Intelligence Bio-inspirée

TP Apprentissage profond par renforcement

1 - Environnement **2**
1.1 - Architecture du projet 2
1.2 - Chargement des modèles enregistrés 2
2 - Deep Q-network sur CartPole **2**
2.1 - Début 2
2.1.1 - Question 1 : Agent aléatoire 2
2.1.2 - Question 2 : Evolution des récompenses 3
2.2 - Experience replay 3
2.2.1 - Question 3 : Buffer d'expériences 3
2.2.2 - Question 4 : Sampling d'un mini-batch 3
2.3 - Deep Q-learning 4
2.3.1 - Question 5 : Réseau de neurones 4
2.3.2 - Question 6 : Approximation des Q-Valeurs 4
2.3.3 - Question 7 : Apprentissage sur mini-batch 5
2.3.4 - Question 8 : Target network 5
2.4 - Résultats sur CartPole-v1 5
3 - Breakout Atari **6**
3.1 - Développement 6
3.1.1 - Question 1 : Preprocessing 6
3.1.2 - Question 2 : Remaniement du code 7
3.1.3 - Question 3 : Réseau de neurones convolutionnel 7
3.1.4 - Question 4 : Optimisation des hyper-paramètres 7
3.1.5 - Question 5 : Vidéo de l'agent & Sauvegarde du réseau 8
3.2 - Discussion des résultats 8

1 - Environnement

1.1 - Architecture du projet

Les sources du projet sont disponibles sur GitHub : https://github.com/jperier/DeepRL_TP

Afin de simplifier la mise en place et la gestion de l'environnement de développement, nous avons réalisé nos expérimentations avec Python 3.7 dans un environnement virtuel réalisé au moyen d'Anaconda 3.

L'ensemble des dépendances requises pour ce projet sont listées dans le fichier requirements.txt. La procédure d'installation de cet environnement virtuel est détaillée dans le fichier Readme.md disponible à la racine du projet.

À savoir : *gym* n'étant pas pleinement compatible avec Windows, nous avons donc utilisé (lors de nos tests sur ce système d'exploitation) une version légèrement modifié de *gym[atari]* (évoquée dans la procédure d'installation pour Windows).

1.2 - Chargement des modèles enregistrés

Au fur et à mesure de nos expérimentations, nous avons sauvegardé un certain nombre de modèles dans l'optique de continuer l'apprentissage ultérieurement ou lorsque ceux-ci avaient atteint un niveau suffisant.
Ces modèles peuvent être évalués grâce au script *eval_model* disponible à la racine du projet. Ce script permet de charger un modèle pré-entraîné et de le voir jouer 3 ou 5 parties suivant l'environnement choisi.

2 - Deep Q-network sur CartPole

2.1 - Début

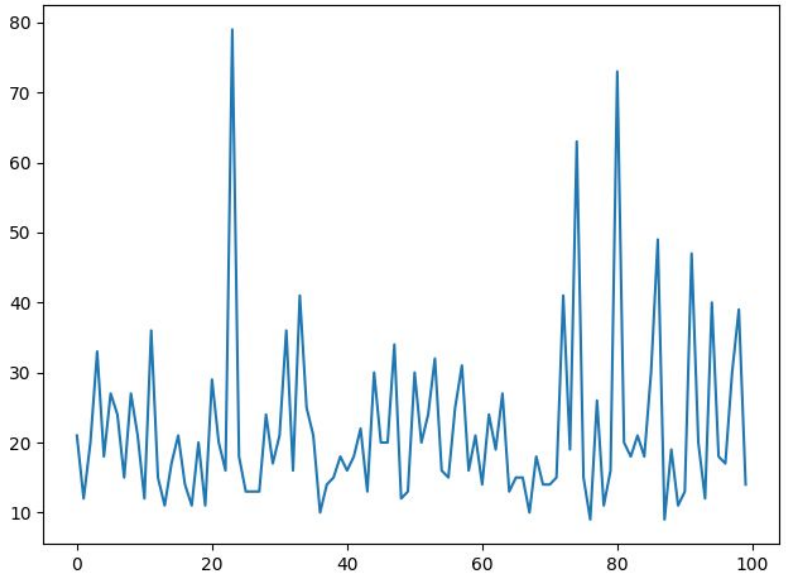
2.1.1 - Question 1 : Agent aléatoire

Le code de cet agent est disponible dans le fichier *random_agent.py*. Celui-ci est (à peu de choses près) entièrement basé sur l'exemple réalisé par les créateurs de *gym*.

Afin de nous familiariser avec les environnements *gym*, nous avons préféré conserver ce code dans un fichier séparé (tandis que nous avons rassemblé les autres agents/réseaux de neurones réalisés dans des modules).

2.1.2 - Question 2 : Evolution des récompenses

Pour visualiser l'apprentissage de l'agent (et donc l'évolution des récompenses reçues au cours de ses expérimentations avec l'environnement), nous affichons la somme des récompenses obtenues par l'agent au cours d'un épisode en fonction de l'*epoch* en cours sur un graphique réalisé avec *matplotlib*.



Graphe des récompenses de l'agent aléatoire.

2.2 - Experience replay

2.2.1 - Question 3 : Buffer d'expériences

Le code du buffer de mémoire de l'agent est disponible dans le module *agents.py* (au sein de la fonction *memorize* de la classe *SimpleAgent*).
Afin de simplifier la manipulation de la mémoire, cette fonction ne prend qu'un paramètre : *interaction*. Dans notre cas, ce paramètre est un tuple contenant *(state, action, next_state, reward, done)* et est construit dans la boucle principale juste après que l'agent ait agi.
Le principal intérêt du tuple est sa facilité de manipulation lors de l'apprentissage par batch.

2.2.2 - Question 4 : Sampling d'un mini-batch

Le code de cette fonction de sampling est disponible dans la fonction *get_batch* (située juste après *memorize*, voir ci-dessus).
Cette fonction retourne un tableau contenant *n* élément du buffer (ou vide si le buffer ne compte pas suffisamment d'éléments).

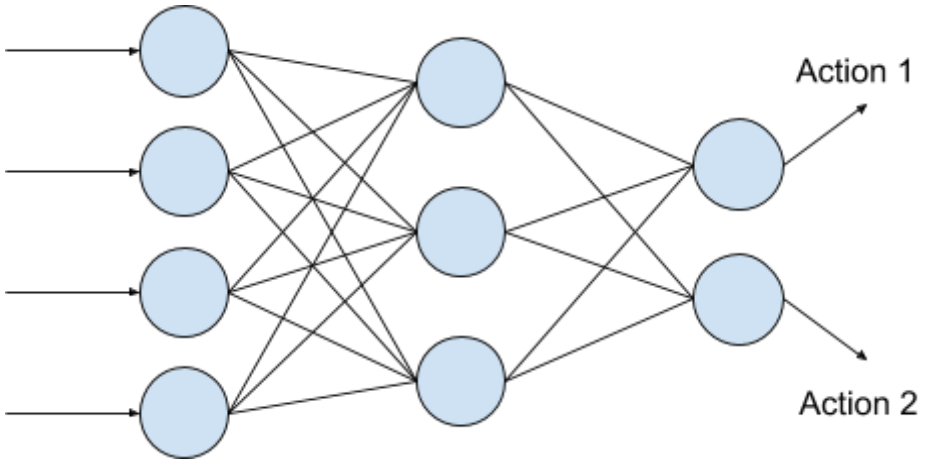
2.3 - Deep Q-learning

2.3.1 - Question 5 : Réseau de neurones

Pour faciliter la réutilisation du code, nous avons implémenté les réseaux de neurones dans le module *networks.py*. Ceux-ci sont sous la forme de classes contenant le modèle PyTorch et l'ensemble des paramètres nécessaires à son apprentissage (comme la fonction de *loss* et l'optimiseur utilisé).

Pour ce premier réseau de neurones, nous avons 2 couches de neurones avec 4 neurones d'entrée et 3 neurones de sortie pour la couche cachée, et 3 neurones d'entrée et 2 neurones de sortie pour la couche de sortie.
Entre ces deux couches, nous avons ajouté la fonction d'activation ***leaky_relu***. Nous avons choisi cette fonction d'activation car elle est très similaire à ReLU, donc très simple, mais a l'avantage de ne pas avoir de gradient qui s'annule pour certaines valeurs.

Les dimensions de ces couches correspondent à la taille des états de l'environnement *CartPole-v1* et la taille du domaine d'action. Nous avons choisi que la taille de la couche cachée corresponde à la moyenne des tailles des couches "externes".



2.3.2 - Question 6 : Approximation des Q-Valeurs

En ce qui concerne la stratégie d'exploration de l'agent, nous n'avons implémenté que la stratégie ε-greedy. Celle-ci prend en paramètres les q_valeurs estimées par le réseau de neurones et renvoie la meilleure action parmi celles-ci avec une probabilité (1 - ε) (et une action aléatoire avec une probabilité ε).
Afin de maximiser l'exploration lorsque l'agent n'a pas encore appris, nous avons mis en place une stratégie d'atténuation linéaire d'ε au fur et à mesure de l'apprentissage. Ainsi après chaque entraînement du réseau de neurones, ε est multiplié par une valeur de decay (proche de 1 pour s'assurer une convergence suffisamment lente).

Le code de cette exploration est disponible dans le module *agents.py*, dans la classe *GreedyExploration*.

2.3.3 - Question 7 : Apprentissage sur mini-batch

NB : À partir de ce moment-là, nous avons fait l'erreur de passer les séquences d'apprentissage entre les épisodes (nous n'entraînions l'agent qu'après la fin de l'épisode), ce qui rallongeait considérablement le nombre d'épisodes requis pour constater un apprentissage satisfaisant.

À partir de maintenant, notre agent utilise 3 fonctions principales tout au long de son apprentissage :

- *act(observation, reward, done)* : l'agent utilise ici son réseau de neurones pour estimer la Q-valeur des différentes actions possibles et choisit (suivant sa politque ε-greedy) s'il explore ou effectue la meilleure action possible. Cette action est ensuite appliquée à l'environnement par le biais de la fonction *step* de ce dernier.
- *memorize(interaction)* : l'agent sauvegarde le tuple *(state, action, next_state, reward, done)* dans son buffer.
- *train()* : une fois l'épisode en cours terminé, l'agent récupère un petit échantillon de données de son buffer (via la fonction *get_batch*) et l'utilise pour mettre à jour les poids de son réseau de neurones.

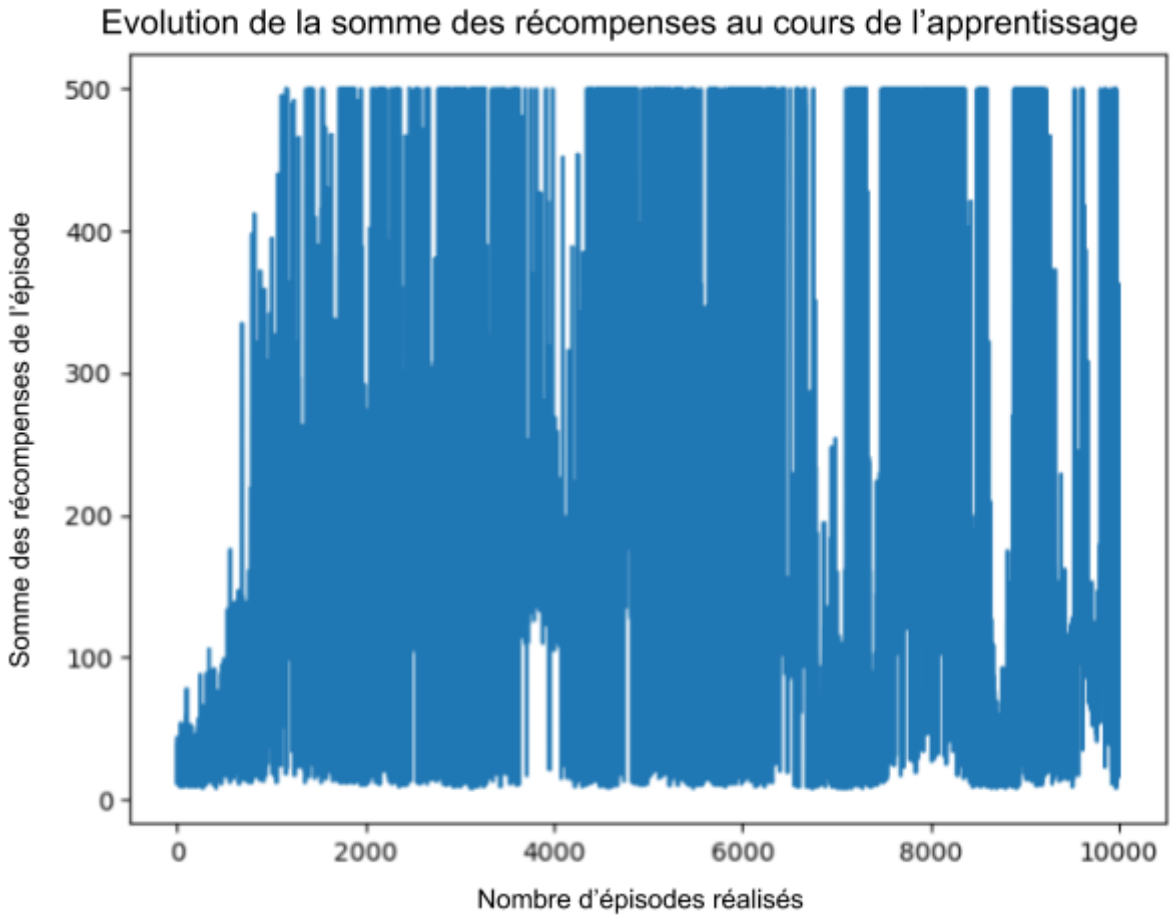
2.3.4 - Question 8 : Target network

Nous avons choisi d'implémenter le *target network* selon la solution "N étapes". Nous calculons ainsi les Q-valeurs cibles avec ce réseau qui sera mis à jour toutes les N étapes d'apprentissage en clonant le premier réseau.

La version finale de cet agent se lance par l'intermédiaire du fichier *simple_agent.py*. Ce dernier se charge d'instancier l'environnement, l'agent et son réseau.

2.4 - Résultats sur CartPole-v1

Malgré notre erreur sur les séances d'apprentissage, nous avons réussi à obtenir des résultats plutôt satisfaisants, avec un agent qui atteint le maximum possible sur ce jeu.



On constate que l'apprentissage reste assez instable. Cependant, cela est surement dû au fait que l'agent explore de nouvelles possibilités et donc sa performance peut baisser de manière significative car une "petite exploration" peut facilement lui faire perdre la partie très vite.

3 - Breakout Atari

L'ensemble des tests réalisés sur l'environnement *BreakoutNoFrameSkip-v4* ont été réalisés dans les fichiers *atari_agent.py* (cet agent utilise encore le mauvais apprentissage, c'est-à-dire qu'il apprend entre chaque épisode) et *atari_agent2.py* (cet agent apprend après chaque action effectuée).

3.1 - Développement

3.1.1 - Question 1 : Preprocessing

Pour réaliser le preprocessing de cet environnement, nous avons avons utilisé les wrappers disponibles par défaut avec *gym*, soient *AtariPreprocessing* et *FrameStack*.

Le premier permet de réduire la taille de l'écran à un carré de 84 pixels de côté, de n'afficher qu'une *frame* à l'agent toutes les 4 *frames* réelles (cette technique était utilisé dans le papier

de recherche afin d'accélérer l'apprentissage) mais également de passer tous les pixels en gris (pour omettre la gestion des couleurs).
Tandis que le second permet de conserver les 4 dernières *frames* affichées (pour en extraire l'accélération de la balle notamment).

3.1.2 - Question 2 : Remaniement du code

Nous n'avons presque pas eu besoin d'adapter notre code, néanmoins nous avons tout de même choisi de modifier la fonction d'apprentissage afin de permettre d'apprendre "en batch", c'est à dire de mettre à jour les valeurs des poids du réseau avec la moyenne du gradient des élément du batch. Ainsi, nous avons pu réduire les temps de calculs et utiliser nos cartes graphiques par le biais de Cuda.

3.1.3 - Question 3 : Réseau de neurones convolutionnel

Le réseau de neurones convolutionnel que nous avons implémenté est similaire au réseau évoqué dans le papier de recherche. Il dispose de 3 couches de convolution et d'un réseau *fully connected* de 2 couches :

- ➔ Conv1
  - ◆ in channels = 4
  - ◆ out channels = 32
  - ◆ kernel = 8
  - ◆ stride = 4
- ➔ Conv2
  - ◆ in channels = 32
  - ◆ out channels = 64
  - ◆ kernel = 4
  - ◆ stride = 2
- ➔ Conv3
  - ◆ in channels = 64
  - ◆ out channels = 64
  - ◆ kernel = 3
  - ◆ stride = 1
- ➔ Linear1
  - ◆ Neurones d'entrée = 3136
  - ◆ Neurones de sortie = 512
- ➔ Linear2
  - ◆ Neurones d'entrée = 512
  - ◆ Neurones de sortie = 4

Nous avons ainsi utilisé les 4 *frames* (conservées grâce au wrapper *FrameStack*) comme *channel* de la première couche de convolution.

3.1.4 - Question 4 : Optimisation des hyper-paramètres

Nous avons testé plusieurs configurations:

- Utiliser les mêmes paramètres que dans l'article (avec RMSProp).

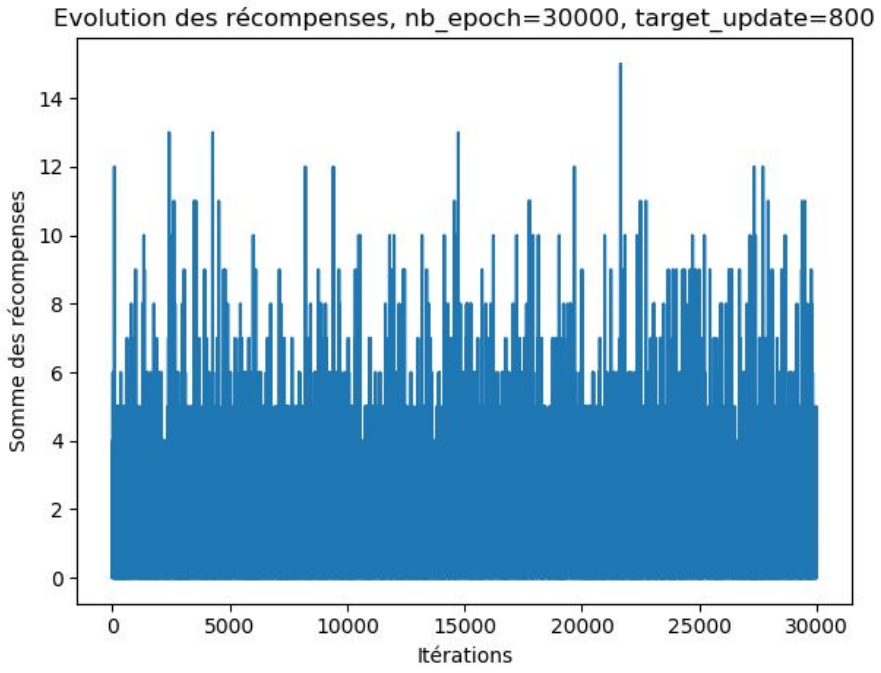
- Utiliser la descente de gradient avec les paramètres par défaut.
- Utiliser ADAM avec les paramètres par défaut.

Néanmoins, ces 3 configurations ont données des résultats similaires (voir *3.2 - Discussion des résultats)*

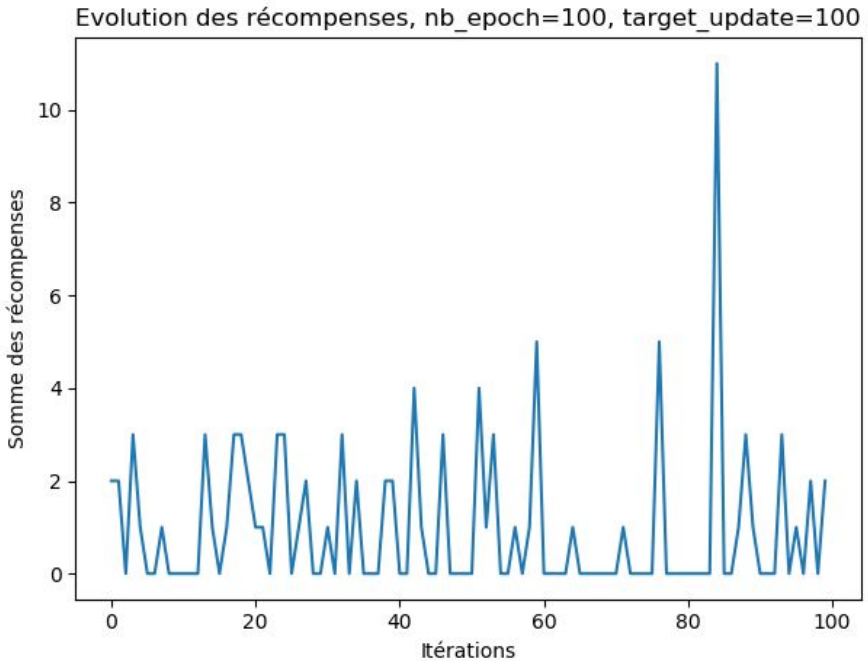
3.1.5 - Question 5 : Vidéo de l'agent & Sauvegarde du réseau

Les méthodes de sauvegarde et de chargement d'un réseau de neurones sont disponibles dans la classe de l'agent : *save* et *load*.
La méthode *save* permet d'inclure le nombre d'epochs réalisées jusque là, afin de pouvoir reprendre un entraînement ou simplement de conserver une trace des paramètres d'apprentissage.
La méthode *load* se charge de recréer le réseau de neurones de l'agent en fonction du (chemin de) fichier passé en paramètre.
Il est possible d'utiliser le script *eval_model.py* pour charger un de nos model et observer son comportement.

3.2 - Discussion des résultats



Les résultats de notre agent sur le *Breakout* sont particulièrement mauvais. En réalisant 30 000 parties, celui-ci n'a rien appris. Nous pensions au début que cela était dû à la manière dont nous le faisions apprendre, nous avons ainsi corrigé notre code afin de respecter la méthode d'apprentissage décrite dans l'article de recherche.



Les 100 parties de Breakout ont généré plus de 23700 frames, soit autant de pas d'apprentissage pour l'agent.

Toutefois, le comportement de l'agent n'a montré aucune amélioration significative. Nous avons pensé à un certain nombre de raisons pour lesquelles notre agent n'apprend rien telles que :

- Le temps d'apprentissage : Il est indiqué dans le papier que la stratégie ε-greedy utilisée repose sur un ε qui décroit linéairement pendant 1 millions de frames. Avec les quelques 23000 frames que nous avons, il semble normal que nous n'atteignons pas les résultats du papier. Toutefois, nous devrions percevoir une évolution de l'apprentissage, ce qui n'est pas le cas ici.
- Une mise à jour du *target network* trop rapide : Tandis que les rédacteurs du papier précisent utiliser un *target update* de 10 000, notre *target update* est limité à 100. Néanmoins, cette différence devrait principalement se ressentir dans l'instabilité du réseau. L'agent devrait donc pouvoir réaliser des bons scores, qui seraient suivis par des scores plus mitigés (voire mauvais).
- Une erreur d'implémentation : Bien que ceci n'ait pas fait partie de nos premières idées, cette piste s'est avérée prometteuse lorsque nous avons relancé un *simple_agent* sur l'environnement *CartPole-v1*. En effet, celui-ci ne montrait que des comportements erratiques, bien loin des résultats que nous avions pu en tirer jusque là. En cherchant plus avant, nous avons fini par constater que c'était l'apprentissage par batch qui perturbait l'agent. Il est donc fort probable que nos mauvais résultats sur le *Breakout* provenaient d'un problème à ce niveau-là.

Note de fin : Même si cela n'impactera pas notre note, nous comptons relancer des tests sans l'apprentissage par batch et laisser un message sur le Readme du projet si c'était effectivement la source du problème.