

Entrainement d'une voiture autonome via RL

P.G Basile^{1*}

¹ École Normale Supérieure Paris-Saclay, France

15 Mai 2024

RESUME

Le projet vise à améliorer le transfert sim2real en utilisant une méthode de randomisation de domaine. Le modèle de voiture autonome est entraîné sur simulateur avant d'être transféré sur la voiture réelle format 1:10. Le document vise à aider les prochains étudiants du projet.

Mots clés: Reinforcement Learning – Simulation to Reality – PPO algoritm – Autonomous car

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^{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.

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.

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

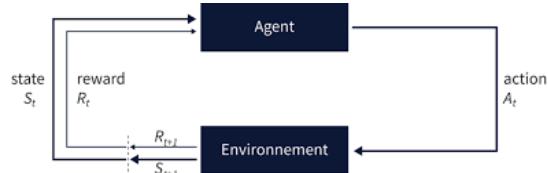


Figure 1. Schéma de l'apprentissage par renforcement

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.

2 P.G Basile

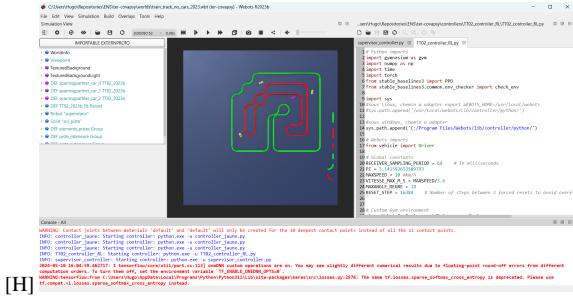


Figure 2. Capture d'écran de Webots

- **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.

2.2 Le circuit

Le circuit dans l'environnement 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 extérieure et intérieure 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.

2.3 La voiture

La voiture utilisée dans le simulateur est une voiture RC au format 1/10^{ème}. Elle a pour objectif de reproduire le plus fidèlement possible une voiture réelle. La voiture est équipée d'un lidar qui permet

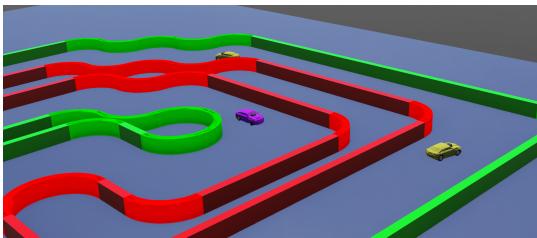


Figure 3. Forme du circuit



Figure 4. Capture d'écran de la voiture

de mesurer la distance entre la voiture et les murs du circuit à 360 degrés avec une résolution de 1 degré.

2.4 Les secteurs

Afin de pouvoir comparer différentes fonctions de récompense et mesurer la performance de la voiture il est possible de mesurer le temps au tour de la voiture (à condition qu'elle finisse un tour complet). Plus de précision sont données dans la section 4.1 Comme la voiture met un certain temps à apprendre à conduire et qu'elle ne termine un tour qu'au bout d'un certain temps, il est intéressant de diviser le circuit en secteurs. Les différentes portions du circuit sont représenté sur la figure 5.

Afin de calculer les temps, on doit pouvoir accéder à la position de la voiture. Or la position n'est accessible qu'au superviseur webots. Pourtant nous avons besoin de cette information dans le controller de la voiture (TT02_RL_controller.py) car c'est là qu'est défini la fonction reward et nous souhaitons pouvoir utiliser ces informations de temps au tours/secteurs pour calculer une récompense.

Webots ne permet pas de définir des variables globales ou d'importer des variables définies dans d'autres fichiers dans un soucis de modélisation : dans le monde réel la voiture ne peut pas accéder à sa position si elle n'a pas les capteurs nécessaires. Pour

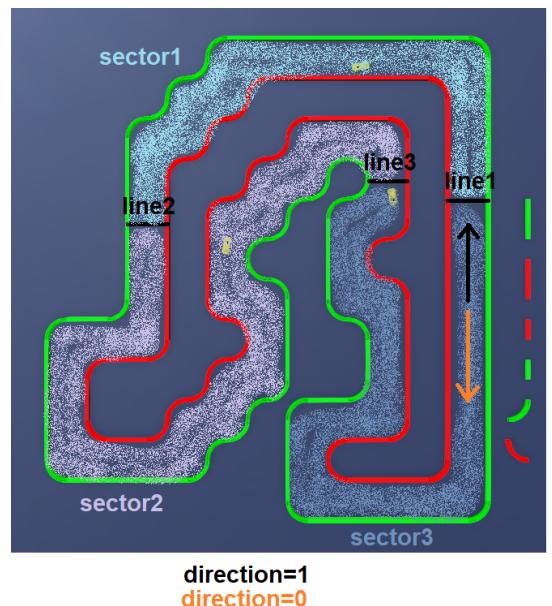


Figure 5. Description des secteurs

contourner ce problème, nous avons ajouter au controller ainsi qu'à la voiture (qui sont tous deux des objets robots dans Webots) des émetteurs/receveur.

Ainsi on indique à la simulation que notre voiture est munie d'un capteur radio capable de recevoir des informations du controller, notamment ses temps au tours. Ces récepteurs sont nommés `LapEmitter` et `LapReceiver` dans les codes `supervisor_controller.py` et `TT02_RL_controller.py` disponibles ici dans le dossier `simulatedCar/webotsWorld/controllers`.

La même mécanique est utilisée pour que la réinitialisation lors d'un crash : le superviseur a besoin des données lidar pour savoir si la voiture a crashé. Ces données sont envoyées par la voiture au superviseur via un émetteur/recepteur nommé `resetEmitter` et `resetReceiver`.

Les performances de la voiture (temps au tour, secteur etc.) sont directement affichable dans tensorboard. Pour ouvrir tensorboard il suffit de :

- ouvrir une invite de commande depuis le dossier controller TT02
- Executer la commande suivante : `bash python3 -m tensorboard.main --logdir PPOTensorboard/`
- Récupérer l'adresse https donnée par la console et la coller dans un navigateur

3 APPLICATION À LA VOITURE AUTONOME

Pour appliquer l'apprentissage par renforcement à une voiture autonome, nous devons définir correctement l'espace d'observation et l'espace d'action en tenant compte des contraintes du monde réel. En effet, tout ce qui est possible dans un simulateur n'est pas forcément réalisable dans le monde réel.

3.1 Espace d'action

L'espace d'action est limité aux commandes que l'agent peut envoyer à la voiture. Cela inclut:

- La vitesse de la voiture
- L'angle de direction de la voiture

Dans notre cas, les actions seront le fait d'augmenter ou de diminuer les consignes de vitesse et d'angle de direction.

3.2 Espace d'observation

L'espace d'observation doit inclure toutes les informations pertinentes que la voiture peut obtenir de son environnement. Dans notre cas, 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.

Dans notre cas nous avons donné les données du lidar à l'instant présent et à l'instant précédent, cela permet de donner une information sur la vitesse de la voiture et une certaine inertie à la voiture. Nous avons également les anciennes consignes de vitesse et

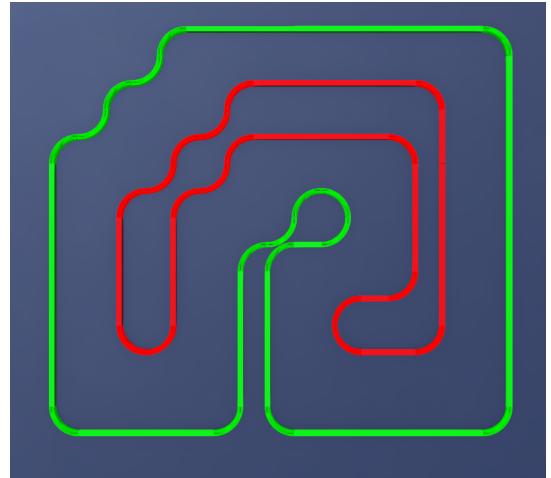


Figure 6. Piste d'entraînement pour l'apprentissage par renforcement

d'angle de direction pour donner une information sur la dynamique de la voiture.

4 ENTRAÎNEMENT DU RÉSEAU DE NEURONES

4.1 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:

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 conforme 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

4 P.G Basile

- **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`.

Afin de pouvoir une metrics cohérente entre différentes fonctions de récompenses, il est possible de mesurer le temps au tour et au secteur de la voiture. Ainsi on peut comparer différentes fonctions de récompenses et mesurer la performance de la voiture. Plusieurs autres paramètres ont été ajoutés, comme le nombre de collisions, le nombre de tours, le nombre de secteurs, le temps total, le meilleur temps au tour. Tout est affichable

`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.

4.2 Réseau de neurones avec Stable-Baselines3

La librairie Stable-Baselines3 permet de créer un réseau de neurones géré par l'algorithme 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 : [frame=lines, framesep=2mm, baselinestretch=1.2, bg-color=LightGray, fontsize=, linenos] python 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 nouveaux 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`.

4.3 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 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:

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

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.

4.4 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.

Entrainement

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

```
[ frame=lines, framesep=2mm, baselinestretch=1.2, bg-color=LightGray, fontsize=, linenos ]python env = WebotsGymEnvironment() checkEnv(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.

5 SIMULATION TO REAL WORLD

5.1 Le problème du SimToReal

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 d'éviter d'avoir une voiture dont les performances sont bonnes sur simulateur mais mauvaises sur la voiture réelle.

L'entraînement sur simulateur a beaucoup d'avantages. Tout d'abord, il 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 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.

Plusieurs solutions ont été explorées pour améliorer le passage du simulateur à la voiture réelle. Nous allons les détailler dans la suite de cette section. Nous avons implémenté un certain nombre de fonctions dans le but de randomiser le domaine (*domain randomization*). En ajoutant de l'aléatoire à la simulation on évite d'overfitter le cadre proposé dans le simulateur, et le modèle est capable de répondre à un environnement variable et imparfait. Ainsi le modèle est plus robuste et le passage sim2real est facilité(@warning Citation 'tobin2017domain' on page 5 undefined).

Voici les fonctionnalités que nous allons détailler plus bas :

- Amélioration de la ressemblance entre le simulateur et le monde réel
 - Ajout de bruit de mesure
 - Ajout d'aléatoire dans les commandes
 - Ajout d'obstacles aléatoires

5.2 Amélioration de la ressemblance entre le simulateur et le monde réel

5.2.1 Amélioration de la simulation

La première solution pour améliorer le passage du simulateur à la voiture réelle est d'avoir une simulation et un monde réel les plus proches possibles. Dans cet effort, deux solutions sont possibles :

- Améliorer la simulation pour qu'elle soit la plus proche possible de la réalité.
- Modifier le monde réel pour qu'il soit le plus proche possible de la simulation.

Pour améliorer la simulation, on peut jouer sur différents

6 P.G Basile

paramètres du moteur physique. Par exemple, dans la simulation initiale de Webots, le couple des moteurs de direction est de 1e4 Nm ce qui n'est pas réaliste. Le moteur physique de Webots est puissant et les paramètres par défaut ne sont pas réalistes. Il est par exemple possible d'ajouter des coefficients de frottements entre les pneus et le sol via `WorldInfo -> contactProperties`. Il suffit ensuite de remplir les champs `contactMaterial` des objets `Solid "sol_piste"` et `wheel`.

Toutefois la modification des paramètres physiques de la simulation est délicate et des comportements inattendus apparaissent très souvent. Par exemple l'observation des roues se désagrégant après quelques secondes de simulation lorsque des frottements sont simulés.

On peut toutefois lister quelques paramètres modifiables pour améliorer la simulation:

- **Coefficient de frottement** : Il est possible d'ajouter/modifier le coefficient de frottement entre les roues et le sol pour que la voiture glisse plus ou moins.
- **Masse de la voiture** : La masse de la voiture peut être modifiée.
- **Puissance des moteurs** : Le couple et la vitesse max des moteurs peut être augmentée ou diminuée.
- **Le nombre de roues motrices** : Selon la voiture réelle utilisée, il peut être nécessaire de modifier le nombre de roues motrices.
- **Angle de braquage des roues** : L'angle de braquage des roues peut être modifié pour que la voiture puisse tourner plus ou moins.
- **Lidar** : Les paramètres du Lidar peuvent être modifiés.

5.2.2 Correction sur la voiture réelle

Il est aussi possible de jouer sur certains paramètres physiques réels. Les **moteurs** Dynamixel utilisés pour la direction sont configurables via Dynamixel Wizard

Il est très important de faire tourner la voiture réelle une fois la simulation prise en main pour prendre conscience des ces écarts simulation/réalité et essayer de les corriger. Certains sont propres à la voiture et peuvent être corrigés directement via le code python de contrôle de la voiture.

5.2.2.1 Lidar En testant le bon fonctionnement du **Lidar**, nous avons constaté que certains points Lidar valaient 0 dans une zone dégagée. Dans ce cas, nous avons implémenté une simple interpolation linéaire pour remplacer ces valeurs par des valeurs cohérentes.

5.2.2.2 Batterie Le niveau de la **batterie** influence aussi beaucoup la réactivité des moteurs. Si c'est un paramètre qui peut être pris en compte dans la simulation, s'assurer d'utiliser une batterie chargée pour les tests réels est une solution plus efficace.

Nous avons identifié les performances suivantes :

5.2.2.3 Moteur de direction Les **angles de braquage** peuvent être étudiés sur la voiture réelle. La commande du moteur de direction utilisé sur la voiture réelle (Dynamixel AX12) se fait via l'envoie d'un nombre entier. Les commandes dépendent des

	Batterie vide	0.8 m/s en imposant 1 m/s
[H]	Batterie pleine	2.3 m/s en imposant 1 m/s
	Vitesse max batterie pleine	10 m/s

paramètres de configuration sur Dynamixel Wizard. Il existe deux butées pour les moteurs :

- Une butée mécanique qui empêche le moteur de tourner plus loin sur le système de direction.
- Une butée logicielle qui empêche le moteur de tourner plus loin que la valeur maximale ou minimale configurée (configurable sur Wizard)

Selon le système de direction utilisé les angles de braquage limites ne sont pas les mêmes. Pour `voitureBasile` les angles sont de -18° et 18°. Les paramètres identifiés sont :

Comme on peut le constater la commande n'est pas linéaire et la plage -18° à 0° est plus grande que la plage 0° à 18°. Nous verrons dans la section 5.6 comment prendre en compte ces non linéarités dans l'entraînement. Les paramètres du moteur de direction peuvent être testés via le code `test_direction_AX12.py` disponible ici.

5.2.2.4 Moteur de propulsion Le moteur de propulsion utilisé est commandé via un signal PWM. Le test du moteur de propulsion se fait via le code `test_propulsion.py` disponible ici. Les performances identifiées sont les suivantes :

Pour ces raisons un entraînement sur simulateur est inévitable.

5.3 Ajout de bruit

Un entraînement supplémentaire de la voiture réelle pourrait s'avérer très pertinent en plus de l'entraînement sur simulation pour améliorer le passage sim2real. Dans ce cas, une voie d'exploration est d'utiliser un modèle de type Progressive Neural Network, qui permet au modèle d'intelligemment utiliser ses connaissances acquises sur simulateur tout en continuant d'apprendre sur le circuit réel (@warning Citation 'rusu2022progressive' on page 6 undefined).

Pour les voitures travaillant avec la vision, les images issues de la simulation sont encore plus éloignées des images de la caméra réelle que ce n'est le cas pour le Lidar. Il existe des architectures de modèles combinant CNN et vision transformer construits pour se concentrer sur les parties importantes de l'image et qui possèdent des bonnes capacités de généralisation (@warning Citation 'li2023style' on page 6 undefined).

	Angle souhaité	Commande moteur
[H]	-18° (butée gauche)	300
	0° (milieu)	460
	18° (butée droite)	570

Table 1. Commande moteur à imposer en fonction de l'angle de direction souhaité

	Paramètre	Valeur
[H]	Point zéro	7.09
	Point mort	0.399
	Delta prop max	1.5

Utiliser une simulation plus simple pourrait améliorer le passage sim2real. Webots est un moteur physique complexe et en plus d'être complexe à manipuler, il est difficile de comprendre les comportements inattendus. Il semblerait qu'une simulation très simpliste comme présentée dans cette vidéo pourrait présenter de meilleures performances, en plus d'être plus accessible. Cette idée selon laquelle le passage sim2real est améliorer lorsque la simulation est plus réaliste (et donc plus complexe) est contredite par les résultats de (@warning Citation 'pmlr-v205-truong23a' on page 6 undefined) qui montrent que des simulations plus simples peuvent être plus efficaces pour le passage sim2real.

5.4 Introduction du bruit pour améliorer la simulation

Pour améliorer le passage sim2real l'ajout de bruit est très efficace. En effet, les modèles de réseaux de neurones sont très sensibles aux données d'entrée. Si les données d'entrée sont trop propres, le modèle risque de surapprendre sur ces données et de ne pas généraliser sur des données plus bruitées.

Ces bruits peuvent intervenir à deux niveau :

- les mesures des capteurs
- les actions de la Voiture

5.4.1 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. Cette approche n'a pas encore été étudiée, nous avons considéré que les mesures du Lidar étaient suffisamment consistentes.

5.4.2 Bruit sur les actions

Les actions de la voiture, telles que l'angle de direction ou la vitesse, sont 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. Ainsi lorsque le réseau de neurones est transposé sur la voiture réelle, il ne sera pas surpris si les moteurs ne répondent pas exactement comme dans la simulation.

Les l'espace de commande que l'on obtient en sortie du réseau de neurones est un angle de direction et une vitesse. Il y a donc deux champs auxquels on peut ajouter du bruit à chaque itération :

- La commande de vitesse
- La commande d'angle

Ce bruit peut être ajouté dans les fonctions `set_vitesse_m_s()` et `set_direction_deg()` du fichier `TT02_controller_RL.py`. Ainsi à chaque commande en sortie du réseau de neurone, la voiture ne répondra pas exactement comme attendu mais répondra à une consigne bruitée.

Ce type de bruit n'a pas été implémenté. Pour notre étude nous avons préféré nous concentrer sur l'ajout d'aléatoire sur les commandes de la voiture mais non pas à chaque timesteps (cad à chaque nouvelle commande est associé un bruit aléatoire) mais à chaque réinitialisation de la voiture. Cette approche est explicitée dans la sous-section 5.5

5.5 Ajout d'aléatoire dans les commandes

Nous avons implémenté un système de randomisation des directions et des vitesses pour améliorer la robustesse du modèle. A chaque réinitialisation de la voiture, un biais aléatoire est ajouté à la commande de direction et de vitesse. L'aléatoire peut être activé ou désactivé via la variable `self.add_randomness` dans le fichier `TT02_controller_RL.py`.

5.5.1 Randomisation des directions

Commençons par décrire comment nous avons modélisé la réponse de la voiture à une consigne d'angle.

5.5.1.1 Modélisation Comme nous l'avons vu dans le tableau 1 l'angle des roues sur la voiture réelle n'est pas linéairement relié à la consigne moteur (et donc à l'angle du moteur de direction). Ainsi la modélisation suivante a été choisie pour modéliser sur le simulateur quel angle prennent les roues de la voiture en fonction de la consigne d'angle :

$$\alpha = \left(\frac{b}{324} \right) \theta^2 + \theta + b \quad (2)$$

Cette fonction a été tracée figure 7

où :

- α : L'angle en degrés réellement imposé à la voiture.
- θ : La consigne d'angle en degrés.
- b : Le biais directionnel, une constante ajoutée pour ajuster l'angle lorsque la consigne est $\theta = 0$.

Cette fonction passe par les points $(-18, -18)$, $(0, b)$ et $(18, 18)$.

Ainsi nous respectons les angles de butée de la voiture réelle

5.5.1.2 Ajout d'aléatoire Pour améliorer la robustesse du modèle, à chaque remplacement de la voiture, on initialise aléatoirement le paramètre `self.random_dir_bias` qui représente b . La valeur est choisie aléatoirement entre `[-self.max_random_dir_bias, self.max_random_dir_bias]`

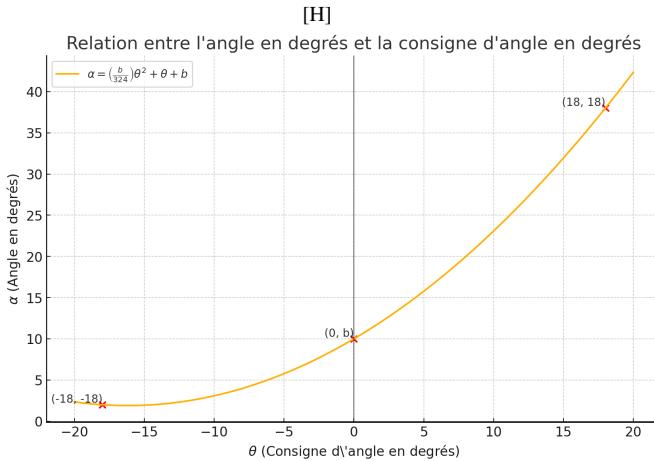


Figure 7. Modélisation de l'angle des roues en fonction de la consigne d'angle

5.6 Randomisation des vitesses

5.6.0.1 Modélisation Pour la vitesse, nous avons choisi de modéliser la réponse de la voiture à une consigne de vitesse de la manière suivante :

$$v_{\text{out}} = k \cdot v_{\text{in}} + b \quad (3)$$

- v_{out} : La vitesse réellement demandée à la voiture par le simulateur.
- v_{in} : La consigne de vitesse.
- k : Facteur de proportionnalité.
- b : Le biais ajouté à la vitesse.

5.6.0.2 Ajout d'aléatoire Ici aussi, à chaque remplacement de la voiture, on initialise aléatoirement le paramètre `self.random_speed_bias` qui représente b ainsi que le paramètre `self.random_speed_prop` qui représente k .

5.7 Ajout d'obstacles aléatoires

Pour améliorer la robustesse du modèle, nous avons ajouté des obstacles aléatoires sur le circuit (voir figure 8). Ces obstacles sont des blocs cubiques. Ils sont placés aléatoirement sur le circuit à chaque réinitialisation de la voiture. Leurs positions sont choisies aléatoirement parmi une liste de positions prédefinies.

La génération d'obstacles est gérée dans le code `supervisor_controller.py` disponible ici dans le fichier `controllers\TT02_controller_RL`. Le nombre d'obstacle à placer peut être modifié directement dans boucle while principale à la ligne

```
indexes = random.sample(range(0, len(
    obstacles_positions)), nb_obstacles)
```

L'ajout d'obstacles peut être désactivé via la variable `self.add_obstacle` dans le fichier `supervisore_controller.py`.

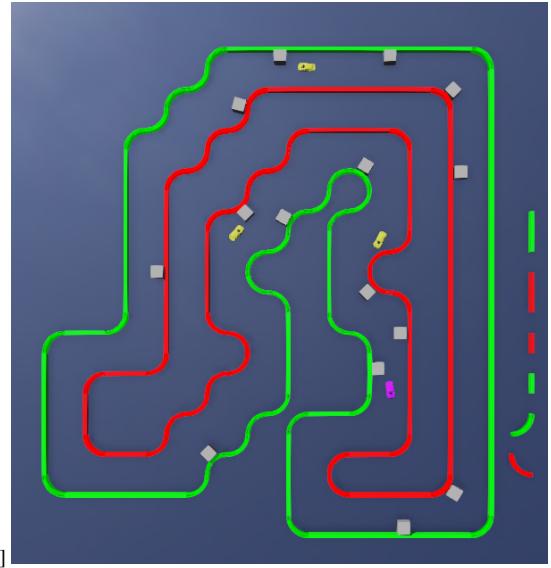


Figure 8. Circuit avec obstacles aléatoires

5.8 Position de départ aléatoires

Tout comme pour les obstacles aléatoires, la voiture démarre à une position aléatoire sur le circuit à chaque réinitialisation. La position de départ est choisie aléatoirement parmi une liste de positions prédefinies. On ajoute de plus un angle de départ aléatoire pour que la voiture ne démarre pas toujours dans la même direction.

5.9 Résultats

Des entraînements avec l'ajout d'aléatoire ont été lancés. Lorsque les paramètres `max_random_dir_bias`, `max_random_speed_prop`, `max_random_speed_bias` et le nombre d'obstacles ne sont pas trop élevés, la voiture est capable de conduire correctement et de faire des tours de circuit.

Le modèle obtenu est donc en théorie beaucoup plus robuste que précédemment pour des performances similaires. Cependant après des tests sur la voiture réelle, il s'avère que le modèle ne performe pas mieux que celui obtenu avant l'ajout d'aléatoire.

Nous expliquons ce résultat par un manque d'optimisation du code de la voiture réelle. L'élément limitant les performances de la voiture réelle n'est pas le modèle entraîné sur simulateur mais la façon dont ce modèle est utilisé sur la voiture réelle[†] via le code `programme_course.py` disponible ici. Il semble donc que travailler sur le code de la voiture réelle soit plus pertinent que de continuer d'essayer d'améliorer le modèle.

5.10 Améliorations possibles

nous conseillons en premier lieu de travailler sur le code de la voiture réelle pour améliorer les performances de la voiture. En effet, les performances de la voiture réelle sont très loin de

[†] les voitures utilisant le RL les plus performantes lors de la courses sont celles qui ont beaucoup travaillé sur le code voiture

celles obtenues sur simulateur. Ci dessous quelques autres pistes d'améliorations que nous n'avons pas pu explorer/implémenter.

Un entraînement supplémentaire de la voiture réelle pourrait s'avérer très pertinent en plus de l'entraînement sur simulation pour améliorer le passage sim2real. Dans ce cas, une voie d'exploration est d'utiliser un modèle de type Progressive Neural Network, qui permet au modèle d'intelligemment utiliser ses connaissances acquises sur simulateur tout en continuant d'apprendre sur le circuit réel (@warning Citation 'rusu2022progressive' on page 9 undefined).

Pour les voitures travaillant avec la vision, les images issues de la simulation sont encore plus éloignées des images de la caméra réelle que ce n'est le cas pour le Lidar. Il existe des architectures de modèles combinant CNN et vision transformer construits pour se concentrer sur les parties importantes de l'image et qui possèdent des bonnes capacités de généralisation (@warning Citation 'li2023style' on page 9 undefined).

Utiliser une simulation plus simple pourrait améliorer le passage sim2real. Webots est un moteur physique complexe et en plus d'être complexe à manipuler, il est difficile de comprendre les comportements inattendus. Il semblerait qu'une simulation très simpliste comme présentée dans cette vidéo pourrait présenter de meilleures performances, en plus d'être plus accessible. Cette idée selon laquelle le passage sim2real est améliorer lorsque la simulation est plus réaliste (et donc plus complexe) est contredite par les résultats de (@warning Citation 'pmlr-v205-truong23a' on page 9 undefined) qui montrent que des simulations plus simples peuvent être plus efficaces pour le passage sim2real.

6 PASSAGE À LA VOITURE RÉELLE

Une fois le réseau de neurones entraîné sur le simulateur, nous pouvons le transférer à la voiture réelle. C'est la dernière étape du projet. Cette transition du monde virtuel au monde réel nécessite une adaptation des algorithmes et des fonctionnalités pour s'adapter aux contraintes et aux spécificités de la réalité.

ADAPTATION DES COMMANDES

Pour garantir une interprétation adéquate des commandes générées par le réseau de neurones sur la voiture réelle, nous avons développé deux fonctions équivalentes à celles utilisées dans le simulateur : `set_vitesse_m.s()` et `set_direction_degre()`. Toutefois, cette fois-ci, les commandes sont transmises en m/s et en degrés de direction au lieu d'être directement appliquées. De plus, pour le Lidar, nous avons pris soin de traiter les données afin d'éviter les valeurs nulles dans les parties "utiles" du Lidar.

6.0.1 Chargement du réseau de neurones

Le chargement du réseau de neurones a été effectué en utilisant la fonction `load()` de la bibliothèque Stable-Baselines3. Une fois chargé, le réseau peut être utilisé via la fonction `predict()`, prenant en entrée les observations et renvoyant les actions prédictes. Toutefois, contrairement à la simulation, l'accès direct à la vitesse et à la direction n'est pas possible en raison de l'absence de système



Figure 9. Piste de qualification avec obstacles

d'asservissement. Seules les consignes de vitesse et d'angle de l'instant précédent sont fournies en tant qu'observations.

Lidar

Pour exploiter le Lidar, nous avons mis en place une classe `Lidar_TT02()` pour initialiser, démarrer et acquérir les valeurs du Lidar. Deux threads ont été lancés : `thread_scan_lidar` pour une acquisition continue des données Lidar, et `thread_conduite_autonome` pour la gestion du pilotage de la voiture. Dans la fonction `conduite_autonome()`, les données Lidar sont traitées pour prendre des décisions, telles que reculer en cas de détection d'obstacle.

6.1 Résultats

Dans un premier temps nous avons testé la voiture avec un des réseaux de neurones qui nous semblait le plus performant. Dans un premier temps nous faisons tourner la voiture seul sur un circuit sans obstacles, c'est la phase de qualification. La voiture est évaluée sur le temps qu'elle met pour faire deux tours du circuit. Notre voiture a réalisé un temps de 30,22 secondes à la première manche de la qualification. La deuxième phase de la qualification consiste à faire tourner la voiture sur un circuit avec des obstacles. Notre voiture n'a pas réussi à terminer le circuit et a été arrêtée par un obstacle, le temps retenue a été de 90 secondes. La voiture a donc réalisé un temps total de 120,22 secondes la plaçant en 11^{eme} position de la qualification.

Suite à cela, toutes les voitures sont mises en piste pour une course de 5 tours. Notre voiture a terminé en première position de la manche 2 de la course.

Une observation faites lors de la course est que la voiture se déplace en "zigzag" sur les lignes droites. Ce comportement était déjà présent sur le simulateur mais moins prononcé. Ce problème est probablement dû à un manque de précision dans le modèle de la simulation sur le simulateur. Un des paramètres de la voiture dans le simulateur est le couple maximale du servomoteur de la direction. Ce paramètre est fixé dans la simulation Webots à une valeur beaucoup trop élevée par rapport à la réalité. Cependant, baissé ce paramètre dans la simulation Webots a pour conséquence de complètement casser la simulation, la voiture n'avance plus sur le simulateur. C'est un problème à résoudre pour améliorer la simulation.



Figure 10. Piste de course

Un autre point qui pourrait améliorer les déplacements en "zigzag" de la voiture est d'ajouter un terme pénalisant le fait de changer de direction dans la fonction de récompense. Cela permettrait potentiellement d'encourager la voiture à rester dans la même direction lorsqu'elle est sur une ligne droite. C'est une piste d'amélioration pour la suite du projet.

7 CONCLUSION

Pour conclure, l'apprentissage par renforcement est une méthode très efficace pour entraîner une voiture autonome. Nous avons pu entraîner un réseau de neurones sur un simulateur et le transférer à une voiture réelle. La voiture a réalisé des performances très satisfaisantes lors de la course. Cependant, des améliorations sont possibles pour améliorer les performances de la voiture. Il serait nécessaire de mettre en place un système d'asservissement de la vitesse et de la direction pour permettre à la voiture de suivre les consignes données par le réseau de neurones. En effet la vitesse à consignes équivalentes n'est pas la même en fonction de la charge de la batterie.