Apprentissage profond par renforcement
Compte-rendu de TP

Nelly Barret et Juliette Reisser

5 janvier 2020

Table des matières

1 Préliminaires

Notre TP se trouve à l'adresse suivante : https://github.com/NellyBarret/IA5-TP-APR. Pour des raisons de clarté et de concision, la documentation ainsi que

les commentaires ont été retirés du code présent dans ce rapport. Pour les mêmes raisons, certaines parties de code ont étés remplacés par des commentaires (repérables par un `##`).

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

La suite de cette section décrit les préliminaires du fonctionnement des agents implémentés. Elle se trouve en annexe (section 6) si besoin.

2 Compilation et lancement

Chaque agent est implémenté dans un fichier qui lui est propre. Ainsi il suffit de lancer l'exécution du fichier voulu dans un IDE (e.g. Pycharm). La fonction `main` de chaque fichier permet de créer l'environnement et l'agent ainsi que de faire apprendre l'agent sur l'environnement.

3 Agent aléatoire sur Cartpole

Fichier correspondant : randomCartpole.py

3.1 Définitions et fonctionnalités

3.1.1 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 3.1.2)

3.1.2 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")
agent = RandomAgent(env.action_space)
nb_episodes = 1000 # (1)
liste_rewards = [] # (2)
for i in range(nb_episodes):
    total_reward = 0
    env.reset()
    while True: # (3)
        action = agent.act()
        _, reward, done, _ = env.step(action)
        total_reward += reward
        if done:
            break
        liste_rewards.append(total_reward) # (2)
evolution_rewards(liste_rewards)  # (2)
print("Meilleure recompense obtenue", max(liste_rewards), "lors de
    l'episode", liste_rewards.index(max(liste_rewards))) # (2)
env.close()
```

Listing 1 – 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 devons 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 influant 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 de manière significative le temps d'exécution.

```
# (2)
```

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).

```
# (3)
```

Dans cette partie là du code, nous rentrons dans un épisode. Dans cet épisode, l'agent devra 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 :

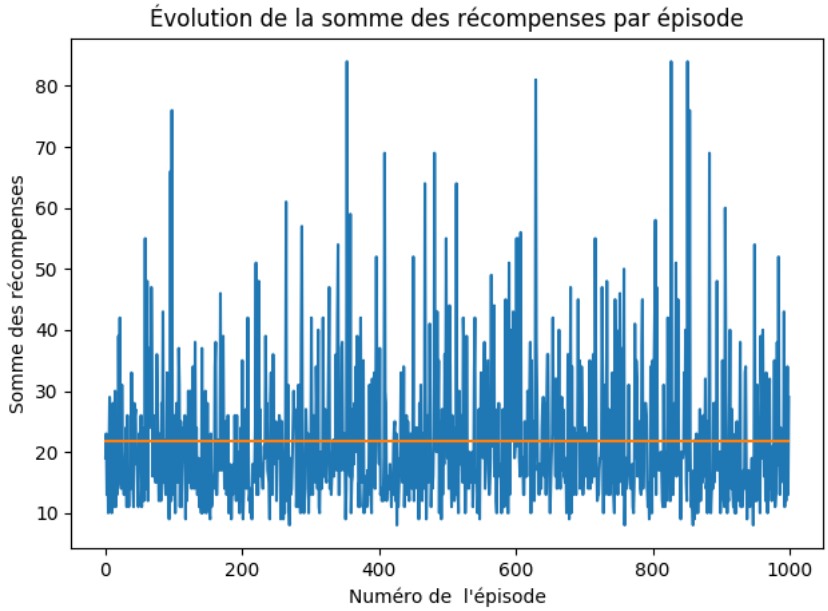
```
class RandomAgent:
    def __init__(self, action_space):
        ## assignation des parametres a leur variable eponyme

    def act(self):
        return self.action_space.sample()
```

Listing 2 – Implémentation de l'agent aléatoire

Après avoir choisi une action, l'agent l'exécute dans l'environnement via la fonction `step`, 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.

3.2 Analyse de performances

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



La courbe bleue représente la somme des récompenses obtenue pour chaque épisode. La courbe orange représente la moyenne des récompenses pour les `nb_episodes` épisodes. 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).

3.3 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.

3.3.1 Définition de la mémoire

```
class Memory:
    def __init__(self, max_size, batch_size):
        ## assignation des parametres a leur variable eponyme
        self.memory = [[] for _ in range(self.max_size)]
        self.position = 0
```

```
6
7       def add(self, state, action, reward, next_state, done):
8           self.memory[self.position] = [state, action, reward,
    next_state, done]
9           self.position = (self.position + 1) % self.max_size
10
11      def sample(self):
12          if self.__len__() < self.batch_size:
13              return None
14          else:
15              batch = random.sample(self.memory, self.batch_size)
16              return batch
17
18      def __len__(self):
19          return sum(len(item) > 0 for item in self.memory)
```

Listing 3 – Implémentation de la mémoire d'un agent

Cette classe définit la mémoire de l'agent. Elle stocke sous forme de liste les expériences que l'agent a avec l'environnement. 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. Une expérience se formalise ainsi :

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

Cette mémoire a un nombre maximal d'éléments (variable `max_size`), donc quand un nouvel élément est inséré et qu'il n'y a plus de place, ce nouvel élément remplace le plus ancien. Elle se formalise 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 la mémoire arrive à sa capacité maximale, la position revient 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 par l'insertion au fur et à mesure dans la mémoire et le fait de n'avoir qu'une taille maximale comme condition.

3.3.2 Définition de l'agent

```
class ExperienceReplayAgent:
    def __init__(self, action_space, batch_size):
        ## assignation des parametres a leur variable eponyme
        self.memory = Memory(100, self.batch_size)

    def act(self):
        return self.action_space.sample()

    def remember(self, state, action, reward, next_state, done):
        self.memory.add(state, action, reward, next_state, done)

    def creer_batch(self):
        return self.memory.sample()
```

Listing 4 – 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.

```
## creation de l'environnement et de l'agent
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 5 – 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(...)`. 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).

4 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. Nous allons maintenant lui faire apprendre de ses expériences en lui implémentant un algorithme d'apprentissage profond de type Q-learning.

4.1 Construction du modèle neuronal

4.1.1 Définition du modèle

Pour le code, voir la fonction `build_model()` en 4.2.1.

Dans un premier temps, nous avons construit le modèle neuronal de notre agent. La taille de l'entrée est égale à la taille d'un état (en l'occurence 4 pour CartPole) et la taille de sortie est 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 :

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

Nous avons aussi une fonction d'erreur, ici nous avons choisi `mse`, i.e. mean squared error. Cette fonction d'erreur calcule la moyenne des différences au carré entre les valeurs prédites et les valeurs cibles. Cette fonction est la métrique par défaut lorsqu'il s'agit de réseaux neuronaux de ce type, c'est pourquoi nous avons choisi celle-ci.

4.1.2 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 :

4.2 Construction de l'agent

4.2.1 Définition de l'agent

La construction de cet agent se base donc sur un modèle de réseau neuronal et une mémoire. Les paramètres utilisés sont détaillés dans la section suivante en 4.2.2

```
class DQNAgent:
    def __init__(self, params):
        ## assignation des parametres a leur variable eponyme

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

    def build_model(self):
        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

  # (1)
    def act(self, state, policy="greedy"):
        if policy == "greedy":
            ## politique e-greedy
        elif policy == "boltzmann":
            ## politique Boltzmann
```

```
        else:
            return env.action_space.sample()

    def remember(self, state, action, reward, next_state, done):
        self.memory.add(state, action, reward, next_state, done)

  # (2)
    def experience_replay(self):
        ## experience replay avec calcul des q-valeurs

    # (3)
    def update_target_network(self):
        self.target_model.set_weights(self.model.get_weights())
```

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

```
# (1)
```

Voir section 4.2.3

```
# (2)
```

Voir section 4.2.4

```
# (3)
```

Voir section 4.2.5

Le programme principal de l'agent DQN reprend le pseudo-code donné en section 1. Pour chaque épisode, l'agent choisit une action, la réalise dans l'environnement, ajoute cette nouvelle expérience à sa mémoire et comptabilise ses récompenses.

```
## creation de l'environnement et de l'agent avec ses parametres
liste_rewards = []
global_counter = 0
    for i in range(nb_episodes):
        state = env.reset()
        steps = 1
        sum_reward = 0
        while True:
            action = agent.act(state, "greedy")
            next_state, reward, done, _ = env.step(action)
            agent.remember(state, action, reward, next_state, done)
            state = next_state
            sum_reward += reward
            agent.experience_replay()
            if done:
                print("Episode", i,"- nombre de pas : ", steps, "-
    somme recompenses", sum_reward)
                break
            if global_counter % update_target_network == 0:
                agent.update_target_network()
            steps += 1
            global_counter += 1
        liste_rewards.append(sum_reward)
        if save_weights and i % save_step == 0:
            agent.model.save_weights("./cartpole_dqn.h5")
    evolution_rewards(liste_rewards)
    print("Meilleure recompense obtenue", max(liste_rewards), "lors
     de l'episode", liste_rewards.index(max(liste_rewards)))
```

Listing 7 – Programme principal de l'agent DQN

4.2.2 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.
- Le nombre d'épisode est maintenant mis à 200 car le problème du CartPole est considéré comme résolu si un score de 195 a été atteint avant 200 épisodes.
- Le target network est mis à jour toutes les 100 itérations (au global). Cela permet au réseau de se remettre à jour assez souvent pour apprendre mais de rester dans des temps d'apprentissage raisonnables.

```
if __name__ == '__main__':
    env = gym.make('CartPole-v1')

    # constantes pour l'agent DQN
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
    memory_size = 100000
    batch_size = 64
    gamma = 0.99
    learning_rate = 0.001
    exploration_rate = 1
    exploration_decay = 0.995
    exploration_min = 0.01

    # constantes pour l'execution
    nb_episodes = 200
```

```
    update_target_network = 100
    save_weights = False
    save_step = 10

    # creation de l'agent avec ses parametres
    params = {
        'state_size': state_size,
        'action_size': action_size,
        'memory_size': memory_size,
        'batch_size': batch_size,
        'gamma': gamma,
        'learning_rate': learning_rate,
        'exploration_rate': exploration_rate,
        'exploration_decay': exploration_decay,
        'exploration_min': exploration_min
    }
    agent = DQNAgent(params)
```

Listing 8 – Paramétrage de l'agent DQN

4.2.3 Politiques

La fonction `act(...)` définit maintenant deux politiques : ϵ-greedy et Boltzmann (et aléatoire le cas échéant). La politique est donnée comme paramètre à la fonction `act(...)` et peut prendre comme valeurs `greedy` ou `boltzmann`. Nous allons maintenant détailler chacune de ces deux politiques.

La politique ϵ-greedy permet soit d'explorer les états en renvoyant une action aléatoire soit d'intensifier l'apprentissage en renvoyant la meilleure action prédite par la réseau. En comparaison de la poitique greedy, elle a l'avantage d'explorer les possibilités en choisissant des actions aléatoires.

```
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])
```

Listing 9 – Politique e-greedy

La politique de Boltzmann permet de choisir une action a_k selon la probabilité définie comme suit :

$$P(s, a_k) = \frac{e^{\frac{Q(s,a_k)}{\tau}}}{\sum_i e^{\frac{Q(s,a_i))}{\tau}}}$$

Plus τ est grand, plus la politique aura tendance à être aléatoire puisque le quotient $\frac{Q(s,a_k)}{\tau}$ tendra vers 0 donc la quotient donnant la probabilité de Boltzmann donnera des probabilités assez proches. Avec des probabilités proches, cela revient à choisir une action aléatoirement puisqu'aucune ne se distingue des autres. Inversement, plus τ est petit, plus les écarts entre probabilités seront forts, ce qui permettra de choisir la meilleure action avec une grande probabilité.

```
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
```

```
7       boltzmann_probabilities = [0 for _ in range(len(q_values[0]))]
8       for i in range(len(q_values[0])):
9           sum_q_values += numpy.exp(q_values[0][i] / tau)
10      for i in range(len(q_values[0])):
11          current_q_value = q_values[0][i]
12          boltzmann_probabilities[i] = numpy.exp(current_q_value/tau)
            / sum_q_values
13      return numpy.argmax(boltzmann_probabilities)
```

Listing 10 – Politique de Boltzmann

4.2.4 Calcul des Q-valeurs

Le calcul des Q-valeurs se base sur l'équation de Bellman qui se définit ainsi :

$$Q(s, s', a, a') = r(s, a) + \gamma * max_{a'} Qt(s', a')$$

où Q est le réseau de neurones, Qt le target network, s et s' des états, a et a' des actions, r la fonction de récompense.

L'idée de cette fonction est d'apprendre sur les expériences passées. Dans un premier temps, on construit le batch sur lequel on va apprendre. Ensuite, en le parcourant, on calcule les Q-valeurs de chaque action pour un état donné en appliquant la formule de Bellman. Quand l'épisode se termine, l'équation de Bellman se résume à la récompense puisqu'il n'existe pas de prochain état. La ligne 3 prédit les valeurs des différentes actions avec le modèle courant. Après avoir calculé la Q-valeur de l'action, on met à jour le réseau (ligne 8). Enfin, le réseau réajuste ses poids grâce à la fonction `fit`. Après avoir parcouru tout le batch, on met à jour le taux d'exploration (voir section 4.2.2).

```
1  ## creation du batch dans la variable batch
2  for state, action, reward, state_next, done in batch:
3    q_values = self.model.predict(state)
4    if done:
5      q_update = reward
6    else:
7      q_update = reward + self.gamma * numpy.max(self.target_model.
       predict(next_state)[0])
8    q_values[0][action] = q_update
9    self.model.fit(state, q_values, verbose=0)
10 ## mise a jour du taux d'exploration
```

Listing 11 – Expérience repaly de l'agent DQN

4.2.5 Target model

Nous avons pu observer que l'agent apprend des comportements intéressants mais qu'il n'était pas très stable. Dans cette optique, nous avons ajouté un second réseau (`target_network`) dans le but de stabiliser le réseau. Ce réseau se base sur le même modèle que l'original. La mise à jour est une recopie des poids du réseau de base dans le target network (voir section 4.2.2).

5 Breakout Atari

Dans le problème du Breakout, le but est d'éliminer toutes les briques en les touchant avec une balle. Le balle rebo,ndit sur une plateforme et c'est cette

plateforme qu'il faut drigier à droite et à gauche pour faire rebondir la balle et continuer à jouer. Un comportement intéressants serait de casser les briques d'un côté pour faire passer la balle derrière le mur de briques.

6 Annexes

6.1 Préliminaires - suite

Dans cette section préliminaire, nous allons 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 défini 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 12 – 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.