Rapport TP IBI DRL – Najib EL KHADIR 1511175 M2 IA 2020/21

1 – Préliminaires

Le code source du projet est disponible sur mon dépôt public : https://github.com/NajibXY/DRL

J'ai utilisé Python 3.8, PyTorch dans un environnement virtuel Pipenv.

La méthodologie de réplicabilité est décrite dans README.md et les dépendances sont enregistrées dans requirements.txt.

2 – Deep Q-network sur CartPole

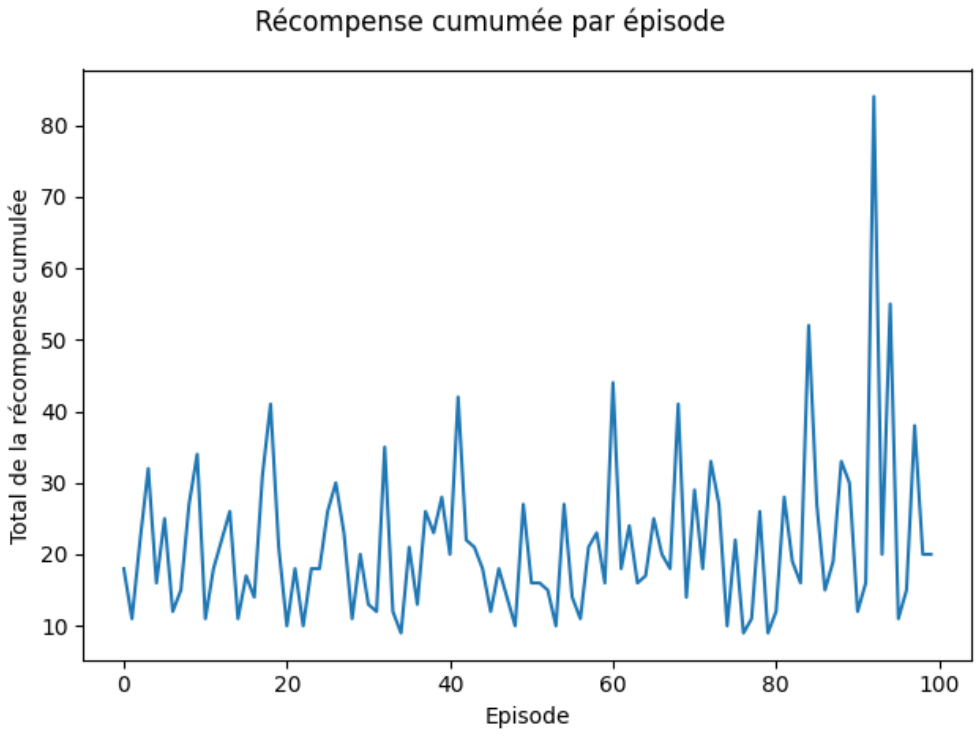
2.1 – Début

- Le code de l'agent aléatoire est disponible dans le fichier random_agent.py. On peut changer l'environnement dans lequel il agit en modifiant la valeur default de la ligne :

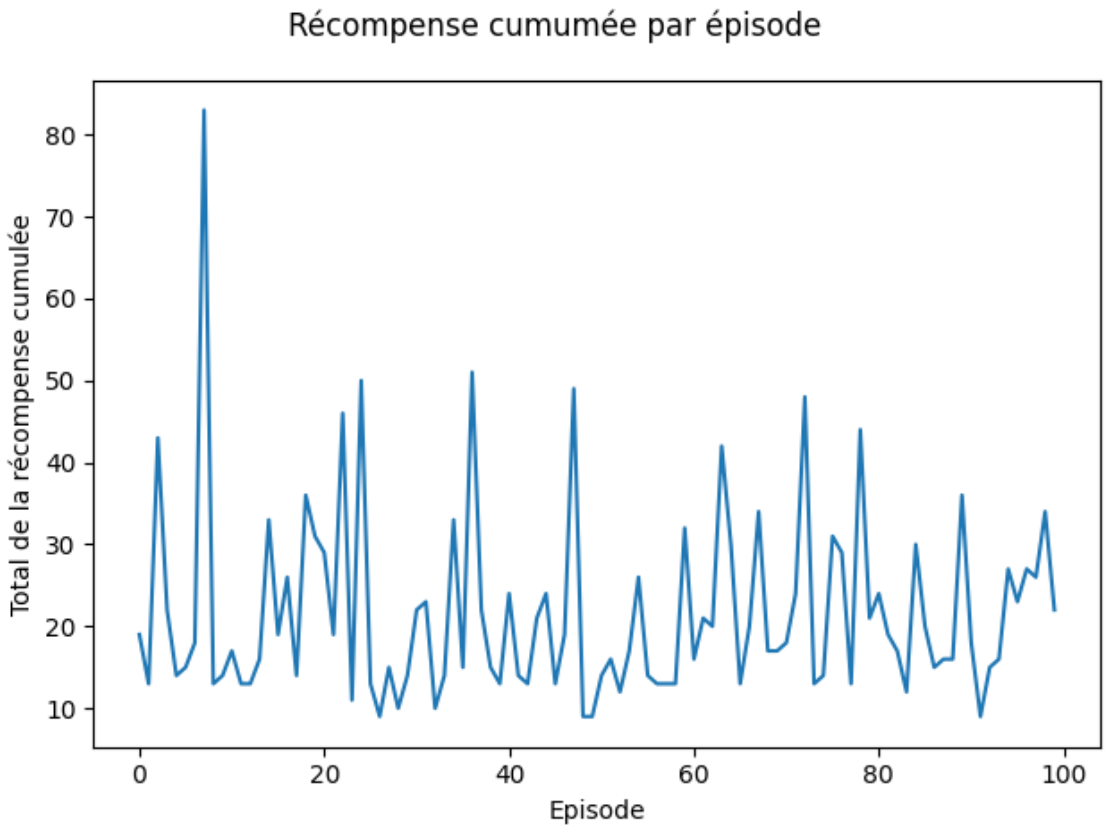
```
parser.add_argument('env_id', nargs='?', default='CartPole-v1',
help='Select the environment to run')
```

- Afin de visualiser l'évolution de la récompense, j'ai utilisé matplotlib pour afficher la récompense cumulée par épisode afin d'avoir un aperçu global des performances de l'agent sur un total de 100 épisodes

Exemple de résultat n°1 :



Exemple de résultat n°2 :



Sans surprise, les résultats sont chaotiques et oscillent aléatoirement, il n'y a pas d'apprentissage.

2.2 – Experience replay

Donnée = Interaction

La structure d'une interaction est définie par la classe Interaction en tant que tuple classique (e, a, s, r, f).

Buffer

La classe Buffer implémente quant à elle le buffer, un buffer a un size et un tableau d'interactions (données) duquel on pop l'interaction à l'indice 0 à chaque fois que l'on rajoute une interaction si cela nous fait dépasser la limite de taille.

La classe buffer possède donc une méthode .append(), mais également une méthode .sample() permettant d'implémenter la fonction de sampling pour récupérer un minibatch de données aléatoires dans les interactions stockées par le buffer, et ce en utilisant la méthode .sample(x, n) du package random où n est la taille du sample : `random.sample(self.buffr, ).`

2.3 – Deep Q-learning

Le code de cette partie a été implémentée dans le fichier dql.py

Question 5

Le code du réseau de neurones est disponible dans la classe QModel, ce réseau est composé d'une couche d'entrée de taille 4 (un état de 4 dimensions), 3 couches fully connected de 32 neurones, et la couche de sortie de 2 neurones (les 2 actions possibles en sortie).

Question 6

J'ai implémenté deux stratégies d'exploration : e-greedy et boltzmann, pour en choisir une il faut modifier la ligne correspondante dans la méthode d'initialisation de la classe DQN_Agent. Le code des deux stratégies est implémenté dans la fonction .act de cette même classe. Pendant l'apprentissage les actions aléatoires sont favorisées et progressivement l'agent se tourne vers les actions plus efficaces.

Question 7

Pour favoriser la convergence vers des maximums locaux et accélérer le temps de calcul tensoriel de PyTorch, je calcule l'erreur sur l'ensemble du minibatch en une seule passe sur le réseau. Le code de l'algorithme d'apprentissage est implémenté dans la méthode learn de la classe DQL_Agent.

Question 8

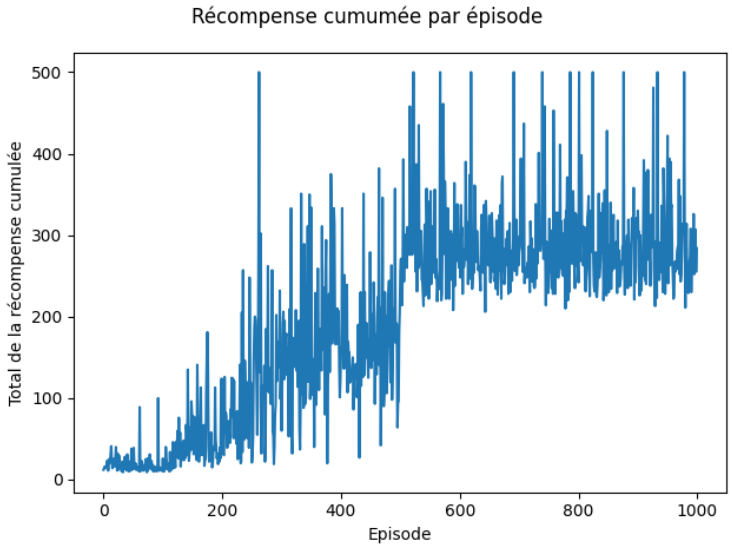
Les deux approches de mise à jour ont été implémentées :

- Avec les hyperparamètres alpha pour l'approche de Polyak
- Avec la fréquence de recopiage pour l'approche par copie

La méthode learn de DQL_Agent implémente également la mise à jour du Target Network.

Après manipulation des hyperparamètres, j'ai pu obtenir un résultat satisfaisant sur 500 épisodes d'entrainement et 500 épisodes de test :

Avec la stratégie e-greedy :



Résultats obtenus avec la stratégie e-greedy et les paramètres suivants :

```
self.gamma = 0.95
self.freq_copy = 1000
self.tau = 1
self.tau_decay = 0.999
self.min_tau = 0.2
self.sigma = 1e-3
self.alpha = 0.01
```

Désolé je n'ai pas pu continuer sur la partie breakout car je n'ai pas réussi à faire fonctionner VizDoom sur Windows malgré des heures d'essais des solutions de mes camarades et de celles proposées par le professeur ou par le net… Et mon ordinateur étant en train de doucement rendre l'âme, je n'ai même pas pu mettre en place une VM pour le faire tourner sur un linux virtuel.

FIN