

# Apprentissage profond par renforcement

## Compte-rendu de TP

Nelly Barret et Juliette Reisser

5 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 :

- L'**espace des actions** (`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.
- L'**espace des observations**, ou **espace d'états**, (`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` qui représente la **récompense** gagnée par l'agent après exécution de l'action sur l'environnement
- `done` qui est un booléen indiquant si l'épisode courant est fini, e.g. le bâton est tombé

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,  $\epsilon$ -greedy et Boltzmann.
- `step(action)` qui exécute une action sur l'environnement et retourne les 3 variables expliquées ci-dessus.

Maintenant que nous avons défini les variables importantes, nous allons définir le fonctionnement général des agents.

```

1 env = gym.make("nom-environnement")
2 agent = Agent()
3
4 Pour chaque episode dans (0, nb_max_episodes), Faire
5     Tant que True, Faire:
6         action = agent.act()
7         next_state, reward, done, info = env.step(action)
8         Si done == True, Faire:
9             break

```

Listing 1 – Pseudo-code du fonctionnement de l’agent dans son environnement

Dans un premier temps il est important de créer l’environnement voulu dans la variable `env` et de créer un agent. C’est cet agent que nous allons entraîner dans l’environnement. Le principe de l’entraînement de l’agent est le suivant : l’agent va interagir avec l’environnement pendant `nb_max_episodes` épisodes. Un épisode correspond à toutes les actions que l’agent va pouvoir faire jusqu’à ce qu’une condition ne soit plus respectée (e.g. le bâton est tombé). Un épisode est donc une boucle qui se termine grâce à une condition, ici c’est `done`. Durant cette boucle, l’agent choisit une nouvelle action à faire grâce à sa fonction `act()` puis l’exécute dans l’environnement avec la fonction `step(action)`. Pour résumer, un épisode est défini par la boucle `Tant que` et l’agent s’entraîne pendant `nb_max_episodes` épisodes.

## 2 Agent aléatoire sur Cartpole

Fichier correspondant : [randomCartpole.py](#)

### 2.1 Définitions et fonctionnalités

#### 2.1.1 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 allons maintenant spécifier les variables que nous avons défini en section 1.

- 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]
- La méthode `act()` implémente une politique aléatoire (c.f. section 2.1.2)

#### 2.1.2 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`). Il n’a pas de mémoire de ses précédentes expériences, ne prend pas en compte les gains futurs, ... . En somme, il exécute simplement des actions choisies aléatoirement.

```

1 env = gym.make("CartPole-v1") # (1)
2 agent = RandomAgent(env.action_space) # (1)
3 nb_episodes = 1000 # (2)
4 liste_rewards = [] # (3)
5 for i in range(nb_episodes):
6     total_reward = 0
7     env.reset()
8     while True: # (4)
9         action = agent.act()
10        _, reward, done, _ = env.step(action)
11        total_reward += reward
12        if done:
13            break
14        liste_rewards.append(total_reward) # (3)
15 evolution_rewards(liste_rewards) # (3)
16 print("Meilleure recompense obtenue", max(liste_rewards), "lors de
    l'episode", liste_rewards.index(max(liste_rewards))) # (3)
17 env.close()

```

Listing 2 – Programme principal de l'agent aléatoire

Le code ci-dessus montre l'instanciation du pseudo-code de l'entraînement d'un agent (en section 1).

1 # (1)

Nous créons un environnement Gym avec comme paramètre le nom de l'environnement à créer (ici, "CartPole-v1"). Nous créons ensuite notre agent aléatoire. Comme vu précédemment, celui-ci redéfinit la méthode `act()`.

1 # (2)

Nous devons ensuite choisir le nombre d'épisode sur lequel l'agent va s'entraîner. Comme l'agent a une politique aléatoire, le nombre d'épisode n'est pas un facteur influent sur les performances de l'agent puisque celui-ci exécute toujours des actions choisies aléatoirement. Nous prendrons comme paramètre 1000, ce qui permet de vérifier que l'agent interagisse correctement avec l'environnement sans pour autant que cela n'augmente (trop) le taux d'exécution.

1 # (3)

Cette partie traite de l'évolution de la somme des récompenses perçues par l'agent. Pour chaque épisode nous ajoutons la récompense obtenue pour l'action choisie ce qui permet d'avoir une récompense « globale » par épisode. La fonction `evolution_rewards` permet de tracer la courbe correspondant à ces récompenses par épisode.

Nous avons aussi récupéré la meilleure somme de récompense obtenue ainsi que le numéro de l'épisode où elle a été obtenue (ligne 16).

1 # (4)

Dans cette partie là du code, nous rentrons dans un épisode. Dans cet épisode, nous devons choisir une action puis l'exécuter dans l'environnement. La première étape est faite par la méthode `act()`. Comme l'agent suit une politique aléatoire celle-ci retourne simplement une action parmi celles de l'espace d'actions de l'agent. Cela se traduit par le code ci-dessous :

```

1 class RandomAgent:
2     """
3     Agent qui choisit des actions de maniere aleatoire
4     """

```

```

5  def __init__(self, action_space):
6      """
7      Initialisation generale
8      """
9      self.action_space = action_space
10
11  def act(self):
12      """
13      Choisit une action aleatoirement parmi l'espace d'actions
14      """
15      return self.action_space.sample()

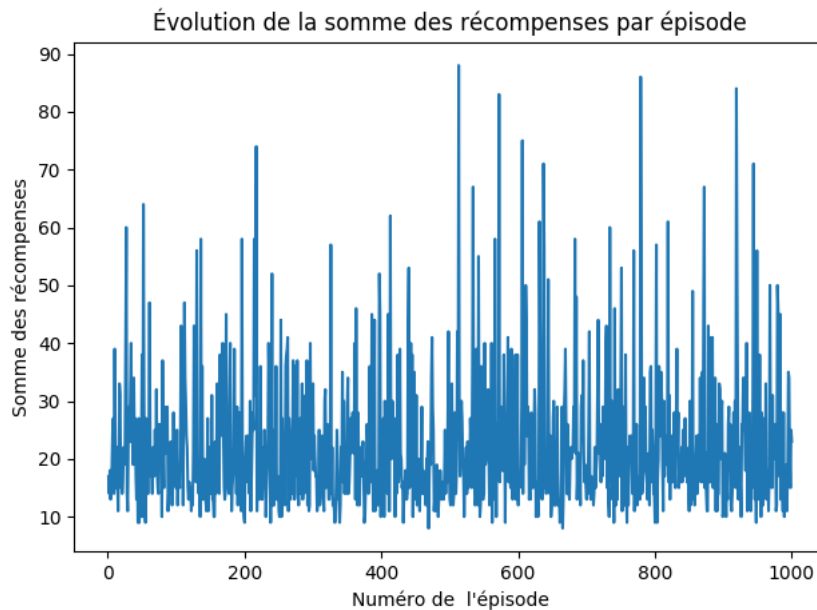
```

Listing 3 – Implémentation de l'agent aléatoire

Après avoir choisi une action, l'agent l'exécute dans l'environnement, ce qui lui permettra d'obtenir une récompense (ajoutée à la somme des récompenses de l'épisode en cours) et le booléen `done`. Si ce booléen s'évalue à vrai, l'épisode se termine et le suivant commence. Sinon l'épisode continue et l'agent choisit une nouvelle action.

## 2.2 Analyse de performances

Pour évaluer notre agent aléatoire, 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).

## 2.3 Experience replay

Fichier correspondant : [ExperienceReplayAgent.py](#)

Cet agent choisit aléatoirement ses actions et implémente une mémoire ce qui lui permettra par la suite d'apprendre par rapport à ses expériences passées.

### 2.3.1 Définition de la mémoire

```
1 class Memory:
2     """
3     Classe representant la memoire de l'agent
4     """
5     def __init__(self, max_size, batch_size):
6         """
7         Initialise la memoire de l'agent
8         @param max_size: taille maximale de la memoire
9         @param batch_size: taille du batch genere
10        """
11        self.max_size = max_size
12        self.memory = [[] for _ in range(self.max_size)]
13        self.position = 0
14        self.batch_size = batch_size
15
16    def add(self, state, action, reward, next_state, done):
17        """
18        Ajoute une experience a la memoire de l'agent
19        @param state: etat courant de l'agent
20        @param action: action choisie par l'agent
21        @param reward: recompense gantee
22        @param next_state: etat d'arrivee (apres action)
23        @param done: True si l'experience est finie
24        """
25        self.memory[self.position] = [state, action, reward,
26        next_state, done]
27        self.position = (self.position + 1) % self.max_size
28
29    def sample(self):
30        """
31        Construit un batch aleatoire sur la memoire de l'agent
32        :return: le batch d'experiences
33        """
34        if self.__len__() < self.batch_size:
35            return None
36        else:
37            batch = random.sample(self.memory, self.batch_size)
38            return batch
39
40    def __len__(self):
41        """
42        Retourne le nombre d'elements (non nuls) dans la memoire
43        :return: le nombre d'elements dans la memoire
44        """
45        return sum(len(item) > 0 for item in self.memory)
```

Listing 4 – Implémentation de la mémoire d'un agent

Cette classe définit la mémoire de l'agent. Elle stocke les expérience que l'agent a avec l'environnement. La mémoire est donc modélisée par une liste d'expériences. Chaque expérience est une liste de 5 éléments : l'état courant de l'agent (variable `state`), l'action choisie par l'agent (variable `action`), la récompense obtenue pour cette action (variable `reward`), l'état dans lequel va arriver l'agent après exécution de l'action (variable `next_state`) et la variable `done` qui indique si l'agent a terminé l'épisode ou non.

$$expe_i = [state_i, action_i, reward_i, nextstate_i, done_i]$$

Cette mémoire a un nombre maximal d'éléments, donc quand un nouvel élément est inséré et qu'il n'y a plus de place, ce nouvel élément remplace le plus ancien. Elle est donc composée comme suit :

$$memory = \bigcup_{i=1}^{max\_size} expe_i = \begin{bmatrix} [state_1, action_1, reward_1, nextstate_1, done_1] \\ \dots \\ [state_n, action_n, reward_n, nextstate_n, done_n] \end{bmatrix}$$

De plus, nous avons deux fonctions pour interagir avec la mémoire :

- *L'ajout d'une nouvelle expérience* via la fonction `add(...)` : cela ajoute à la mémoire de l'agent ce qu'il vient d'expérimenter dans l'environnement. Ce processus lui permettra par la suite d'apprendre de ses expériences passées.
- *La création d'un batch* via la fonction `sample()` : cette fonctionnalité crée un batch d'expériences en choisissant des éléments de manière aléatoire dans la mémoire.

Enfin, deux conditions sont à respecter :

1. Le dépassement de la taille maximale doit être prévu. Il est prévu par le modulo utilisé pour la position de la nouvelle expérience à insérer. En effet, quand le buffer arrive à sa capacité, le modulo repart à 0, ce qui permet de remplacer les plus anciennes expériences par les nouvelles.
2. La mémoire est indépendante de l'environnement, et donc de l'espace d'action et d'état. Cette condition est respectée du fait d'insérer au fur et à mesure dans la mémoire et de n'avoir qu'une taille maximale comme condition.

### 2.3.2 Définition de l'agent

```

1 class ExperienceReplayAgent:
2     """
3     Agent qui choisit des actions de maniere aleatoire
4     """
5     def __init__(self, action_space, batch_size):
6         """
7         Initialisation generale
8         @param action_space: espace d'actions (0 ou 1)
9         @param batch_size: taille du batch
10        """
11        self.action_space = action_space
12        self.batch_size = batch_size
13        self.memory = Memory(100, self.batch_size)
14        self.position = 0
15
16    def act(self):
17        """
18        Choisit une action aleatoirement parmi l'espace d'actions
19        """
20        return self.action_space.sample()
21
22    def remember(self, state, action, reward, next_state, done):
23        """

```

```

24     Ajoute une interaction a la memoire de l'agent
25     @param state: etat courant
26     @param action: action effectuee
27     @param reward: recompense recue de l'environnement
28     @param next_state: etat dans lequel on arrive
29     @param done: pour arreter l'agent quand il a fini
30     """
31     self.memory.add(state, action, reward, next_state, done)
32
33     def creer_batch(self):
34         """
35         Cree un batch de taille self.batch_size sur la base de la
36         memoire
37         @return le batch
38         """
39         return self.memory.sample()

```

Listing 5 – Implémentation de l'agent utilisant son expérience

Comme vu précédemment, cet agent redéfinit la méthode `act()` et se base toujours sur une politique aléatoire. Cet agent a une fonctionnalité supplémentaire, celle de se souvenir d'une expérience grâce à la fonction `remember(...)` qui permet d'ajouter une expérience à sa mémoire, comme définit ci-dessus.

```

1 env = gym.make("CartPole-v1")
2 agent = ExperienceReplayAgent(env.action_space, 20)
3
4 nb_episodes = 100
5 for i in range(nb_episodes):
6     env.reset()
7     while True:
8         action = agent.act()
9         next_state, reward, done, _ = env.step(action)
10        agent.remember(state, action, reward, next_state, done)
11        if done:
12            break
13    batch = agent.creer_batch()
14    env.close()

```

Listing 6 – Programme principal de l'agent utilisant l'expérience replay

Enfin, nous devons modifier quelque peu le programme principal afin que l'agent enregistre les nouvelles expériences qu'il a avec l'environnement. Le squelette (créations de l'environnement et de l'agent, boucle sur le nombre d'épisodes, boucle pour chaque épisode) ne change pas. En revanche l'agent ajoute chacune de ses expériences dans sa mémoire grâce à la fonction `remember(...)` qui ajoute la nouvelle expérience à sa mémoire. Après avoir réalisé tous les épisodes, l'agent peut créer le batch sur sa mémoire. La création du batch est simplement un tirage aléatoire de  $n$  expériences où  $n$  est la taille du batch. Par exemple, ici

nous créons un batch de 20 expériences (défini lors de la création de l'agent).

## **2.4 Deep Q-learning**

**2.4.1 Question 5**

**2.4.2 Question 6**

**2.4.3 Question 7**

**2.4.4 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**