Apprentissage par renforcement multi-agents, De SlimeVolley à la RoboCup : Représentation d'un environnement avec Gym

Pélagie Alves Elias Debeyssac Alexis Hoffmann Alexis Lhéritier
Nicolas Majorel Sébastian Pagès

Février 2021

Contents

1 Représentation d'un environnement avec Gym

Il y a deux concepts clés dans l'apprentissage par renforcement, l'environnement et l'agent. (voir Apprentissage par renforcement pour les définitions). Gym fournit principalement une abstraction de la représentation d'un environnement [2] avec l'interface Env, il faut donc implémenter cette interface pour pouvoir créer un environnement et interagir avec celui-ci , respecter les consignes de la documentation [3] pour qu'il soit correctement interprétable par les différents algorithmes disponibles dans la bibliothèque stable_baselines.
En revanche, Gym laisse la liberté à l'utilisateur de représenter un agent de la manière dont il souhaite, il n'y a pas de restrictions.

Il faut définir plusieurs attributs de base qui sont réutilisés dans les algorithmes d'apprentissage par renforcement pour déterminer l'action la plus adéquate, par exemple pour l'algorithme PPO qui cherche à optimiser la politique, une règle utilisée par un agent pour décider des actions à entreprendre.

1.1 Attributs à définir

- L'**espace d'observations** (observation_space).
- L'**espace d'actions** (action_space).

Ces espaces sont representés par des classes concrètes [4] implémentant l'interface Space [5] du module Gym.

	Space Types	About
1.	**Box**	A *n* dimensional box, where each coordinate lies between a bound defined by [low,high].

Independent bound for each dimension:
>>> space = gym.spaces.Box(low=np.array([-1.0, -2.0]), high=np.array([2.0, 4.0]), dtype=np.float32)
>>> print(space, space.sample())
Box(2,) [0.37700886 -0.12235405], returns a 2 element vector, each corresponding to one dimension, the 1st dimension value is sampled from interval [-1.0, 2.0] and the 2nd dimension sampled from interval [-2.0, 4.0].
Identical bound for each dimension:
>>> space = gym.spaces.Box(low=-1.0, high=2.0, shape=(3, 4), dtype=np.float32)
>>> print(space,space.sample())
Box(3, 4) [[0.29874545 0.84560674 1.102614 1.6697267]
[1.4066843 1.8870455 0.48101822 1.7307336]
[-0.49271297 0.8122731 -0.7641185 -0.45864782]]
returns a tensor or a 3 x 4 multidimensional matrix.		

2.	**Discrete**	The space consists of n distinct points each mapped to an integer value in the interval [0, n-1].
>>> gym.spaces.Discrete (4), **Discrete(4),** the defined space will contain 4 discrete points with each point mapped to an integer in the interval [0,3].		

Figure 1: description

3.	**Dict**	A dictionary of simple space types to make a complex space.
>>> gym.spaces.Dict({"position": gym.spaces.Discrete(2), "velocity": gym.spaces.Discrete(3)}) **Dict(position:Discrete(2), velocity:Discrete(3)),** a dictionary space is created which consists of 2D discrete space for position and a 3D discrete space for velocity.		
4.	**Multi_discrete**	Multi-Dimensional discrete space, consists of a series of discrete action spaces with different number of actions in each. Parameterized by passing an array of positive integers for the number of actions for each discrete action space.
>>> gym.spaces.MultiDiscrete([5,2,3]) **MultiDiscrete([5 2 3]),** defines 3 actions spaces of 5,2 and 3 actions each.		
5.	**Multi_binary**	Defines an n-dimensional binary space, where the argument to **MultiBinary()** defines the number of dimensions n.
>>> gym.spaces.MultiBinary(6) **MultiBinary(6),** defines a 6 dimensional binary space.		
6.	**Tuple**	Tuple is a product of simple spaces.
>>> gym.spaces.Tuple((gym.spaces.Discrete(2), gym.spaces.Discrete(3))) **Tuple(Discrete(2), Discrete(3))**		

Figure 2: description

Tel que précisé dans la documentation de gym , il est préférable d'éviter les versions différentes d'espaces et réutiliser les espaces existants spécifiés dans l'interface Space du module Gym pour permettre la comptabilité avec la plupart des modèles.

1.2 Méthodes à implémenter[1]

- **step(self, action)**:

  La fonction accepte une action fournie par l'agent et retourne un tuple *(observation, reward, done, info)*.

  Entrées :

  – action : une action fournie par l'agent

  Sorties :

  – observation(object): observation de l'agent dans l'environnement actuel.
  – reward(float) : montant de la récompense retournée après l'action précédente.
  – done(bool): si l'épisode est terminé, dans ce cas, plusieurs appels à la suite de cette fonction peuvent engendrer des comportements indéterminés.
  – info(dict) : contient des informations auxiliaires.

- **reset(self)**:

  Réinitialise l'environnement à son état initial et renvoie une observation de cet état.
  Cette fonction ne doit pas réinitialiser le générateur de nombres aléatoires de l'environment (graine, seed).
  Les variables aléatoires dans l'état de l'environnement doivent être échantillonnées indépendamment entre plusieurs appels à 'reset ()'. En d'autres termes, chaque appel de 'reset ()'

devrait produire un environnement convenable pour un nouvel épisode, indépendant des épisodes précédents.

Sorties :

- observation(object): l'observation initiale.

- **close(self)**: Permet de détruire l'environnement.

- **seed(self, seed=None)**:

  Définit la graine pour le(s) générateur(s) de nombre aléatoires de cet environnement.

Ci-dessous, un exemple simple d'utilisation de l'environnement Gym:

```
import gym
env = gym.make('CartPole-v0')
for i_episode in range(20):
    observation = env.reset()
    for t in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)
        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
env.close()
```

Figure 3: description[6]

2 Représentation de l'environnement SlimeVolleyBall

2.1 Éléments de l'environnement

- Les éléments fixes : on distingue le filet, le sol et les bordures de la fenêtre, plus concrètement ces éléments définissent le perimètre de jeu des slimes et de la balle.
  Seulement le filet est un objet à part entière avec des coordonnées, une largeur et une hauteur. Le sol et les bordures de la fenêtre sont des variables pour contrôler le périmètre de l'espace de jeu.

- Les éléments dynamiques : les slimes et la balle (coordonnées, dimensions, vitesse), parmi ces éléments seulement les slimes sont des agents, la balle ne décide pas de l'action à effectuer, elle n'interprète pas l'environnement pour modifier sa trajectoire, il n'y a pas non plus de notions de récompenses pour celle-ci, elle suit seulement une trajectoire qui dépend de sa vitesse et des interactions avec l'environnement.
  Par défaut, l'agent que l'on entraîne est celui à droite, l'agent à gauche est contrôlé par un réseau de neurones [7] (jeu JavaScript de 2015).

- Le générateur de nombres aléatoires : dans SlimeVolleyBall, à chaque nouvelle manche, la balle est propulsée depuis une position fixe du terrain à une certaine vitesse. Cette vitesse est déterminée par une séquence prédéfinie de paramètres dépendant de la graine (seed) utilisée. L'idée d'utiliser une graine est de pouvoir répéter exactement le même

environnement, mais avec des politiques d'actions différentes des agents pour comparer les résultats.

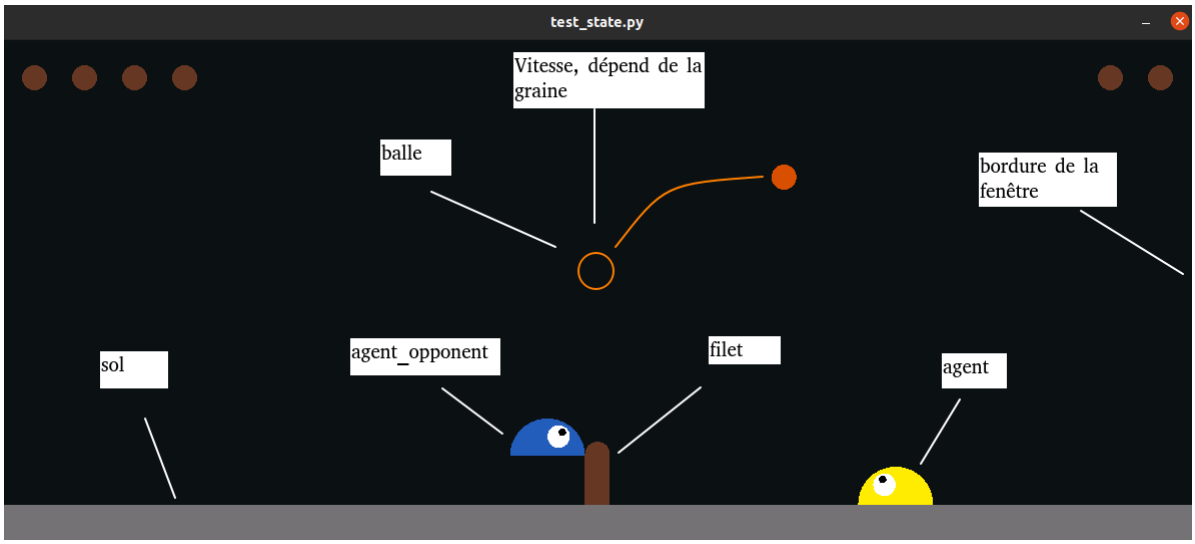

Figure 4: description

2.2 Types d'environnement

On distingue trois types d'environnement dans le jeu SlimeVolleyBall.

Environment Id	Observation Space	Action Space
SlimeVolley-v0	Box(12)	MultiBinary(3)
SlimeVolleyPixel-v0	Box(84, 168, 3)	MultiBinary(3)
SlimeVolleyNoFrameskip-v0	Box(84, 168, 3)	Discrete(6)

Figure 5: différents environnements de SlimeVolleyGym

L'environnement *SlimeVolley-v0* correspond à un environnement avec un **mode d'observation par états**.
L'environnement *SlimeVolleyPixel-v0* correspond à un environnement avec un **mode d'observation par pixels**.
L'environnement *SlimeVolleyNoFrameskip-v0* correspond à un environnement avec un **mode d'observation par pixels (Atari)**.

Le mode d'observation par pixels (Atari) encapsule en fait les comportements du mode d'observation par pixels. Les deux environnements travaillent sur les mêmes données d'observation.
Cependant, les scripts pour entraîner le modèle dans le code base n'utilisent pas le mode d'observation par pixels mais plutôt le mode d'observation par pixels (Atari).
On détaillera par la suite ces différentes espaces.

2.3 Espaces d'observations

- **Mode d'observation par états** : Les deux agents et la balle sont representés par un vecteur à 12 dimensions :

$$\begin{bmatrix} x\_agent & y\_agent & \dot{x}\_agent & \dot{y}\_agent \\ x\_ball & y\_ball & \dot{x}\_ball & \dot{y}\_ball \\ x\_opponent & y\_opponent & \dot{x}\_opponent & \dot{y}\_opponent \end{bmatrix}$$

Ici, x représente la coordonnée abscisse, y la coordonnée ordonnée, $\dot{x}$ la vitesse en abscisse, $\dot{y}$ la vitesse en ordonnée.

Cet espace est représenté par un objet de type *Box* (implémentation de la classe Space du module gym), cet espace représente une "boîte" à n dimensions. Ici, chaque paramètre de la matrice à un intervalle de valeurs acceptées (valeurs basses et hautes).

Ce mode d'observation est utilisé dans l'environnement SlimeVolley-v0.

observation	x_agent	y_agent	$\dot{x}\_agent$	$\dot{y}\_agent$	...
min	-3.4028235e+38	-3.4028235e+38	-3.4028235e+38	-3.4028235e+38	...
max	3.4028235e+38	3.4028235e+38	3.4028235e+38	3.4028235e+38	...

```
import numpy as np
from gym import spaces

    high = np.array([np.finfo(np.float32).max] * 12)
    self.observation_space = spaces.Box(-high, high)
```

- **Mode d'observation par pixels (Atari)**

Entraîner un agent à jouer au Slime Volleyball uniquement à partir d'images pixelisées est plus difficile, car l'agent doit travailler avec un espace d'observations plus grand, mais il doit surtout apprendre à déduire les informations importantes dans les images telles que les position des agents, de la balle, les vitesses qui ne sont pas explicitement fournies. Pour obtenir ces informations, on doit pré-traiter les images du mode d'observations par pixels.

Le pré-traitement standard Atari [8] pour les agents consiste d'abord à convertir chaque image RGB en échelle de gris (on passe de 3 valeurs R = 0-255, G = 0-255, B = 0-255 à une seule valeur de 0-255), à les redimensionner en 84x84 pixels (84x168 à la base), ensuite, 4 images consécutives sont empilées en une seule observation afin que les informations temporelles locales puissent être déduites.

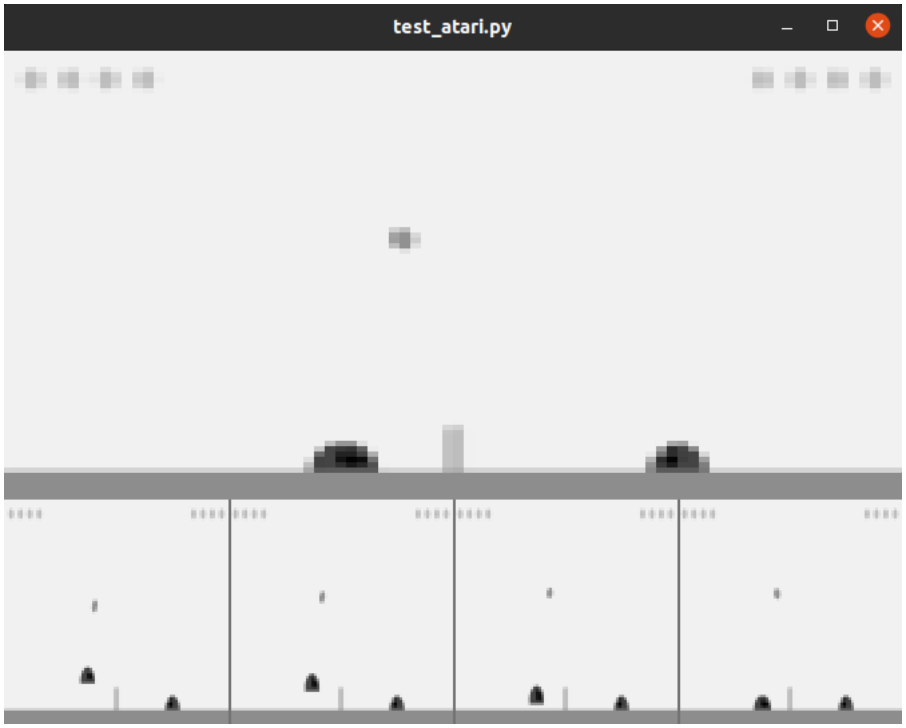

Figure 6: Environnement SlimeVolleyNoFrameskip-v0

Cet environnement est représenté par un objet de type *space.Box*, une matrice de dimensions 84 pixels en hauteur, 168 en largeur en RGB. Il a la particularité de montrer le fonctionnement du pré-traitement des images décrit.

Ce mode est utilisé dans l'environnement SlimeVolleyPixel-v0 et SlimeVolleyNoFrameskip-v0.

Pour de plus amples informations, il faut se diriger vers le script **test_atari** [9] qui crée un environnement SlimeVolleyBall Atari compatible.

```
from gym import spaces

    self.observation_space = spaces.Box(low=0, high=255,
      shape=(PIXEL_HEIGHT, PIXEL_WIDTH, 3), dtype=np.uint8)
```

2.4 Espaces d'actions

L'espace d'action est soit de type Multibinary soit de type Discrete.

Dans le mode MultiBinary on a un espace d'action représenté par un vecteur de booléens à 3 dimensions avec les champ avancer, reculer, sauter (droite, gauche, haut). La valeur est à 1 si l'action est demandée, 0 dans le cas contraire.

```
from gym import spaces
self.action_space = spaces.MultiBinary(3)
```

$$\begin{bmatrix}\text{avancer} = 1 & \text{reculer} = 0 & \text{sauter} = 1\end{bmatrix}$$

Dans l'exemple ci dessus, le slime fait un saut en avant.

Dans le mode Discrete on a un espace d'action représenté par un vecteur d'entier de 0 à 5, chaque action correspond à un chiffre (0=immobile, 1=avancer, 2=saut avant, 3=saut, 4=saut arrière, 5=reculer).
Le mode MultiBinary est utilisé dans SlimeVolley-v0 et SlimeVolleyPixel-v0.
Le mode Discrete est utilisé dans SlimeVolleyNoFrameskip-v0.

2.5 Affichage

On illustre ici, un exemple d'implémentation de la méthode **render(self, mode='human')** de l'interface Env.

Pour stocker les éléments visuels d'une forme géométrique, l'implémentation choisie par le créateur de SlimeVolleyGym est de créer une variable canvas qui va permettre de représenter diverses formes géométriques (rectangle, cercle, demi-cercle).
En fonction du mode d'observations (états, pixels), le type du canvas est différent.
Ci-dessous les fonctions à appeler pour créer et modifier un canvas dans le programme :

```
create_canvas(canvas, c):
rect(canvas, x, y, width, height, color):
half_circle(canvas, x, y, r, color):
circle(canvas, x, y, r, color):
```

- **Affichage : observations par états** :

  Dans le cadre de l'**observations par états**, le canvas est créé en tant que **Viewer** object [10] (Objet natif de gym). Un Viewer object permet par exemple de créer un rectangle à partir de sa largeur et de sa hauteur. Le rectangle est tracé depuis le point en haut à gauche de celui-ci.
  Un exemple d'utilisation pour créer un rectangle et l'afficher :

```
from gym.envs.classic_control import rendering as rendering

    canvas = rendering.Viewer(WINDOW_WIDTH, WINDOW_HEIGHT)
    box = rendering.make_polygon([(0,0), (0,-height),
    (width, -height), (width,0)])
    # ...
    canvas.add_onetime(box)
    canvas.render(return_rgb_array = mode=='rgb_array')
```

- **Affichage : observations par pixels** :

Pour le mode d'**observations par pixels**, le canvas contient une matrice de dimensions [3 n] avec n le nombre de pixels de l'objet et 3 pour chaque valeur RGB de l'image.
Exemple :

$$\begin{bmatrix} 0|12|5 & 145|44|88 & 168|11|3 & 27|255|0 & 42|99|4 \\ 102|88|11 & 0|14|24 & 127|122|211 & 35|44|77 & 2|43|44 \\ 64|64|64 & 42|1|2 & 24|4|9 & 18|255|254 & 0|1|255 \end{bmatrix}$$

Les données affichées sont en fait similaires aux données utilisées pour l'observation.

Le programme utilise le module **OpenCV** [11], celui-ci permet de représenter des formes géométriques sous forme de matrices, mais également réaliser des opérations mathématiques sur des images (retourner, réduire l'image), toutes les structures de tableaux OpenCV sont converties vers et à partir de tableaux Numpy.

Initialisation du canvas dans le programme :

```
import numpy as np
    background_color = (11, 16, 19)
    canvas = np.ones((WINDOW_HEIGHT, WINDOW_WIDTH, 3),
    dtype=np.uint8)
    for background_color in range(3):
        canvas[:, :, background_color] *= c[background_color]
```

Un exemple pour dessiner un rectangle sur le canvas :

```
import cv2
    canvas = cv2.rectangle(canvas, (round(x),
    round(WINDOW_HEIGHT-y)), (round(x+width),
     round(WINDOW_HEIGHT-y+height)),
    color, thickness=-1, lineType=cv2.LINE_AA)
```

Le résulat est interprété avec l'objet **SimpleImageViewer** [10] (Objet natif de gym) à partir de la matrice, on peut l'afficher ensuite :

```
from gym.envs.classic_control import rendering as rendering
    self.viewer = rendering.SimpleImageViewer(maxwidth=2160)
    self.viewer.imshow(canvas)
```

2.6 Exécuter une action dans l'environnement

Voici un exemple d'implémentation de la méthode **step(self, action)** de l'interface Env.
La méthode se contente principalement d'exécuter les actions des agents, le paramètre *action* contient l'action de l'agent de droite, comme spécifier plus haut de base l'agent de gauche est contrôlé par la politique de base du réseau de neurones (objet **BaselinePolicy** [12]), cependant, on a également un hyperparamètre *otherAction* qui permet de modifier le comportement de base

de celui-ci et donc de changer l'action prédite en utilisant un autre algorithme.
Les actions sont donc envoyées au moteur interne du jeu (objet **Game** [12]) qui va contrôler celles-ci (déplacement, gestion des collisions, marquage de points), la fonction qui contrôle tout ce mécanisme renvoie le résultat selon la perspective de l'agent de droite (donc 1 si celui-ci marque un point, -1 s'il perd un point, 0 s'il ne passe rien).
La méthode vérifie ensuite si un des deux agents n'a plus de vies ou si le temps maximum sans marquer de points est dépassé, dans ce cas, le jeu est terminé.
Finalement, la méthode retourne plusieurs informations, comme spécifier plus haut dans la partie "Représentation d'un environnement avec Gym".

Plus précisément, un tuple avec les informations suivantes :

- Un booléen pour indiquer si le jeu est terminé
- Les observations selon la perspective de l'agent de droite :
  - observation par états : même type de données renvoyées (voir "Mode d'observation par états"), cependant les informations renvoyées sont relatives par rapport à l'agent (voir objet **RelativeState** [12]), "un agent jouant d'un côté ou de l'autre du terrain doit voir les observations de la même manière".
  - obervations par pixels : renvoie la matrice qui est spécifiée dans la partie au dessus pour l'affichage.
- La récompense selon la perspective de l'agent de droite
- Un tuple auxiliaire contenant :
  - Nombres de vies restantes pour l'agent de droite
  - Nombres de vies restantes pour l'agent de gauche
  - Les observations selon la perspective de l'agent de gauche :
    - observation par états : relatives par rapport à l'agent de gauche.
    - obervations par pixels : la matrice utilisée pour l'affichage est retournée avec la fonction flip du module **OpenCV**.

2.7 Création et utilisation de la graine

Pour créer un générateur de nombres aléatoires, l'implémentation de la méthode **seed(self, seed=None)** utilise le module **utils** de **Gym** pour créer une seed.

```
from gym.utils import seeding
    self.np_random, seed = seeding.np_random(seed)
    self.game = Game(np_random=self.np_random)
```

Ensuite, à partir de cette seed, on peut générer une séquence prédéfinie de vitesses pour la balle.

```
ball_vx = self.np_random.uniform(low=-20, high=20)
ball_vy = self.np_random.uniform(low=10, high=25)
```

3 Bibliographie

References

[1] OpenAI, Getting Started with Gym, http://gym.openai.com/docs/ (2016).

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, W. Zaremba, Openai gym, https://arxiv.org/pdf/1606.01540.pdf (2016). arXiv:arXiv:1606.01540.

[3] D. Ha, G. Brockman, J. Schulman, T. B. Brown, J. Cooper, G. Powell, B. Kossovics, Y. Li, X. Zuo, W. Douglas, A. Nichol, T. Blackwell, T. Alcorn, S. Sidor, O. Sigaud, M. Plappert, J. Schneider, K. Hartikainen, J. Turner, C. Hesse, A. Gleave, A. Singh, D. Timbrell, S. Mysore, Classe OpenAI Gym principale, https://github.com/openai/gym/blob/master/gym/core.py (2020).

[4] Cban2020, States, observation and action spaces in reinforcement learning, https://medium.com/swlh/states-observation-and-action-spaces-in-reinforcement-learning-569a30a8d (2020).

[5] X. Zuo, T. Deleu, S. Thiagarajan, J. J. Hunt, E. Leurent, C. Hesse, Défini l'espace d'actions et l'espace d'observations, https://github.com/openai/gym/blob/master/gym/spaces/space.py (2020).

[6] OpenAI, OpenAI Gym, https://github.com/openai/gym (2016).

[7] D. Ha, Neural slime volleyball, blog.otoro.net (2015).
URL https://blog.otoro.net/2015/03/28/neural-slime-volleyball/

[8] D. Ha, Pixel Observation Environment, https://github.com/hardmaru/slimevolleygym/blob/master/TRAINING.md (2020).

[9] D. Ha, Environnement d'observations par pixels atari (slimevolleyball) compatible, https://github.com/hardmaru/slimevolleygym/blob/master/test_atari.py (2020).

[10] D. Ha, G. Brockman, T. Blackwell, J. Schneider, J. Schulman, S. kant, O. Klimov, P. Freire, A. Botev, A. S. et Abdelrahman Ahmed, Objet viewer, représentation des formes géométriques, mode d'observation par états, https://github.com/openai/gym/blob/master/gym/envs/classic_control/rendering.py (2020).

[11] A. Mordvintsev, A. K, Gui features in opencv, https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_table_of_contents_gui/py_table_of_contents_gui.html (2013).

[12] D. Ha, Fichier principal pour définir et représenter les environnement slimevolleygym, https://github.com/hardmaru/slimevolleygym/blob/master/slimevolleygym/slimevolley.py (2020).