



UNIVERSITÉ
CAEN
NORMANDIE

Université de Caen Normandie
UFR des Sciences
Département Informatique

1ère année de master informatique

Rapport : Projet Annuel

Thomas Vasse, Sacha Pronost, Noah Berneaud, Thomas Neveu

Mai 2023

Encadrant : Alexis Lechervy
Jury : Alexis Lechervy & Patrick Lacharme

Année universitaire : 2022 / 2023
Soutenu le 17 mai 2023

Table des matières

| | | |
|----------|---|-----------|
| 1 | Introduction | 3 |
| 2 | Datasets | 5 |
| 2.1 | Dataset personnelle | 5 |
| 2.2 | Nouveau dataset | 6 |
| 3 | Scenarios : | 8 |
| 3.1 | Srunner | 8 |
| 3.2 | Scenario automatique | 10 |
| 3.3 | Scenarios prédéfini | 11 |
| 3.3.1 | Scénario 1 : | 12 |
| 3.3.2 | Scénario 2 : | 12 |
| 3.3.3 | Scénario 3 : | 12 |
| 3.3.4 | Scénario 4 : | 13 |
| 3.3.5 | Scénario 5 : | 13 |
| 3.3.6 | Scénario 6 : | 14 |
| 4 | Modèle de détection sémantique | 16 |
| 4.1 | Logiciel de test | 16 |
| 4.2 | Rappel | 17 |
| 4.3 | Création d'un environnement Anaconda | 17 |
| 4.4 | Implémentation de la détection sémantique/ Avancement | 18 |
| 4.4.1 | Donnée d'apprentissage | 18 |
| 4.4.2 | Modèle | 19 |
| 4.4.3 | Apprentissage | 20 |
| 4.4.4 | Évaluation | 20 |
| 5 | Serveur | 22 |
| 5.1 | Etat de l'art | 22 |
| 5.2 | Refonte | 23 |
| 5.2.1 | Déplacement d'images | 23 |
| 5.2.2 | Interface web | 23 |
| 6 | Conclusion | 26 |

1 Introduction

Le développement de la navigation autonome a connu des avancées significatives grâce aux techniques d'intelligence artificielle, en particulier l'apprentissage profond. Dans notre précédent rapport, nous avons entrepris la création d'un modèle d'apprentissage profond pour la segmentation sémantique, c'est-à-dire la détection des différents éléments environnants d'un véhicule. Nos expérimentations ont été réalisées en utilisant le simulateur CARLA, et nous avons fourni une description détaillée de notre approche dans le rapport correspondant.

Pour rappel au cours du premier semestre, nous avions réalisé les travaux suivants :

- La prise en main du simulateur CARLA (API du simulateur, Capteurs de conduite disponible, problèmes rencontrés,...)
- Utilisé et expliqué la détection sémantique avec Lightning Flash (Définition, Bibliothèque Lightning Flash, Installation, Implémentation)
- Utilisation d'un Dataset déjà existant
- Le commencement de la création du partie Client - Serveur (Introduction de Flask et son fonctionnement)

Au cours de ce second semestre, nous avons poursuivi nos travaux en élargissant notre champ d'étude. Tout d'abord, nous nous sommes concentrés sur la création de notre propre jeu de données spécifique pour l'entraînement du modèle. Nous avons collecté des images provenant des capteurs du véhicule, notamment grâce aux caméras implémentées directement dans le simulateur CARLA. Nous avons ensuite créé des scénarios complexes dans le simulateur CARLA, afin de reproduire des situations de conduite réalistes et variées.

Dans ce rapport, nous décrirons les différents scénarios mis en place et expliquerons en détail la méthode de détection sémantique que nous avons utilisée. Nous fournirons également une analyse des résultats obtenus, en comparant les prédictions de navigation autonome de notre modèle avec celles obtenues automatiquement à l'aide d'une caméra fournie par le simulateur CARLA.

En résumé, ce rapport présentera les résultats et les avancées obtenus au cours du second semestre de notre projet. Nous mettrons l'accent sur la constitution de notre propre jeu de données, la création de scénarios plus complexes, l'approfondissement de la détection sémantique, ainsi que les développements réalisés dans la partie client-serveur de notre

système. Les détails de chaque aspect seront exposés dans les sections suivantes pour une compréhension approfondie de notre travail et de ses résultats.

2 Datasets

Durant le 1er semestre, nous avons utilisé un dataset déjà existant afin d'entraîner et de tester notre modèle de détection sémantique. Celui-ci marchait convenablement, mais nous avons pris la décision de faire notre propre dataset. Nous avons donc enregistré de nombreuses images venant de notre simulateur.

2.1 Dataset personnelle

Afin d'obtenir notre dataset, nous avons utilisés :

- 8 météos différentes pour que notre modèle soit efficace dans différentes situations
- 2 caméras :
 - RGB afin de servir d'image d'exemple
 - Sémantique afin de servir d'image labellisée

Le dataset est créé selon le pseudo-code suivant : Nous obtenons donc un ensemble de 8

Algorithm 1 Crédation d'un dataset

Générer et placer un ensemble de voitures sur la carte

Mettre l'auto-pilote aux voitures

$V1 \leftarrow$ notre véhicule

Générer et placer $V1$ sur la carte

$CamRGB \leftarrow$ caméra RGB

$CamSeg \leftarrow$ caméra Sémantique

$V1 \leftarrow V1 + CamRGB \cup CamSeg$

Mettre l'auto-pilote de $V1$

$Weathers \leftarrow [ensemble de météo aléatoire]$

while $Weathers \neq []$ **do**

$W \leftarrow Weathers[0]$

$Weathers \leftarrow Weathers - W$

 Changer la météo de la carte par W

while 60 secondes **do**

 Enregistrer une image depuis $CamRGB$ dans le dossier " $W + '/RGB/'$ "

 Enregistrer une image depuis $CamSeg$ dans le dossier " $W + '/Seg/'$ "

 Attendre 2 secondes pour ne pas avoir les mêmes images plusieurs fois

| | | | |
|-------------------|---|------------------|---------------------|
| 📁 All_Image | ✖ | 01/02/2023 14:33 | Dossier de fichiers |
| 📁 ClearNoon | ✖ | 01/02/2023 14:32 | Dossier de fichiers |
| 📁 ClearSunset | ✖ | 01/02/2023 14:32 | Dossier de fichiers |
| 📁 HardRainSunset | ✖ | 01/02/2023 14:32 | Dossier de fichiers |
| 📁 MidRainSunset | ✖ | 01/02/2023 14:32 | Dossier de fichiers |
| 📁 SoftRainNoon | ✖ | 01/02/2023 14:32 | Dossier de fichiers |
| 📁 SoftRainSunset | ✖ | 01/02/2023 14:33 | Dossier de fichiers |
| 📁 WetCloudySunset | ✖ | 01/02/2023 14:33 | Dossier de fichiers |
| 📁 WetSunset | ✖ | 01/02/2023 14:33 | Dossier de fichiers |

FIGURE 2.1 – Fichier du dataset

| | | | |
|----------------|---|------------------|---------------------|
| 📁 rgb | ✖ | 15/02/2023 14:26 | Dossier de fichiers |
| 📁 segmentation | ✖ | 15/02/2023 14:26 | Dossier de fichiers |

FIGURE 2.2 – Fichier du dataset dans chaque dossier ci-dessus

fichiers +1 regroupant toutes les images RGB et Sémantique sans distinction des météos. Nous obtenons un total d'environ 430 images. Ceci reste assez faible pour un dataset, mais nous n'avons pas les ordinateurs très puissants pour augmenter la taille.

Résultat : Lorsque nous entraînons notre modèle sur le dataset, nous nous sommes rendu compte que le résultat n'était pas du tout celui escompté. En effet, le modèle prédisait des images qui n'avaient aucun sens.

La raison principale est l'auto-pilote de Carla, car notre voiture fonçait au bout de 3 secondes dans un mur, une voiture, un panneau... Nous obtenons donc un dataset qui contenait de nombreuses images inexploitables et le rendant inintéressant.

Les solutions qui auraient pu améliorer le dataset est :

- Faire un trajet spécifique à la voiture sans crainte de dysfonctionnement (très long à faire)
- Prendre le contrôle manuel de la voiture et conduire correctement (très coûteuse en ressource → images saccadé → très dur de contrôler le véhicule)

2.2 Nouveau dataset

Afin d'améliorer notre modèle, en sachant qu'il nous été donc impossible de générer notre propre dataset, nous avons donc décidé de télécharger un dataset déjà existant sur internet, mais comptant beaucoup plus d'image que le précédent.

Le dataset que nous avons choisi se trouve sur le lien suivant :

<https://npm3d.fr/kitti-carla> [1]

Il contient un grand nombre d'image de caméra de voiture, ainsi que les images de la détection sémantique correspondante. Cependant, nous nous sommes rendu compte que les images de la détection sémantique ne correspondaient pas aux images que nous voulions.

En effet il s'agissait d'image crée pour être lisible pour l'homme :



FIGURE 2.3 – Image détection sémantique original

Afin de résoudre ce problème, nous nous sommes renseigné sur la documentation de carla afin d'obtenir la valeur de la classe correspondante aux valeurs **rgb** de l'image. A l'aide de cette documentation il nous a donc été possible de transformer toute les images pour en produire d'autre capables d'entraîner notre modèle. Pour ce faire nous avons donc créer un dictionnaire contenant pour chaque valeur des pixels de l'image la valeur de pixels correspondante à la classe :

```
color_dict = {(0, 0, 0, 255): (0, 0, 0, 255),
(70, 70, 70, 255): (1, 0, 0, 255),
(100, 40, 40, 255): (2, 0, 0, 255),
(55, 90, 80, 255): (3, 0, 0, 255),
(220, 20, 60, 255): (4, 0, 0, 255),
(153, 153, 153, 255): (5, 0, 0, 255),
(157, 234, 50, 255): (6, 0, 0, 255),
(128, 64, 128, 255): (7, 0, 0, 255),
(244, 35, 232, 255): (8, 0, 0, 255),
(107, 142, 35, 255): (9, 0, 0, 255),
(0, 0, 142, 255): (10, 0, 0, 255),
(182, 182, 156, 255): (11, 0, 0, 255),
(220, 220, 0, 255): (12, 0, 0, 255),
(70, 138, 180, 255): (13, 0, 0, 255),
(61, 0, 81, 255): (14, 0, 0, 255),
(150, 100, 100, 255): (15, 0, 0, 255),
(230, 150, 140, 255): (16, 0, 0, 255),
(180, 165, 180, 255): (17, 0, 0, 255),
(250, 170, 30, 255): (18, 0, 0, 255),
(110, 190, 160, 255): (19, 0, 0, 255),
(170, 120, 50, 255): (20, 0, 0, 255),
(45, 60, 150, 255): (21, 0, 0, 255),
(145, 170, 190, 255): (22, 0, 0, 255)}
```

FIGURE 2.4 – Dictionnaire valeur rgb

En appliquant ce dictionnaire sur chaque pixel de toutes les images, nous obtenons les labels que nous utiliserons par la suite.

3 Scenarios :

3.1 Srunner

SRunner est un composant du simulateur CARLA qui permet d'automatiser les scénarios de conduite et de tester les algorithmes de conduite autonomes. SRunner permet aux utilisateurs de spécifier des tâches à effectuer pour le véhicule autonome, telles que le suivi d'une trajectoire, la reconnaissance de panneaux de signalisation, l'évitement d'obstacles, etc.



FIGURE 3.1 – Suivi d'une trajectoire

Les utilisateurs peuvent créer des scénarios en définissant des tâches et en spécifiant des conditions de réussite pour ces tâches. Par exemple, un scénario pourrait être de faire parcourir une distance donnée à un véhicule autonome sans accident. SRunner exécute ensuite ces scénarios en utilisant des agents intelligents qui simulent les comportements des piétons et des autres véhicules dans l'environnement de simulation CARLA.

Supposons que nous voulons créer un scénario où un véhicule autonome doit changer de voie en toute sécurité pour éviter une collision avec un autre véhicule.

Nous pouvons utiliser SRunner pour créer ce scénario de la manière suivante :

1. Tout d'abord, nous créons un scénario en définissant les tâches pour le véhicule autonome, telles que "changer de voie vers la gauche" et "maintenir une distance

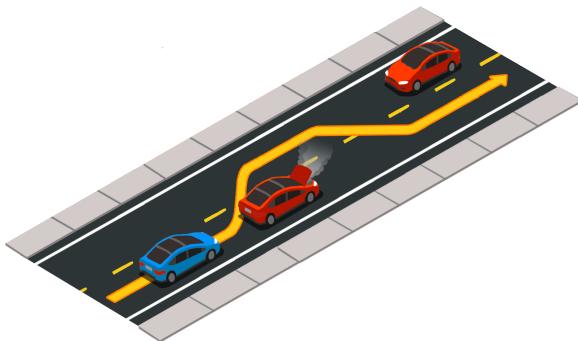


FIGURE 3.2 – Collision avec un autre véhicule

de sécurité par rapport à un autre véhicule”.

2. Ensuite, nous spécifions les conditions de réussite pour chaque tâche. Par exemple, pour la tâche ”changer de voie vers la gauche”, nous pouvons spécifier que la tâche est réussie si le véhicule autonome a changé de voie et que la distance entre le véhicule autonome et l’autre véhicule est supérieure à une certaine distance de sécurité.
3. Nous utilisons ensuite SRunner pour exécuter ce scénario dans CARLA Simulator.

En utilisant SRunner avec CARLA Simulator, nous pouvons tester et valider les algorithmes de conduite autonome dans un environnement simulé avant de les déployer sur la route. Cela permet de minimiser les risques pour la sécurité et d’optimiser les performances des algorithmes avant leur utilisation dans des conditions réelles.

En résumé, SRunner est un outil puissant pour tester les algorithmes de conduite autonome et les scénarios de conduite.

3.2 Scenario automatique

Suite à des problèmes d'installation non résolu de Srunner, nous avons du trouver une solution alternative. L'idée était donc de créer nous-même des scénarios sans l'aide de la librairie "Srunner". Notre but était de fixer deux points :

- Un point de départ
- Un point d'arriver

La voiture devait suivre un trajet qui lie les deux points de manière autonome.

La solution la plus évidente était d'appliquer l'auto-pilote de notre voiture jusqu'au point final. Malheureusement, nous ne sommes pas en capacité de réaliser ceci, car Carla n'implémente pas cette possibilité. En effet, l'auto-pilote ne possède pas d'intelligence artificielle. Il respecte simplement les règles de la route et choisi des directions aléatoires. De plus, nous ne pouvons pas influencer ses choix.

Une deuxième solution était la possibilité de faire nous-même notre trajet entre les deux points. Sur chaque route, il y a un ensemble de "waypoints". Chaque waypoint est accessible par une position absolue et nous avons la capacité d'observer ses voisins. Il est donc possible de construire un trajet grâce à un ensemble de points.

Algorithm 2 Crédit d'un trajet automatique

```
waypoint_deb ← waypoint le plus proche de la position de départ
waypoint_fin ← waypoint le plus proche de la position de fin
waypoints_trajet ← [ ]
waypoint_actuel ← waypoint_deb
while waypoint_actuel != waypoint_end do
    next_waypoints ← la liste de tout les waypoints voisins
    waypoint_actuel ← next_waypoints[0]
    waypoints_trajet ← waypoints_trajet + waypoints_actuel
```

Une fois la liste obtenue, nous pouvons placer la voiture sur le waypoint de début, puis appliquer une force en direction du prochain waypoint pour la faire avancer et faire ceci itérativement. Nous arrivons donc à la fin au waypoint final.

Ceci est théoriquement faisable, mais nous n'avons pas réussi à mettre en place cette méthode, car la fonction de calcul de trajet est extrêmement coûteuse puisqu'elle prend un ensemble de waypoints sans chercher à se rapprocher du waypoint final. Par conséquent, nous arrivons à une liste qui contient énormément de waypoint et qui semble être pas loin d'une boucle infinie. De plus, le trajet obtenu serait loin d'être le plus rapide, car nous n'avons pas de notion de plus court chemin dans la fonction.

Une dernière méthode serait de regarder le code source de la librairie "Srunner". En effet, il y a de grandes chances pour que la librairie se base sur un ensemble de méthodes déjà implémenté dans Carla. Nous pouvons donc voir les méthodes utilisées et leurs fonctionnements pour les adapter à notre problème. Cette méthode a été émise lors de notre dernière réunion de projet annuel et nous n'avons pas pu la tester par manque de temps.

3.3 Scenarios prédefini

Nous savons donc qu'il n'est pas possible de créer des scénarios de manière "intelligente" simplement en indiquant un point de départ et un point de fin.

Nous avons donc décidé de faire des scénarios sur mesure, c'est-à-dire des scénarios dans lesquels nous choisissons la carte, l'endroit, le nombre de voitures et leurs positions.

Nous avons établi 5 scénarios :

1. Un scénario dans lequel la voiture roule sur une route
2. Un scénario dans lequel la voiture effectue un dépassement
3. Un scénario dans lequel la voiture sort d'un parking
4. Un scénario dans lequel la voiture doit s'adapter à un feu tricolore
5. Un scénario dans lequel la voiture doit s'insérer sur une route

3.3.1 Scénario 1 :

But : Le but est simplement d'observer la conduite de la voiture sans situation particulière.

Nous plaçons notre voiture sur une ligne droite dans laquelle nous plaçons des voitures dans le sens opposé. De plus, nous ajoutons des piétons sur le trottoir pour se rapprocher des situations réels.

Résultat d'expérience : La voiture réagit normalement (ne fonce pas sur les piétons ou sur les voitures d'en face.).



FIGURE 3.3 – Scenario 1

3.3.2 Scénario 2 :

But : Le but est d'observer si la voiture fait un dépassement lorsque la situation se présente.

Nous possédons une double voie à sens unique. Sur la voie de droite, nous plaçons notre véhicule. Devant ce dernier, nous ajoutons 3 véhicules immobiles. La voie de gauche est utilisée pour faire un dépassement.

Résultat d'expérience : La voiture ne réagit pas comme souhaitée. Au lieu de faire un dépassement, la voiture s'arrête derrière les autres et attend.

3.3.3 Scénario 3 :

But : Le but est d'observer si la voiture est capable de sortir de manière autonome d'un parking.

Nous plaçons notre voiture dans un parking qui possède de nombreux véhicules à côté.



FIGURE 3.4 – Scenario 2

Résultat d'expérience : La voiture ne réagit pas comme souhaitée. Elle ne gère pas du tout les voies de parking et fonce direct dans un véhicule.



FIGURE 3.5 – Scenario 3

3.3.4 Scénario 4 :

But : Le but est d'observer si la voiture est capable de gérer des feux tricolore.

Nous plaçons notre voiture dans une voie qui possède un feu tricolore. Nous contrôlons les couleurs du feu.

Résultat d'expérience : La voiture réagit comme prévu.

- Si le feu est vert, alors la voiture continue sa route sans s'arrêter.
- Si le feu est orange, alors la voiture continue sa route
- Si le feu est rouge, alors la voiture s'arrête, est attend que le feu revienne au vert

3.3.5 Scénario 5 :

But : Le but est d'observer si la voiture est capable de s'insérer sur une autoroute à partir d'une voie d'insertion.

Nous positionnons notre véhicule sur la voie d'insertion. Nous ajoutons deux véhicules sur l'autoroute. Notre propre voiture se trouve à une distance telle des deux autres que, lorsqu'il est temps de s'insérer sur la route, nous devons nous arrêter, car les deux autres



FIGURE 3.6 – Scenario 4

voitures passent juste à ce moment-là.

Résultat d'expérience : La voiture réagit correctement. Au moment de l'insertion, la voiture s'arrête, laisse passer les deux véhicules et s'insère.



FIGURE 3.7 – Scenario 5

3.3.6 Scénario 6 :

But : Le but est d'observer si la voiture est capable de freiner au niveau d'un passage piéton.

Nous positionnons notre véhicule sur une voie proche d'un passage piéton. Nous ajoutons un piéton sur le trottoir et sur point de passer. Nous le faisons traverser et mettons la voiture en marche.

Résultat d'expérience : Le résultat est celui voulu. La voiture s'arrête au niveau du passage piéton.

Afin de réaliser tous ces scénarios, nous avons principalement utilisé ces fonctions :



FIGURE 3.8 – Scenario 6

— Générer un véhicule :

```
1     world.get_blueprint_library().find("vehicle")
```

— Générer un point de spawn :

```
1     carla.Transform(carla.Location(x, y, z))
```

— Placer le véhicule :

```
1     world.spawn_actor(blueprint, spawn_point)
```

— Activer l'auto-pilote du véhicule :

```
1     vehicle1.set_autopilot(True)
```

Un ensemble de fonctions offre la possibilité de placer la caméra à un endroit précis et d'obtenir ses coordonnées. Ceci est très utile pour le placement des véhicules dans l'environnement. Nous avons juste à placer notre caméra, obtenir ses coordonnées et placer un véhicule aux coordonnées.

```
1 camera = world.get_spectator()
2 camera_transform = camera.get_transform()
3 camera_location = camera_transform.location
4 camera_rotation = camera_transform.rotation
5 print(f"Camera location: {camera_location}")
6 print(f"Camera rotation: {camera_rotation}")
```

4 Modèle de détection sémantique

Cette partie du rapport sera consacrée aux explications ainsi qu'à la présentation des implémentations que nous avons effectuées pour la partie détection sémantique de notre projet, ainsi que tout ce qui englobe ce sujet.

4.1 Logiciel de test

Durant la première partie de notre projet, afin de visualiser l'efficacité de notre modèle nous affichions directement les images de sortie dans un fichier **Jupyter Notebook**. Cependant, nous avons rapidement constaté que cette méthode était redondante et peu pratique.

C'est pourquoi nous avons décidé de créer un logiciel en Python à l'aide de la bibliothèque **Tkinter** afin de simplifier cette tâche et de pouvoir tester nos modèles plus efficacement.

L'application fonctionne ainsi :

1. Dans un premier temps, il est possible de choisir un modèle parmi une liste de modèles présents dans notre projet.
2. Il est ensuite possible d'upload un certain nombre d'images que le logiciel va récupérer pour pouvoir ensuite être traité.
3. Finalement, en appuyant sur un bouton, le logiciel va appliquer le modèle sélectionné sur les images ajoutées, et renvoyer ces images dans un dossier correspondant.

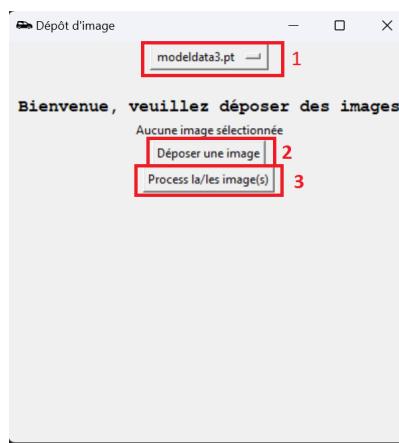


FIGURE 4.1 – Logiciel test modèle

Grâce à ce logiciel, nous avons pu vérifier visuellement la performance des modèles au cours de l'avancement du projet et comparer leurs différences de manière plus pratique et efficace.

4.2 Rappel

Comme expliqué dans le rapport précédent, l'un de nos principaux objectifs dans ce projet est d'implémenter un réseau de neurones pour la détection sémantique.

Le but de la détection sémantique est de détecter et d'identifier des éléments spécifiques dans une image, tels que des objets ou des concepts, et de les catégoriser en conséquence.

Dans notre cas, il s'agit de traiter des images de la vision d'une voiture afin de détecter et de reconnaître les différents éléments présents dans l'image, tels que les feux de signalisation, les panneaux de signalisation, les piétons et les autres véhicules.

Pour ce faire nous avons utilisé la bibliothèque **LIGHTNING FLASH**, une bibliothèque haut niveau basé sur **Pytorch** nous permettant ainsi de développer et de mettre en place rapidement des réseaux de neurones. [2]

Nous avions défini plus précisément ce qu'était la détection sémantique ainsi que les réseaux de neurones dans notre rapport précédent. Si vous souhaitez obtenir davantage d'informations, nous vous invitons à consulter la partie "**Sémantique**" de ce rapport.

4.3 Crédit d'un environnement Anaconda

Tout au long du projet, nous avons été confrontés à de nombreux problèmes de packages Python. En effet la bibliothèque **lightning flash** est régulièrement mise à jour, ce qui implique un certain nombre de nouveautés et de changement dans le code. De plus il nous est souvent arrivé de faire face à des conflits entre différents packages en fonction de leur version. Ces problèmes ont été un frein dans le développement du projet, c'est pourquoi nous avons décidé d'opter pour une solution qui nous permettrait de travailler tout du long de l'année sans nous soucier de ces problèmes.

L'un des membres de notre groupe a réussi à faire fonctionner tous les packages dont nous avions besoin simultanément. Nous avons donc décidé de **freeze** notre environnement Python dans un fichier puis de créer un environnement virtuel Anaconda contenant tous ses packages avec leur version. Cet environnement est ainsi figé et ne bougera pas pendant le projet.

Cette approche nous permet certes de garantir la pérennité de notre code, mais elle pose un autre problème : les versions des différentes bibliothèques restant figées, nous n'avons pas pu profiter des nouvelles implémentations qu'elles proposent. Cependant, nous avons choisi de faire ce compromis pour ce projet afin de garantir la stabilité de notre environnement de travail et éviter les problèmes liés aux conflits de versions.

4.4 Implémentation de la détection sémantique/ Avancement

Durant cette partie, nous allons vous expliquer les différentes modifications et tests que nous avons effectués pour arriver à la création de notre modèle de détection sémantique.

4.4.1 Donnée d'apprentissage

Dans le rapport précédent, nous vous avions expliqué que nous utilisions des images récupérées sur internet pour le dataset. Désormais nous utilisons de nouvelles images comme nous vous l'avons expliqué dans la partie **Datasets** précédemment. Nous nous sommes ainsi rendu compte de l'importance de la quantité de données pour entraîner un modèle d'apprentissage profond. Nous sommes ainsi passé de **1000** image à **10 000**.

Cependant, un aspect crucial que nous n'avions pas pris en compte lors du choix de notre dataset est la **diversité** des données. En effet, même si notre dataset est plus grand, s'il ne contient que des situations similaires, le modèle ne sera pas assez autonome et sera moins performant dans des situations spécifiques. Il est donc important de veiller à ce que le dataset soit diversifié pour que le modèle puisse apprendre à généraliser et à s'adapter à des situations variées.

Encore une fois, toutes ces données vont donc être chargées à l'aide de l'objet **SemanticSegmentationData.from_folders** de Lightning Flash. Cet objet contient donc le chemin pour les images d'entraînement ainsi que les images labelliser.

Nous avons constaté la grande facilité d'implémentation que nous offre la bibliothèque. En effet comme expliqué dans le rapport précédent, il est possible de spécifier le pourcentage de données à utiliser pour créer un jeu de validation en fournissant simplement cet argument à l'objet. Ainsi, nous n'avons pas besoin de séparer manuellement les données. De plus, cet objet prend également en compte le nombre de classes dans nos images étiquetées, ainsi que la taille du batch, ce qui nous permet de gagner du temps et de développer rapidement.

4.4.2 Modèle

Dans la première partie de notre projet nous avions utilisé le modèle pré-entraîné nommé **mobilenetv3_large_100** (réseau de neurones convolutionnel). Ce modèle est pratique car il peut être embarqué dans des "petites" machines comme des téléphones portables par exemple. Nous avions utilisé ce modèle car nous ne possédons pas de machine avec de grosses puissances de calcul, ainsi il nous étés possibles de faire apprendre ce modèle "rapidement".

Cependant le fait que ce modèle soit fait pour l'embarqué comporte un désavantage, il est moins puissant que d'autres modèles car il fonctionne avec peu de paramètres ("peu" dans le monde actuel des réseaux de neurones).

C'est pourquoi nous avons décidé d'utiliser un modèle pré-entraîné plus puissant. L'objet **SemanticSegmentation** de **LIGHTNING FLASH** propose de nombreux modèles, nous avons décidé d'utiliser le modèle **resnet50** pour les raisons suivantes :

- **Précision** : Il s'agit d'un modèle plus profond et plus complexe que MobileNetV3, ce qui lui permet d'avoir une meilleure précision pour des tâches de classification d'images plus complexes. Pour une application de conduite autonome comme la notre, il est donc primordial d'avoir un modèle plus précis comme celui-ci.
- **Quantité de données** : Comme expliqué précédemment nous utilisons désormais une quantité de données plus importantes que précédemment. Les modèles plus profonds comme ResNet50 ont besoin de plus de données pour être entraînés correctement, ce qui est moins le cas pour le modèle précédent.
- **Robustesse** : ResNet50 est plus apte à généraliser correctement sur des scènes diverses et variées que MobileNetV3, ce qui est très utile pour des applications comme la conduite autonome.

Cependant, l'utilisation d'un modèle plus puissant comme celui-ci pose également différent problème :

- **Coût de calcul** : Comme expliqué précédemment, nous ne possédons pas de machine puissante, cependant un modèle comme ResNet50 nécessitent plus de ressources pour être exécuté et entraîner impliquant donc un entraînement et un temps d'exécution plus long.
- **Sur-apprentissage** : Les modèles plus complexes peuvent être plus sujets au **sur-apprentissage**, c'est-à-dire qu'ils peuvent apprendre à reconnaître des motifs spécifiques dans les données d'entraînement qui ne sont pas généralisables à des données inconnues.

En prenant en compte tous ces éléments, et dans l'optique d'utiliser notre modèle pour de la conduite autonome, nous avons décidé d'opter pour un modèle plus puissant. Nous préférions prioriser la précision de notre modèle plutôt que sa rapidité, tout en sachant qu'il est possible d'obtenir une grosse puissance de calcul dans une voiture.

Nous utilisons la même tête de réseau que pour le précédent modèle (tête **fpn** pour "Feature Pyramid Network"), car elle est adaptée pour les tâches de segmentation d'ob-

jets de tailles diverses, pratique donc dans notre cas où l'image peut contenir de nombreuses informations de tailles différentes (voitures, piétons, arbres etc...).

Voici donc l'implémentation dans notre projet de ce modèle :

```
1 model = SemanticSegmentation(  
2     backbone="resnet50",  
3     head="fpn",  
4     num_classes=datamodule.num_classes, # Dans notre cas 23  
5     classes  
6 )
```

4.4.3 Apprentissage

De la même manière que précédemment, nous utilisons l'objet **flash.Trainer** afin d'entraîner notre modèle, et ce encore une fois avec 3 époques, et la strategy **freeze**.

Cependant, avec un plus grand nombre d'images et un réseau plus puissant, l'apprentissage s'est avéré beaucoup plus long sur nos machines personnelles. Par conséquent, nous avons été limités dans le nombre d'entraînements que nous pouvions effectuer, ce qui signifie que nous n'avons pas pu augmenter le nombre d'époques, ce qui rend le modèle moins puissant qu'il aurait pu l'être.

4.4.4 Évaluation

Afin de pouvoir quantifier la différence entre les différents modèles que nous testions, nous avons décidé de mettre en place une évaluation. Pour cela, avant de créer notre datamodule, nous avons séparé les données en donnée d'entraînement et donnée de test, et ce, à l'aide de la fonction *train_test_split* de la bibliothèque **sklearn** :

```
1 X = []  
2 for filename in os.listdir('data/'):   
3     X.append('data/images_rgb/'+filename)  
4 y = []  
5 for filename in os.listdir('data/'):   
6     y.append('data/image_ss/'+filename)  
7 X_train, X_test, y_train, y_test = train_test_split(X,  
8 y, test_size=0.2, random_state=42)
```

Pour l'évaluation, nous avons décidé d'utiliser la metric *Accuracy* de la bibliothèque **torchmetrics**, et ce, en précisant qu'il s'agit d'une labélisation multi-classes à 23 classes. Ainsi, en récupérant la prédiction du modèle, et la vérité terrain, nous pouvons calculer pour chaque image de notre jeu de test la précision du modèle :

```
1 accuracy = 0
2 accuracyMetric = torchmetrics.Accuracy(multiclass=True,
3 num_classes=23)
4 for i in range(len(X_test)):
5     prediction = torch.argmax(predictions[i][0]['preds'],
6     0).flatten()
7     true = moduleTest.test_dataloader().dataset[i]['target']
8     .flatten()
9
10    prediction = prediction.numpy().astype(int)
11    true = true.numpy().astype(int)
12    prediction = torch.from_numpy(prediction)
13    true = torch.from_numpy(true)
14    accuracy += accuracyMetric(prediction, true)
15 print(accuracy/len(X_test))
```

Finalement, pour notre nouveau modèle et avec le dataset que nous vous avons présenté dans la partie **Nouveau dataset** dans la page 6 de notre rapport, **90.48 %** des exemples de test sont correctement classifiés.

5 Serveur

5.1 Etat de l'art

Au premier semestre nous nous sommes heurté à la problématique suivante : comment relier le simulateur (carla) et le modèle de détection sémantique qui fondamentalement agit indépendamment dans leurs architectures respectives mais qui a besoin l'un de l'autre pour fonctionner ? Le simulateur a besoin de connaître les éléments de son environnement pour prendre des décisions sur la conduite du véhicule et le modèle de détection sémantique ne peut fonctionner sans image à lui injecter.

Nous nous sommes rendu compte que cette relation était tout bonnement une relation client-serveur avec dans notre cas :

- *le serveur* : Il sagit ici du modèle de détection sémantique. Il offre ses services au client
- *le client* : Il sagit du simulateur carla qui envoie au serveur des images à traiter et attend de lui des informations nécessaires à sa prise de décision.

Partant de ce constat, nous avons implémenté une structure web faisant office de passerelle entre le client et le serveur. Notre choix s'est porté sur la librairie **Flask** disponible depuis le gestionnaire de librairie python : **pip**.

l'Intérêt de Flask porte sur sa légèreté, ce qui cohérent pour notre utilisation où il s'agit exclusivement de trafic d'image de petite taille. De plus étant léger, le serveur peut tourner sur de petites configurations matérielles comme les nôtres.

Au premier semestre, une première version de l'architecture du serveur avait été effectuée. Cependant il ne s'agissait exclusivement que de la partie serveur. De plus ce dernier n'était pas fonctionnel, il s'agissait là que de tests.

5.2 Refonte

En repartant de l'architecture déjà effectuée. Il nous a fallu changer notre façon de concevoir l'architecture client-serveur.

En effet dans un premier temps nous pensions que le client envoyait une image au serveur, qui effectuait la détection sémantique et enfin faisait le renvoi de cette image auprès du client.

Le problème était que le simulateur (client) n'a pas besoin de recevoir les images segmentées. Cependant il a besoin de connaître des informations dans sa prise de décision.

Par exemple : "Freine, tourne à droite, accélère ..."

Nous ne pensions pas que le serveur avait également pour tâche de réaliser la politique de prise de décision qui rapproche les classes détectées dans l'image, et une action à effectuer par le véhicule. Cette partie n'a pas été effectuée par manque de temps.

À la place lorsqu'il y a la présence de la classe correspondant aux piétons, le serveur envoie un message d'alerte au client lui avertissant d'un danger

5.2.1 Déplacement d'images

L'un des principaux problèmes a été de manipuler des images au format PNG et de devoir les transférer sur le serveur tout simplement car une image est lourde et la succession d'enregistrement d'images pourrait abîmer le disque.

La solution est d'encoder l'image dans un format bytes comme le montre le code suivant :

```
1  image = Image.open(image_dir)
2  buffered = BytesIO()
3  image.save(buffered, format="PNG")
4  img_str = base64.b64encode(buffered.getvalue())
5
6  labeled_image = Image.open(labeled_image_dir)
```

Une fois binarisée, l'image est sous un format équivalant au string et peut donc être manipulée par concaténation ou être envoyé sous forme de data et non sous forme de fichier.

Nous envoyons au serveur l'image à segmenter ainsi qu'une image déjà labélisée par le simulateur faisant office de référence. Une fois reçue par le serveur, elles sont séparées et stockées dans une **FIFO** (First In first Out), où le premier élément qui y est ajouté sera le premier à sortir.

5.2.2 Interface web

Avant nous n'avions aucun moyen de visualiser les images que nous avions segmenté. Pour remédier à cela une page html permet désormais de visualiser en temps réel l'image

à traité, sa segmentation et sa labélisation d'origine ainsi que le message à retourner au client.

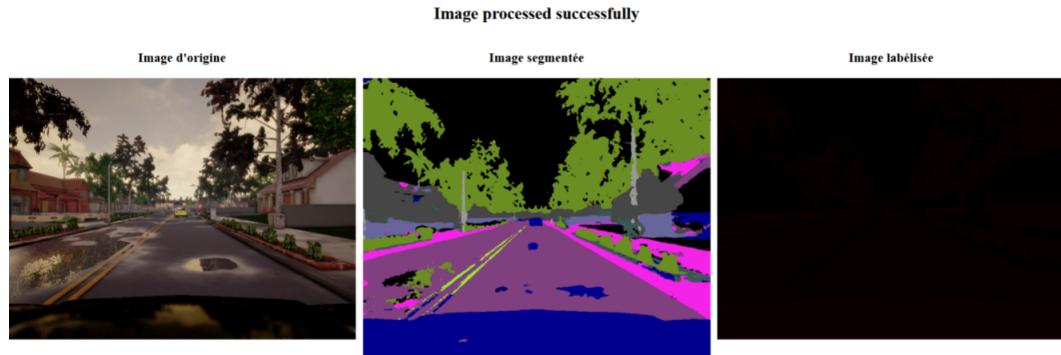


FIGURE 5.1 – Page web montrant les résultats de la segmentation

Il est possible de mettre à jour dynamiquement les informations à afficher sur la page html grâce à un module de flask se nommant **flask-socketio**

A chaque segmentation, la socket envoie les nouvelles informations des variables à afficher avec la fonction **socketio.emit()**

```
1 #Sur l'app.py (serveur)
2 socketio.emit('update_result', {
3     'original_img': app.original_img,
4     'predicted_img': app.predicted_img,
5     'labeled_img': app.labeled_img,
6     'message': app.message
7 })
8
9 #Sur la page html
10 const socket = io.connect('http://localhost:5000');
11
12     socket.on('update_result', function(data) {
13         updateElements(data);
14     });
15
16     function updateElements(data) {
17         console.log("Received update:", data);
18         document.getElementById('original_img').src = "data:
image/png;base64," + data.original_img;
```

```
19     document.getElementById('predicted_img').src = "data:  
20         image/png;base64," + data.predicted_img;  
21     document.getElementById('labeled_img').src = "data:  
22         image/png;base64," + data.labeled_img;  
23     document.getElementById('message').innerHTML = data.  
24         message;  
25 }
```

La fonction **socketio.on()** permet quant à elle de détecer un événement ici "update-result" et de mettre à jour ses informations (les images...)

6 Conclusion

Notre projet visait à exploiter les capacités du simulateur de conduite autonome Carla pour développer un système de détection sémantique d'images. Nous avons réussi à mettre en place une architecture efficace et fonctionnelle qui permet d'envoyer des images capturées dans le simulateur vers un serveur distant, où un modèle de réseau de neurones que nous avons conçu les analyse pour les labéliser.

Ce projet nous a offert une opportunité unique de maîtriser l'utilisation d'un simulateur, en générant des véhicules, en les positionnant judicieusement et en créant un trafic réaliste. En outre, nous avons eu la chance de mettre en œuvre des fonctionnalités avancées, telles que la création d'un dataset et la mise en place de scénarios complexes.

Ce projet nous a permis d'acquérir des connaissances approfondies sur la vision par ordinateur et l'apprentissage automatique. Nous avons également développé des compétences pratiques en utilisant des outils tels que le simulateur **Carla**, les bibliothèques de traitement d'images et les **frameworks** d'apprentissage automatique.

Les principaux défis auxquels nous avons été confrontés furent :

- La puissance de nos ordinateurs, que ce soit pour l'apprentissage des algorithmes, ou pour le lancement du simulateur Carla
- Les problèmes de version de package python
- Le choix du dataset pour le modèle
- La complexité d'utilisation du simulateur Carla et ses fonctionnalités limitées.

Malgré les nombreux problèmes auxquels nous avons été confrontés, nous avons su faire preuve de créativité et d'adaptabilité pour atteindre nos objectifs.

Nous avons également consacré une part significative de notre projet à la conception et à l'entraînement d'un modèle de réseau de neurones pour la détection sémantique des images. En exploitant des architectures avancées telles que les réseaux de neurones pré-entraînés comme **Resnet** ou **MobileNet**.

En conclusion, notre projet a été une expérience enrichissante qui a permis de mettre en pratique nos connaissances théoriques en combinant des domaines tels que la conduite autonome, la vision par ordinateur et l'apprentissage automatique. Nous espérons que notre travail contribuera à l'avancement de la recherche dans ces domaines et inspirera de nouvelles idées et développements.

La perspective que nous pouvons envisager de ce projet est la création d'une véritable conduite autonome à l'aide de notre modèle de détection sémantique, et du simulateur Carla. Ceci implique de réaliser une politique de décision en fonction des éléments détectés par la segmentation de l'image.

De plus, notre projet a encore du potentiel pour être amélioré en termes de performances du réseau de neurones ainsi que du temps de calcul. Une possibilité d'optimisation serait d'explorer l'utilisation de **CUDA** afin de bénéficier de l'accélération GPU.

Bibliographie

- [1] Jean-Emmanuel Deschaud. KITTI-CARLA : a KITTI-like dataset generated by CARLA Simulator. *arXiv e-prints*, 2021.
- [2] Lightning Flash Developers. Lightning Flash SEMANTIC SEGMENTATION Documentation. https://lightning-flash.readthedocs.io/en/stable/reference/semantic_segmentation.html, 2020.