RLD : Compte-rendu des TPs

Keyvan Beroukhim et Laura Nguyen

13 février 2020

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.

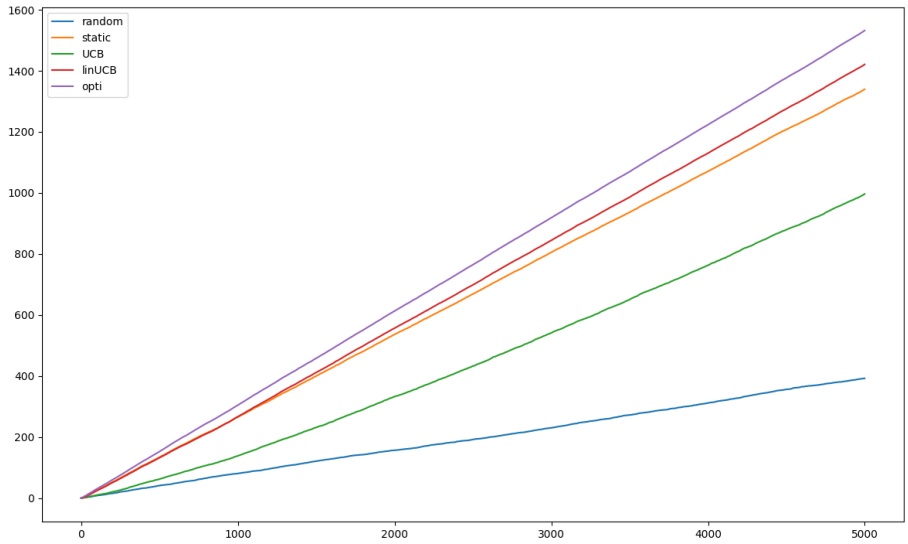
Les deux dernières baselines ne sont normalement pas disponibles étant donné que pour pouvoir les utiliser il faut connaître au préalable les taux de clics de tous les annonceurs à chaque itération. On compare ces baselines avec les stratégies UCB (*Upper-Confidence Bound*) et LinUCB (*Linear Upper-Confidence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

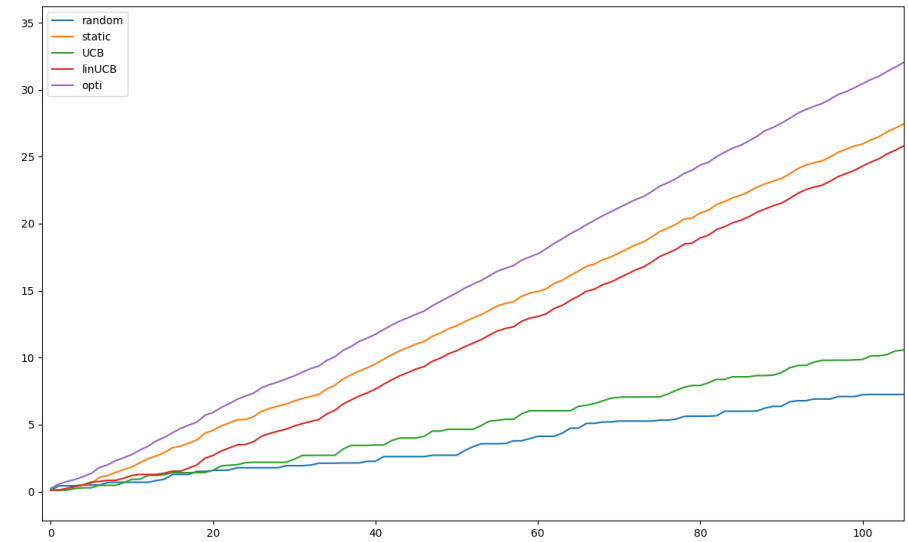
L'algorithme UCB mesure ce potentiel avec une borne de confiance supérieure du reward, qui, sommée à la moyenne empirique du gain associée à une action, fournit une borne supérieure de l'espérance du gain de cette action. On choisit l'action qui maximise cette borne de confiance.

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 transition possible

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.

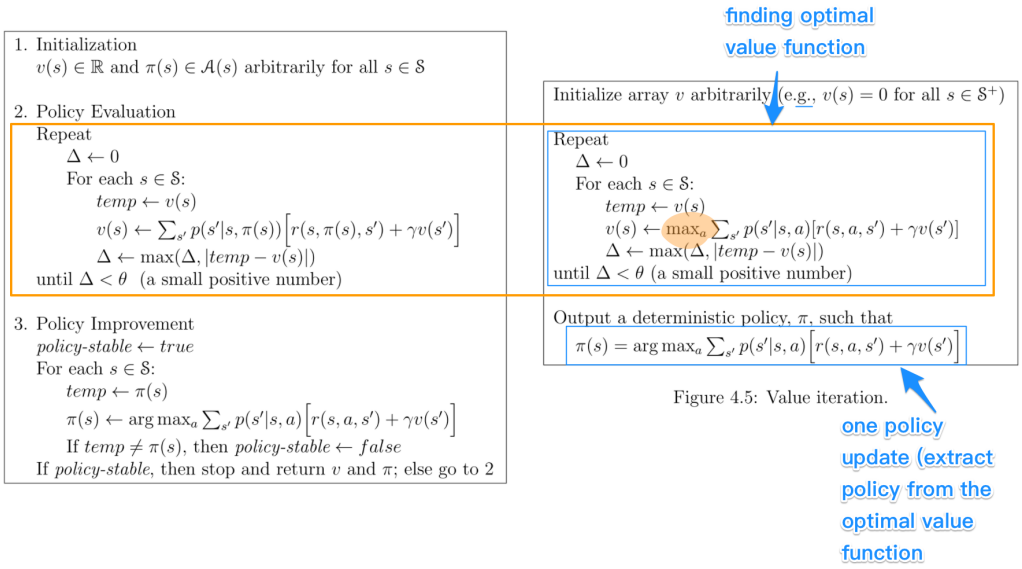


Figure 4.5: Value iteration.

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).

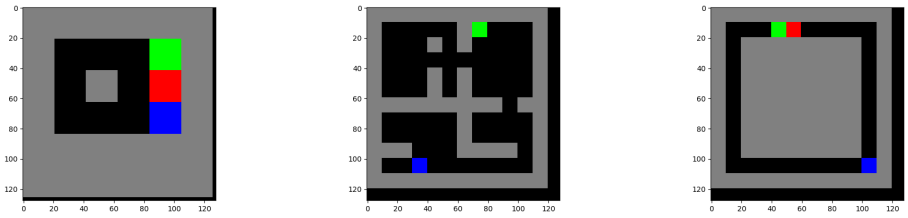


FIGURE 3 – Plans numéro 0, 5 et 10 de `gridworld`

	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.

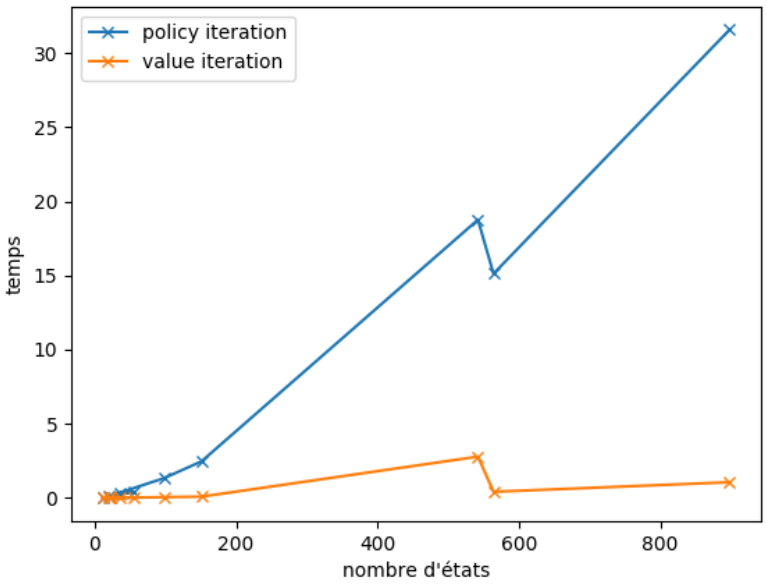


FIGURE 4 – L'algorithme Value Iteration converge plus rapidement que Policy Iteration vers la politique optimale.

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. Il se sert de son modèle du monde pour générer des transitions en interne qui sont utilisées pour mettre à jour 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.

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).

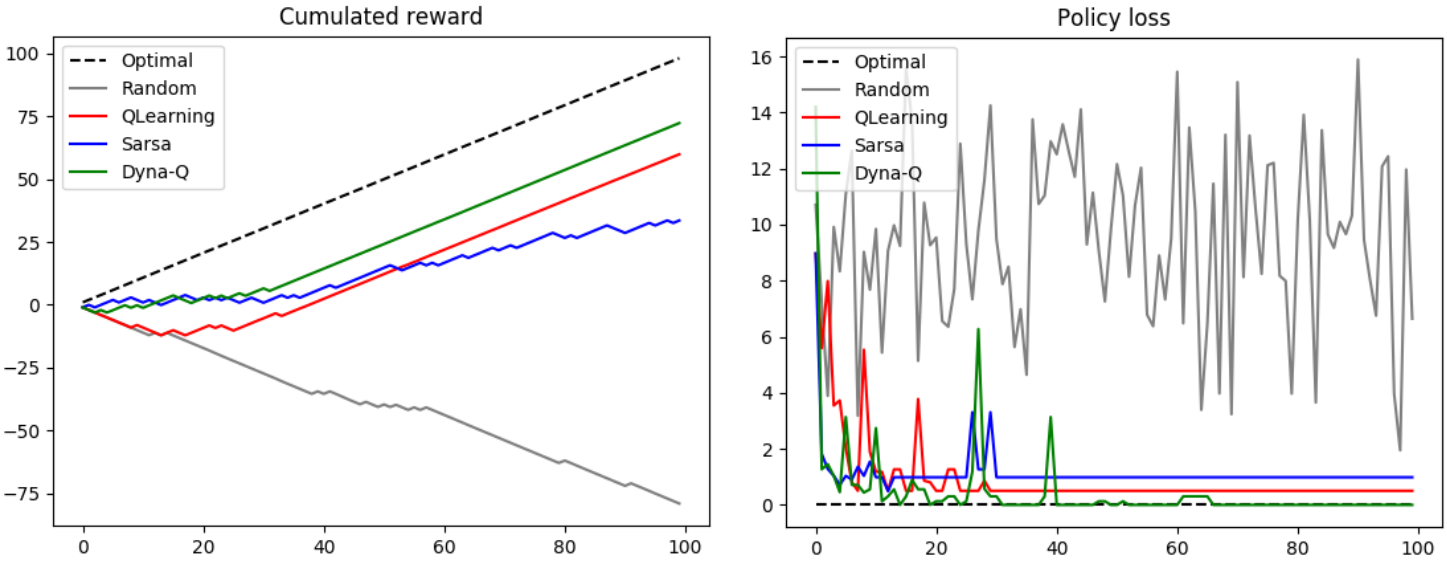


FIGURE 6 – Comparaison de performances des agents

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.

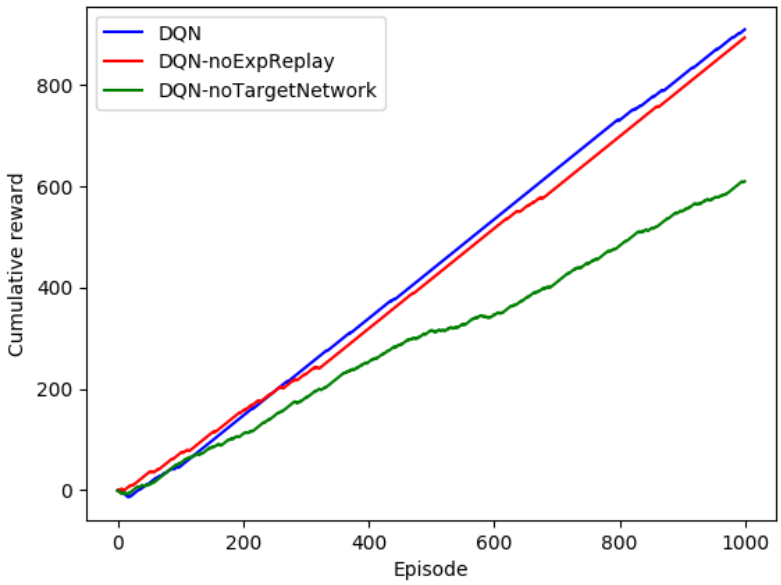
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[\ r_t + \gamma\, max_{a' \in A(s_{t+1})} Q(s_{t+1}, a') - Q(s_t, a_t)] \quad \text{(Q-Learning)}$$
$$\theta \quad \leftarrow \quad \theta \ - \ \alpha\, \nabla_\theta\ [(r_t + \gamma\, max_{a' \in A(s_{t+1})} Q(s_{t+1}, a') - Q(s_t, a_t))^2] \quad \text{(Deep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

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

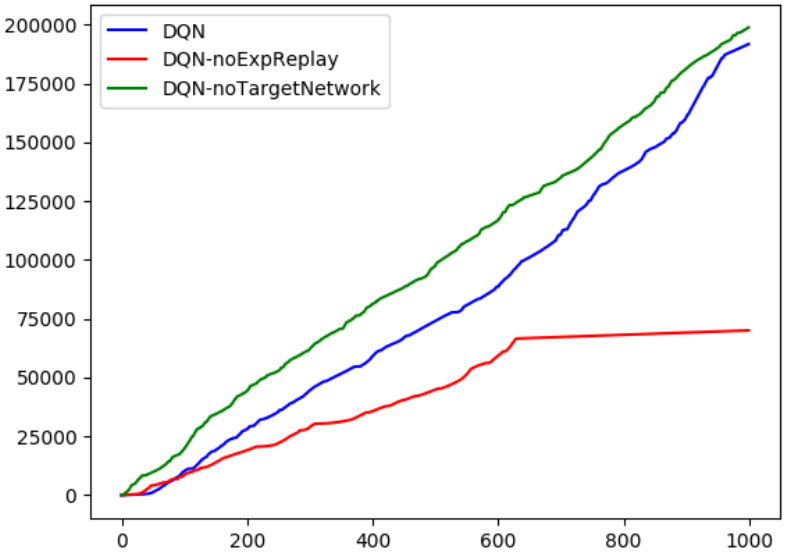
En utilisant directement s_{t+1} pour mettre à jour les paramètres du réseau Q, on ne prend pas en compte les corrélations entre échantillons d'une même trajectoire. L'utilisation d'un buffer d'experience replay, qui stocke les transitions et permet d'échantillonner des mini-batchs pour l'apprentissage, réduit la corrélation entre les exemples, permet de ne pas oublier d'informations importantes et permet 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 de paramètres θ, on obtient ainsi une distribution de probabilités sur les actions.
- Le réseau 'critic' apprend la valeur des états. Cette valeur servant 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 de la baseline 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. Nous utilisons différentes visualisations afin de comprendre comment se passe l'apprentissage.

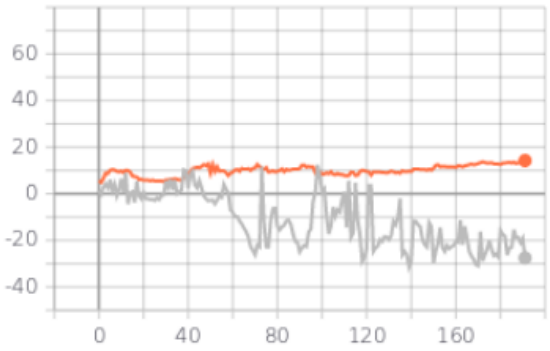


FIGURE 11 – loss θ (gris) - loss V(orange)

θ reçoit le plus souvent des loss négatives, cela peut s'interpréter 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.

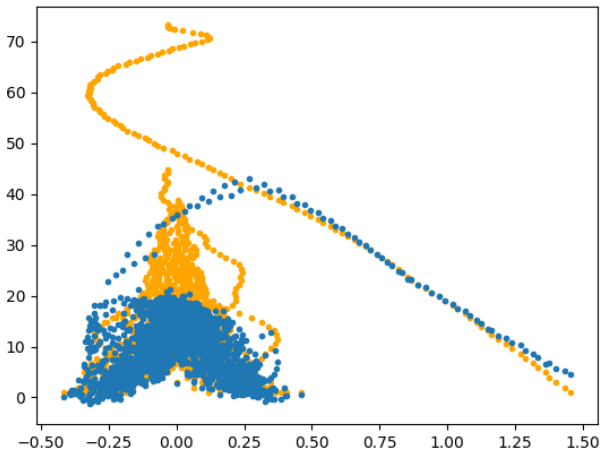


FIGURE 12 – reward obtenus (orange) - valeurs prédites par V (bleu)

La figure 12 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.

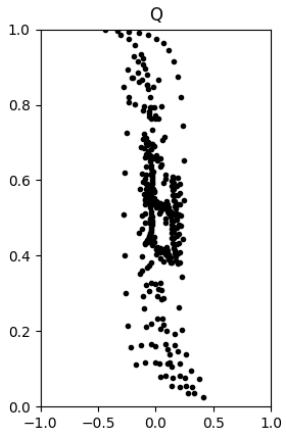


FIGURE 13 – sortie du réseau Q

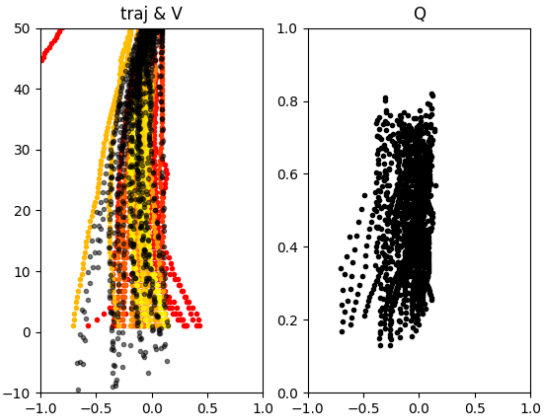


FIGURE 14 – valeur des états (à gauche) sortie du réseau Q (à droite)

La figure 13 montre la probabitlité pour l'agent d'accélérer à gauche en fonction de la distance du cart au pole. 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 14, 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 A2C s'étant révélées infructueuses, les améliorations suivantes nous amènent à l'algorithme PPO.

6 Advanced 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. De plus, le pas d'apprentissage est très difficile à régler pour les problèmes de renforcement, ce qui provoque des phénomènes de vanishing ou exploding gradients.

Enfin, les algorithmes *on-policy* mettent à jour les poids de la politique en supposant que les exemples présentés ont été obtenus en suivant cette même politque. Cela empêche l'utilisation d'experience replay et rend donc l'apprentissage instable et nécessite de beaucoup interagir avec l'environnement. Les techniques d'*importance sampling* conçues pour rendre l'apprentissage *off-policy* en pondérant l'importance des trajectoires passées ne parviennent pas à réduire la variance car les pondérations rajoutées sont aussi source de variance.

On souhaite limiter les changements de politiques et faire en sorte que chaque changement garantisse des rewards plus importants. L'idée de la méthode TRPO (*Trust Region Policy Optimization*) est de maximiser une estimation $\mathcal{L}$ de l'espérance de la fonction avantage à l'intérieur d'une région de confiance (*trust region*). Pour ce faire, TRPO utilise le *gradient naturel*, qui est le gradient dans l'espace des politiques. L'utilisation de ce gradient permet de stabiliser l'entraînement en utilisant le système de trust region empêchant les changements de politique trop brutaux.

Le calcul du gradient naturel étant très coûteux, l'algorithme PPO (*Proximal Policy Optimization*) propose une approximation du premier ordre rapide à calculer. Les mauvais déplacement sont tolérés de temps à autre car ils sont compensés par le gain de rapidité des méthodes type descente de gradient. PPO comporte deux versions : avec pénalité KL adaptative et avec objectif clippé.

PPO with Adaptive KL Penalty Une contrainte *soft* est rajoutée et garantit que l'on reste à l'intérieur d'une région de confiance.

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}(\theta) - \beta_k \hat{D}_{KL}(\pi_\theta \| \pi_{\theta_k})$$

PPO with Clipped Objective

On considère le ratio entre la nouvelle politique et l'ancienne : $r_t(\theta) = \dfrac{\pi_\theta(a|s)}{\pi_{old}(a|s)}$.

$$\mathcal{L}^{CLIP}_{\theta_k}(\theta) = \mathbb{E}_{\tau\sim\pi_k}\left[\sum_{t=0}^{\tau}\left[min(r_t(\theta)\hat{A}^{\pi_k}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}^{\pi_k}_t)\right]\right]$$

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}^{CLIP}_{\theta_k}(\theta)$$

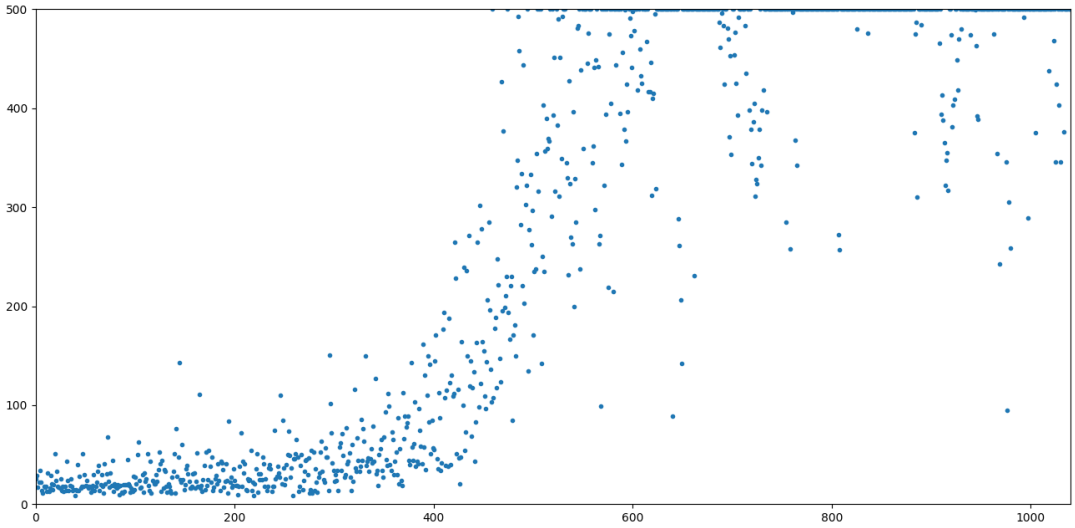


FIGURE 15 – reward de l'agent PPO au fil des itérations (CartPole)

L'agent PPO réalise de très bonnes performances. Sur CartPole, le score de l'agent augmente très rapidement à partir de la $400^{\text{ème}}$ 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

DDPG est un algorithme actor-critic off-policy. Il reprend DQN et étend ce dernier aux espaces continus. De manière off-policy, DQN apprend simultanément une estimation de la fonction Q optimale ainsi qu'une politique déterministe μ.

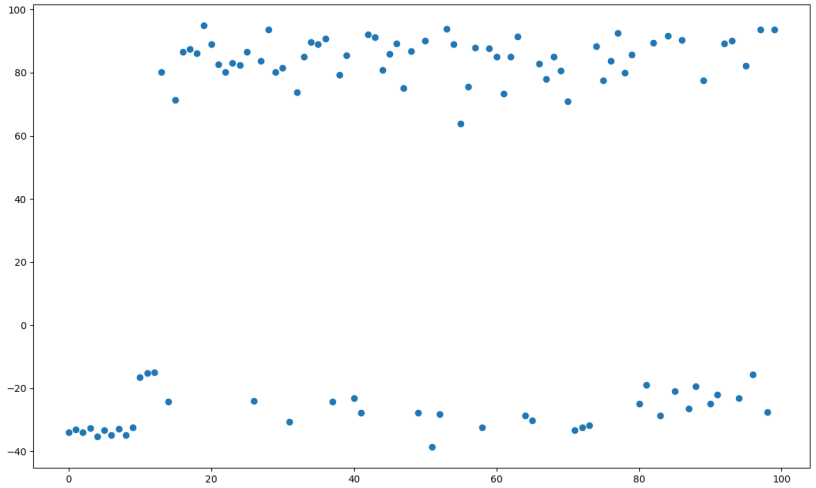
Quand l'espace des actions possibles est continu, on ne peut plus calculer les valeurs de Q pour chaque action et choisir la meilleure. Cependant, la fonction Q est désormais présumée différentiable par rapport à l'action a. On peut donc approximer $\max_a Q(s,a)$ par $Q(s,\mu(s))$.

En reprenant les concepts d'Experience Replay et Target Network de DQN, DDPG apprend la fonction Q en minimisant la loss suivante par SGD :

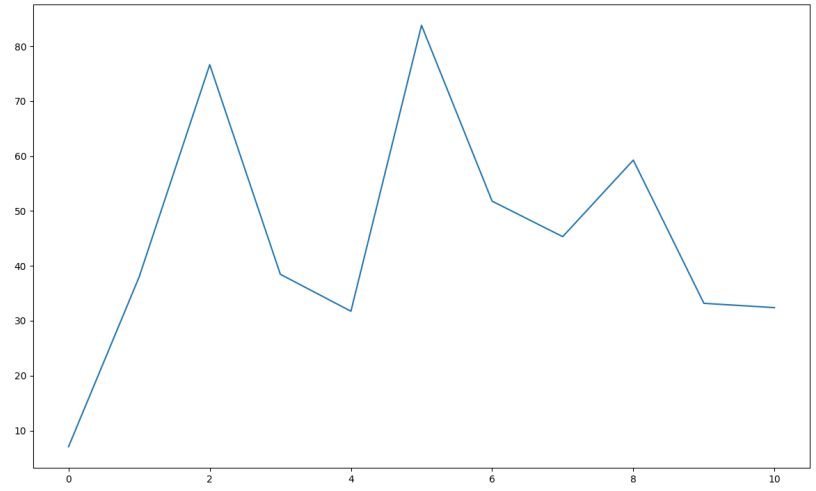
$$L(\phi,\mathcal{D}) = \mathbb{E}_{(s,a,r,s',d)\sim\mathcal{D}}\left[\left(Q_\phi(s,a) - (r+\gamma(1-d)Q_{\phi_{targ}}(s',\mu_{\theta_{targ}}(s'))\right)^2\right]$$

La politique déterministe fournit l'action maximisant $Q_\phi(s,a)$ pour chaque état s. Elle est apprise par montée de gradient (uniquement par rapport à θ) avec la fonction objectif suivante :

$$\max_\theta \mathop{\mathbb{E}}_{s\sim\mathcal{D}}\left[Q_\phi(s,\mu_\theta(s)\right]$$



(a) reward obtenus en entraînement au fil des épisodes



(b) moyenne des rewards obtenus en évaluation au cours de l'entraînement

FIGURE 16 – rewards obtenus en entraînement et en test

8 Multi Agent : MADDPG

Le cadre multi-agents est constitué de problèmes de décision séquentiels où plusieurs agents interagissent simultanément avec le même environnement, affectant ainsi le processus d'apprentissage de chacun.

On ne peut pas simplement étendre le RL classique aux problèmes multi-agents en entraînant indépendamment les agents. En effet, les transitions ne sont plus stationnaires : elles dépendent des actions des autres agents. Or les algorithmes de RL, offrant pour la plupart des garanties de convergence, nécessitent que l'environnement soit stationnaire.

Pour pallier à ce problème, l'algorithme MADDPG introduit l'idée que, si l'on connaît les actions de chaque agent, alors l'environnement est stationnaire, et ce même si les politiques des agents évoluent.

MADDPG reprend DDPG en l'étendant au cas multi-agents :
— la critique de chaque agent est apprise en utilisant les actions et observations de l'ensemble des agents
— la politique de chaque agent n'est conditionnée que par sa propre observation

9 GAN

On s'intéresse maintenant aux modèles génératifs, qui ont pour objectif de générer de nouvelles données plausibles selon une distribution de probabilité p_{data}. Cette distribution est inconnue et l'on possède seulement des exemples d'apprentissage tirés selon cette loi, qui vont nous servir à l'approximer.

Un *Generative Adversarial Network* (GAN) est composé de deux réseaux adverses :
— le générateur, dont l'objectif est de produire des données plausibles qui trompent le discriminateur ;
— le discriminateur, qui doit réussir à différencier les données réelles de celles produites par le générateur.

On expérimente les GANs sur un problème de génération de visages, à partir du dataset `CelebA`. Selon un ensemble de visages d'entraînement, il s'agit d'apprendre à générer des visages qui paraissent les plus réalistes possibles tout en conservant une certaine diversité dans les distributions de sortie. On emploie pour cela une architecture DCGAN, qui utilise des réseaux de neurones convolutionnels pour le générateur et le discriminateur.

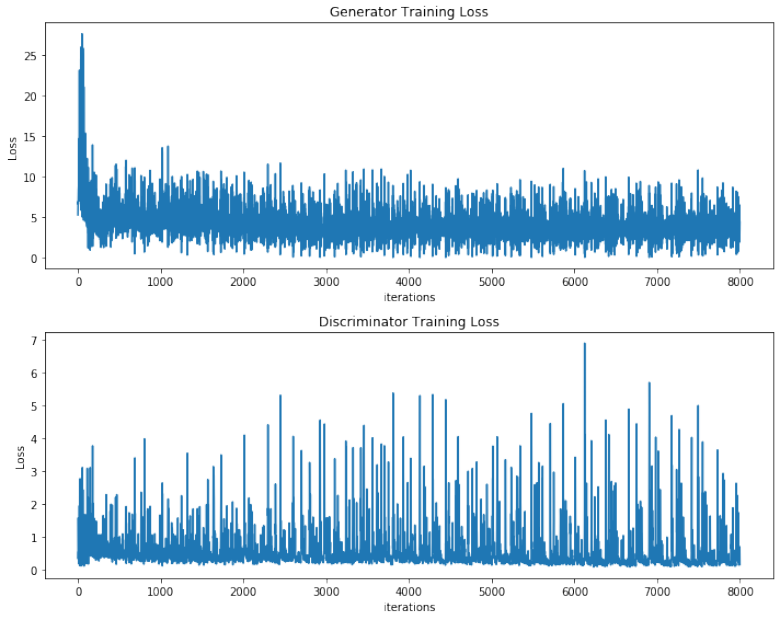
Les visages générés à différentes étapes de l'apprentissage (début, milieu et fin) ainsi que l'évolution des loss du générateur et du discriminateur sont présentés sur

la figure 17.

On constate que l'apprentissage est très instable : sur 8000 itérations, la loss de chaque réseau oscille sans jamais converger. Ce phénomène se remarque également sur les visages générés car il y a très peu de différence entre les résultats produits en plein milieu de l'apprentissage (itération 5000/8000) et les résultats générés à la fin (itération 8000/8000). En revanche, les visages générés à l'itération 5000 sont bien plus plausibles et diversifiés que ceux produits à l'itération 1000. On constate cependant qu'en zoomant, même les visages correspondant à la meilleure génération ne sont pas du tout réalistes.

Il est assez difficile d'interpréter les courbe de loss : on ne constate aucune diminution et comme les réseaux sont antagonistes, elles ne cessent d'osciller.

(a) Évolution des générations (itérations 1000, 5000 et 8000)



(b) Évolution des loss

FIGURE 17 – Apprentissage d'un GAN

10 VAE

Un auto-encodeur variationnel (VAE) est un modèle génératif qui reprend le principe d'un auto-encodeur classique en modifiant le processus d'encodage des données.

1. L'entrée x est encodée par une distribution sur l'espace latent, $q_\phi(z|x)$. Cette dernière est une gaussienne permettant d'approximer $p(z|x)$.
2. Le vecteur z est obtenu en échantillonnant selon $q_\phi(z|x)$.
3. Le décodeur produit une distribution $p_\theta(x|z) \approx p(x|z)$ (on utilisera une loi de Bernoulli). En échantillonnant selon cette dernière, on obtient une image.

L'encodeur et le décodeur sont des réseaux de neurones. En plus de l'objectif de reconstruction des données, une régularisation explicite de l'espace latent est introduite : $q_\phi(z|x)$ est contrainte à ressembler à une distribution normale centrée réduite. La fonction de coût est donc la suivante :

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

On implémente une architecture constituée uniquement de couches linéaires et de non-linéarités. On expérimente sur le dataset MNIST, qui contient 70 000 images de chiffres manuscrits allant de 0 à 9 (60 000 en train, 10 000 en test). Par souci de temps de calcul, les modèles ne seront entraînés que sur 10 000 données en train et 1000 en validation.

On étudie l'influence de la dimension de l'espace latent, z, sur la reconstruction des images (figure 18). On constate qu'avec une valeur très petite ($z = 2$), les images reconstruites par le réseau sont très floues, et ce tout au long de l'entraînement. En augmentant cette dimension ($z = 16$), on observe une amélioration des résultats, les images sont bien plus nettes. Cependant, fixer une valeur plus grande ($z = 64$) ne génère pas de meilleures images.

(a) Epoch 1/10 (b) Epoch 5/10 (c) Epoch 10/10 (d) Test

FIGURE 18 – Images originales et reconstruites en apprentissage et en test.

On trace également les coordonnées du vecteur μ, de dimension 2, obtenu pour chaque image de test (figure 19). On peut voir que l'espace latent a une structure interprétable. En effet, d'après la matrice de confusion de la figure 20, les chiffres 0, 1 et 7 sont les plus faciles à identifier correctement (associés aux trois plus hauts taux de rappel). Or, ils forment les trois groupes les plus distinguables sur notre

visualisation. Les autres classes sont davantage confondues, ce qui est en accord avec la disposition des données dans l'espace latent.

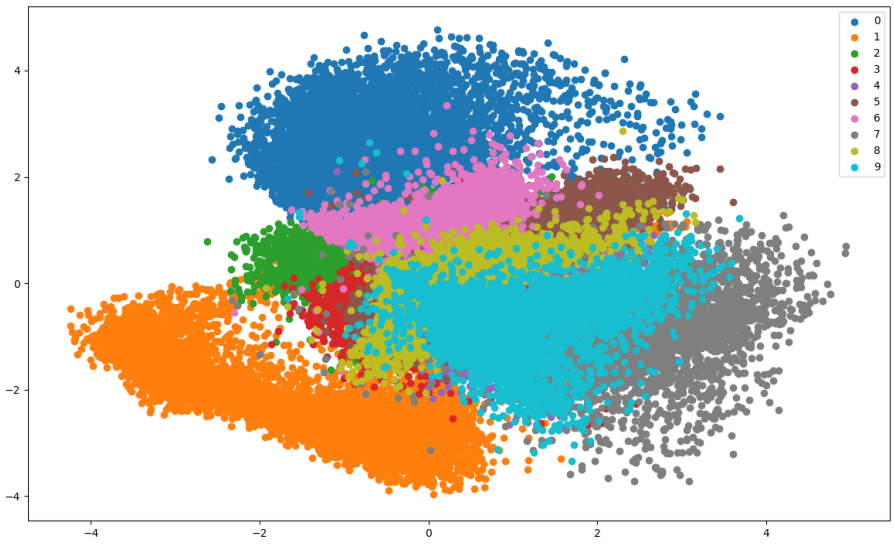


FIGURE 19 – Visualisation du vecteur μ obtenu pour chaque donnée de test.

	predicted 0	predicted 1	predicted 2	predicted 3	predicted 4	predicted 5	predicted 6	predicted 7	predicted 8	predicted 9	recall
actual 0	954	0	0	7	1	10	6	3	7	3	96%
actual 1	0	1031	4	3	1	4	1	2	16	2	97%
actual 2	12	21	852	18	11	8	14	20	29	5	86%
actual 3	2	5	9	899	1	71	0	12	23	7	87%
actual 4	2	8	2	2	861	7	7	1	4	89	88%
actual 5	7	5	9	24	3	833	12	8	12	2	91%
actual 6	11	6	2	0	6	31	902	0	8	1	93%
actual 7	3	10	5	3	7	7	1	1041	0	14	95%
actual 8	2	28	4	29	2	31	1	9	882	21	87%
actual 9	7	3	1	7	10	11	1	44	4	873	91%
precision	95%	92%	96%	91%	95%	82%	95%	91%	90%	86%	accuracy 91%

FIGURE 20 – Matrice de confusion obtenue sur MNIST (source).