État de l'art de l'apprentissage par renforcement.

Hugo Rimlinger, encadré par Jilles S. Dibangoye

26 octobre 2019

1 Introduction

L'utilisation toujours plus grande des robots dans notre vie quotidienne pousse les chercheurs à élaborer des méthodes afin de résoudre des problèmes de plus en plus complexes. Créer des intelligences artificielles susceptibles d'apprendre pour élargir leur domaine d'utilisation est un enjeu majeur pour avoir des intelligences artificielles capables de s'adapter à un environnement réel et d'adopter le comportement optimal dans une situation. C'est le but recherché par l'apprentissage par renforcement, une branche de l'intelligence artificielle à laquelle cet état de l'art sera consacré. Pour cela, les chercheurs se sont basés sur un mécanisme simple mais essentiel que l'on retrouve largement dans la nature : l'apprentissage par essai-erreur.

Lorsque nous sommes confrontés à une situation que l'on ne connaît pas, nous allons adopter un certain comportement. De ce comportement nous allons tirer une expérience qui nous permet ensuite de construire un enseignement afin d'améliorer notre prise de décision face à une situation similaire. Tel est le mécanisme d'apprentissage qu'essaye de traduire les techniques d'apprentissage par renforcement. Les premières méthodes d'apprentissage par renforcement font leur apparition dans les années 90 avec des algorithmes comme le Q-learning (Watkins, 1992 [14]) mais la formalisation mathématique des problèmes de contrôle remonte aux années 50 à travers les travaux du mathématiciens Bellman.

Récemment, grâce à l'augmentation de la puissance de calcul des ordinateurs et à l'explosion de la taille des bases de données ces algorithmes ont pu être implémentés dans des environnements réels. Alpha Go, un algorithme créé par Google DeepMind [6], est un parfait exemple des avancées spectaculaires dans le domaine de l'apprentissage par renforcement. En effet cette intelligence artificielle a réussi l'exploit de battre le meilleur joueur de Go du monde, un jeu pourtant réputer pour sa complexité. Nous étudierons au travers de cet état de l'art comment les chercheurs ont formalisé le problème d'apprentissage et comment l'évolution des algorithmes a permis d'obtenir des résultats dépassant les performances humaines pour des problèmes de décisions extrêmement complexe comme les stratégies de jeu.

Table des matières

2 Le problème d'apprentissage par renforcement

Dans cette première partie, nous commencerons par étudier comment les chercheurs ont décidé de modéliser les problèmes de prise de décisions séquentielles. Pour pouvoir appliquer un algorithme d'apprentissage par renforcement, il nous faut trois composantes :

— un agent (il s'agit de l'entité à laquelle on va faire apprendre)
— l'environnement (le modèle dans lequel évolue l'agent)
— un système de récompense (pour pouvoir quantifier la qualité d'une action)

La nature de l'agent et de l'environnement diffèrent en fonction des applications, il peut s'agir d'un jeux vidéo (largement utilisés pour mesurer les performances des algorithmes), d'un véhicule ou encore d'un robot. Le cadre très général permettant de mettre en place les algorithmes d'apprentissage permet de résoudre une grande variété de situations, il suffit d'identifier dans le problème les trois entités énoncées ci-dessus. Une fois ces trois composantes réunies, il est possible de modéliser l'apprentissage de l'agent à travers son interaction avec l'environnement (voir figure 1).

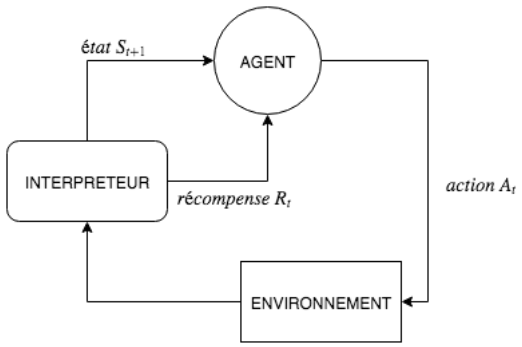

FIGURE 1 – schéma d'interaction entre l'agent et son environnement

l'interaction entre l'agent et son environnement peut être résumée en trois étapes :

1. l'agent se trouve dans un état courant S_t.
2. il choisit une action A_t parmi toutes les actions possibles.
3. il transite vers un état S_{t+1}
4. et récupère une récompense R_t pouvant être négative ou positive.

La transition (S_t, A_t, S_{t+1}, R_t) représente le résultat de l'interaction entre l'agent et son environnement. C'est cette transition qui va constituer la base de l'apprentissage de l'agent, l'objectif étant, à partir de ces interactions d'améliorer la prise de décision de l'agent, afin de maximiser le cumule des récompenses. (nous reviendrons sur les méthodes permettant d'améliorer le comportement dans les partie suivantes).

Le modèle de récompense. La récompense peut être définie comme un signal qui rend compte de la préférence de l'agent pour une action prise dans un état donné. Elle peut par exemple représenter le score d'un jeu vidéo : si l'agent

gagne des point après une action alors la récompense est positive, au contraire si celui-ci perd la partie la récompense va être négative. La fonction de récompense n'est pas connue car il s'agit d'une propriété de l'environnement (i.e la récompense obtenue lors d'une transition d'un état à un autre). C'est néanmoins l'utilisateur qui fixe le but de l'agent, i.e. les paramètres qu'il va devoir maximiser pour réaliser la tâche. Si ce modèle est faux, alors le comportement de l'agent ne permettra pas de résoudre le problème.

2.1 Définition du comportement de l'agent

La politique. L'objectif de l'apprentissage par renforcement est de trouver la fonction permettant de choisir la meilleure action dans un état donné. Le fait de prendre une action A_t dans un état S_t est appelé politique (ou règle décisionnelle), notée $\pi(S_t)$ ou $\pi(A_t|S_t)$ (selon que la politique soit déterministe ou stochastique). Résoudre le problème de décision revient à calculer la politique optimale, que l'on notera $\pi^*(a|s)$. Pour cela l'agent va devoir répéter un grand nombre de fois la tâche à effectuer pour acquérir suffisamment de transitions et améliorer son comportement. Cette politique doit aussi permettre à l'agent d'explorer son environnement pour en avoir une connaissance suffisante. Si on explore un seul chemin, qui ne correspond pas forcément aux états permettant de maximiser le cumule des récompenses, le comportement de l'agent ne sera pas optimal. Il faut donc que l'agent soit aussi capable de prendre des décisions aléatoires pour explorer de nouveaux états. La dualité entre exploration et optimisation est une réelle problématique des algorithmes d'apprentissage par renforcement, nous verrons dans la suite comment cela a été géré.

la fonction de valeur. Pour savoir si une politique permet de prendre la bonne action A_t dans l'état S_t, il nous faut une fonction capable de mesurer sa performance. Ce critère de performance est modélisé par une fonction de valeur. En fonction de la façon dont nous allons étudier le problème, nous allons utiliser l'une ou l'autre des fonctions de valeurs. $V^\pi(S_t)$ est la fonction utilisée pour un problème de prédiction, elle représente la qualité de la politique $\pi(A_t|S_t)$ dans l'état S_t. Dans les problèmes de prédiction, on ne s'intéresse pas aux choix des actions ni à la dynamique de transition entre les états. Au contraire, dans un problème de contrôle, où nous allons véritablement interagir avec l'environnement, il est important de connaître la qualité de la règle décisionnelle pour chaque paire état-action. On note cette fonction $Q^\pi(S_t, A_t)$.

2.2 Type de tâche à réaliser

Comme nous l'avons évoqué la transition (S_t, A_t, S_{t+1}, R_t) ne constitue qu'une infime partie de l'expérience permettant à l'agent d'apprendre. Pour que celui-ci puisse résoudre le problème de décision il faut que son apprentissage s'effectue sur l'ensemble des paires états-action de l'environnement dans lequel il évolue. Nous distinguons dans ce cas deux types de problèmes :

- les problèmes épisodiques (qui possèdent un état initial S_0, un état final S_n)
- les problèmes continus (où c'est l'utilisateur qui choisit de mettre fin à la simulation)

Même si la modélisation du problème reste relativement la même, cela a une conséquence importante sur la définition des fonctions de valeur. Dans le cas d'une tâche épisodique la fonction de valeur a pour expression [5] :

$$Q^{\pi}(s,a) = \mathrm{E}_{\pi}[\sum_{t=0}^{\infty} \gamma^t R_t]$$

La fonction de valeur représente l'espérance du cumule des récompenses décomptée, à partir de l'état initial et en suivant la politique $\pi(a|s)$. Le facteur γ permet de prendre plus ou moins en compte les récompenses futures dans l'estimation du cumule des récompenses et de s'assurer que la somme ne diverge pas.

Pour les tâches continues, dans la mesure où le problème ne possède pas de fin, on ne peut pas calculer l'espérance de la somme des récompenses directement. On utilise donc un nouveau terme qui va mesurer la moyenne des récompenses attendues en suivant la politique $\pi(a|s)$ [5] :

$$\rho^{\pi} = \lim_{N\to\infty} \frac{1}{N} \sum_{t=0}^{N-1} R_t$$

Enfin, on va comparer la somme des récompenses en partant de l'état S_t par rapport à la moyenne des récompenses attendues. Cela nous permet de mesurer l'avantage que nous avons à partir d'un certain état par rapport à un autre et donc évaluer la performance de la politique :

$$Q^{\pi}(s,a) = \lim_{N\to\infty} \mathrm{E}_{\pi}[\sum_{t=0}^{N-1} R_t - \rho^{\pi}]$$

Plus la valeur de la fonction est grande pour une paire (S_t, A_t), plus la politique associée sera performante. On comprend ici que ces deux fonctions (politique et fonction de valeurs) sont fortement corrélées, nous verrons qu'optimiser l'une ou l'autre de ces deux fonctions aura une importance sur les classes d'algorithmes utilisées pour résoudre le problème de décision.

La modélisation complète de l'interaction entre l'agent et son environnement en fonction des deux types de tâches est donnée ci-dessous.

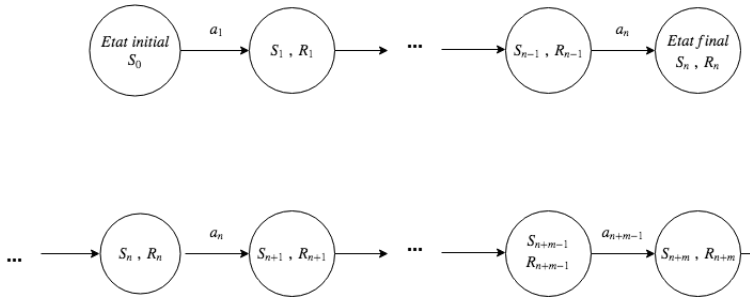

FIGURE 2 – Figure 2.a (haut) : représentation d'une tâche épisodique ; Figure 2.b (bas) : représentation d'une tâche continue

Comme nous pouvons le voir à travers ces schémas, le type de tâche à réaliser ne change pas la nature de l'expérience acquise par l'agent, il s'agit pour les deux cas d'un ensemble de transition (S_t, a_t, S_{t+1}, R_t). L'ensemble des transitions rencontrées par l'agent lors d'un épisode ne représente pas une expérience

suffisante pour que l'agent puisse apprendre une règle décisionnelle optimale. Il va donc falloir répéter ces épisodes un grand nombre de fois pour que celui-ci ait une connaissance suffisante de son environnement. C'est sur cette expérience acquise lors de la phase d'apprentissage que les méthodes algorithmiques permettent à l'agent d'améliorer son comportement.

Les méthodes d'apprentissage par renforcement s'articulent toutes sur le même schéma de fonctionnement : acquérir de l'expérience à travers la répétition d'une tâche en essayant d'explorer le plus possible. Il faut ensuite être capable de traiter cette expérience afin d'en tirer un enseignement afin de converger vers le comportement optimal. Dans la prochaine partie nous allons donc voir comment les différents algorithmes d'apprentissage par renforcement exploitent et traduisent les concepts que nous avons introduit au cours de cette première partie.

3 Les différentes classes d'algorithmes

Nous allons voir au travers de cette partie que les algorithmes de référence dans la littérature reposent sur un certains nombre de propriétés qui dépendent de la nature du problème et de la solution. En combinant les différentes caractéristiques que nous allons détailler dans cette partie il est possible de retrouver la plupart des algorithmes existant.

3.1 Dimension 1 : dépendance par rapport à l'environnement

On distingue ici deux classes de solutions pour pouvoir mettre en place les algorithmes d'apprentissage par renforcement. L'une où l'agent va évoluer dans son environnement sans en connaître la nature et sans chercher à l'apprendre (model-free) et la seconde où la phase d'apprentissage de la politique va être précédée par une période de modélisation de l'environnement (model-based). Dans les méthodes model-based, l'agent va chercher à connaître les probabilités de transition entre les états, i.e la dynamique de l'environnement, et agir en conséquences. Les propriétés que nous détaillerons dans la suite de cette partie peuvent être appliquées tant aux algorithmes model-free que model-based.

Une fois que nous avons choisit si une connaissance a priori du modèle est nécessaire il nous faut choisir sur quelle fonction la mise à jour va être appliquée (i.e sur la politique ou la fonction de valeur).

3.2 Dimension 2 : type de résolution : direct/indirect

Comme nous l'avons introduit, l'apprentissage par renforcement tente de résoudre le problème de décision en améliorant le comportement de l'agent. Cette résolution peut-être de deux nature : soit direct, i.e. on applique les mise à jours sur la politique, ou indirect, en travaillant sur la fonction de valeur. La première catégorie d'algorithme est appelée *policy-based* et la seconde *value-based*. Une dernière classe d'algorithme permet à la fois d'améliorer la fonction de valeur et la politique conjointement. Ces algorithmes sont appelés *acteur-critique* avec la

partie "acteur" représentée par la politique et "critique" qui désigne la fonction de valeur.

Quelque-sois la fonction que nous allons mettre à jour il est faut choisir la nature de l'approximation entre la fonction optimale π^* que l'on cherche et la politique π que l'on calcul.

3.3 Dimension 3 : Architectures d'approximation

Historiquement, les premiers algorithmes d'apprentissage par renforcement se basaient sur une résolution asymptotique du problème de décision sans utiliser d'architecture d'approximation. Dans ce type de résolution, l'agent doit parcourir l'ensemble des paires (S_t, A_t) pour pouvoir converger vers la politique optimale. De plus, il faut répéter cette action un nombre suffisamment grand de fois pour que l'agent puisse choisir la meilleure action possible dans un état donné. On comprend vite pourquoi ce type de solution est limité, si l'espace d'état est très grand, le temps de calcul nécessaire pour approcher le comportement optimal est infiniment long. Cela est d'autant plus vrai pour un environnement réel où l'espace des états possible est infini. Il a donc fallu apporter une structure d'approximation à ces algorithmes pour pouvoir réduire l'espace des états afin de l'appliquer à des environnements de grande taille. L'objectif des méthodes approchées est de déterminer les principales caractéristiques de l'environnement pour pouvoir généraliser l'expérience acquise dans un état à tous les états qui lui ressemble.

3.3.1 Approximation par combinaison linéaire de fonction

Cette architecture d'approximation cherche à représenter l'ensemble des états par une combinaison linéaire de fonctions qui permettent de décrire l'environnement dans lequel évolue l'agent. La fonction de valeur approximée a pour expression [10] :

$$Q_\pi(s, a; \theta) = \sum_{i=1}^{n} \theta_i \phi_i(s, a) \approx Q_\pi(s, a) \text{ ou } Q_\pi(s, a; \theta) = \theta \phi^T$$

avec $\theta = [\theta_1, \theta_2, ...\theta_n]$ le vecteur de poids et $\phi = [\phi_1, \phi_2, ...\phi_n]$ l'ensemble des fonctions caractéristiques permettant de décrire l'état. On recherche donc $Q_\pi(s, a, \theta)$, de dimension égale au vecteur θ et paramétrée par celui-ci. Chacune des composantes θ_i vient ajuster la valeur retournée par la fonction ϕ_i correspondante. C'est donc sur le vecteur θ que vont s'effectuer les mises à jour (voir partie 4.2). La principale difficulté de cette méthode est de déterminer le vecteur ϕ permettant de représenter au mieux les caractéristiques de l'environnement. Il revient à l'utilisateur d'identifier les données importantes du problème (règle d'un jeu, distance entre l'agent et son objectif, etc...) car, si la représentation de l'environnement est fausse ou mauvaise, la politique résultante le sera aussi. De plus, cette méthode n'est pas applicable si la politique optimale ne peut pas être appoximée par une fonction linéaire.

A cause de ces contraintes, l'approximation par combinaison linéaire de fonctions n'est pas beaucoup utilisée pour des environnements réels où le nombre de paramètre à identifier est grand. Elle constitue tout de même une méthode

efficace pour résoudre certains jeu (comme le Pacman par exemple) où les caractéristiques de l'environnement sont simple à identifier.

3.3.2 Approximation par réseau de neurones

L'objectif de cette architecture est de remplacer la phase de caractérisation de l'environnement par l'humain par un réseau de neurones. Cette technologie permet, en passant une représentation de l'environnement à l'entrée du réseau, de réduire l'ensemble des états en un vecteur de poids de dimension finie. La mise à jour de la politique ou de la fonction de valeur est ensuite la même que pour l'approximation par combinaison linéaire de fonction (voir partie 4.2).

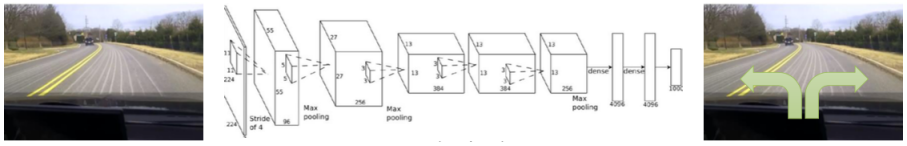

FIGURE 3 – schéma d'un réseau de neurones, à gauche la représentation de l'environnement, à droite le résultat du réseau de neurones (les actions possibles). (schéma tiré de la présentation : "Deep reinforcement Learning for multi robots system" de Jilles S. Dibangoye).

Nous ne rentrerons pas dans le détail du fonctionnement des réseaux de neurones, car ce n'est pas l'objet de cet état de l'art, mais notons tout de même qu'à la sortie de chaque couche de neurones est appliquée une fonction d'activation. La politique (ou fonction de valeur) est donnée par la composition de ces fonctions. En plus d'être capable de déterminer les caractéristiques d'un environnement , les réseaux de neurones permettent d'introduire de la non-linéarité, ce qui est nécessaire pour approcher la politique si celle-ci n'est pas linéaire. Le principal désavantage de cette technique est que le temps de convergence vers la politique optimale est très long. Si le réseau est complexe (avec beaucoup de couches de neurones) il faut un grand nombre de transitions pour l'entraîner. Il faut donc que le réseau de neurones soit adapté au problème en terme de complexité.

Une fois que nous avons choisit quel type d'architecture d'approximation nous allons utiliser, il faut choisir la façon dont nous allons explorer l'environnement. Ce qui est, rappelons le, l'un des principes fondamentaux de l'apprentissage par renforcement.

3.4 Dimension 4 : nature de l'exploration

Nous allons voir dans cette partie une notion centrale de l'apprentissage par renforcement : l'utilisation de la politique dans le problème de décision. Nous avons ici deux méthodes principales : les problèmes *off-policy* et *on-policy*.

on-policy. Dans ce type d'algorithme, la politique que nous cherchons à améliorer est la même que celle qui est utilisée pour choisir une action dans l'état

courant. Cela peut paraître contradictoire car nous cherchons, à travers cette méthode, à approcher la règle décisionnelle optimale, tout en continuant à explorer l'environnement (sélectionner aléatoirement des actions ne correspond pas au comportement optimal).

off-policy. Pour ces méthodes, la politique qui permet de choisir la prochaine action de l'agent n'est pas la même que celle qui est mise à jour à la fin de l'épisode. La première, qui garanti l'exploration, est notée $\mu(S_t)$ tandis que la seconde, correspond au comportement optimal, elle est notée $\pi(S_t)$.

Nous verrons dans la suite de cet état de l'art les conséquences que ces deux classes d'algorithmes peuvent avoir dans le processus d'apprentissage de l'agent.

Toutes ces caractéristiques permettent de construire la majeure partie des algorithmes d'apprentissage par renforcement. La dernière dimension qu'il nous reste a traiter est la méthode de mise à jour. Cette dernière permet de faire réellement apprendre à l'agent et constitue la partie la plus importante des algorithmes.

4 Les règles de mise à jour

Dans la première partie nous allons nous intéresser aux méthodes de mises à jour de la fonction de valeur dans le cas tabulaire, c'est à dire sans utiliser d'architecture d'approximation (i.e une résolution exacte du problème de décision). Cela nous permettra d'introduire deux méthodes majeures de l'apprentissage par renforcement : la méthode de Monte-Carlo et le TD-learning. Nous donnerons également leurs équivalents approchés dans la seconde partie qui traitera des méthodes de mises à jour se reposant sur une architecture d'approximation.

4.1 Méthodes tabulaire

4.1.1 Monte-Carlo

La Méthode de Monte-Carlo est une classe d'algorithme permettant d'estimer la fonction de valeur afin de déterminer la politique optimale. C'est une technique qui ne nécessite pas une connaissance a priori du modèle (nous n'avons pas besoins de connaître l'environnement pour l'appliquer). Il faut seulement être capable de récupérer l'état dans lequel se trouve l'agent et la récompense associée à une transition entre deux états. Cette méthode peut être scindée en trois parties :

1. une première partie ou un épisode va être généré par la politique courante (initialisé au début de l'algorithme), à partir d'un état pris de manière aléatoire
2. une seconde partie où l'on va calculer la fonction de valeur associée à partir des transitions qui ont été visitées pendant l'épisode
3. une dernière où l'on va améliorer la politique

Le principe de la méthode Monte-Carlo est donné dans le schéma ci-dessous [5] :

FIGURE 4 – schéma du fonctionnement de base de la méthode de Monte-Carlo

Ce schéma nous montre que, une fois que nous avons évalué la fonction de valeur correspondant à la politique qui a été utilisée pour générer l'épisode, nous allons calculer une nouvelle politique. Pour estimer la fonction de valeur obtenue après réalisation de l'épisode, on calcul la moyenne du cumule des récompenses (nous nous intéresserons ici à un problème de contrôle pour illustrer la différence entre *on-policy* et *off policy*).

Méthode on-policy :

Dans la mesure où nous connaissons l'ensemble des récompenses qui ont été récupérées lors de la réalisation de l'épisode, nous allons calculer la fonction de récompense à partir de :

$$G_k(S_t, A_t) = \sum_{i=t}^{T} \gamma^i R_i$$

Avec G_k la somme des récompenses cumulées lors de l'épisode généré à l'instant k à partir de la paire (S_t, A_t) et en suivant la politique $\pi(A_t|S_t)$. A partir de cette expression nous pouvons définir la fonction de valeur dans la méthode de Monte-Carlo [5] :

$$Q(S_t, A_t) = \frac{1}{T} \sum_{k=1}^{T} G_k$$

La fonction de valeur représente la moyenne du cumule des récompenses pour les trajectoires générées à partir de S_t. Pour éviter d'avoir à stocker l'ensemble des récompenses, on utilise la mise à jour incrémentale de Monte Carlo qui est donnée par :

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$$

Avec $G_t = R_t + \gamma R_{t+1} + ... + \gamma^n R_{T-1}$, la somme des récompenses totales depuis l'état S_t jusqu'à l'état final pour l'épisode généré à l'instant t. Alpha correspond au facteur d'apprentissage, qui permet de prendre plus ou moins en compte l'apprentissage acquis lors de la transition. La différence $(G_t - Q(S_t, A_t))$ mesure l'erreur entre la moyenne des récompenses $Q(S_t, A_t)$ et la véritable somme des récompenses, obtenue lors de la réalisation de l'épisode (l'objectif de la mise à jour étant de réduire cette différence).

Une fois les Q-valeurs calculées, nous pouvons déterminer une meilleure politique à partir de celles-ci :

$$\pi'(A_t|S_t) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(S_t)|} & \text{si } a = A^* \\ \frac{\epsilon}{|A(s)|} & \text{sinon} \end{cases}$$

avec $A^* = \underset{a}{argmax} Q(S_t, A_t)$ l'action permettant de maximiser la fonction de valeur, $\epsilon \ll 0$ un paramètre fixé par l'utilisateur et $|A(S_t)|$ le nombre d'action possible dans l'état courant. Cette politique est appelée ϵ-greedy policy car elle va assigner une forte probabilité à l'action permettant de maximiser le cumule des récompense dans l'état S_t et une faible probabilité à toutes les autres. De cette façon, l'action A^* va être tirée par la politique avec une grande probabilité dans la phase de génération de l'épisode car c'est elle qui correspond au comportement optimal (dans l'état de la politique). La probabilité est uniformément répartie sur le reste des actions possible et $\pi(a|s) > 0 \ \forall a \in A$ permet à l'agent de continuer d'explorer son environnement. La trajectoire pour l'épisode suivant sera générée à partir de la nouvelle politique, déduite de la fonction de valeur.

méthode off-policy :

La méthode de Monte-Carlo off-policy suit le même principe de base que la méthode on-policy, c'est à dire que l'apprentissage est effectué à la fin de l'épisode. La politique courante qui est utilisée pour générer les transitions peut être une ϵ-geedy policy (ou tout autre soft-policy qui permet d'assigner une probabilité non nulle de tirage pour toutes les actions possible dans un état).

Enfin, pour améliorer la politique $\pi(a|s)$, on prend l'action qui permet de maximiser la fonction de valeur pour chaque état S_t qui ont été parcouru pendant l'épisode : $\pi(A_t|S_t) = \underset{a}{argmax} Q(A_t, S_t)$.

Les méthodes de mises à jour par Monte-Carlo ont permis de résoudre bon nombre de problèmes de décisions mais comportent un principal désavantage : il faut attendre que l'agent ait terminé un épisode pour pouvoir commencer les mises à jour. Cette méthode est donc relativement lente pour converger vers la politique optimale. Cela est d'autant plus vrai si la tâche à effectuer est très longue.

4.1.2 TD-learning

Cette méthode de mise à jour est liée à la méthode de Monte-Carlo dans le sens où l'on va tenter de réduire l'erreur entre l'estimation du cumule des récompenses et une valeur cible. Néanmoins, au lieu de différer l'apprentissage à la fin de l'épisode, nous allons maintenant effectuer les mises à jour de la fonction de valeur à partir de l'expérience acquise lors d'une transition (c'est pourquoi on parle d'algorithme "on-line"). La fonction de valeur a donc pour expression [5] :

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

On retrouve $Q_(S_t, A_t)$ la connaissance qui avait été acquise lors des mises à jours précédentes tandis que la seconde partie de l'équation représente l'expérience nouvelle. On a $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$ une estimation des récompenses futures dans l'état suivant, corrigée par la récompense R_{t+1}. L'objectif de la mise à jour va donc être de réduire l'écart entre la fonction de valeur courante $Q(S_t, A_t)$ et la valeur cible $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$.

Ici nous nous sommes intéressé au TD-learning qui met à jour sa fonction de valeur dès qu'une nouvelle transition est effectuée par l'agent. Or il est également possible de différer cette mise à jour pour un certain nombre de transition, on appelle cette méthode le *n-step TD-learning*. L'expression de la mise à jour est donnée ci-dessous [9] [5] :

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_t + \gamma R_{t+1} + ... + \gamma^n R_{t+n} + \gamma Q(S_{t+n+1}, A_{t+n+1}) - Q(S_t, A_t))$$

Là où le TD(0) estime les récompenses futures à travers $R_{t+1} + \gamma V(S_{t+1}$, le TD(n) permet au contraire d'avoir une meilleure estimation de cette valeur cible, corrigée par les récompenses obtenues lors des n prochain états. Cette différence majeure permet au n-steps TD learning d'obtenir un apprentissage plus qualitatif en aillant une meilleure estimation de la valeur cible. On peut aussi remarque que si n est suffisamment grand pour que l'agent ait le temps de finir sa tâche, le n-step TD-learning sera équivalent à la méthode de Monte-Carlo. Il s'agit donc d'une variante permettant d'unifier le TD-learning et la méthode Monte-Carlo.

Ces deux méthodes de mises à jour représentent le socle de bon nombre d'algorithme dans le cas tabulaire mais comme nous l'avons expliqué précédemment, ce type de résolution est limité au cas d'environnement discret de petite taille. C'est pourquoi, dans la suite de cette partie, nous allons nous focaliser sur les méthodes de mises à jour d'algorithmes qui reposent sur une architecture d'approximation.

4.2 Descente de gradient stochastique, mise à jour avec architecture d'approximation

Cette mise à jour est utilisée dans le cas où une architecture d'approximation a été appliquée à l'environnement de départ (voir partie 3.3). On cherche donc a calculer une valeur approchée de la politique, ou de la fonction de valeur, se rapprochant le plus possible de la fonction optimale. Pour cela, on va chercher à minimiser l'erreur quadratique moyenne [5] [9] [10] :

$$MSVE(\theta) = \frac{1}{2}\mathrm{E}[Q^{\pi}(s,a) - Q(s,a;\theta)]^2$$

Pour pouvoir minimiser l'erreur entre la valeur cible ($Q^{\pi}(s,a)$) et la valeur approchée que l'on cherche à optimiser $Q(s,a;\theta)$, on va appliquer l'algorithme de la descente de gradient. Cette méthode consiste a ajuster le vecteur de poids θ selon la direction qui permet de minimiser l'erreur :

$$\theta_{t+1} = \theta_t - \frac{1}{2}\alpha\nabla(Q^{\pi}(S_t,A_t) - Q(s,a;\theta_t))^2$$
$$\theta_{t+1} = \theta_t + \alpha(Q^{\pi}(S_t,A_t) - Q(S_t,A_t;\theta))\nabla_{\theta}Q(S_t,A_t;\theta_t)$$

Cette méthode de mise à jour nécessite donc de connaître la dérivée partielle de la fonction $Q(s,a;\theta$ pour chaque poids θ_i. Dans l'expression ci-dessous, on utilise $Q^{\pi}(s,a)$ comme valeur cible, alors que cette fonction n'est pas connue. Pour pouvoir appliquer cette mise à jour, il va donc falloir utiliser une autre fonction pour pouvoir minimiser l'erreur quadratique moyenne. La plupart du temps, c'est l'estimation du cumule des récompenses dans l'état suivant (corrigé par la récompense récupérée lors de la transition) qui fera office de valeur cible :

$$\theta_{t+1} = \theta_t + \alpha(R_{t+1} + \gamma Q(S_{t+1},A_{t+1};\theta_t) - Q(S_t,A_t;\theta_t))\nabla_{\theta}Q(S_t,A_t;\theta_t)$$

L'objectif de cette mise à jour va donc être de minimiser l'erreur quadratique moyenne entre $R_{t+1} + \gamma Q(S_{t+1},A_{t+1})$, l'estimation du cumule des récompenses dans l'état S_{t+1} corrigée par la récompense R_{t+1}, et la valeur courante $Q(S_t,A_t;\theta_t)$.

Dans cette partie nous n'avons pour le moment traité uniquement des règles de mises à jour orientées value-based, ors il existe aussi toute une gamme d'algorithme dont la mise à jour se base directement sur la politique (comme nous l'avons introduit dans la partie 3.2). Ces méthodes ont pour avantage de faire varier lentement le comportement de l'agent, contrairement aux méthodes value-based où la politique est évaluée à chaque mise à jour. Les algorithmes policy-based sont souvent plus performant pour des situations où l'on veut converger vers la politique optimale de façon continue.

4.3 Les règles de mises à jour policy-based avec architecture d'approximation

Dans cette partie nous allons nous intéresser à des méthodes de mises à jour de la politique, paramétrée par une architecture d'approximation, afin de s'approcher de l'optimale. Comme pour la descente de gradient où nous cherchions à minimiser l'erreur quadratique moyenne, il va falloir définir un critère de performance adapté aux méthodes policy-based [5] : $\eta(\theta) = V_{\pi_\theta}(S_0)$. Avec $v_{\pi_\theta}(S_0)$ qui donne une estimation de la qualité de la politique π_θ en partant de l'état S_0.

La mise à jour de la politique va avoir pour objectif de maximiser le critère de performance. Pour cela, on va utiliser l'algorithme de montée de gradient dans la direction de $\eta(\theta)$, ce qui nous donne la règle de mise à jour générale du vecteur de poids θ :

$$\theta_{t+1} = \theta_t + \alpha \nabla \widehat{\eta_{\theta_t}}$$

Avec $\widehat{\eta_{\theta_t}}$ une estimation du critère de performance. Le but de cette section va être de présenter plusieurs types de mise à jour permettant de calculer la valeur approchée de la politique.

4.3.1 REINFORCE Monte-Carlo policy gradient

La première règle de mise à jour de la politique que nous allons détailler se base sur la méthode de Monte-Carlo (voir 4.1.1) et sur le théorème de "policy-gradient" pour nous donner l'expression de l'estimation du critère de performance [11] :

$$\nabla \eta_{\theta_t} = \mathrm{E}_{\pi_\theta}[\gamma^t G_t \nabla_\theta \pi(A_t|S_t;\theta_t)]$$

Ce qui nous permet enfin d'arriver à la l'expression de la mise à jour du vecteur de poids θ (en modifiant légèrement l'expression du critère de performance) :

$$\theta_{t+1} = \theta_t + \alpha \gamma^t G_t \nabla_\theta \log(\pi(A_t|S_t;\theta_t))$$

Cette expression nous montre que le vecteur θ_{t+1} va être mis à jour dans la direction du gradient de la politique, proportionnellement à la valeur G_t qui correspond au cumule des récompense obtenu à la fin de la réalisation d'une tâche épisodique (en partant de l'état S_t jusqu'à l'état final). Cela a pour effet de favoriser le choix de l'action A_t qui permet d'obtenir la plus grande valeur de retour (G_t) dans l'état S_t lors du prochain épisode.

Nous avons vu les mises à jours value-based, tabulaires et approchées, nous avons également étudier une première règle de mise à jour policy-based avec le Reinforce Monte Carlo policy gradient. Nous allons maintenant nous intéresser au seul type de mise à jour que nous n'avons pas encore introduit : la méthode *acteur-critique*.

4.3.2 REINFORCE with Baseline et acteur-critique

Dans les méthode de policy-gradient comme REINFORCE que nous avons étudier juste avant, la mise à jour du vecteur de poids est effectuée dans le sens du gradient de la politique, proportionnellement à la valeur de retour G_t. Comme cette valeur dépend de l'ensemble des récompenses futures, elle va beaucoup varier pour chaque épisode, même pour des politiques similaires (dans le cas de la réalisation d'une tâche complexe). Cela induit de la variance dans l'apprentissage car le vecteur de poids varie beaucoup à chaque mise à jour. Pour diminuer la variance, au lieu d'utiliser la seule valeur de retour pour la mise à jour, on va calculer la différence $G_t - V(S_t;w_t)$. Comme $V(S_t;w_t)$ représente l'estimation de la récompense totale, la différence $G_t - V(S_t;w_t)$ va donc traduire la qualité de la politique dans l'état courant, sans avoir l'influence de l'ensemble des états futures, ce qui va avoir pour effet de diminuer la variance. Cela nous amène à une nouvelle mise à jour du vecteur de poids [5] :

$$\theta_{t+1} = \theta_t + \alpha\gamma(G_t - V(S_t; w_t))\nabla_\theta \log(\pi(A_t|S_t; \theta_t))$$

Il est a noter que $V(S_t, w_t)$ nécessite l'utilisation d'un deuxième vecteur de poids w, cette méthode utilise donc deux réseaux de neurones : un premier pour la politique et le second pour la fonction de valeur. La mise à jour de la fonction de valeur est effectuée selon la méthode de la descente de gradient stochastique nous que avons déjà détaillé (en utilisant bien sûr G_t comme valeur cible pour minimiser l'erreur quadratique).

Dans cette méthode de REINFORCE with Baseline, $V(S_t; w_t)$ n'est pas utilisée pour la prédiction. On va donc devoir attendre la fin de l'épisode pour pouvoir commencer l'apprentissage, en se basant sur la valeur de retour G_t. Comme pour la méthode de Monte-Carlo classique, cela va avoir un impact important sur le temps de convergence vers la politique optimale. Pour pallier à cela, nous allons maintenant étudier une méthode qui permet de réaliser un apprentissage à chaque intervalle de temps en utilisant une prédiction sur la récompense totale.

Acteur critique Ici, l'objectif est d'être capable de mettre à jour le vecteur de poids θ à chaque intervalle de temps. Pour cela, au lieu d'attendre d'avoir la valeur de retour G_t à la fin de l'épisode, on va utiliser une estimation de la récompense totale : $R_{t+1} + V(S_{t+1}; w_{t+1})$. La mise à jour va donc avoir pour expression [2] :

$$\theta_{t+1} = \theta_t + \alpha\gamma(R_{t+1} + \gamma V(S_{t+1}; w_{t+1}) - V(S_t; w_{t+1}))\nabla_\theta \log(\pi(A_t|S_t; \theta_t))$$

On retrouve dans cette équation la mise à jour utilisée pour le TD-Learning que nous avons déjà rencontré. De plus, on pourra noter que $V(S_t; w_{t+1})$ est paramétré par w_{t+1}. Cela s'explique par le fait que le vecteur de poids w est mise à jour avant θ.

L'ensemble des règles de mise à jour que nous avons vu dans cette partie ne constitue qu'une étape de l'apprentissage par renforcement. Nous allons donc maintenant étudier quelques algorithmes qui permettent de résoudre des problèmes de décision séquentiel en se basant sur les différentes propriétés algorithmiques de la partie 3 et les règles de mises à jour que nous venons d'introduire.

5 Présentation d'algorithmes

Dans cette section nous allons nous intéresser au fonctionnement de quelques algorithmes. Nous verrons que leurs structures globale sont similaires et traduisent les thématiques que nous avons abordé dans la partie 1 (répétition de la tâche à effectuer, interaction avec l'environnement, etc...). En effet, dans les algorithmes d'apprentissage par renforcement, on retrouve certaines étapes communes :

1. une phase d'initialisation :
   — de la fonction de valeur ou de la politique (sur toutes paires état-action pour les méthodes tabulaires)
   — du vecteurs de poids (des vecteurs de poids pour les méthodes acteur-critique)
   — des différents paramètres $(\alpha, \gamma, \epsilon, ...)$
2. Répétition de la tâche épisodique (indéfiniment ou pour un certain nombre d'épisode) jusqu'à l'état final (ou que l'utilisateur mette fin pour les tâches continues)
3. initialisation de l'état S_0 pour les tâches épisodiques
4. pour chaque instant t (i.e transition) on va effectuer un certain nombre d'étapes :
   — choix d'une action (selon la politique)
   — interaction avec l'environnement (i.e. transition vers un état S_{t+1})
   — on observe la récompense R_t correspondante
   — mise à jour :
      — de la fonction de valeur
      — de la politique
      — du vecteur de poids
      — etc...
   — étapes supplémentaires en fonction de l'algorithme
5. on recommence la tâche en reproduisant les mêmes étapes.

5.1 Quelques algorithmes tabulaires

Monte-Carlo Nous avons détaillé la méthode Monte Carlo précédemment, la spécificité de cet algorithme étant de générer un épisode à partir de la politique, de récupérer la valeur de retour G_t (cumule des récompenses). Enfin il faut mettre à jour la fonction de valeur n fonction d'une des deux méthodes que nous avons détaillé dans la partie 4.1.1 (en fonction que l'on soit dans une démarche off-policy ou on-policy).

SARSA [5] Cet algorithme se base sur le TD-Learning pour mettre à jour sa fonction de valeur. Le TD-learning, que nous avons introduit dans la partie 4.1.2 sous la forme d'un problème de prédiction (avec $V(S_t)$) peut être également utilisé pour des problèmes de contrôles. La mise à jour de SARSA est donnée par :

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

La valeur $Q(S_{t+1}, A_{t+1})$ est donnée par la politique courante. SARSA est en effet un algorithme on-policy, c'est à dire que la politique est à la fois utilisée pour choisir l'action A_t et pour estimer le cumule des récompenses attendu (i.e $R_{t+1}+\gamma Q(S_{t+1}, A_{t+1})$). Il faut que cette politique soit une "soft-policy" (comme ϵ-greedy que nous avons détaillé dans la partie 4.1.1) pour garantir l'exploration.

Q-learning [14] Cet algorithme correspond à la version off-policy de SARSA. La seule différence est que nous n'allons pas utilisé la même politique pour le choix des action et pour l'estimation du cumule des récompenses. Pour le choix des actions, comme pour SARSA, il faut utiliser une "soft-policy" pour être sûr de parcourir l'environnement. Une seconde politique ("greedy-policy") est utilisée pour l'estimation. Cela nous donne la mise à jour de la fonction de valeur :

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \underset{a}{argmax} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

Pour l'estimation du cumule des récompenses, on utilise donc la fonction de valeur maximale dans l'état suivant, corrigée par la récompense R_{t+1}. Le principe étant, comme toujours, de mettre à jour la fonction de valeur afin de réduire l'erreur entre la fonction de valeur courante et la valeur cible.

Pour ces deux méthodes il est possible d'utiliser une architecture d'approximation et d'effectuer la mise à jour sur le vecteur de poids (en utilisant la méthode de la descente de gradient stochastique, voir partie 3.3). De même il suffit de reprendre l'expression du n-step TD-Learning (voir partie 4.1.2) pour obtenir le n-step SARSA et n-step Q-Learning (il faut bien sur stocker les n transitions en mémoire pour effectuer les mises à jour). Dans la partie suivante nous allons étudier deux algorithmes de référence dans la littérature : le "Deep Q-Network" (DQN) et le "Deep deterministic Policy Gradient" (DDPG). Ces deux algorithmes utilisent un réseau de neurones pour obtenir le vecteur de poids. Nous allons discuter de leurs principales caractéristiques et de leur principe de fonctionnement général.

5.2 Deep Q-Network

Cet algorithme [4] est une extension du Q-Learning qui utilise un réseau de neurones comme architecture d'approximation pour représenter les états. La mise à jour s'effectue donc sur le vecteur de poids selon la méthode de descente de gradient stochastique :

$$\theta_{t+1} = \theta_t + \alpha \delta_t \nabla_\theta Q(S_t, A_t; \theta_t)$$

$$\delta_t = R_{t+1} + \gamma \underset{a}{argmax} Q(S_{t+1}, A_{t+1}, \theta^-) - Q(S_t, A_t; \theta_t)$$

La seule différence notable par rapport à la mise à jour que nous avons détaillé dans la partie 4.2 est : $Q(S_{t+1}, A_{t+1}, \theta^-)$. Pendant un certain nombre de mise à jour (fixé par l'utilisateur, on va utiliser la fonction de valeur paramétrée par θ^- comme valeur cible. Cela permet de résoudre des problèmes de stabilité dans les mises à jour. En effet, si la valeur cible qui est utilisée pour minimiser l'erreur quadratique moyenne varie à chaque mise à jour, on observe un problème

de stationnarité. Pour résoudre cela on va prendre une copie du vecteur de poids à un instant t et utiliser la fonction de valeur ainsi figée pour mettre à jour le vecteur de poids sur un certain nombre de transition.

La seconde spécificité de cet algorithme est d'utiliser une mémoire pour stocker les transitions. En effet une des conditions de convergence du Q-Learning approché est d'utiliser des échantillons i.i.d (indépendants et identiquement distribués) entre deux mise à jour. Ors, les transitions récupérées entre deux instants t et $t+1$ sont fortement corrélées entre elles, de part la nature de la tâche à réaliser (l'environnement évolue de façon continu). Pour obtenir des échantillons i.i.d on va stocker les transitions dans une file pour les utiliser plus tard. Lorsque la mise à jour est effectuée, on prend aléatoirement l'une des transitions dans la mémoire. La véritable mise à jour du DQN à l'instant t sera donc [3] :

$$\theta_{t+1} = \theta_t + \alpha\delta_t \nabla_\theta Q(S_j, A_j; \theta)$$
$$\delta_t = R_{j+1} + \gamma \underset{a}{argmax} Q(S_{j+1}, A_{j+1}; \theta^-) - Q(S_j, A_j; \theta_t)$$

avec $(S_j, A_j, S_{j+1}, R_{j+1}$ la transition qui a été stockée dans la mémoire à l'instant j.

Une dernière méthode pour améliorer le Q-Learning approché (et qui peut aussi être utilisée pour sa résolution tabulaire) est d'utiliser deux réseaux de neurones pour avoir deux fonctions de valeurs Q_1 et Q_2 [13]. Dans le Q-Learning, on utilise $\underset{a}{argmax} Q(S_{t+1}, A_{t+1})$ pour l'estimation du cumule des récompenses. Cela entraîne une surestimation de la fonction de valeur qui ne poserait pas de problème si elle était uniforme pour toutes les paires état-action. Ors cette surestimation est variable, ce qui induit un biais important dans la politique optimale estimée.

Pour pallier à ce problème on va utiliser la fonction de valeur Q_1 comme valeur cible dans la mise à jour de Q_2 et inversement. On aura donc deux mises à jours différentes, une pour chaque réseau de neurones :

$$\theta^{(1)}_{t+1} = \theta^1_t + \alpha\delta^{(1)}_t \nabla_\theta Q_1(S_j, A_j; \theta^{(1)}_t)$$
$$\delta^{(1)}_t = R_{j+1} + \gamma \underset{a}{argmax} Q_2(S_{j+1}, A_{j+1}, \theta^{(2-)}_t) - Q_1(S_j, A_j; \theta^{(1)}_t)$$
$$\theta^{(2)}_{t+1} = \theta^1_t + \alpha\delta^{(2)}_t \nabla_\theta Q_2(S_j, A_j; \theta^{(2)}_t)$$
$$\delta^{(2)}_t = R_{j+1} + \gamma \underset{a}{argmax} Q_1(S_{j+1}, A_{j+1}, \theta^{(1-)}_t) - Q_2(S_j, A_j; \theta^{(2)}_t)$$

Comme la fonction de valeur Q_1 est issue de transitions différentes que Q_2 l'estimation sur le cumule des récompenses peut être considérée comme non-biaisée, ce qui permet d'améliorer grandement les performances du DQN. A chaque instant t nous allons mettre à jour l'un ou l'autre des vecteurs de poids (avec une probabilité de 0.5). Les actions qui sont choisies a chaque itération de l'algorithme (hormis lorsque l'on choisit de prendre une action aléatoire) correspondent au maximum de $Q_1(S_t, A_t; \theta^{(1)}_t)$ et $Q_2(S_t, A_t; \theta^{(2)}_t)$.

Cet algorithme est très efficace pour des espaces d'action discret (c'est à dire fini) car dans ce cas il est simple de trouver l'action qui permette de maximiser la fonction de valeur dans l'état suivant. Pour un espace d'action continu cette opération est plus difficile et demande un temps de calcul considérable. Cela est d'autant plus contraignant que cette recherche du maximum s'effectue à chaque

mise à jour. Le prochain algorithme que nous allons étudier permet de lever cette limitation.

5.3 Le Deep Deterministic Policy Gradient

Cet algorithme [1] va avoir pour but de déterminer le maximum de la fonction de valeur dans l'état suivant (afin d'estimer le cumule des récompenses attendues jusqu'à la fin de l'épisode). Pour cela, nous allons chercher une politique permettant d'estimer :

$$\mu(s;\theta) \approx \underset{a}{argmax} Q(s,a;w)$$

Pour trouver le maximum de $Q(s,\mu(s;\theta);w)$, on va utiliser l'algorithme de la montée de gradient (que nous avons étudier dans la partie 4.3.) sur la fonction de valeur [7] :

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta Q(S_t, \mu(S_t;\theta_t); w_t)$$

en utilisant le théorème de dérivation de fonctions composées on obtient :

$$\theta_{t+1} = \theta_t + \alpha(\nabla_\mu Q(S_t, \mu(S_t;\theta_t); w_t) \nabla_\theta \mu(S_t;\theta_t))$$

Cette équation entraîne la politique μ à estimer l'action permettant de maximiser la fonction de valeur pour un état donné. Enfin, les Q-Valeurs vont être mise à jour selon la même méthode que le DQN :

$$w_{t+1} = w_t + \alpha \delta_t \nabla_w Q(S_j, \mu(S_j;\theta_t); w_t)$$
$$\delta_t = R_{j+1} + \gamma Q(S_{j+1}, \mu(S_t;\theta'_t), w'_t) - Q(S_j, \mu(S_t;\theta_t); w_t)$$

Rappelons que cette mise à jour permet de minimiser l'erreur quadratique entre la valeur cible, estimation du cumule des récompenses dans l'état suivant, corrigée par R_{t+1} et paramétré par w', le vecteur de poids "figé". On remarquera que, comme pour le DQN, on utilise la transition récupéré à l'instant j pour mettre à jour le vecteur de poids à l'instant t (il faut donc aussi utiliser une mémoire pour stocker les transitions).

Comme pour le DQN, il va falloir utiliser une valeur cible pour pouvoir mettre à jour la fonction de valeur. Dans cet algorithme, au lieu de prendre une valeur figée du réseau de neurones, nous allons faire en sorte que les vecteurs de poids des valeurs cibles évoluent lentement [7] :

$$w'_{t+1} = \tau w_t + (1-\tau) w'_t$$

$$\theta'_{t+1} = \tau \theta_t + (1-\tau) \theta'_t$$

avec $\tau \ll 1$, θ' et w' les vecteurs de poids cibles. Faire évoluer lentement ces vecteurs est une condition nécessaire, comme pour le DQN, pour que la fonction de valeur soit stable. Il en résulte néanmoins un ralentissement de l'apprentissage.

Dans le DQN l'action qui est choisie dans l'état courant est déterminer soit de façon aléatoire avec une probabilité ϵ, soit à partir de $\underset{a}{argmax} Q(s,a;w)$. Pour choisir les actions dans le DDPG, on va utiliser la politique μ. Ors nous savons que cette fonction va tendre vers une politique déterministe au fur et à mesure des mises à jour. Pour garantir l'exploration, on va rajouter du bruit dans la politique :

$$A_t = \mu(S_t; \theta_t) + n$$

avec n un bruit blanc gaussien, afin de continuer de choisir des actions qui ne correspondent pas au comportement optimal.

A ma connaissance il n'existe pas d'algorithme combinant la méthode de DDPG et du double Q-Learning. Pourtant la mise à jour sur la fonction de valeur utilise une approximation du maximum de la fonction de valeur dans l'état S_{t+1}. Comme nous l'avons remarquer dans le DQN, cela a pour effet de surestimer la fonction de valeur et donc l'estimation du cumule des récompenses ce qui induit un biais dans la politique apprise. Il pourrait donc s'agir d'une amélioration de cette technique qui n'a pas encore été explorée. Cette méthode devrait néanmoins demander une importante puissance de calcul car il faudrait utiliser deux réseaux de neurones, un pour chaque fonction de valeur.Il s'agit tout de même d'une piste intéressante pour poursuivre cet état de l'art...

6 Conclusion

Nous avons put voir au cours de cet état de l'art comment les scientifiques ont réussi à traduire les concepts fondamentaux de l'apprentissage au travers des algorithmes que nous avons étudier. La répétition de la tâche à effectuer, le retour d'expérience et l'amélioration du comportement de l'agent sont autant de problématiques que permettent de résoudre ces algorithmes, avec une modélisation adéquate. L'apprentissage par renforcement est un domaine de recherche très prometteur, tant sur le plan théorique d'applicatif. En effet, certain problèmes de navigation autonome utilisent déjà le Reinforcement Learning combiné avec des architectures de réseaux de neurones supervisés (pour interpréter l'environnement dans lequel l'agent évolue).

Références

[1] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv :1509.02971*, 2015.

[2] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv :1312.5602*, 2013.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540) :529, 2015.

[5] Andrew G. Barto Richard S. Sutton. *Reinforcement Learning : An introduction*. Second edition, 2016.

[6] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv :1712.01815*, 2017.

[7] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[8] Richard S Sutton. *Introduction to reinforcement learning*, volume 135.

[9] Richard S Sutton, Hamid R Maei, and Csaba Szepesvári. A convergent $o(n)$ temporal-difference algorithm for off-policy learning with linear function approximation. In *Advances in neural information processing systems*, pages 1609–1616, 2009.

[10] Richard S Sutton, Hamid Reza Maei, Doina Precup, Shalabh Bhatnagar, David Silver, Csaba Szepesvári, and Eric Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 993–1000. ACM, 2009.

[11] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[12] John N Tsitsiklis and Benjamin Van Roy. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081, 1997.

[13] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 2, page 5. Phoenix, AZ, 2016.

[14] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4) :279–292, 1992.