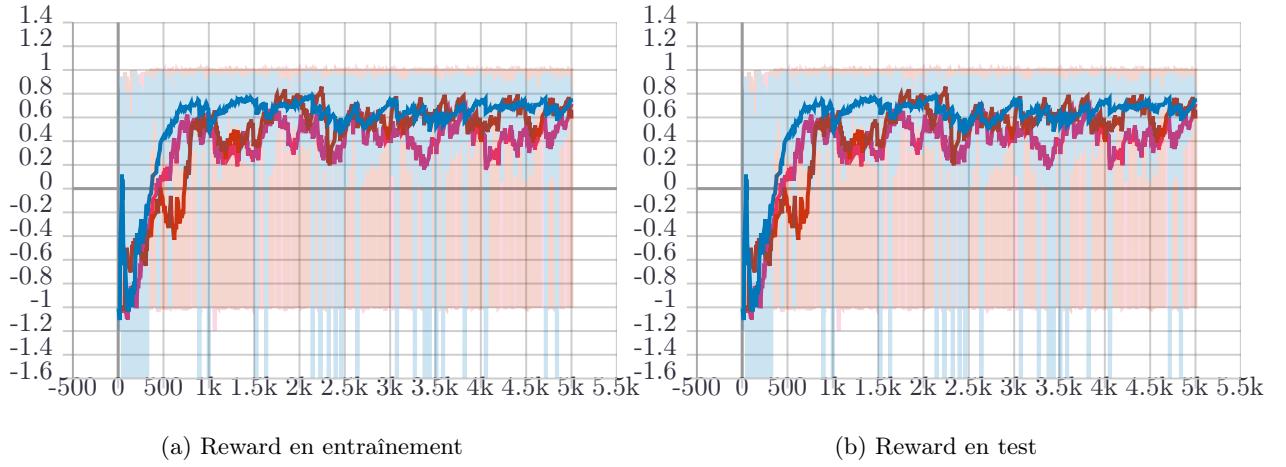


FIGURE 29: Reward par itération pour différents paramétrages. **Bleu** : DQN avec stratégie greedy, entraîné avec *SGD* sur un réseau de 100 neurones. **Rose** :  $\epsilon$ -greedy de 0.3 ,SGD,  $Q$  de 100 neurones. **Rouge** : stratégie UCB, entraîné avec SGD,  $Q$  de 100 neurones.

**analyse.** Pour la carte 0, avec le paramétrage considéré, DQN, ne semble pas converger vers le maximum 1 mais converge vers une valeur moins élevée et reste très instable. La version greedy est semble légèrement plus stable.

**LunarLander.** LunarLander est un jeu où l’objectif est de faire alunir une fusée, qui a une vitesse et une direction initiales aléatoires. Le reward obtenu dépend de la vitesse à laquelle la fusée s’est posée et de la distance de la cible (ainsi que du temps mis à alunir, du fait que les deux pieds touchent le sol, du fuel dépensé, etc.). 4 actions possibles par pas de temps : ne rien faire, allumer le moteur de gauche, le moteur

central ou le moteur de droite. Les observations sont des vecteurs de 8 réels donnant diverses informations sur la position, la direction et la vitesse de la fusée.

**Paramétrage.** On considère différents réglages différents pour cet environnement dans la figure 30. Nous faisons tourner DQN jusqu'à 1000 itérations et on analyse le comportement de chaque combinaison de paramètres.

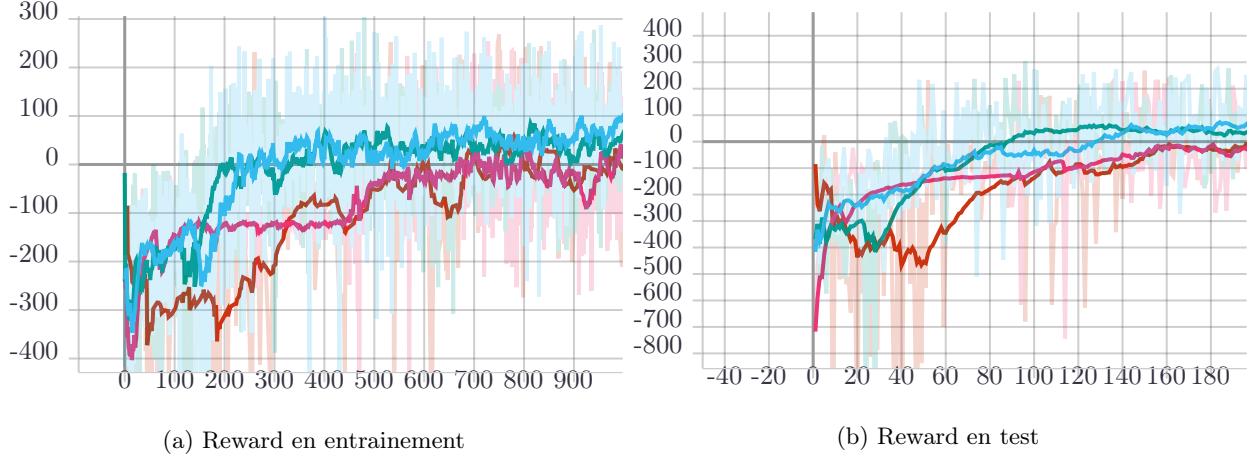


FIGURE 30: Reward par itération sur l'environnement *LunarLander*. **Rouge :**  $\epsilon$ -greedy avec *weight decay* entraîné en utilisant *ADAM* sur un réseau de 200 neurones. **Bleu :** *SGD* avec  $\epsilon$ -greedy de 0.2 sur un réseau de 200 neurones. **Vert :** stratégie  $\epsilon$ -greedy avec  $\epsilon = 0.3$ , entraîné avec *SGD*, *Q* de 2 couches de 100 neurones. **Rose :** Même paramètres que précédemment mais avec la stratégie greedy

**Analyse.** Sur 1000 itération, les algorithmes sont loin de converger vers des scores satisfaisant. on peut observer que la version avec adam est moins stable et que la version avec la stratégie greedy atteint des scores inférieurs. Pour vérifier si avec plus d'itérations, DQN convergerait vers une meilleure solution, On continue l'exécution jusqu'à 3000 itération dans la figure 31.

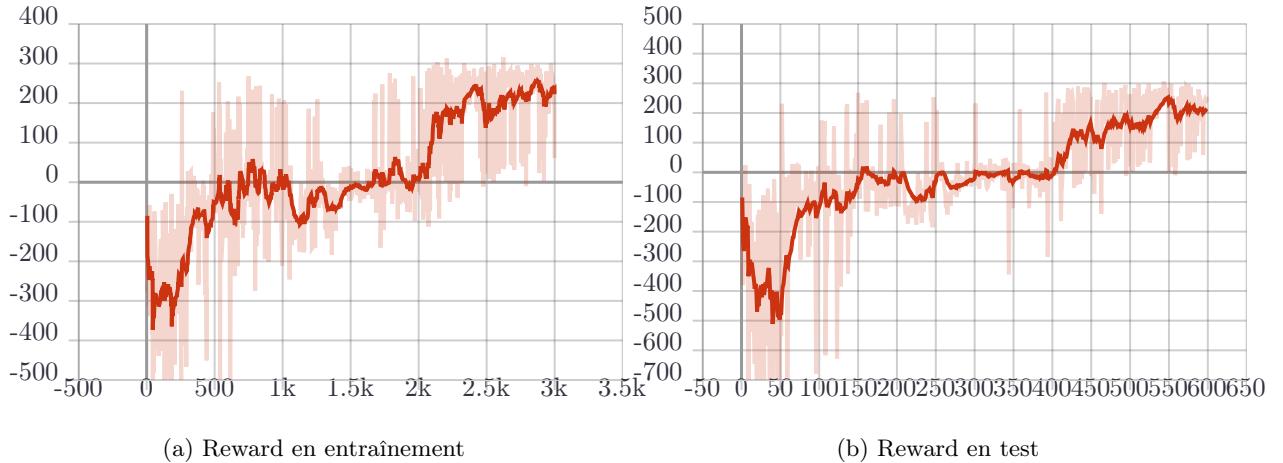


FIGURE 31: Reward par itération sur l'environnement *LunarLander* sur 3000 itérations. **Rouge :**  $\epsilon$ -greedy avec *weight decay* entraîné en utilisant *ADAM* sur un réseau de 200 neurones.

**Analyse.** On remarque qu'on continuant l'exécution, DQN arrive à atteindre des scores de 250. DQN reste très instable et ne suit pas une comportement régulier.

### 4.3 Target Network

Dans cette section, on examine l'ajout d'un réseau cible ou *target network*. Cette technique consiste à séparer l'apprenant  $Q$  et la cible  $Q^-$  dans le but de stabiliser la cible et dé-corréler l'apprenant et la cible.

**Paramètres.** On examine cette technique sur CartPool en gardant les meilleurs paramètres de la figure 28. On fixe l'optimiseur à *ADAM*, le réseau à 200 neurones,  $\alpha = 0.01$ ,  $\gamma = 0.999$ . La seule différence est dans la séparation du réseau courant et cible. On met à jour la cible toutes les 100 actions.

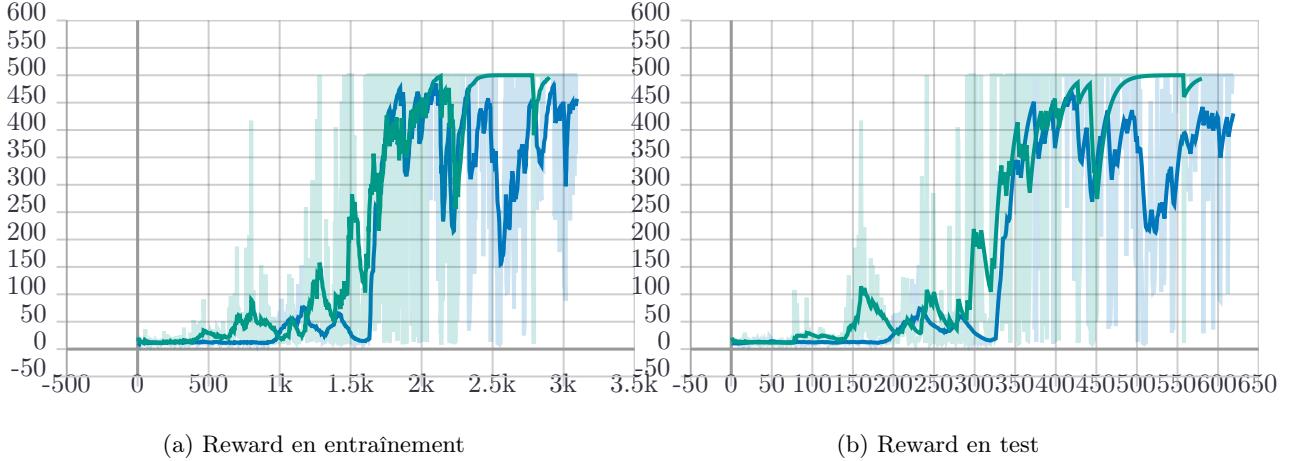


FIGURE 32: Reward par itération sur l'environnement *Cartpool* sur 3000 itérations. **Gris :** version avec target network mis à jour chaque 100 actions. **Vert :** Meilleure version sans target network.

**Analyse.** Comme on peut le remarquer, on changeant seulement le target network, on n'observe pas l'effet escompté. L'agent semble suivre le même comportement qu'avec le target network mais en ne convergeant pas vers un score stable.

### 4.4 Expérience Replay

Dans cette section on examine la technique *Experience replay* qui consiste à enregistrer des transitions dans une mémoire et apprendre à partir de mini-batchs échantillonnés dans cette mémoire. L'algorithme est donné dans la figure 33.

---

**Algorithm 1** Deep Q-learning with Experience Replay

---

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
    Initialise sequence  $s_1 = \{x_1\}$  and preprocess sequenced  $\phi_1 = \phi(s_1)$ 
    for  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \max_a Q^*(\phi_{s_t}, a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Set  $s_{t+1} = s_t, a_t, r_t, \phi_{t+1}$  in  $\mathcal{D}$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
    end for
end for

```

---

FIGURE 33: Algorithme DQN avec Experience replay

**Paramètres.** On teste cette technique d'abord sur CartPool. On garde les meilleurs paramètres de l'expérience 28. On fixe la capacité du buffer à 10000 et la taille du batch à 100.

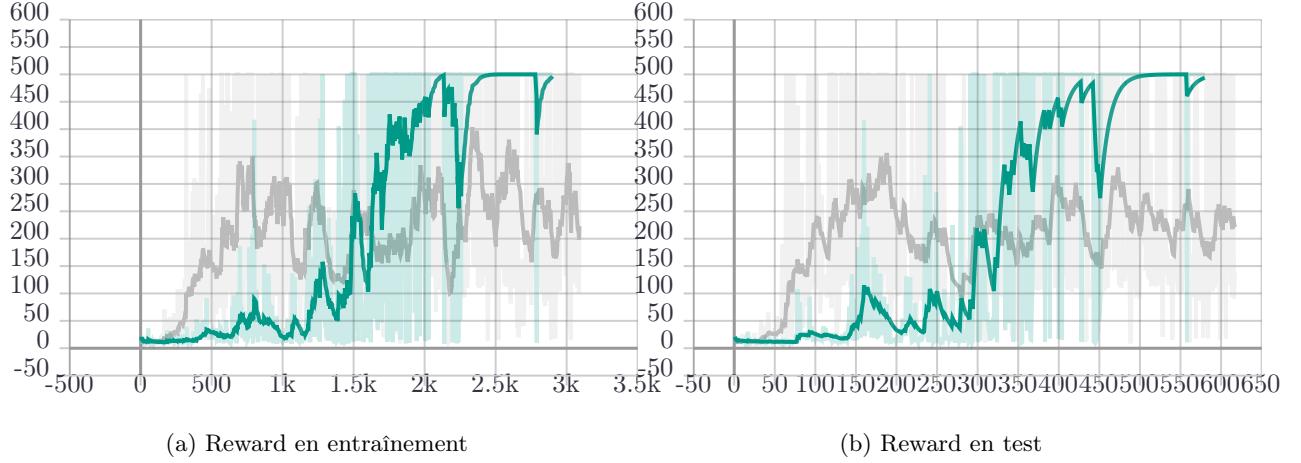


FIGURE 34: Reward par itération sur l'environnement *Cartpool* sur 3000 itérations. **Gris** : version avec expérience replay. **Vert** : Meilleur version sans expérience replay.

**Analyse.** la version avec experience replay atteint des résultats autour de 250 beaucoup plus tot que la version sans experience replay. Cependant la version avec experience replay ne converge pas vers la valeur maximale 500, mais fluctue entre 250 et 400.

#### 4.5 Prioretized Experience Replay

On examine maintenant la technique *Experience replay* avec échantillonage avec priorité. Dans ce cas de figure, on ne tire plus les transitions uniformément depuis le buffer.

**CartPool :**

**Paramètres.** On teste cette technique d'abord sur CartPool. On garde les meilleurs paramètres de l'expérience 28. On fixe la capacité du buffer à 10000 et la taille du batch à 100.

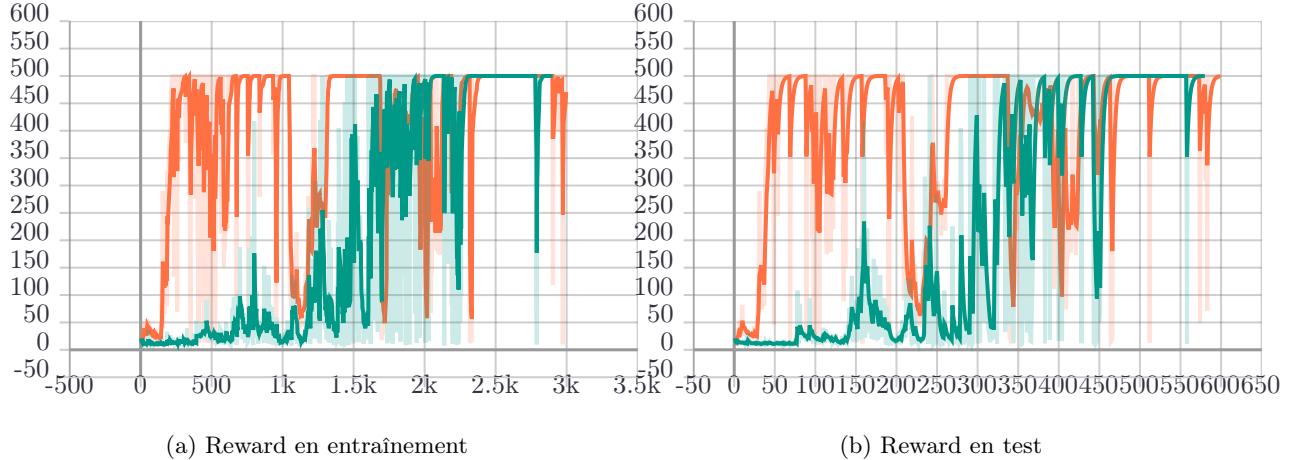


FIGURE 35: Reward par itération sur l'environnement *Cartpool* sur 3000 itérations. **Orange** : version avec prioritized experience replay. **Vert** : meilleure version sans expérience replay.

**Analyse.** la version avec *prioretized experience replay* converge très tôt, mais reste assez instable. Les regrets se dégradent puis se stabilisent. Néanmoins il est évident que cette méthode améliore grandement les

performances de l'agent.

**GridWorld sur le plan 0 :**

**Paramètres.** On teste cette technique d'abord sur GridWorld. On garde les meilleurs paramètres de l'expérience 29. On fixe la capacité du buffer à 10000 et la taille du batch à 10.

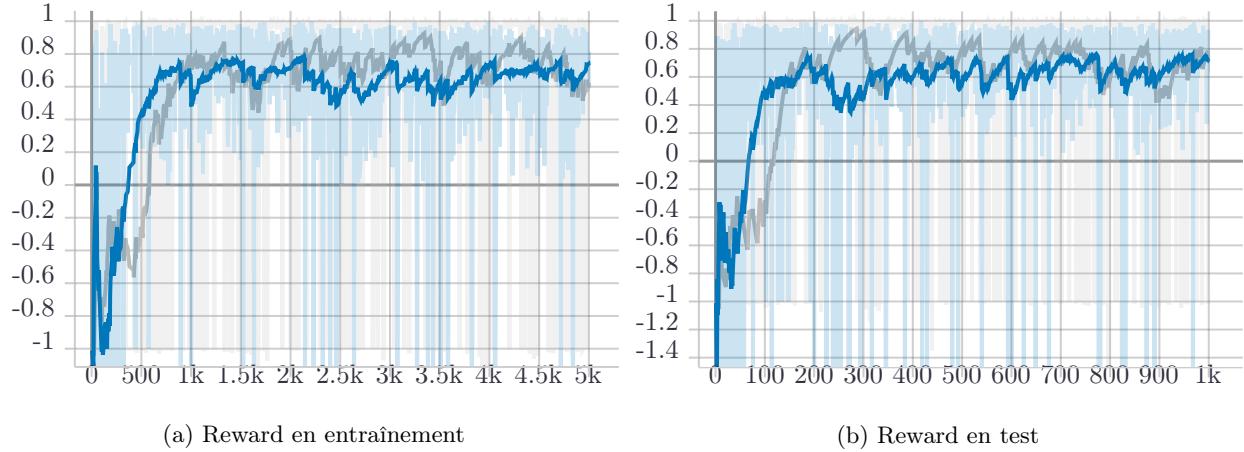


FIGURE 36: Reward par itération sur l'environnement *GridWorld* sur 5000 itérations. **Gris :** version avec priorized experience replay. **Bleu :** meilleure version sans expérience replay.

**Analyse.** la version avec *prioritized experience replay* ne converge pas plus tôt, mais arrive à atteindre légèrement plus que la version sans experience replay mais reste tout aussi instable. Pour les paramètres considérés, cette méthode n'arrive pas au résultat maximal sans fluctuer.

## 4.6 Conclusion

A travers ce tome, nous avons exploré l'algorithme Deep Q network et différentes variantes sur trois environnements différents.

## 5 TME 5. Policy Gradients

### 5.1 Introduction

Le but de ce TME est d'explorer les approches de renforcement Policy Gradients à travers 3 simulations : *CartPool*, *GridWorld* et *LunarLander*.

### 5.2 Algorithme Batch Actor Critic

Nous commençons par l'algorithme Batch Actor critic de base, donné en figure 74.

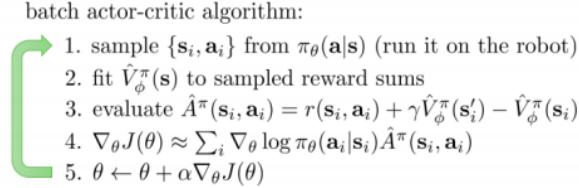


FIGURE 37: Algorithme Batch Actor-Critic

### 5.3 Version avec TD(0)

Nous commençons d'abord par entraîner  $V_\phi$  par TD(0), c'est à dire que  $V_t$  est comparé à  $r_t + \gamma V_{t+1}$  comme dans la figure 74) Nous testons cet algorithme sur les 3 environnements séparément.

#### 5.3.1 Cartpool

**Paramètres.** Nous fixons  $V_\phi$  à un réseau avec une seule couche cachée avec des activations ReLU de même pour  $\pi_\theta$ , la couche cachée contient 128 neurones. Durant nos expériences, nous avons trouvé que partager la première couche entre les deux réseaux permettait d'avoir de meilleures performances. Pour les paramètres du modèle, on fixe le pas d'apprentissage pour  $V_\phi$  et  $\pi_\theta$  à 0.001. On entraîne les deux réseaux à chaque 10 transitions collectées. La cible pour  $V_\phi$  est mise à jour à chaque optimisation. Le discount est fixé à  $\gamma = 0.99$ . Les résultats sont représentés en figure 38.

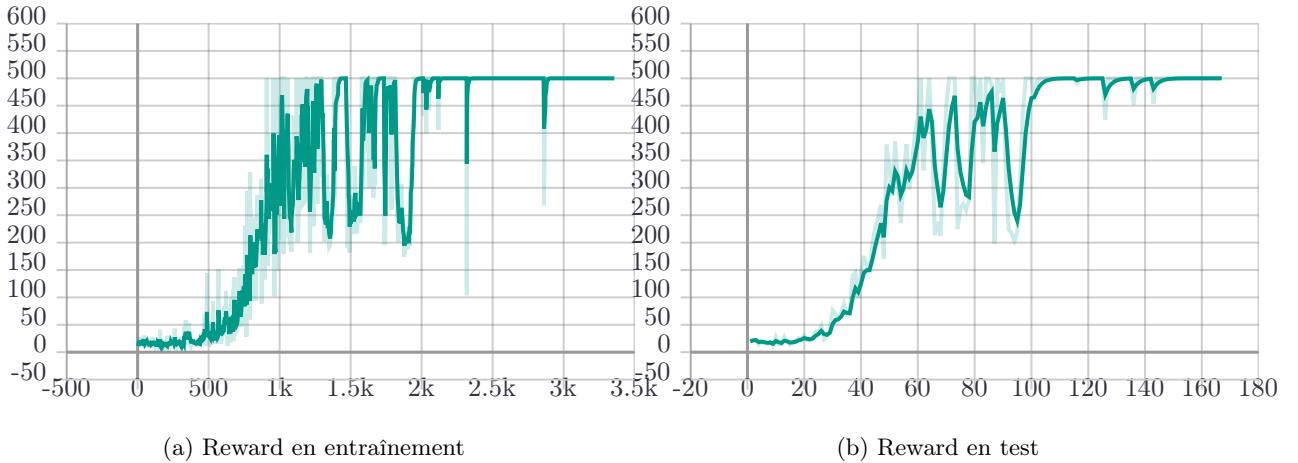


FIGURE 38: Reward par épisode sur Cartpool.

**Analyse.** Batch Actor critic donne des résultats clairement supérieurs au TME4 avec DQN, l'algorithme converge en 1000 épisodes alors que DQN sans *prioritized experience replay* converge en 2500 épisodes. Cependant Actor critic converge plus tard que DQN avec *prioritized experience replay* mais il est bien plus stable. DQN souffre de plus de variance que la version Batch actor critic ce qui explique le gain de stabilité.

### 5.3.2 Lunar Lander

**Paramètres.** Nous gardons les mêmes paramètres que précédemment, seul la taille des couches cachées dans  $V_\phi \pi_\theta$  changent, elles passent à 300 neurones. 40.

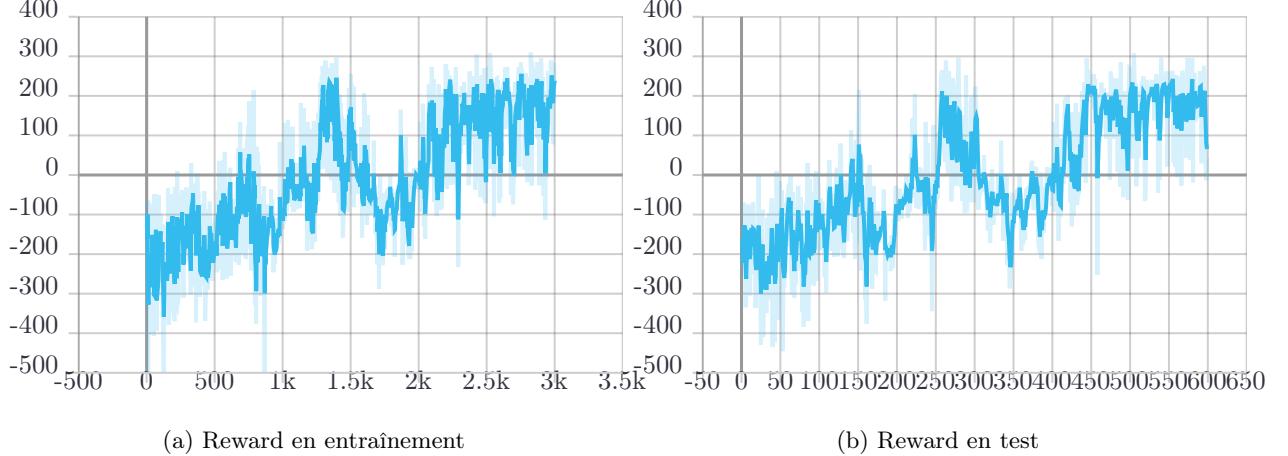


FIGURE 39: Reward par épisode sur lunar lander.

**Analyse.** On atteint un reward supérieur à 200 à partir de 1500 épisodes, c'est à dire que lunar lander est résolu à partir de 1500 épisodes. Cependant, les performances décroissent puis remontent à 2200 épisodes. Nous obtenons ici aussi de meilleurs résultats qu'avec DQN du TME 4 qui atteint pour la première fois un score  $> 200$  qu'après 2000 épisodes. On remarque aussi que cet algorithme est beaucoup plus stable que DQN, puisqu'on sépare l'acteur du critique et on réduit la variance.

### 5.3.3 Grid world

**Paramètres.** On garde les paramètres de la fois précédente sauf pour le nombre de transition pour chaque optimisations qui est fixée à 5 car gridworld effectue moins d'étapes en général. La cible pour  $V_\phi$  est mise à jour à chaque optimisation.  $\gamma = 0.99$ . Les résultats sont représentés en figure ??.

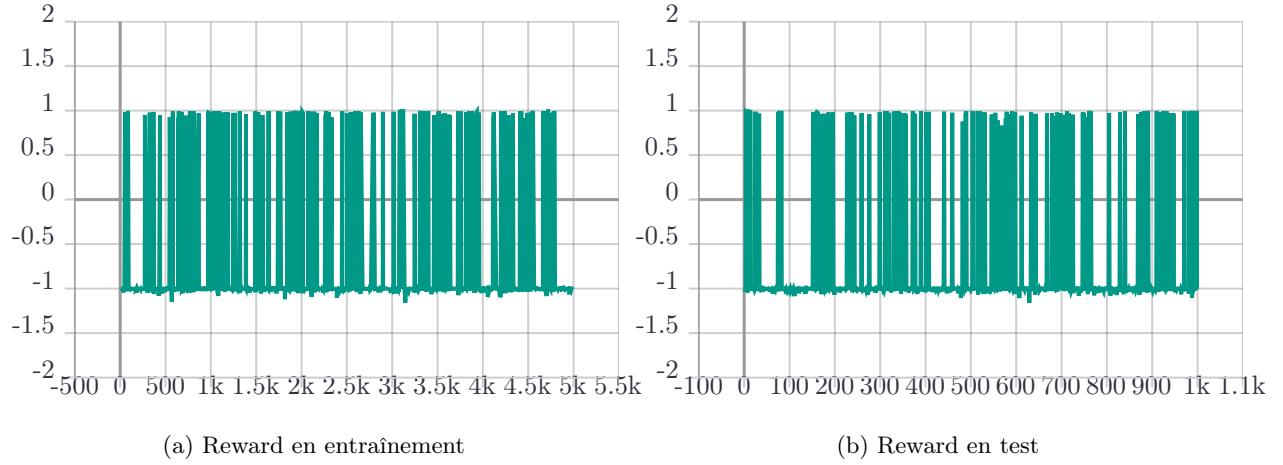


FIGURE 40: Reward par épisode sur lunar lander.

**Analyse.** Cette fois Actor critic donne de très mauvais résultats, l’algorithme ne converge vers aucune valeur et il oscille entre le maximum et le minimum. Alors qu’avec DQN, même s’il est très instable, converge vers 0.8. Ce comportement peut s’expliquer par le fait que Grid world a une combinaison d’états action beaucoup plus grandes, ce qui fait que pour les méthodes policy gradients, il y’a un problème lors de l’étape d’exploration. Une fois le critique convergé, l’algorithme n’explore plus et nous nous retrouvons avec une politique instable. Dans DQN, l’agent continue à explorer avec une probabilité  $\epsilon$ , ce qui fait que le modèle continue à apprendre et ne converge que vers une solution qui satisfait un grand nombre de transitions états actions (on continue à sampler). C’est donc un problème d’exploration suffisante de l’environnement.

#### 5.4 Version avec Rollout MonteCarlo

Dans cette partie  $V_t$  est comparé à  $R_t$ . Nous normalisons également les  $R_t$  dans un but de réduire la variance.

**Paramètres.** Nous gardons les mêmes paramètres que précédemment. On compare les résultats de la méthode MonteCarlo Rollout face aux batch Actor critic avec TD(0). Les résultats sont représentés en figure 41.

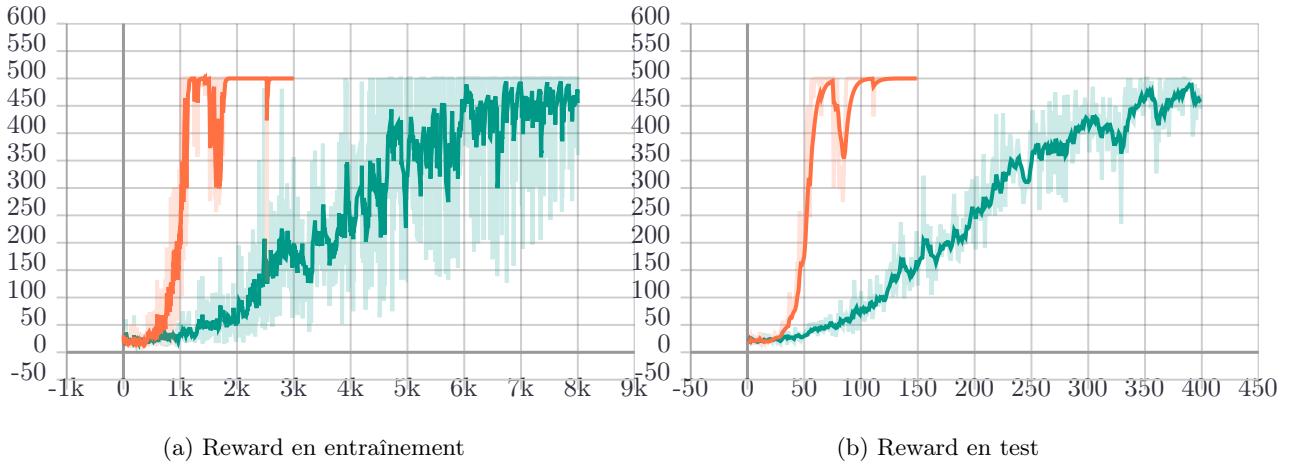


FIGURE 41: Reward par épisode sur Cartpool.

**Analyse.** Actor critic en utilisant les  $R_t$  converge beaucoup plus lentement que la version avec TD(0). C'est un résultat attendu car la version avec rollout montecarlo dépend du nombre d'évaluation de la critique. La politique est améliorée en ajustant les paramètres dans la direction de l'estimation du gradient. Puisque les méthodes Monte-Carlo ont tendance à avoir plus de variances, un grand nombre d'échantillons est nécessaire pour atteindre des approximations correctes. La version avec TD(0) est donc plus appropriée pour cette tâche.

## 5.5 Conclusion

Durant ce tome, nous avons exploré les méthodes de policy gradient à travers deux versions de batch actor critic sur trois environnements différents.

## 6 TME 6-7. Advanced Policy Gradients

### 6.1 Introduction

Le but de ce TME est d'explorer les approches avancées de Policy Gradients à travers 3 simulations : *Cart-Pool*, *GridWorld* et *LunarLander*.

Plus précisément, nous explorons deux versions de PPO. Cette méthode utilise plusieurs étapes de monté de gradient stochastique pour chaque mise à jour de la politique, cela permet d'exploiter au mieux chaque échantillon sans causer de grandes mises à jour si elles deviennent destructives.

### 6.2 Algorithme PPO Adaptative KL

Nous commençons par l'algorithme PPO Adaptative KL. Cette méthode se base sur une mesure de divergence entre deux politiques. Plus concrètement, on rajoute une pénalité de divergence Kullback-Leibler entre l'ancienne politique et la mise à jour. Ceci évite de diverger et réduit la variance.

Nous définissons un plafond (**thereshlod**) qui permet de maîtriser de façon **adaptative** la pénalité KL. Cette contrainte ne doit pas excéder le plafond adaptatif.

#### 6.2.1 Cartpool

**Paramètres.** Nous gardons une architecture similaire au tme 5 où l'acteur et la critique partagent une couche linéaire de 256 neurones. Nous fixons le plafond  $\delta$  à 0.01,  $\beta = 1$  au début. Le pas d'apprentissage  $\alpha = 0.0005$ , le discount  $\gamma = 0.98$ ,  $\lambda = 0.95$ .

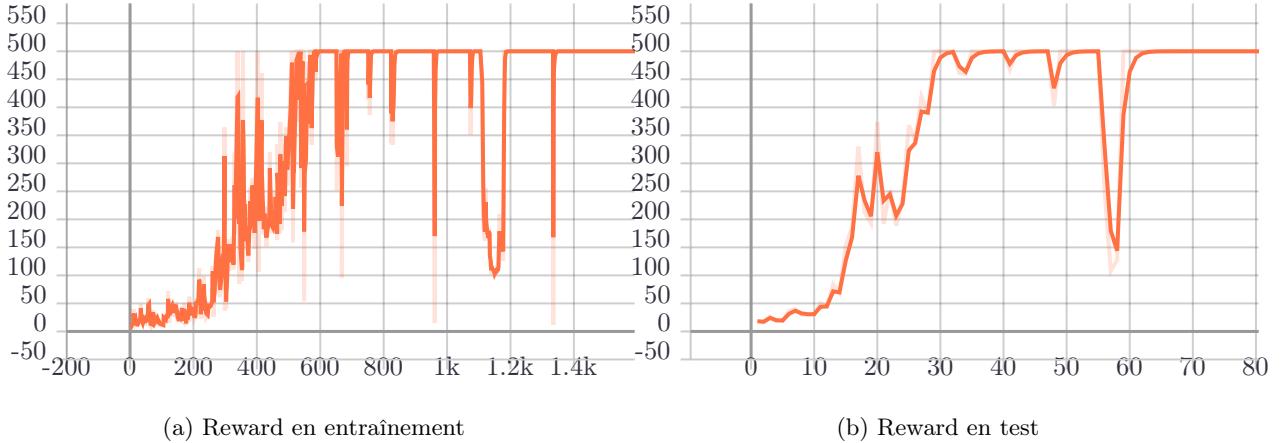


FIGURE 42: Reward par épisode sur Cartpool.

**Analyse.** Cette version converge beaucoup vite que A2C, cela s'explique par le fait qu'on fait K mises à jour à chaque sampling donc beaucoup plus que A2C. On remarque aussi qu'il est beaucoup plus stable, et c'est une conséquence de l'utilisation de la mesure de divergence.

#### 6.2.2 Lunar Lander

**Paramètres.** La différence avec le test précédent est uniquement dans l'architecture du réseau, on augmente le nombre de neurones dans les couches cachées à 300 et on garde tous les autres paramètres intacts.

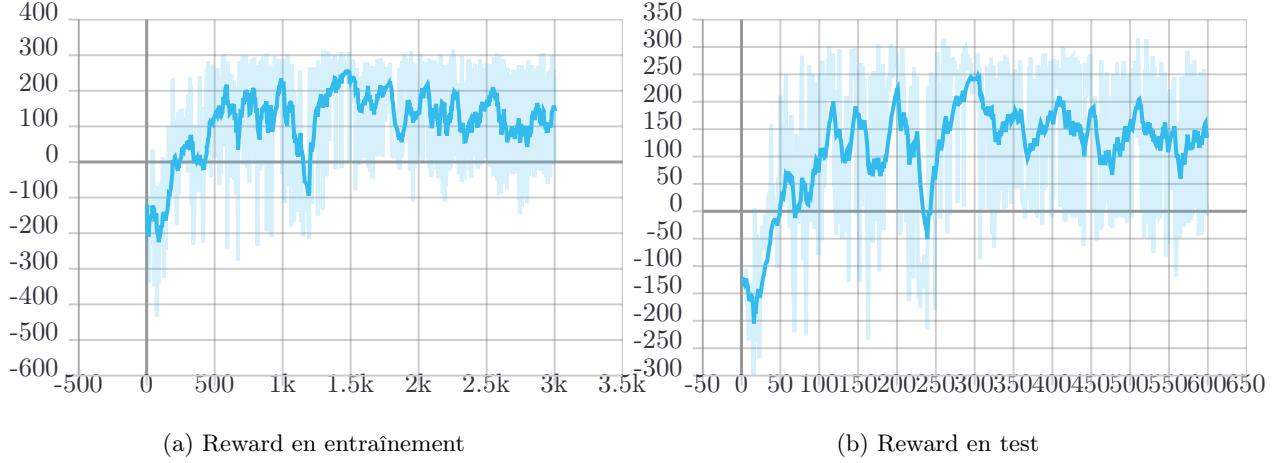


FIGURE 43: Reward par épisode sur lunar lander.

**Analyse.** Ici aussi, on remarque la rapidité de PPO comparé à A2C. On converge en 500 épisodes, alors qu'il en faut 1500 voir plus de 2200 épisodes pour atteindre un score de plus de 200. c'est une conséquence des multiples mises à jour de POO.

### 6.3 KL reversed

On compare ici deux versions de PPO avec un coût KL adaptatif, la version d'origine  $\bar{D}_{KL}(\theta_k|\theta)$  comme dans le papier original et en mode inversé  $\bar{D}_{KL}(\theta|\theta_k)$ .

On teste pour deux environnements, cart pool et lunar lander.

#### 6.3.1 Cartpool

**Paramètres.** On garde les mêmes paramètres que pour la version KL normale. On compare les deux versions en train et en test. On test chaque 20 épisodes pour 5 épisodes.

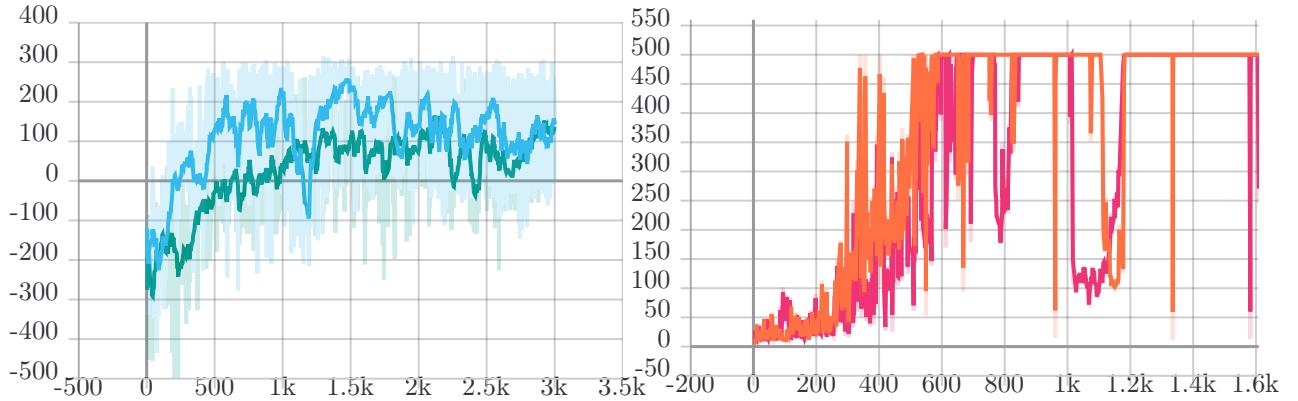


FIGURE 44: Reward par épisode pour la version KL inversée.

**Analyse.** Dans les deux environnements, la version KL inversé est beaucoup moins stable que la version originale, c'est plus flagrant sur cart pool que sur lunar. Ceci est sûrement expliqué par le fait qu'on mesure à chaque fois la  $D_{kl}$  par rapport à une cible mouvante  $\theta_k$  donc la mesure n'a plus le même sens que dans la version originale.

## 6.4 PPO Clipped objective

PPO fait entraîne pour K tour la politique en utilisant un fonction objectif conjuguée clippée. La nouvelle fonction objectif remplace l'objectif policy gradient et améliore la stabilité de l'entraînement en limitant le changement fait à la politique à chaque étape.

On teste sur les deux environnements cartpool et lunar lander.

### 6.4.1 Cartpool

**Paramètres.** L'architecture de  $V_\phi$  et  $\pi_\theta$  partage encore la première couche linéaire de 256 neurones en sortie.  $\alpha = 0.0005$ ,  $\gamma = 0.98$   $\lambda = 0.95$ , quant au paramètres de clipping  $\epsilon = 0.2$ ,  $K = 3$  et on entraîne chaque 20 transitions.

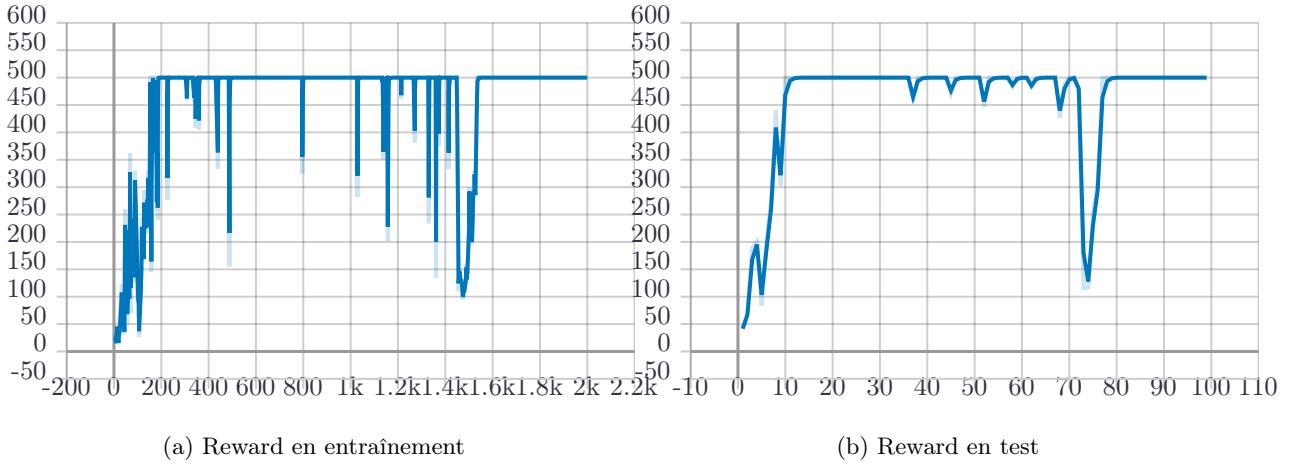


FIGURE 45: Reward par épisode sur Cartpool.

**Analyse.** Cette version converge en 200 épisodes alors qu'avec la version KL, c'est plutôt en 500 épisodes. Elle est aussi beaucoup plus stable. Ceci peut s'expliquer par le fait que la version clippée empêche l'algorithme de sur-apprendre au début et de tomber sur des politiques sous optimales.

### 6.4.2 Lunar Lander

**Paramètres.** La seule différence avec cart pool tient dans l'architecture qui est maintenant de 400 neurones au lieu de 200. les autres paramètres sont les mêmes,  $\alpha = 0.0005$ ,  $\gamma = 0.98$   $\lambda = 0.95$ , quant au paramètres de clipping  $\epsilon = 0.2$ . On teste deux valeurs de K différentes et on optimise chaque 40 transitions collectées.

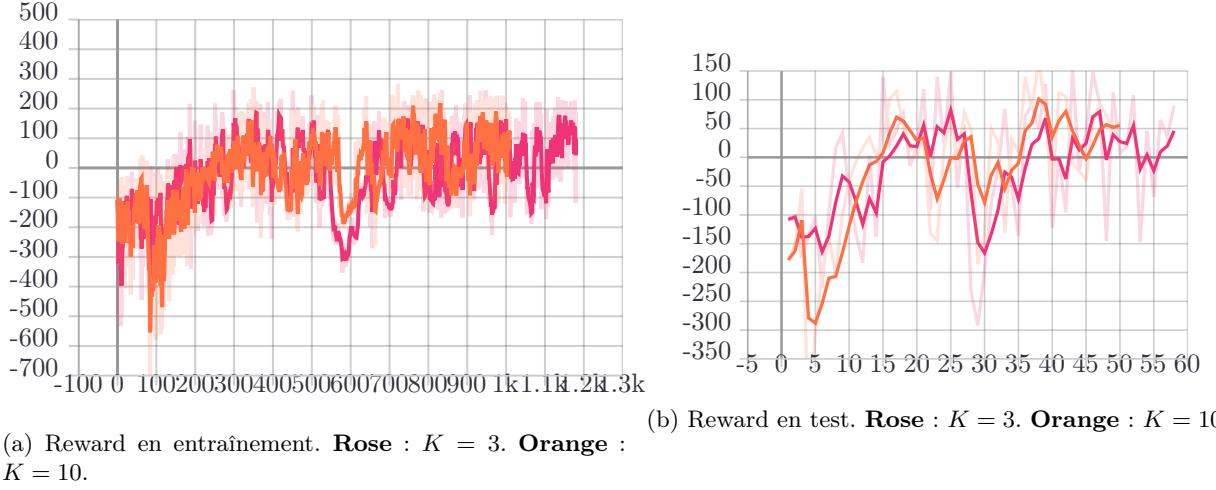


FIGURE 46: Reward par épisode sur lunar lander.

**Analyse.** On remarque qu'on atteint 200 après 300 épisodes alors qu'il en faut 500 pour la version KL. On remarque qu'il n'y a pas de différence entre la version avec  $K = 3$  et  $K = 10$ , cela veut dire qu'après les premières itérations, le gradient devient nul et n'impacte pas grandement les performances.

## 6.5 Comparaison

Dans cette partie, on fait tourner chaque algorithme pour 10 itérations et on affiche les rewards moyennées.

### 6.5.1 Cartpool

**Paramètres.** Pour cette comparaison entre les différents algorithmes, on garde la même architecture pour l'acteur et le critique entre POO KL et PPO clippé et AC. Nous fixons le plafond  $\delta$  à 0.01,  $\beta = 1$  au début. Le pas d'apprentissage  $\alpha = 0.0005$ , le discount  $\gamma = 0.98$ ,  $\lambda = 0.95$ . Pour la version clippée,  $\epsilon = 0.2$ .

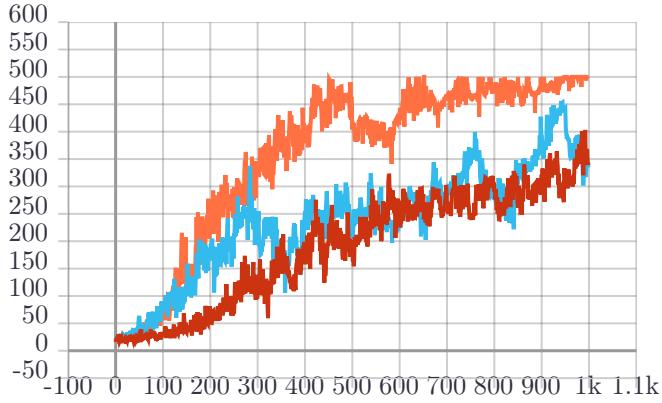


FIGURE 47: Reward par épisode sur cart pool. **Orange** : PPO Clipped. **Bleu** : PPO KL. **Bleu** : A2C.

**Analyse.** Comme remarqué au paravent, la version clippée converge plus rapidement et est plus stable. La version KL est meilleure que A2C mais est rapidement rattrapée. Cela s'explique par le fait qu'A2C nécessite beaucoup plus de mise à jour que la version PPO.

### 6.5.2 Lunar Lander

**Paramètres.** On garde la même architecture pour l'acteur et le critique entre POO KL et PPO clippé et A2C. La couche partagée est fixée à 300 neurones. Nous fixons le plafond  $\delta$  à 0.01,  $\beta = 1$  au début. Le pas d'apprentissage  $\alpha = 0.0005$ , le discount  $\gamma = 0.98$ ,  $\lambda = 0.95$ . Pour la version clippée,  $\epsilon = 0.2$ . On optimise chaque 40 transitions collectées. K est fixée à 10.

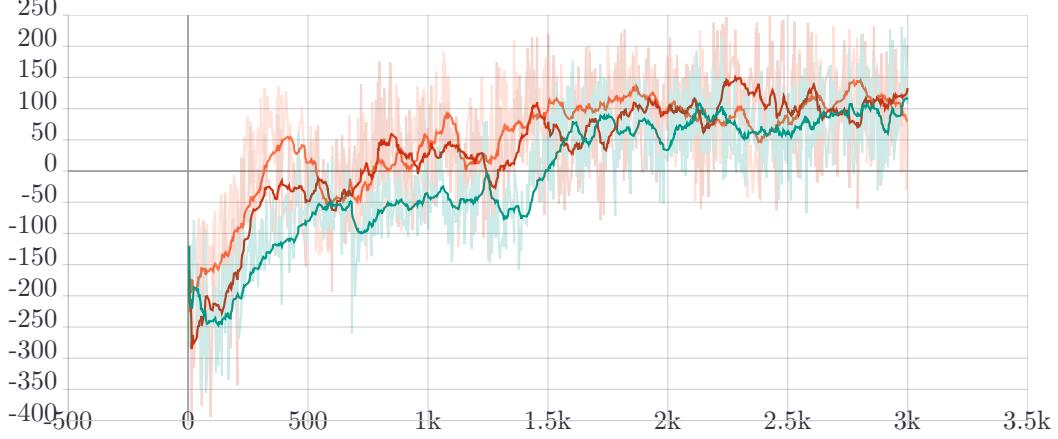


FIGURE 48: Reward par épisode sur lunar lander. **Orange** : PPO Clipped. **Rouge** : PPO KL. **Vert** : A2C.

**Analyse.** Cette fois, nous n'avons pas de distinctions aussi clairs entre les algorithmes. PPO clippée est généralement supérieurs, mais les deux algorithmes se rejoignent souvent. A2C est moins bon au début mais il rejoint les versions de PPO au bout de 650 épisodes.

## 6.6 Bonus : Entropie

On considère une version avec coût d'entropie évitant aux politiques de converger trop rapidement vers des solutions sous-optimales. On essaie ce cout pour les deux versions de PPO.

**Paramètres.** Nous fixons le poids de la pénalité entropique à 0.01, les autres paramètres sont exactement les mêmes que pour les versions sans entropie.

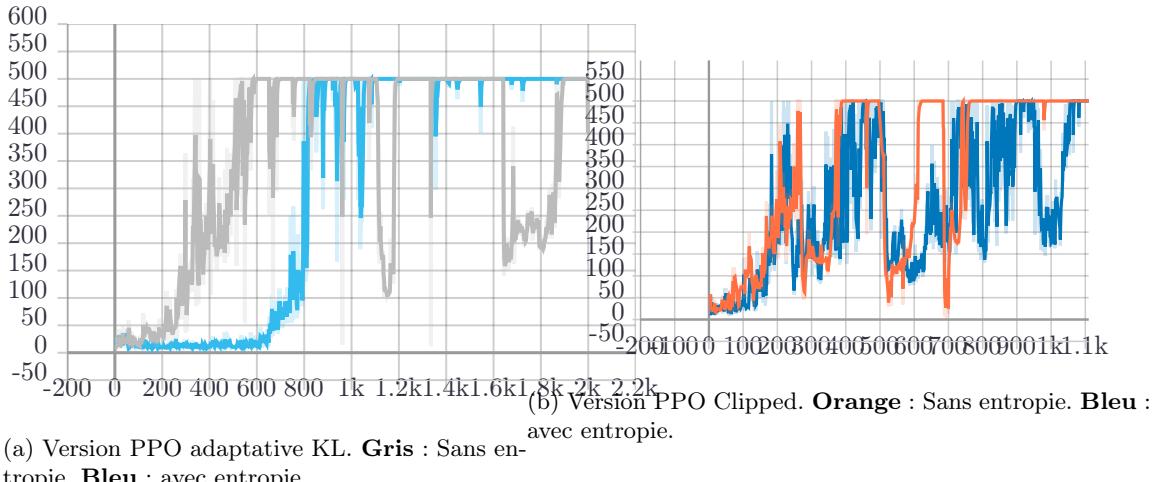


FIGURE 49: Reward par épisode sur cart pool pour les versions de PPO avec entropie

**Analyse pour la version KL.** Pour la version KL avec entropie, on remarque que l'algorithme est beaucoup plus stable, on converge vers 500 avec très peu de perturbation. Il est vrai qu'on converge 200 épisodes plus tard et c'est un résultat attendu car la version avec entropie explore plus et ne converge pas rapidement vers une politique sous optimale.

**Analyse pour la version PPO.** Pour la version PPO avec entropie, on voit que ça ne marche pas du tout, la politique est beaucoup plus instable et c'est encore une fois attendu car le clipping qui est fait aux estimations d'avantages prend déjà en compte l'entropie.

## 6.7 Conclusion

Durant ce tme, nous avons exploré les méthodes de policy gradient à travers deux versions de PPO en utilisant ou non des versions entropiques du coût et des environnements différents.

## 7 TME 8. Continuous Actions

### 7.1 Introduction

Le but de ce TME est d'explorer DDPG. C'est une adaptation des idées de Deep Q-Learning au domaine des actions continues. C'est un algorithme de type actor-critic, model-free basé sur les gradient de la policy qui peut opérer sur un espace d'actions continue. On teste ddpg sur 3 environnements différents et à chaque fois on essaie d'optimiser les paramètres pour pouvoir résoudre les problèmes.

### 7.2 Pendulum

**Environnement.** C'est un environnement à action unidimensionnel dans une plage entre  $[-2, 2]$  avec 3 observations.

**Architecture de Q.** On prend un réseau avec une couche cachée de 128 et une autre de 64, on projette les observations sur 64 dimensions et on fait de même pour les actions. On concatène ces deux vecteurs pour les donner à la première couche cachée.

**Architecture de  $\mu$ .** On prend un réseau avec deux couches cachées de 128 et 64.

**Paramètres.** On pose les pas d'apprentissages  $\epsilon_\mu = 0.0005$ ,  $\epsilon_q = 0.001$ . Le discount est mis à  $\gamma = 0.99$ , la taille de chaque batch est mise à 32. Le paramètres du soft update est mis à  $= 0.005$ . On met à jour 10 fois à la fin de chaque trajectoire.

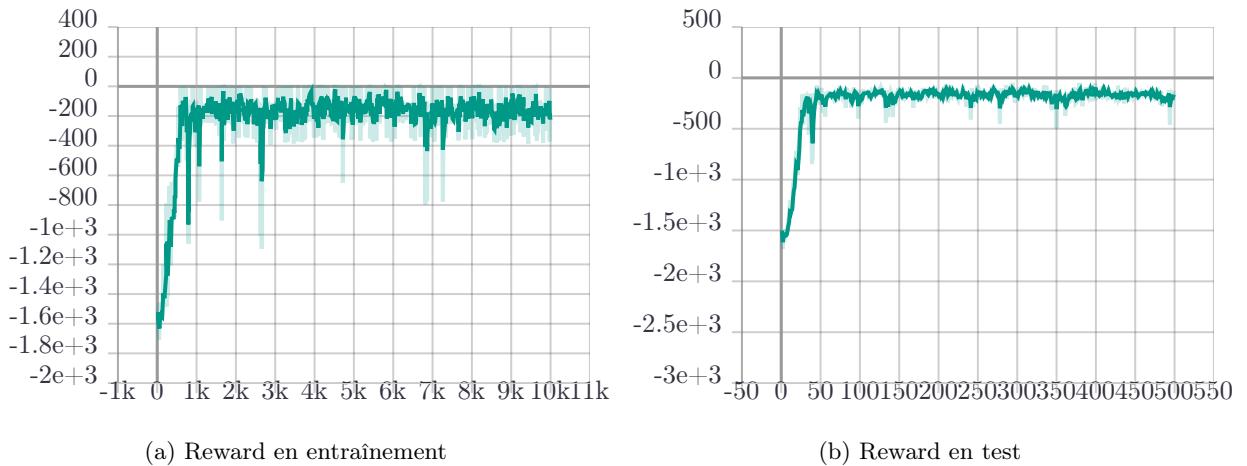


FIGURE 50: Reward par épisode sur Pendulum.

**Analyse.** L'algorithme converge vers une valeur de -200 assez rapidement et reste assez stable. C'est la valeur de résolution du problème. Il converge rapidement du fait du grand nombre de mise à jour (10) qu'on fait à chaque fois.

### 7.3 LunarLander continu

**Environnement.** La différence avec le lunarlander habituel est dans le fait que maintenant on a 8 dimensions d'observations pour 2 actions continue dans entre -1 et 1.

**Architecture de Q.** On prend un réseau avec deux couches cachées une de 400, l'autre de 300. On projette les observations sur 200 dimensions et on fait de même pour les actions. On concatène ces deux vecteurs pour les donner à la première couche cachée.

**Architecture de  $\mu$ .** On prend un réseau avec deux couches cachées de 400 et 300 neurones.

**Paramètres.** On pose les pas d'apprentissages  $\epsilon_\mu = 0.0005$ ,  $\epsilon_q = 0.001$ . Le discount est mis à  $\gamma = 0.99$ , la taille de chaque batch est mise à 32. Le paramètres du soft update est mis à  $\alpha = 0.005$ . On met à jour 20 fois à la fin de chaque trajectoire. On laisse l'algorithme explorer pendant au moins 6000 transitions avant d'entraîner.

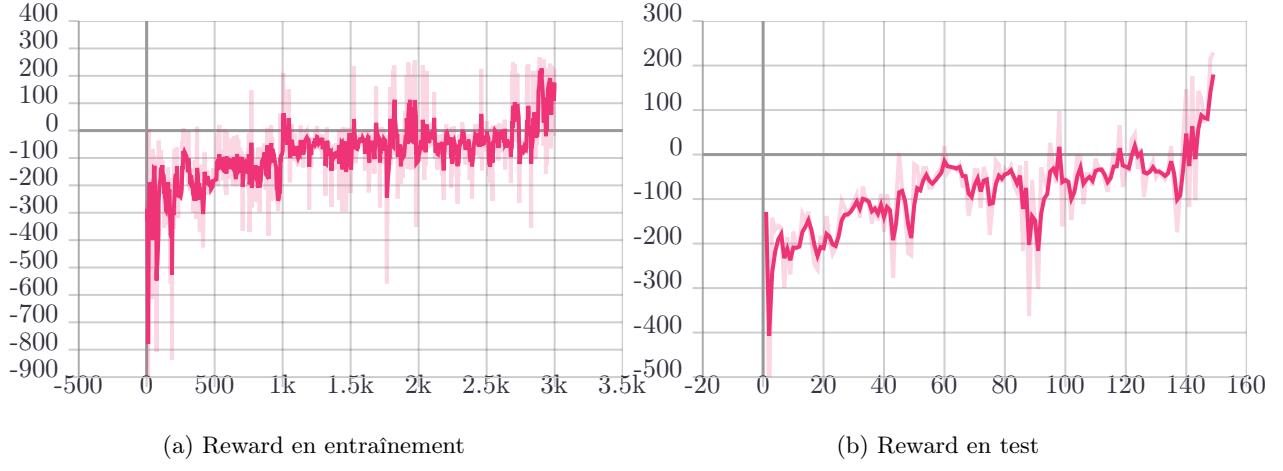


FIGURE 51: Reward par épisode sur lunar lander continuous.

**Analyse.** Comparé à l'environnement discret de lunar lander, DDPG est beaucoup plus lent à atteindre un score de 200. Il le fait éventuellement au bout de plus de 3000 itérations. Cela est peut être du au fait que l'architecture qu'on a considéré est trop lourde ou bien que dans le cas continue, on nécessite plus de de mise à jour du gradient pour arriver à converger.

## 7.4 Mountain Car

**Environnement.** C'est un environnement où il faut faire franchir à une voiture une pente à une dimension. Les observation sont en 2 dimensions  $[-1.2, 0.6]$  et  $[-0.07, 0.07]$ . L'espace des actions est  $\mathbb{R}$  les actions positives pour aller de l'avant et négatives pour revenir en arrière.

**Architecture de Q.** Ici aussi, on prend un réseau avec deux couches cachées une de 400 et l'autre de 300, on projette les observations sur 200 dimensions et on fait de même pour les actions. On concatène ces deux vecteurs pour les donner à la première couche cachée.

**Architecture de  $\mu$ .** On prend un réseau avec deux couches cachées de 400 et 300 neurones.

**Paramètres.** On pose les pas d'apprentissages  $\epsilon_\mu = 0.0001$ ,  $\epsilon_q = 0.001$ . Le discount est mis à  $\gamma = 0.99$ , la taille de chaque batch est mise à 40, la taille finale du batch 10000. Le paramètres du soft update est mis à  $\alpha = 0.001$ . On met à jour 100 fois à la fin de chaque trajectoire.

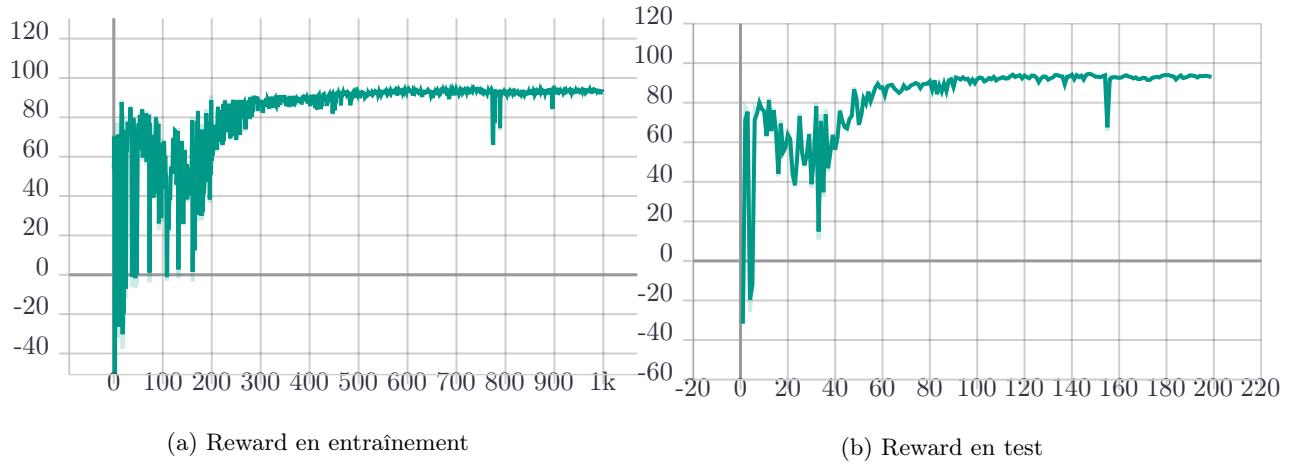


FIGURE 52: Reward par épisode sur Mountain car.

**Analyse.** Sur cet environnement, ddpg arrive à résoudre le problème c'est à dire arriver à un score de plus de 90 très rapidement, au bout de 300 épisodes. C'est du au fait qu'on apprend beaucoup à chaque épisode (100 entraînement par épisodes) et aussi que c'est un environnement plus facile que lunar lander continuous.

## 7.5 Conclusion

Durant ce tme, nous avons exploré la méthode de deep deterministic policy gradient sur trois environnements différents.

## 8 TME 9. Generative Adversarial Networks

### 8.1 Introduction

Les "Generative Adversarial Networks - GANs", sont des modèles génératifs à densité implicite (on ne peut pas calculer  $p(x)$  mais on peut en échantillonner). Supposons que nous avons des données  $x_i \sim p_{data}$ , et que nous voulons échantillonner de  $p_{data}$ . L'idée est d'introduire une variable latente  $z$  suivant une distribution à priori  $p(z)$ . On échantillonne  $z \sim p(z)$ , et on la passe à un réseau qu'on appelle générateur  $x = G(z)$ .  $x$  suit donc la densité implicite du générateur  $p_G$  et nous voulons avoir  $p_{data} \approx p_G$ . Pour ce faire, on construit une architecture composée de deux réseaux :

- Un réseau discriminateur  $D$ , qui prend en entrée des données réelles et des données générées par  $G$ . Ce réseau est entraîné dans le but de distinguer les données réelles des données générées (classification 0|1).
- Un réseau générateur  $G$ , qui reçoit des variables  $z \sim p(z)$ , et qui est entraîné à convertir ces dernières en des données échantillonées de  $p_G \approx p_{data}$ , en trompant le réseau discriminateur.

Les deux réseaux sont entraînés conjointement, avec une perte de jeu mini-max :

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

- Dans le premier terme  $x \sim p_{data}$ .  $D$  veut maximiser  $D(x)$  puisqu'il s'agit de données réelles.
- Dans le deuxième terme,  $z \sim p(z)$ .  $G$  veut maximiser  $D(G(z))$  (tromper le discriminateur) et  $D$  veut minimiser  $D(G(z))$  puisqu'il s'agit de données générées.

Il est aisément vérifiable mathématiquement que ce jeu mini-max atteint son objectif pour  $p_G = p_{data}$ . Plus précisément, le minimum global est atteint pour :

1.  $D_G^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_G(x)}$  (discriminateur optimal pour tout  $G$ ).
2.  $p_G(x) = p_{data}(x)$  (générateur optimal pour le discriminateur optimal).

Cependant,  $G$  et  $D$  sont des réseaux de neurones, et donc nous n'avons aucune garantie sur leur convergence vers la solution optimale.

Les réseaux  $D$  et  $G$ , sont entraînés par mise à jour alternée de gradient. Pour  $t \in \{1, \dots, T\}$  :

1. Mise à jour de  $D$  :  $D = D + \alpha_D \times \frac{dV}{dD}$
2. Mise à jour de  $G$  :  $G = G - \alpha_G \times \frac{dV}{dG}$

Au début de l'entraînement,  $G$  est peu performant et  $D$  peut facilement distinguer les données réelles des données générées.  $D(G(z))$  est très proche de 0. On fait face à un cas de **Vanishing Gradient**. Pour régler ce problème, on entraîne  $G$  à minimiser  $-\log(D(G(z)))$  au lieu de  $\log(1 - D(G(z)))$ . Par conséquent,  $G$  aura de forts gradients au début de l'apprentissage.

### 8.2 Résultats :

#### 8.2.1 Architecture :

Dans ce TME, on va entraîner des DCGANs utilisant des réseaux convolutifs pour le discriminateur comme pour le générateur, et ce, pour une tâche de génération de visages avec une base de données de 202599 de photos de visages de célébrités. En voici un échantillon :



FIGURE 53: Échantillon d'images du dataset "img align celeba"

L'architecture du discriminateur est la suivante :

```

Discriminator(
    (main): Sequential(
        (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (1): LeakyReLU(negative_slope=0.2, inplace=True)
        (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (4): LeakyReLU(negative_slope=0.2, inplace=True)
        (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (7): LeakyReLU(negative_slope=0.2, inplace=True)
        (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
        (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (10): LeakyReLU(negative_slope=0.2, inplace=True)
        (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
        (12): Sigmoid()
    )
)

```

Il s'agit d'une série de convolutions, avec des activations ReLu et des batchNorm. Ce réseau réduit la taille de l'image en entrée. Le dernière couche du réseau est une sigmoïde.

L'architecture du générateur est la suivante :

```

Generator(
    (main): Sequential(
        (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
)

```

```

(1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(2): ReLU(inplace=True)
(3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(5): ReLU(inplace=True)
(6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(8): ReLU(inplace=True)
(9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(11): ReLU(inplace=True)
(12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
(13): Tanh()
)
)

```

Il s'agit d'une série de déconvolutions (convolution transposée), avec des activations ReLU et des batch-Norm. Ce réseau augmente la taille du code donné en entrée pour obtenir une sortie de même dimension que l'image. La dernière couche du réseau est une tanh.

### 8.2.2 Hyper-paramètres :

- learning rate :  $1e - 4$  pour le discriminateur comme pour le générateur.
- batch size : 128.
- dimension de l'espace latent : 100.
- 1 mise à jour des paramètres de D, suivie d'une mise à jour des paramètres de G, pour chaque batch.  
À chaque époque, on parcourt tous le dataset.

## 8.3 Évolution de l'erreur du discriminateur et celle du générateur durant l'entraînement :

On affiche l'erreur du générateur  $\log(D(G(z)))$  et celle du discriminateur  $\log(D(x)) + \log(1 - D(G(z)))$  pour chaque époque de l'entraînement :

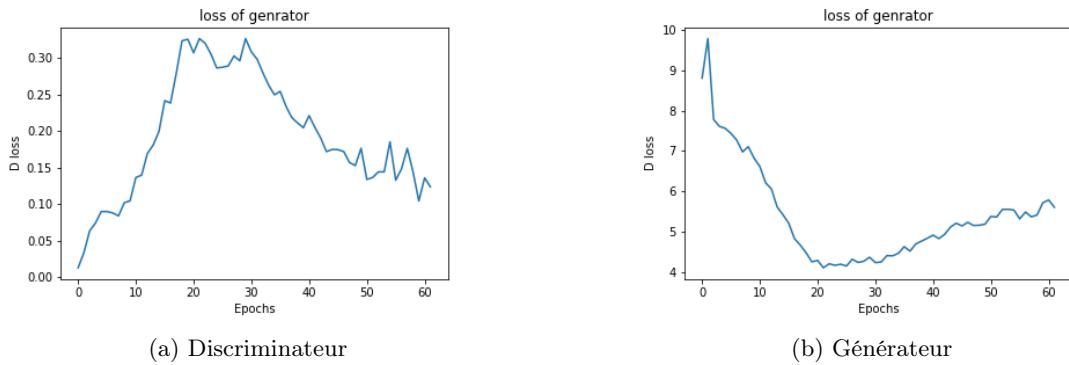


FIGURE 54: Erreurs de G et de D durant l'entraînement.

L'erreur du discriminateur augmente au début de l'apprentissage augmentant de 0 à 0.3 au bout de 20 époques, ensuite la loss atteint un plateau durant 10 époques (semblant indiquer un équilibre) et décroît

de l'époque 30 à l'époque 40. La loss du générateur, quant à elle, se comporte inversement (décroît, stagne et remonte), cette dualité s'explique par le fait que la loss est formulée comme un jeu mini-max. Il est difficile d'interpréter les loss du générateur et du discriminateur compte tenu de la nature mini-max de la loss (le générateur et le discriminateur sont en "duel"), une amélioration de l'un implique forcément une détérioration de l'autre. Ces courbes ne sont pas très intuitives et ne donnent pas de réelles indications sur la qualité des données générées, elles indiquent uniquement que  $D$  et  $G$  n'ont pas atteint un équilibre.

#### 8.4 Évolution de la classification des données réelles et des données générées par le discriminateur :

On affiche la moyenne des sorties du discriminateur sur les données réelles ainsi que sur les données générées pour chaque époque de l'entraînement :

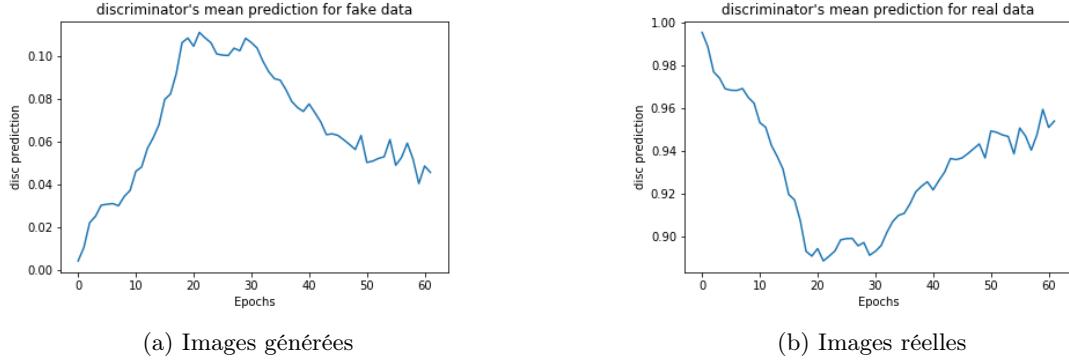


FIGURE 55: Moyenne de D sur les images réelles et sur les images générées (sortie de G).

Au début de l'apprentissage (époque 0), le discriminateur est performant (parfaite classification), car  $G$  est peu performant et  $D$  peut facilement distinguer les données réelles des données générées. Au fur et à mesure  $G$  apprend et donc la prédiction de  $D$  sur les données générées augmente ( $G$  arrive à tromper  $D$  sur quelques exemples), ensuite la tendance s'inverse à partir de l'époque 30. La dualité est aussi observée ici, et ces courbes correspondant exactement aux courbes de pertes traitées plus haut.

#### 8.5 Image générées au long de l'apprentissage :

On affiche les images générées tout au long de l'apprentissage :

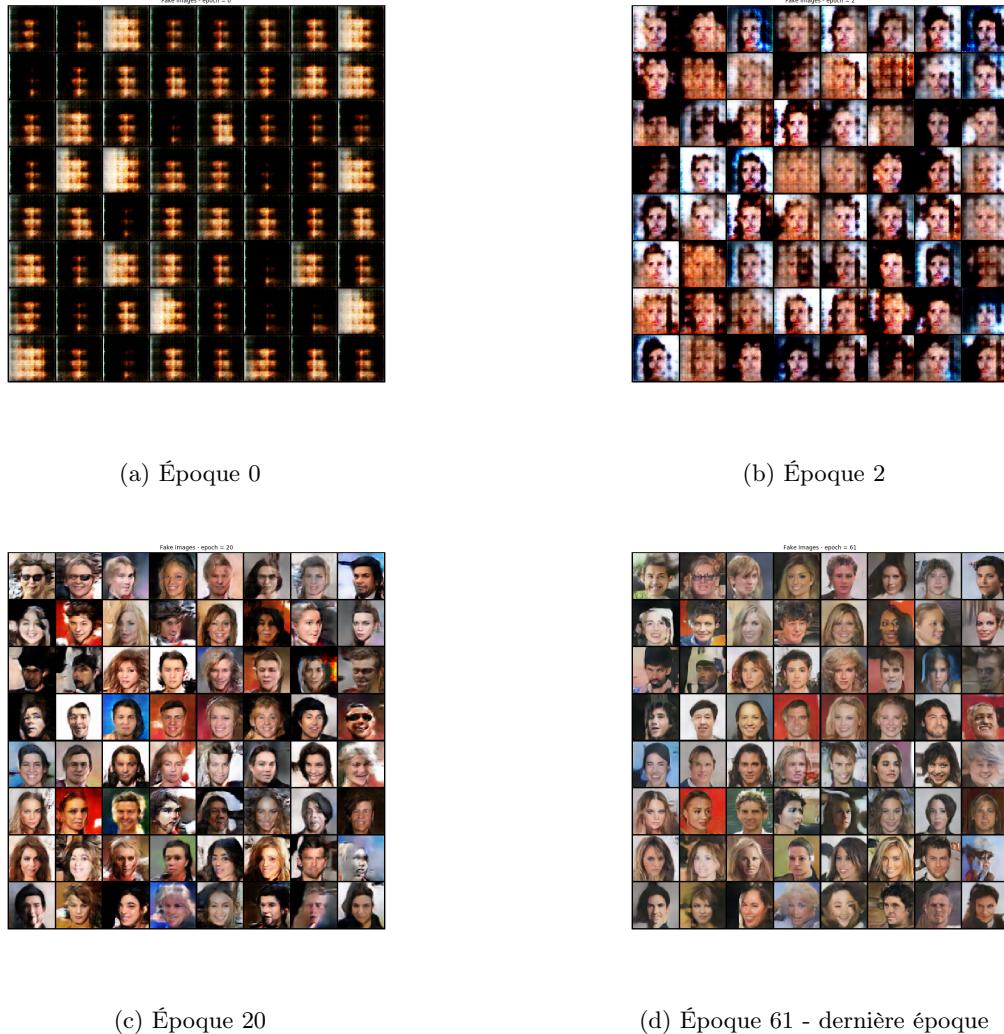


FIGURE 56: Images générées pour une variable latente  $z$  fixée au long de l'apprentissage.

À la fin de la première époque (époque 0), les images générées sont indistinguables. Au bout d'une époque supplémentaire, on commence à distinguer des formes de visages. Au bout de 20 époques, des visages sont clairement générés (avec des textures différentes, couleurs...). À la fin de l'apprentissage (époque 61), certains visages sont extrêmement réalistes. On en conclut que la non-convergence (pas d'équilibre) ne veut pas dire que le réseau n'apprend rien comme le prouvent ces échantillons. Et comme dit précédemment, puisqu'il s'agit de réseaux de neurones rien ne nous garantit que l'état de stabilité est nécessairement avantageux, ça ne nous laisse que les échantillons pour juger de la qualité de l'apprentissage.

## 8.6 GANs sur le dataset MNIST :

### 8.6.1 Architecture :

Dans cette, on va entraîner un GAN sur le dataset de chiffres manuscrits "MNIST".

L'architecture du discriminateur est la suivante :

```

Discriminator(
    (lin1): Linear(in_features=784, out_features=1024, bias=True)
    (lin2): Linear(in_features=1024, out_features=512, bias=True)
    (lin3): Linear(in_features=512, out_features=256, bias=True)
    (lin4): Linear(in_features=256, out_features=1, bias=True)
)

```

Il s'agit d'une succession de couches linéaires qui réduisent la dimension de l'image à 1 (classification binaire), ces couches linéaires font le parallèle avec les convolution de l'architecture DCGANs. On emploie aussi des activations Leaky Relu après chaque couche intermédiaire et une activation tanh pour la dernière couche.

L'architecture du générateur est la suivante :

```

Generator(
    (lin1): Linear(in_features=100, out_features=256, bias=True)
    (lin2): Linear(in_features=256, out_features=512, bias=True)
    (lin3): Linear(in_features=512, out_features=1024, bias=True)
    (lin4): Linear(in_features=1024, out_features=784, bias=True)
)

```

Il s'agit d'une successions de couches linéaires qui augmentent la dimension du code  $z$  donné en entrée, la sortie de ce réseau est de même dimension que les images. Ces couches linéaires font le parallèle avec les dé-convolutions dans l'architecture DCGANs. On emploie aussi des activations Leaky Relu après chaque couche intermédiaire et un dropout. La dernière couche est une sigmoïde pour la normalisation entre 0 et 1.

### 8.6.2 Hyper-paramètres :

- learning rate :  $1e - 4$  pour le discriminateur comme pour le générateur.
- batch size : 64.
- dimension de l'espace latent : 100.
- dropout : 0.3.
- 1 mise à jour des paramètres de D, suivie d'une mise à jour des paramètres de G, pour chaque batch.  
À chaque époque, on parcours tous le dataset.

## 8.7 Évolution de l'erreur du discriminateur et celle du générateur durant l'entraînement :

On affiche l'erreur du générateur  $\log(D(G(z)))$  et celle du discriminateur  $\log(D(x)) + \log(1 - D(G(z)))$  pour chaque époque de l'entraînement :

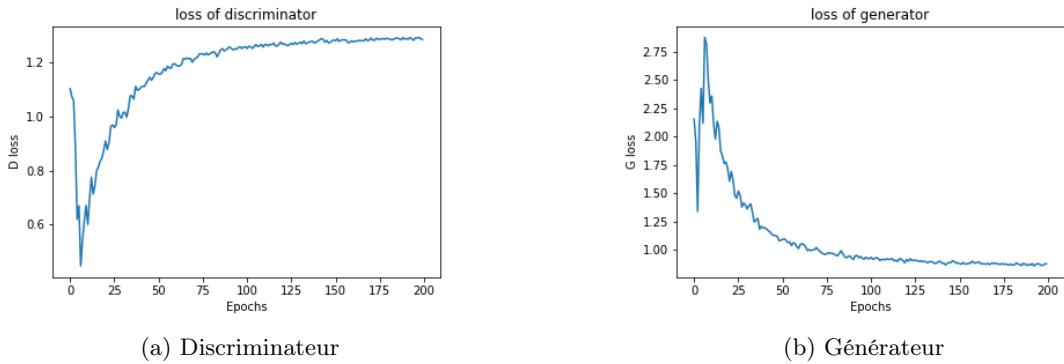


FIGURE 57: Erreurs de G et de D durant l'entraînement.

On observe une dualité dans les deux courbes ce qui est encore une fois dû à la formulation de la loss. La loss du discriminateur décroît au bout de la première époque ( $G$  peu performant) et remonte ( $G$  apprend), et inversement pour la loss du générateur. Cependant, contrairement à la loss dans l'architecture DCGANs, il semble ici que les deux réseaux aient atteint un équilibre.

## 8.8 Évolution de la classification des données réelles et des données générées par le discriminateur :

On affiche la moyenne des sorties du discriminateur sur les données réelles ainsi que sur les données générées pour chaque époque de l'entraînement :

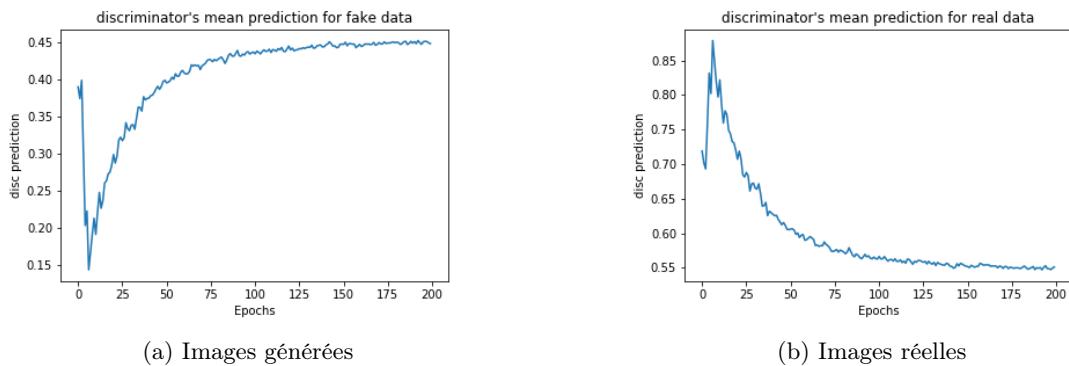
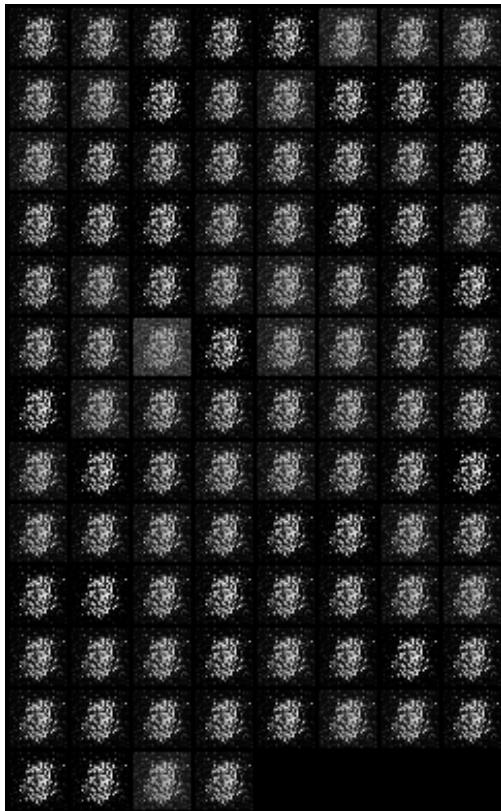


FIGURE 58: Moyenne de D sur les images réelles et sur les images générées (sortie de G).

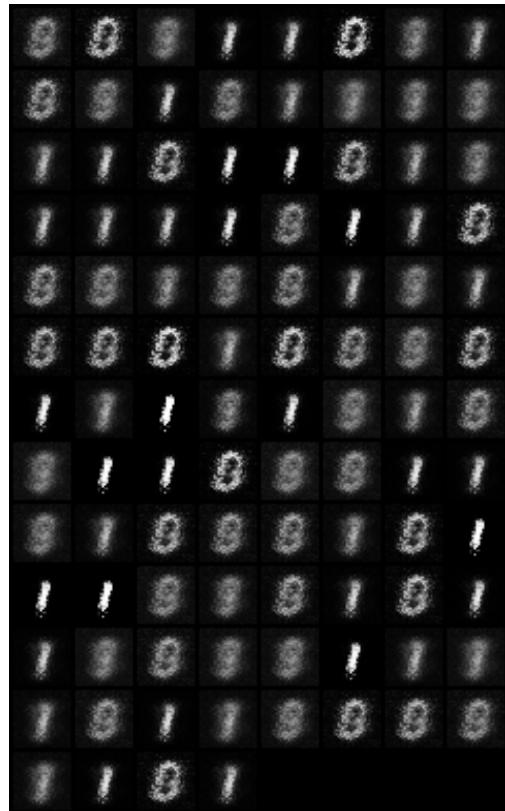
Les courbes correspondent exactement au résultats observé en affichant les pertes. Il est à noter que pour l'équilibre vers lequel les deux réseaux convergent, la prédiction moyenne du discriminateur pour les données réelles et les données générées soit très proches : 0.45 pour les données générées et 0.55 pour les données réelles, ce qui indique une bonne qualité des images générées.

## 8.9 Image générées au long de l'apprentissage :

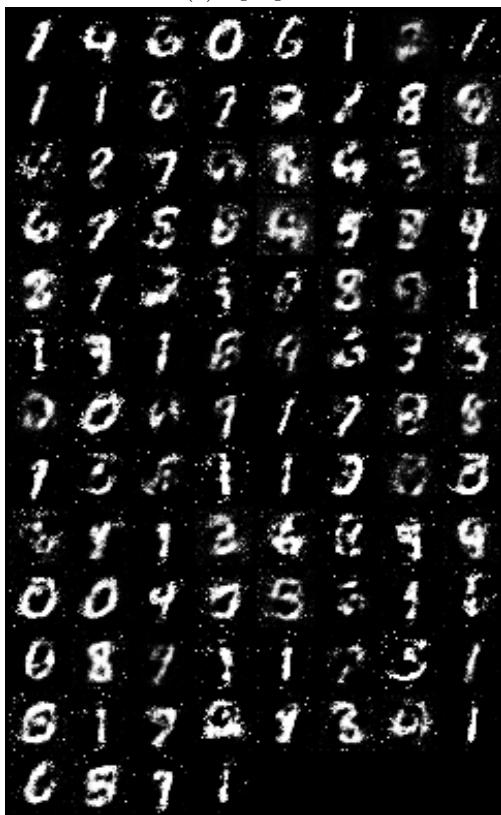
On affiche les images générées tout au long de l'apprentissage :



(a) Époque 0



(b) Époque 4



(c) Époque 20



(d) Époque 199 - dernière époque

FIGURE 59: Images générées pour une variable latente  $z$  fixée au long de l'apprentissage.

Comme pour la tâche précédente, plus on avance dans l'apprentissage plus la qualité des chiffres générées est meilleure.

## 9 TME 10. Variational Auto-Encoders

### 9.1 Introduction

Les auto-encodeurs variationnels sont des modèles génératifs à densité explicite approximée (on calcule une approximation de  $p(x)$ ). Supposons que nous avons des données  $x_i \sim p_{data}$ , et que nous voulons échantillonner de  $p_{data}$ . L'idée, comme dans les GANs, est l'introduction d'une variable latente  $z$ , cependant le raisonnement pour échantillonner de  $p_{data}$  est différent.

L'hypothèse qui est faite dans les VAE est que les données  $\{x_i\}_{1 \leq i \leq N} \sim p_{data}$  sont générées d'une variable latente non observée  $z$  qui suit un prior  $p(z)$ . Donc pour échantillonner  $x \sim p_{data}$ , il suffit de :

- Échantillonner  $z \sim p(z)$ .
- Échantillonner  $x$  de  $p_\theta(x|z)$ .

Le but de l'apprentissage étant de maximiser la vraisemblance des données  $p_\theta(x)$ , cette formulation n'est pas concevable, car pour obtenir  $p_\theta(x)$  deux alternatives sont possibles :

- Marginaliser :  $p_\theta(x) = \int_z p_\theta(x, z) dz = \int_z p_\theta(x|z)p_\theta(z) dz$ , ce qui n'est pas calculable ( $z$  est non observée).
- Règle de bayes :  $p_\theta(x) = \frac{p_\theta(x|z)p_\theta(z)}{p_\theta(z|x)}$ , et nous n'avons pas  $p_\theta(z|x)$ .

La solution est donc d'entraîner un réseau pour approximer  $p_\theta(z|x)$  par  $q_\phi(z|x)$ . Ensuite, on peut approximer la vraisemblance  $p_\theta(x)$ , par ce qu'on appelle la "Variational Lower Bound" donnée par :

$$\log p_\theta(x) \geq \mathbb{E}_{z \sim q_\phi(z|x)} [\log(p_\theta(x|z)) - dkl(q_\phi(z|x), p(z))]$$

En maximisant cette borne inférieure, on espère maximiser la vraisemblance.

Au final, l'architecture est composée de deux réseaux :

- Un réseau **Encodeur**, qui prend en entrée les données  $x_i \sim p_{data}$ , et qui renvoie la distribution  $q_\phi(z|x) = N(\mu_{z|x}, \Sigma_{z|x})$ .
- Un réseau **Décodeur**, qui prend en entrée les variables latentes  $z$ , et qui renvoie la distribution  $p_\theta(x|z)$

Ces deux réseaux sont entraînés conjointement de la manière suivante :

1. Donner à l'encodeur les données  $x_i$ , et récupérer une distribution sur les codes  $z$ .
2. Échantillonner  $z$  de la distribution récupérée de 1.
3. Donner les codes  $z_i$  échantillonnés dans 2 au décodeur pour récupérer une distribution sur les données  $x_i$ .
4. Minimiser la borne inférieure variationnelle, qui se décompose en deux parties :
  - $dkl(q_\phi(z|x), p(z))$  : la distribution en sortie de l'encodeur doit être proche du prior  $p(z)$ , il s'agit d'un terme de régularisation.
  - $\mathbb{E}_{z \sim q_\phi(z|x)} [\log(p_\theta(x|z))]$  : Les données  $x_i$  sont vraisemblablement sous la distribution en sortie du décodeur. Il s'agit d'un terme de reconstruction.

Après entraînement, les étapes pour échantillonner de  $p_{data}$  sont les suivantes :

1. Échantillonner  $z \sim p(z)$ .
2. Récupérer à partir de  $z$ , une distribution sur les données  $x_i$  grâce au décodeur.
3. Échantillonner de la distribution obtenue en 2.

### 9.2 Résultats :

### 9.3 Architecture :

Dans ce TME, on va entraîner un VAE sur le dataset de chiffres manuscrits "MNIST". On fait l'hypothèse que  $p(z) = N(0, I_d)$ . On fait aussi l'hypothèse que  $q_\phi(z|x) = N(\mu_{z|x}, \Sigma_{z|x})$ . On prend  $\Sigma_{z|x}$  une matrice diagonale pour des raisons de complexité. Car en effet, si on prend une dimension de l'espace latent à 100 avec un batch size de 64, la taille de la matrice de variance covariance sera  $64 \times 100 \times 100$ , avec cette formulation on se retrouve avec une taille de  $64 \times 100$ . De plus, avec une matrice  $\Sigma_{z|x}$  diagonale la divergence de kullback-leibler  $dkl(q_\phi(z|x), p(z))$  a une forme explicite.

Le décodeur quant à lui, est soldé par une sigmoïde. Avec nos chiffres normalisées en  $[0, 1]$ , ce dernier renvoie directement une image échantillonée de  $p_\theta(x|z)$ . Pour le terme de reconstruction, on utilise la binary cross entropy loss entre l'image originale et la reconstruction en sortie du décodeur.

L'architecture du VAE est la suivante :

```
VAE(
  (encoder): Encoder(
    (lin1): Linear(in_features=784, out_features=256, bias=True)
    (sigma): Linear(in_features=256, out_features=2, bias=True)
    (mu): Linear(in_features=256, out_features=2, bias=True)
  )
  (decoder): Decoder(
    (lin1): Linear(in_features=2, out_features=256, bias=True)
    (lin2): Linear(in_features=256, out_features=784, bias=True)
  )
)
```

L'encodeur est constitué d'une première couche linéaire, suivie d'une activation Relu. Ensuite deux couches linéaires indépendantes l'une pour la matrice de variance covariance et l'autre pour la moyenne.

Le décodeur est constitué de deux couches linéaires (qui tel un auto encodeur classique font les opérations inverses de l'encodeur) qui restitue un résultat de même dimension que les images en entrées. Ce résultat est ensuite donné en entrée à une fonction sigmoïde.

#### 9.4 Hyper-paramètres :

- learning rate :  $3e - 4$ .
- batch size : 64.
- dimension de l'espace latent : 2.

#### 9.5 Évolution de la perte durant l'apprentissage :

On affiche la perte ("variational lower bound) tout au long de l'apprentissage :

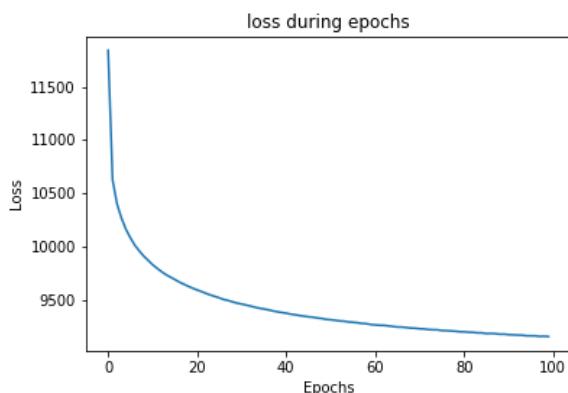


FIGURE 60: Perte ("variational lower bound) durant l'apprentissage.

La perte décroît régulièrement tout au long de l'apprentissage. Puisque nous sommes en train de maximiser une borne inférieure de la vraisemblance, il est difficile de juger de la qualité de l'approximation de  $p_{data}$ . En effet, cette borne peut être "vacuous".

**Reconstruction et échantillonage durant l'apprentissage :** On affiche les images reconstruites (sortie du décodeur) ainsi que le résultat de l'échantillonnage pour un  $z \sim p(z)$  fixé tout au long de l'apprentissage :

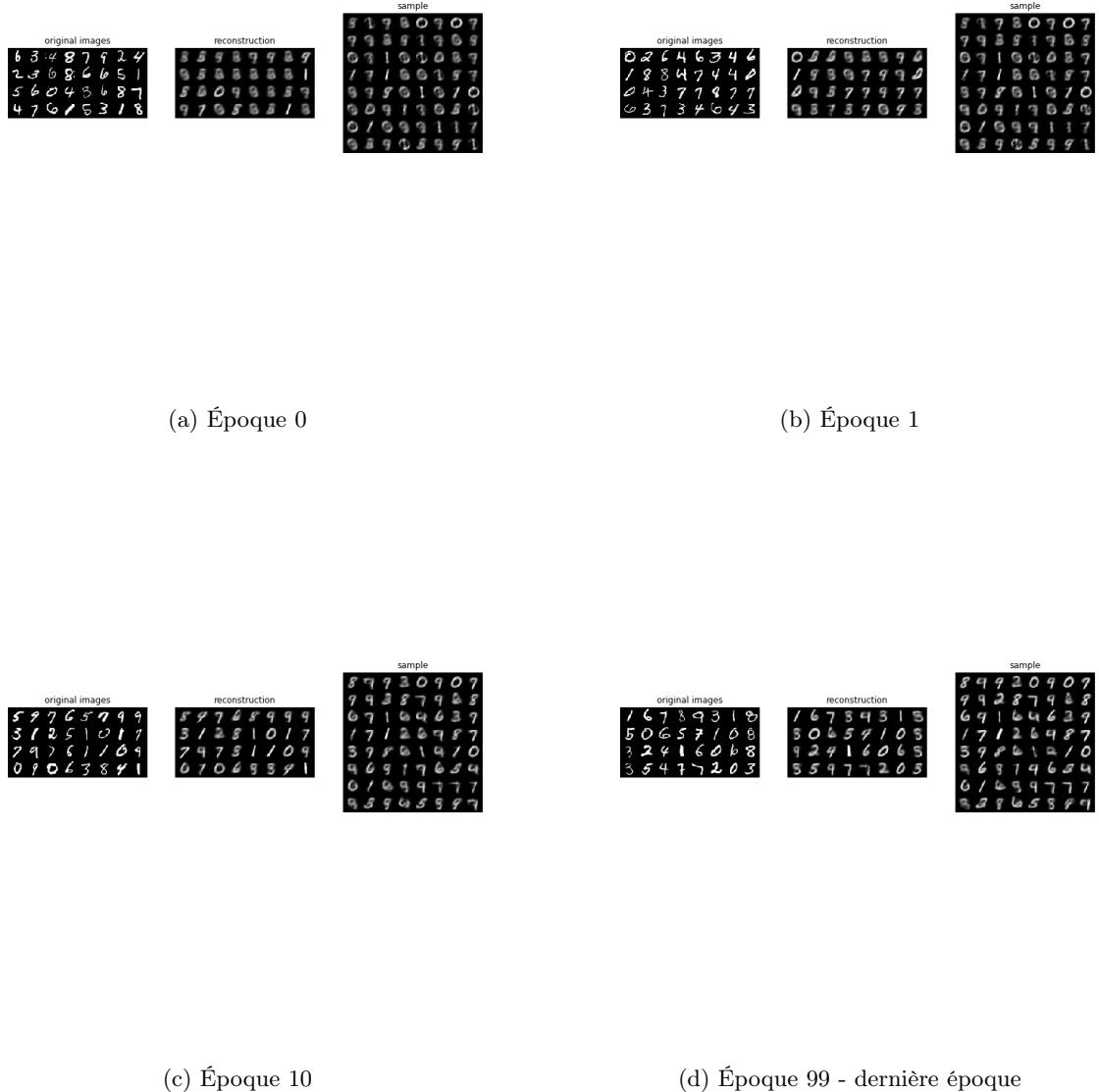


FIGURE 61: Qualité de la reconstruction et de l'échantillonnage en fonction des époques d'apprentissage.

Puisque la loss n'était pas indicative sur la qualité de l'approximation de  $p_{data}$ , on affiche les reconstructions et un résultat d'échantillonnage (pour un  $z \sim p(z)$  fixé) à la fin de chaque époque. On remarque qu'au bout de la première époque, les chiffres générés et reconstruits sont indistinguables. Plus on avance dans l'apprentissage, plus la qualité de la reconstruction et de l'échantillonage s'améliorent, ce qui prouve que maximiser la variational lower bound maximise la vraisemblance des données. Au bout de 10 époques, on distingue la forme des chiffres et à la fin de l'apprentissage, on atteint une qualité de reconstruction et d'échantillonage acceptables.

## 9.6 Effet de la dimension de l'espace latent sur la reconstruction :

Dans cette section on s'intéresse à l'effet de la dimension de l'espace latent sur la qualité de la reconstruction et de l'échantillonage :

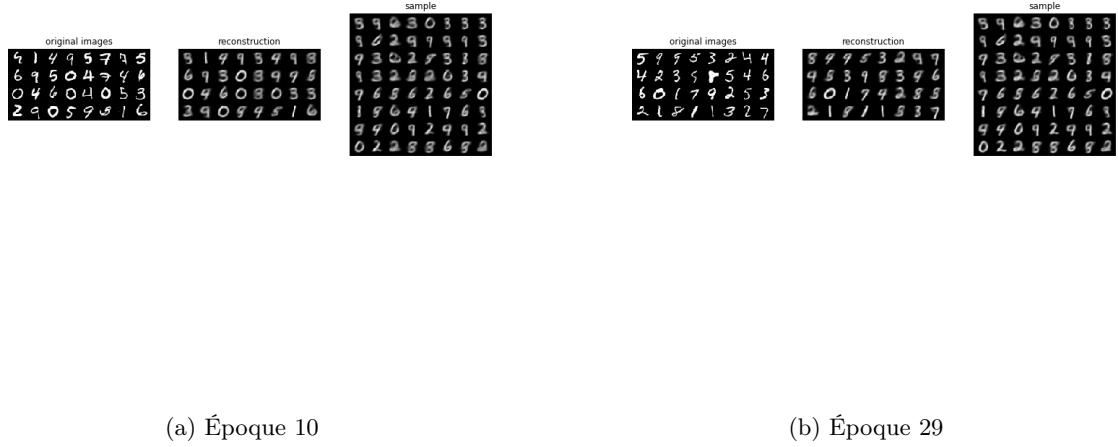


FIGURE 62: Qualité de la reconstruction et de l'échantillonnage avec  $dim_z = 2$ .

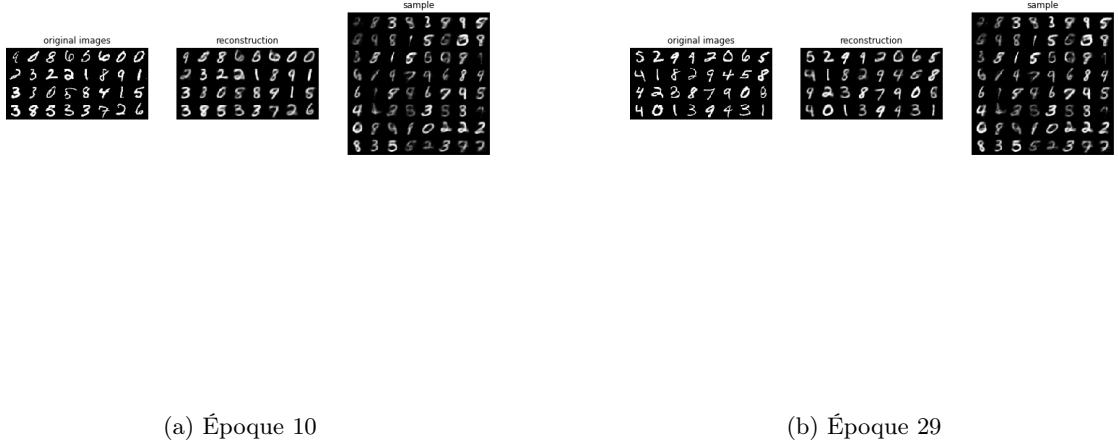


FIGURE 63: Qualité de la reconstruction et de l'échantillonnage avec  $dim_z = 5$ .

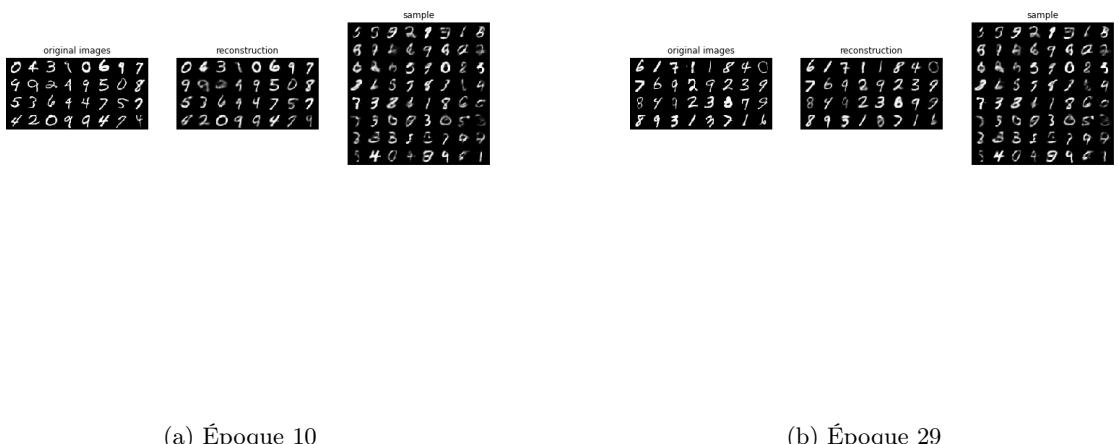


FIGURE 64: Qualité de la reconstruction et de l'échantillonnage avec  $dim_z = 10$ .

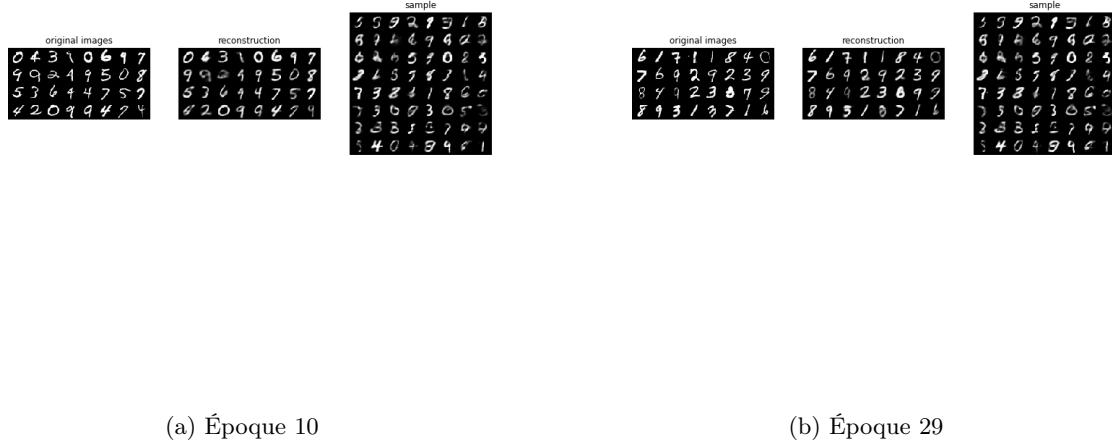


FIGURE 65: Qualité de la reconstruction et de l'échantillonnage avec  $dim_z = 100$ .

- Pour la qualité de la reconstruction, on note une légère amélioration de la reconstruction pour des dimensions de l'espace latent plus grandes (on a complexifié le modèle et donnée plus d'expression à l'espace latent), cependant les résultats restent assez similaire on en conclut que  $dim_z = 2$  est suffisant pour obtenir des résultats satisfaisants.
- Pour ce qui est de l'échantillonage, on note une nette détérioration plus on augmente la dimension de l'espace latent, ceci peut être signe de surapprentissage puisque le terme de régularisation justement  $dkl(q_\phi(z|x), p(z))$ , contraint à ce que la distribution en sortie de l'encodeur doit être proche du prior  $p(z)$ , et pour générer l'échantillon on utilise un bruit blanc samplié de  $p(z)$ .

## 9.7 Analyse de l'espace latent :

Dans cette partie nous nous intéressons à l'interprétation de l'espace latent. Pour une taille de l'espace latent de 2, on échantillonne des variables latentes  $z \sim p(z)$  en faisant varier  $z_1$  et  $z_2$ . Et on observe la reconstruction associée :

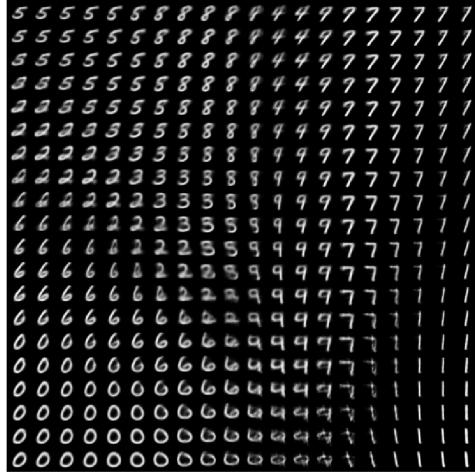


FIGURE 66: Reconstruction en variant les dimensions de la variable latente  $z$ .

Ceci est un résultat de l'hypothèse que le prior  $p(z) = N(0, I_d)$  : les dimensions de  $z = (z_1, z_2)$  sont indépendantes. Il en résulte que ces dimensions porte un certain sens, par exemple dans la partie inférieure, on remarque par exemple que plus on augmente  $z_1$ , plus les chiffres se transforment régulièrement d'un 0 à un 1, et plus on augmente  $z_2$  plus les chiffres se transforment régulièrement de 0 vers un 6. C'est ce qu'on appelle **Distangling Factors of Variation**. De plus, il y a une certaine continuité dans les changements induits par les variations de  $z_1$  et de  $z_2$ , ce qui indique que les dimensions de l'espace latent encodent des propriétés haut niveau des données.

On trace également les embeddings ( $z|x$  sortie de l'encodeur) dans l'espace latent en fonction de leur label ( $\mu$  avec un espace latent de dimension 2).

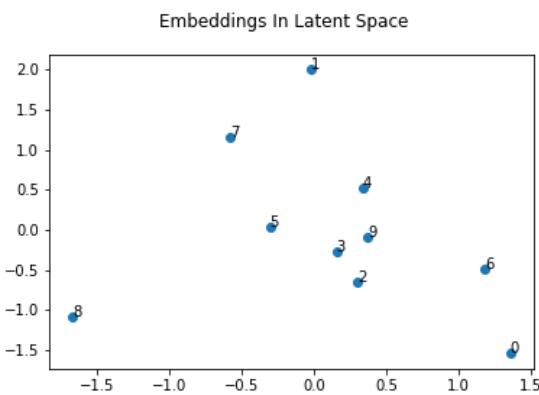


FIGURE 67: Embedding dans l'espace latent des images en fonction de leurs classes.

On remarque par exemple que les chiffres qui ont des formes assez similaires 9, 3, 4, 5 sont proches. Un autre exemple est le 1, qui est généralement facile à classifier et donc dans l'espace latent il est isolé. Il y a donc une notion de distance par rapport à des propriétés haut niveau des données.

Contrairement au GANs, les VAE n'apprennent pas à générer des images de  $p_{data}$  uniquement, ils apprennent aussi à représenter les données à travers les variables latentes  $z|x$ . On a vu aussi qu'en manipulant les variables latentes on peut avoir un effet sur les données générées. On peut imaginer une procédure pour éditer des images.

1. Entraîner le VAE sur les images.
2. Après entraînement, donner une image en entrée à l'encodeur et récupérer une distribution  $q_\phi(z|x)$ .
3. Sampler de la distribution récupérée en 2.
4. Modifier des dimensions du code samplé et le donner au décodeur pour récupérer une distribution  $p_\theta(x|z)$ .
5. Sampler de cette distribution pour récupérer une image éditée.

## 10 TME 11. Multi-Agents RL

### 10.1 Introduction

Dans ce TME, nous explorons les approches d'apprentissage par renforcement profond au domaine multi-agent. Plus précisément, on implémente MADDPG, une adaptation des méthodes actor-critic qui considère les politiques des autres agents et qui est capable d'apprendre des politiques qui requièrent une coordinations complexe entre plusieurs agents.

Nous testons MADDPG sur 3 environnements, certains avec des scénarios coopératifs et d'autres compétitifs.

### 10.2 Cooperative navigation : Simple spread

**Environnement.** Il consiste en 3 agents qui doivent se rapprocher le plus possible d'une cible et ne doivent pas entrer en collisions, c'est un environnement collaboratif. La dimension des actions est de 2.

**Architecture des Q.** Pour chaque critique  $Q$ , on utilise un réseaux à deux couches cachées, la première de 500 et la deuxième de 300. On projette toutes les actions sur 250 et toutes les observations sur 250, on concatène puis on le donne à la première couche cachée.

**Architecture des  $\mu$ .** Deux couches cachés de 500 et 200 neurones.

**Paramètres.** On pose les pas d'apprentissages  $\epsilon_\mu = 0.0001$ ,  $\epsilon_Q = 0.001$ . Le discount est mis à  $\gamma = 0.95$ . Le paramètre du soft update est mis à  $= 0.01$ . Le poids du bruit, est initialisé à 1 et on y applique un decay de 0.000002 à chaque actions.

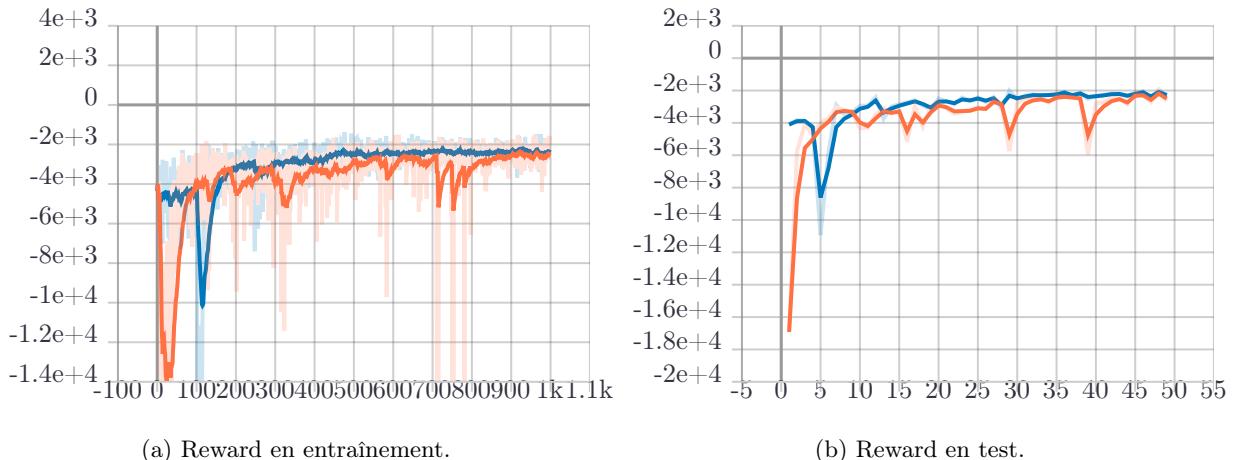


FIGURE 68: Reward par épisode sur Simple Spread. **Orange** : version avec batch de taille 1000 et un buffer de taille 100000. **Bleu** : version avec batch de 100 sur un buffer de taille 10000.

**Analyse.** L'algorithme converge rapidement vers une valeur de -2000 et reste stable une fois arriver là. La stabilité vient du fait que le bruit a de moins en moins de poids plus les épisodes augmentent. Cette valeur n'est pas nécessairement optimale mais bien meilleur que les valeurs aléatoire qui frôlaient les -15000. Un résultat attendu est que la version qui considère beaucoup plus de transitions à chaque fois converge plus vite mais est beaucoup plus instable.

### 10.3 Physical déception : simple adversary

**Environnement.** Il consiste en 3 agents, 2 qui collaborent pour couvrir des cibles et empêcher le troisième d'atteindre une cible. C'est un environnement collaboratif-compétitif. La dimension des actions est toujours de 2. les observations n'ont pas la même dimension pour tous les agents.

**Architecture des Q.** Comme précédemment, pour chaque critique Q, on utilise un réseaux à deux couches cachées, la première de 500 et la deuxième de 300. La différence majeure avec l'expérience précédente est que les dimensions des observations de chaque agent n'est pas la même. Pour simplifier l'implémentation, on rajoute un padding. Cette action n'a pas d'incidence car les mêmes positions sont toujours paddées, donc elle est prise en compte par les agents lors de l'apprentissage. On projette toutes les actions sur 250 et toutes les observations sur 250, on concatène puis on le donne à la première couche cachée.

**Architecture des  $\mu$ .** Deux couches cachés de 500 et 200 neurones.

**Paramètres.** On pose les pas d'apprentissages  $\epsilon_\mu = 0.0001$ ,  $\epsilon_Q = 0.001$ . Le discount est mis à  $\gamma = 0.95$ . Le paramètre du soft update est mis à  $\alpha = 0.01$ . Le poids du bruit, est initialisé à 1 et on y applique un decay de 0.000002 à chaque actions.

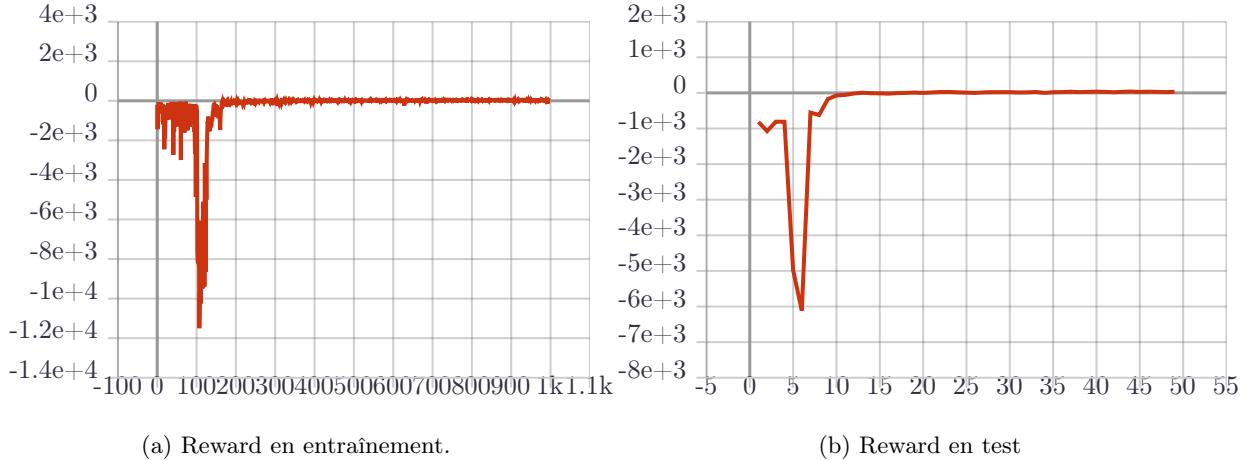


FIGURE 69: Reward par épisode sur Simple Adversary.

**Analyse.** L'algorithme converge rapidement vers une valeur positive de 50. On rappel qu'on est dans un contexte adversarial, cet environnement est donc beaucoup plus instable, le but étant d'arriver à un *équilibre de Nash*. C'est donc un résultat attendu qu'on ne converge pas vers une valeur très élevée.

### 10.4 Predator Prey : Simple Tag

**Environnement.** Il consiste en 3 agents qui collaborent pour attraper une proie. C'est aussi un environnement collaboratif-compétitif. La dimension des actions est toujours de 2. les observations n'ont pas la même dimension pour tous les agents.

**Architecture des Q.** On garde exactement la même architecture que précédemment.

**Architecture des  $\mu$ .** Deux couches cachés de 500 et 200 neurones.

**Paramètres.** On pose les pas d'apprentissage  $\epsilon_\mu = 0.0001$ ,  $\epsilon_Q = 0.001$ . Le discount est mis à  $\gamma = 0.95$ . Le paramètre du soft update est mis à  $= 0.01$ . Le poids du bruit, est initialisé à 1 et on y applique un decay de 0.000002 à chaque actions. On laisse les agents explorer pendant 100 épisodes.

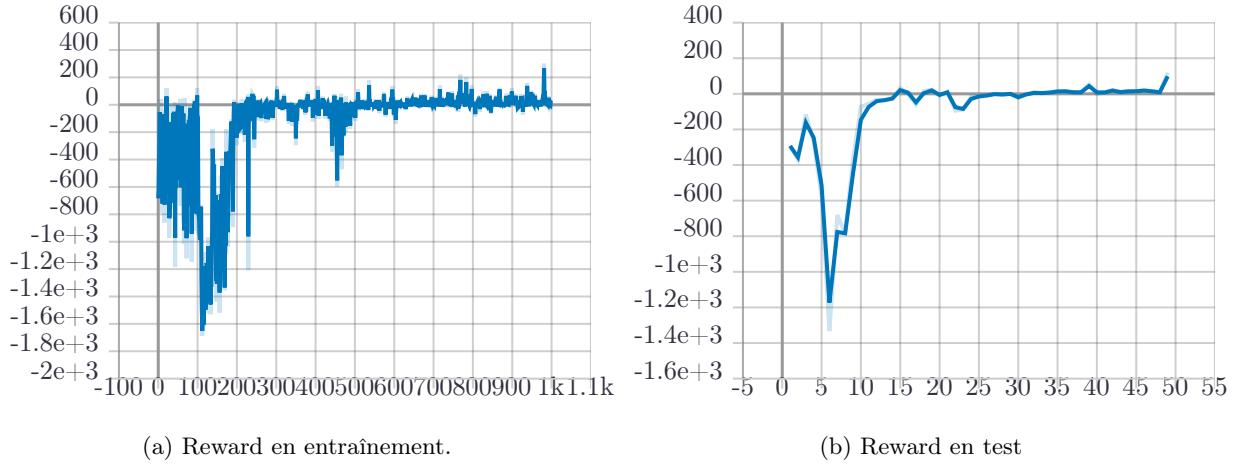


FIGURE 70: Reward par épisode sur Simple Tag

**Analyse.** Pour 1000 épisodes, l'algorithme oscille entre plusieurs valeurs positives, on remarque qu'il y a un pic autour des derniers épisodes de 200. Il est possible que les agents convergent vers une meilleure politique stable avec plus de temps d'entraînement. On remarque que cette politique n'est pas très stable.

## 10.5 Conclusion

Durant ce tme, nous avons exploré la méthode de MADDPG sur trois environnements différents.

# 11 TME 12-13. Imitation Learning

## 11.1 Introduction

Dans certains cas, il est plus facile d'avoir accès à un comportement désiré (trajectoire) plutôt que de spécifier une fonction de reward qui aboutira au final sur un comportement similaire. Les approches utilisées dans ces cas sont appelés "apprentissage par imitation". Où l'on dispose des traces d'un expert (trajectoires) et où notre agent construit sa politique en imitant cet expert. Nous nous intéressons à deux approches d'apprentissage par imitation. Nommément : **Behavioral Cloning** et **Generative Adversarial Imitation Learning**.

## 11.2 Behavioral Cloning :

L'algorithme le plus simple de l'apprentissage par imitation est le *Behavioral Cloning* (clonage de comportement) qui consiste simplement à apprendre la politique de l'expert tel un problème d'apprentissage supervisé de la manière suivante :

1. Récupérer des traces d'experts (des trajectoires  $\tau^*$ ).
2. Générer des paires  $(s_t^*, a_t^*)$  iid de couples état, action.
3. Apprendre la politique  $\pi_\theta$ , en utilisant un algorithme d'apprentissage supervisé pour minimiser une fonction de coût  $L(a^*, \pi_\theta(s))$ .

Dans ce qui va suivre nous allons tester le Behavioral Learning sur Lunar Lander :

- On récupère une trajectoire d'un agent expert PPO sur ce jeu.
- On maximise la log-vraisemblance  $\sum_{(a_i, s_i)} \log(\pi_\theta(a_i | s_i))$ . Critère entropy loss.
- On modélise  $\pi_\theta$  par un réseau de neurones à deux couches avec des non-linéarités  $tanh$  ( $\theta$  étant les paramètres de ce réseau).

On affiche l'estimation du reward cumulé (moyenné sur 100 épisodes du jeu) à chaque 100 pas de gradients (on fixe `batch_size = len(dataset)`, et on prend `lr = 0.003`) :

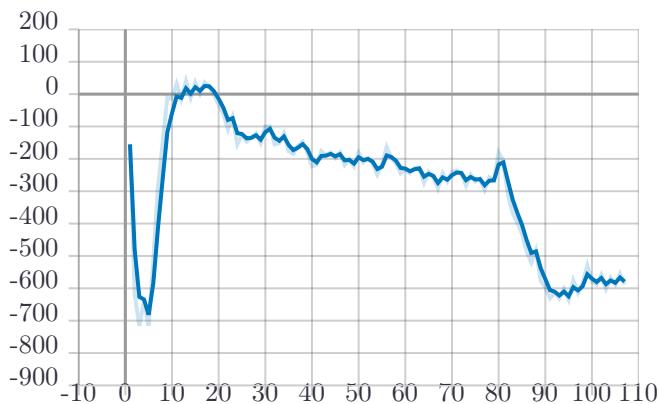


FIGURE 71: Estimation du reward cumulé (moyenné sur 100 épisodes du jeu) à chaque 100 pas de gradients pour Behavioral Cloning.

Les meilleures performances entre 0 et 30 sont atteintes autour de 1000 mise à jour du gradient ( $10 \times 100$ ). Au delà, on fait face à du sur-apprentissage. Comparé aux 250 de l'expert, on peut dire que cet algorithme n'est pas adapté pour ce jeu et pour les données qu'on possède.

Le behavioral cloning souffre de plusieurs problèmes :

- L'hypothèse iid qu'on fait lors de l'apprentissage supervisé n'est pas vérifiée , d'où la nécessité de méthodes pour palier à ce manque.
- L'agent peut se retrouver dans des états que l'expert n'a jamais visité et donc sur lesquels il n'a jamais été entraîné, la réaction de l'agent est donc imprévisible et peut conduire à ce qu'on appelle *catastrophic failure* i.e. se retrouver dans des états très défavorables.

Outre sa simplicité. Le behavioral cloning marche très bien pour certaines applications, comme les applications qui n'ont pas besoin de planification long terme et pour lesquelles les trajectoires de l'expert couvre l'espace et où choisir une mauvaise action n'a pas de conséquences très graves.

### 11.3 Generative Adversarial Imitation Learning :

L'algorithme GAIL consiste à apprendre une politique qui permet d'explorer des zones connues (sur lesquelles on a des données de l'expert), cet algorithme fait intervenir :

- Un discriminateur  $D$ , qui est un réseau de neurones qui à pour but de distinguer les transitions de l'expert des transitions de la politique.
- Une politique  $\pi_\theta$  qu'on apprend à générer des transitions indistinguables de celles de l'expert. On utilise dans ce TME *PPO*.

#### 11.3.1 Expérimentations :

##### Hyper-paramètres :

- learning rate  $1e - 3$  pour les deux réseaux.
- 1 mise à jour du discriminateur et 10 mises à jour de PPO toutes les 64 transitions.
- Un discriminateur à deux couches 64, 32.
- Clipp PPO à  $\epsilon = 0.2$

On trace le reward moyen (100 épisodes de tests) après chaque 1000 épisodes d'entraînement :

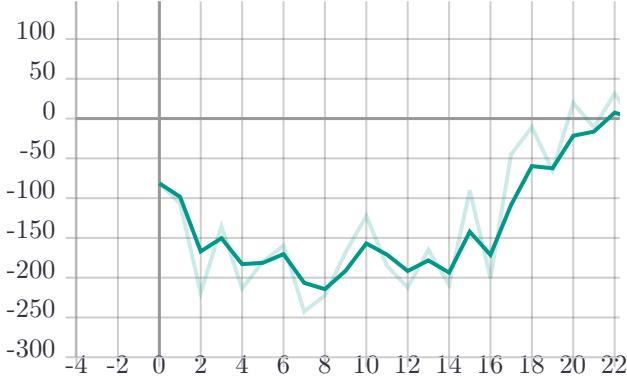


FIGURE 72: Reward moyen (100 épisodes de tests) après chaque 1000 épisodes d'entraînement - GAIL.

On remarque que le reward moyen augmente effectivement et atteint un reward  $> 0$ . Cependant pour des raisons computationnelles nous ne sommes pas parvenu à reproduire les résultats indiqués sur le sujet du TME (en effet les expérimentations ont été conduites pour 60 000 épisodes d'apprentissages).

## 12 TME 14. Curriculum RL

### 12.1 Introduction

Les approches "Implicit Curriculum RL" interviennent lorsqu'on fait face à des tâches relativement complexes où on est en présence de récompenses parcimonieuses : on obtient une récompense non nulle uniquement dans les cas où on atteint l'objectif. Les approches par renforcement classiques ne sont pas adaptées à ce cadre, car il n'y a aucun signal qui guide l'agent vers l'objectif, d'autant plus lorsque l'objectif est difficile à atteindre. Dans ce tme on s'intéresse à deux algorithmes dédiés à ces cas de figure : **DQN avec but** et **HER : Hindsight Experience Replay**.

### 12.2 DQN avec but :

L'adaptation de DQN à ce cadre est assez simple. Soit  $\mathcal{A}, \mathcal{S}, \mathcal{G}$ , respectivement : l'ensemble des actions possibles, l'ensemble des états possibles et l'ensemble de buts dans l'environnement. On considère alors  $Q : \mathcal{A} \times \mathcal{S} \times \mathcal{G}$ , il suffit de considérer  $Q_\phi : \mathcal{A} \times \mathcal{S}'$  où  $\mathcal{S}'$  est un ensemble état-buts dont chaque élément est la concaténation de d'un état de  $\mathcal{S}$  et d'un but de  $\mathcal{G}$ .

#### 12.2.1 Expérimentations :

##### Hyper-paramètres :

- On utilise DQN classique (au lieu de double DQN) avec une couche cachée de 200.
- Paramètre d'exploration 0.2 sans decay.
- On considère des épisodes de taille maximales 100 (on arrête l'épisode si le nombre de transitions dépasse 100).
- Buffer size et update frequency à 1000.
- Learning rate à  $1e - 3$ .
- Carte Plan2Multi : qui contient des buts proches de l'agent.

On affiche la moyenne des rewards sur 100 épisodes de test tous les 1000 épisodes d'apprentissage :

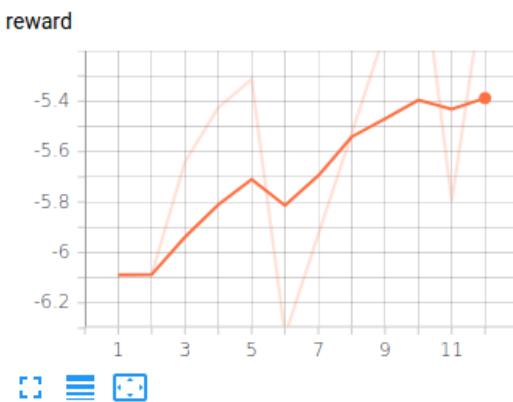


FIGURE 73: Reward moyen tout les 1000 épisodes d'apprentissage.

On remarque que le reward moyen augmente tout au long de l'apprentissage. Avec un reward moyen maximale de  $-5.4$  atteint au bout de 11000 épisodes d'apprentissage, on est en présence d'un agent quasi parfait. Car, puisqu'on limite la taille des épisodes à 100 et que la perte associée aux buts est de 1 et au autres cases est de  $-0.1$ , la perte maximale est de  $-10$ , donc si on fait moins, cela veut dire que l'agent atteint le but en moins de 100 transitions. Cependant, notre version est moins performante que celle donnée en exemple sur le sujet (stabilité, reward maximale), ceci est du à l'utilisation du DQN classique au lieu de Double-DQN.

### 12.3 HER : Hindsight Experience Replay

La carte précédente sur laquelle a été testé DQN avec buts contient des buts relativement proche de l'agent qui le guident vers les buts plus difficiles d'accès. Avec la carte *plan2.txt*, où il y a un unique but très difficile d'accès (case [33, 38]), DQN avec buts est inadapté (reward de  $-10$ ). Pour palier à ce cas, on implémente l'algorithme *HER*, qui rajoute des buts intermédiaires permettant de guider l'agent vers le but de l'environnement. La procédure est la suivante :

- On rajoute dans le buffer toutes transitions de l'épisode avec comme but celui visé par l'épisode (but réel de l'environnement) et le reward associé (1).
- On rajoute aussi à la fin de chaque épisode, toutes les transitions de ce dernier avec comme but le dernier état atteint. On associe à ces transitions un reward de 1 si elles mènent à ce dernier état et  $-0.1$  sinon. Ces buts fictifs vont inciter l'agent à explorer et le guident vers le but réel (ces buts fictifs ne servent qu'à apprendre le réseau de valeurs).

On affiche la moyenne des rewards sur 100 épisodes de test tous les 1000 épisodes d'apprentissages :



FIGURE 74: Reward moyen tout les 1000 épisodes d'apprentissage.

L'agent ne parvient pas au but et stagne à  $-10 = (100 \times -0.1)$ . Pour analyser le comportement de l'agent on affiche aussi l'évolution des coordonnées de la dernière transition atteinte de chaque épisode :

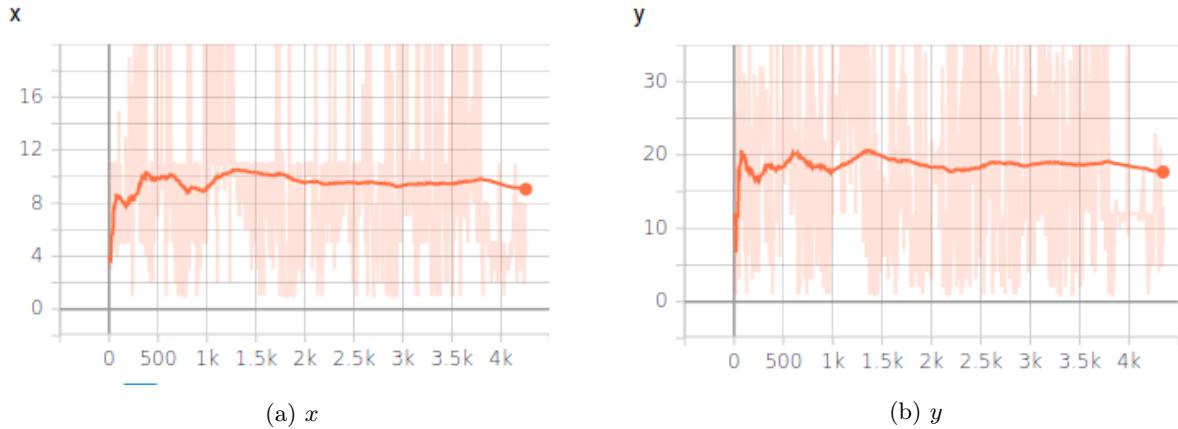


FIGURE 75: Évolution du dernier état atteint par l'agent en fonction des épisodes d'apprentissage.

On remarque que les coordonnées évoluent (les buts intermédiaires l'incitent effectivement à explorer) contrairement au résultat obtenu avec DQN avec but (l'agent stagne autour de [1, 1]), cependant l'agent ne parvient pas à la dernière case et reste bloqué dans un région avoisinante (autour de [11, 20]). Il est possible que la différence des résultats avec ce qui a été exposé en TME soit du à l'utilisation du DQN simple au lieu de double DQN. Il se peut aussi qu'avec l'utilisation de DQN, l'agent nécessite plus d'épisodes d'apprentissage pour atteindre un politique optimale.

# 13 Projet

## 13.1 Introduction

L'objectif du projet est le challenge kaggle pierre-papier-ciseaux. Les règles du jeu sont très simples : deux joueurs s'affrontent et doivent choisir à chaque tour un objet parmi le triplet Pierre, Papier, Ciseaux. Si les deux agents font le même choix, nous obtenons 0 comme reward sinon c'est + ou - 1.

Nous voulons ici adapter les algorithmes d'apprentissage par renforcement vu en cours a ce contexte. Nous définissons un épisode comme étant 1000 jeux entre deux agents et l'épisode est gagné si par un agent remporte 20 parties de plus que l'autre.

**Règle du jeu.** Chaque joueur gagne ou perd un point selon la configuration : le papier gagne sur la pierre, la pierre gagne sur les ciseaux et les ciseaux gagne sur le papier. Le jeu se termine au bout de 1000 tours et le gagnant est celui qui a plus de 20 points d'avance.

## 13.2 Baseline : Random

Pour le jeu de pierre papier ciseaux, nous prenons comme baseline, la stratégie qui consiste a jouer un coup aléatoirement à chaque fois. Ce n'est pas une mauvaise stratégie car tirer une action aléatoire maximise le gain en espérance. Pour visualiser cette baseline, on évalue deux agents aléatoires l'un contre l'autre.

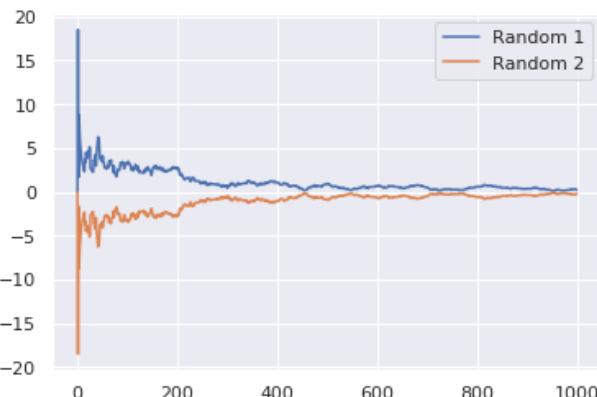


FIGURE 76: Reward cumulé moyen pour deux agents aléatoires par épisode.

**Analyse.** C'est un jeu a somme nulle, donc il est normal que les courbes soient complémentaires.

## 13.3 DQN

**Méthodologie.** La premier algorithme de RL que l'on applique à ce problème est Deep Q network. Nous testons la version de base, sans prioritized experience replay. L'état qu'on donne à chaque itération au réseaux  $Q$  est construit a partir des observations. Nous concaténons le coup joué par l'adversaire au coup précédent, ('lastOpponentAction'), le temps restant ('remainingOverageTime') et le reward précédent.

**Paramètres.** les paramètres de DQN sont comme suit : taille du buffer : 10000,  $\gamma = 0.999$ ,  $\alpha = 0.01$ , taille du batch : 200, fréquence de mise a jour de la cible : 100. L'exploration est faite par  $\epsilon$ -greedy avec weight decay. Le reseau Q a une couche cachée de 100 neurones avec des activations linéaires.

**Entraînement.** Nous l'entraînons pendant 150 épisodes contre l'agent complètement aléatoire.

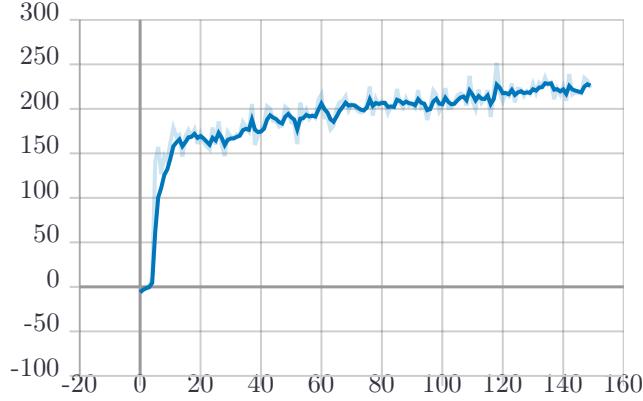


FIGURE 77: Reward moyen par épisode en entraînement pour l'agent DQN contre random

**Analyse.** DQN converge vers moins du quart de tout les points possibles lors d'un épisode. Contre random, on sait que la meilleure stratégie sur le long terme ne peut être qu'aléatoire aussi.

### 13.3.1 DQN vs Random.

Nous évaluons l'agent obtenu contre la stratégie random pendant 100 épisodes.

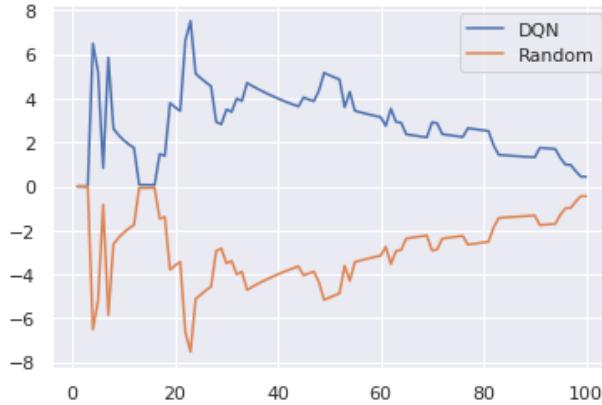


FIGURE 78: Reward cumulé moyen pour l'agent DQN contre l'agent aléatoire.

**Analyse.** Au début, DQN semble remporter plus de parties mais l'agent random reprend du terrain pour converger vers une valeur égale. C'est un comportement attendu car DQN prend l'action avec la plus grande valeur Q pour un état. Si on retombe sur le même état, on choisit exactement la même action alors que le random n'a pas cette contrainte ce qui fait que l'effet des premiers rewards positifs s'estompent comme on peut le voir sur la courbe. De plus, la version qu'on utilise de DQN sélectionne les actions selon  $\epsilon$ -greedy avec weight decay. Une fois convergé, la valeur de  $\epsilon$  devient beaucoup plus petite et donc l'agent explorera moins, ce qui le rend vulnérable face à autre agent aléatoire.

### 13.3.2 DQN vs Statistical.

Nous l'évaluons ici contre l'agent "statistical".

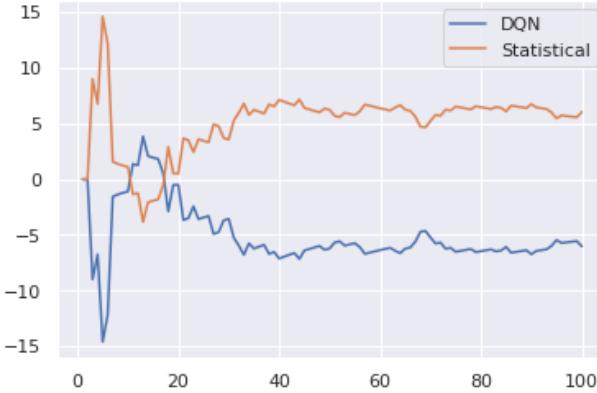


FIGURE 79: Reward cumulé moyen pour l'agent DQN contre l'agent statistique.

**Analyse.** Cette fois ci, l'agent DQN est complètement inadapté, C'est bien sur attendu, il est entrainé sur un joueur qui joue de façon complètement aléatoire et n'est pas adapté au joueur statistique.

### 13.3.3 DQN vs Reactionnary.

Nous l'évaluons également contre l'agent "reactionnary"

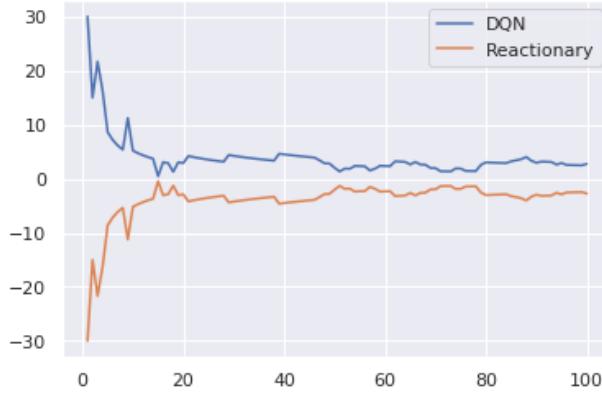


FIGURE 80: Reward cumulé moyen pour l'agent DQN contre l'agent réactionnaire.

**Analyse.** Cette fois ci, l'agent DQN conserve un avantage même sur le long terme sur l'agent réactionnaire. Ceci s'explique par le fait que DQN a été entraîné face a un agent aléatoire, il est donc assez imprévisible pour que l'agent réactionnaire ne puisse pas prédire ses coups.

## 13.4 Conclusion

Nous avons testé un algorithme d'apprentissage par renforcement appliqué au problème de pierre, papier, ciseaux et nous l'avons évalué contre 3 baselines avec des résultats cohérents.