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

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

```
env = gym.make("nom-environnement")
agent = Agent()

Pour chaque episode dans (0, nb_max_epsiodes), Faire
    Tant que True, Faire:
        action = agent.act()
        next_state, reward, done, info = env.step(action)
        Si done == True, Faire:
            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_epsiodes` 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 d'ans 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.

```
env = gym.make("CartPole-v1")  # (1)
agent = RandomAgent(env.action_space)  # (1)
nb_episodes = 1000 # (2)
liste_rewards = [] # (3)
for i in range(nb_episodes):
    total_reward = 0
    env.reset()
    while True: # (4)
        action = agent.act()
        _, reward, done, _ = env.step(action)
        total_reward += reward
        if done:
            break
        liste_rewards.append(total_reward) # (3)
evolution_rewards(liste_rewards)  # (3)
print("Meilleure recompense obtenue", max(liste_rewards), "lors de
    l'episode", liste_rewards.index(max(liste_rewards))) # (3)
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)
```

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()`.

```
# (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.

```
# (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).

```
# (4)
```

Dans cette partie là du code, nous rentrons dans un épisode. Dans cet épisode, nous devrons 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 :

```
class RandomAgent:
    """
    Agent qui choisit des actions de maniere aleatoire
    """
```

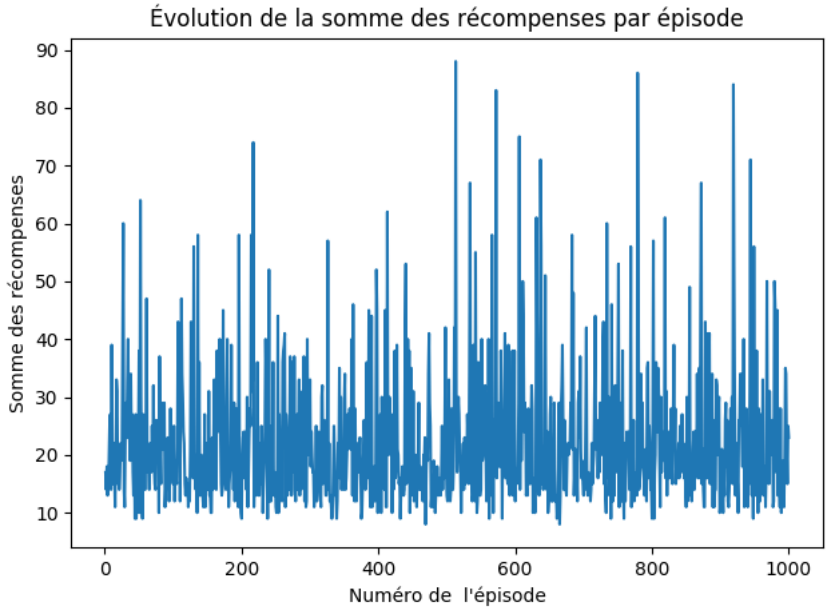
```
    def __init__(self, action_space):
        """
        Initialisation generale
        """
        self.action_space = action_space

    def act(self):
        """
        Choisit une action aleatoirement parmi l'espace d'actions
        """
        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

```python
class Memory:
    """
    Classe representant la memoire de l'agent
    """
    def __init__(self, max_size, batch_size):
        """
        Initialise la memoire de l'agent
        @param max_size: taille maximale de la memoire
        @param batch_size: taille du batch genere
        """
        self.max_size = max_size
        self.memory = [[] for _ in range(self.max_size)]
        self.position = 0
        self.batch_size = batch_size

    def add(self, state, action, reward, next_state, done):
        """
        Ajoute une experience a la memoire de l'agent
        @param state: etat courant de l'agent
        @param action: action choisie par l'agent
        @param reward: recompense gagnee
        @param next_state: etat d'arrivee (apres action)
        @param done: True si l'experience est finie
        """
        self.memory[self.position] = [state, action, reward,
    next_state, done]
        self.position = (self.position + 1) % self.max_size

    def sample(self):
        """
        Construit un batch aleatoire sur la memoire de l'agent
        :return: le batch d'experiences
        """
        if self.__len__() < self.batch_size:
            return None
        else:
            batch = random.sample(self.memory, self.batch_size)
            return batch

    def __len__(self):
        """
        Retourne le nombre d'elements (non nuls) dans la memoire
        :return: le nombre d'elements dans la memoire
        """
        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 :

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

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

```
class ExperienceReplayAgent:
    """
    Agent qui choisit des actions de maniere aleatoire
    """
    def __init__(self, action_space, batch_size):
        """
        Initialisation generale
        @param action_space: espace d'actions (0 ou 1)
        @param batch_size: taille du batch
        """
        self.action_space = action_space
        self.batch_size = batch_size
        self.memory = Memory(100, self.batch_size)
        self.position = 0

    def act(self):
        """
        Choisit une action aleatoirement parmi l'espace d'actions
        """
        return self.action_space.sample()

    def remember(self, state, action, reward, next_state, done):
        """
```

```
        Ajoute une interaction a la memoire de l'agent
        @param state: etat courant
        @param action: action effectuee
        @param reward: recompense recue de l'environnement
        @param next_state: etat dans lequel on arrive
        @param done: pour arreter l'agent quand il a fini
        """
        self.memory.add(state, action, reward, next_state, done)

        def creer_batch(self):
        """
        Cree un batch de taille self.batch_size sur la base de la memoire
        @return le batch
        """
        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.

```
env = gym.make("CartPole-v1")
agent = ExperienceReplayAgent(env.action_space, 20)

nb_episodes = 100
for i in range(nb_episodes):
    env.reset()
    while True:
        action = agent.act()
        next_state, reward, done, _ = env.step(action)
        agent.remember(state, action, reward, next_state, done)
        if done:
            break
    batch = agent.creer_batch()
    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).

3 Deep Q-learning sur CartPole

Dans la section précédente nous avons défini le problème du CartPole ainsi qu'un agent basique qui choisit aléatoirement ses actions. Nous lui avons ensuite ajouté une mémoire en vu qu'il puisse apprendre de ses expériences passées.

3.1 Construction du modèle neuronal

3.1.1 Définition du modèle

Dans un premier temps, nous avons construit notre modèle neuronal. Celui-ci se compose d'une taille d'entrée égale à la taille d'un état (en l'occurence 4 pour CartPole) et d'une taille de sortie égale aux nombre d'actions (2 pour CartPole). Nous avons une seule couche cachée de taille 24 et qui a pour fonction d'activation `relu`, i.e. que les neurones s'activent sur les entrées positives car la fonction relu peut se formaliser ainsi :

$$relu(x) = \max\{0, x\} \text{ où } x \text{ est une entrée}$$

3.1.2 Paramétrage du modèle

Afin de trouver les meilleurs paramètres pour notre modèle neuronal, nous avons effectué plusieurs tests. Ci-dessous un tableau récapitulant les paramètres testés ainsi que leurs résultats :

3.1.3 Définition de l'agent

3.1.4 Paramétrage de l'agent

Nous avons plusieurs paramètres à prendre en compte et à ajuster à la résolution du problème de CartPole. Ces paramètres sont les suivants :

- La taille de la mémoire (`memory_size`) : si la mémoire est trop petite, les batchs construits auront beaucoup de ressemblance (du fait de la petite taille de l'espace de tirage) donc le réseau apprendra peu. Par défaut la taille de la mémoire est fixée à 100000.
- La taille du batch (`batchsize`) : c'est sur ce batch que le réseau va se mettre à jour et donc apprendre. Si le batch est trop petit, le réseau apprendra peu ; s'il est trop grand, le temps d'apprentissage sera considérable.
- Le gamma qui définit l'importance des récompenses à l'infini. Plus ce paramètre est proche de 1, plus l'agent aura tendance à attendre une meilleure récompense dans les états futurs (il privilégie une plus grosse récompense lointaine). Inversement, plus ce paramètre est proche de 0 plus l'agent va privilégier les récompenses proches. Nous l'avons défini à 0.99 afin de prendre en compte les récompenses lointaines.
- Le taux d'apprentissage (`learning_rate`) : nous l'avons défini à 0.001, valeur communément acquise par la communauté scientifique. Ce taux permet de plus ou moins apprendre l'erreur (entre la prédiction que le réseau a fait et la valeur cible).
- Le taux d'exploration (`exploration_rate`) est utilisé dans les stratégies ϵ-greedy et Boltzmann. Ce taux permet de choisir si l'agent va faire une action aléatoire ou l'action qu'il prédit comme meilleure. Dans le premier cas, cela permet de diversifier l'apprentissage, i.e. d'explorer de nouveaux états. Dans le second, cela permet d'intensifier l'apprentissage, i.e. de rester dans des états proches pour augmenter la somme des récompenses. Ce taux débute à 1 puis est diminué par le facteur `exploration_decay` (0.995) jusqu'à son minimum `exploration_min` (0.01). Cela permet notamment de

faire baisser le nombre d'explorations au fur et à mesure que l'agent apprend et devient meilleur.

— nbepisodes 200

— updatetargetnetwork 100

```
class DQNAgent:
    """
    Classe representant l'agent DQN et son reseau
    """
    def __init__(self, params):
        """
        Initialise le reseau et l'agent
        @param params: dictionnaire contenant les parametres du reseau et de l'agent
        """
        self.state_size = params['state_size']  # taille de l'entree du reseau
        self.action_size = params['action_size']  # taille de sortie du reseau

        self.memory = Memory(params['memory_size'], params['batch_size'])  # memoire pour l'experience replay
        self.batch_size = params['batch_size']

        self.gamma = params['gamma']
        self.learning_rate = params['learning_rate']
        self.exploration_rate = params['exploration_rate']
        self.exploration_decay = params['exploration_decay']
        self.exploration_min = params['exploration_min']

        self.model = self.build_model()
        self.target_model = self.build_model()

    def build_model(self):
        """
        Construit le modele neuronal
        """
        model = Sequential()
        model.add(Dense(24, input_dim=self.state_size, activation='relu'))
        model.add(Dense(24, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
        model.compile(loss='mse', optimizer=Adam(lr=self.learning_rate))
        return model

    def act(self, state, policy="greedy"):
        """
        Choisit une action pour l'etat donne
        @param state: etat courant de l'agent
        @param policy: la politique utilisee par l'agent
        """
        if policy == "greedy":
            if numpy.random.rand() < self.exploration_rate:
                return random.randrange(self.action_size)
            else:
                q_values = self.model.predict(state)
                return numpy.argmax(q_values[0])
        elif policy == "boltzmann":
            if numpy.random.rand() <= self.exploration_rate:
                return env.action_space.sample()
            else:
```

```
                tau = 0.8
                q_values = self.model.predict(state)
                sum_q_values = 0
                boltzmann_probabilities = [0 for _ in range(len(
    q_values[0]))]
                for i in range(len(q_values[0])):
                    sum_q_values += numpy.exp(q_values[0][i] / tau)
                for i in range(len(q_values[0])):
                    current_q_value = q_values[0][i]
                    boltzmann_probabilities[i] = numpy.exp(
    current_q_value/tau) / sum_q_values
                 return numpy.argmax(boltzmann_probabilities)
        else:
            return env.action_space.sample()

    def remember(self, state, action, reward, next_state, done):
         """
        Ajoute une interaction a la memoire de l'agent
        @param state: etat courant
        @param action: action effectuee
        @param reward: recompense recue de l'environnement
        @param next_state: etat dans lequel on arrive
        @param done: pour arreter l'agent quand il a fini
        """
        self.memory.add(state, action, reward, next_state, done)

    def experience_replay(self):
        """
        Calcule les predictions, met a jour le modele et entraine
    le reseau
        """
        # states, q_val = [], []
        # batch = self.memory.sample()  # creation du batch a
    partir de la memoire de l'agent
        # if batch is not None:
        #     for state, action, reward, next_state, done in batch:
        #         # predictions des q-valeurs pour toutes les
    actions de l'etat
        #         q_values = self.model.predict(state)
        #         # mise a jour de la Q-valeur de l'action de l'
    etat
        #         if done:
        #             q_values[0][action] = reward
        #         else:
        #             q_values[0][action] = reward + self.gamma *
    numpy.max(self.target_model.predict(next_state)[0])
        #         states.append(state[0]) # contient tous les etats
        #         q_val.append(q_values[0])  # contient les
    predictions des q_valeurs
        #         # TODO: a quel endroit ?
        #         # self.model.fit(state, q_values, batch_size=len(
    states), verbose=0)
        #         pred = self.model.predict(state)
        #         y = reward + self.gamma * numpy.max(self.
    target_model.predict(next_state)[0])
        #         self.model.fit(pred, y, batch_size=self.
    state_size, verbose=0)
        #     # mise a jour du reseau sur le batch
        #     # self.model.fit(numpy.array(states), numpy.array(
    q_val), batch_size=len(states), verbose=0)
        #     if self.exploration_rate > self.exploration_min:
        #         self.exploration_rate *= self.exploration_decay
```

```
102         if len(self.memory) < self.batch_size:  # self.memory.
    batch_size:
103             return
104         # batch = self.memory.sample()
105         batch = random.sample(self.memory, self.batch_size)
106         for state, action, reward, state_next, done in batch:
107             q_update = reward
108             if done:
109                 q_values[0][action] = reward
110             else:
111                 q_values[0][action] = reward + self.gamma * numpy.
    max(self.target_model.predict(next_state
112             q_values = self.model.predict(state)
113             q_values[0][action] = q_update
114             self.model.fit(state, q_values, verbose=0)
115         self.exploration_rate *= self.exploration_decay
116         self.exploration_rate = max(self.exploration_min, self.
    exploration_rate)
117 %TODO : comparer dfit dans et hors du for
118
119
120     def update_target_network(self):
121         """
122         Met a jour le target model a partir du model
123         """
124         self.target_model.set_weights(self.model.get_weights())
125
126
127 if __name__ == '__main__':
128     env = gym.make('CartPole-v1')
129
130     # constantes pour l'agent DQN
131     state_size = env.observation_space.shape[0]
132     action_size = env.action_space.n
133     memory_size = 100000
134     batch_size = 64  # 64
135     gamma = 0.99  # 0.99  # importance des recompenses a l'infini
136     learning_rate = 0.001  # taux d'apprentissage de l'erreur entre
     la cible et la prediction
137     exploration_rate = 1  # pour savoir si on prend une action
    random ou la meilleure action
138     exploration_decay = 0.995  # pour faire descendre l'
    exploration_rate pour baisser le nombre d'explorations au fur
    et a mesure que l'agent apprend et devient meilleur
139     exploration_min = 0.01
140
141     # constantes pour l'execution
142     nb_episodes = 200
143     update_target_network = 100  # pas pour mettre a jour le target
     network
144     save_weights = False  # True pour sauvegarder les poids du
    reseau dans un fichier tous les save_step episodes
145     save_step = 10  # pas pour sauvegarder les poids du reseau
146
147     # creation de l'agent avec ses parametres
148     params = {
149         'state_size': state_size,
150         'action_size': action_size,
151         'memory_size': memory_size,
152         'batch_size': batch_size,
153         'gamma': gamma,
154         'learning_rate': learning_rate,
```

```
155         'exploration_rate': exploration_rate,
156         'exploration_decay': exploration_decay,
157         'exploration_min': exploration_min
158     }
159     agent = DQNAgent(params)
160     liste_rewards = []  # liste des recompenses obtenues pour
    chaque episode, permet de tracer le plot
161     global_counter = 0
162     for i in range(nb_episodes):
163         state = env.reset()
164         # [ 0.0273208   0.01715898 -0.03423725  0.01013875] => [[
    0.0273208   0.01715898 -0.03423725  0.01013875]]
165         state = numpy.reshape(state, [1, env.observation_space.
    shape[0]])  # TODO: pour avoir un vecteur de 1
166         steps = 1
167         sum_reward = 0
168         while True:
169             action = agent.act(state, "greedy")  # choix d'une
    action (greedy: soit aleatoire soit via le reseau)
170             next_state, reward, done, _ = env.step(action)  # on "
    execute" l'action sur l'environnement
171             next_state = numpy.reshape(next_state, [1, env.
    observation_space.shape[0]])  # TODO:
172             agent.remember(state, action, reward, next_state, done)
173             state = next_state
174             sum_reward += reward
175             agent.experience_replay()
176             if done:
177                 print("Episode", i, "- nombre de pas : ", steps, "-
     somme recompenses", sum_reward)
178                 break
179             if global_counter % update_target_network == 0:
180                 # on met a jour le target network tous les `
    update_target_network` pas
181                 print("Le target network se met a jour")
182                 agent.update_target_network()
183             steps += 1
184             global_counter += 1
185         liste_rewards.append(sum_reward)
186         if save_weights and i % save_step == 0:
187             print("Sauvegarde des poids du modele")
188             agent.model.save_weights("./cartpole_dqn.h5")
189     evolution_rewards(liste_rewards)
190     print("Meilleure recompense obtenue", max(liste_rewards), "lors
     de l'episode", liste_rewards.index(max(liste_rewards)))
```

Listing 7 – Programme principal de l'agent utilisant l'expérience replay