Apprentissage profond par renforcement
Compte-rendu de TP

Nelly Barret et Juliette Reisser

4 janvier 2020

1 Préliminaires

Notre TP se trouve à l'adresse suivante : https://github.com/NellyBarret/IA5-TP-APR

Nous avons principalement utilisé les libraires suivantes :

— Gym pour les environnements d'apprentissage
— Keras (de Tensorflow) pour les modèles de réseaux neuronaux
— Numpy pour les calculs
— Matplotlib pour les tracés de courbe

Nous allons d'abord définir quelques variables communes aux différentes implémentations. Ces variables font partie de la logique même utilisée par Gym.

Nous avons accès à deux variables importantes quant à la définition de l'environnement :

— `env.action_space` qui définit les actions possibles pour l'agent. Chaque action est un entier, e.g. 0 pour aller à gauche, 1 pour aller à droite.
— `env.observation_space` qui définit un tableau représentant les métriques importantes de l'environnement, e.g. la position d'un élément, les frames d'un jeu...

Nous avons aussi 3 variables importantes quant à l'exécution d'actions sur l'environnement :

— `next_state` qui définit le nouvel état (après exécution de l'action sur l'environnement)
— `reward` représente la récompense gagnée par l'agent après exécution de l'action sur l'environnement
— `done` est un booléen pour indiquer si l'épisode courant est fini (i.e. le bâton est tombé ou sorti de l'environnement)

Enfin, nous avons deux méthodes importantes quant à la communication environnement/agent :

— `act()` qui retourne l'action choisie par l'agent. C'est dans cette fonction que nous implémenterons les politiques (de sélection d'action), i.e. aléatoire, ϵ-greedy et Boltzmann.
— `step(action)` qui exécute une action sur l'environnement et retourne les 3 variables expliquées ci-dessus.

2 Deep Q-network sur Cartpole

2.1 Début

2.1.1 Question 1

Fichier correspondant : randomCartpole.py

2.1.1.1 Définition de l'environnement

Dans l'environnement CartPole (variable `env`), nous avons un bâton posé en équilibre sur un élément que l'on peut faire bouger à gauche ou à droite pour rééquilibrer le bâton. L'objectif principal pour l'agent est de maintenir le bâton assez vertical pour que celui-ci ne tombe pas et/ou ne sorte pas de l'environnement.

Nous avons accès à deux variables importantes quant à la définition de l'environnement :

- Ici, l'espace des actions est de taille 2 car l'agent peut faire bouger l'élément à gauche (action 0) ou à droite (action 1).
- L'espace des états est de taille 4 car il est défini comme suit : [position de l'élément, vitesse de l'élément, angle du bâton, taux de rotation du bâton]

Enfin, nous avons deux méthodes importantes quant à la communication environnement/agent :

- `act()` utilise une politique aléatoire
- `step(action)`

2.1.1.2 Définition de l'agent

L'agent aléatoire suit une politique aléatoire pour choisir l'action qu'il va exécuter dans l'environnement. Il les choisit parmi ses actions possibles (disponibles dans `env.action_space`). L'agent aléatoire choisit simplement des actions des manière aléatoire puis les exécute sur l'environnement.

```
# (1)
env = gym.make("CartPole-v1")
agent = RandomAgent(env.action_space)

# (2)
nb_episodes = 100

# (3)
for i in range(nb_episodes):
    state = env.reset()
    # (4)
    while True:
        action = agent.act() # (5)
        _, reward, done, _ = env.step(action) # (6)
        if done: # (7)
            break
    env.close()
```

Listing 1 – Programme principal de l'agent aléatoire

Partie code :

```
# (1)
```

Nous créons un environnement Gym avec comme paramètre le nom de l'environnement à créer. Nous créons ensuite notre agent aléatoire. Celui-ci redéfinit les méthodes suivantes :

— `act()` : cette méthode renvoie une action parmi les actions possibles

— `sample()`. La p

```
# (2)
```

```
# (3)
```

```
# (4)
```

```
# (5)
```

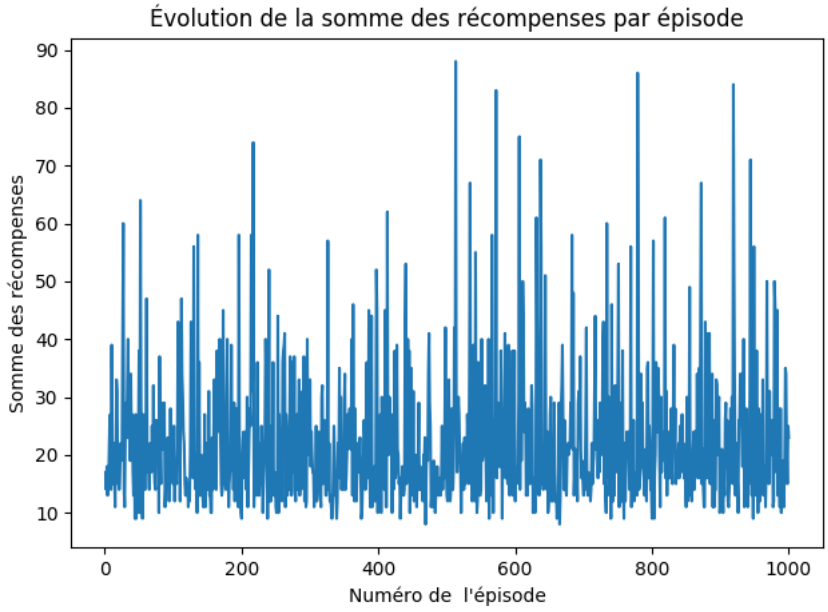
```
# (6)
```

```
# (7)
```

2.1.2 Question 2

Fichier correspondant : randomCartpole.py

Pour évaluer l'agent, nous avons réalisé un graphique traçant la somme des récompenses obtenues pour chaque épisode.



Nous pouvons observer qu'il n'y a pas de tendance particulière. Cela semble cohérent avec le fait que l'agent n'a aucun moyen d'apprendre de ses erreurs (pas de mémoire ni de rétro-propagation) et qu'il ne choisisse pas les meilleures actions (politique aléatoire).

2.2 Experience replay

2.2.1 Questions 3 et 4

Fichier correspondant : ExperienceReplayAgent.py

Dans cet agent, nous avons d'abord créé un objet Memory qui représente la mémoire de l'agent. Cette mémoire stocke les informations (i.e. expériences) sous la forme d'un tableau d'éléments :

$$[state, action, reward, next_state, done]$$

Le tableau de toutes ces expériences représente la mémoire de l'agent.

Sur cette mémoire, nous avons principalement deux fonctions :

- *L'ajout d'une nouvelle expérience* : cette fonctionnalité permet à l'agent de se souvenir de ce qu'il vient d'expérimenter dans l'environnement. Comme la mémoire a une taille limitée, le nouveaux éléments remplacent les anciens quand celle-ci est pleine.
- *La création d'un batch* : cette fonctionnalité permet de choisir des éléments de manière aléatoire dans la mémoire et de retourner ce batch d'expériences.

2.3 Deep Q-learning

2.3.1 Question 5

2.3.2 Question 6

2.3.3 Question 7

2.3.4 Question 8

3 Breakout Atari

3.0.1 Question 1

3.0.2 Question 2

3.0.3 Question 3

3.0.4 Question 4

3.0.5 Question 5