RLD : Compte-rendu des TPs

Keyvan Beroukhim et Laura Nguyen

23 décembre 2019

1 Problème de bandits

L'objectif est d'expérimenter les modèles UCB et LinUCB pour de la sélection de publicité en ligne. Nous disposons des taux de clic sur les publicités de 10 annonceurs pour 5000 articles, ainsi que les profils de ces articles (les contextes).

Pour chaque visite, l'objectif est de choisir la publicité d'un des 10 annonceurs permettant d'engrenger le plus fort taux de clics. Il s'agit d'un problème de bandit manchot où les machines sont les annonceurs, les récompenses sont les taux de clics et le but est de maximiser le taux de clics cumulés sur les 5000 visites.

On met en place plusieurs baselines :
- stratégie random : à chaque itération, on choisit n'importe quel annonceur
- stratégie StaticBest : à chaque itération, on choisit l'annonceur avec le meilleur taux de clics cumulés au total
- stratégie optimale : à chaque itération, on choisit l'annonceur ayant le meilleur taux de clics à cette itération.

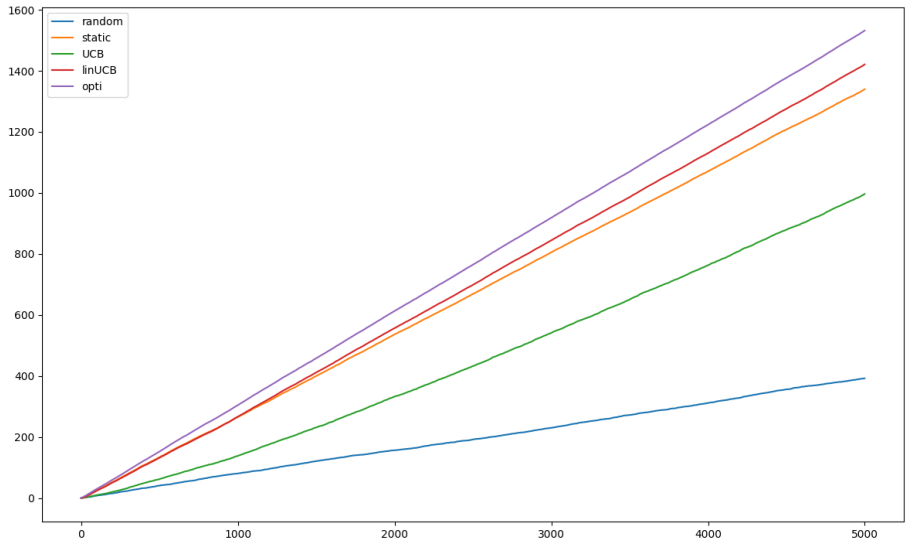
Les deux dernières baselines ne sont normalement pas disponibles étant donné que pour pouvoir les utiliser il faut connaître au préalable les taux de clics de tous les annonceurs à chaque itération. On compare ces baselines avec les stratégies UCB (*Upper-Confidence Bound*) et LinUCB (*Linear Upper-Confidence Bound*).

UCB. La stratégie UCB se base sur le principe de l'*optimisme face à l'incertitude* (Optimism in the Face of Uncertainty) : on choisit d'être optimiste vis-à-vis des options très incertaines, et donc de favoriser les actions pour lesquelles l'estimation du reward associé n'est pas encore assez fiable. On préfère ainsi aller explorer les actions avec un fort potentiel de générer une valeur optimale.

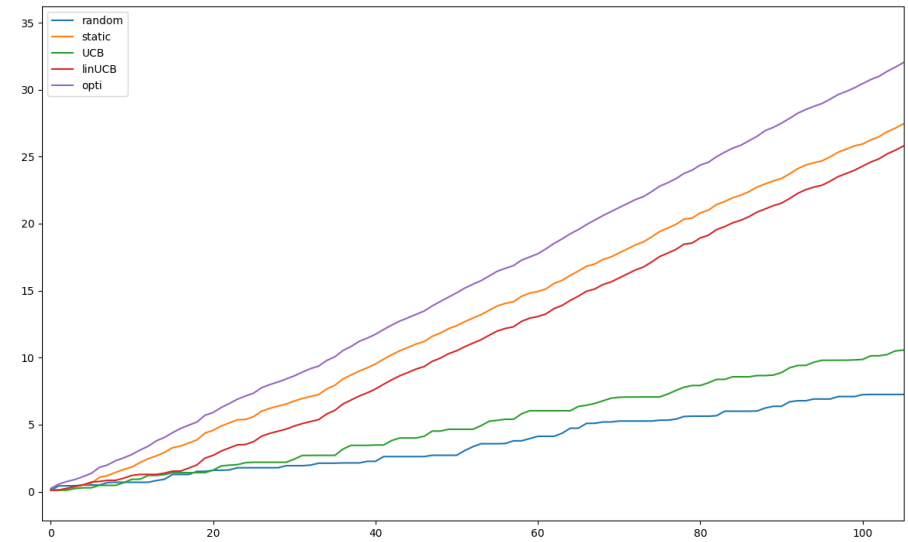
L'algorithme UCB mesure ce potentiel avec une borne de confiance supérieure du reward, qui, sommée à la moyenne empirique du gain associée à une action, fournit une borne supérieure de l'espérance du gain de cette action. On choisit l'action qui maximise cette borne de confiance.

LinUCB. L'algorithme LinUCB propose de prendre en compte le contexte de la décision à chaque instant, ce qui peut être utile pour prévoir les variations des taux observés. L'espérance du reward associé à la décision dépend linéairement du choix de cette dernière.

La figure 1 contient les courbes des rewards cumulés obtenues avec chaque stratégie. On remarque une nette différence entre UCB et LinUCB : prendre en compte les contextes permet d'avoir un cumul de gains bien plus important. De plus, la stratégie LinUCB devient meilleure que StaticBest à partir d'environ 1000 visites.



(a) Cumul sur 5000 visites



(b) Cumul sur les 100 premières visites

FIGURE 1 – Courbes des gains cumulés obtenus avec chaque stratégie

2 Programmation Dynamique : Offline Planning

Certains problèmes peuvent être représentés par des 'Markov Decision Process' ou **MDP**. Un MDP est composé de 4 éléments :

— un ensemble S d'états
— un ensemble A d'actions
— une distribution de probabilité P(s' | s, a) sur les états d'arrivée étant donné un état initial et une action effectuée.
— une espérance de gain R(s, a, s') pour chaque transitions possibles

Quand le MDP est connu, nous pouvons déterminer la politique optimale (c.à.d maximisant l'espérance de gain) sans interagir avec l'environnement, nous sommes dans un cas *d'offline planning*. Nous testons ici deux algorithmes convergeant vers la politique optimale.

Value Iteration. Cet algorithme consiste à mettre à jour de manière itérative la valeur des états.

Policy Iteration. Cet algorithme consiste à alterner entre le calcul de la valeur des sommets selon une politique π et la mise à jour de la politique π.

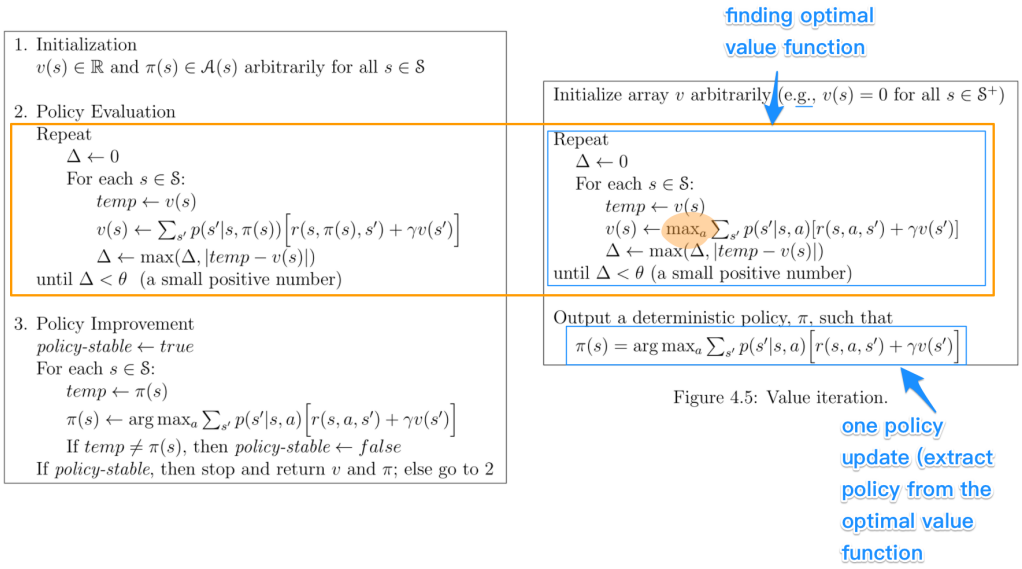


Figure 4.5: Value iteration.

FIGURE 2 – Algorithmes Policy et Value Iteration

Nous expérimentons ces modèles d'algorithmes de programmation dynamique sur `gridworld`, un MDP classique. Il s'agit d'une tâche où un agent (point bleu) doit récolter des éléments jaunes dans un labyrinthe 2D et terminer sur une case verte, tout en évitant les cases roses (non terminales) et rouges (terminales).

Nous évaluons les performances, en terme de moyenne de reward cumulé par épisode, d'une stratégie aléatoire et des algorithmes Policy Iteration et Value Iteration sur trois plans différents de `gridworld` (figure 3).

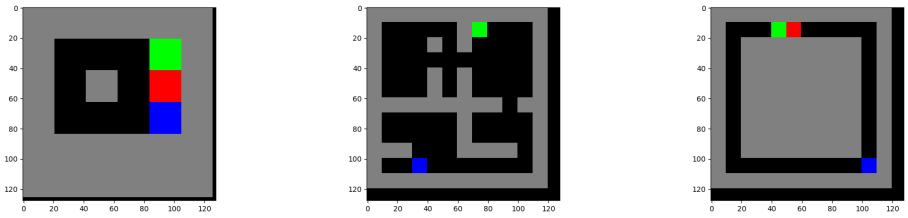


FIGURE 3 – Plans numéro 0, 5 et 10 de `gridworld`

	Random Strategy	Policy Iteration	Value Iteration
Plan 0	-0.789 ± 0.620	0.981 ± 0.011	0.981 ± 0.011
Plan 5	0.319 ± 0.856	1.943 ± 0.005	1.943 ± 0.006
Plan 10	-1.014 ± 0.243	0.958 ± 0.004	0.958 ± 0.004

TABLEAU 1 – Moyenne du gain cumulé par épisode obtenu sur plusieurs cartes selon trois algorithmes différents

Les scores de l'agent Random permettent d'évaluer la difficulté du problème et le gain de performance obtenu avec Policy Iteration et Value Iteration. Les deux algorithmes de plannification obtiennent les mêmes scores car ils ont bien convergé vers la politique optimale.

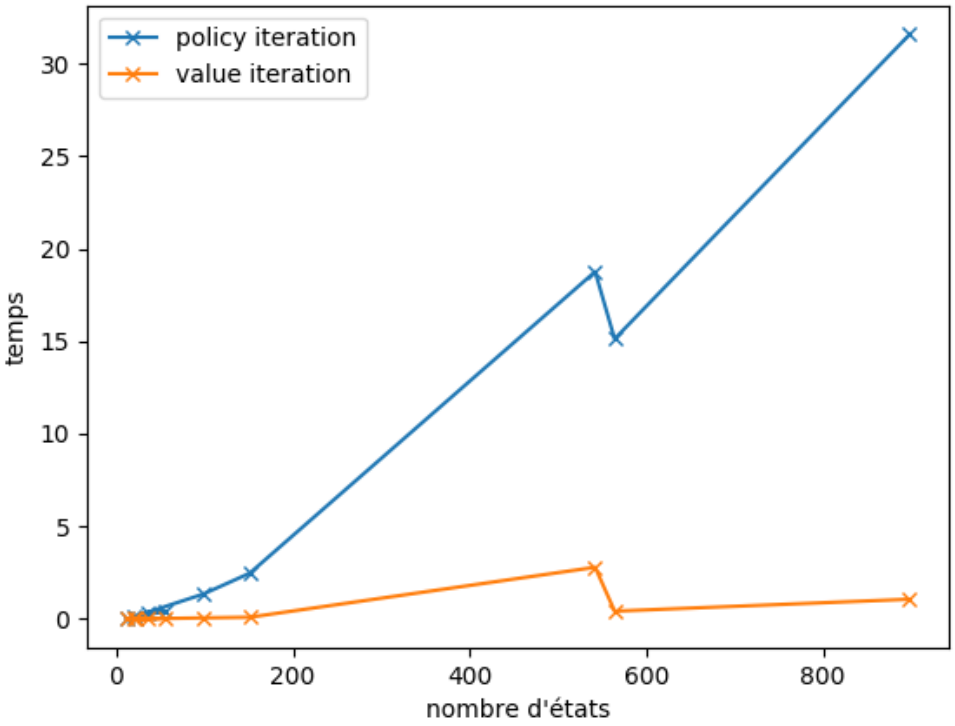


FIGURE 4 – L'algorithme Value Iteration converge plus rapidement que Policy Iteration vers la politique optimale.

3 Q-Learning : value-based models

Quand le MDP n'est pas connu, un agent a besoin d'interagir avec l'environnement afin de déterminer la politique optimale : c'est du reinforcement learning.

Les algorithmes *value-based*, visent à apprendre une fonction Q(s, a) représentant l'intérêt d'effectuer l'action a à partir de l'état s. Une fois ces valeurs apprises, la politique consiste simplement à choisir pour chaque état l'action de plus haute valeur. Quand les états et les actions sont discrets, nous pouvons utiliser une version tabulaire de Q. Les détails des mises à jour sont présentés sur la figure 5.

- choisir l'action à émettre a_t et l'émettre
- observer r_t et s_{t+1}
- $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma\, max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$

(a) Q-Learning

- émettre a_t
- observer r_t et s_{t+1}
- choisir l'action a_{t+1} en fonction de la politique d'exploration
- $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma\, Q(s_{t+1}, a') - Q(s_t, a_t)]$

(b) Sarsa

FIGURE 5 – Algorithmes value-based

Dyna-Q est un algorithme hybride, il est model-based car il apprend les valeurs de P et de R et il est value-based car il apprend aussi les valeurs de Q.

Nous expérimentons ces trois approches de renforcement value-based sur le problème du `gridworld`. La figure 6 contient les courbes d'apprentissage sur 1000 épisodes, les mesures sont relevées tous les 10 épisodes.

Dans l'exemple ci-dessus, l'agent QLearning n'a pas une politique optimale mais reçoit néanmoins un reward optimal, cela s'explique par le fait que les mauvais choix de politique sont faits sur des états dans lesquels l'agent ne tombe pas.

Les gains cumulés représentent la performance globalement réalisée par un agent jusqu'à ce point.

La "policy loss" loss est obtenue en calculant la valeur de chaque état selon la politique de l'agent (on utilise la fonction de l'algorithme Policy Iteration) et nous comparons ces valeurs aux valeurs optimales (obtenues en entraînant un agent Policy Iteration par exemple).

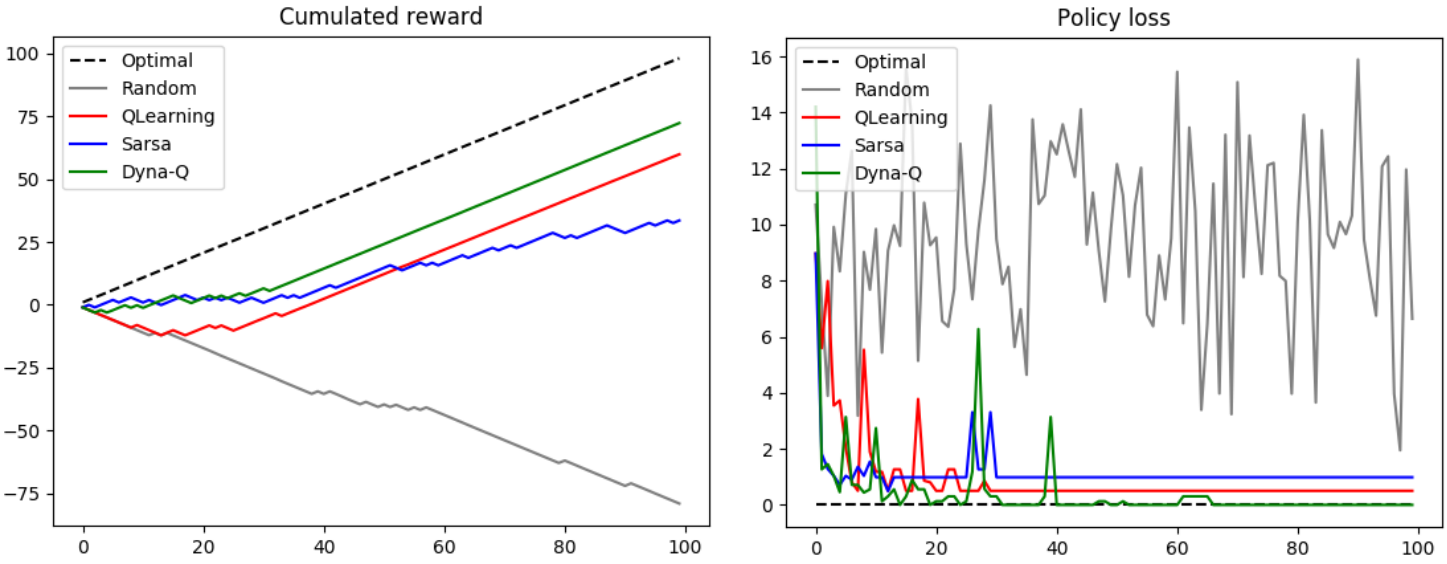


FIGURE 6 – Comparaison de performances des agents

Les performances obtenues par les agents relèvent de l'aléatoire mais la tendance générale obtenue est que l'algorithme Dyna-Q converge le plus rapidement vers la politique optimale, QLearning est un peu plus lent et l'algorithme Sarsa n'y arrive généralement pas.

Dans l'exemple ci-dessus, l'agent QLearning n'a pas une politique optimale mais reçoit néanmoins un reward optimal, cela s'explique par le fait que les mauvais choix de politique sont faits sur des états dans lesquels l'agent ne tombe pas.

Les hyper-paramètres utilisés sont :

— Le paramètre ϵ pour l'exploration ϵ-greedy que l'on initialise à 1 et fait décroître exponentiellement (d'un facteur 0.999).

— Le learning-rate des agents que l'on initialise à 0.5 et fait décroître exponentiellement (d'un facteur 0.9995) afin que les politques convergent .

4 Deep Q-Learning : problèmes à états continus

Quand les états sont continus, on ne peut pas apprendre de matrice Q. On peut alors se servir d'un réseau de neurones pour représenter la fonction Q. Le réseau prend en entrée un vecteur représentant l'état et retourne la valeur de chaque action.

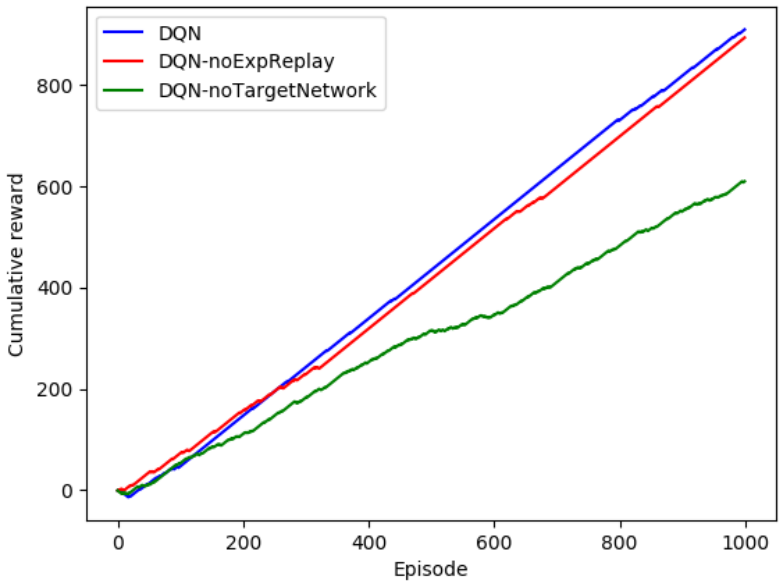
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[\ r_t + \gamma\, max_{a' \in A(s_{t+1})} Q(s_{t+1}, a') - Q(s_t, a_t)] \quad \text{(Q-Learning)}$$
$$\theta \quad \leftarrow \quad \theta \ - \ \alpha\, \nabla_\theta\ [(r_t + \gamma\, max_{a' \in A(s_{t+1})} Q(s_{t+1}, a') - Q(s_t, a_t))^2] \quad \text{(Deep Q-Learning)}$$

FIGURE 7 – Comparaison entre les algorithmes Q-Learning et DQN

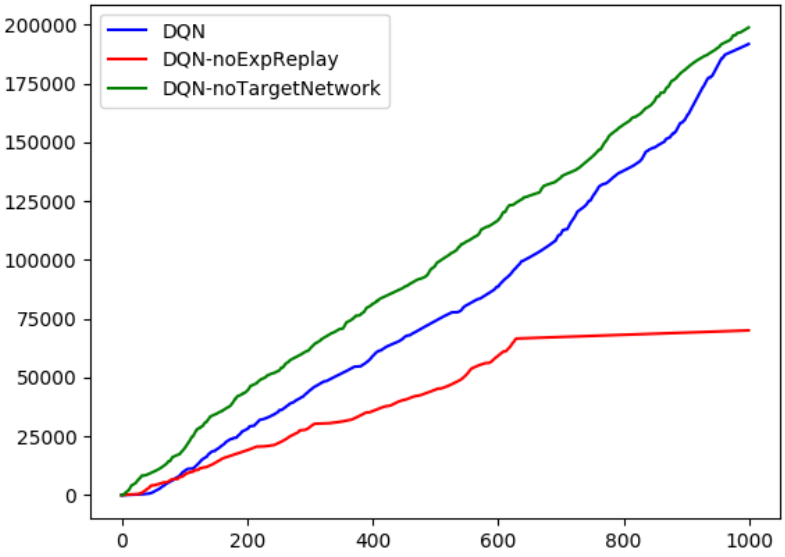
En utilisant directement s_{t+1} pour mettre à jour les paramètres du réseau Q, on ne prend pas en compte les corrélations entre échantillons d'une même trajectoire. L'utilisation d'un buffer d'experience replay, qui stocke les transitions et permet d'échantillonner des mini-batchs pour l'apprentissage, réduit la corrélation entre les exemples, permet de ne pas oublier d'informations importantes et de paralléliser l'entraînement.

Avec l'algorithme de Q-Learning, on modifie une case de la matrice Q à chaque étape, alors qu'avec DQN c'est tout le réseau qui est modifié. Utiliser un 'target network' permet alors de stabiliser l'apprentissage.

Nous expérimentons l'algorithme DQN sur `cartpole`, et une version adaptée de `gridworld`. Afin d'évaluer l'importance de l'experience replay et du target network, nous testons des versions avec et sans. Les courbes d'apprentissage correspondantes se trouvent sur la figure 8.



(a) `gridworld` (carte 0)



(b) `cartpole`

FIGURE 8 – Cumul du reward par épisode sur 1000 itérations

Utiliser un buffer d'experience-replay et un target-network améliore la stabilité de l'agent DQN. Sur gridworld, DQN obtient les rewards maximums, sur cartpole, il obtient un score de 200 environ.

5 Policy gradients : A2C

Les algorithmes *policy-based* s'intéressent directement à la politique π_θ, avec θ l'ensemble des paramètres (généralement un réseau de neurones). L'objectif est de trouver une politique π_θ qui génère des trajectoires maximisant la somme des récompenses : on cherche donc à maximiser $J(\theta) = \sum_\tau \pi_\theta(\tau) R(\tau)$. Les paramètres optimaux de cette politique sont appris par montée de gradient.

$$\nabla_\theta J(\theta) = \sum_\tau \pi_\theta(\tau) \left[\sum_{t=0}^{|\tau|-1} \nabla_\theta \, log \, \pi_\theta(a_t|s_t) \, \mathcal{R}(\tau) \right]$$
$$\nabla_\theta J(\theta) \approx \nabla_\theta \, log \, \pi_\theta(a_t|s_t) \, \mathcal{R}(\tau) \quad \text{(version online)}$$

FIGURE 9 – Gradient utilisé dans l'algorithme REINFORCE

Avec cette méthode, recevoir un reward positif augmente la probabilité de refaire l'action effectuée, et ce même dans le cas où n'importe quelle autre action aurait été meilleure. Introduire une baseline permet dans ces cas-là de réduire fortement la variance en ne conservant que l'avantage tiré de l'action choisie.

L'algorithme *Advantage Actor Critic (A2C)* entraîne deux réseaux de neurones :

— Le réseau 'actor' est la politique, elle peut être obtenue en appliquant un softmax sur un réseau θ. On obtient ainsi une distribution de probabilités sur les actions ;

— Le réseau 'critic' apprend la valeur des états. Cette valeur sert de baseline. On obtient un 'avantage' $A(s_t, a_t) = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$

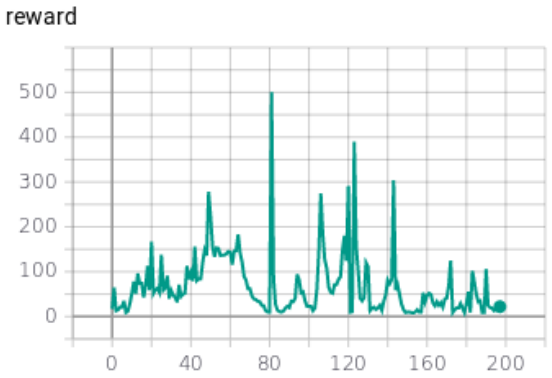


FIGURE 10 – reward de A2C sur 200 itérations

L'agent A2C est très instable, il atteint parfois un reward à 500 mais en moyenne ses résultats sont inférieurs à ceux de DQN.

Une première amélioration est de faire de l'exploration. L'agent commence par jouer aléatoirement jusqu'à avoir effectué assez de fois chaque action (autrement, le réseau se restreint parfois à l'utilisation d'une seule action). A la fin de cette

période de warmup, on entraîne le réseau V sur les trajectoires récoltées. On commence ensuite à entraîner V et θ à la manière de A2C.

Un peu à la manière des GAN, nous faisons face à un entraînement simultané de deux réseaux : θ est appris en se servant du réseau V, et le réseau V est appris à partir des trajectoires générées par θ. Ceci cause une grande instabilité :
- Si V apprend trop vite, quand l'agent obtient un bon score, le réseau V va s'attendre à ne recevoir que des bons scores. Aux itérations suivantes l'agent n'arrive géréralement pas à reproduire son score et θ va recevoir de nombreux avantages négatifs faussant tout ce qu'il avait appris auparavant, jusqu'à ce que le réseau V rebaisse ses attentes.
- A l'inverse, si V n'apprend pas assez vite, θ va recevoir à chaque itération un grand avantage, et soufrira d'une forte variance de la même manière que si V n'était pas présent.

Nous avons essayé différentes architectures neuronales pour V et Q, ainsi que différents learning rate, scheduler, optimizer, et paramètres de regularisation. Nous avons rajouté de l'experience replay pour le réseau V afin qu'il ne sur-apprenne pas les derniers scores obtenus par l'agent.

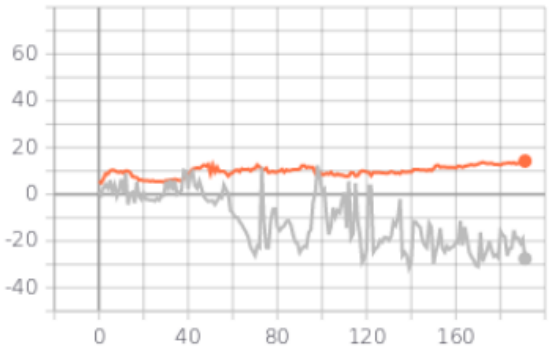
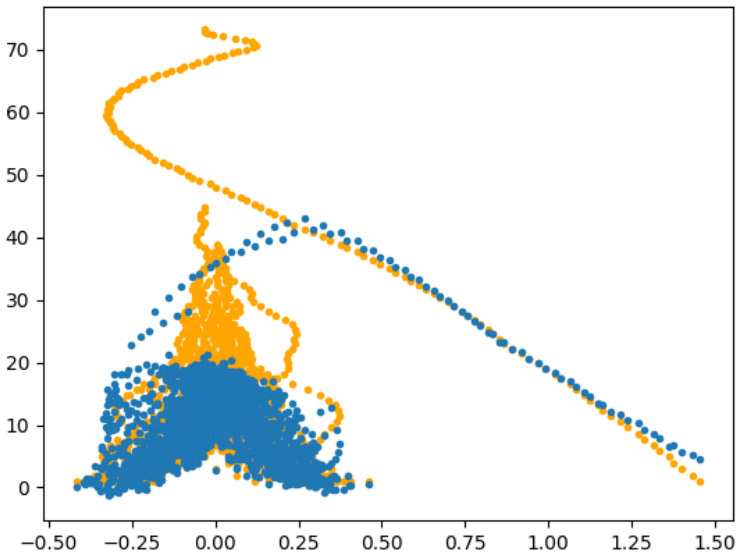


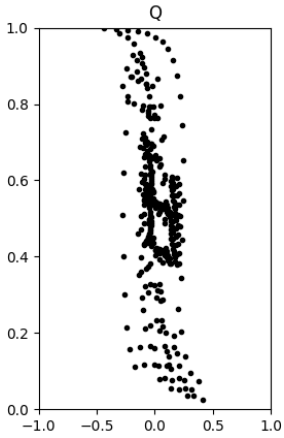
FIGURE 11 – loss θ (gris) V(orange)

Nous utilisons différentes visualisations afin de comprendre comment se passe l'apprentissage.

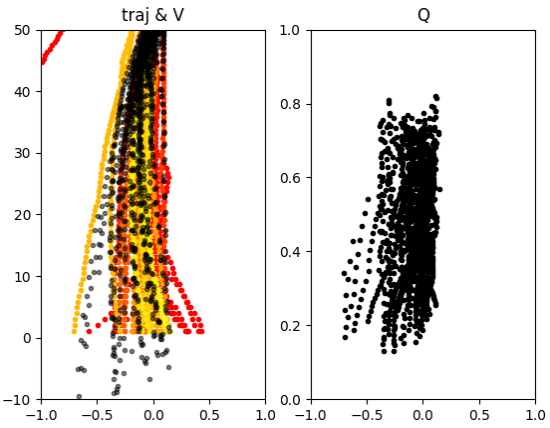
Le θ reçoit le plus souvent des loss négatives (figure a), on peut interpréter cela comme le fait qu'on dise souvent à l'agent de ne pas refaire les actions qu'il vient d'effectuer, mais qu'on ne lui dise pas quelles actions effectuer.



(a) reward obtenus (orange) valeurs prédites par V (bleu)



(b) sortie du réseau Q



(c) valeur des états (à gauche) sortie du réseau Q (à droite)

FIGURE 12 – Visualisations

La figure a représente pour chaque état la valeur de l'état en ordonnée et la distance du 'cart' au 'pole' en abscisse. L'ensemble des observations et le réseau V utilisés sont ceux obtenus à la fin du warmup. Pendant le warmup, les actions effectuées sont aléatoires et donc la vitesse du cart est en général assez faible. Le réseau V est une baseline correcte disant que plus on est sous le bâton meilleur est l'état. La trajectoire durant plus longtemps (le cart suit le bâton) est aussi apprise.

La figure b montre la probabitlité pour l'agent d'accélérer à gauche. Peu de temps après la fin du warmup, le nuage de point suit une sigmoïde.

Après le warmup, la visualisation des figures précédentes est moins pertinente car elle ne prend pas en compte la vitesse. Dans la figure d, le nuage de point des actions ne suit plus une sigmoïde comme il le faisait pour des vitesses faibles.

Les tentatives d'amélioration de l'algorithme se sont révélées infructueuses. Les améliorations suivantes nous amènent à l'algorithme PPO.

6 Policy gradients : PPO

Une itération de descente de gradient dans l'espace des paramètres de la politique peut avoir un effet insignifiant ou très significatif dans l'espace des politiques. Le 'gradient naturel' est le gradient dans l'espace des politiques. L'utilisation de ce gradient permet de stabiliser l'entraînement en utilisant un système de 'Trust-Region' empêchant les changements de politique trop brutaux.
Le calcul du gradient naturel pouvant être très coûteux, l'algorithme Proximal Policy Optimization propose une approximation rapide à calculer. Par ailleurs, il change les $log(\pi(a|s))$ en $r = \dfrac{\pi(a|s)}{\pi_{old}(a|s)}$.

$$\nabla_\theta = min(r \cdot A,\ \ clip(r, 1-\epsilon, 1+\epsilon) \cdot A)$$

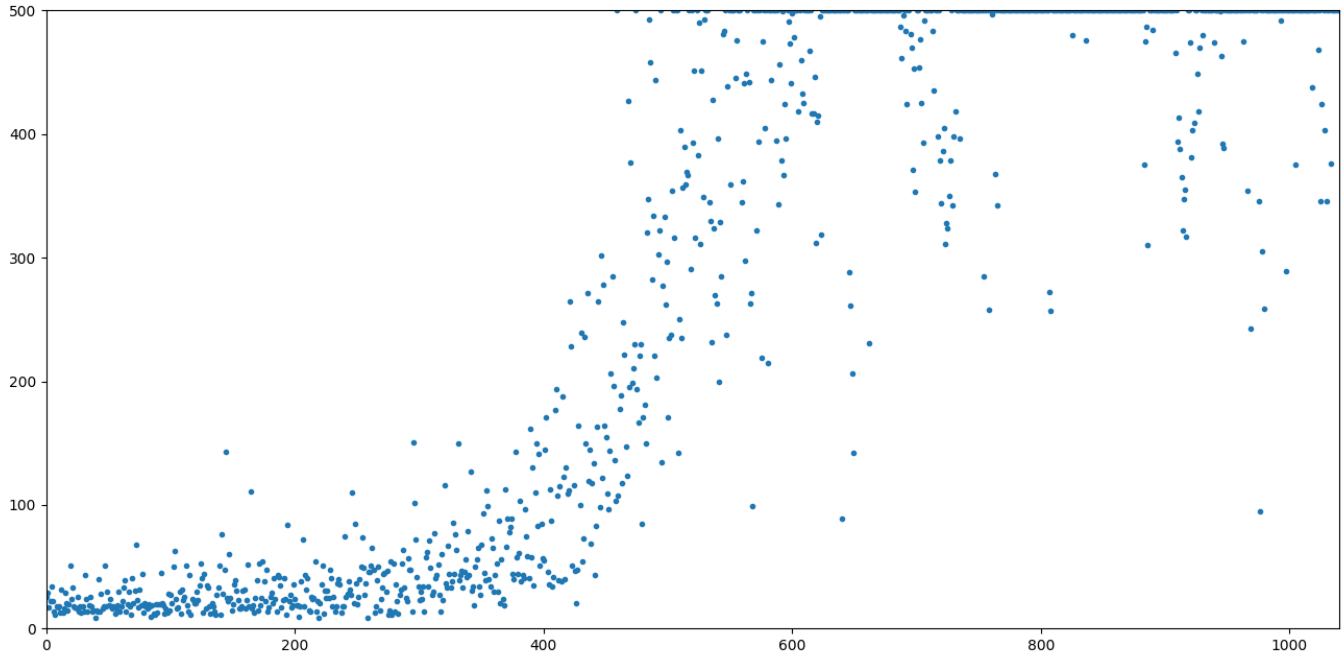


FIGURE 13 – reward de l'agent PPO au fil des itérations

L'agent PPO réalise de très bonnes performances. Le score de l'agent augmente très rapidement à partir de la 400^{me} partie environ jusqu'à atteindre un score moyen proche de 500. L'algorithme est très stable, les performances de l'agent ne se dégradent jamais au fil des parties jouées.

7 Actions continues : DDPG

Quand le nombre d'actions possibles est continu, on ne peut plus calculer la valeur de chaque action et choisir la meilleure. $argmax_a Q(s,a) \rightarrow Q(s,\mu(s))$

8 Multi Agent : MADDPG

aaa