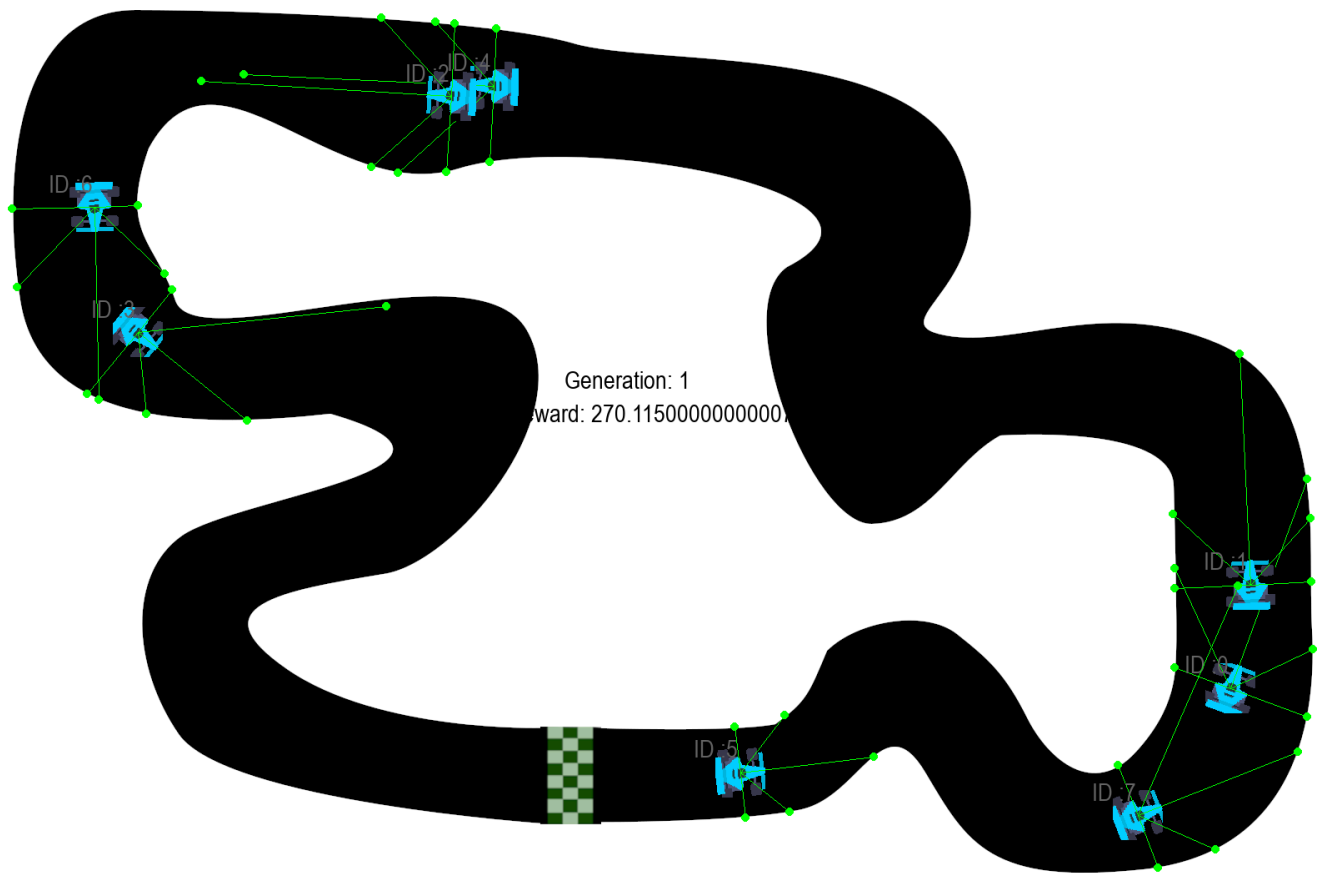


# Rapport de Projet IA

Conduite Autonome 2D



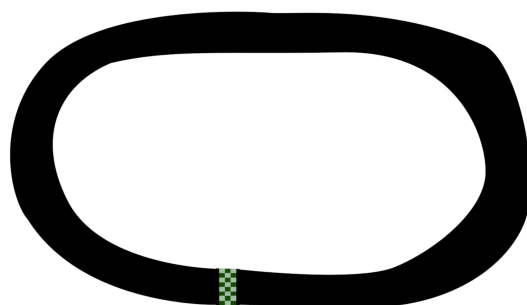
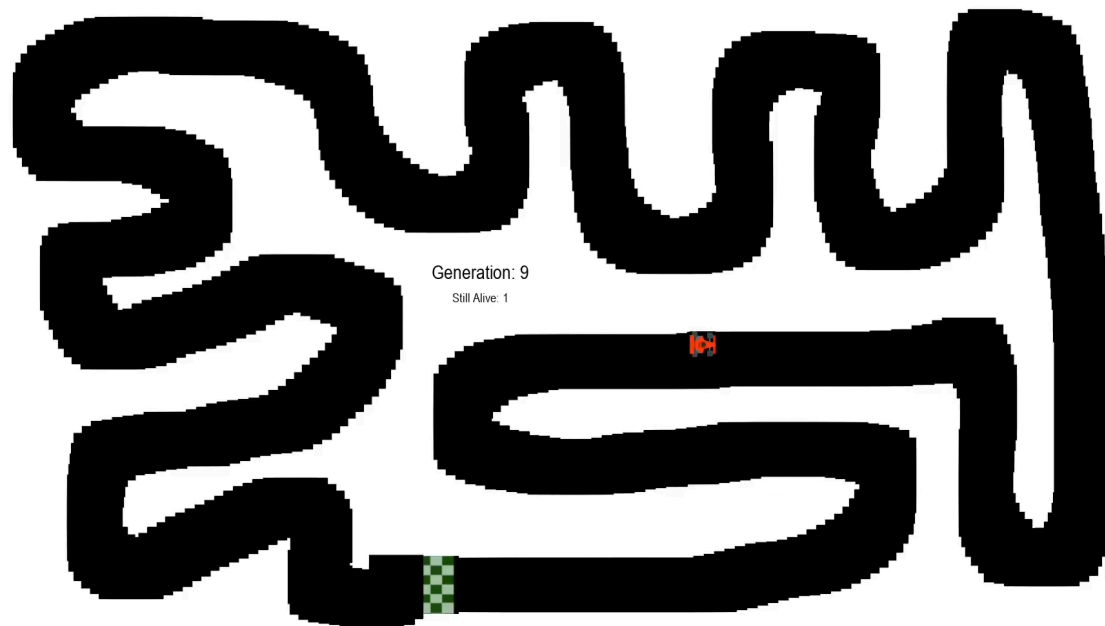
CARDOT Clément  
GUILBAUD Maxime  
GUAIS Clément

# Tables des Matières

<b>Tables des Matières</b>	<b>1</b>
<b>1. Explication et formalisation du problème</b>	<b>2</b>
<b>2. Description des données</b>	<b>3</b>
<b>3. Travail Effectué</b>	<b>4</b>
3.1. Analyse de l'existant	4
3.2. Algorithme NEAT	4
3.3. Migration vers Stable Baselines	5
3.3.1. Algorithmes	6
3.3.1.1 A2C	6
3.3.1.1 PPO	7
3.4. Récompenses	8
3.4.1 Secteurs	8
3.4.2 Optimisation des Récompenses	8
3.5. Le multiprocessing	9
<b>4. Résultats Obtenus</b>	<b>11</b>
4.1. Algorithme NEAT	11
4.2. Algorithme A2C	11
<b>5. Discussion et ouverture</b>	<b>14</b>
5.1. Entraînement multi-circuit	14
5.2. Radars	14
5.3. Environnement 3D	14
5.3.1 CARLA Simulator	14
5.3.2 Donkey Car	15
<b>6. Références:</b>	<b>16</b>

# 1. Explication et formalisation du problème

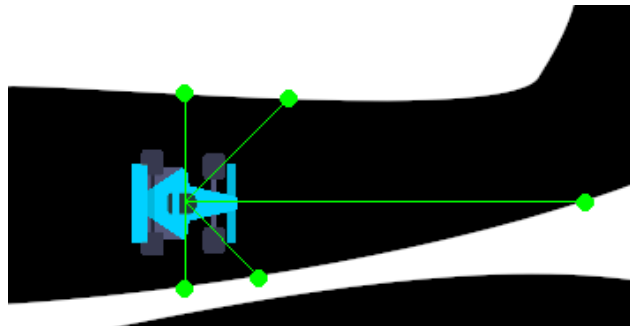
L'objectif du projet est d'apprendre un circuit à une voiture afin qu'elle puisse faire des tours d'un circuit en autonomie. Par la suite, on pourra tester le modèle sur d'autres circuits plus ou moins similaires afin de tester la robustesse de l'apprentissage.



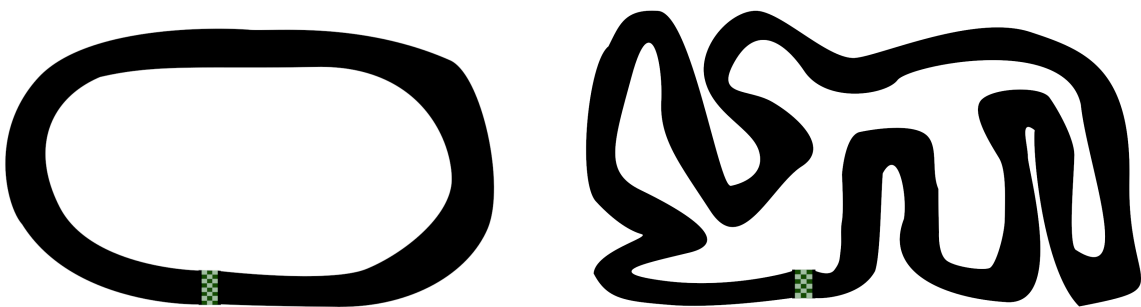
## 2. Description des données

Dans notre cas, les données d'entrées sont celles fournies par les capteurs présents sur la voiture. Il y a en tout **5 capteurs renvoyant des informations sur la distance à la bordure**. A chaque rafraichissement, les capteurs calcul la distance entre la voiture et le mur le plus proche (collision avec un pixel blanc), si la distance est supérieur à 300 pixels, la distance retenu est 300 pixels.

Une fois cette distance calculée, afin de **normaliser les données entre 0 et 1**<sup>1</sup> le capteur va donc diviser la valeur mesurer par la valeur maximum : 300 pixels. Ainsi chaque capteur génère un variable allant de 0 à 1 correspondant à la distance du mur le plus proche.



On remarque tout de suite que la qualité des données transmises par les capteurs sont en grande partie dépendantes de la carte choisie pour l'apprentissage. Le choix de la carte est donc important pour avoir un bon apprentissage, celle-ci pouvant être plus ou moins complexe.



**normaliser les données entre 0 et 1**<sup>1</sup>: Il s'agit d'une bonne pratique lors d'un apprentissage par renforcement. En effet, il a été démontré que les IA ont de meilleures performances lorsque les données sont normées entre 0 et 1.

Par la suite on utilise des algorithmes de renforcement et à partir des données d'entrées les **données de sorties** seront la **l'accélération** (ou freinage de la voiture) ainsi que sa **direction**.

## 3. Travail Effectué

### 3.1. Analyse de l'existant

Afin de nous lancer dans ce projet, il nous a fallu dans un premier temps **définir un environnement d'entraînement** : une voiture, un circuit, des capteurs... Il s'agit d'un travail qui peut vite demander beaucoup de temps. Etant donné que notre temps lors de ce projet est limité et qu'il serait préférable pour nous de nous concentrer sur l'IA, nous avons donc décidé de **reprendre un environnement déjà existant**.

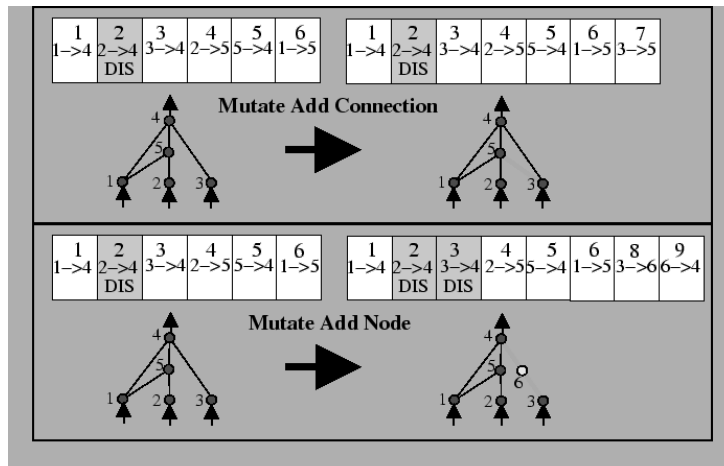
Il existe de nombreux projets sur Youtube et GitHub qui implémentent de l'apprentissage par renforcement sur des voitures avec un circuit en 2D. Après en avoir parcouru un certain nombre, nous avons sélectionné celui du compte Youtube et GitHub "NeuralNine". Son projet présente de nombreux avantages : **plusieurs circuits disponibles, système de capteur simple, algorithme NEAT...**

Dans la suite de ce rapport, vous allez voir que le projet initial a été grandement modifié, mais le choix d'avoir commencé avec cette base nous aura permis de pousser beaucoup plus loin notre objectif initial.

### 3.2. Algorithme NEAT

Dans un premier temps, nous avons donc repris l'algorithme fourni afin de pouvoir l'entraîner et le tester sur les différents circuits. Un modèle a été créé pour chaque circuit et ils ont été testé sur tous les circuits.

L'algorithme **NEAT** (NeuroEvolution of Augmenting Topologies) est un algorithme évolutif utilisé pour évoluer des réseaux de neurones artificiels. Contrairement aux approches d'optimisation classiques qui ajustent simplement les poids des connexions dans un réseau de neurones fixe, NEAT est capable d'**évoluer la structure même du réseau**, y compris **le nombre et la configuration des neurones ainsi que les connexions entre eux**. NEAT utilise les étapes d'un algorithme génétiques, c'est-à-dire qu'il va utiliser les étapes de **sélection, reproduction et mutation** afin d'avoir une diversité génétique. De plus, NEAT utilise un mécanisme de **complexité graduelle des structures neuronales** au fil des générations. Les innovations qui améliorent les performances ont plus de chances de survivre et de se propager dans la population.



Les performances sont évaluées par une fonction de reward. Celle-ci est déterminée par la distance parcourue de la voiture mais cela pose des problèmes abordés plus dans la partie sur les récompenses.

Pour la phase d'entraînement :

- Le paramétrage s'est fait par itération. **La population** ne devait pas être trop petite pour ne pas gêner l'apprentissage. Elle a été fixée à 100 afin de permettre un apprentissage convenable. **Le nombre de génération** quant à lui était ajusté afin que le circuit soit fini par au moins 1 voiture.
- Le nombre de génération dépend principalement de la complexité de la carte. Ci-dessous on retrouve le nombre de génération minimum et le temps d'apprentissage afin de compléter un tour avec au moins 1 voiture avec une population de 100 voitures
- A la fin de l'apprentissage, le meilleur génome (celui avec le plus grand reward) était sauvegardé en tant que modèle de la carte.

Pour la phase de test :

- Utilisation d'un **programme paramétré** avec comme argument la carte jouée et le modèle à utiliser. On peut de cette façon tester chaque modèle sur toutes les cartes.

### 3.3. Migration vers Stable Baselines

Afin de challenger ce modèle nous avons décidé de migrer vers stable-baselines. Stable-Baselines est une bibliothèque open-source en Python conçue pour l'entraînement, l'évaluation et le déploiement d'algorithmes d'apprentissage par renforcement.

Nous avons choisi cette bibliothèque car elle dispose d'une documentation très complète et c'est une des bibliothèque de machine learning les plus facile à utiliser. De plus, elle dispose de différents algorithmes qui peuvent être interchangé très facilement afin de comparer leurs performances.

### 3.3.1. Algorithmes

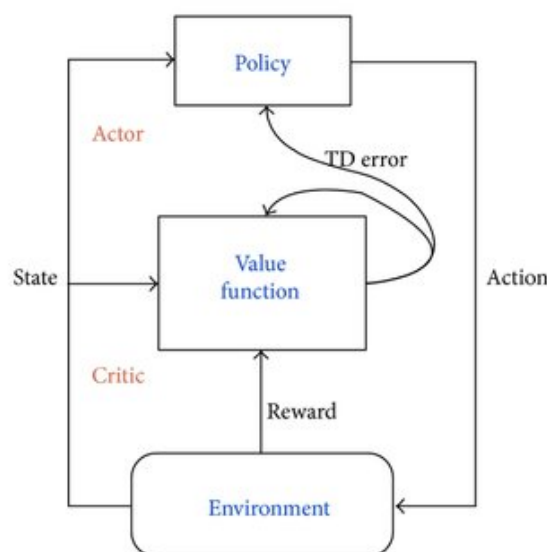
La bibliothèque stable baselines permet l'utilisation de différents algorithmes nous permettant de pouvoir les comparer pour utiliser le plus performant.

#### 3.3.1.1 A2C

L'algorithme **A2C** (Advantage Actor-Critic) est un algorithme de renforcement qui utilise **des acteurs**. L'acteur est responsable de prendre des actions dans l'environnement. Il prend en entrée l'état actuel de l'environnement et génère une distribution de probabilités sur les actions possibles. Cela se traduit par les actions de vitesse et rotation sur la voiture.

Il y a ensuite **la critique**, une phase d'évaluation de la qualité des actions prises par l'acteur. Cette évaluation permet de calculer la valeur d'un état, qui représente l'expected return à partir de cet état. Plus la voiture parcourt une grande distance, plus cet état aura une valeur élevée.

**L'avantage** va par la suite modifier la politique des acteurs et de la critique afin de maximiser la valeur de l'État.



Pour la phase d'entraînement :

- On utilise **8 environnements en parallèle** afin de maximiser l'utilisation du CPU.
- 
- on laisse le temps d'entraînement minimum afin qu'au moins une voiture finisse un tour.
- Le modèle à la fin est sauvegardé afin d'être testé sur toutes les cartes par la suite.

Pour la phase de test :

- On teste le modèle sur les cartes et on note si la voiture arrive à faire un tour.

### 3.3.1.1 PPO

L'algorithme **PPO** (Proximal Policy Optimization) est très similaire à A2C dans son fonctionnement. Il introduit une mesure de proximité entre les nouvelles et anciennes politiques pour éviter des mises à jour trop drastiques, ce qui contribue à la stabilité de l'apprentissage.

Dans un premier temps l'agent interagit avec l'environnement ce qui influence la récompense calculée.

Ensuite l'algorithme effectue des mises à jour sur la politique des agents afin de maximiser une fonction objectif clippée. Cette fonction utilise le ratio de probabilité entre les anciennes et nouvelles politiques afin de limiter le changement par rapport à la politique précédente.

La fonction objectif clippée :

$$L(\theta) = E[\min(r_t(\theta) \cdot A_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon) \cdot A_t)]$$

où :

- $r_t(\theta)$  est le ratio de probabilité entre les nouvelles et anciennes politiques.
- $A_t$  est l'avantage de l'action à l'instant  $t$ .
- $\epsilon$  est un hyperparamètre de clip, généralement un petit nombre.

Sa principale différence avec l'algorithme A2C est la mise à jour de sa politique. A2C utilise la méthode du gradient ascendant standard, tandis que PPO introduit une forme de proximité avec **une mise à jour en lots pour stabiliser l'apprentissage**. Cela rend l'algorithme PPO plus stable mais plus lent à l'apprentissage.

Ensuite, que ce soit pour la phase d'apprentissage ou de test, cela reste similaire aux conditions utilisées pour l'algorithme A2C.



## 3.4. Récompenses

### 3.4.1 Secteurs

Pour assurer la cohérence de la navigation, nous avons utilisé les **capteurs** déjà existant afin de détecter des couleurs différentes sur le circuit et ainsi mettre en place un système de secteurs. Ce dernier permet au véhicule de mémoriser le secteur dans lequel il se trouve. En cas d'atteinte d'un secteur non anticipé, des mécanismes de **récompenses** ou de **pénalités** sont activés pour orienter le véhicule dans la bonne direction.

Exemples d'application :

- Si la voiture circule dans le sens inverse, le système de secteurs déclenche des **pénalités** en conséquence.
- Si la voiture semble tourner en rond, elle recevra **moins de récompenses** que ses congénères.

### 3.4.2 Optimisation des Récompenses

Initialement, la **distance** parcourue constituait une métrique pertinente pour évaluer les performances du véhicule. Cependant, nous avons entrepris d'améliorer le processus. Dans cette optique, nous avons exploré l'idée de baser les **récompense** sur le **temps**, mettant ainsi l'accent sur la survivabilité du véhicule. Cependant, lors des premiers essais, ce choix a entraîné un comportement indésirable où le véhicule tendait à effectuer des rotations sur lui-même, minimisant ainsi les risques.

Dans l'objectif d'obtenir l'effet inverse, nous avons cherché à équilibrer les récompenses et à introduire des **pénalités temporelles**, avec l'intention de favoriser la vitesse du véhicule. Cependant, cette approche s'est avérée contre-productive, car le véhicule avait tendance à se déplacer le plus rapidement possible dans un mur, soulignant ainsi la nécessité d'une optimisation plus précise.

Pour une évaluation plus fine des performances, nous avons finalement intégré la **vitesse moyenne** dans le processus de récompense. Cette modification a permis une évaluation plus nuancée, en favorisant une conduite rapide et efficace tout en évitant des comportements indésirables tels que des rotations excessives ou des collisions contre les parois.

Ces ajustements ont été essentiels pour accroître la robustesse de l'IA face à des scénarios variés, soulignant l'importance cruciale d'une calibration précise des récompenses pour obtenir des comportements de conduite optimaux.

Amélioration des performances observées pour réaliser 1 tour (A2C) :

- **Carte 1** : De 1 minute à 26 secondes (division par ~2)
- **Carte 2** : De 2 minutes 38 secondes à 1 minute 3 secondes (division par ~2.5)
- **Carte 3** : De 7 minutes 12 secondes à 2 minutes 35 secondes (division par ~2.8)
- **Carte 4** : De 3 minutes 6 secondes à 1 minute 31 secondes (division par ~2)

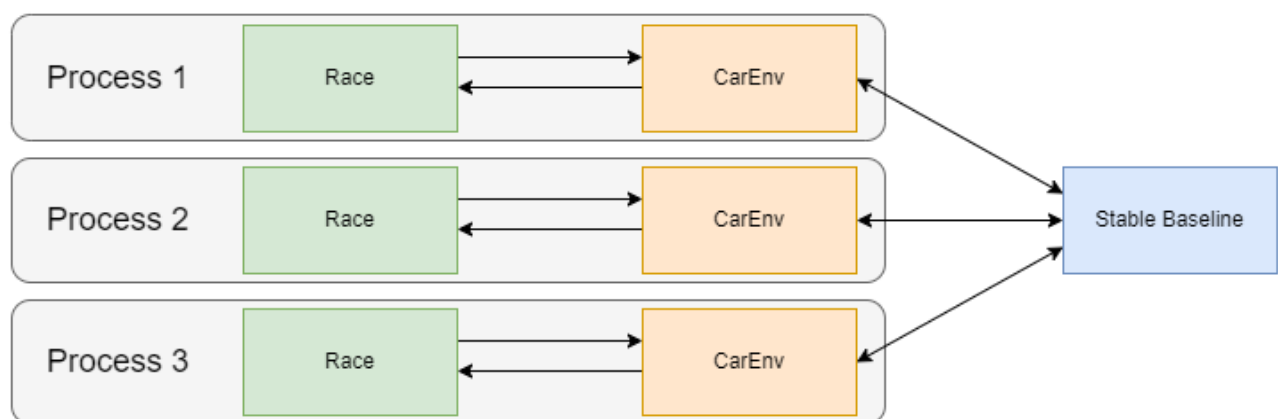
### 3.5. Le multiprocessing

L'algorithme de base avec NEAT permettait la simulation de plusieurs voitures en simultan  afin de cr er des vagues comparable   des g n ration du mod le. Mais tout cela en **mono-processing** !

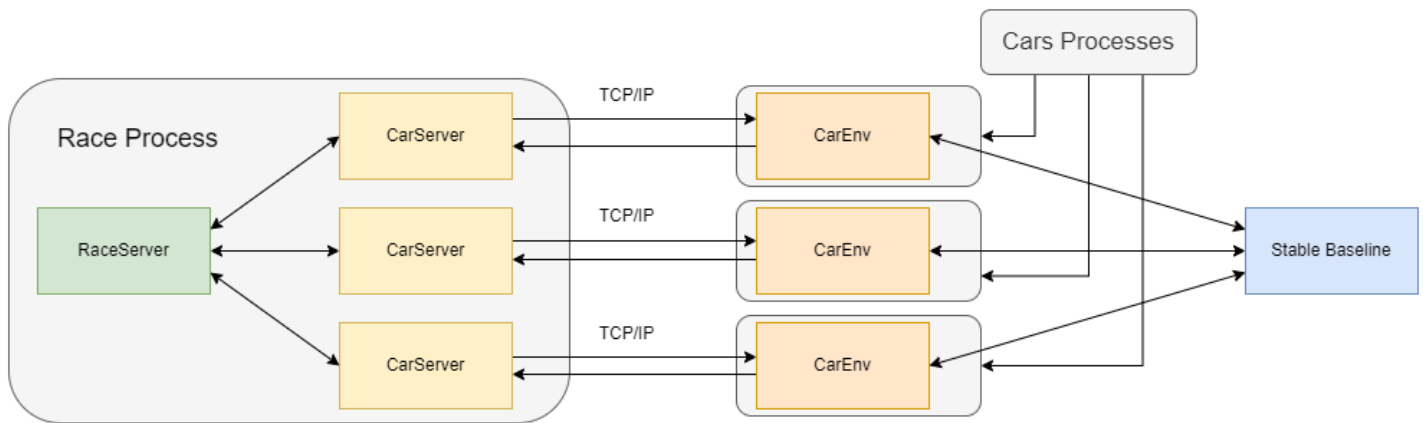
Dans un objectif d'am lioration des performances lors de l'apprentissage, nous avons voulu mettre en place un apprentissage **multi-processus** afin d'utiliser la pleine puissance du CPU.

Avec stable-baselines, il a fallu changer beaucoup de choses sur l'architecture de notre syst me. En effet, la plupart des framework de machine learning d finissent un environnement sur lequel **UNE instance** du mod le s'entra ne. Si on veut acc l rer l'apprentissage en utilisant plusieurs c urs du CPU, le framework va g n rer autant d'instances de l'environnement que du mod le. Dans notre cas, ce n'est pas souhaitable car l'environnement est relativement lourd et nous pr f rerions visualiser toutes les voitures sur le m me circuit.

Il a donc fallu trouver une alternative. une solution qui nous est tout de suite venu   l'esprit a  t  de transformer le syst me actuel mono-process en un syst me multi-process. Mais pour partager la m me instance du circuit cela se complique. Puisque les voitures sont chacune sur un processus ind pendant, l'appel vers le Singleton de la classe du circuit cr e   chaque fois une nouvelle instance du circuit. On se retrouve donc avec **autant de circuit que de voiture...** Pour que plusieurs processus partagent le m me circuit, il a donc fallu trouver une autre solution.



Nous avons donc décidé de transformer notre circuit en un **serveur TCP/IP** et les instances de nos modèles en Client. De ce fait, lors du lancement de notre script, nous définissons un serveur avec le nombre de clients attendu, puis nous lançons l'apprentissage du modèle avec le même nombre de clients.



L'implémentation du multiprocessing permet un gain de temps lors de l'apprentissage assez conséquent, en effet cela permet à l'IA de s'entraîner avec beaucoup plus d'itérations par secondes. Le gain de performance peut s'estimer à une multiplication par le nombre de cœurs CPU utilisés.

Exemple avec la durée de l'apprentissage:

*1 Core = 5 mins*

*2 Core = 2 mins 30 secs*

*4 Core = 1 mins 15 secs*

*8 Core = 37.5 secs*

PS : Ceci est une estimation théorique car dans les faits l'apprentissage nécessite quand même un minimum de temps afin de parcourir correctement l'entièreté du circuit.

## 4. Résultats Obtenus

### 4.1. Algorithme NEAT

Avec l'algorithme NEAT le temps d'apprentissage correspond au temps mis pour qu'au moins 1 voiture finisse un tour.

	Carte 1	Carte 2	Carte 3	Carte 4
Nombre de Génération	2	5	40	9
Temps d'apprentissage	~5sec	~10sec	4min 50sec	~44sec

Les conclusions des tests sont : **plus la carte est complexe avec des virages diversifiés** pour la phase d'apprentissage **plus le modèle est adaptable sur les autres cartes**. Un exemple de cela est le modèle entraîné par la carte 1. Ce modèle entraîné sur un circuit où l'on ne tourne qu'à gauche ne sait pas tourner à droite quand cela s'impose. Un second exemple est le modèle entraîné par le circuit 4. Ce modèle n'est entraîné que pour un type de virage (virage à 90 degrés) ce qui le rend peu robuste lorsque que les virages sont plus larges ou plus courts. Ci-dessous se trouve un tableau montrant pour chaque modèle si la voiture a réussi à finir un tour sur toutes les cartes :

	Carte 1	Carte 2	Carte 3	Carte 4
Modèle carte 1				
Modèle carte 2				
Modèle carte 3				
Modèle carte 4				

### 4.2. Algorithme A2C

Avec l'algorithme A2C le temps d'apprentissage correspond au temps mis pour qu'au moins 1 voiture finisse un tour.

	Carte 1	Carte 2	Carte 3	Carte4
Temps d'apprentissage	~25sec	1min25	5min25	2min04

Ensuite lorsque l'on teste les modèles entraînés sur les cartes pendant 30 minutes, la voiture n'arrive pas à finir un tour sur les cartes les plus complexes malgré le fait qu'elle pouvait faire plusieurs tours pendant l'entraînement. On peut expliquer cela par une exploration insuffisante rendant le modèle non adapté au changement d'environnement. Ci-dessous se trouve le résultat des modèles entraînés sur chaque carte pendant 30 minutes environ :

	Carte 1	Carte 2	Carte 3	Carte 4
Modèle carte 1				
Modèle carte 2				
Modèle carte 3				
Modèle carte 4				

### 4.3. Algorithme PPO

Avec l'algorithme PPO le temps d'apprentissage correspond au temps mis pour qu'au moins 1 voiture finisse un tour. On remarque que le temps minimum pour qu'une voiture finisse un tour augmente rapidement en fonction de la complexité de la carte.

	Carte 1	Carte 2	Carte 3	Carte4
Temps d'apprentissage	3 min 34 sec	25 min 10 sec	plus d'1H	/

### 4.4. Comparaison

L'algorithme le plus performant en rapidité d'apprentissage et robustesse parmi les trois utilisés est donc l'algorithme NEAT. Même si pendant l'apprentissage l'algorithme A2C peut finir un tour plus vite que l'algorithme NEAT, le modèle doit avoir un temps d'entraînement nettement supérieur pour que celui-ci soit utilisé sur les différentes cartes. Enfin il faut un temps d'entraînement de plusieurs heures minimum afin que le l'algorithme PPO puisse fournir un résultat mais il se peut qu'il soit plus robuste que les deux autres méthodes au vu de la stabilité de l'apprentissage.

## 5. Discussion et ouverture

### 5.1. Entraînement multi-circuit

Nous avons vu que le circuit sur lequel le modèle s'entraîne impacte fortement les performances du modèle sur les autres circuits. En effet, le modèle généré se retrouve vite en surapprentissage afin de coller au mieux au circuit sur lequel il s'entraîne. Afin d'éviter ce biais, nous pourrions entraîner le modèle sur plusieurs circuits en simultanée. Chaque voiture serait connectée à un circuit pour nourrir l'algorithme avec des données plus variées.

Un entraînement de ce type devrait en théorie permettre au modèle d'être très polyvalent par la suite.

### 5.2. Radars

Dans notre simulation, chaque voiture possède cinq radars. Ces radars sont comparables à ceux que l'on retrouve sur les radars de recul des voitures actuelles. Pour de la conduite autonome cependant, les véhicules d'aujourd'hui sont plus souvent équipés de capteurs LIDAR et/ou de caméras. Cela permet une bien meilleure anticipation de la trajectoire et détection des obstacles. Un axe d'amélioration pourrait donc être d'améliorer notre système de radar dans la simulation. A noter que ce genre de capteur engendrerait une augmentation drastique des données à traiter par l'IA. Il faudrait donc sans doute effectuer un traitement des données de radars en amont afin de réduire la charge de l'IA.

### 5.3. Environnement 3D

Notre simulateur nous a permis de générer un modèle capable de se déplacer dans un environnement en 2 dimensions. Afin de se rapprocher d'un environnement réaliste, l'étape suivante serait donc de passer à 3 dimensions. Bien que les voitures ne volent pas et se déplacent donc techniquement sur 2 dimensions, la prise d'information des capteurs elle, doit se faire sur 3 dimensions. Il existe aujourd'hui des simulateurs permettant de facilement obtenir ce genre d'environnement.

#### 5.3.1 CARLA Simulator



CARLA est un projet Open Source sponsorisé notamment par Intel et Toyota. Cet outil permet d'obtenir un environnement 3D afin d'entraîner des modèles d'IA à la conduite de véhicule. L'environnement est généré grâce à l'Unreal Engine ce qui permet d'obtenir des images et une physique extrêmement réalistes.

### 5.3.2 Donkey Car

Donkey Car est un autre projet Open Source qui s'apparente plutôt à un framework permettant d'entraîner des modèles capables de conduire des voitures miniatures. Ces voitures sont généralement composées d'un RaspBerry ou d'un NVidia Jetson nano qui permet d'embarquer le modèle directement dans la voiture. Cependant, afin de rendre accessible le projet au plus grand nombre, la communauté a créé un simulateur nommé DonkeySim. Celui-ci est comparable à CARLA, cependant il tourne avec le moteur Unity et est beaucoup moins réaliste en termes de physique et de graphismes.



## 6. Références:

- Vidéo Youtube de la chaîne “NeuralNine”  
 Self-Driving AI Car Simulation in Python
- Code source de NeuralNine  
<https://github.com/NeuralNine/ai-car-simulation>
- Vidéo Youtube de la chaîne “Dave’s Space” expliquant clairement la différence entre Thread et MultiProcessing  
 threading vs multiprocessing in python
- Stable-Baselines ([documentation](#))
- DonkeyCar  
<https://www.donkeycar.com>
- CARLA Sim  
<http://carla.org/>