Ecole Normale Supérieure de Paris-Saclay

Rapport TER

TER - Voiture autonomes avec apprentissage par renforcement et lidar

---

15 mai 2024

Miquel Hugo

Plus Basile

Table des matières

1 Introduction

1.1 Contexte

Les voitures autonomes sont un sujet de recherche très actif depuis quelques années. En effet, elles pourraient révolutionner le monde des transports en permettant de réduire les accidents de la route, de diminuer la consommation d'énergie et de réduire les embouteillages. Cependant, il reste encore de nombreux défis à relever pour que les voitures autonomes soient utilisées à grande échelle. En particulier, il est nécessaire de développer des algorithmes d'apprentissage par renforcement qui permettent à une voiture autonome d'apprendre à conduire de manière autonome.

1.2 Objectif et travail réalisé

L'objectif de ce TER est de développer un algorithme d'apprentissage par renforcement qui permet à une voiture RC au format $1/10^{\text{ème}}$ de conduire de manière autonome sur un circuit. Dans un premier temps, nous avons utilisé Webots pour la simulation, gym et stable baselines pour l'apprentissage par renforcement. Dans un second temps, nous avons transféré le réseau de neurones du simulateur à la voiture réelle. La voiture est équipée d'un lidar qui permet de mesurer la distance entre la voiture et les murs du circuit.

Note : Presenter Webots, la voiture réelle, la voiture sur simulateur, le lidar, le circuit etc...

1.3 L'apprentissage par renforcement

L'apprentissage par renforcement est une méthode d'apprentissage automatique qui permet à un agent d'apprendre à prendre des décisions en interagissant avec un environnement.

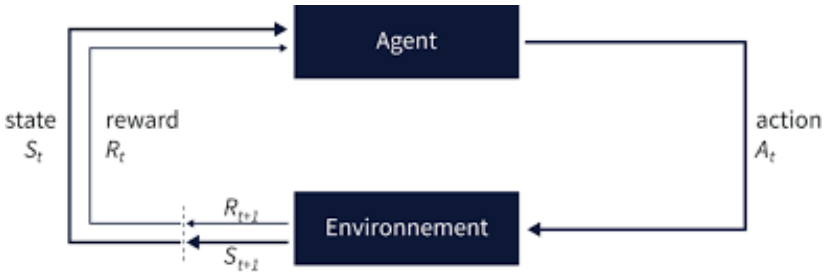

FIGURE 1 – Schéma de l'apprentissage par renforcement

L'agent prend des actions dans l'environnement et reçoit une récompense en fonction de l'action qu'il a prise. L'objectif de l'agent est de maximiser la somme des récompenses qu'il reçoit au cours des itérations.

Lors de chaque étape, l'agent reçoit une observation de l'environnement dans lequel il évolue. Sur la base de cette observation, l'agent prend une décision parmi un ensemble d'actions possible appelé espace des actions. Cet espace peut dépendre de l'état dans lequel se trouve l'agent.

Un exemple simple est celui d'un jeu d'échecs dans lequel l'observation correspond à la position de chacune des pièces sur l'échiquier et l'espace des actions est l'ensemble des déplacements possibles des pièces. Naturellement, on souhaite que l'agent réalise la meilleure action possible suivant l'observation reçue. Pour atteindre ce but, l'agent

applique une politique d'action (notée π) qu'il utilise pour sa prise de décision. À chaque récompense obtenue, cette politique est mise à jour. On espère ainsi atteindre une politique optimale.

Pour entraîner un agent, plusieurs types de méthodes peuvent être utilisées. Ces méthodes estiment la somme des récompenses futures que l'agent devrait obtenir. Ces récompenses sont pondérées pour favoriser les récompenses à court terme. La politique obtenue est souvent modélisée par un réseau de neurones, dont l'actualisation modifie les poids du réseau.

Les méthodes d'apprentissage par renforcement peuvent être classées en trois catégories principales :

- **Méthodes basées sur la valeur (value-based)** : Ces méthodes se concentrent sur l'estimation de la récompense cumulative optimale que l'agent peut obtenir. Elles cherchent à obtenir une récompense cumulative maximale.
- **Méthodes basées sur la politique (policy-based)** : Ces méthodes se concentrent sur l'optimisation de la politique de l'agent. Les valeurs de récompense peuvent ne pas être calculées directement.
- **Méthodes acteur-critique (actor-critic)** : Ces méthodes utilisent deux réseaux de neurones. Le premier réseau choisit l'action à effectuer, tandis que le second réseau évalue cette action en la comparant à l'action prévue.

2 Simulation

2.1 Le simulateur Webots

Le logiciel utilisé pour la simulation est Webots R2023b. Webots est un logiciel de simulation de robotique développé par Cyberbotics. Il permet de simuler des robots dans un environnement 3D que l'on peut personnaliser. Dans notre cas, nous avons utilisé Webots pour simuler une voiture RC sur un circuit. Nous avons utilisé le langage de programmation Python pour contrôler la voiture dans le simulateur.

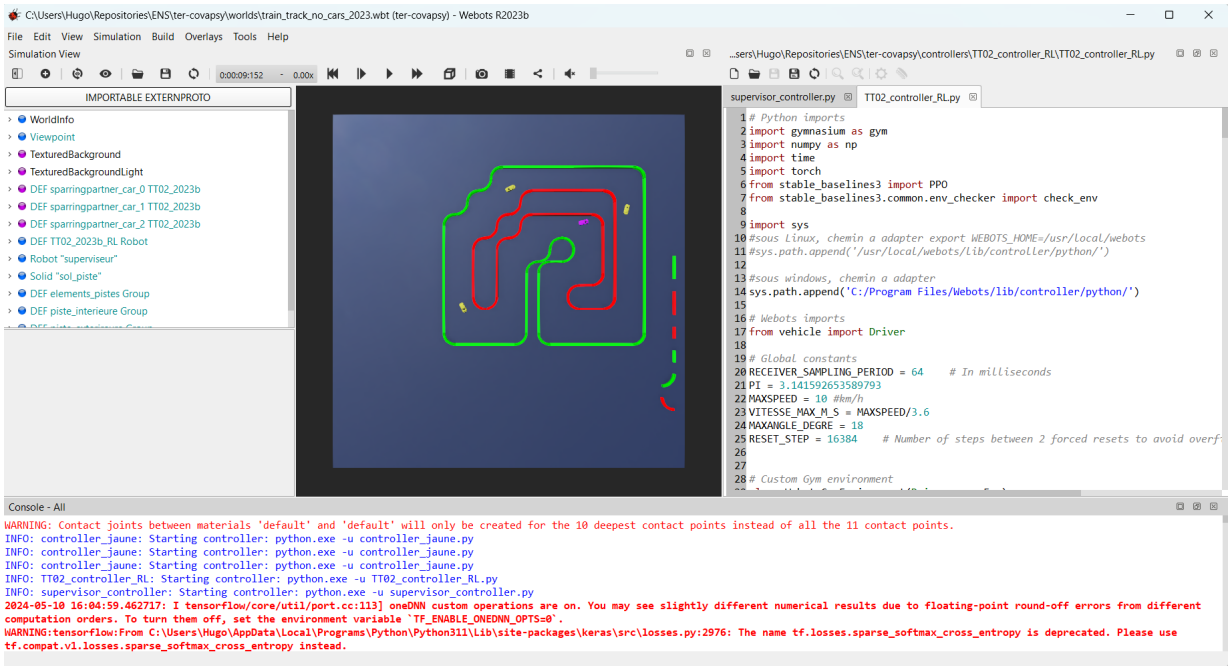

FIGURE 2 – Capture d'écran de Webots

Note : Présenter le circuit, la voiture, le lidar, le code Python etc...

2.2 Le circuit

Le circuit dans l'environment de Webots a pour but de simuler un circuit réel sur lequel la voiture autonome doit apprendre à conduire. Les murs du circuit sont composés de blocs de couleur différentes pour les bordures extérieur et intérieur du circuit. Ces murs ont une auteur d'une dizaine de centimètres qui permettent au lidar de mesurer la distance entre la voiture et les murs.

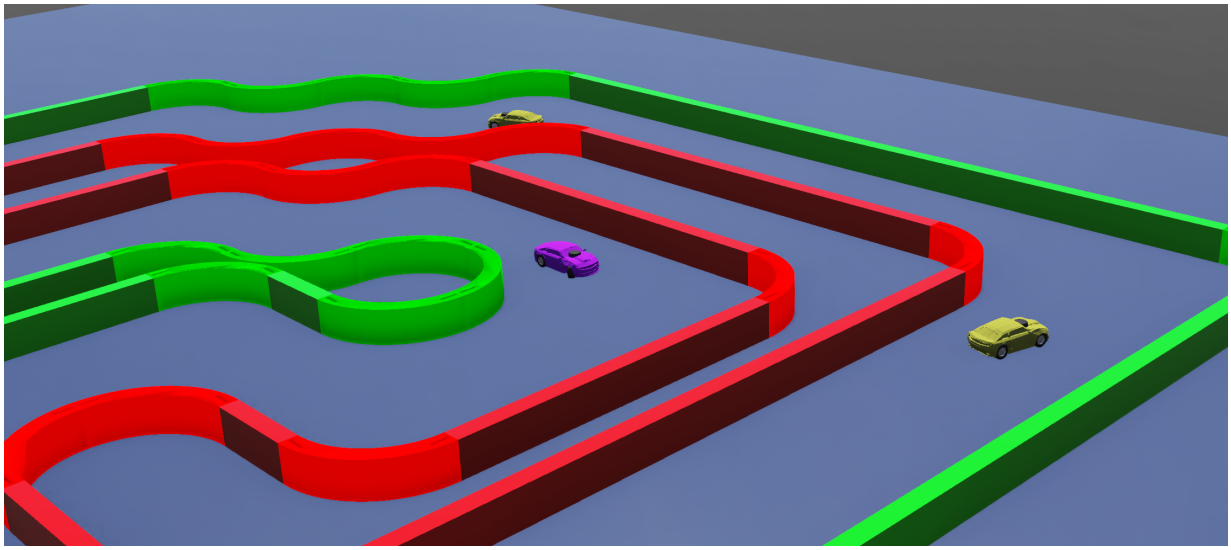

FIGURE 3 – Capture d'écran du circuit

2.3 La voiture

La voiture utilisée dans le simulateur est une voiture RC au format $1/10^{\text{ème}}$. Elle a pour objectif de reproduire le plus fidèlement possible une voiture réelle. La voiture est équipée d'un lidar qui permet de mesurer la distance entre la voiture et les murs du circuit.

FIGURE 4 – Capture d'écran de la voiture

2.4 Le lidar

3 Simulation to real world

L'objectif du SimToReal est de transférer un réseau de neurones entraîné sur un simulateur à une voiture réelle. Une fois le réseau de neurones entraîné, il est transféré à la voiture réelle pour qu'elle puisse conduire de manière autonome sur un circuit réel. Un des principaux défis du SimToReal est de reproduire le plus fidèlement possible les conditions du monde réel dans le simulateur.

La phase d'entraînement sur simulateur a plusieurs avantages. Tout d'abord, elle permet de réduire le temps et le coût de l'entraînement. En effet, il est possible de simuler des milliers d'épisodes en quelques heures alors qu'il faudrait plusieurs jours pour réaliser le même nombre d'épisodes sur une voiture réelle. De plus, la simulation permet de tester des scénarios dangereux pour la voiture réelle sans risquer de l'endommager. Sur la voiture réelle, il faut aussi la replacer à la main après chaque crash dans un mur. Il est donc plus efficace de réaliser l'entraînement sur simulateur.

4 Introduction du bruit pour améliorer la simulation

Pour que la simulation soit la plus proche possible de la réalité, il est crucial d'introduire des éléments de bruit. Ces bruits peuvent affecter les mesures et les actions de la voiture, et ainsi permettre au réseau de neurones de mieux généraliser lors du passage au monde réel.

4.1 Bruit sur les mesures

Dans un environnement réel, les capteurs ne fournissent pas des mesures parfaites. Les mesures du Lidar peuvent toujours être affectées par de petites interférences ou des variations mineures. Pour simuler ces conditions, on peut ajouter un léger bruit gaussien aux mesures du Lidar. Cela permet au réseau de neurones de s'adapter à des données moins parfaites, similaires à celles qu'il rencontrera dans le monde réel.

4.2 Bruit sur les actions

Les actions de la voiture, telles que l'angle de direction ou la vitesse, peuvent également être sujettes à des variations imprévues. Par exemple, un servomoteur peut ne pas toujours répondre de manière identique à une même commande en raison de l'usure ou des variations de tension. Pour prendre en compte ces imperfections, on peut ajouter du bruit aux actions commandées par le réseau de neurones. Cela aide à rendre l'agent plus robuste face aux variations qu'il pourrait rencontrer sur une voiture réelle.

5 Application à la voiture autonome

Pour appliquer l'apprentissage par renforcement à une voiture autonome, il est essentiel de définir correctement l'espace d'observation et l'espace d'action en tenant compte des contraintes du monde réel.

5.1 Espace d'observation

L'espace d'observation doit inclure toutes les informations pertinentes que la voiture peut obtenir de son environnement. Dans notre cas, nous utilisons le Lidar pour mesurer les distances aux obstacles. Si un système de contrôle de la vitesse est en place, la vitesse actuelle de la voiture pourrait également faire partie de l'espace d'observation.

5.2 Espace d'action

L'espace d'action est limité aux commandes que l'agent peut envoyer à la voiture. Cela inclut l'incrémentation de l'angle de direction via le servomoteur et l'incrémentation de la vitesse via le moteur.

6 Entraînement du réseau de neurones

6.1 Algorithme d'apprentissage

Circuit d'entraînement

La piste d'entraînement utilisée est conçue pour offrir une variété de situations afin que la voiture puisse apprendre à gérer différentes circonstances qu'elle pourrait rencontrer en course. Le circuit est composé de virages à gauche et à droite, de longues lignes droites, de virages en épingle et d'une section en diagonale. La piste retenue est la suivante :

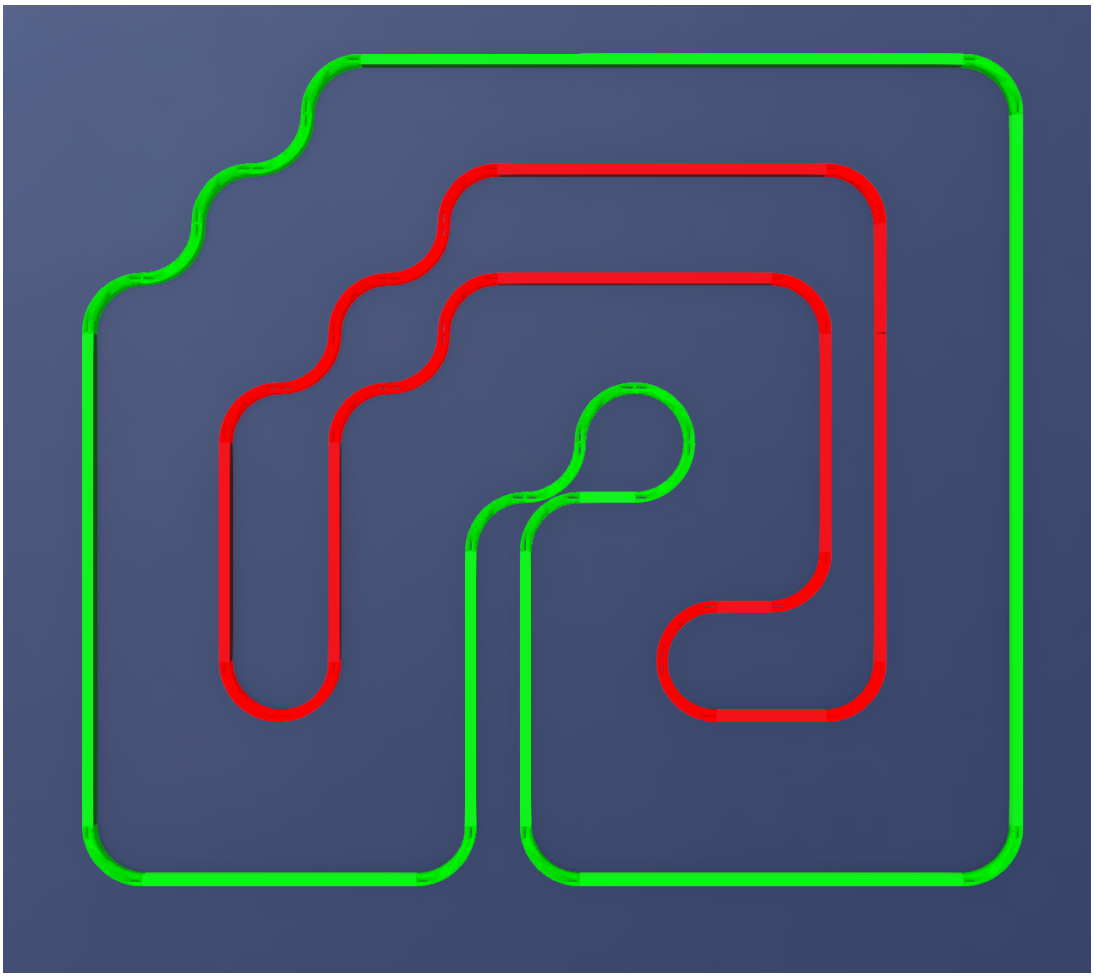

FIGURE 5 – Piste d'entraînement pour l'apprentissage par renforcement

Sur ce circuit, nous avons ajouté trois autres voitures qui roulent à vitesse modérée pour simuler des situations de dépassement. Ces voitures suivent un algorithme de conduite simple qui consiste à rester à distance des murs.

Pour cet environnement, nous avons également intégré un robot "Superviseur" qui repositionne les voitures à des positions aléatoires lorsque nous souhaitons recommencer un épisode. Ce "Superviseur" choisit aléatoirement une position et une direction sur le circuit pour replacer les voitures, assurant ainsi une situation initiale nouvelle à chaque début d'épisode. La communication avec le "Superviseur" se fait via un système émetteur-récepteur intégré dans Webots.

Environment Gym

La bibliothèque Gym est une bibliothèque implémentée en Python qui permet de gérer la partie apprentissage par renforcement d'un réseau de neurones. Dans le cas de notre projet, Gym ne propose pas d'environnement adapté. Il a donc été nécessaire de créer un tout nouvel environnement. Gym 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 donnant la démarche pour repartir au début d'un épisode
- **step()** : fonction faisant évoluer l'environnement

Toutes ces fonctions sont rassemblées dans une classe que nous avons nommée `WebotsGymEnvironment`. Dans cette classe, nous avons rajouté quatre fonctions propres à la voiture :

- **get_lidar_mm()** : Fonction qui renvoie un tableau de Lidar dans le bon format avec des valeurs cohérentes en mm.
- **set_vitesse_m_s()** : Fonction qui prend en argument une vitesse en m/s et qui la convertit en km/h avant de l'envoyer à la voiture.
- **set_direction_degre()** : Fonction qui prend un angle en degré et le convertit en radian avant de l'envoyer à la voiture.

Nous allons détailler par la suite chacune des fonctions.

get_observation()

Dans la fonction `get_observation()`, on appelle la fonction `get_lidar_mm()` pour récupérer les observations du Lidar. Ce tableau sera le tableau donné en entrée du réseau pour les observations du Lidar à "l'instant présent". Pour ce qui est du tableau à "l'instant précédent", on stocke à chaque appel de la fonction le tableau d'observation dans une variable interne de la classe. Si la fonction est appelée depuis la fonction `reset()`, on donne le même tableau pour les deux parties de l'espace d'observation concernant le Lidar puisque la voiture a été repositionnée. Pour la vitesse et la direction, Webots nous permet de mesurer la vitesse et la direction de la voiture dans le simulateur. Ce sont ces données que l'on donne au réseau de neurones. De même dans le cas où l'observation est demandée par la fonction `reset()`, on indique que ces deux grandeurs sont nulles. Il est à noter que les valeurs données à l'espace d'observation sont normalisées.

get_reward()

Dans la fonction `get_reward()`, on donne la récompense associée à l'état dans lequel se trouve la voiture. On distingue deux états possibles pour la voiture. Le premier état est celui d'une situation de collision. On regarde la plus petite valeur du tableau de Lidar actuel et si celui-ci est inférieur à 120 mm, on considère qu'il y a collision. Dans le

cas de la collision, on donne un malus de -400 moins la vitesse de la voiture. 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. Les fonctions de récompenses seront détaillées plus tard. On indique aussi ici si l'on a terminé l'épisode via la variable `done`.

reset()

Dans la fonction `reset()`, on indique la démarche à suivre lorsque l'on veut retourner au début d'un épisode. Le reset se fait dans le cas d'un crash de la voiture ou si le nombre d'actions autorisées par épisode est atteint. Au moment du reset, on donne des consignes de vitesse et d'angle nuls et on envoie une indication de reset au robot "Superviseur". À la fin de la routine de réinitialisation, on renvoie une observation.

step()

Dans la fonction `step()`, on indique que l'on fait un pas dans le processus d'apprentissage. Dans cette fonction, on fait avancer la voiture avec les actions récupérées depuis le réseau de neurones. Ensuite, on récupère une observation depuis le Lidar et on fait faire un pas au processus d'apprentissage. On calcule la récompense avant de retourner toutes les informations obtenues.

6.2 Réseau de neurones avec Stable-Baselines3

La librairie Stable-Baselines3 permet de créer un réseau de neurones géré par l'algorithmes d'apprentissage par renforcement PPO (Proximal Policy Optimization). Stable-Baselines3 propose un large choix de paramètres pour l'apprentissage. Voici un descriptif des paramètres utilisés ainsi que leurs valeurs définies dans notre programme :

- **policy** : Type de politique utilisée. Dans notre cas, nous utilisons 'MultiInputPolicy' pour gérer les multiples entrées.
- **env** : Environnement d'apprentissage Gym.
- **learning_rate** : Taux d'apprentissage fixé à $5 \cdot 10^{-4}$.
- **n_steps** : Nombre d'actions autorisées par épisode (2048).
- **batch_size** : Taille du lot de données pour chaque mise à jour (64).
- **n_epochs** : Nombre d'époques d'entraînement par itération de `n_steps` (10).
- **gamma** : Facteur de discount pour la récompense future (0.99).
- **gae_lambda** : Facteur pour le calcul de l'estimateur de l'avantage généralisé (0.95).
- **clip_range** : Valeur de clipping pour PPO (0.2).
- **vf_coef** : Coefficient de la fonction de perte de la valeur (1).
- **ent_coef** : Coefficient d'entropie pour la fonction de perte (0.01).
- **device** : Composant sur lequel faire tourner l'algorithme (dans notre cas, 'cuda :0' pour l'utilisation du GPU).
- **tensorboard_log** : Emplacement pour enregistrer les données de Tensorboard ('./PPO_Tensorboard').

Voici un extrait de notre code de définition de modèle :

```
# Définition du modèle
model = PPO(
    policy="MultiInputPolicy",
    env=env,
    learning_rate=5e-4,
    verbose=1,
    device='cuda:0',
```

```
    tensorboard_log='./PPO_Tensorboard',
    # Paramètres additionnels
    n_steps=2048,
    batch_size=64,
    n_epochs=10,
    gamma=0.99,
    gae_lambda=0.95,
    clip_range=0.2,
    vf_coef=1,
    ent_coef=0.01
)
```

Nous pouvons ensuite entraîner le modèle avec la fonction `model.learn()`. Cette fonction prend en argument le nombre d'itérations d'entraînement. Une fois l'entraînement terminé, nous pouvons sauvegarder le modèle avec la fonction `model.save()`.Nous pouvons dans un second temps charger le modèle avec la fonction `model.load()` ce qui nous permet, par exemple de reentrainer le modèle avec de nouveau paramètres ou avec une nouvelle fonction de récompense. Nous pouvons également visualiser les données d'entraînement avec Tensorboard en exécutant la commande `tensorboard -logdir ./PPO_Tensorboard`.

6.3 Fonction de récompense

La fonction de récompense est un élément crucial de l'apprentissage par renforcement. Elle permet de guider l'agent vers les actions qui maximisent la récompense. C'est un élément délicat à définir car une mauvaise fonction de récompense peut entraîner des comportements indésirables de l'agent. En effet si l'on ne prend pas assez en compte le crash de la voiture, l'agent pourrait apprendre à foncer dans les murs pour maximiser la vitesse. Si l'on ne prend pas en compte la vitesse, l'agent pourrait apprendre à rester immobile pour éviter les collisions et rester éloigné des murs. Il est donc important de trouver un équilibre entre ces aspects.

Dans notre cas, nous avons défini une fonction de récompense qui prend en compte la vitesse de la voiture et la distance minimale donnée par le Lidar. La récompense est définie comme suit :

$$\text{reward} = \begin{cases} -400 - 10 * \text{vitesse} & \text{si collision} \\ 18 * (\text{mini} - 0.018) + 2 * \text{vitesse} & \text{sinon} \end{cases} \tag{1}$$

Où mini est la distance minimale donnée par le Lidar et vitesse est la vitesse de la voiture. La récompense est négative en cas de collision et dépend de la vitesse de la voiture. Cela permet de fortement pénaliser les collisions à haute vitesse. Dans le cas où il n'y a pas de collision, la récompense dépend de la distance minimale donnée par le Lidar et de la vitesse de la voiture. Cela permet de favoriser les actions qui permettent à la voiture de rouler vite tout en restant loin des murs pour éviter les collisions.

Nous avons également essayé de prendre en compte le temps de passage au tour. Pour cela, nous avons adapté le superviseur afin de détecter les passages sur la ligne d'arrivée et de notifier la fonction de récompense pour donner une récompense positive lors d'un passage sur la ligne d'arrivée. Plus le temps au tour est court, plus la récompense est élevée. Cela permet de former la voiture à vraiment aller vite et à réaliser un temps au tour le plus rapide possible. Cette approche encourage l'agent à optimiser non seulement la vitesse et la sécurité, mais aussi l'efficacité globale de ses déplacements sur le circuit.

6.4 Mise en place de la simulation

Après avoir défini l'environnement Gym, le réseau de neurones et la fonction de récompense, nous pouvons lancer l'entraînement de la voiture autonome.

Entraînement

La première étape consiste à déclarer l'environnement Gym et à vérifier que celui-ci est correctement défini avec la fonction `check_env()` :

```
env = WebotsGymEnvironment()
check_env(env)
```

Après cela, nous pouvons définir le modèle et lancer l'entraînement avec la fonction `model.learn()`. Nous donnons en argument `total_timesteps` qui correspond au nombre total d'itérations d'entraînement. Nous pouvons ensuite sauvegarder le modèle avec la fonction `model.save()` qui prend en argument le nom du fichier de sauvegarde.

7 Conclusion

Note : Il faut ajouter la partie ou on ameliore la simulation webot avec les obstacles, basile l'a fait. Il faut details Stable-baselines et expliquer les hyperparameter qu'on a utilisé. policy, env, gamma, batch size, n_epochs, n_steps, ent_coef, learning_starts, tensorboard_log, verbose, n_cpu_tf_sess, n_timesteps etc...

decrir le deroulement de la simulation, les problemes, presenter les graphs de PPO (tres jolie graph d'entrainempent)

presenter le temps au tour comme critere de performance