UPMC, M1 INFORMATIQUE

Projet: Amélioration d'un algorithme d'apprentissage par renforcement profond

YOU Jiang, SPORTICH Benjamin

Encardant: Olivier Sigaud, olivier.sigaud@isir.upmc.fr

May 15, 2017

Contents

1 Introduction 2

2 Préambule théorique 2

3 DDPG 9

4 Amélioration de DDPG 12

5 Problème : MountainCarContinuousVO 14

6 Tests de performance 15

7 Conclusion et travail futur 19

1 Introduction

L'apprentissage par renforcement est un des nombreux modèles de machine learning étudiés dans le champ de l'intelligence artificielle et celui qui a démontré la plus grande progression ces dernières années.
En effet, une fois le succès de *Deep Q-Networks* lancé [6], le monde de la recherche réalisa que l'apprentissage par renforcement était adapté à résoudre les problèmes de grande dimension, surtout que la communauté scientifique se heurtait au fait que dans la robotique, les espaces de travail sont continus. Logiquement, une discrétisation trop précise fait retomber dans le même problème de dimension qu'évoqué mais une trop naïve occasionne une perte d'information trop importante. Google DeepMind a alors fourni un algorithme solide raisonnant dans les espaces continus : $DDPG$ ou *Deep Deterministic Policy Gradient* [7] qui repose en grande partie sur les travaux sur *DPG (Deterministic Policy Gradient)* et *DQN*. Cet algorithme est notamment connu pour avoir obtenu de biens meilleurs scores que tous les autres algorithmes d'apprentissages sur des jeux ATARI et parfois même meilleurs que celui d'un expert humain. Le but de nos travaux sera d'implémenter une amélioration d'une variante de *DQN* (qui battait largement l'algorithme original de *DQN* sur 41 des 49 jeux ATARI sur lesquels le test a eu lieu) dans *DDPG* pour stabiliser l'algorithme et améliorer ses performances. Cette amélioration se nomme « Prioritized Experience Replay » et est développée dans le papier suivant pour *DQN* [5].

2 Préambule théorique

Nous allons commencer par un rappel des notions utilisées dans le problème suivant pour comprendre en profondeur les raisons du choix de l'algorithme, son fonctionnement, les améliorations potentielles et celles qui ont été implémentées.

Apprentissage par renforcement :

L'apprentissage par renforcement référence un problème dont le but est d'apprendre, à partir d'expériences, ce qu'il convient de faire en différentes situations, de façon à optimiser une récompense quantitative au cours du temps. En général, on considère un agent autonome plongé au sein d'un environnement avec qui il est en interaction permanente. L'agent doit prendre des décisions en fonction de son état courant et en fonction de l'action exécutée, l'environnement procure à l'agent une récompense ou une punition, qui permet à celui-ci d'adapter son comportement en conséquence. L'agent cherche donc à maximiser ses récompenses et le comportement décisionnel optimal associé est appelé stratégie ou politique.

Le problème est modélisé par un Processus Décisionnel Markovien :
Processus décisionnel de Markov

- S : Espace des états possibles du système. Cet espace peut être fini, discret ou continu.
- A : Espace des actions possibles dans le système. Cet espace peut être fini, discret ou continu.
- $T : S \times A \times S \rightarrow [0;1]$: fonction de transition ; $T(s,a,s')$ représente la probabilité d'atteindre l'état s' en partant de l'état s et en exécutant l'action a. Dans le cas déterministe, cette fonction ne prend ses valeurs que dans $0,1$
- $r : S \times A \rightarrow$: fonction qui associe une récompense à un couple état-action

Le formalisme des processus décisionnels Markoviens définit $s_{(t+1)}$ et $r_{(t+1)}$ en $f(s_t, a_t)$ c'est à dire que l'état à t+1 et la récompense à t+1 sont des fonctions de l'état et de l'action à l'instant t. Le modèle sous forme de processus décisionnel Markovien est valide car : chaque état contient toutes les informations nécessaires, les transitions ne dépendent pas du temps et l'état suivant ne dépend de l'action que d'un unique agent.

Fonction de valeur

- Une politique ou stratégie est une fonction $\pi : S \rightarrow A$. Dans le cas déterministe c'est un vecteur avec une action par état.
- La fonction de valeur $V^{\pi} : S \rightarrow R$ représente à partir de chaque état le cumul des utilités futures en suivant π
- La fonction de valeur d'action $Q^{\pi} : S \times A \rightarrow R$ représente le cumul des utilités au long terme de faire chaque action en chaque etat (puis en suivant π). C'est une matrice avec une valeur par état et par action.

Objectif : Trouver une politique qui maximise le cumul des utilités sur le long terme.

On obtient une récompense pour chaque action dans les états. Ainsi à l'étape t on obtient la récompense r_t.
Pour calculer le cumul des utilités, on utilise le critère actualisé (γ-pondéré) agrégeant les utilités immédiates de r. Il est défini de la façon suivante :

$$\begin{cases} (s_0) = r_0 + \gamma V(s_1) \\ V^{\pi}(s_{t_0}) = \sum_{t=t_0}^{\infty} \gamma^t r(s_t, \pi(s_t) \end{cases}$$

Avec $\gamma \in [0;1]$, le facteur d'actualisation :
- si $\gamma = 0$, seule l'utilité immédiate est importante,
- si $\gamma = 1$, les utilités futures sont aussi importantes que l'utilité immédiate.
Dans le cas présent, on définira $\gamma = 0.99$ car on veut avoir une vision lointaine.

- Cas stochastique :

$$V^{\pi}(s) = \sum_{a} \pi(s,a) + [r(s,a) + \gamma \sum_{s'} P(s'|s,a) V^{\pi}(s')]$$

- Cas déterministe :

$$V^{\pi}(s) = r(s,\pi(s)) + \gamma \sum_{s'} P(s'|s,a) V^{\pi}(s')$$

On rappelle que

$$T(s_t, a_t, s_{t+1}) = P(s'|s,a)$$

On rappelle que le cas déterministe est un cas particulier du stochastique.

On appelle fonction de valeur optimale notée V^* la fonction de valeur associée à la politique qui rapporte le plus.

Famille de methodes

- *Critique* : fonction de valeur d'action → évaluation de la politique
- *Acteur* : la politique elle même
- *Value iteration* est une méthode "critique pur" : elle itère sur la fonction de valeur (d'action) jusqu'à convergence puis elle en déduit une politique optimale. C'est à dire elle va rechercher V^* en mettant à jour V en cherchant le choix d'utilité maximum (en choisissant la meilleure action possible) qu'elle back-propagera jusqu'à convergence.
- *Policy iteration* est une méthode "acteur-critique" : elle met à jour en parallèle une politique et une fonction de valeur (d'action)
- Dans le cas continu (a.k.a notre cas), on s'intéresse aussi aux méthodes "acteur pur" : descente de gradient de performance sur les politiques

En programmation dynamique les fonctions r et T (respectivement de récompense et de transition) sont données. Le but de l'apprentissage par renforcement est de construire la politique optimale π^* en ne connaissant a priori ni r ni T :

- *Apprentissage par Renforcement Direct (model-free)* : Construire π^* directement sans construire ni r ni T
- *Acteur-critique* : Cas particulier d'apprentissage par renforcement direct
- *Apprentissage par Renforcement Indirect (model based)* : Construire un modèle de r et T et s'appuyer sur cette connaissance pour calculer π^*

Erreur de différence temporelle

Si les estimations de $V(s_t)$ et de $V(s_{t+1})$ se révèlent exactes, on aurait :

$$V(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} ...$$

$$V(s_{t+1}) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} ...$$

Donc

$$V(s_t) = r_t + \gamma V(s_{t+1})$$

L'erreur de différence temporelle se définit donc de la façon suivante : $\lambda_k = r_k + \gamma V(s_{k+1}) - V(s_k)$
Elle mesure l'erreur entre les valeurs calculées des estimations de $V(s_k)$ et les valeurs qu'elles devraient avoir.

Pour corriger cette erreur, il faut faire évoluer $V(s_k)$ dans le sens de λ_k. C'est à dire si la valeur de $_k$ est positive, on augmente V et si elle est négative, on décrémente V :

$$V(s_t) = V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

avec α le coefficient d'évolution ou d'apprentissage.

Q-learning

Tous les algorithmes qui travaillent sur la fonction V peuvent être étendus à la fonction Q, cela permet de choisir l'action.
On a alors $Q^* = TQ^*$

Pour chaque sample $(s_t, a_t, r_{t+1}, s_{t+1})$, on a :

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Sauf que cette méthode (on-policy) bloque l'exploration : on est obligé de connaître a_{t+1}. La convergence est plus compliquée.
On remplace donc $Q(s_{t+1}, a_{t+1})$ par $max_a Q(s_{t+1}, a_{t+1})$ On a plus besoin de connaître a_{t+1}, la méthode est dite (off-policy) et la convergence est assurée avec suffisamment d'exploration.

Pour résumer le Q-learning :
- On construit une table d'états x actions (Q-table)
- On l'initialise intelligemment
- Appliquer l'équation de mise à jour après chaque action

Le Q learning devient vite très coûteux si l'espace d'action est grand et encore plus s'il est continu, vu que l'on doit calculer le maximum à chaque itération (ce qui dans le cas continu passe à un problème d'optimisation).

Architecture acteur-critique naïve

Pourquoi passer du Q-learning à la méthode acteur-critique ?
Dans Q-learning, il faut chercher le maximum sur les actions dans la Q-table à chaque pas, ce qui coûte très cher (encore plus dans le cas continu). On peut stocker la meilleure valeur pour chaque état, cela coûte moins cher. Stocker le maximum revient à stocker la politique. La politique se met donc à jour localement en fonction de la valeur de du critique.

La *Policy iteration* se fait sur des espaces d'actions et d'états discrets et dans le cas stochastique. La mise à jour du critique entraîne une mise à jour de l'acteur ; le critique calcule la mise à jour temporelle δ et met à jour

$$V_k(s) = V_k(s) + \alpha_k \delta_k$$

.
L'acteur est mis à jour par :

$$P^\pi(a|s) = P^\pi(a|s) + \alpha_k \delta_k$$

La recherche du maximum sur les actions n'est plus nécessaire. Par contre il faut savoir tirer une action à partir d'une politique probabiliste (pas évident pour les actions continues).
On se trouve dans le cas d'apprentissage par renforcement : on ne connaît donc pas les fonction de transition et de récompense. On tient donc à jour en parallèle une représentation de la fonction de valeur d'action (critique) et une représentation de la politique (acteur). Une mise à jour de la fonction de valeur d'action induit une mise à jour locale de la politique. (Pourquoi locale? Car une fois la valeur de Q mise à jour, la politique se met à jour en fonction des valeurs des actions voisines).
La prise d'informations sur l'environnement entraîne les mises à jour du critique et de l'acteur.

La méthode acteur critique est un cas particulier d'apprentissage direct et est proche de la policy iteration.

Algorithme DQN

Les algorithmes précédents se révèlent relativement inefficace dans certains cas (notamment dans l'apprentissage par renforcement de robots) car l'estimation de la fonction de la valeur d'action par régression linéaire est trop imprécise. La mise en place de réseaux de neurones profonds pour l'estimation de la fonction permet de surmonter le problème car ce genre d'outils est beaucoup plus puissant. L'enjeu est alors de découvrir le bon jeu de paramètres dans le large espace de départ. L'algorithme repose alors sur une propagation arrière efficace dans le réseau de neurones profonds.
L'algorithme qui a permis des avancées majeures dans le domaine est appelé *DQN (pour Deep-Q-Networks)* [6] et a permis d'apprendre à jouer à plus de 400 jeux Atari avec les mêmes réglages/paramétrages (espace d'action discret).

Rappel : Neurone Artificiel

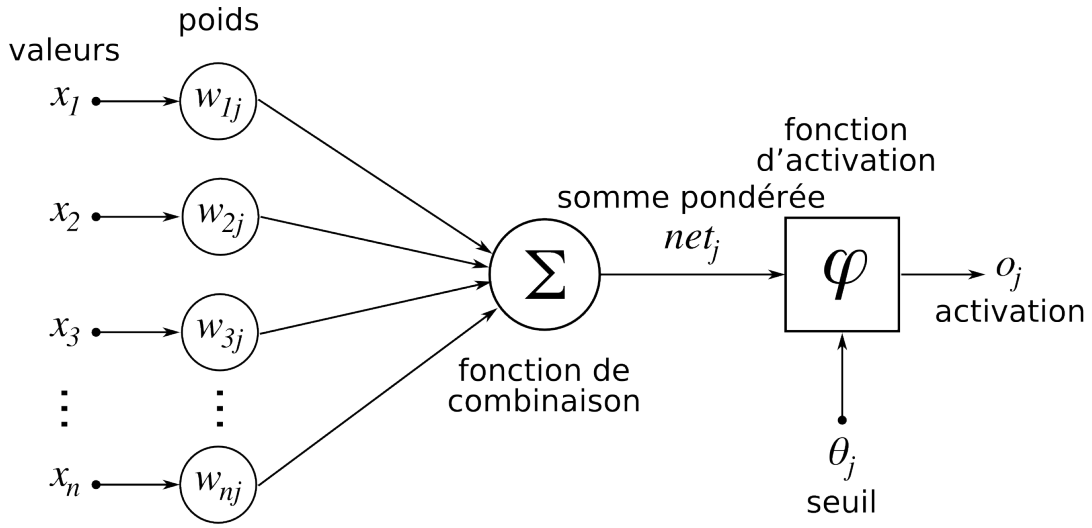

Figure 2.1: Schéma d'un neurone artificiel classique [4]

La somme de chaque entrée x_i multipliée par le poids correspondant w_i et du biais b nourrit la fonction f dont la sortie sera donc $y_i = f(\sum_i w_i x_i + b$. Les poids w_i sont les paramètres du neurone et cherchent à se rapprocher du modèle. Dans un réseau de neurones, plusieurs couches de neurones sont consistuées et la sortie d'une couche constitue à l'entrée de l'autre. La grande non-linéarite de cette construction permet d'approximer des fonctions complexes. Les paramètres des neurones sont initialisés aléatoirement. Entrainer un réseau de neurones consiste à optimiser ses paramètres pour minimiser le coût de la fonction de perte que l'on montrera plus loin.

Le modèle admet ses limites dans le fait qu'il requiert une sortie du neurone par action. On trouve l'action en calculant le maximum (comme dans Q-learning)
Le Q réseau est paramétré par θ.

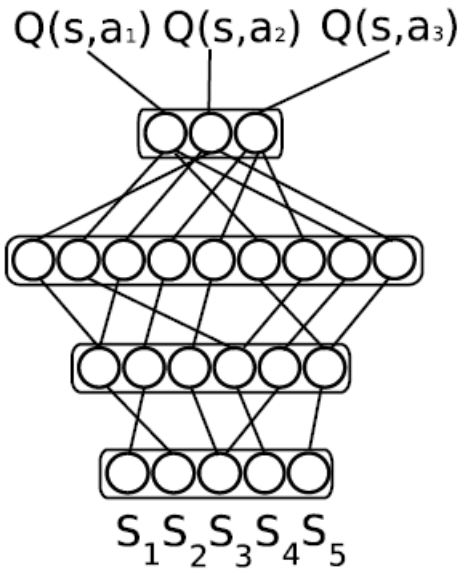

Figure 2.2: Réseau de neurone de DQN [2]

L'apprentissage supervisé consiste à minimiser une fonction de perte, le plus souvent par la méthode des moindres carrés en fonction de la sortie :

$$L(s,a) = y^*(s,a) - Q(s,a|\theta))^2$$

par propagation arrière sur les poids du critique θ.
Pour chaque sample i, le Q-network minimise l'erreur de différence temporelle :

$$_i = r_i + \gamma max_a Q(s_{i+1}, a|\theta) - Q(s_i, a_i|\theta)$$

Ainsi, avec un minibatch de n exemples s_i, a_i, r_i, s_{i+1}, on procède sur :

$$y_i = r_i + \gamma max_a Q(s_{i+1}, a|\theta)$$

Donc on met à jour θ en minimisant la fonction de perte :

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta))^2$$

Comme $y_i = r_i + \gamma max_a Q(s_{i+1}, a|\theta)$ est une fonction de Q, ce n'est pas vraiment de l'apprentissage supervisé, et l'algorithme n'est pas stable.
Du coup, on va calculer sur un réseau de cibles séparé $y_i = r_i + \gamma max_a Q'(s_{i+1}, a|\theta')$ et mettre à jour pas à chaque pas de temps mais toutes les K itérations.

ReplayBuffer

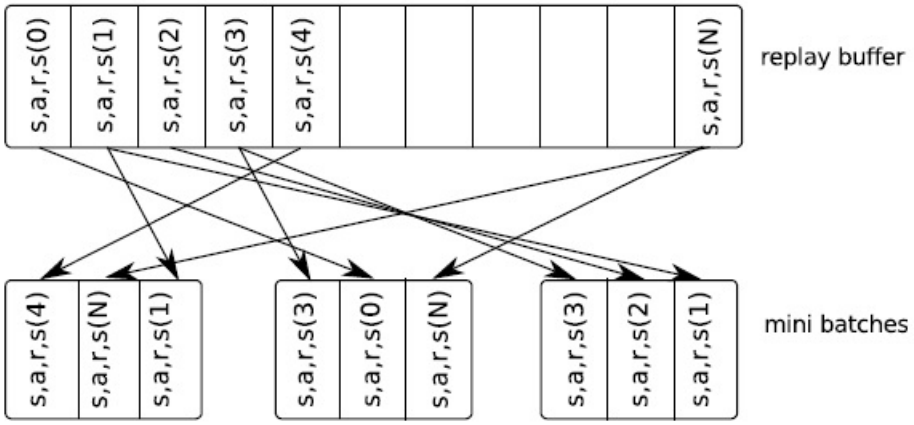

Figure 2.3: ReplayBuffer [2]

Le replay buffer se constitue au fur et à mesure de l'algorithme en ajoutant un sample (s_i, a_i, r_i, s_{i+1}) à chaque itération. C'est ce qui constitue le stock des expériences passées et permet l'apprentissage. Dans la plupart des algorithmes on part de l'hypothèse que les samples sont indépendants et identiquement distribués, ce qui n'est évidemment pas le cas ici. En effet, lors du même épisode les couple etat-actions sont corréles temporellement.
De ce fait, pour entraîner le réseau, on va constituer un minibatch (c'est à dire une sélection réduite de samples du replay buffer). Les samples du minibatch sont tirés aléatoirement parmi le replay buffer pour "casser" les liens entre les samples.
La gestion du replay buffer n'est pas optimale, nous reviendrons dessus pour améliorer l'algorithme.

3 DDPG

L'algorithme DDPG [7] est un algorithme de type acteur critique (cf. explication précédente) qui représente, dans un espace continu, la fonction de valeur d'action $Q(s,a)$ et la politique π par des réseaux de neurones profonds. Il consiste en un mélange de l'algorithme DQN [6], Deterministic Policy Gradient (DPG) (DPG) (Silver et al., 2014) et de la Batch Normalization [8].

Dans DDPG, le réseau de l'acteur obtient un vecteur d'actions à partir d'un vecteur d'états de manière déterministe, tout en apprenant une politique déterministe, beaucoup plus facile qu'une stochastique (l'espace de recherche étant beaucoup plus petit).

Le pas de temps est ici représenté par la variable t et de ce fait s_t, a_t, r_t représentent respectivement le vecteur d'état, le vecteur d'action et la valeur de la fonction de récompense au moment t.

A chaque interaction avec l'environnement, le quadruple (s_t, a_t, r_t, s_{t+1}) est stocké dans le replay buffer. On rappelle que le replay buffer est le stock des expériences passées constitué pour entraîner les réseaux de neurones. A chaque itération de l'apprentissage, un minibatch de samples est construit de manière aléatoire à partir du replay buffer, créant artificiellement une indépendance et une identique distributivité (une astuce empruntée à DQN qui permet d'améliorer la stabilité de l'algorithme). Une autre subtilité empruntée à DQN pour aider à la stabilisation : au lieu de procéder directement sur les réseaux de l'acteur et du critique pour les calculs , on procède sur des reseaux cibles appelés "target network" qui sont mis à jour plus lentement. En pratique, ils suivent le réseau de base de la manière suivante : $\theta' = \theta' \times (1-\Gamma) + \theta$ avec un Γ petit, et θ les paramètres du réseau considéré.

De ce fait, le critique apprend le couple état-action en minimisant l'erreur de différence de temporelle :

$$\delta_t = r_t + \gamma Q^{'}(s_{t+1}, \pi^{'}(s_{t+1})|\theta^{'}) - Q(s_t, a_t|\theta)$$

où γ est le facteur d'actualisation, Q le critique, $Q^{'}$ le target critique, π la target politique et θ et $\theta^{'}$ respectivement les paramètres du critique et du target critique.

Cette différence dans la mise à jour va permettre de stabiliser le critique et de l'aider à la convergence.

L'algorithme minimise l'erreur quadratique sur le minibatch avec la descente de gradient et en utilisant la fonction de perte : $L = \frac{1}{N}\sum_{i\in m}\delta_i^2$ où N est la taille du mini-batch et m son contenu.

Les propriétés générales des réseaux de neurones nous assurent que le critique va approximer de manière correcte la fonction de valeur d'action pour chaque point de l'espace d'état, sans nécessairement avoir un nombre important de sample. Cependant, il faut se rappeler que la méthode de descente de gradient est une méthode d'optimisation locale et ne garantit pas la convergence vers un optimum global.

Ensuite, l'acteur est entraîné en utilisant le gradient de la politique déterministe (comme proposé dans Silver et al., 2014) :

$$\nabla_w \pi s, a) = \mathbb{E}_{\rho(s)}[\nabla_a Q(s, a|\theta)\nabla_w \pi(s, w)]$$

Le gradient est calculé premièrement par propagation arrière du gradient de la fonction de valeur d'action en suivant les actions à travers le critique. Calculer le gradient en respect des actions est similaire à le faire en respect des poids comme noté dans (Hafner Riedmiller, 2011). Ensuite l'algorithme propage en arrière le gradient obtenu dans l'acteur avec respect des paramètres des neurones de sortie, dans les neurones d'entrée.

L'entièreté de l'algorithme repose sur la propagation arrière fournie dans les libraires de deep learning (nous utilisons Tensor Flow ici). Actuellement, le gradient propagé à travers le réseau de l'acteur exprime la direction dans laquelle aller dans l'espace de la politique afin d'avoir la meilleure récompense pour un état donné.

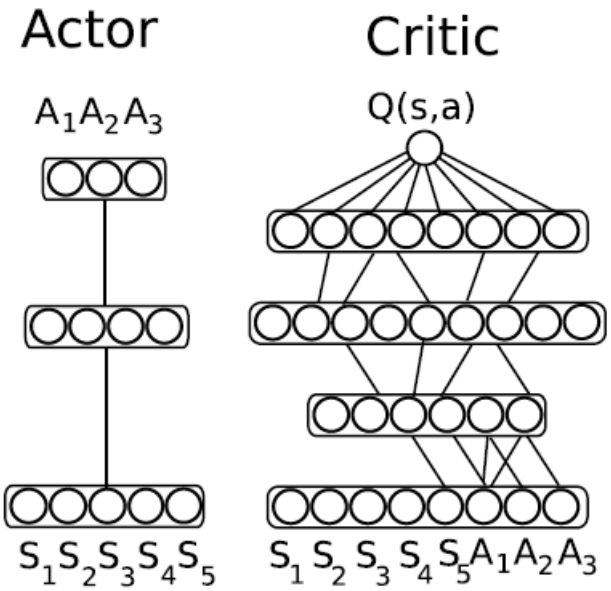

Figure 3.1: L'acteur et le critique du DDPG [2]

La troisième composante permettant d'améliorer la stabilité et d'augmenter la vitesse d'apprentissage, la batch normalization n'est pas développée ici, ni implémentée. Pour plus de détails on peut consulter la publication originale sur le sujet : [8].
Le réseau acteur et le réseau critique sont entraînés/"apprennent" après chaque étape dans l'environnement. Bien que le processus d'apprentissage hérite des propriétés de la méthode off-policy de DPG, il la performe parallèlement à l'exécution d'un épisode, ainsi l'algorithme améliore la politique pendant l'interaction avec l'environnement.

Cependant, l'apprentissage des réseaux peut être plus ou moins couplé du processus de sélection des samples et de la gestion du replay buffer qui peut être critique pour l'efficacité de l'algorithme (de Bruin et al., 2015).

Fonctionnement simplifié de l'algorithme :
- Nourrir l'acteur avec l'état, générer l'action - Nourrir le critique avec l'état et l'action, générer $Q(s,a|\theta^Q)$
- Mettre à jour le critique en minimisant la fonction de perte (mise à jour de θ)
- Calculer $\nabla_a Q(s,a|\theta)$
- Mettre à jour l'acteur (mise à jour de w)
- Ajouter le sample dans le replay buffer

4 Amélioration de DDPG

Pourquoi trier par TD-error ?

Notre but sera le suivant, poursuivre l'implémentation de DDPG développée par Arnaud de Broissia en apportant des modifications au niveau du replay buffer. C'est à dire ne plus choisir constituer les mini batches aléatoirement à partir du replay buffer mais classer les sample en fonction de leur erreur de différence temporelle (de manière décroissante) pour accélérer l'apprentissage. . En effet, les samples ayant une erreur de différence temporelle élevée sont plus importants pour apprendre le réseau car une erreur de différence temporelle élevée signifie que le sample constitue un cas inconnu (nécessaire à la progression). Un sample de cas inconnu sera en effet mal estimé par le réseau (écart fort par rapport à sa valeur réelle) et produira donc une erreur de différence temporelle forte.
Ainsi contrairement à l'ancienne méthode qui donnait la priorité aux cas les plus "connus", c'est à dire les cas les plus présents dans le replay buffer, sans aucun regard sur leur importance pour atteindre le but (ce qui faisait peu de sens).
Ainsi la convergence du réseau, devrait être deux fois plus rapide [5] .

Implémentation du nouveau replay buffer

On a conservé les tailles auparavant implantées qui sont de 60 000 samples pour le replay buffer et de 64 pour le minibatch.

Dans l'algorithme DDPG original, pour créer le minibatch on utilise une loi aléatoire uniforme sur la taille du replay buffer. Ce n'est pas une solution idéale : cette approche va sélectionner les samples à la même fréquence qu'ils ont été "vécu", sans considération pour leur importance dans l'apprentissage. on va utiliser une loi différente pour la distribution dite "loi de puissance" une fois le replay buffer trié. [1]
Lorsque le replay buffer est trié, chaque sample pourrait être associé à son erreur de différence temporelle pour le tirage mais les samples à erreur de différence temporelle faible auraient alors très peu de chance d'être sélectionnés. Pour y remédier, on utilise v pour représenter la valeur de priorité $v_i = \frac{1}{i^\alpha}$, où i est le rang d'un sample. Voici la probabilité pour chaque sample:

$$P_i = \frac{v_i}{\sum_k v_k}$$

Cette distribution est très un peu couteuse ; complexité en K=40 000. Un tirage de la sorte serait trop coûteux au vu de la taille du replay buffer $K = 40000$ samples. Pour améliorer l'algorithme, on va regrouper les samples : on découpe le replay buffer trié, en N groupes appelés "segments", on choisit le segment en suivant la "loi de puissance", puis on tire à l'intérieur du segment uniformément un sample [1]. Chaque segment a donc pour valeur de

priorité la somme des v_i des samples qui le constitue et comme probabilité :

$$p_n = \frac{\sum_{i \in i_n} v_i}{\sum v_k}$$

où n est l'indice du segment, et i_n est l'ensemble des indices de sample dans segment n.
Pendant l'exécution, on remarque que dans le cas d'un réseau de petite taille, le risque d'oublier les actions apprises auparavant. Pour contourner le problème et conserver les expériences apprises importantes, on crée un second buffer nomme "bests" qui contiendra les "meilleurs samples". Lors de la constitution du minibatch, chaque sample aura une probabilité de $\frac{9}{10}$ de provenir du buffer usuel et une probabilité de $\frac{1}{10}$ de provenir du buffer "bests".
Pour trier le replay buffer avant de constituer le minibatch, on utilise une fonction de tri déjà implémentée dans Python. Pour ne pas trop ralentir l'algorithme, le replay buffer ne sera trié qu'une fois toutes les 10 étapes.

-

Algorithme DDPG

```
Initialise the replay buffer D
Initialise the Parameters of the Actor networks Pi
Initialise the Parameters of the Critic networks Q
for t in [1,T]:
    Agent observe a state s_t, and play an action a_t
    Agent observe the state s_t+1, and the reward r_t
    Store transition (s_t, a_t, r_t, s_t+1) in D
    Get random samples of m transitions from D
    Actor and Critic networks studies these samples
    Perform gradient decent algorithme on Q then on Pi
    update the network with a frequency tau
```

-

-

Algorithme DDPG avec TD error prioritised buffer

```
Initialise the replay buffer D
Initialise the Parameters of the Actor networks Pi
Initialise the Parameters of the Critic networks Q
for t in [1,T]:
    Agent observe a state s_t, and play an action a_t
    Agent observe the state s_t+1, and the reward r_t
    Store transition (s_t, a_t, r_t, s_t+1) in D
    Get samples of m transitions from replay buffer sorted by TD error
    Actor and Critic networks studies these samples
    Transitions update the new TD errors of these samples
    Perform gradient decent algorithme on Q then on Pi
    update the network with a frequency tau
```

5 Problème : MountainCarContinuousVO

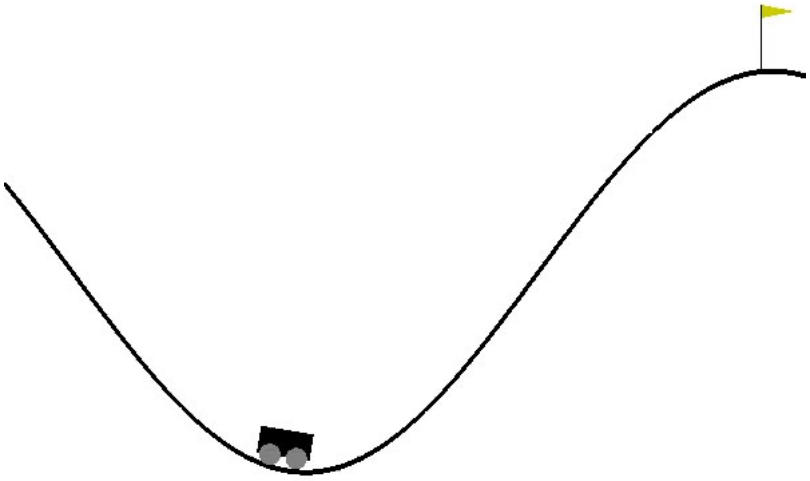

Figure 5.1: Mountain car version continue, source: http://gym.openai.com

Contexte : Une voiture sous-performante doit gravir une colline unidimensionnelle pour atteindre une cible. La cible est située en haut de la colline située à droite de la position de départ de la voiture. Sur la gauche de la position de départ, se trouve une autre colline que la voiture peut utiliser pour gagner de l'énergie potentielle et de l'accélération pour atteindre la cible sur la colline d'en face. En haut de cette seconde colline, la voiture ne peut dépasser un certain point (mur invisible).

État du problème:

- **État** : Position (paramètre horizontal) x Vélocité (paramètre vertical)
  $s \in [-1.2, 0.5] x [-0.4, 0.4]$
- **Action** : Accélération(paramètre horizontal)
  $a \in [-1, 1]$
- **Reward** :
  $r \in [-0.1, 0[\cup \{100\}$
- **But** :
  l'agent gagne une récompense supérieure à 90 pour 100 épisodes consécutifs

Remarque: Si l'agent n'arrive pas à l'objectif (c'est à dire au drapeau en haut de la colline de gauche) à l'état suivant, il obtient une récompense négative. S'il atteint l'objectif, il obtient une récompense de $1 - \sum_{a_i} V(a_i)$, où $V(a_i)$ sont les récompenses des actions précédentes a_i.

Analyse du problème : La difficulté du problème est que l'agent ne peut pas conduire directement jusqu'au drapeau (c'est à dire se placer simplement en état terminal), il faut monter sur la colline de gauche, puis redescendre dans la vallée pour gagner en vitesse et monter sur la colline de droite jusqu'au drapeau.
Algorithme utilisé : Puisque la combinaison de l'action et l'état explose, et pire dans le cas continu, alors il est impossible de traiter ce cas avec la matrice de décision markovienne. On choisit donc DDPG et on modélisera le critique et l'acteur avec un réseau de neurones.

6 Tests de performance

Architecture du code

Dans le code DDPG modifié à la "prioritized experience replay" , on a modifié plusieurs fichiers :
DDPG_gym.py, qui construit les graphes, et les autres fonctions d'exécution et d'apprentissage,
test_ddpg_mc.py, qui contient les fonctions de tests,
replay_buffer.py, qui contient les fonctions de tri, de sélection des samples du mini batch,
mecanics.py, qui contient les calculs de l'erreur de différence temporelle,
DDPG_mc_config.xml qui contient les paramètres des graphes et qu'on utilise pour configurer l'algorithme .

Manuel d'utilisation

1, Télécharger les bibliothèques:

```
#open ai gym
git clone https://github.com/openai/gym.git
cd gym
sudo pip install -e .
cd ..

#tensorflow version CPU
sudo apt-get install python-pip python-dev
export TF_BINARY_URL=https://storage.googleapis.com/tensorflow/linux/cpu/
tensorflow-0.10.0-cp27-none-linux_x86_64.whl
sudo pip install --upgrade $TF_BINARY_URL

#DDPG avec amlioration
git clone git@github.com:yyyy1999/DDPG.git
```

2, Exécution

```
#get in the parent repository
#export PYTHONPATH=$PYTHONPATH:(path to your DDPG directory's parent)
export PYTHONPATH=$PYTHONPATH:${HOME}
```

python .DDPG/test/test_ddpg_mc.py

Expérience

L'objectif de l'expérience est de comparer les performances du DDPG original et celles de la version améliorée donnant la priorité aux samples à haute erreur de différence temporelle.

Les paramètres de tests:

Paramètres	Acteur	Critique
h_1	200	200
h_2	100	100
α	0.3	0.05
τ	0.00001	0.00001
r	-	0.0001
γ	-	0.99
α_l	-	0.4-1.0

Où h_1, h_2 sont le nombre de neurones des deux 'hidden layer', α le taux d'apprentissage, et τ le taux de mise à jour. De plus, γ est le facteur d'actualisation et r est le coefficient de régularisation.

Pour comparer le fonctionnement d'exposant α_l de la loi de puissance, on exécute l'algorithme pour $\alpha_l \in \{0.4, 0.6, 0.8, 1.0\}$. En fait, si l'exposant α_l est grand, l'agent va travailler quasi exclusivement avec des samples à erreurs de différence temporelle grandes. Une fois l'erreur minimisée pour ce sample, on a peu de chances de le revoir car il sera déplacé au fond du replay buffer trié [1]. Du coup, un $alpha_1$ trop grand risquerait d'empêcher la convergence du réseau : en ne travaillant que sur des très hautes erreurs de différence temporelle, l'algorithme créerait une boucle. En effet minimiser l'erreur sur certains samples pourrait feire remonter l'erreur d'autres qui se retrouveraient alors en tête du replay buffer lors du prochain tri, qui à leur tour feraient remonter l'erreur du premier sample, empêchant ainsi toute convergence.
Si α_l est petit, il est proche d'une loi uniforme.
Ces résultat est donné par 2 exécutions chacun jusqu'à 500 épisodes

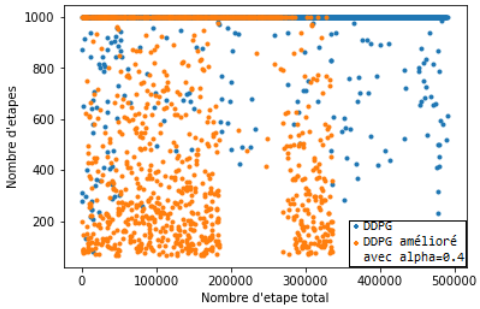

Figure 6.1: Performance de ddpg et la version amélioré avec $\alpha_l = 0.4$

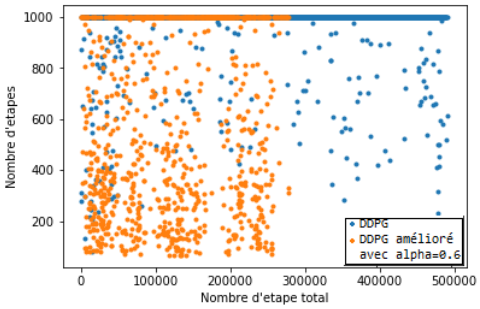

Figure 6.2: Performance de ddpg et la version amélioré avec $\alpha_l = 0.6$

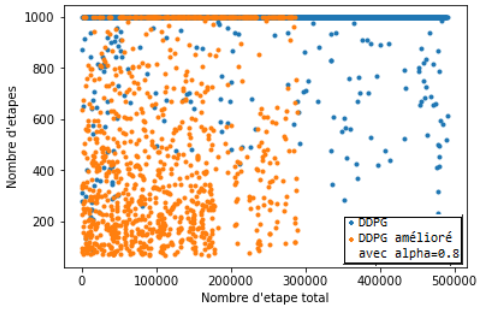

Figure 6.3: Performance de ddpg et la version amélioré avec $\alpha_l = 0.8$

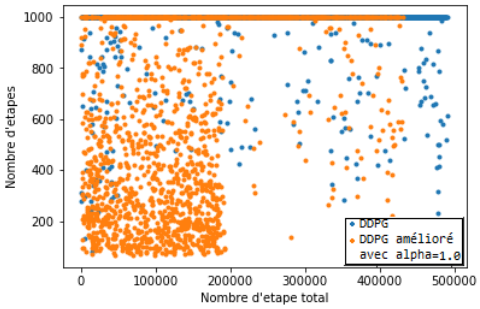

Figure 6.4: Performance de ddpg et la version amélioré avec $\alpha_l = 1.0$

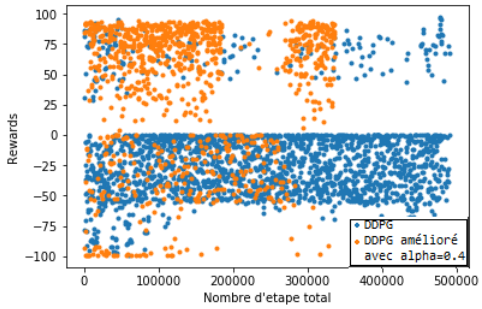

Figure 6.5: Performance de ddpg et la version amélioré avec $\alpha_l = 0.4$

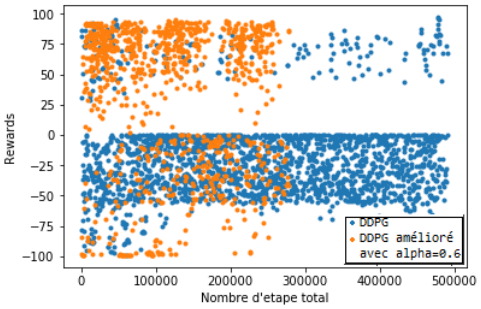

Figure 6.6: Performance de ddpg et la version amélioré avec $\alpha_l = 0.6$

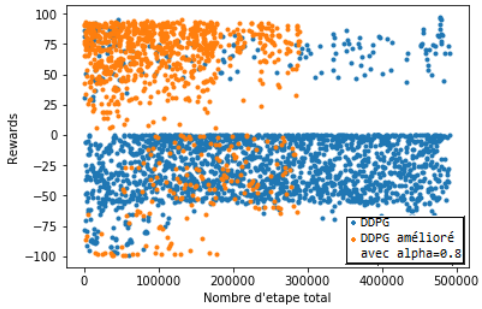

Figure 6.7: Performance de ddpg et la version amélioré avec $\alpha_l = 0.8$

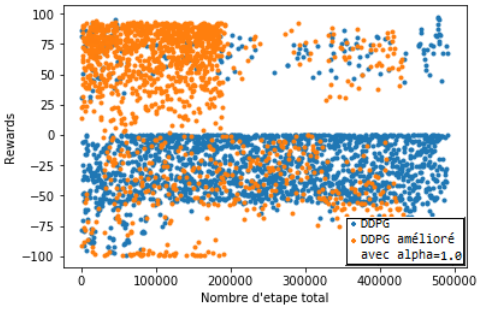

Figure 6.8: Performance de ddpg et a version amélioré avec $\alpha_l = 1.0$

L'algorithme travaille sur les premiers segments du replay buffer.
Les samples correspondant à l'état juste avant l'arrivée ont une grande erreur de différence temporelle, car la récompense de ce dernier sample est environ 99 contrairement à tous les autres états plus éloignés de l'objectif qui ont un nombre négatif. Pendant l'apprentissage, l'erreur des samples qui approchent de l'état final augmente. En minimisant ces erreurs, le réseau critique donne petit à petit une bonne valeur aux samples que l'agent choisit.

On voit aussi que le paramètre 'learning rate' α contrôle le flexibilité du réseau, c'est à dire si α est grand, le réseaux s'adapte vite. Le paramètre τ contrôle la fréquence de mise à jour, et on constate que s'il est petit alors le réseau est stable.

Voici le groupe de paramètre le plus stable que noys avons trouvé pour l'instant:

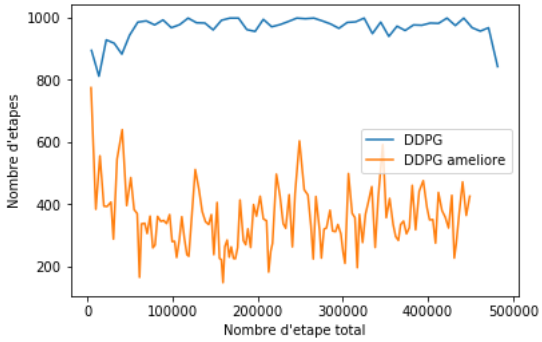

Figure 6.9: Performance de ddpg et la version amélioré avec $\alpha_l = 0.6$, $\gamma = 0.99$, $\tau = 0.000002$, $h1 = 400, h_2 = 200$

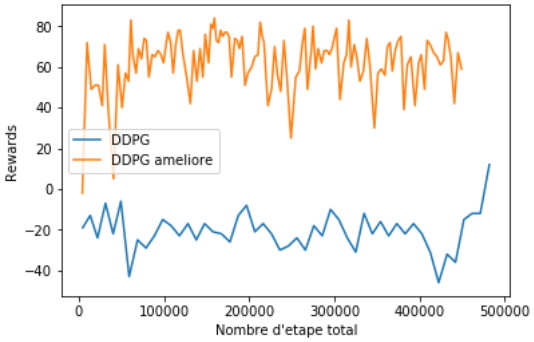

Figure 6.10: Performance de ddpg et la version amélioré avec $\alpha_l = 0.6$, $\gamma = 0.99, \tau = 0.000002$, $h_1 = 400, h_2 = 200$

7 Conclusion et travail futur

L'algorithme DDPG est instable, il est difficile de trouver des paramètres qui lui permette de converger, d'où la grande irrégularité qu'il peut y avoir dans les résultats. On a d'abord essayé de diminuer le taux de mise à jour du réseau, afin d'avoir une performance stable. En observant les samples ayant une grande erreur de différence temporelle, l'agent apprend en premier les samples proche de l'état final puis remonte vers les samples proches de l'état initial. L'agent met du temps à apprendre les samples dans la vallée car il y a beaucoup de samples très semblables (c'est en effet là où se trouve la plus grande incertitude). En fait, comme notre amélioration est glouton en erreur de différence temporelle, l'algorithme risque de converger plus tôt à une solution localement optimale (en apprenant trop les mêmes exemples on aurait une "perte de généralité"). Par conséquent, quand il rencontre des exemples inconnus mais potentiellement importants, il évalue très haut leur erreur de différence temporelle.

Dans les expériences d'échecs, l'agent apprend quelque fois très vite donc le réseau ne converge jamais. Pire encore, le réseau critique réalise un 'flip-flop' sur les transitions similaires qui sont nombreuses (c'est à dire qu'il perd ce qu'il a appris avant quand il apprend d'autres cas). Le critique ne donne alors jamais une estimation correcte car le minibatch n'est constitué que de samples qui bouclent entre eux.
De plus ,si le γ est plus petit, alors l'agent gagne de temps en temps, mais il ne converge pas vers une solution optimale.

Dans le futur, le travail à poursuivre est d'assurer la stabilité de l'algorithme. Nous avons diminué le taux de mise à jour τ, pour que les réseaux critiques et acteurs soient stables. Cependant le problème de la convergence demeure. On observe qu'à partir de 200 épisodes, le nombre d'étape semble périodique. Il croître à 800 d'abord puis descend à 200 avant de recommencer le cycle. De plus, l'agent n'arrive pas à jouer consécutivement des épisodes

ayant une récompense supérieure à 90. On devine que l'agent ne voit pas trop la différence entre les samples au fond de la vallée, trouver l'entrée d'un "bon chemin" étant très couteux. Les possibilités d'entrées potentielles étant nombreuses aux états voisins de l'état initial il y a de grandes chances de ne pas faire la bonne action (trouver la bonne entrée du chemin). Au vu du nombre d'étapes limitées par épisode, il n'aura pas forcément le temps de corriger son "mauvais départ". On peut aussi réduire le coefficient $alpha$ de la loi de puissance (constitution du minibatch) au fur et à mesure de l'exécution jusqu'à presqu'atteindre une loi uniforme.

References

[1] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver *Prioritized experience replay*, 2016

[2] Olivier Sigaud, *Deep Reinforcement Learning Algorithms*, 2016

[3] Arnaud de Broissia, Olivier Sigaud, *Study of a deep reinforcement learning algorithm*, 2016

[4] Simon Ramstedt, *Deep Reinforcement Learning for Continuous Control*, 2016

[5] Schaul, T., Quan, J., Antonoglou, I., Silver, D. *Prioritized experience replay*, 2015

[6] Mnih, et al. *Human-level control through deep reinforcement learning*, 2015

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

[8] Ioffe S. Szegedy C. (2015). *Batch normalization: Accelerating deep network training by reducing internal covariate shift*