Apprentissage par renforcement et transfert simulation vers réalité pour la conduite de voitures autonomes

Culture Sciences de l'Ingénieur

La Revue 3E.I

Édité le 23/05/2024

école normale supérieure paris-saclay

Rania BENNANI[1] – Kévin HOARAU[1] – Anthony JUTON[2]

[1] *Elève de 3ème année au département Nikola Tesla de l'École Normale Supérieure Paris-Saclay*
[2] *Professeur agrégé de physique appliquée au département Nikola Tesla de l'École Normale Supérieure Paris-Saclay*

Cette ressource fait partie du N°112 de La Revue 3EI de mai 2024 et s'intègre au « Dossier Intelligence Artificielle » [2] de Culture Sciences de l'Ingénieur.

Cette ressource présente l'apprentissage par renforcement de la conduite sur circuit d'une voiture autonome 1/10ème, en simulation, puis le transfert du réseau de neurones du simulateur dans la voiture réelle, en utilisant Webots, gymnasium et Stable-Baselines3. Elle est issue du travail de Kévin Hoarau lors de sa participation à la Course de Voitures Autonomes de Paris Saclay, CoVAPSy 2023 [6], repris et validé par plusieurs équipes en 2024.

Figure 1 : Voiture en apprentissage sur le simulateur Webots

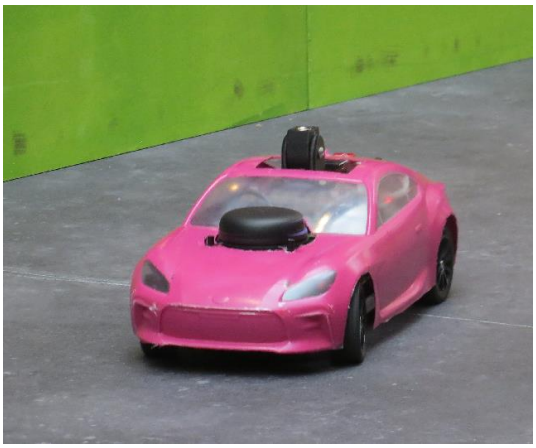

Figure 2 : Voiture réelle, avec le réseau de neurones issu de la simulation

Le simulateur utilisé, Webots, peu gourmand en ressource et open source, permet à chacun d'expérimenter l'apprentissage par renforcement profond sur cet exemple réaliste, même sans disposer de voiture pour le passage à la réalité.

La voiture 1/10ème instrumentée d'un coût modeste (moins de 1000 euros) permet à travers cet exercice d'appréhender le transfert simulation → réalité et les difficultés associées pour une mise en œuvre concrète et matérielle de l'intelligence artificielle. La voiture et le simulateur sont présentés en détails dans les ressources « *Course Voitures Autonomes Paris Saclay (CoVAPSy) : Travaux pratiques autour des voitures autonomes* » [7], « *CoVaPSy : Premiers programmes python sur la voiture réelle* » [8] et « *CoVaPSy : Mise en œuvre du Simulateur Webots* » [9] du numéro 111 de la revue 3EI.

La ressource « *Apprentissage par renforcement de la conduite d'un véhicule sur AirSim* » de Ludovic de Mattteis et Saša Radosalvjevic [3] a servi de point de départ à ce travail. Webots [11] a

été préféré à AirSim pour sa légèreté et sa facilité de mise en œuvre et l'expérience acquise précédemment a permis d'aller jusqu'au transfert de la simulation à la réalité.

1 - Introduction

L'apprentissage par renforcement (*Reinforcement Learning* en anglais) est une catégorie de Machine Learning. L'article « *Introduction à l'apprentissage par renforcement* » [1] *du* « *Dossier Intelligence Artificielle* » [2] présente cette méthode en détail. Cette ressource vient en complément, en proposant un exemple avancé de mise en œuvre de l'apprentissage par renforcement pour la conduite de voitures autonomes réelles sur un circuit.

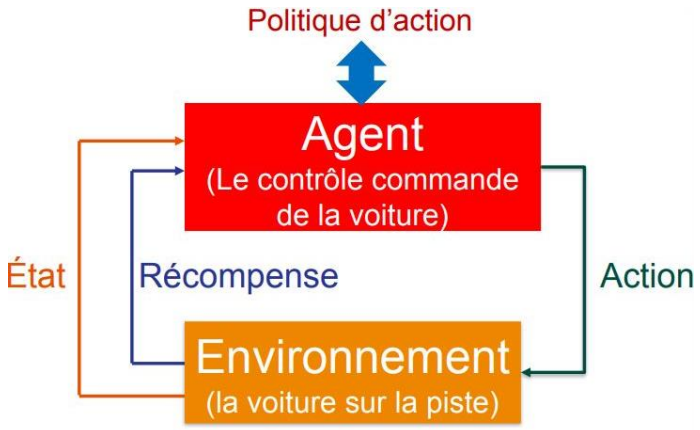

Figure 3 : Schéma explicatif de l'apprentissage par renforcement

En quelques mots, on se place dans un ensemble {**agent**, **environnement**} où une **action** choisie et réalisée par l'agent, en fonction de l'**état** de l'environnement, peut entraîner une modification de l'environnement.

Le processus d'apprentissage vise à doter l'agent d'une politique d'action lui permettant de faire les meilleurs choix. Une **récompense** est alors attribuée à l'agent dont la valeur dépend de si l'action est positive ou négative pour l'agent. Lors de chaque étape de l'apprentissage, l'agent reçoit une observation de l'environnement dans lequel il évolue. Suivant cette observation, l'agent prend une décision d'action. La décision est prise dans un ensemble d'actions appelé espace des actions. Cet espace peut dépendre de l'état.

Un exemple simple est celui d'un jeu d'échec dans lequel l'observation correspond à la position de chacune des pièces de l'échiquier et l'espace des actions est l'ensemble des déplacements possibles des pièces (un fou ne peut pas être déplacé au lancement de partie par exemple). Naturellement, on souhaite que l'agent réalise la meilleure action possible suivant l'observation reçue. L'agent, pour atteindre ce but, applique une politique d'action (notée par la suite π) qu'il utilise pour sa prise de décision. A chaque récompense obtenue, cette politique est mise à jour. Au fil des épisodes d'apprentissage, on espère ainsi atteindre une politique optimale menant à la victoire, quel que soit l'adversaire.

Dans cet article, la voiture doit, à partir de l'observation de l'environnement par son capteur Lidar, agir sur la propulsion et la direction pour parcourir le plus vite possible la piste. L'environnement est constitué de la voiture, de la piste et des voitures adverses et l'agent est le programme de pilotage de la voiture.

Des bibliothèques existent pour l'apprentissage par renforcement profond. Ici est utilisée la bibliothèque Stable-Baselines3 de PyTorch [13], référence dans le domaine de l'IA. Après plusieurs essais, c'est l'algorithme PPO (Proximal Policy Optimization) qui a donné les meilleurs résultats et

est donc retenu pour cet article, associé à Gymnasium [12], ensemble d'outils développés par OpenAI pour l'apprentissage par renforcement, repris depuis par Farama foundation.

2 - Présentation des 4 étapes menant à une conduite autonome

Le travail se présente en quatre étapes, les deux premières s'intéressant à la mise en place de la voiture réelle et de son modèle simulé, la troisième à l'apprentissage automatique sur le simulateur et la dernière au transfert du réseau du simulateur vers la voiture réelle.

2.1 – Étape 1 : fonctions de base sur la voiture réelle

La première étape consiste à équiper la voiture des capteurs et actionneurs nécessaire à la conduite autonome et à développer les fonctions Lidar, direction, propulsion pour la voiture réelle, avec un algorithme de conduite basique. Cette étape est expliquée dans la ressource « *CoVaPSy : Premiers programmes python sur la voiture réelle* » [8].

Les fonctions sont fournies sur le dépôt git [5].

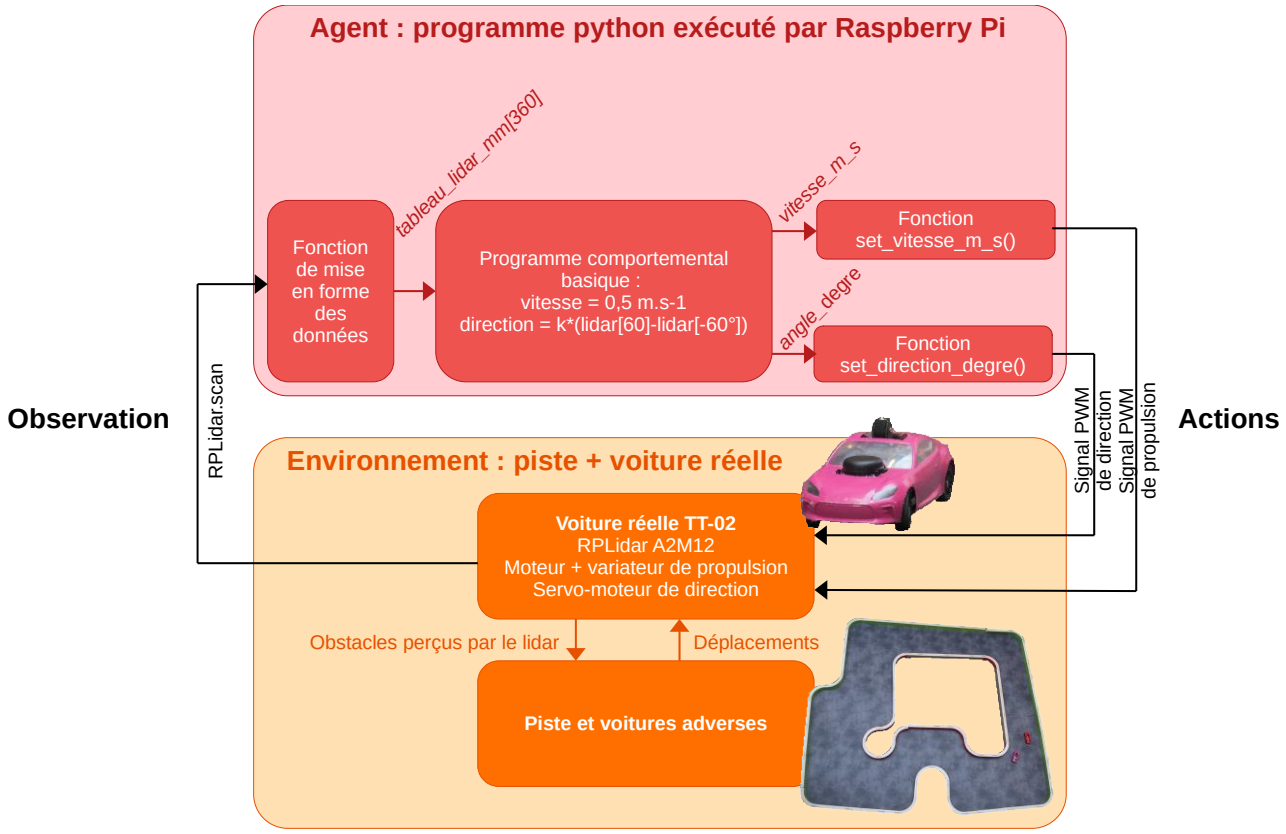

Figure 4 : Fonctions de base de la voiture réelle

2.2 – Étape 2 : fonctions de base sur le simulateur

La deuxième étape consiste à concevoir un modèle simulé de la voiture le plus proche de la voiture réelle et à développer les fonctions pour cette voiture simulée, se comportant comme les précédentes de sorte de fonctionner avec le même algorithme de conduite.

En plus de la présentation du simulateur Webots, cette étape est expliquée dans la ressource « *CoVaPSy : Mise en œuvre du Simulateur Webots* » [9].

Les fonctions et le modèle de voiture TT-02 simulée sont fournies avec le projet de base du simulateur, sur le dépôt git [5].

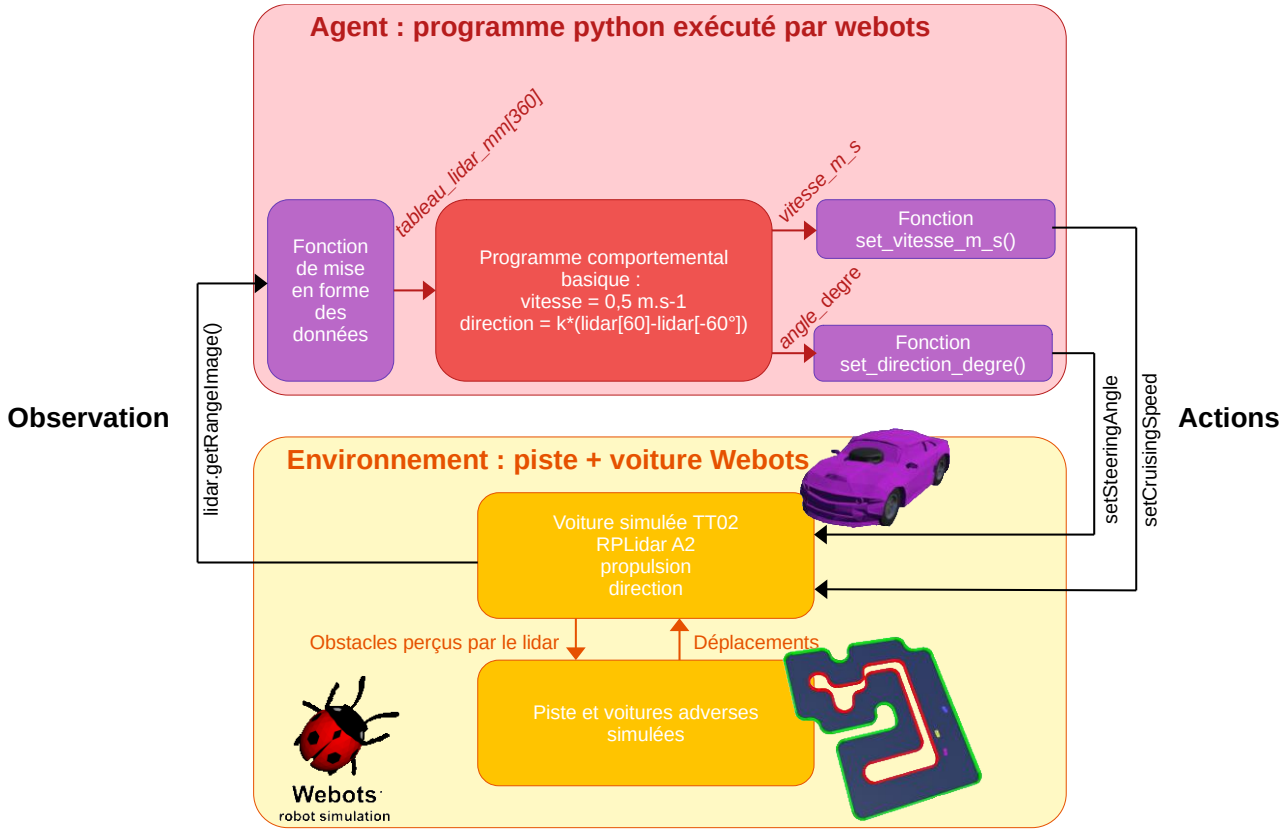

Figure 5 : Fonctions de base du simulateur

2.3 – Étape 3 : Apprentissage automatique de la conduite sur simulateur

La troisième étape remplace l'algorithme basique de conduite par un réseau de neurones et met en place l'apprentissage par renforcement de ce réseau de neurones pour aboutir à une conduite performante. C'est l'objet de la partie 3 de cette ressource.

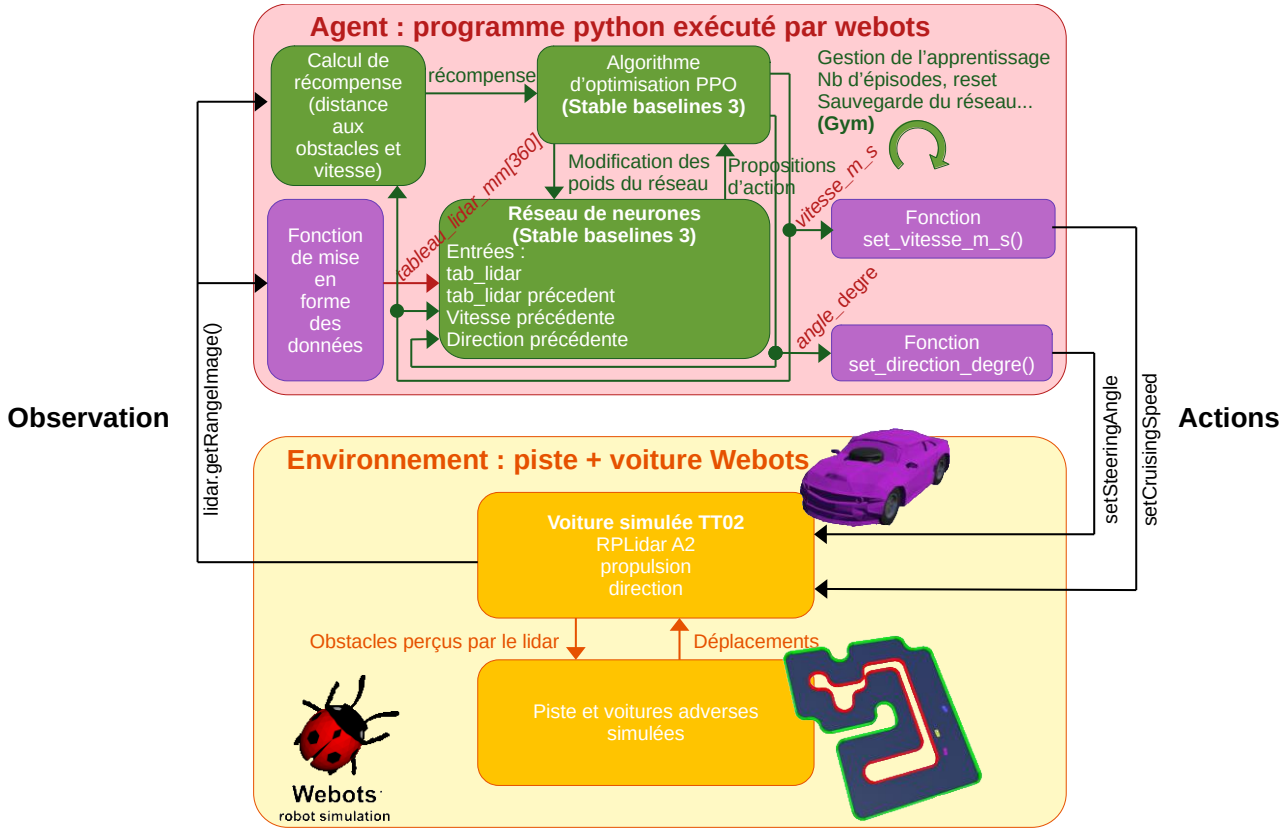

Figure 6 : Fonctions utilisées pour l'apprentissage par renforcement sur le simulateur

2.4 - Étape 4 : transfert du réseau de neurones dans la voiture réelle

Enfin, la dernière étape consiste à transférer le réseau de la voiture simulée dans la voiture réelle pour permettre à la voiture de concourir lors de la course de voitures autonomes. C'est l'objet de la partie 4 de cet article.

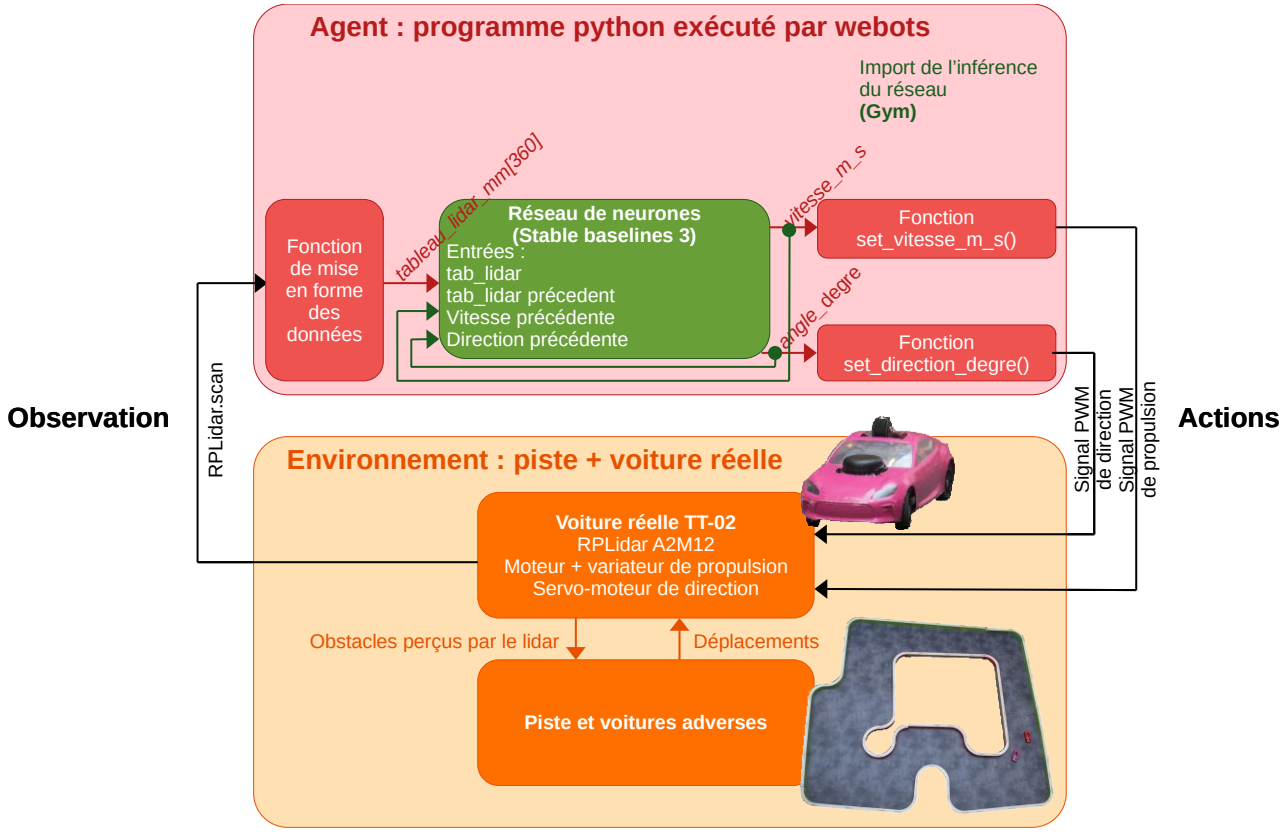

Figure 7 : Utilisation de l'inférence du réseau de neurones issu de l'apprentissage pour la conduite de la voiture réelle

3 - Apprentissage par renforcement sur simulateur

L'utilisation d'un simulateur est indispensable pour réaliser les milliers d'essais d'un apprentissage par renforcement. Pour que le transfert du simulateur à la réalité soit possible, une première exigence est d'avoir un modèle simulé proche de la voiture réelle Tamiya TT-02. La première étape du travail a donc consisté à développer ce modèle et à ajuster ses paramètres à partir des voitures des bibliothèques Webots et de la documentation de la voiture réelle. L'ensemble est décrit dans la ressource « *CoVaPSy : Mise en œuvre du Simulateur Webots* » [9] et le projet de base est disponible en téléchargement sur le dépôt github [5].

Cet article utilise le projet *Simulateur_CoVAPSy_Webots2023b_RL.zip* (en annexe de ce présent article [15]) issu de celui de l'article cité ci-dessus [9], avec en plus le superviseur et des voitures *sparring partners* [5]. Le simulateur utilisé est Webots, de Cyberbotics, dans sa version R2023b. C'est un logiciel open source permettant de créer un « monde » (*world* dans Webots) dans le but de simuler des machines robotiques. Le programme pilotant la voiture est écrit en Python. Il est également possible de programmer en Java, C++ ou Matlab.

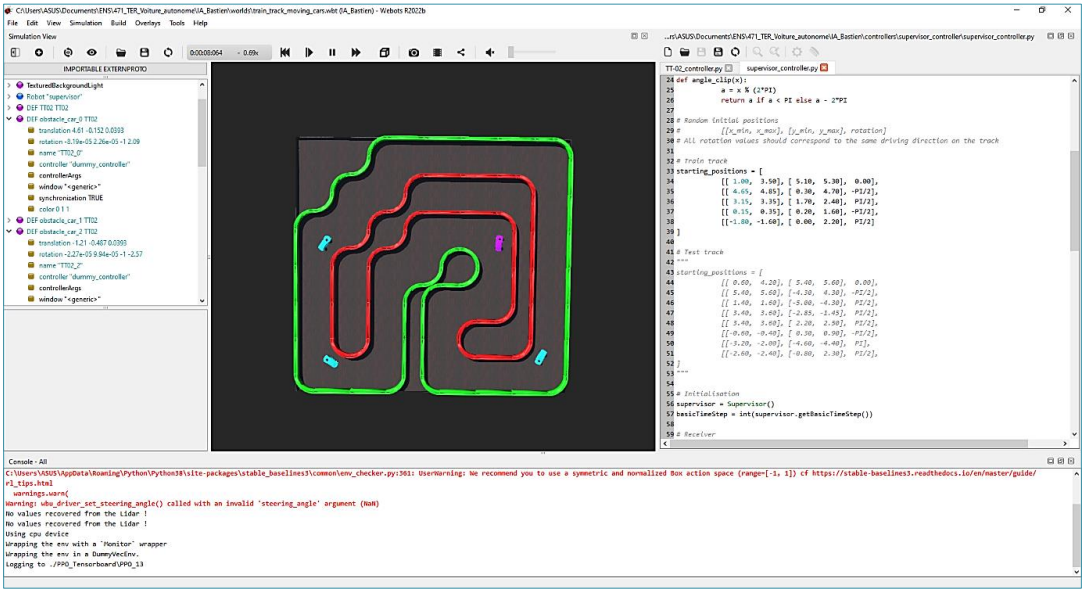

Figure 8 : Interface du simulateur Webots

Le code étant assez complexe, il est recommandé, au moins pour l'agent (le *controller* de la voiture) d'utiliser un logiciel de développement python plus complet que l'éditeur de Webots, comme expliqué dans l'article « CoVaPSy : Mise en œuvre du Simulateur Webots » [9].

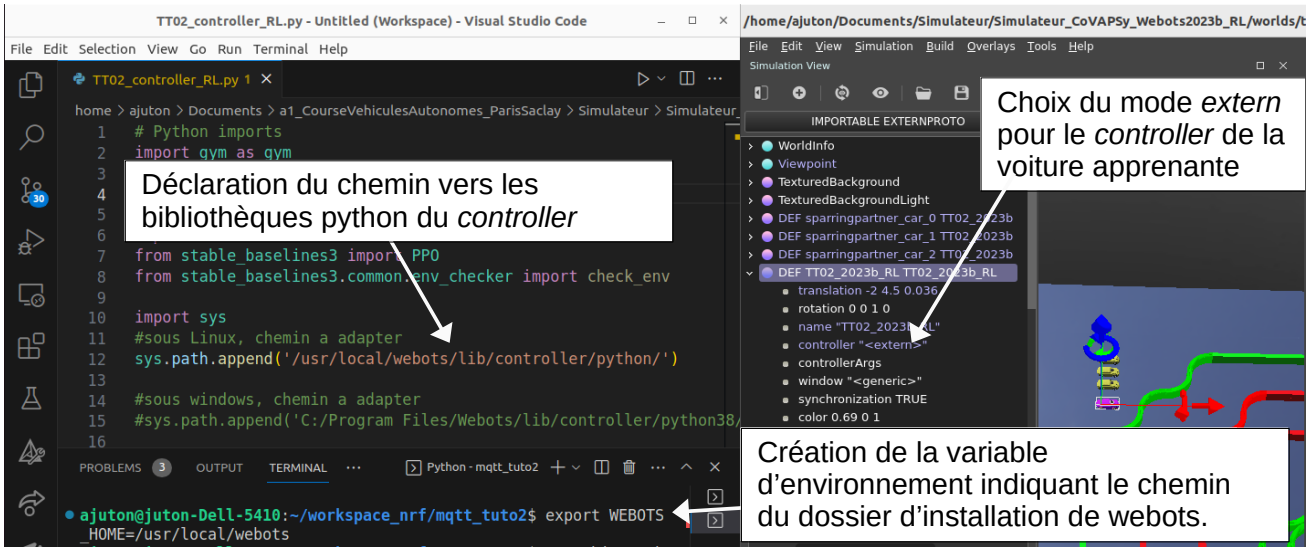

Figure 9 : Configuration de Webots et du programme python pour utiliser un logiciel de développement extérieur à Webots

En plus du modèle de la voiture, pour pouvoir entraîner efficacement celle-ci sur le simulateur, il est nécessaire de disposer de plusieurs éléments qui seront détaillés par la suite :

- Les pistes, à la fois pour l'apprentissage et l'évaluation ;
- Un superviseur capable de replacer les voitures pour les réinitialisations ;
- La bibliothèque Stable-Baselines3 pour la génération du réseau de neurones et son apprentissage avec l'algorithme PPO ;
- La bibliothèque Gymnasium pour gérer le processus d'apprentissage, le lien avec l'environnement Webots, et l'utilisation de l'inférence.

3.1 - Pistes utilisées

Pour un apprentissage par renforcement, il est nécessaire de disposer d'un maximum de situations différentes, représentatives des situations dans lesquelles se trouvera la voiture réelle. Dans le cas

présent, il faut une piste avec de longues lignes droites, des virages dans chaque sens, des virages à 180°, etc. On propose donc la piste suivante comme étant la piste d'entraînement :

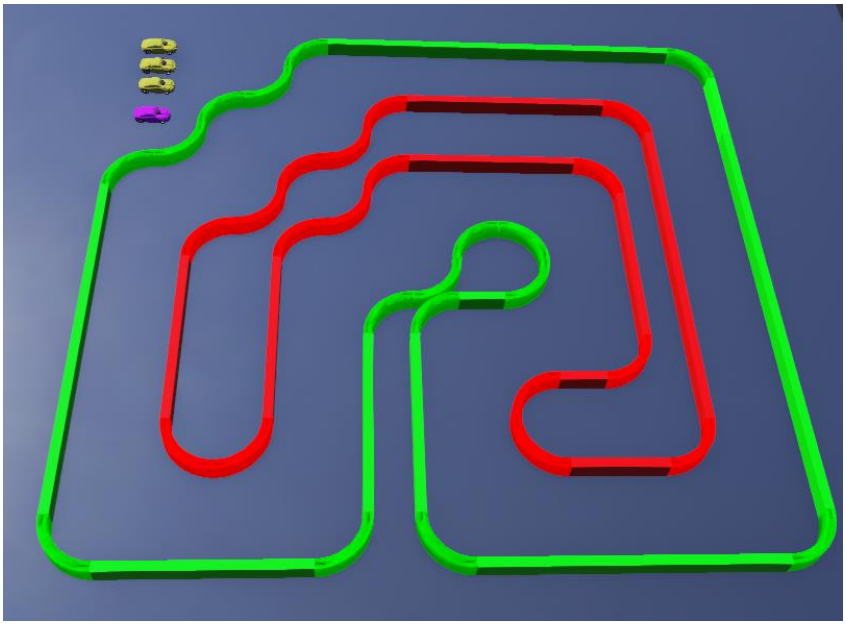

Figure 10 : Piste d'entraînement pour l'apprentissage par renforcement

Sur cette piste, on ajoute trois voitures jaunes ayant pour modèle la TT-02 et ayant un *controller* (programme de conduite) très simple (la vitesse est constante et l'angle de la direction est proportionnel à la distance mesurée à 60° moins celle mesurée à -60°). En démarrant de manière aléatoire sur la piste, on obtient ainsi une piste représentative de la course.

Dans un processus d'apprentissage, il est nécessaire de valider ce qui a été appris sur la piste d'entraînement sur une autre piste, pour vérifier notamment qu'il n'y a pas eu de sur-apprentissage (la voiture apprend à aller très vite sur la piste d'essai mais est incapable de rouler sur une autre piste). Pour cela, est créée une piste de validation avec cinq voitures en plus.

Remarque : la piste de test fournie mériterait d'être un peu élargie car elle présente des zones plus étroites que sur la piste d'entraînement et sur les pistes réelles.

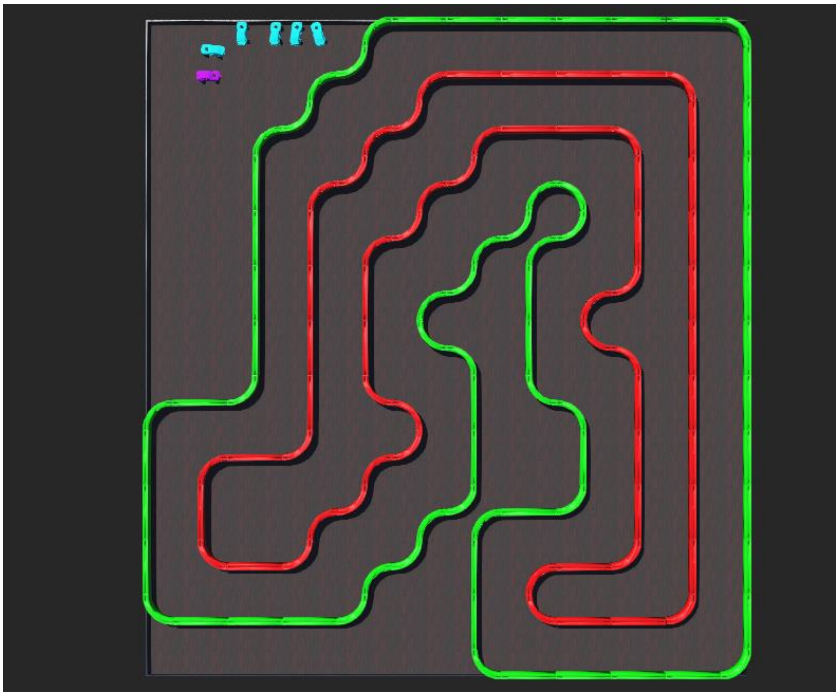

Figure 11 : Piste de validation pour l'apprentissage par renforcement

3.2 - Le superviseur

Sur ces pistes, pour pouvoir lancer l'apprentissage, il est nécessaire de gérer l'ensemble des voitures lors d'une phase de réinitialisation. Pour cela est ajouté un robot « **superviseur** » qui permet de repositionner toutes les voitures.

Le superviseur reçoit, via un récepteur, un message de l'agent de la voiture en apprentissage lors d'une collision pour remettre l'ensemble des voitures en place de manière aléatoire pour un nouvel épisode. Il possède pour cela un tableau d'encadrements de positions dans lesquelles est choisie aléatoirement une valeur pour affecter « intelligemment » une position aléatoire à chaque véhicule. Cela évite que deux voitures ne se touchent au départ, que des voitures ne suivent la piste en sens inverse ou qu'une voiture ne soit face au mur. Le choix de valeurs de positions et de sens de rotation aléatoires évite le sur-apprentissage (la voiture apprend un algorithme efficace uniquement pour un départ dans la position initiale de l'apprentissage).

Une fois la mise en place effectuée, le superviseur, via un émetteur, envoie un message à l'agent indiquant que les voitures sont prêtes pour redémarrer. Les voitures *sparring partners* sont repositionnées par le superviseur mais ne communiquent pas. La voiture apprenant la conduite est un nœud TT02_2023b_RL différent du TT02_2023b par la présence d'un émetteur et d'un récepteur.

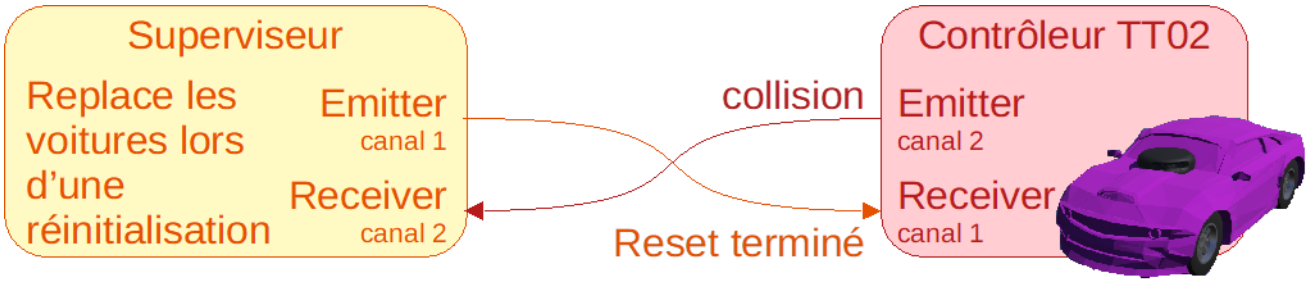

Figure 12 : Messages échangés par les émetteurs/récepteurs du superviseur et de la voiture en apprentissage

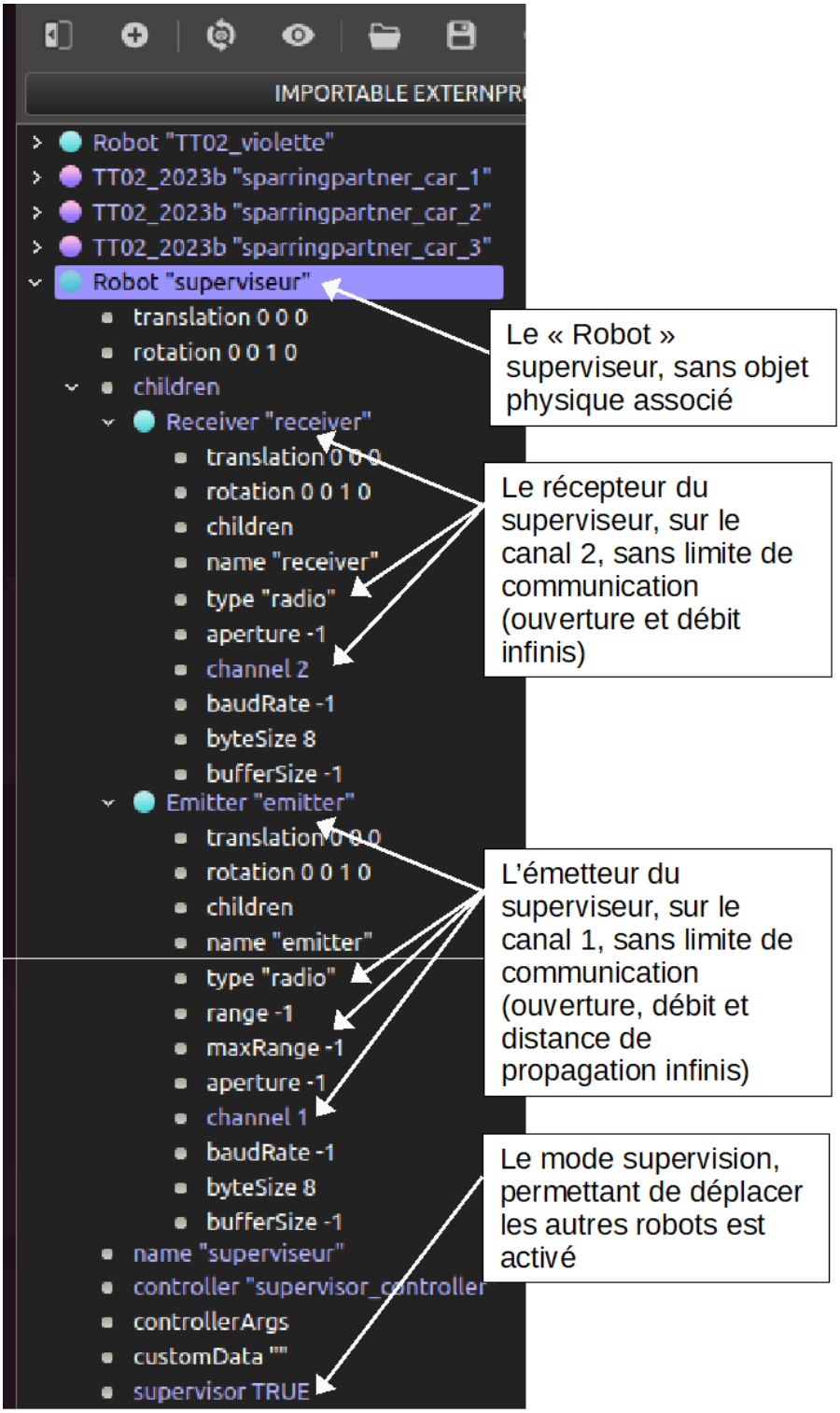

Figure 13 : Le superviseur dans l'arborescence des éléments du projet

On détaille ci-dessous le code du superviseur :

```python
# Python imports
import random

# Webots imports
from controller import Supervisor

# Global constants
PI = 3.141592653589793
RECEIVER_SAMPLING_PERIOD = 64 # In milliseconds

# /!\ Set the number of sparring partner cars
NB_SPARRINGPARTNER_CARS = 3

# Clipping function
def value_clip(x, low, up):
        return low if x < low else up if x > up else x

# Angle normalization function (Webots angle values range between -pi and pi)
def angle_clip(x):
        a = x % (2*PI)
        return a if a < PI else a - 2*PI

# Positions intiales aléatoires
# (plusieurs positions initiales auxquelles s'ajoute un peu d'aléatoire) [[x_min, x_max], [y_min, y_max], rotation]
# Les angles correspondents au même sens de rotation pour les voitures

# Train track
starting_positions = [
        [[ 0.00,  4.00], [ 5.10,  5.30],  0.00],
        [[ 4.95,  5.15], [ 0.30,  4.00], -PI/2],
        [[ 2.70,  4.50], [-1.25, -0.75],  PI  ],
        [[ 3.20,  3.45], [ 2.00,  2.40],  PI/2],
        [[ 1.90,  2.70], [ 3.10,  3.30],  PI  ],
        [[ 0.00,  0.25], [-0.50,  1.50], -PI/2],
        [[-2.20, -1.90], [-0.50,  2.30],  PI/2]
]

"""
Test track
starting_positions = [
        [[ 0.60,  4.20], [ 5.40,  5.60],  0.00],
        [[ 5.40,  5.60], [-4.30,  4.30], -PI/2],
        [[ 1.40,  1.60], [-5.00, -4.30],  PI/2],
        [[ 3.40,  3.60], [-2.85, -1.45],  PI/2],
        [[ 3.40,  3.60], [ 2.20,  2.50],  PI/2],
        [[-0.60, -0.40], [ 0.30,  0.90], -PI/2],
        [[-3.20, -2.00], [-4.60, -4.40],  PI],
        [[-2.60, -2.40], [-0.80,  2.30],  PI/2],
]
"""

# Initialisation
supervisor = Supervisor()
basicTimeStep = int(supervisor.getBasicTimeStep())

# Receiver et emitter
receiver = supervisor.getDevice("receiver")
receiver.enable(RECEIVER_SAMPLING_PERIOD)
emitter = supervisor.getDevice("emitter")
packet_number = 0

# Recuperation des liens vers les noeuds voitures
tt_02 = supervisor.getFromDef("TT02_2023b_RL")
tt_02_translation = tt_02.getField("translation")
tt_02_rotation = tt_02.getField("rotation")
sparringpartner_car_nodes = [supervisor.getFromDef(f"sparringpartner_car_{i}") for i in range(NB_SPARRINGPARTNER_CARS)]
sparringpartner_car_translation_fields = [sparringpartner_car_nodes[i].getField("translation") for i in range(NB_SPARRINGPARTNER_CARS)]
sparringpartner_car_rotation_fields = [sparringpartner_car_nodes[i].getField("rotation") for i in range(NB_SPARRINGPARTNER_CARS)]

erreur_position = 0
```

Positions de départ. Le superviseur choisit pour la voiture apprenante et pour les voitures sparring partner une position dans les plages de valeur proposées avec un angle aléatoire autour de l'angle proposé.

Idem pour le circuit de validation

Configuration des émetteur et récepteur pour la communication avec la voiture apprenante

Récupération des liens vers la voiture apprenante et vers les voitures sparring partner pour contrôler ensuite leur position pour les débuts d'épisodes

Compteur des erreurs de réinitialisation

Figure 14 : Première partie du code python du superviseur : initialisations

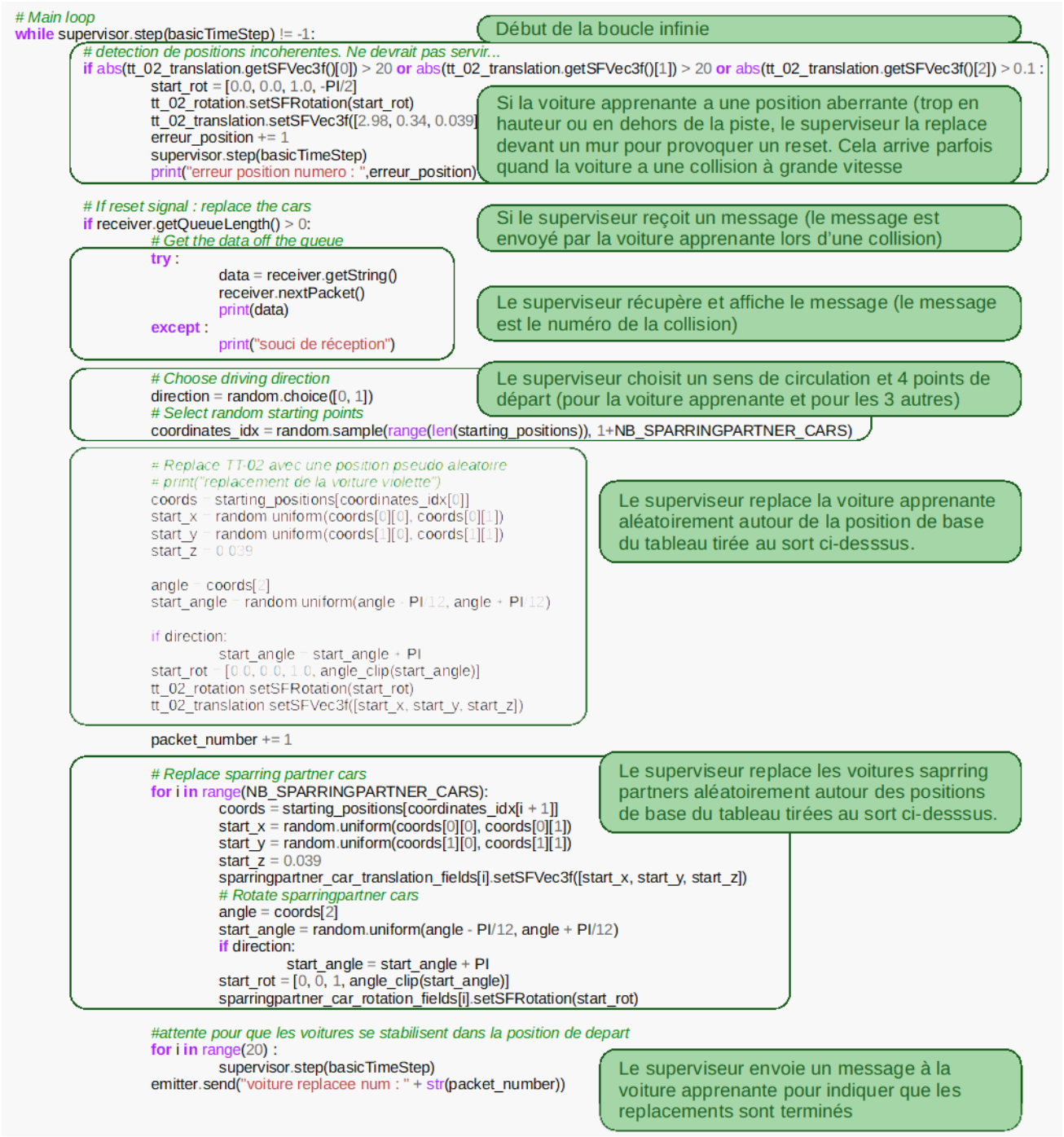

Figure 15 : Seconde partie du code python du superviseur : la boucle principale

3.3 - L'agent (le contrôleur de la voiture) et l'interface Gymnasium

Gymnasium (nommée Gym à sa création) est une bibliothèque Python développée par OpenAI dont l'évolution est, depuis peu, suivie par Farama Foundation, qui permet de gérer l'apprentissage par renforcement de manière normalisée, faisant le lien entre l'environnement (Webots ici mais Gymnasium en propose plusieurs) et l'algorithme d'apprentissage de la politique de l'agent (le contrôleur de conduite de la voiture ici).

Pour plus de détails sur Gymnasium et son utilisation, se référer à la ressource « *Introduction aux bibliothèques Gym et Stable-Baselines pour l'apprentissage par renforcement* » [4] du « *Dossier Intelligence Artificielle* » [2].

Gymnasium (et numpy, demandé par Gymnasium) s'installe à l'aide de la commande suivante :

```
pip3 install numpy
pip3 install gymnasium
```

Dans le cas de la course de voitures autonomes, un premier travail est de faire le lien entre Gymnasium et le « monde » Webots décrit ci-dessus. Gymnasium exige qu'un environnement contienne les fonctions suivantes :

- *get_observation()* : fonction renvoyant les observations de l'environnement.
- *get_reward()* : fonction donnant la récompense selon l'action effectuée par l'agent.
- *reset()* : fonction exécutant la démarche pour repartir au début d'un épisode.
- *step()* : fonction faisant évoluer l'environnement d'un pas.

Toutes ces fonctions sont rassemblées dans une classe nommée ici « WebotsGymEnvironment ». Dans cette classe ont été rajoutées les trois fonctions propres à la voiture, décrites dans l'article sur le simulateur [9] :

- *get_Lidar_mm()* : fonction qui renvoie un tableau des valeurs acquises par le Lidar dans le même format que la voiture réelle avec des valeurs cohérentes en mm.
- *set_vitesse_m_s()* : fonction qui prend en argument une vitesse en $m.s^{-1}$ pour contrôler la propulsion de la voiture. Cette fonction existe aussi sur la voiture réelle.
- *set_direction_degre()* : Fonction qui prend en argument un angle en degré pour contrôler la direction de la voiture. Cette fonction existe aussi sur la voiture réelle.

Tout commence par la construction de l'environnement avec **la fonction *__init__()***

```
class WebotsGymEnvironment(Driver, gym.Env):
        def __init__(self):
                super().__init__()

                #valeur initiale des actions
                self.consigne_angle = 0.0          # en degres
                self.consigne_vitesse = 0.1        # en m/s

                #compteur servant à la supervision de l'apprentissage
                self.numero_crash = 0              # compteur de collisions
                self.nb_pb_lidar = 0               # compteur de problèmes de communication avec le lidar
                self.nb_pb_acqui_lidar=0           # compteur de problèmes d'acquisition lidar
                self.nb_demarrage_lidar=0          # compteur de démarrages de lidar
                self.reset_counter = 0    # compteur de pas d'apprentissage pour arrêter un épisode après n pas
                self.packet_number = 0             # compteur de messages envoyés

                # Emitter / Receiver
                self.emitter = super().getDevice("emitter")
                self.receiver = super().getDevice("receiver")
                self.receiver.enable(RECEIVER_SAMPLING_PERIOD)

                # Lidar initialisation
                self.lidar = super().getDevice("RpLidarA2")
                self.lidar.enable(int(super().getBasicTimeStep()))
                self.lidar.enablePointCloud()

                # Action space
                self.action_space = gym.spaces.Box(low=np.array([-1, -1]), high=np.array([1, 1]), dtype=np.float32)

                # Observation space (description de l'espace d'observation : lidar, lidar_1, vitesse_1 et direction_1)
                self.observation_space = gym.spaces.Dict({
                "current_lidar":gym.spaces.Box(np.zeros(201), np.ones(201), dtype=np.float64),
                "previous_lidar":gym.spaces.Box(np.zeros(201), np.ones(201), dtype=np.float64),
                "previous_speed":gym.spaces.Box(np.array([0]), np.array([1]), dtype=np.float64),
                "previous_angle":gym.spaces.Box(np.array([-1]), np.array([1]), dtype=np.float64),
                })
```

Figure 16 : Code python de la fonction __init__()

Il est nécessaire de décrire les espaces d'action et d'observation dans la classe de l'environnement Gymnasium. Ici, pour l'espace d'action :

- Un scalaire compris entre -1 et 1 pour l'incrément en vitesse
- Un scalaire compris entre -1 et 1 pour l'incrément en angle

Les valeurs sont normalisées car Stable-Baselines3 préfère des valeurs normalisées. Ce n'est que par la suite que l'on multiplie par 0.5 m/s pour la vitesse et 9° pour l'angle.

L'espace d'observation est le suivant :

- Un tableau de 201 scalaires compris entre 0 et 1 pour les données actuelles du Lidar
- Un tableau de 201 scalaires compris entre 0 et 1 pour les données précédentes du Lidar
- Un scalaire compris entre 0 et 1 pour la vitesse actuelle de la voiture (la consigne)
- Un scalaire compris entre -1 et 1 pour la direction actuelle de la voiture (la consigne)

Les valeurs sont normalisées en les divisant par leurs valeurs maximales possibles respectives.

La fonction *get_observation()*

```
# Get Lidar observation
def get_observation(self,init=False):
        tableau_lidar_mm = self.get_lidar_mm()
        i = 0
        while(tableau_lidar_mm[0] == 0) and (i<50) :
                #on essaie d'avoir un tableau de valeur correct !
                self.nb_pb_acqui_lidar+=1
                print("souci d'aquisition lidar"+str(self.nb_pb_acqui_lidar))
                i = i+1
                tableau_lidar_mm = self.get_lidar_mm() #lidar en mm

        if init:
                current_lidar=tableau_lidar_mm.astype("float64")/12000
                previous_lidar=tableau_lidar_mm.astype("float64")/12000
                previous_speed=[0]
                previous_angle=[0]
        else:
                #grandeurs normalisées pour observation
                previous_lidar=self.observation["current_lidar"]
                current_lidar=tableau_lidar_mm.astype("float64")/12000
                previous_speed=[self.consigne_vitesse/VITESSE_MAX_M_S]
                # si on a un capteur de vitesse sur la voiture relle :
                # previous_speed=[(super().getTargetCruisingSpeed()/3.6)/VITESS
                previous_angle=[self.consigne_angle/MAXANGLE_DEGRE]
                # si on a un capteur de vitesse sur la voiture relle :
                # previous_speed=[(super().getSteeringAngle()*180/PI)/MAXANGLE

        observation = {
                        "current_lidar": current_lidar,
                        "previous_lidar":previous_lidar,
                        "previous_speed":previous_speed,
                        "previous_angle":previous_angle,
                }
        # print(observation["current_lidar"])
        self.observation=observation
        return observation
```

Récupération du tableau des données lidar, en mm

Tant que le tableau est incorrect (on ne peut avoir 0 cm devant le lidar, on recommence l'acquisition

Au début d'un nouvel épisode, on met la valeur actuelle de tableau lidar dans la valeur du précédent.

On remplit les 4 champs de l'observation ; valeur actuelle du lidar, valeur précédente et valeurs des commandes de vitesse et de direction. Les valeurs sont normalisées de sorte d'être entre -1 et +1

On conserve observation dans self.observation (car la valeur actuelle du lidar sera la valeur précédente au prochain pas)

Figure 17 : Code python de la fonction get_observation()

La fonction renvoie les valeurs actuelles normalisées du Lidar à « l'instant présent », les valeurs à « l'instant précédent », (récupérées de l'observation précédente), ainsi que les valeurs normalisées des commandes de direction et de vitesse. Si la fonction est appelée depuis la fonction *reset()* (init = True), on donne le même tableau pour les deux parties de l'espace d'observation concernant le Lidar puisque la voiture a été repositionnée.

Ce sont ces données qui seront les entrées du réseau de neurones.

La fonction *get_reward()*

```
# Reward function
def get_reward(self, obs):
        reward = 0
        done = False
        mini = 1
        #recherche de la distance la plus faible mesuree par le lidar entre -40et +40°
        for i in range(-40,40) :
                if (obs["current_lidar"][i] < mini and obs["current_lidar"][i]!=0) :
                        mini = obs["current_lidar"][i]
        # print(mini)
        #si le lidar touche un mur ou si la voiture va trop vite (chute en dehors du sol de la piste)
        if mini < 0.014 or super().getCurrentSpeed() > 30.0 : #0.014 <-> 160 m
                # Crash
                self.numero_crash += 1
                reward = -300
                done = True

        else:
                #Récompense pour une grande distance aux obstacles et une grande vitesse
                reward = 12 * (mini-0.014) + 3 * super().getTargetCruisingSpeed()
                #print("reward : "+str(reward))

        #Reset si la voiture a fait beaucoup de pas
        self.reset_counter += 1
        if self.reset_counter % RESET_STEP == 0:
                print("Reset")
                done = True

        return reward, done
```

On vérifie si il y a collision en regardant la distance la plus faible à l'avant de la voiture (entre -40° et +40°)

Si il y a collision (160 mm correspondant à la longueur du capot devant le lidar) ou si la vitesse de la voiture est aberrante, done = True et la récompense est très mauvaise (-300)

Si il n'y a pas collision, la récompense valorise une grande distance aux obstacles (une valeur de mini importante) et une grande vitesse.

Si la voiture a fait plus de 16384 pas, on demande à démarrer un nouvel épisode (done = True)

Figure 18 : Code python de la fonction get_reward()

La fonction *get_reward()* attribue la récompense associée à l'état dans lequel se trouve la voiture et indique si l'épisode est terminé (variable done). On distingue deux états possibles pour la voiture. Le premier état est celui d'une collision. Dans le cas de la collision, on donne un malus de -300 « points ». Le deuxième état regroupe toutes les situations autres que celle de collision. On donne comme récompense ici une valeur dépendant de la vitesse actuelle de la voiture ainsi que la distance minimale donnée par le tableau de Lidar à l'avant de la voiture. Les fonctions de récompenses seront détaillées plus tard. On indique aussi ici si l'on a terminé l'épisode via la variable done.

Le calcul de la récompense est très important dans l'apprentissage. Il faut donner des récompenses à la fois pour valider de petites avancées (être loin des obstacles) et pour atteindre l'objectif (aller vite). Trop d'importance donnée à la vitesse amène la voiture à aller très vite en ligne droite pour s'écraser au premier virage. Trop peu d'importance donnée à la vitesse amène la voiture à rouler moins vite, au milieu de la piste.

La fonction *reset()*

```
# Reset the simulation
def reset(self):
        # Reset speed and steering angle et attente de l'arrêt de la voiture
        self.consigne_vitesse = 0.0
        self.consigne_angle = 0.0
        self.set_vitesse_m_s(self.consigne_vitesse )
        self.set_direction_degre(self.consigne_angle)
        for i in range(20) :
                super().step()
        self.reset_counter = 0
        print("crash num : " +str(self.numero_crash))

        if(self.numero_crash != 0):
                #attente de l'arrêt de la voiture
                while abs(super().getTargetCruisingSpeed()) >= 0.001 :
                        print("voiture pas encore arrêtée")
                        super().step()

                # Return an observation
                self.packet_number += 1
                # Envoi du signal de reset au Superviseur pour replacer les voitures
                self.emitter.send("voiture crash numero " + str(self.packet_number))
                super().step()
                #attente de la remise en place des voitures
                while(self.receiver.getQueueLength() == 0) :
                        self.set_vitesse_m_s(self.consigne_vitesse )
                        super().step()

                data = self.receiver.getString()
                self.receiver.nextPacket()
                print(data)

                self.consigne_vitesse = 0
                self.consigne_angle = 0
                self.set_vitesse_m_s(self.consigne_vitesse )
                self.set_direction_degre(self.consigne_angle)
                #on fait quelques pas à l'arrêt pour stabiliser la voiture si besoin
                while abs(super().getTargetCruisingSpeed()) >= 0.001 :
                        print("voiture pas arrêtée")
                        self.set_vitesse_m_s(self.consigne_vitesse )
                        super().step()

        return self.get_observation(True)
```

Arrêt de la voiture (avec quelques pas de simulation pour qu'elle s'arrête réellement), remise à zéro des compteurs

Attente de l'arrêt de la voiture (a priori inutile désormais)

Envoi du message au superviseur indiquant qu'il faut lancer un nouvel épisode (et donc repositionner les voitures)

Attente du message du superviseur indiquant que les voitures sont repositionnées (on en profite pour arrêter de nouveau la voiture, ce qui devrait être inutile normalement…)

Affichage du message reçu

Arrêt de la voiture, parfois mise en mouvement par le déplacement par le superviseur

Figure 19 : Code python de la fonction reset()

La fonction *reset()* est lancée dans le cas d'une collision de la voiture ou si le nombre d'actions autorisées par épisode est atteint. La fonction *reset()* démarre un nouvel épisode. Pour cela, elle envoie un message au superviseur et attend qu'il ait fini le repositionnement des voitures. En plus de cela, plusieurs instructions visent à arrêter les voitures. En effet, le déplacement par le superviseur d'une voiture non arrêtée engendre des mouvements incohérents, d'où les multiples instructions d'arrêt de la voiture et d'attente qu'elle soit stabilisée.

La fonction *step()*

```
# Step function
def step(self, action):
        current_speed = self.consigne_vitesse
        current_angle = self.consigne_angle
        self.consigne_vitesse=(current_speed+action[0]*0.05)
        self.consigne_angle=(current_angle+action[1]*9.0)

        # saturations
        if self.consigne_angle > MAXANGLE_DEGRE  :
                self.consigne_angle = MAXANGLE_DEGRE
        elif self.consigne_angle < -MAXANGLE_DEGRE  :
                self.consigne_angle = -MAXANGLE_DEGRE

        if self.consigne_vitesse > VITESSE_MAX_M_S :
                self.consigne_vitesse = VITESSE_MAX_M_S
        if self.consigne_vitesse < 0.1:
                self.consigne_vitesse = 0.1

        self.set_vitesse_m_s(self.consigne_vitesse)
        self.set_direction_degre(self.consigne_angle)
        super().step()

        obs = self.get_observation()
        reward, done = self.get_reward(obs)
        return obs, reward, done, {}
```

Application de l'action[0] sur la vitesse et de l'action[1] sur la direction

Saturations avec les valeurs maximales d'angle de direction et de vitesse

On évite que la voiture ne s'arrête

On envoie à la voiture les nouvelles commandes de vitesse et de direction

On renvoie la nouvelle observation de l'environnement et la nouvelle récompense

Figure 20 : Code python de la fonction step()

La fonction *step()* correspond à un pas dans le processus d'apprentissage. Dans cette fonction on fait avancer la voiture avec les actions issues de la politique d'action de l'agent (compromis entre hasard/exploration et exploitation du réseau de neurones pendant l'apprentissage). Ensuite, on récupère une observation depuis le Lidar et on calcule la récompense avant de retourner toutes les informations obtenues.

Une fois créée l'interface avec l'environnement au format Gymnasium, il est possible d'utiliser une bibliothèque d'apprentissage par renforcement.

3.4 - Bibliothèque Stable-Baselines3

La bibliothèque Stable-Baselines3 propose plusieurs algorithmes d'apprentissage par renforcement, plus ou moins adaptés pour des problèmes différents. Après plusieurs essais, c'est l'algorithme PPO (Proximal Policy Optimization) qui a donné les meilleurs résultats. Il se caractérise par une évolution des paramètres évitant les discontinuités dans les résultats.

Pour installer Stable-Baselines3, on utilise la commande :

```
pip3 install stable-baselines3
```

La bibliothèque Stable-Baselines3 permet de plus de créer le réseau de neurones (*model* =) avant d'utiliser l'algorithme d'apprentissage pour en optimiser les poids (*model.learn()*). Elle propose un large choix de paramètres pour l'apprentissage.

Pour plus de détails sur la bibliothèque Stable-Baselines3 et son utilisation, se référer à la ressource « *Introduction aux bibliothèques Gym et Stable-Baselines pour l'apprentissage par renforcement* » [4] du « *Dossier Intelligence Artificielle* » [2] et au site web officiel de Stable-Baselines3 [13].

Voici un descriptif des quelques paramètres utilisés ainsi que les valeurs choisies dans le programme de référence :

- ***policy*** : Type de politique utilisée. Ici, pour de multiples entrées ('MultiInputPolicy')

- ***env*** : Environnement d'apprentissage Gym
- ***learning_rate*** : Facteur d'apprentissage, il correspond à l'importance donné à un nouveau pas par rapport à ce qui a été acquis avant. Plus il est faible, plus l'apprentissage est lent et stable (5.10−4)
- ***verbose*** : indique si on affiche les résultats (récompense acquises, nombre de pas avant crash…) en cours d'apprentissage
- **device** = 'cpu' : choisit une exécution uniquement sur le cpu ou avec la carte vidéo ('cuda')
- **tensorboard_log** : Emplacement pour enregistrer les données de Tensorboard

Les autres paramètres (décrits sur le site web de Stable-Baselines3) n'ont pas été explorés et laissés à leur valeur par défaut.

```
def main():
        t0 = time.time()                        # variable t0 pour mesurer la durée de l'apprentissage
        env = WebotsGymEnvironment()            # création de l'environnement
        print("environnement créé")
        check_env(env)                          # vérification de l'environnement
        print("vérification de l'environnement")

        # Model definition                      # paramètres de l'environnement pour l'apprentissage
        model = PPO(policy="MultiInputPolicy",
                env=env,
                learning_rate=5e-4,
                verbose=1,
                device='cpu',
                tensorboard_log='./PPO_Tensorboard',)

        ##########################################################################
        # A lancer pour rejouer un réseau déjà entraîné, à commenter pour entraîner un réseau
        ##########################################################################
        # Load learning data
        # print("demonstration")
        # model = PPO.load("PPO_results_20240516")
        ##########################################################################

        ##########################################################################
        # A lancer pour entraîner le réseau, à commenter pour rejouer un réseau déjà entraîné
        ##########################################################################
        print("début de l'apprentissage")
        model.learn(total_timesteps=1000000)            # lancement de l'apprentissage, pour 1 M de pas
        t1 = time.time()                                # variable t1 pour mesurer la durée de l'apprentissage
        print("fin de l'apprentissage après " + str(t1-t0) + "secondes")
        # Save learning data
        model.save("PPO_results_20240517")              # sauvegarde des poids du réseau entraîné
        ##########################################################################

        ###########################################
        # Démonstration du fonctionnement sur 10000 pas
        obs,_ = env.reset()             # RAZ de l'environnement et récupération de l'observation
        print("Demo of the results.")
        c = 0                           # RAZ du cumul de la récompense
        for _ in range(10000):
        # Play the demo
                action, _ = model.predict(obs, deterministic=True) # choix de l'action à partir du modèle (le réseau)
                obs, reward, done, _, _ = env.step(action)      # exécution de l'action et récup. de l'observation
                c += reward                                     # et de la récompense et d'un éventel crash
                if done:                                        # RAZ si crash
                        obs,_ = env.reset()
                print("Exiting.")                               # Affichage de la récompense cumulée
                print(c)                                        # au bout des 10000 pas

if __name__ == '__main__' :
        main()
```

Figure 21 : Création du réseau, lancement de l'apprentissage (model.learn()), sauvegarde du réseau et démonstration du résultat

La bibliothèque permet de lancer l'apprentissage grâce à la fonction *learn()*. Cette fonction prend en paramètre le nombre d'épisodes que va comporter la phase d'apprentissage. Une fois l'apprentissage terminé, il est possible de sauvegarder le réseau de neurones ce qui est particulièrement intéressant pour notre projet dans le but de charger ce réseau dans la voiture réelle. La fonction *load()* permet notamment de charger un réseau de neurones déjà entraîné.

Il est également possible de repartir d'un réseau entraîné en combinant *load* et *learn* :

```
model=PPO.load("nom_du_reseau_entraine")
print("modèle chargé")
model.set_env(env)
print("set_env effectué")
# Training
t0 = time.time()
print("début de l'apprentissage")
model.learn(total_timesteps=12e7,reset_num_timesteps=False)
t1 = time.time()
print("fin de l'apprentissage en t = " + str(t1-t0) + " s")
# Save learning data
model.save("nom_du_reseau_entraine_v2")
```

Stable-Baselines, si l'argument *verbose* est à 1 lors de la création de l'environnement, donne de manière périodique un retour sur l'apprentissage avec notamment la moyenne du nombre de pas des derniers épisodes (ep_len_mean) et la récompense totale moyenne (ep_rew_mean).

```
-----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 345         |
|    ep_rew_mean          | 3.06e+03    |
| time/                   |             |
|    fps                  | 103         |
|    iterations           | 137         |
|    time_elapsed         | 2706        |
|    total_timesteps      | 280576      |
| train/                  |             |
|    approx_kl            | 0.0048211683 |
|    clip_fraction        | 0.0355      |
|    clip_range           | 0.2         |
|    entropy_loss         | -1.8        |
|    explained_variance   | 0.815       |
|    learning_rate        | 0.0005      |
|    loss                 | 6.74e+03    |
|    n_updates            | 1360        |
|    policy_gradient_loss | -0.00103    |
|    std                  | 0.595       |
|    value_loss           | 1.75e+04    |
-----------------------------------------
```

Figure 22: Messages Stable Baselines en cours d'apprentissage

3.5 - Tensorboard

Stable-Baselines3 est compatible avec tensorboard, outil très populaire de monitoring de l'évolution des grandeurs d'apprentissage. Il fait partie de la bibliothèque open source tensorflow développée par Google.

Le dossier de sauvegarde de Tensorboard est donné en paramètre du modèle lors de la création de l'environnement (tensorboard_log =).

Lancé alors au début de l'apprentissage, Tensorboard fait l'acquisition d'un certain nombre de valeurs représentatives de la performance du système (dont la récompense moyenne totale d'un épisode ep_rew_mean) et propose un tableau de représentations graphiques des grandeurs acquises, permettant ainsi de comparer les performances de différents set d'hyperparamètres.

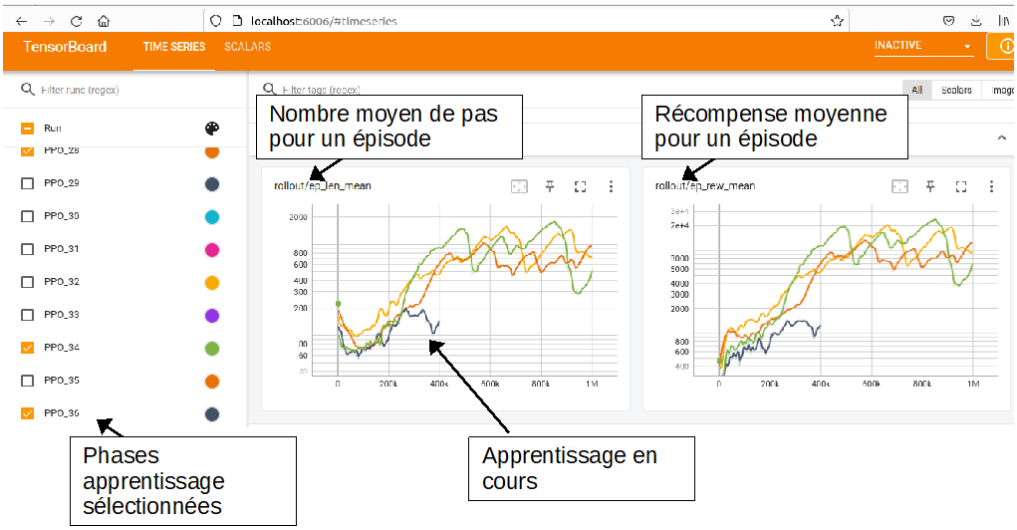

Figure 23 : Tableau d'affichage d'évolution des grandeurs d'apprentissage tensorboard

L'outil est décrit en détail sur sa page web [14]. Il s'installe comme un module python.

```
Pip3 install tensorboard
```

Une fois installé, et après avoir lancé au moins une fois un apprentissage avec tensorboard en paramètre, le tableau tensorboard est créé par la commande tensorboard suivi du dossier où sont stockées les acquisitions (dossier qui est par défaut à l'emplacement du fichier python de l'apprentissage) :

```
tensorboard –logdir PPO_Tensorboard
```

ou

```
python3 -m tensorboard.main –logdir PPO_Tensorboard/
```

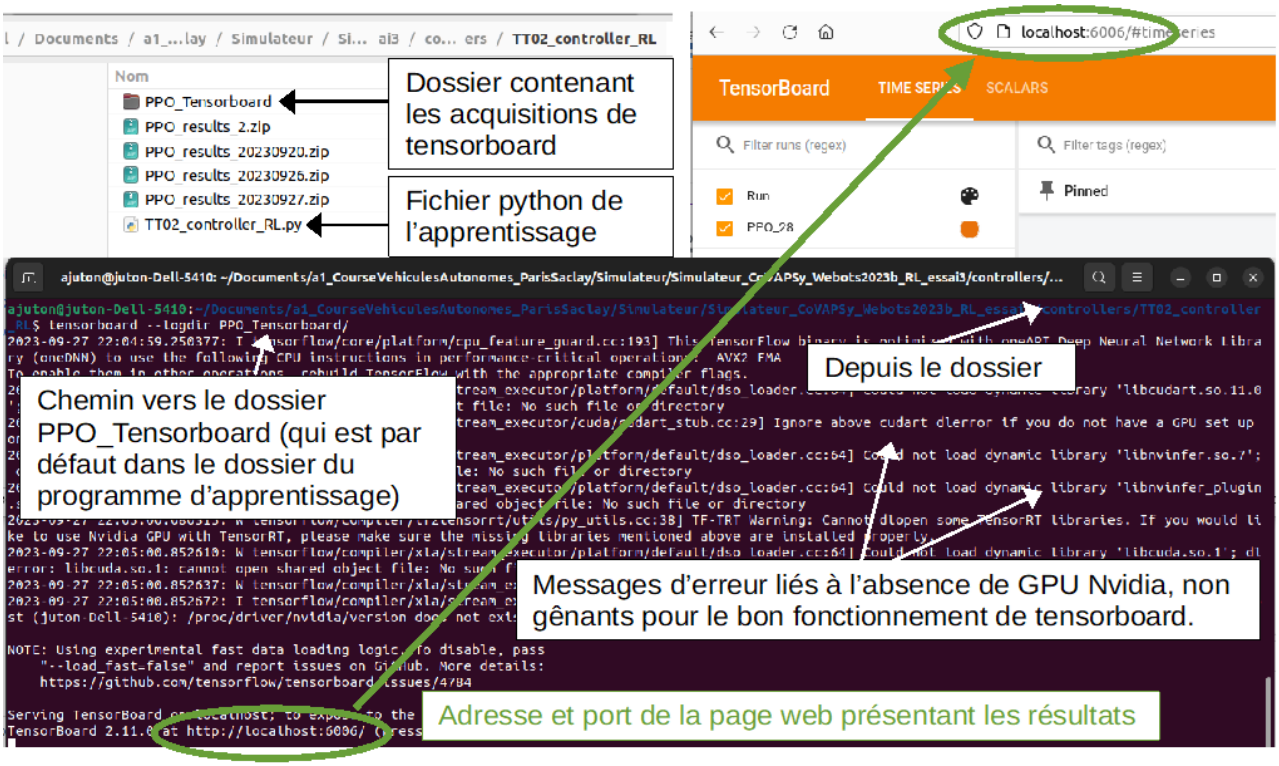

Figure 24 : Exécution de tensorboard

3.6 - Fonctions de récompense

La fonction de récompense est un des hyperparamètres les plus importants dans un processus d'apprentissage par renforcement. Elle est aussi un des plus problématiques. En effet, c'est la récompense qui influe sur comment l'apprentissage s'effectue. Par exemple, si l'on ne sanctionne pas suffisamment le crash de la voiture alors celle-ci pourrait avoir tendance à foncer dans le mur. Autre cas : si on ne favorise pas la vitesse alors la voiture aura tendance à rouler lentement voire être complètement à l'arrêt. Il est donc nécessaire de trouver une bonne fonction de récompense pour avoir le meilleur comportement possible. Dans cet exemple, il a été choisi de donner un malus de -300 lors d'un crash et dans toutes les autres situations la fonction suivante :

$$reward = 12 \times distanceAuMurLePlusProche + 3 \times vitesse$$

```
reward = 12 * (mini-0.014) + 3 * super().getTargetCruisingSpeed()
```

avec *mini*, la distance la plus courte du tableau de Lidar et 0,014, la distance entre le bord de la voiture et le centre du Lidar.

Cette fonction récompense amène la voiture à rouler au centre de la piste. C'est fonctionnel mais pas optimal.

4 - Exemples d'apprentissage

Les principes du simulateur et des bibliothèques nécessaires pour l'apprentissage par renforcement étant expliqués, cette partie revient en détail sur l'entraînement du réseau de neurones.

Dans le cas de la voiture autonome, il est tout d'abord nécessaire de définir un espace d'observation cohérent avec la réalité car toutes les grandeurs ne sont pas mesurables sur la voiture. Ainsi, L'espace d'observation est basé sur le Lidar qui permet de donner la distance des différents obstacles. Le Lidar de simulation est très proche du Lidar réel, ce qui rendra le transfert du réseau de la simulation vers la voiture réelle plus robuste. Ce n'est pas le cas avec une caméra aux images en simulation moins proches des images réelles (luminosité, couleurs, alentours de la piste, …).

Ont été ajoutées à l'espace d'observation les valeurs du Lidar au tour précédent (ce qui permet d'avoir une « dérivée » des valeurs du Lidar), les consignes de vitesse et de direction précédentes. Un asservissement de vitesse est prévu sur la voiture réelle pour avoir une vitesse égale à la consigne quel que soit l'état de la batterie (ce n'est pas le cas sans cet asservissement). Cet asservissement étant lié à la mesure de la vitesse réelle, il permettrait de mettre en entrée du réseau de neurones la vitesse réelle de la voiture et non la consigne, ce qui semble plus intéressant.

Plusieurs algorithmes d'entraînement de réseaux de neurones sont présents dans la Library Python Stable-Baselines3 :

- "Proximal Policy Optimization" (PPO) : Algorithme de type "policy-based"
- "Deep Q-Network" (DQN) : Algorithme de type "valued-based"
- "Soft Actor-Critic" (SAC) : Algorithme de type "actor-critic"

Au regard, de nombreux tests, l'algorithme qui a été retenu pour la suite est l'algorithme PPO, qui converge, entre autres, le plus rapidement.

Remarque : les collisions sont simplifiées par Webots. A chaque collision, il affiche un message Warning. Il est possible de ne pas afficher ces messages en cliquant droit sur la console et en sélectionnant dans Level tous les messages, sauf les warnings : *Error, Info* et *All Controller*

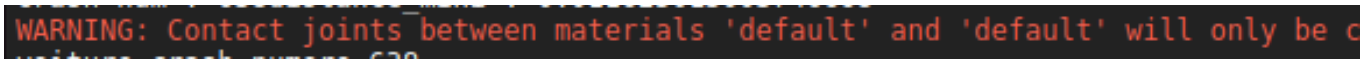

Figure 25: Message de warning obtenu lors d'une collision

4.1 - Principe de l'algorithme PPO :

L'algorithme PPO [10] a pour but d'être facile à implémenter, d'être facile à paramétrer et d'avoir une bonne efficacité au niveau des échantillons. Cet algorithme est un algorithme de type « policy-based » avec lequel on peut utiliser un ensemble d'action discret ou continu.

L'algorithme PPO utilise le principe de « *Trust Region Policy Optimization* ». Ce principe s'assure que la politique π_{old} ne soit pas trop éloignée de la politique actualisée π.

Les références sur le fonctionnement détaillé de PPO sont données dans l'article cité ci-dessus [10] et plus brièvement dans la documentation de Stable-Baselines [13].

4.2 - Modèle de réseau de neurones

Le réseau de neurones utilisé est celui proposé par défaut par Stable-Baselines / PPO, composé de deux sous-couches (*Hidden layers* en anglais). Les neurones d'entrée (*Input layer*) sont au nombre de 404 (201 entrées pour le Lidar actuel, 201 pour le Lidar précédent, 2 pour les consignes de direction et vitesse). Les neurones de sortie sont au nombre de deux correspondant à l'incrément de commande de vitesse et l'incrément de commande de direction.

Dans le but d'améliorer le comportement de la voiture, d'autres données ont été rajoutées dans l'espace d'observation : d'une part les données du Lidar à « l'instant précédent » ont été ajoutées afin de donner une continuité dans l'observation des obstacles. De l'autre, la vitesse et la direction de la voiture avant une nouvelle commande du réseau de neurones ont été ajoutées.

Le schéma ci-dessous illustre le réseau de neurones utilisé. D'autres structures ont été testées (plus de neurones par couche, plus de couches), sans apporter de gain significatif.

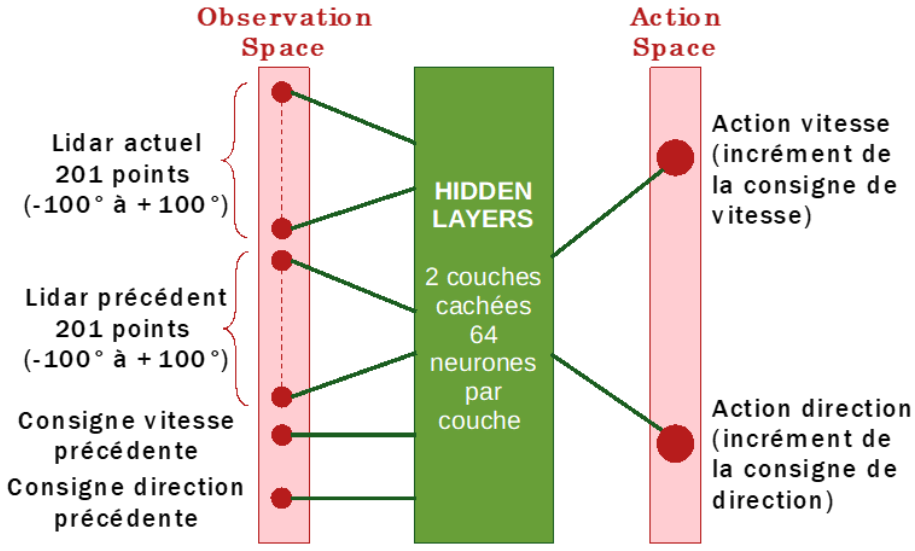

Figure 26 : Modèle du réseau de neurones utilisé

4.3 - Résultats obtenus

L'entraînement dure de 2 à 6 heures selon la puissance de l'ordinateur. Le tableau ci-dessous présente d'une part les résultats obtenus avec le modèle de réseau de neurones sur la piste d'entraînement, de l'autre les résultats obtenus en contre-la-montre sur la piste de validation :

Réseau de neurones sur la piste d'entraînement		
Réseau N°	**Particularité**	**Observation sur la piste d'entraînement**
1	Configuration de base	- Evite les obstacles - Percute parfois les autres voitures - Dépasse les voitures même dans les lignes droites
2	Lors d'un crash : $reward = -300 - n°_{crash}$	- Capable de faire plusieurs tours sans se crasher - Peut éviter les autres voitures et les dépasser - Ralentit beaucoup dans les virages
3	500 000 épisodes pour l'apprentissage	- Pas de différence notable avec les autres réseaux de neurones
4	*gae_lambda=0.90*	- Mouvement un peu hasardeux - Ne tourne pas certaines fois
5	*ent_coef=0.02*	- Fait plusieurs tour sans crash - Roule un peu moins vite que les autres réseaux des neurones
6	*ent_coef=0.02* *angle entre -2° et 2°*	- A réussi à faire un tour complet - Lente dans les virages - Assez réactive dans les lignes droites

Réseaux de neurones sur la piste de validation			
Réseau N°	**Particularité**	**Temps de passage (en minutes)**	**Observation sur la piste de validation (sans obstacles)**
1	Configuration de base	1 :03 :74	- Ne s'est pas crashée – Ralentit dans les virages
2	Lors d'un crash : $reward = -300 - n°_{crash}$	1 :31 :26	- S'est crashée une fois - A réussi mieux dans un sens - Ralentit dans les virages
3	500 000 épisodes pour l'apprentissage	2 :05 :93	- S'est crashée une fois - A réussi mieux dans un sens - Ralentit dans les virages
4	*gae_lambda=0.90*	Aucun	- S'est crashée plusieurs fois - N'arrive pas à gérer les virages après les lignes droites - Voiture la plus rapide mais incapable de faire un tour de piste
5	*ent_coef=0.02*	2 :25 :27	- Ralentit beaucoup dans les lignes droites - Réagit bien dans les virages
6	*ent_coef=0.02* *angle entre -2° et 2°*	2 :17 :49	- N'avance pas très vite - Oscille moins dans les lignes droites

Au vu de ces résultats, on peut en conclure que le réseau de neurone fourni est celui qui donne les résultats les plus satisfaisants. En revanche, lors du passage à la réalité, c'est le réseau n°5 qui a tout de même été choisi puisqu'il a un meilleur comportement sur le simulateur. Dans la réalité, il est préférable de choisir un réseau de neurones le plus stable possible (où la voiture ne se crashe pas) au détriment d'une vitesse moins importante.

5 - Passage à la réalité

Il s'agit ici de la dernière étape du projet, le transfert du réseau de neurones entraîné du simulateur à la voiture réelle.

5.1 - Implémentation dans la voiture réelle

Comme indiqué dans la partie 2, ont été développées deux fonctions similaires à celles présentes dans le simulateur : *set_vitesse_m_s()* et *set_direction_degre()*. Elles sont expliquées dans la ressource [8].

Dans le code, on remarque qu'on s'assure de ne pas laisser de valeur à 0 dans la partie « utile » du Lidar (les valeurs des indices 0 à 99 et celles de 260 à 359). Pour cela, si une valeur 0 (valeur que ne peut donner le Lidar) apparaît entre deux valeurs non nulles, on la considère comme étant la moyenne de ces deux dernières. On se retrouve ainsi dans les mêmes dispositions que la simulation au niveau du Lidar.

Le réseau de neurones, il a pu être chargé grâce la fonction *load()* de Stable-Baselines3. Il est possible d'exploiter ce réseau grâce à la fonction *predict()*, déjà utilisée dans la partie démo en fin d'apprentissage sur le simulateur. Elle prend en argument l'observation et renvoie les actions. Les observations sont stockées sous forme de dictionnaire.

Un objet a été créé pour le châssis, avec les fonctions de déplacement de la voiture :

```python
from rplidar import RPLidar
import numpy as np
import time
from rpi_hardware_pwm import HardwarePWM
import threading
from stable_baselines3 import PPO

################################################
class Chassis():

    def __init__(self):
        #Parametres a ajuster sur chaque voiture
        self.pwm_stop_prop = 7.42
        #...

        #Configuration des PWM
        self.pwm_prop = HardwarePWM(pwm_channel=0, hz=50)
        self.pwm_dir = HardwarePWM(pwm_channel=1, hz=50)

        self.vitesse_consigne = 0
        self.direction_consigne =0

    def demarrage_voiture(self):
        print("demarrage voiture")
        #Démarrage des PWM
        self.pwm_prop.start(self.pwm_stop_prop)
        self.pwm_dir.start(self.angle_pwm_centre)

    def get_direction(self,obs=False):
        if obs:
            return self.direction_consigne/self.angle_degre_max
        else:
            return self.direction_consigne

    def get_vitesse(self,obs=False):
        if obs:
            return self.vitesse_consigne/self.vitesse_max_m_s_hard
        else:
            return self.vitesse_consigne

    def set_direction_degre(self,angle_degre,adresse=1) :
        #voir article Premiers programmes python sur la voiture réelle [8]

    def set_vitesse_m_s(self,vitesse_m_s):
        #voir article Premiers programmes python sur la voiture réelle [8]

    def recule(self):
        #voir article Premiers programmes python sur la voiture réelle [8]

    def arret_voiture(self):
        self.pwm_prop.stop()
        self.pwm_dir.stop()
        print("PWM arrêtées")
```

Import du module stable_baselines3 et du module threading

Classe chassis : ensemble de fonctions liées au déplacement de la voiture. Voir [8] pour les détails

Figure 27 : Import des modules et objet Chassis()

Pour récupérer en continu les données du Lidar, pour éviter que le buffer ne déborde, on utilise une tâche *thread_scan_Lidar* chargée de récupérer les données Lidar et une tâche *thread_conduite_autonome* chargée d'exploiter les données reçues pour conduire la voiture avec le réseau de neurones.

Pour utiliser le Lidar, un objet Lidar_TT02() a été créé permettant de l'initialiser, le démarrer ainsi qu'acquérir les valeurs. Au démarrage du programme, on lance un thread *thread_scan_Lidar* qui permet de faire l'acquisition des données du Lidar en continue. On active et désactive un drapeau dans le code pour indiquer lorsque l'on veut récupérer ces données acquises. Le traitement de ces données comme décrit plus haut se fait en dehors de ce *thread*.

```
class Lidar_TT02():

    def __init__(self):
        #connexion et démarrage du lidar
        self.lidar = RPLidar("/dev/ttyUSB0",baudrate=256000)
        self.tableau_lidar_mm=np.zeros(360)
        self.acqui_lidar=np.zeros(360)
        self.drapeau_nouveau_scan = False
        self.scan_avant_en_cours = False
        self.Run_lidar = False

    def demarrage_lidar(self):
        self.lidar.connect()
        print (self.lidar.get_info())
        self.lidar.start_motor()
        time.sleep(2)

    def lidar_scan(self):
        while(self.Run_lidar == True):
            try:
                for _,_,angle_lidar,distance in self.lidar.iter_measures(scan_type='express'):
                    angle = min(359,max(0,359-int(angle_lidar)))

                    if(angle >= 260) or (angle <= 100):
                        self.acqui_lidar[angle] = distance
                    if(angle<260) and (angle>110) and (self.scan_avant_en_cours == True):
                        self.drapeau_nouveau_scan = True
                        self.scan_avant_en_cours = False
                    if(angle >= 260) or (angle <= 100):
                        self.scan_avant_en_cours = True
                    if(self.Run_lidar == False):
                        break;
            except:
                print("souci acquisition Lidar")

    def get_values(self):
            for i in range (-100,101):
                self.tableau_lidar_mm[i] = self.acqui_lidar[i]
            self.acqui_lidar = np.zeros(360)

            for i in range(-98,99):
                if self.tableau_lidar_mm[i] == 0:
                    if self.tableau_lidar_mm[i-1] != 0 and self.tableau_lidar_mm[i+1] != 0:
                        self.tableau_lidar_mm[i] = (self.tableau_lidar_mm[i]+self.tableau_lidar_mm[i])/2
            self.drapeau_nouveau_scan =  False
            return self.tableau_lidar_mm

    def get_drapeau(self):
        return self.drapeau_nouveau_scan

    def get_run(self):
        return self.Run_lidar

    def set_run(self, valeur):
        self.Run_lidar = valeur

    def arret_lidar(self):
        self.lidar.stop_motor()
        self.lidar.stop()
        time.sleep(1)
        self.lidar.disconnect()
```

Classe lidar : ensemble de fonctions liées à l'acquisition des valeurs mesurées par le lidar. Voir [8] pour les détails

Initialisation de la connexion et des variables. Démarrage du lidar, vérification de la connexion par get_info()

Fonction lancée ensuite dans un thread, en charge des acquisitions du lidar. A chaque passage de l'angle 100, un drapeau est levé pour indiquer que l'acquisition est terminée.

Fonction lancée par la tâche conduite pour récupérer les valeurs du lidar, en enlevant les éventuels points « erreur » de valeur 0.

Figure 28 : Objet Lidar_TT02()

En parallèle de ce thread (ou tâche), on lance un second thread *thread_conduite_autonome* attaché à la fonction *conduite()* qui est le programme de pilotage de la voiture. Dans la fonction *conduite()*, on commence par récupérer les données du Lidar puis on procède au traitement de ces données. On récupère les valeurs traitées tous les 20 degrés pour faciliter l'analyse des obstacles. Dans le cas où l'on considère qu'il y a collision (valeur minimale inférieure à un certain seuil), on recule dans la direction opposée à l'obstacle détectée (indice de cette valeur minimale). Si ce n'est pas le cas, on laisse la main au réseau de neurones pour le pilotage.

```
def conduite():
        global lidar
        global voiture
        print("début conduite")
        tableau_lidar_mm = np.zeros(360)
        previous_lidar = np.zeros(360)
        while lidar.get_run() == True:
            if lidar.get_drapeau() == False:
                time.sleep(0.01)
            else:
                tableau_lidar_mm = lidar.get_values()

                min_secteur = [0]*10
                for index_secteur in range(0,10) :
                    angle_secteur = -90 + index_secteur*20
                    min_secteur[index_secteur] = 12000
                    for angle_lidar in range(angle_secteur-10,angle_secteur+10) :
                        if tableau_lidar_mm[angle_lidar] < min_secteur[index_secteur] and
tableau_lidar_mm[angle_lidar] != 0 :
                            min_secteur[index_secteur] = tableau_lidar_mm[angle_lidar]
                    if min_secteur[index_secteur] == 12000 : #aucune valeur correcte
                        min_secteur[index_secteur] = 0

                if (min_secteur[4]<=160 and min_secteur[4] !=0)\
                        or (min_secteur[3]<=160 and min_secteur[3] !=0)\
                            or (min_secteur[2]<=160 and min_secteur[2] !=0) :
                    angle_degre = -18
                    voiture.set_direction_degre(angle_degre)
                    print("mur à droite")
                    voiture.recule()

                elif (min_secteur[5]<=160 and min_secteur[5] !=0)\
                        or (min_secteur[6]<=160 and min_secteur[6] !=0)\
                            or (min_secteur[7]<=160 and min_secteur[7] !=0) :
                    angle_degre = +18
                    voiture.set_direction_degre(angle_degre)
                    print("mur à gauche")
                    voiture.recule()

                else :
                    current_lidar=tableau_lidar_mm.astype("float64")/12000
                    previous_speed=np.array([float(voiture.get_vitesse(obs=True))])
                    previous_angle=np.array([float(voiture.get_direction(obs=True))])
                    if previous_angle[0] > 1 :
                        previous_angle[0] = 1
                    elif previous_angle[0] < -1 :
                        previous_angle[0] = -1

                    obs={"current_lidar":current_lidar,
                         "previous_lidar":previous_lidar,
                         "previous_speed":previous_speed,
                         "previous_angle":previous_angle,
                         }

                    action,_= modele.predict(obs,deterministic=True)
                    angle=float(voiture.get_direction(obs=True))+action[1]*18.0

                    voiture.set_direction_degre(angle)
                    nouvelle_vitesse = float(voiture.get_vitesse())+action[0]
                    if (nouvelle_vitesse <0.1) :
                        vitesse_m_s = 0.1
                    else :
                        vitesse_m_s = nouvelle_vitesse
                    voiture.set_vitesse_m_s(vitesse_m_s)
```

Fonction conduite, en charge d'appliquer la politique de conduite. Elle sera attachée à une tâche conduite.

Tant que le drapeau indiquant l'arrivée de nouvelles données n'est pas levé, la tâche est en sommeil.

Quand le drapeau est levé, on récupère les données et la fonction get_value() abaisse le drapeau.

On cherche par secteur de 20°, la valeur minimale, pour voir si la voiture est bloquée dans un mur.

Si la voiture mesure moins de 160 mm dans le secteur 3 ou 4, elle est bloquée dans un mur à droite => elle recule en braquant vers la droite.

Si la voiture mesure moins de 160 mm dans le secteur 5 ou 6, elle est bloquée dans un mur à gauche => elle recule en braquant à gauche.

Si la voiture n'est pas bloquée dans un mur :
→ on calcule l'observation et
→ on choisit l'action à l'aide du réseau de neurones entraîné sur le simulateur
→ on applique les actions

Figure 29 : La fonction conduite() qui sera attachée à la tâche conduite_autonome()

On remarque que pour éviter une situation bloquée dans certains cas, la voiture ne peut rester immobile.

Le programme principal se contente alors de créer des instances des objets, de charger le réseau de neurones et de créer et lancer les tâches *thread_scan_Lidar()* et *thread_conduite_autonome()* présentées ci-dessus :

```
lidar= Lidar_TT02()
voiture = Chassis()

modele= PPO.load("chemin_vers_le_reseau_entraine.zip")
```

Dans le programme principal, on crée une instance lidar et une instance voiture. On charge le réseau entraîné sur le simulateur

```
while True :
    x = input("Appuyer sur 'c' pour commencer: ")
    while True:
        if x=="c":
            #connexion et démarrage du lidar
            lidar.demarrage_lidar()
            lidar.set_run(True)
            print("lidar démarré")

            voiture.demarrage_voiture()
            print("voiture démarée")

            # Création du thread lidar
            thread_scan_lidar = threading.Thread(target=lidar.lidar_scan)
            thread_scan_lidar.start()
            time.sleep(1)

            x = input("Appuyer sur une touche 'g' pour commencer la course: ")
            if x=='g':
                # Création du thread conduite
                thread_conduite_autonome = threading.Thread(target = conduite)
                thread_conduite_autonome.start()

            while True :
                try :
                    x=input("Appuyer sur 'a' pour arrêter la voiture\n")
                    time.sleep(1)
                    if x=='a' :
                        lidar.set_run(False)
                        voiture.set_vitesse_m_s(0)
                        break
                except KeyboardInterrupt: #récupération du CTRL+C
                    print("arrêt du programme")
                    lidar.set_run(False)
                    voiture.set_vitesse_m_s(0)
                    break

            #Destruction des threads
            thread_scan_lidar.join()
            thread_conduite_autonome.join()
            #arrêt et déconnexion du lidar
            lidar.arret_lidar()
            voiture.arret_voiture()
```

Démarrage du lidar et de la voiture, création et lancement de la tâche thread_scan_lidar attachée à la fonction lidar_scan()

Création et lancement de la tâche thread_conduite_autonome attachée à la fonction conduite()

Attente d'un signal pour arrêter la voiture, détruire les tâches et arrêter le lidar.

Figure 30 : Programme principal de la conduite autonome sur voiture réelle

5.2 - Résultats obtenus

Les résultats présentés ici sont ceux de la voiture réelle avec le réseau de neurones avec *ent_coef = 0.02*. En effet, il s'agit de celui qui a montré le meilleur comportement en réel.

Les résultats sont ceux réalisés le jour de la course de voitures autonomes 2023. Une première phase concerne une piste sans obstacle où le temps comptabilisé est le temps que prend la voiture pour réaliser deux tours.

Figure 31 : 1ère piste de qualification

Les résultats obtenus sont les suivants :

	1er passage	2ème passage
Temps de passage (2 tours de piste)	31 sec	25 sec

On a pu observer que sans aucun obstacle, la voiture réussit très bien à se diriger et ne prend aucun mur pendant son tour. L'apprentissage est donc concluant sur ce point puisque la conduite est très satisfaisante et conforme à ce qui est attendu. En revanche pour atteindre ce résultat, une réduction du coefficient la consigne de vitesse s'imposait. En effet, sans cette correction, la voiture allait trop vite et se prenait quelques fois les murs dans les tests effectués.

Le problème de cette correction est que sans asservissement de vitesse, celle-ci dépend de l'état de charge de la batterie. In fine, la voiture réagit bien mais est assez lente en comparaison avec les autres voitures. Aussi, on peut observer que dans les lignes droites, la voiture a tendance à vaciller contrairement à la simulation.

La deuxième phase se fait avec une nouvelle piste et des obstacles fixes, comme présenté ci-dessous.

Figure 32 : 2ème piste de qualification

Les résultats sont les suivants :

	1er passage	2ème passage
Temps de passage (2 tours de piste)	32 sec	34 sec

En présence d'obstacles fixes, la voiture se comporte un peu moins bien, l'entraînement ayant eu lieu avec quelques voitures sparring partner que la voiture agent pouvait pousser un peu avant qu'une collision ne soit détectée. Sur la piste réelle, la voiture a tendance à ne pas suffisamment tourner pour éviter l'obstacle. En revanche, c'est une bonne situation pour tester la marche arrière couplée au réseau de neurones. La voiture a pu finir ses deux tours malgré des collisions avec les obstacles. On peut conclure que la phase d'entraînement sur simulateur n'a pas suffisamment d'obstacle fixe.

Enfin, la dernière partie de l'évènement était la course en elle-même. Là encore, il y a eu deux courses. Ici pas de chrono à présenter mais quelques observations sont possibles. Lors de la première course, la voiture n'a pas pu finir la course à la suite d'un carambolage et une conduite à contre-sens. Durant la deuxième course cependant, la voiture a pu terminer la course en 2e position. Elle termine 3ème de la compétition. La vidéo « *Apprentissage par renforcement pour la conduite de voiture autonome* » [17], met en valeur le transfert de la simulation vers la réalité.

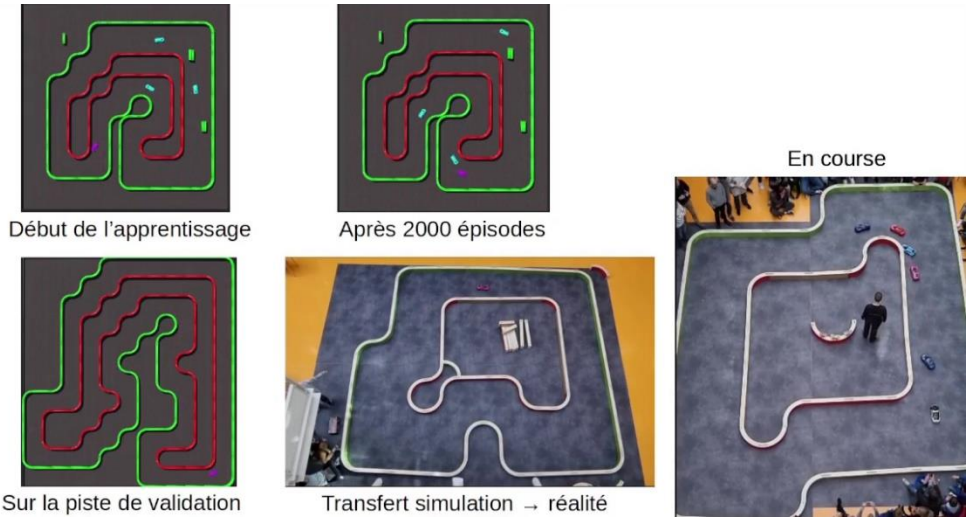

Figure 33 : Extrait de la vidéo « Apprentissage par renforcement pour la conduite de voiture autonome » [17]

On retrouve la voiture rose conduite par réseau de neurones lors de la course 2023 que l'on peut visionner sur le lien suivant : « *Course de voitures autonomes 2023* ».

5.3 - Pistes de travail

La course 2023 a mis en évidence quelques défauts, devenus pistes de travail :

- Un asservissement de vitesse via un microcontrôleur permet d'avoir une vitesse indépendante de la charge de la voiture et conforme à la consigne, améliorant la reproductibilité. On peut de plus ajouter la vitesse réelle en entrée du réseau de neurones, sur le simulateur comme sur la voiture réelle.
- Un servomoteur numérique (Dynamixel AX-12 ou HerkuleX DRS101) permet d'améliorer la dynamique de la direction. Il serait également possible de lire la position réelle du moteur, pour l'utiliser en entrée du réseau de neurones.
- La prise en compte de la dynamique (voire même aussi de la non-linéarité) de la direction réelle dans le simulateur. La direction dans le simulateur est plus dynamique que sur la voiture réelle, ce qui semble être à la source des ondulations dans les lignes droites.
- Un entraînement avec une piste (ou plusieurs pistes) plus réaliste : plus de variations sur la largeur de piste, plus d'obstacles.

- L'amélioration de la simulation par un ré-entraînement du réseau, sans chercher à rester éloigné des murs mais en ne regardant que les temps de passage.
- L'ajout d'une caméra permet d'éviter de repartir en contre-sens suite à un accident.
- La mise en œuvre de méthodes d'amélioration du passage de la simulation à la réalité [29].

En 2024, plusieurs voitures ont repris et fiabilisé ce travail, avec asservissement de vitesse pour les uns et servo-moteur numérique pour les autres. 2 voitures de l'ENS Paris Saclay ont remplacé le Lidar par une caméra pour travailler uniquement avec de la vision. Ce sont les voitures UFR Sciences, ENS Paris Saclay et Institut Villebon Georges Charpak (IVGC) de la course 2024 que l'on peut visionner sur le lien suivant : « *Course de Voitures Autonomes Paris-Saclay 2024* ».

6 - Conclusion

Nombreux sont les exemples d'apprentissage par renforcement en simulation. Beaucoup moins nombreux sont ceux qui vont jusqu'à mettre en œuvre l'inférence du réseau de neurones entraîné dans le monde réel.

Cette ressource présente, sur du matériel accessible (logiciels open-source et voiture entre 800 et 1500 euros) une application pour la mise en œuvre de l'apprentissage par renforcement en simulation et le transfert simulation à réalité.

C'est une base de travail solide, avec de nombreux apports possibles par les enseignants et étudiants qui souhaiteront travailler sur ce sujet et participer à son amélioration.

Références :

[1]: Introduction à l'apprentissage par renforcement, A. Juton, V. Noël, R. Lali, juillet 2022, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/introduction-a-lapprentissage-par-renforcement

[2]: Dossier Intelligence Artificielle, 2022, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/dossier-intelligence-artificielle

[3]: Apprentissage par renforcement de la conduite d'un véhicule sur AirSim, L. de Mattteis, S. Radosalvjevic, juillet 2022, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/apprentissage-par-renforcement-dela-conduite-dun-vehicule-sur-airsim

[4]: Introduction aux bibliothèques Gym et Stable-Baselines pour l'apprentissage par renforcement, G. Chérot, A. Godinot, juillet 2022, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/introduction-aux-bibliotheques-gym-et-stablebaselines-pour-lapprentissage-par-renforcement

[5]: Dépôt GitHub public sur les voitures autonomes https://github.com/ajuton-ens/CourseVoituresAutonomesSaclay

[6]: Site web de la course de voitures autonomes de Paris Saclay : https://ajuton-ens.github.io/CourseVoituresAutonomesSaclay/

[7]: Course Voitures Autonomes Paris Saclay (CoVAPSy) : Travaux pratiques autour des voitures autonomes, T. Boulanger, E. Délègue, K. Hoarau, A. Juton, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/covapsy-tp-autour-des-voitures-autonomes

[8]: CoVaPSy : Premiers programmes python sur la voiture réelle, T. Boulanger, E. Délègue, K. Hoarau, A. Juton, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/covapsy-premiers-programmes-python-sur-voiture-reelle

[9]: CoVaPSy : Mise en œuvre du Simulateur Webots, T. Boulanger, E. Délègue, K. Hoarau, A. Juton, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/covapsy-mise-en-oeuvre-du-simulateur-Webots

[10]: John Schlman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov. "Proximal Policy Optimization Algorithms". In : (2017), https://arxiv.org/pdf/1707.06347.pdf

[11]: Site officiel de Webots : https://cyberbotics.com/

[12]: site officiel de gymnasium : https://gymnasium.farama.org/

[13]: site officiel de Stable-Baselines3: https://stable-baselines3.readthedocs.io

[14]: site officiel de tensorboard : https://www.tensorflow.org/tensorboard?hl=fr

[15]: Annexes Apprentissage par renforcement et transfert simulation vers réalité pour la conduite de voitures autonomes, R. Bennani, K. Hoarau, A. Juton, https://eduscol.education.fr/sti/si-ens-paris-saclay/ressources_pedagogiques/apprentissage-renforcement-transfert-simulation-vers-realite-pourla-conduite-voitures-autonomes

[16]: Wei Zhu, Xian Guo, Dai Owaki, Kyo Kutsuzawa, Mitsuhiro Hayashibe. "A Survey of Sim-to-Real Transfer Techniques Applied to Reinforcement Learning for Bioinspired Robots". In: IEEE Transactions on Neural Networks and Learning Systems (sept. 2021), https://ieeexplore.ieee.org/document/9552429

[17]: Apprentissage par renforcement pour la conduite de voiture autonome, R. Bennani, K. Hoarau, A. Juton, mai 2024, https://eduscol.education.fr/sti/si-ens-paris-sacaly/ressources_pedagogiques/apprentissage-par-renforcement-pour-la-conduite-de-voiture-autonome