TME4 RLD

Deep Q-Network

Brient, Hein - Décembre 2021

1 Objectifs

Ce TP se focalise sur l'implémentation et l'étude de l'algorithme Deep Q-Network (DQN, Volodymyr Mnih et al.) ainsi que sur plusieurs techniques développées pour améliorer ses performances. Nous avons testé ces algorithmes sur trois environnement classiques : Gridworld, Cartpole et LunarLander et avons étudié et comparé leurs performances.

2 Algorithmes : Deep Q-Network et dérivées

La représentation tabulaire de l'environnement utilisée par l'algorithme Q-learning dans le TME précédent s'appuie sur des estimations individuelles des valeurs Q pour chaque état-action. Cette méthode de représentation présente rapidement des limites, notamment lorsque l'espace des états-actions devient très grand, voire infini. Le but est alors d'apporter à l'agent la capacité de généraliser les états rencontrés selon leur ressemblances.

Une méthode efficace est d'implémenter une fonction d'approximation de la valeur Q, plutôt que l'implémentation d'une table discrète. On va maintenant considérer que chaque état s est défini par un certain nombre de paramètres limité, stockés sous forme de vecteurs et notés $\phi(s)$. Désormais, la fonction $Q(s, a)$ qui associait une valeur à chaque couple état action (s, a) de façon discrète sera approximée par la fonction $Q_{\theta,a}(\phi(s))$ où θ est un ensemble de paramètres associé à la fonction Q. Contrairement au cas précédent, cette nouvelle fonction renvoit à l'aide d'un seul état s donné en entré, l'ensemble des approximation des valeurs $Q(s, a)$ associés à chaque action possible depuis l'état s. L'utilisateur devra alors stipuler quelle action choisir en fonction de ces différentes valeurs, habituellement l'action associée à la valeur la plus élevée. Cette méthode, qui certes peut induire une perte d'information dans la représentation des états, permet de ne stocker que les paramètres θ plutôt que l'ensemble des valeurs $Q(s, a)$, rendant l'algorithme utilisable dans de grands espaces.

Dans le cas linéaire, que nous utiliserons par la suite, la différence temporelle est alors exprimée de la façon suivante :

$$\delta_t = \gamma \times max_{a \in \mathcal{A}(s_{t+1})}(< \theta_a, \phi(s_{t+1}) >) + r_t - < \theta_{a_t}, \phi(s_t) >$$

L'algorithme DQN propose alors d'utiliser un réseau de neurones comme fonction d'approximation $Q_{\theta,a}(\phi(s))$. Le réseau va pouvoir adapter les poids θ en fonction des récompenses obtenues par l'agent, permettant de généraliser les états rencontrés.

2.1 Target Network

En l'état, l'algorithme peut présenter des difficultés à converger à cause d'une valeur cible non stationnaire lors de l'étape d'optimisation. En apprentissage profond, cette valeur est habituellement fixe, rendant l'entraînement stable. Ici, cette valeur sera modifiée à chaque pas d'optimisation puisque l'agent découvre l'environnement au fur et à mesure de ses expériences.

Une solution à ce problème est d'implémenter un deuxième réseau de neurones : un "target network". Dans l'algorithme de base, le même réseau de neurones est utilisé pour approximer la valeur de prédiction et la valeur cible. Le target network est identique au réseau dit courant mais n'est utilisé que pour estimer la valeur cible. Les paramètres de ce réseau sont gelés pendant un certain nombre d'itérations, défini au préalable. Ceci permet à la valeur cible d'être fixe durant chaque phase d'apprentissage prédéfinie par ce nombre d'itérations, facilitant grandement la convergence de la valeur de prédiction vers la valeur cible. Enfin, pour transmettre au réseau cible les nouvelles informations obtenues par l'algorithme durant chaque phase, les paramètres du réseau courant (qui sont modifiés à chaque pas d'optimisation) sont après chacune de ces phases d'apprentissage, copiés dans le réseau cible afin de le mettre à jour.

2.2 Prioritized Replay

Un autre problème présent dans l'algorithme de base est la forte corrélation entre les échantillons d'une même trajectoire. Pour le contrer, un replay buffer d'une taille n prédéfinie peut être mis en place pour enregistrer les N dernières transitions (state, action, reward, next state, done) rencontrées par l'agent. Cette technique a plusieurs bénéfices : elle réduit les corrélations entre exemples d'apprentissage, permet d'accélerer l'apprentissage par optimisation sur mini-batchs de transitions plutôt que sur une seule, et enfin permet aussi de garder en mémoire des transitions passées pour éviter l'oubli d'informations importantes. En pratique, à chaque phase d'apprentissage, l'agent va choisir aléatoirement un batch de transitions de taille n parmi les N transitions du buffer. Le terme couramment utilisé pour définir cette technique est : "Replay".

Il y a alors plusieurs méthodes pour sélectionner le batch de transitions dans le buffer. Le cas le plus basique est le choix aléatoire uniforme, qui peut être amélioré en accordant plus de poids aux transitions plus importantes pour l'apprentissage en l'état actuel (Prioritized Replay). En effet, puisque les exemples sont plus ou moins "loins" de leur valeur cible, il va de soit d'accorder plus d'importance aux transitions mal approximées par le réseau. Ainsi, plutôt que d'échantilloner les exemples uniformément, on peut les sélectionner en fonction de leur écart δ_t à leur valeur cible. La transition i est alors samplée selon la probabilité :

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \qquad avec \qquad pi = |\delta_i| + \epsilon$$

Toutefois, un biais d'évaluation des valeurs Q est introduit avec cette méthode car l'échantillonage n'est pas réalisé avec une probabilité uniforme, ce qui diffère du cas réel rencontré par l'agent. Pour débiaiser le modèle, on peut pondérer le gradient de chaque exemple i par

$$w_i = (\frac{1}{N} \times \frac{1}{P(i)})^\beta$$

Dans ces trois relations, α, β et ϵ sont des hyper-paramètres à fixer. Aussi, afin d'échantillonner efficacement dans un replay buffer de grande taille, des structures de type $SumTree$ sont souvent mises en place.

2.3 Double DQN

L'algorithme de Q-learning tend à sur-estimer la valeur Q cible. Pour résoudre ce problème, l'idée est de décorreler l'erreur d'estimation de la valeur de celle du choix de la valeur maximale. En effet cette corrélation peut être observée dans la relation suivante :

$$max_{a \in \mathcal{A}(s)}(Q_\theta(\phi(s), a)) = Q_\theta(\phi(s), argmax_{a \in \mathcal{A}(s)}(Q_\theta(\phi(s, a))))$$

La décorrélation est effectuée en utilisant le réseau de neurones courant θ pour le choix de la prochaine action et le réseau cible θ^-pour estimer sa valeur. La différence temporelle est alors exprimée de la manière suivante :

$$\delta_t = r_t + \gamma Q_{\theta-}(\phi(s_{t+1}), argmax_{a \in \mathcal{A}(s_{t+1})} Q_\theta(\phi(s_{t+1}, a)) - Q_\theta(\phi(s_t, a_t))$$

2.4 Dueling DQN

Le dueling DQN consiste à adapter l'architecture du réseau à partir de la définition de Q suivante :

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

Q est alors une combinaison d'une fonction de valeur V et une fonction d'avantage A pour chaque action et le réseau va prédire séparemment ces deux éléments puis les ajouter. Cette méthode permet de faire des mises à jour avec prise en compte des valeurs relatives des différentes actions. Intuitivement, dans certaines situations, le choix d'action de l'agent n'a que très peu, voire aucun impact sur les récompenses. Cette architecture permet alors de mieux appréhender le cas où l'agent doit choisir entre des actions de valeurs semblables.

2.5 Noisy DQN

Utiliser une méthode epsilon-greedy pour assurer l'exploration de l'environnement présente certaines limites lorsque les premières récompenses ne peuvent être obtenues

qu'après un grand nombre d'actions. A la place, il est possible d'ajouter un bruit $\epsilon \sim \mathcal{N}(0, 1)$ à chaque paramètre de la couche de sortie. Cela permet d'améliorer grandement l'exploration en la rendant adaptative selon les états et réduit le risque de surapprentissage.

3 Implémentation et Résultats

3.1 Méthodologie

Nous avons implémenté l'algorithme DQN ainsi que l'ensemble des améliorations présentées ci-dessus à partir du code donné. Après de nombreux tests avec différents jeux de paramètres, nous nous sommes rendus compte que les agents étaient très sensibles aux hyper-paramètres, d'autant plus que le côté aléatoire de l'exploration demande à tester chaque agent, pour chaque environnement, plusieurs fois. Ainsi le nombre important d'hyperparamètres à configurer et la longueur des temps de calcul rendraient l'exploration de l'espace des hyperparamètres extrêmement difficile et chronophage sans automatiser le processus et limiter l'étude. La méthodologie suivie a alors été d'implémenter une étape d'optimisation automatique de certains hyperparamètres pour chaque expérience à l'aide de la bibliothèque optuna. Le but étant d'essayer de comparer la performance et la stabilité des différents algorithmes, cette phase d'optimisation peut aussi servir d'évaluation plus juste qu'un paramétrage à la main, dépendant d'un facteur chance plus important. L'étude suivante a été en grande partie réalisée sur l'environnement cartpole, et les meilleurs résultats ont été testé sur les deux autres environnements disponibles à titre indicatif.

Pour faciliter la comparaison des différents modèles et mieux appréhender l'influence de certains paramètres, plusieurs éléments ont été fixés durant l'ensemble de l'étude. Par exemple, l'architecture du réseau de neurones est composé d'une seule couche cachée de 200 neurones. Les nombres de neurones de la couche d'entrée et de sorties sont respectivements le nombre de caractéristique d'un état de l'environnemet étudié $|\phi(s)|$ et le nombre d'actions possibles à l'état s de l'environnement considéré $|\mathcal{A}(s)|$. Par ailleurs, la fonction d'activation utilisée est la fonction tangente hyperbolique. L'optimiseur utilisé est Adam avec un learning rate fixé à 0.0003. La fonction de perte est SmoothL1loss() adaptée en fonction du replay comme présenté précédemment. Le replay buffer et les paramètres pour le prioritized replay sont ceux déjà implémenté dans le code et ne seront pas modifié dans la suite.

Les premiers essais "à la main" avec de nombreux jeux de paramètres ont pu nous aiguiller sur les performances de plusieurs agents et restreindre le domaine d'étude de l'optimisation, qui peut aussi s'avérer très chronophage et inefficace. Comme le replay a un énorme impact positif sur les performances et le temps de calcul de l'algorithme, nous n'avons étudié que les agents avec replay (buffer de 10000 transitions).

L'optimisation automatique utilisée se base sur une fonction *objective*() ayant pour but de maximiser la récompense moyenne sur un certain nombre d'épisodes fixé à 100. Nous avons décidé d'utiliser comme objectif une valeur moyenne pour éviter les jeux d'hyperparamètres rendant les performances de l'agent trop oscillantes. Pour toute l'étude suivante

et donc pour chaque agent et environnement, on a fixé un espace d'hyperparamètres invariable (décrit dans le paragraphe suivant) pour donner la même base d'étude à chaque optimisation. Aussi, chaque optimisation a pu bénéficier du même nombre de jeux d'hyperparamètres à tester : 50. Nous avons aussi fixé plusieurs limites pour nos optimisations. Le temps maximal d'étude pour un jeu d'hyperparamètres est de 10 min, l'évaluation du jeu s'arrête quoi qu'il arrive au plus tard après 10000 épisodes et l'optimisation peut aussi s'arrêter lorsque les 10 derniers score moyen sur 100 itérations ont été égaux au score maximal, à savoir 320 en phase d'entraînement. Ces trois conditions réunies poussent l'optimisation à trouver un jeu de paramètres permettant à l'agent d'à la fois converger rapidement, de s'assurer de la stabilité de l'algorithme une fois le meilleur score atteint et d'avoir une phase de convergence relativement rapide, évitant ainsi les jeux trop instables et coûteux en temps de calcul. Ces limites permettent aussi de grandement limiter les temps de calculs en fixant des point d'arrêt à partir desquels on considère nos algorithmes suffisamment performants. Ces critères dépendent évidemment de ce que l'on attend de nos programmes.

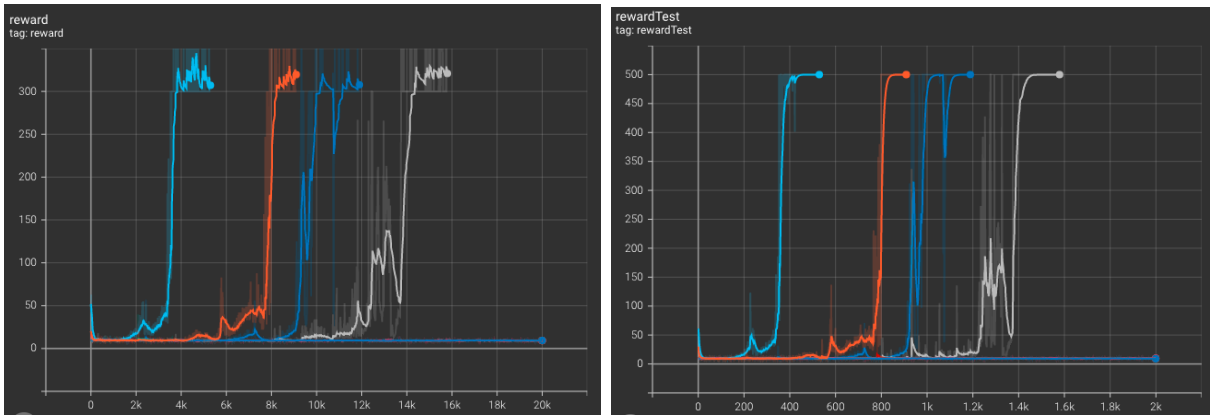
L'espace des hyperparamètres exploré linéairement par l'algorithme est le suivant : Coefficient d'exploration entre 0.001 et 0.2. Discount entre 0.9 et 0.999999. Decay entre 0.9 et 0.999999 Une taille de batch entre 1 et 200 exemples. Une période d'optimisation tous les 5 à 20 évènements. Par ailleurs, la fréquence de transfert des paramètres entre les deux réseaux, lorsqu'elle est utilisée, a été manuellement implémentée à des valeurs relativement faible. En effet, lorsque les fréquences d'optimisation et de transfert étaient trop différentes, l'agent avait beaucoup de difficultés à converger vers de bonnes performances.

Pour un soucis pratique et de temps de calcul, chacun des 50 jeux n'est testé qu'une seule fois lors de l'optimisation, méthode qui ne prend pas vraiment en compte l'aspect aléatoire des agents. Toutefois, la capacité de l'optimisation à trouver un jeu de paramètres efficace traduit d'une certaine manière la stabilité de l'algorithme vis à vis de ces jeux, ce qui est une qualité voulue pour notre agent.

Une fois l'étape d'optimisation terminée, le jeu de paramètres trouvé est testé par l'agent 10 fois sur un total de 10000 épisodes afin de s'assurer ou non de sa capacité à réitérer de bonnes performances. Pour ne pas voir les temps de calcul exploser, on considère que l'agent a convergé lorsque la valeur moyenne des 100 dernière itérations est maximale 10 fois d'affilée, ou 5 fois d'affilée pour certains algorithmes plus demandant.

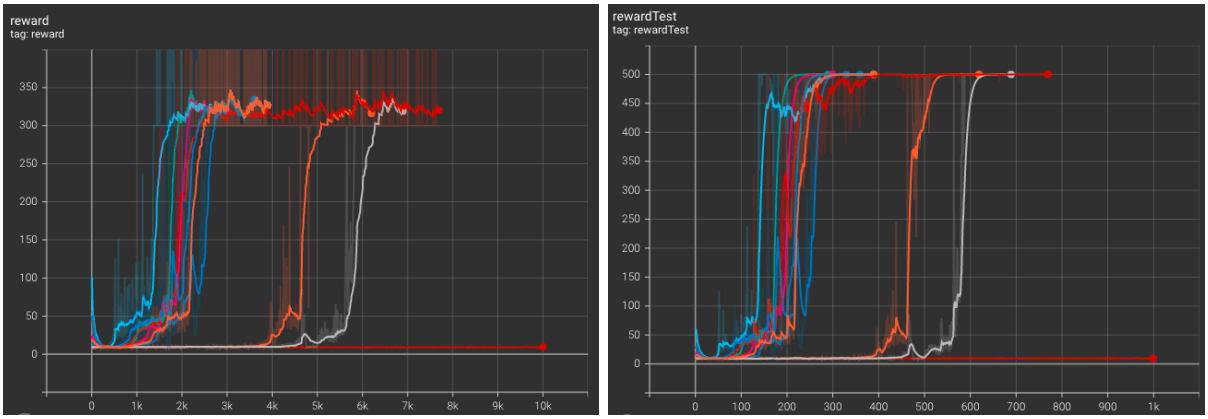
3.2 Résutats

On présente premièrement les courbes résultats sur cartpole pour un DQN avec un target network et un replay uniforme. Les paramètres trouvés par optimisation automatique sont : Coefficient d'exploration de 0,1594. Discount de 0,9912. Decay de 0,9998. Taille de batch de 47. Fréquence d'optimisation de 1/3.



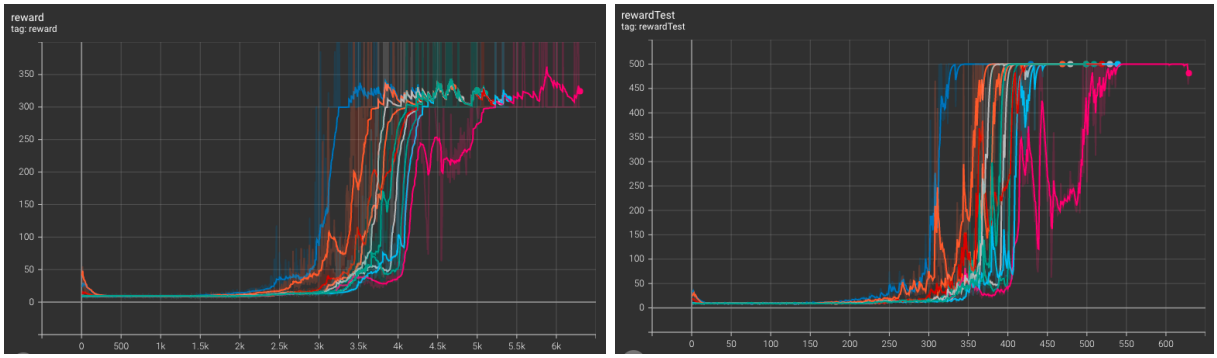
Sur les 10 expériences réalisées, 4 agents ont réussi à converger vers la valeur maximale avant 10000 épisodes. Toutes les phases de convergence sont relativement rapides et les 6 autres agents n'ont pas réussi à apprendre de l'environnement puisque leur score stagne environ à 10. Le comportement des agents est relativement binaire, si l'élément déclencheur de l'apprentissage a lieu, la phase de convergence vers la valeur maximale semble presque inevitable. On peut aussi noter que les agents converge après un nombre d'épisode relativement variable, accentuant ce dernier propos.

On présente ensuite les mêmes résultats avec un prioritized replay où les paramètres suivants ont été utilisés : Coefficient d'exploration de 0.1669. Discount de 0,9985. Decay de 0,9757. Taille de batch de 27. Fréquence d'optimisation de 1/7.



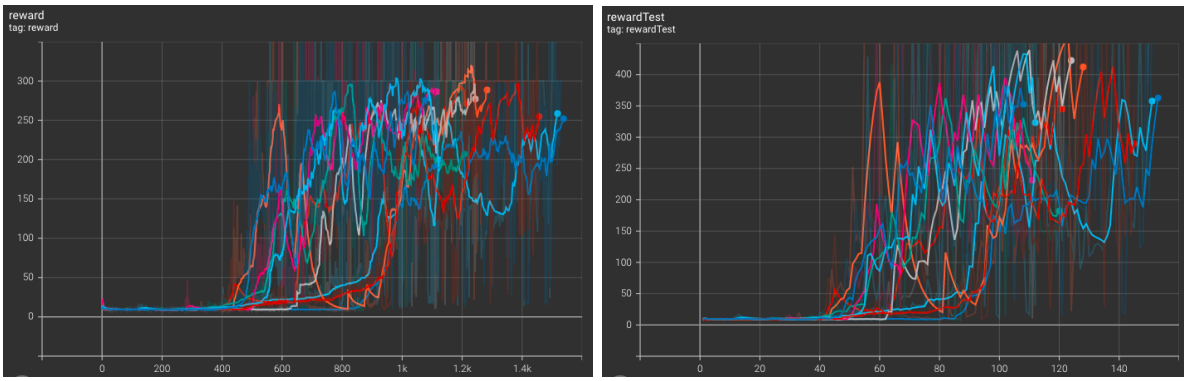
Les performances sont globalement meilleures et plus stables puisque 8 agents sur 10 ont convergé vers la valeur maximale. Toutefois, en contre-partie, les calculs étaient beaucoup plus longs. On remarque que la phase de convergence est aussi beaucoup plus groupée que précédemment.

Pour la suite, on teste l'algorithme DDQN avec le même prioritized replay que dans l'exemple précédent. Les paramètres sont alors : Coefficient d'exploration de 0.0938. Discount de 0,9722. Decay de 0,9381. Taille de batch de 119. Fréquence d'optimisation de 1/7.



Les performances sont encore meilleures et plus stables. Tous les agents ont convergé avant 6000 épisodes et la phase de convergence est très groupée.

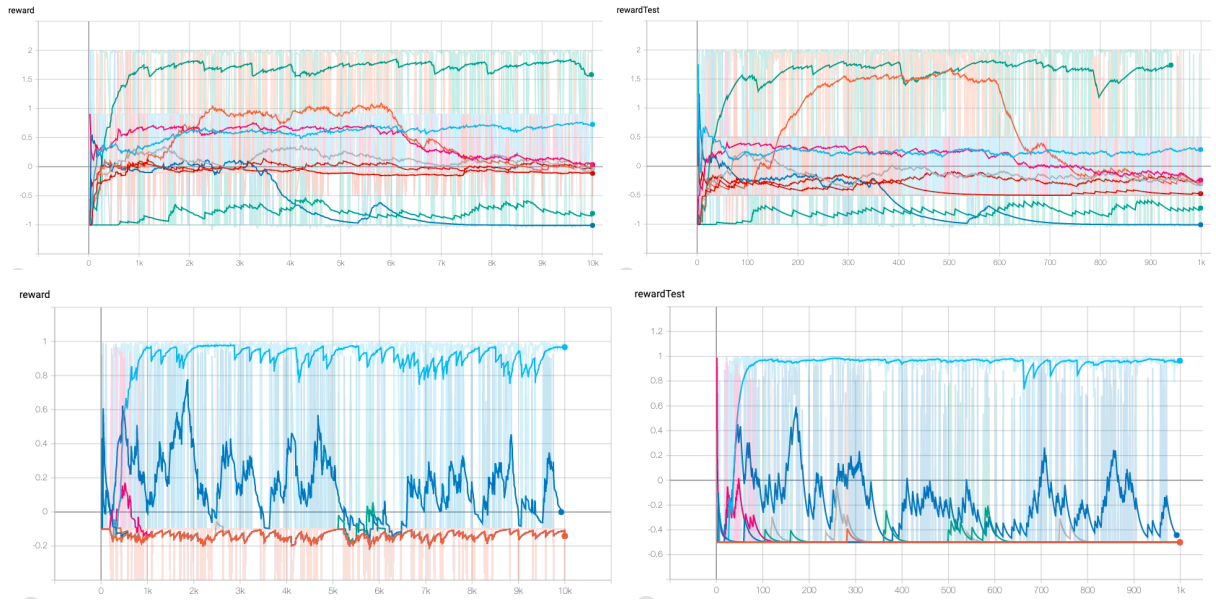
On ajoute ensuite le dueling, pour lequel on a modifié les dernières couches de notre réseau de neurones en implémentant deux double-couches linéaires agissant en parallèle. L'une de taille se sortie 1 pour prédire $V(s)$ et l'autre de taille de sortie $|\mathcal{A}(s)|$ pour prédire $A(s,a)$ pour chaque action possible, comme présenté dans l'article correspondant. Ces deux couches sont finalement croisées selon la formule présentée précédemment pour obtenir les prédictions de $Q(s,a)$. Enfin, on ajoute un bruit aux couches linéaires de notre réseau à l'aide d'une nouvelle classe *NoisyLinear* qui remplace les couches linéaires de notre réseau par des couches linéaires bruitées, dans le but de favoriser l'exploration. Voici les courbes résultat pour un agent Double-Dueling-Noisy-DQN avec prioritized replay sur l'environnement cartpole, avec les mêmes hyperparamètres que dans l'exemple précédent.



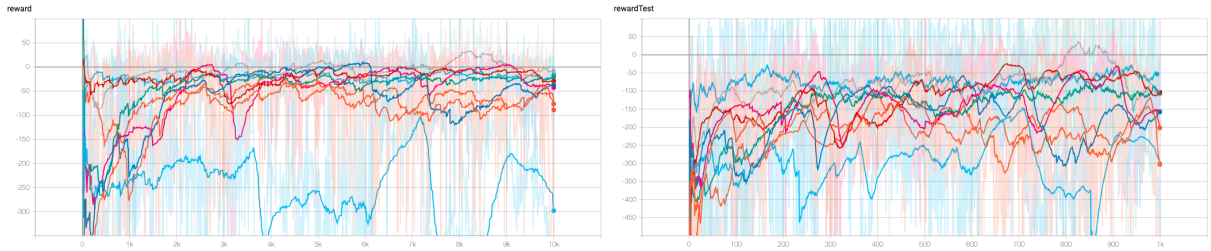
On peut nettement observer les bénéfices de ces deux techniques puisque même avec des hyper-paramètres probablement sous-optimaux, la phase de convergence est entammée pour toutes les expériences avant 1000 épisodes. Avec des paramètres optimaux, on pourrait très probablement obtenir une meilleure convergence comme dans l'exemple précédent.

Ainsi, on a pu observer les différents apports de chaque technique vis-à-vis des performances de l'agent DQN dans l'environnement cartpole. On a aussi vu que les agents étaient extrêmement sensibles aux hyperparamètres et à l'exploration mais qu'après une recherche poussée des paramètres et un agent suffisamment complet, les performances peuvent être très stables, comme en témoigne les deux dernières expériences.

Enfin, nous allons tester avec quelques jeux d'hyperparamètres, les différents agents sur les deux autres environnements. L'étude des hyperparamètres ne sera pas réalisée ici mais la méthodologie est tout autant applicable. Voici les courbes résultats pour une multitude d'agents respectivement sur les plans 1 et 5 de gridworld. Les différents agents sont : DQN, Target, Replay, Target+Replay, Target+Prioritized+Replay, Replay+Double, Replay+Dueling, Replay+Noisy, Replay+Prioritized+Double+Dueling+Noisy. Les paramètres utilisés sont : Coefficient d'exploration de 0.1. Discount de 0,999. Decay de 0,99. Taille de batch de 100. Fréquence d'optimisation de 1/5. Fréquence de transfert de 1/10.



Avec les mêmes paramètres, on a ensuite testé nos différents agents sur l'environnement LunarLander-v2.



A travers ces trois expériences, on a pu à nouveau se rendre compte de l'importance des hyperparamères. En effet, dans la première expérience, l'agent donnant les meilleurs performances est celui le plus fourni en améliorations (prioritized replay, double, dueling, noisy). Cependant pour le plan 5, l'agent DQN simple, bien que ne réussisant pas à obtenir un reward suffisant, est de loin le meilleur. Pour l'environnement Lunar Lander, aucun agent n'a réussi en moins de 10000 épisodes à obtenir des bonnes performances avec ces hyperparamètres. En conclusion, un travail poussé sur la recherche de bons hyperparamètres semble indispensable pour pouvoir tirer partie des différentes améliorations présentées dans ce TP, comme on a pu le montrer sur l'environnement cartpole. En effet, même pour des problèmes relativement simples comme le plan 1 de gridworld, la majeure partie des agents n'ont pas du tout réussi à rapidement converger vers de bonnes performances, alors que leurs performances étaient très bonnes et stables sur cartpole, avec des hyperparamètres adaptés.