Bio inspired : Apprentissage profond par renforcement

Pierre CABANIS (11401778) & Steeven JANNY (11918550)

1 Introduction

Ce compte-rendu détaille la mise en oeuvre et l'étude des techniques d'apprentissage profond par renforcement. On cherche à développer et trouver la meilleure politique d'action pour un agent interagissant dans un certain environnement. Dans ce projet, la théorie du *Reinforcement Learning* est combinée avec celle du *Deep Learning* pour résoudre des jeux d'arcade Atari.

Les algorithmes seront implémentés en Python à l'aide de la librairie Pytorch. L'environnement sera modélisé à l'aide de la librairie Gym.

Ce TP explore dans un premier temps l'apprentissage par renforcement dans l'environnement **Cartpole** puis dans un environnement plus complexe : **Breakout Atari**.

2 Code

Lien du dépôt Github : `https://github.com/PierreCabanis/Deep-RL`

Les fichiers **Cartpole.py** et **Breakout.py** exécutent l'environnement associé. Ces deux fichiers commencent par des paramètres contrôlant les phases d'apprentissage et de test du réseau. Par défaut, ces fichiers sont réglés pour exécuter immédiatement la meilleure politique calculée par les auteurs.

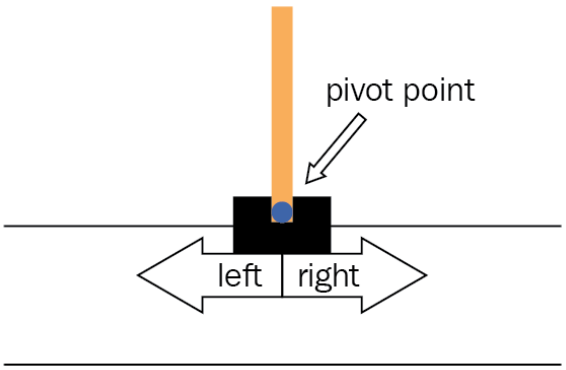
Attention, pour des raisons de compatibilité (Windows 10 64 bits), la bibliothèque gym[atari] a été installé à partir du package disponible sur le Github.

3 Partie 1 : Deep Q-Network sur CartPole

3.1 Présentation

Pour cette section, voir le code de : **Cartpole.py**

Dans l'environnement **Cartpole**, l'agent à deux possibilités d'action : il peut déplacer le chariot (bloc noir) sur la droite ou sur la gauche. Son objectif est de maintenir le bâton (bloc marron) droit. L'agent doit donc apprendre et trouver un **équilibre**.



Le système de récompense par défaut accorde **un point** lorsque le bâton est en équilibre et **0 point** lorsque celui-ci franchit un seuil d'inclinaison. Sur cette base, nous avons implémenté un algorithme de DQN qui tente de maximiser les récompenses reçues par l'agent.

3.2 Algorithme & Implémentation

3.2.1 Présentation générale

La structure du DQN est implémentée dans une classe **DQN.py**, qui présente l'avantage d'être facilement transposable à tout type d'environnement. Celle-ci se compose de trois éléments principaux :

- **Experience-Replay** : un système de mémoire permettant de garder une trace des actions passées et de leurs conséquences
- **Action** : chargé du choix de l'action de l'agent à l'aide des Q-valeurs. Ce système implémente également une politique $\epsilon - greedy$.
- **Learner** : le système chargé de l'apprentissage du réseau de neurones.

A noter que la **structure du réseau de neurone**s peut être précisée **indépendamment** de l'objet DQN : inutile de modifier les fonctions de DQN lorsque l'on change un élément du réseau profond. Les sections suivantes décrivent brièvement les trois systèmes.

3.2.2 Experience replay

Voir la classe **Buffer** dans **DQN.py**

L'agent apprend sa politique à partir d'une mémoire de ses **interactions avec l'environnement**. Cette mémoire est modélisé à partir d'un ensemble de données composé de l'état à l'instant n (s_n),de l'action à l'instant n (a_n) ainsi que de l'état à l'instant $n+1$ et la récompense associée.

Dans un environnement complexe, il est impossible d'imaginer stocker toute l'expérience de l'agent. On utilise donc un *buffer* glissant de taille fixe et paramétrable.

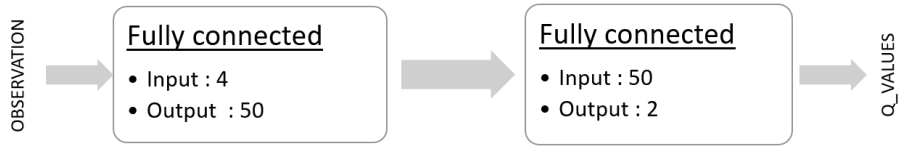
Buffer()
+ state : *Tenseur des états* s_n
+ next_state : *Tenseur des états* s_{n+1}
+ action : *Tenseur des actions* a_n
+ reward : *Tenseur des récompenses* r_n
+ done : *Tenseur booléen indiquant si une partie est finie*
+ append(s_n, $s_{n+1}, a_n, r_n, done$) : *Ajoute les données aux tenseurs*
+ get_batch(N) : *Renvoie un batch de N observations*

Le *buffer* implémenté est indépendant de l'environnement, et stocke les données sous forme de **tenseur Pytorch** (il est apparu que la conversion en tenseur lors de la phase d'apprentissage ralentissait très fortement le temps de calcul par rapport à un stockage direct au format tenseur. Cette petite modification divise la durée moyenne d'un épisode par 33 !).

La classe est munie d'une méthode *get_batch()*. Cette fonction récupère aléatoirement un groupe de données de taille n dans le *buffer* qui servira de **mini-batch** pour l'entraînement.

3.2.3 Action

Le choix de l'action de l'agent utilise une approche *Q-learning* : un réseau de neurones profond prend en entrée l'état courant de l'environnement et donne en sortie un **score** associé à chaque action. Pour ce problème, la structure du réseau de neurones choisie est très simple. Celle-ci doit être fournie en paramètre de la classe **DQN** lors de son instanciation :



Pendant la **phase d'apprentissage**, la méthode *get_ action()* de la classe DQN applique une politique ϵ-*greedy* : l'action choisie est celle recommandée par le réseau avec une probabilité $1 - \epsilon$ ou une action aléatoire avec une probabilité ϵ.

Notons qu'il est également possible de choisir l'action à effectuer en respectant les scores en sortie du réseau de neurones (on applique *softmax* en sortie du réseau pour se ramener à des "probabilités"). Il s'agit de l'**exploration Boltzmann**, qui est utilisable dans le code fourni en réglant les paramètres de configuration (en début de code).

3.2.4 Learner

Le coeur de la classe **DQN** est la méthode *learn()* chargée d'entraîner le réseau de neurones à fournir des scores cohérents pour maximiser les récompenses. On distingue deux réseaux de neurones identiques pour cette étape :

— **targetNet** Q_θ : pour la prédiction de l'action à effectuer.

— **evalNet** $Q_{\theta'}$: pour l'apprentissage

Les poids de *targetNet* sont progressivement mis en jour en fonction de ceux de *evalNet*. On détaille ci-dessous l'algorithme utilisé :

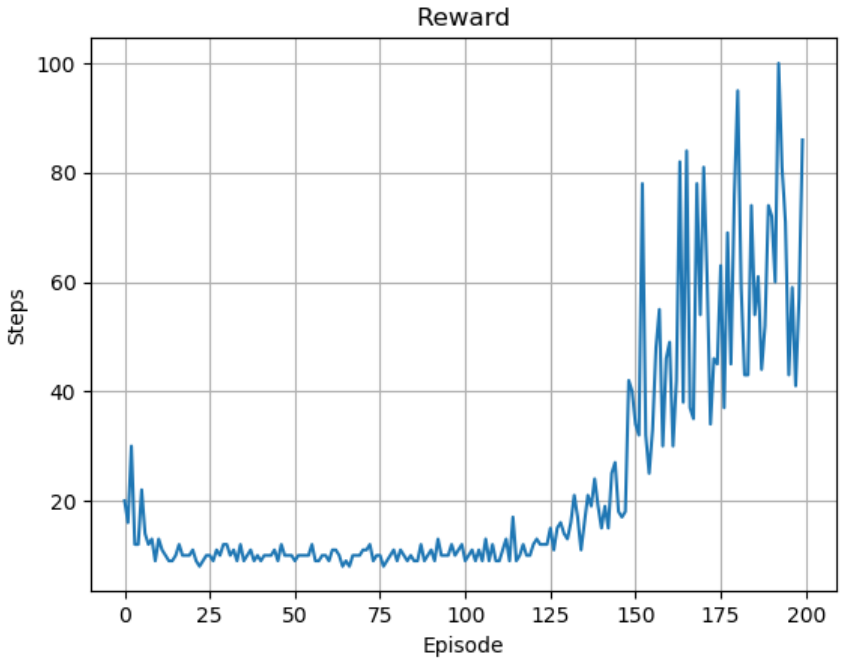
Algorithm 1: Double Q-Learning

Initialisation Q_θ et $Q_{\theta'}$
for *each episode* **do**
  for *each step* **do**
    $a_t \leftarrow \pi(a_t, s_t)$
    $s_{t+1}, r_t \leftarrow$ environnement(a_t)
    store(s_t, a_t, s_{t+1}, r_t)
  end
  for *each update step* **do**
    batch $\leftarrow$ get_batch()
    $Q^*(s_t, a_t) = r_t + \gamma Q_\theta\left(s_{t+1}, \arg\max_{a'} Q_{\theta'}(s_t, a')\right)$
    loss $\leftarrow \left(Q^*(s_t, a_t) - Q_\theta(s_t, a_t)\right)^2$
    backpropagate()
    $\theta' = \alpha\theta + (1 - \alpha)\theta'$
  end
end

Il s'agit d'une implémentation d'un **double-DQN** comme décrit dans *Deep Reinforcement Learning with Double Q-learning*, (Hasselt et al., 2015). Il est cependant possible de désactiver le mode double DQN en réglant les paramètres de configuration (en début de code).

3.3 Résultats

Les essais sur Cartpole montrent le bon fonctionnement de l'algorithme. Le graphique ci-dessous détaille notamment l'évolution de la récompense sur 200 épisodes :



Les valeurs des hyperparamètres sont précisées dans l'annexe A. A noter que l'entraînement se fait sur des épisodes de 200 actions, **même si le bâton est déjà tombé**. En effet, stopper l'entraînement lorsque le bâton tombe ne permet pas à l'agent de mémoriser suffisamment de cas négatifs pour apprendre correctement.

Les hyperparamètres de l'annexe A permettent d'atteindre un score moyen de **71 points** (sur 20 parties), malgré un écart-type très grand (environ 23 points). Nous avons poussé plus loin l'entraînement (environ 10.000 épisodes) pour constater que l'agent atteint une **maîtrise quasi parfaite de l'environnement**, le bâton reste en équilibre pour un temps excédant la patience des auteurs (soit plus de 2 heures).

Les poids du réseau de neurones ont été enregistrés dans un dossier *Save*. Le code **Cartpole.py** est réglé de sorte à charger ses poids et à lancer un test immédiatement sans entraînement. Pour vérifier les performances de notre implémentation, il suffit donc d'exécuter le fichier **Cartpole.py**.

4 Partie 2 : Breakout Atari

4.1 Présentation

L'environnement Breakout Atari est bien plus complexe. L'agent doit apprendre à jouer au casse-brique. Il a donc le contrôle horizontal de la barre verticale. Dans le jeu original, le joueur dispose de 5 vies.

4.2 Implémentations

4.2.1 Pré-traitement et récompense

Les observations consistent cette fois en des **captures d'images** du jeu à chaque instant. Des *wrappers* ont donc été utilisés pour traiter les observations. Nous utilisons le *wrapper* Atari fourni par gym. Celui-ci permet notamment :

— De passer les pixels en **niveau de gris**

— De prendre le pixel **maximum** entre deux images consécutives pour éviter le scintillement

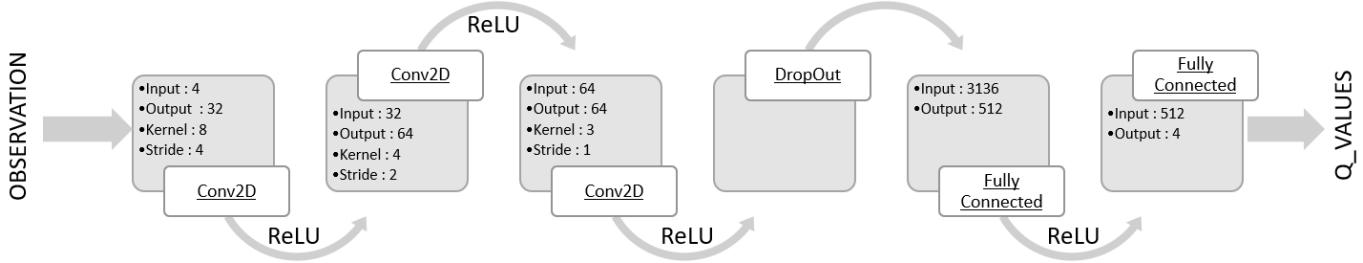
— De **redimensionner** les images en carré de 84×84

— De grouper les images par **paquets de 4**, pour extraire la vitesse et la trajectoire de la balle par exemple

L'agent reçoit **un point** lorsqu'il parvient à briser une brique et perd **10 points** lorsque celui-ci perd une vie.

4.2.2 Réseau de neurones

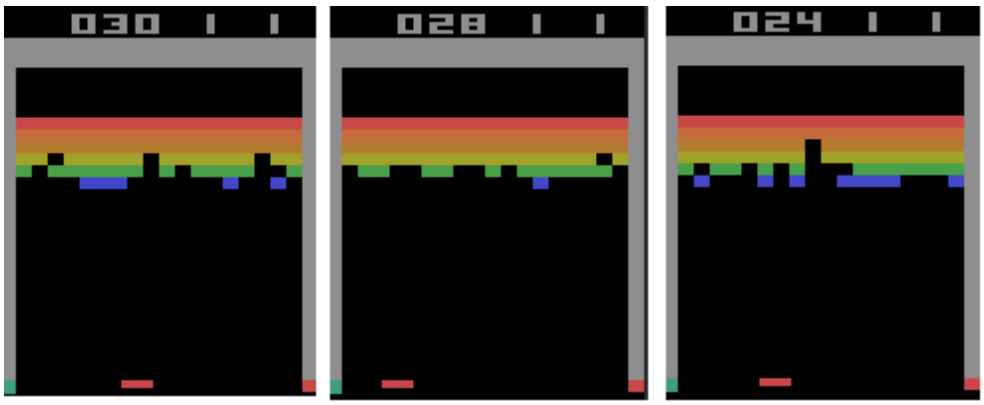
L'utilisation d'images comme observation suggère très fortement l'utilisation de couche convolutionnelle dans la structure du réseau de neurones. En s'inspirant de réseau profond suggéré par l'article de Mnih & al. nous proposons la structure suivante :



4.3 Résultats

Le temps d'entraînement nécessaire pour obtenir un début de résultat est drastiquement plus long. L'algorithme s'est entraîné sur les serveurs de Google Colab (qui mettent à disposition des GPU pour l'entraînement des réseaux de neurones) pendant plusieurs dizaines d'heures. Les auteurs ont relancé régulièrement l'entraînement, le programme étant fait de sorte à enregistrer les poids du réseau de neurones lorsque le score s'améliore et reprendre l'entraînement à partir de ces poids.

Les meilleurs paramètres calculés par les auteurs sont sauvegardés avec les fichiers du projet. Le code de **Breakout.py** est réglé de sorte à tester immédiatement ces poids et donne le score moyen sur 10 parties ainsi que l'écart-type.



Finalement, notre agent atteint un score moyen de **28.4 points** avec un écart-type de **2.1 points** (sur 50 parties). Les résultats sont certes loin de ceux présentés par Mnih et al. mais semblent corrects vis-à-vis de la puissance de calcul disponible.

Il est amusant de remarquer que l'agent parvient à briser des briques à l'intérieur du mur sans briser les briques adjacentes.

5 Bonus

5.1 Dueling DQN

5.1.1 Présentation

Plutôt que d'estimer directement les Q-valeurs $Q^\pi(s,a) = \mathrm{E}[R_t|s_t = s, a_t = a, \pi]$, on souhaite calculer séparément les V-valeurs ainsi que l'avantage qui correspond à la différence entre la Q-valeur et la V-valeur :

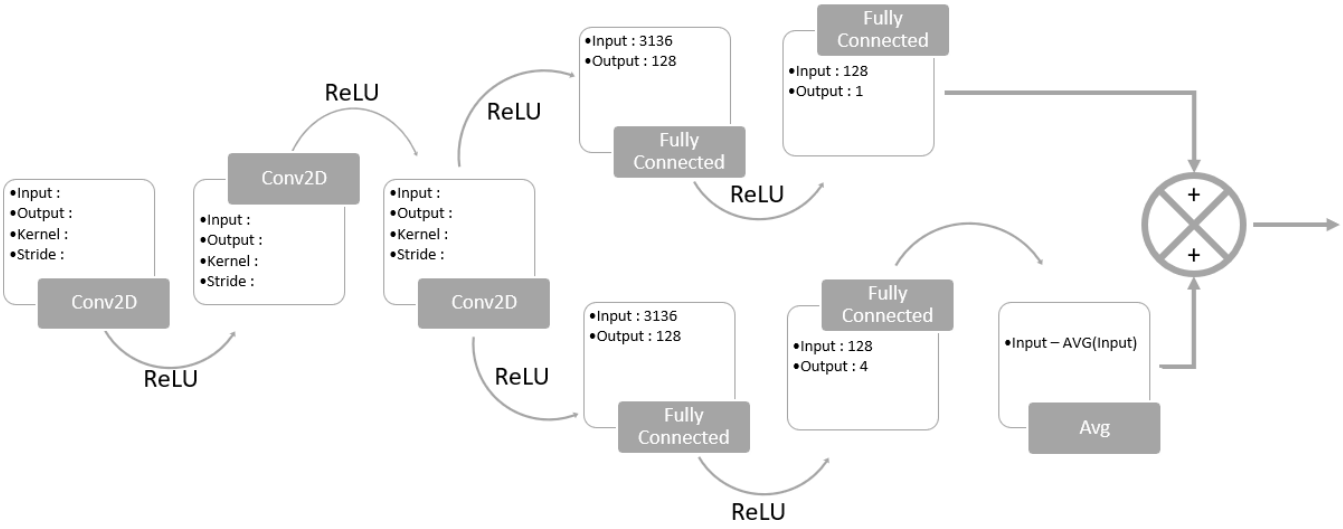
$$\begin{cases} V^\pi(s,a) = \mathrm{E}_{a\sim\pi(s)}[Q^\pi(s,a)] \\ A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s,a) \end{cases}$$

Cette transformation permet de ne pas calculer l'intégralité des scores de chaque action pour tous les états et d'identifier ceux qui ne sont pas critiques. Par exemple, les états de Breakout dans lesquels la balle monte ne sont pas très importants en comparaison à ceux où la balle redescend.

On modifie donc notre réseau de neurones convolutionel en divisant la partie linéaire en deux parties : une pour calculer A^π et une autre pour calculer V^π. Conformément à ce qui à été suggéré par Wang et al., la sortie du réseau est calculée comme ceci :

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

L'architecture du réseau de neurones devient donc :

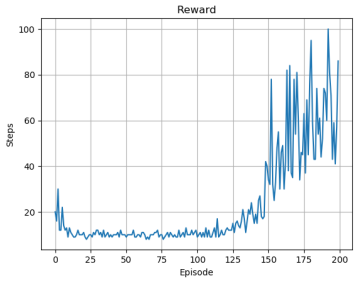
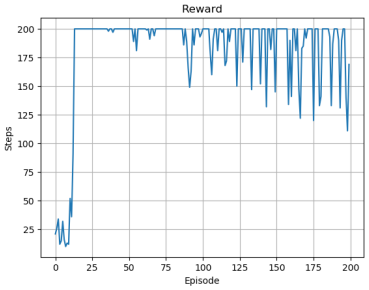


Le reste de la structure du DQN ne change pas. La structure présentée utilise des couches convolutionnelles compatibles avec les environnements Atari, mais pour des raisons de temps de calcul, nous préférons étudier cette amélioration sur les environnements Gym plus classique (en remplaçant les convolutions par des couches *fully connected*.

Dueling DQN est aussi implémenté pour les environnements Atari et est accessible par un paramètre de configuration (en début de code).

5.1.2 Démonstration

Pour évaluer Dueling DQN, nous exécutons 200 épisodes sur CartPole avec et sans cette amélioration et nous comparons le score moyen final, l'écart-type sur 20 épisodes de test ainsi que l'allure de la courbe des récompenses au cours des épisodes. Chaque entraînement est répété 3 fois, et le meilleur résultat est conservé. Les hyperparamètres sont tels que décrits dans l'annexe A.

	Courbe des récompenses	Score moyen	Écart-type
Double DQN		71.0	23.6
Dueling + Double DQN		164.7	24.48

Visiblement, Dueling DQN permet d'améliorer les performances du DQN de façon significative.

6 Conclusion

Finalement, l'approche Deep Q-Learning Network permet effectivement à un agent d'apprendre un politique maximisant l'espérance de récompense. Cependant, cela demande un temps d'entraînement considérable lorsque l'environnement est complexe.

A Paramètres Cartpool

Buffer Size	Learning Rate	ϵ	ϵ min	ϵ decay	Batch size	γ	α
2000	0.0001	0.9	0.1	0.995	32	0.9	0.005

Nb Episode	Nb Step	Start Train
200	200	1000

B Paramètres Breakout

Buffer Size	Learning Rate	ϵ	ϵ min	ϵ decay	Batch size	γ	α
10000	$10^{-3} \rightarrow 10^{-8}$	0.9	0.1	0.995	128	0.99	0.005

Nb Episode	Nb Step	Start Train
10 000	-	1000