

Projet chef d'Oeuvre

Certification Développeur Data IA

Le Machine Learning au service d'une voiture autonome

| | |
|--|----------|
| Le Machine Learning au service d'une voiture autonome | 2 |
| Introduction | 4 |
| Problématique | 5 |
| Besoins du projet | 5 |
| Gestion de projet | 5 |
| Construire le Hardware | 8 |

| | |
|--|-----------|
| Liste des fournitures | 8 |
| Installation de la Raspberry | 9 |
| Installer le Software | 10 |
| Suivi de trajectoire | 10 |
| Détecter les voies | 11 |
| Collecte de données | 12 |
| Baseline via XGBoost | 13 |
| Modèle de Deep Learning | 14 |
| Détection des objets sur la route | 18 |
| Collecte de données | 20 |
| Choix du modèle | 21 |
| Transfert Learning | 21 |
| Convertir le modèle au format Edge TPU | 23 |
| Réponse à la détection | 25 |
| Base de données | 26 |
| Schéma de la base | 26 |
| Création | 27 |
| Remplissage de la base de données | 28 |
| Sauvegarde de la base de données | 29 |
| Performance | 30 |
| API Web | 31 |
| Pourquoi? | 31 |
| Dash + SQL | 31 |
| Conclusion | 32 |

Introduction

Ce rapport finalise près de deux ans de formation auprès de Simplon. Après 4 mois de cours intensifs, je termine par ce projet mes 15 mois d'alternance en entreprise.

Aujourd'hui, Tesla, Google, Uber et GM essaient tous de créer leurs propres voitures autonomes qui peuvent rouler sur des routes du monde réel. De nombreux analystes prévoient que dans les 5 prochaines années, nous commencerons à avoir des voitures entièrement

autonomes dans nos villes, et d'ici 30 ans, presque toutes les voitures seront entièrement autonomes.

Ne serait-il pas cool de construire notre propre voiture autonome en utilisant certaines des mêmes techniques que celles utilisées par les grands ?

Dans ce projet et dans les prochains, nous allons voir comment construire notre propre voiture robotique physique et autonome à partir de zéro avec du deep learning. Il s'agira de faire en sorte que notre voiture détecte et suive les voies, reconnaisse et réagisse aux panneaux de signalisation et aux personnes sur la route.

Pour commencer nous aurons besoin d'une carte Raspberry Pi, d'un kit SunFounder PiCar, d'un TPU Edge ainsi que de quelques accessoires. Nous installerons également tous les pilotes logiciels nécessaires à Raspberry Pi et PiCar.

Les principaux logiciels que nous utiliserons sont Python (le langage de programmation pour les tâches d'apprentissage automatique), OpenCV (une puissante librairie de computer vision) et Tensorflow (le cadre de deep learning de Google). Notez que tous les logiciels que nous utilisons ici sont gratuits et open source.

La première partie de notre projet consistera à utiliser Python et OpenCv pour apprendre à naviguer de manière autonome sur une route sinueuse à une seule voie en détectant les bordures de la voie. Nous entraîneront un modèle de deep learning pour naviguer de manière autonome.

Dans un second temps, nous utiliserons des techniques de machine learning telles que le transfert learning pour détecter des piétons et des panneaux de signalisation et faire répondre notre véhicule en fonction de ce qui est détecté.

Problématique

Mes collègues et moi-même avons dans l'idée de participer à une course de voiture autonome sur Toulouse organisée par Deep Racers (1).

Les voitures doivent commencer par une seule interaction binaire. Cela peut être un bouton sur la voiture, sur un contrôleur, une touche sur un clavier ou équivalent. Aucune autre intervention ne peut avoir lieu avant la fin de la course, sinon, la voiture obtient un « n'a pas fini » (DNF).

Il n'y a pas de règles régissant l'endroit où le calcul doit avoir lieu. Les voitures peuvent disposer de l'informatique embarquée, de l'informatique au sol ou d'exploiter des ressources distantes ou cloud. Le souhait est qu'il s'agisse d'une ligue open source et que tous les designs soient mis sur GitHub et soient facilement copiables après la conclusion de chaque course.

Les véhicules personnalisés doivent répondre aux critères ci-dessous :

- Les voitures sont à l'échelle 1 / 16e ou moins
- Ne coûte pas plus de 400 \$ au total (voiture, ordinateur, capteurs, etc.)

Il y a trois manches, suivies d'une course « en échelle » des six premières voitures, jumelées aux temps les plus proches, se terminant par une course finale entre les deux premières voitures pour le trophée gagnant (pas vraiment un trophée !). Il n'y a pas de pénalité pour sortir des lignes blanches, tant que la voiture est capable de détecter et d'éviter les obstacles sur la route. Les obstacles peuvent être passés de chaque côté.

Besoins du projet

Pour ce projet nous allons donc nous concentrer sur la création d'un prototype de voiture autonome. Ce prototype devra être capable de suivre une route, le mieux possible pour éviter de perdre du temps dans les virages. Il devra aussi être capable de détecter les obstacles et d'agir en conséquence. Il s'agit ici de tester les capacités du machine learning pour voir si nous sommes capables de rendre notre voiture « intelligente », de lui donner une vision et une capacité à réfléchir.

Gestion de projet

La méthode Kanban a été utilisée pour réaliser ce projet. En effet, l'approche Kanban permet de contrôler visuellement le flux de travail. Il s'agit d'observer la façon de travailler de l'entreprise afin de l'améliorer par la suite. De plus, cette méthode flexible permet à l'équipe de suspendre à tout moment le processus de production afin de résoudre un problème bloquant ou une urgence. Pour ce projet quatre bonnes pratiques de la méthode ont été suivies :

- La visualisation : afin de comprendre comment fonctionnent les dispositifs en place et connaître l'état du projet, il est essentiel de visualiser le flux de travail (workflow). Pour cela, utilisez un tableau dont chaque colonne représente une étape (à faire, en cours, terminé). Chaque tâche évolue jusqu'à ce qu'elle soit achevée.
- La limitation du nombre de tâches en cours: chaque étape du tableau ne peut contenir qu'un nombre maximum de tâches en même temps, défini en fonction des capacités de l'équipe. Lorsqu'une tâche est terminée, une nouvelle peut alors être ajoutée. Étant seul à travailler sur le projet, la limite des tâches exécutées en même temps était cruciale.
- La gestion du flux : il est essentiel de suivre, mesurer et consigner le déroulement du travail à travers chaque étape du tableau. Le but est de connaître la vitesse et la fluidité du travail. Chaque fin de week-end je faisais un point sur l'avancée du projet, résumé des points bloquants et idée pour avancer.
- L'identification des opportunités d'amélioration: être capable de discuter d'un problème ou d'un blocage auquel nous sommes confrontés et de trouver des améliorations à mettre en place.

Pour la gestion de ce projet j'ai utilisé le logiciel pour la collaboration et la gestion de travail Smartsheet (2), développée et commercialisée par Smartsheet Inc. Il est utilisé pour attribuer

des tâches, suivre l'avancement du projet, gérer les calendriers, partager des documents et gérer d'autres travaux, à l'aide d'une interface utilisateur tabulaire.

Les différentes étapes de mon projet sont définies sur le tableau de la figure (Figure 1) qui suit :

| Risque | Nom de la tâche | Statut | Date de début | Date de fin | Responsable | % complet |
|--------|---|----------|---------------|-------------|-------------|-----------|
| 1 | Pour en savoir plus sur les diagrammes de Gantt dans SmartSheet, cliquez ici. | | | | | |
| 2 | Section 1 - Rechercher documentation | | 01/09/20 | 11/10/20 | | 100% |
| 3 | Sous-tâche 1 - Etat de l'art | Terminé | 01/09/20 | 20/09/20 | Yohan | 100% |
| 4 | Sous-tâche 2 - Recherche sur l'apprentissage par renforcement | Terminé | 20/09/20 | 04/10/20 | Yohan | 100% |
| 5 | Sous-tâche 3 - Recherche sur les modèles IA | Terminé | 20/09/20 | 11/10/20 | Yohan | 100% |
| 6 | Section 2 - Construire le hardware | | 17/10/20 | 18/10/20 | | 100% |
| 7 | Sous-tâche 1 - Montage de la voiture | Terminé | 17/10/20 | 17/10/20 | Yohan | 100% |
| 8 | Sous-tâche 2 - Installer la raspberry | Terminé | 17/10/20 | 18/10/20 | Yohan | 100% |
| 9 | Sous-tâche 3 - Tester la voiture | Terminé | 18/10/20 | 18/10/20 | Yohan | 100% |
| 10 | Section 3 - Suivre de Vole | | 02/01/21 | 31/01/21 | | 100% |
| 11 | Sous-tâche 1 - Collecter les données | Terminé | 02/01/21 | 10/01/21 | Yohan | 100% |
| 12 | Sous-tâche 2 - Entraîner les modèles IA | Terminé | 10/01/21 | 17/01/21 | Yohan | 100% |
| 13 | Sous-tâche 3 - Tester le modèle sur la voiture | Terminé | 17/01/21 | 31/01/21 | Yohan | 100% |
| 14 | Sous-tâche - Seconde collecte de données | En cours | 24/01/21 | 31/01/21 | Yohan | 50% |
| 15 | Section 4 - Détection d'objet | | 06/02/21 | 31/03/21 | | 66% |
| 16 | Sous-tâche 1 - Collecter les données et labélisation | Terminé | 06/02/21 | 07/02/21 | Yohan | 50% |
| 17 | Sous-tâche 2 - Entraîner les modèles IA en transfert learning | Terminé | 07/02/21 | 28/02/21 | Yohan | 100% |
| 18 | Sous-tâche 3 - Tester le modèle sur la voiture | En cours | 28/02/21 | 31/03/21 | Yohan | 50% |
| 19 | Section 5 - Base de données | | 01/01/21 | 04/04/21 | | 66% |
| 20 | Sous-tâche 1 - Schéma de la base + création | Terminé | 01/01/21 | 31/01/21 | Yohan | 100% |
| 21 | Sous-tâche 2 - Insérer données dans la base | Terminé | 31/01/21 | 31/03/21 | Yohan | 50% |
| 22 | Sous-tâche 3 - Tester la base de données | Terminé | 22/03/21 | 31/03/21 | Yohan | 100% |
| 23 | Sous-tâche 4 - Optimiser la base de données | En cours | 31/03/21 | 04/04/21 | Yohan | 0% |
| 24 | Section 6 - API Web | | 22/02/21 | 31/03/21 | | 100% |
| 25 | Sous-tâche 1 - Recherche d'api web pour la visualisation | Terminé | 22/02/21 | 26/02/21 | Yohan | 100% |
| 26 | Sous-tâche 2 - Ecrire script + requête sql | En cours | 26/02/21 | 31/03/21 | Yohan | 100% |

Figure 1 : Calendrier du projet

En ce qui concerne la gestion du flux, cela peut être résumé comme suit (Figure 2) à la date du 31 Mars :

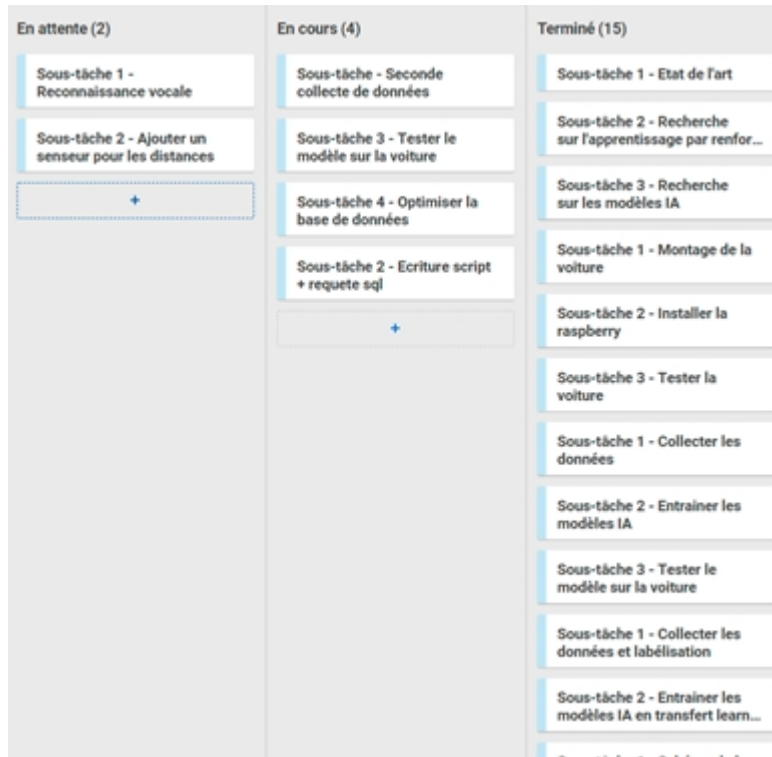


Figure 2: Gestion du flux à la date du 31 Mars 2021

Construire le Hardware

Liste des fournitures

La première étape de ce projet a consisté à construire le modèle de voiture qui va nous servir tout le long. J'ai choisi comme premier prototype une voiture du fabricant SunFounder (3) et plus précisément la version V2.0 (4). J'ai choisi ce kit parce qu'il vient avec une API python open source pour contrôler la voiture contrairement à d'autres vendeurs qui ont des API privé ou basé sur le langage C. De plus, j'ai trouvé un bon nombre de ressources sur ce modèle pour m'aider dans mon projet.

La voiture PiCar-V est équipée d'une caméra USB, de trois circuits imprimés, de câbles de connexion, de trois servo (permettant les mouvements), de deux moteurs et de plaques acryliques.

Nous avons donc la base, il reste la partie la plus importante de notre voiture, son ordinateur de bord représenté par une Raspberry Pi 3b. La raspberry dispose de 4 port USB, d'un port HDMI et un adaptateur secteur permettant de brancher la voiture lorsqu'elle ne roule pas pour pouvoir tester le code. Pour contenir le système opérateur et le software, une carte Micro SD (32 GB) a été utilisée.

Pour faire tourner la voiture, il est nécessaire d'utiliser 2 piles rechargeables 18650 ainsi qu'un chargeur pour recharger les piles durant la nuit.

En tant que complément à la Raspberry Pi, un Google's Edge TPU (Edge signifie qu'il est destiné aux appareils mobiles et embarqués et TPU signifie Tensor Processing Unit) a été utilisé. Alors que le processeur Pi contient beaucoup de puissance de calcul dans un petit paquet, il n'est PAS conçu pour le deep learning. Le tout nouveau Edge TPU de Google (mars 2019), quant à lui, est spécialement conçu pour exécuter des modèles de deep learning écrits dans TensorFlow. L'exécution d'un modèle sur le processeur de Raspberry Pi seul ne peut traiter qu'une image par seconde (FPS), ce qui est à peine en temps réel. De plus, il consomme 100% du processeur et rend tous les autres programmes non réactifs. Mais avec l'aide d'Edge TPU, nous pouvons désormais traiter un plus grand nombre de FPS, ce qui est suffisant pour un travail en temps réel. Et notre processeur reste froid et peut être utilisé pour effectuer d'autres tâches de traitement, comme le contrôle de la voiture.

Et finalement pour entraîner notre voiture à la détection d'objets sur la route, j'ai acheté un set de panneaux de signalisations et de personnes miniatures.

Installation de la Raspberry

Pour installer le système Raspbian sur la carte micro SD pour la Raspberry j'ai suivi ce guide (5). Une fois installé, un bureau de style Windows apparaît sur l'écran. Le problème est que notre voiture est alors connectée à un écran (via HDMI) et à une souris et clavier (via USB), dans ces conditions il n'est pas possible de la faire rouler.

Il a donc fallu créer un accès à distance, pour éviter d'avoir à chaque fois l'écran et le clavier/souris. Pour cela j'ai mis en place un accès à distance SSH et VNC avec l'aide d'une vidéo explicative (6).

Ca y est notre voiture n'est plus reliée à quoi que ce soit, mais nous avons besoin d'accéder aux fichiers présents sur la Raspberry depuis l'ordinateur, de sorte que nous puissions transférer les fichiers depuis et vers la Raspberry Pi. C'est dans cette optique que j'ai utilisé FileZilla. Sous Windows, comme souvent, nous n'allons pas transférer les fichiers directement en lignes de commandes, mais nous allons utiliser une interface graphique à la place. En effet, si FileZilla est à l'origine un client FTP, il fait aussi office de client SFTP et SSH. Via une connexion SSH, il est alors possible d'explorer le système de fichier comme le pourrait votre utilisateur Pi (7).

Une fois que la voiture est construite et la Raspberry Pi connecté, nous avons accès à distance sur notre ordinateur à la voiture grâce à VNC viewer.

Installer le Software

Notre voiture voit ce qu'il entoure via la caméra USB, fournissant ainsi une vidéo. Nous utiliserons OpenCV, une puissante librairie de computer vision open source, pour capturer et transformer ces images afin que nous puissions donner un sens à ce que la caméra voit.

Pour le Deep Learning, TensorFlow de Google est actuellement la librairie Python la plus populaire. Il peut être utilisé pour la reconnaissance d'image, la détection de visage, le traitement du langage naturel et de nombreuses autres applications. Il existe deux méthodes pour installer TensorFlow sur Raspberry Pi:

- TensorFlow pour CPU
- TensorFlow pour Edge TPU Co-Processor

En premier nous installons la version CPU de TensorFlow. Nous n'utilisons pas Pi pour entraîner notre modèle car son processeur est largement insuffisant. Cependant, nous pouvons utiliser le processeur pour faire des inférences basées sur un modèle pré-entraîné. Même si le processeur ne fait que des inférences, il ne peut le faire que sur un modèle relativement peu profond (disons 20 à 30 couches) en temps réel. Pour les modèles plus profonds (plus de 100 couches), nous aurions besoin du Edge TPU. La version de production la plus récente de TensorFlow est la version 2.0, mais pour ce projet, je suis resté sur la version de Tensorflow que je maîtrise le plus, la version 1.15.

Puis dans un second temps, nous installons Tensorflow pour Edge TPU, en suivant ces différentes étapes:

- Tout d'abord il faut suivre cette documentation pour installer The Edge TPU runtime (8)
- Installer tensorflow lite (9)
- Installer la version de Numpy compatible (1.16.5)
- Installer Keras
- Finalement utilisé cette commande: `sudo apt-get install python3-edgetpu`

Cette dernière commande installera avec succès Edge tpu, puis le code devrait fonctionner.

Suivi de trajectoire

Un système d'assistance au maintien de voie comprend deux composants, à savoir la perception (détection de voie) et la planification de trajectoire / mouvement (direction). Le travail de la détection de voie consiste à transformer une vidéo de la route en coordonnées des lignes de voie détectées. Un moyen d'y parvenir est d'utiliser le package de computer vision, que nous avons installé OpenCV. Mais avant de pouvoir détecter les lignes de voie dans une vidéo, nous devons être capables de détecter les lignes de voie dans une seule image. Une fois que nous pouvons le faire, détecter les lignes de voie dans une vidéo consiste simplement à répéter les mêmes étapes pour toutes les images d'une vidéo.

Détecter les voies

Lorsque j'ai mis en place des lignes de voies pour la voiture dans mon salon, j'ai utilisé le ruban bleu pour marquer les voies, car le bleu est une couleur unique dans ma chambre.

La première chose à faire est d'isoler toutes les zones bleues de l'image. Pour ce faire, nous devons d'abord transformer l'espace colorimétrique utilisé par l'image, qui est RVB (Rouge / Vert / Bleu) dans l'espace colorimétrique HSV (Teinte / Saturation / Valeur). L'idée principale derrière cela est que dans une image RVB, différentes parties du ruban bleu peuvent être éclairées avec une lumière différente, ce qui les fait apparaître en bleu plus foncé ou bleu plus clair. Cependant, dans l'espace colorimétrique HSV, le composant Teinte rendra l'intégralité du ruban bleu en une seule couleur, quel que soit son ombrage.

Ensuite, nous devons détecter les bords dans le masque bleu afin que nous puissions avoir quelques lignes distinctes qui représentent les lignes bleues de la voie.

La fonction de détection des contours Canny est une commande puissante qui détecte les contours d'une image. Dans le code ci-dessous, le premier paramètre est le masque bleu de l'étape précédente. Les deuxième et troisième paramètres sont des plages inférieure et supérieure pour la détection de bord, qu'OpenCV recommande d'être (100, 200) ou (200, 400), nous utilisons donc (200, 400).

En réunissant les commandes ci-dessus, vous trouverez ci-dessous la fonction qui isole les couleurs bleues sur l'image et extrait les bords de toutes les zones bleues.

```
def detect_edges(frame):  
    # filter for blue lane lines  
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)  
    show_image("hsv", hsv)  
    lower_blue = np.array([60, 40, 40])  
    upper_blue = np.array([150, 255, 255])  
    mask = cv2.inRange(hsv, lower_blue, upper_blue)  
    show_image("blue mask", mask)  
  
    # detect edges  
    edges = cv2.Canny(mask, 200, 400)  
  
    return edges
```

Le code complet pour effectuer le suivi de voie se trouve dans le dépôt DeepPiCar GitHub (10). Dans le dossier DeepPiCar / driver / code, voici le fichiers d'intérêt:

- hand_coded_lane_follower.py: Il s'agit de la détection de voie

Nous avons appris à notre DeepPiCar à naviguer de manière autonome dans les lignes de voies (LKAS), ce qui est assez impressionnant, car la plupart des voitures sur le marché ne peuvent pas encore le faire (Figure 3). Cependant, ce n'est pas très satisfaisant, car nous avons dû utiliser un code de plus de 200 lignes en python et OpenCV pour détecter la couleur, détecter les bords, détecter les segments de ligne, puis deviner quels segments de ligne

appartiennent à la voie gauche ou droite. En fait, nous n'avons utilisé aucune technique de deep learning. Ne serait-il pas cool de simplement «montrer» à DeepPiCar comment conduire et de lui faire comprendre comment diriger?

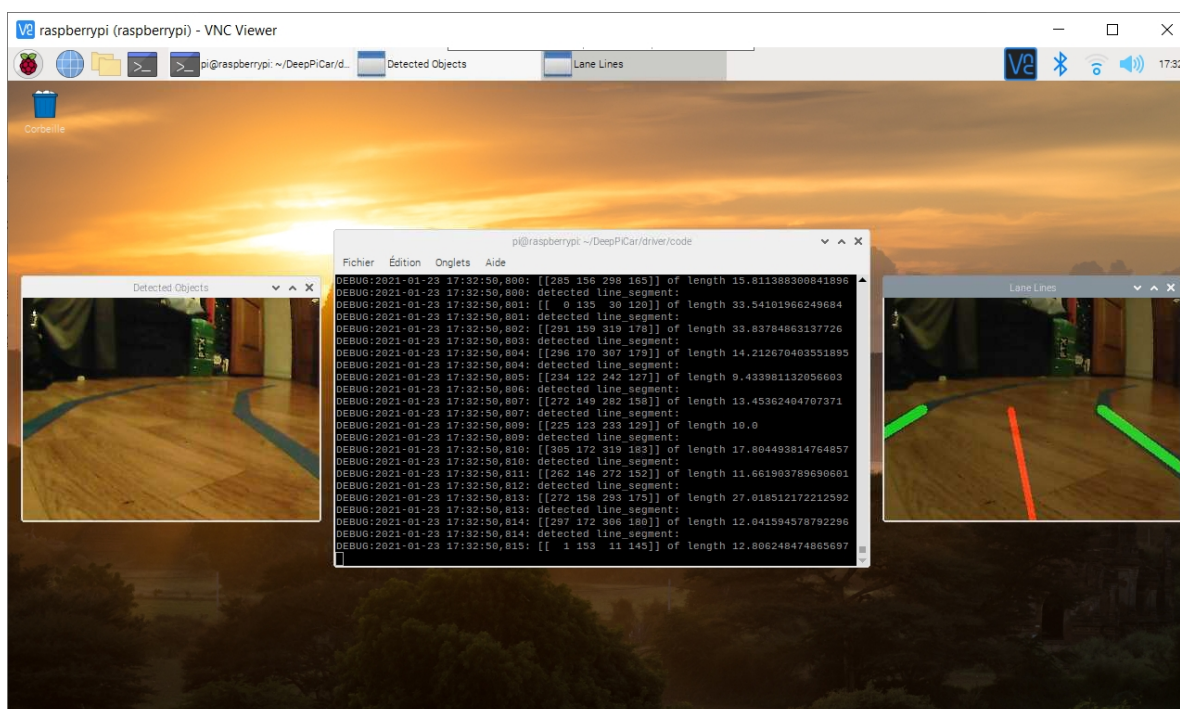


Figure 3: Suivi de voie vue par la voiture

Collecte de données

Heureusement, les chercheurs de Nvidia ont démontré dans cet excellent article (11) qu'en «montrant» à une voiture grandeur nature comment conduire, la voiture apprendrait à conduire par elle-même.

Pour la collecte de données, tout ce que nous avons à faire est d'exécuter le code OpenCV sur la piste plusieurs fois, d'enregistrer les fichiers vidéo et les angles de direction correspondants. Nous pouvons ensuite les utiliser pour entraîner notre modèle Nvidia. Voici le code pour prendre un fichier vidéo et enregistrer les images vidéo individuelles pour l'entraînement. Pour plus de simplicité, j'intègre l'angle de braquage dans le nom du fichier image, donc je n'ai pas à conserver un fichier de mappage entre les noms d'image et les angles de braquage.

```

import cv2
import sys
from hand_coded_lane_follower import HandCodedLaneFollower

def save_image_and_steering_angle(video_file):
    lane_follower = HandCodedLaneFollower()
    cap = cv2.VideoCapture(video_file + '.avi')

    try:
        i = 0
        while cap.isOpened():
            _, frame = cap.read()
            lane_follower.follow_lane(frame)
            cv2.imwrite("%s_%03d_%03d.png" % (video_file, i, lane_follower.curr_steering_angle),
                        frame)
            i += 1
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break
    finally:
        cap.release()
        cv2.destroyAllWindows()

if __name__ == '__main__':
    save_image_and_steering_angle(sys.argv[1])

```

Maintenant que nous avons les données (images) et les labels (angles de braquage), il est temps d'entraîner un modèle de machine learning.

Baseline via XGBoost

Nous avons à faire ici à un problème de régression, nous voulons un modèle capable de prédire l'angle de braquage à partir d'une image. J'ai décidé d'utiliser XGBoost Regressor comme modèle de base car il a tendance à donner de bon résultats et est très facile à utiliser.

La première étape consiste à charger les images, puis de changer les images en array. A partir de là il faut splitter les données en dataset d'entraînement et de validation suivant la répartition 80/20.

Après avoir entraîné notre modèle, il nous faut l'évaluer. Nous avons un R^2 de 95% et une MSE de 30.

Pour essayer d'améliorer ce modèle, un grid search a été utilisé:

```

from sklearn.model_selection import GridSearchCV

# Various hyper-parameters to tune
xgb1 = XGBRegressor()
parameters = {'nthread':[4], #when use hyperthread, xgboost may become slower
              'objective':['reg:linear'],
              'learning_rate': [.03, 0.05, .07], #so called `eta` value
              'max_depth': [5, 6, 7],
              'min_child_weight': [4],
              'subsample': [0.7],
              'colsample_bytree': [0.7],
              'n_estimators': [500]}

xgb_grid = GridSearchCV(xgb1, parameters, cv = 2, n_jobs = 5, verbose=True)

xgb_grid.fit(X_train, y_train)

print(xgb_grid.best_score_)
print(xgb_grid.best_params_)

```

Ce nouveau modèle a un R^2 de 96% et une MSE de 18. Il est bien meilleur que le modèle précédent. Il nous reste à le comparer à notre modèle de Deep Learning.

Modèle de Deep Learning

En ce qui concerne le Deep Learning, j'ai utilisé le modèle de Nvidia, les entrées du modèle Nvidia sont des images vidéo de DashCams à bord de la voiture, et les sorties sont l'angle de braquage de la voiture. Le modèle utilise les images vidéo, en extrait des informations et tente de prédire les angles de braquage de la voiture. Il s'agit d'un apprentissage supervisé, dans lequel des images vidéo (appelées features) et des angles de braquage (appelés labels) sont utilisés lors de l'entraînement. Parce que les angles de braquage sont des valeurs numériques, il s'agit d'un problème de régression.

Au cœur du modèle NVidia, il y a un réseau neuronal convolutif (CNN). Les CNN sont principalement utilisés dans les modèles de Deep Learning de reconnaissance d'image. L'intuition est que CNN est particulièrement efficace pour extraire les caractéristiques visuelles des images de ses différentes couches (alias. Filtres). Par exemple, pour un modèle CNN de reconnaissance faciale, les couches précédentes extrairaient des fonctionnalités de base, telles que la ligne et les bords, les couches intermédiaires extrairaient des fonctionnalités plus avancées, telles que les yeux, le nez, les oreilles, les lèvres, etc., et les couches ultérieures extrairaient une partie ou la totalité d'un visage.

La figure ci-dessous (Figure 4) est tirée de l'article de Nvidia. Il contient environ 30 couches au total, ce n'est pas un modèle très profond selon les normes d'aujourd'hui. L'image d'entrée du modèle (en bas du diagramme) est une image de 66x200 pixels, qui est une image assez basse résolution. L'image est d'abord normalisée, puis passée à travers 5 groupes de couches

convolutives, enfin passée à travers 4 couches neuronales entièrement connectées et arrivée à une seule sortie, qui est l'angle de braquage prédit par le modèle de la voiture.

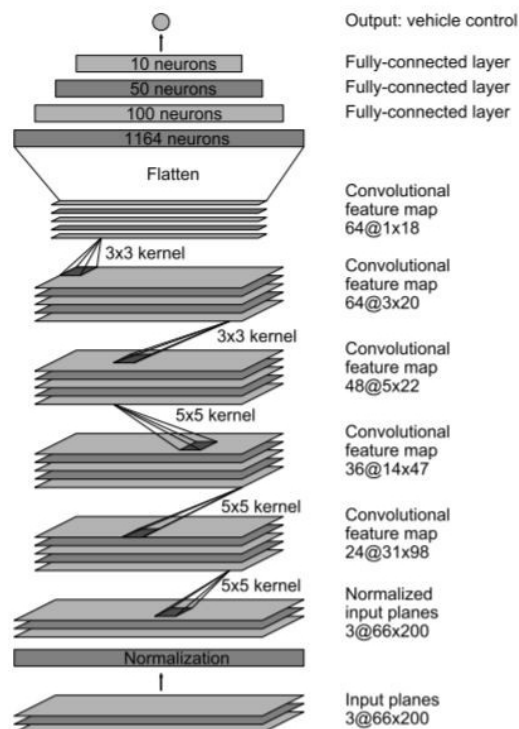


Figure 4: Architecture du Modèle

L'angle prédit par le modèle est ensuite comparé à l'angle de braquage souhaité étant donné l'image vidéo, l'erreur est renvoyée dans le processus d'apprentissage CNN via une rétropropagation. Ce processus est répété en boucle jusqu'à ce que les erreurs (aka perte ou erreur quadratique moyenne) soient suffisamment faibles, ce qui signifie que le modèle a appris à diriger raisonnablement bien. En effet, il s'agit d'un processus d'entraînement à la reconnaissance d'image assez typique, sauf que la sortie prédite est une valeur numérique (régression).

J'ai utilisé Jupyter Notebook pour entraîner ce modèle. La première étape consiste à charger les images depuis nos fichiers locaux sur l'ordinateur. Les images sont nommées de telles façons:

- Video01_054_110.png: où 01 est le trajet suivi, 054 est le 54ème frame et 110 est l'angle de braquage.

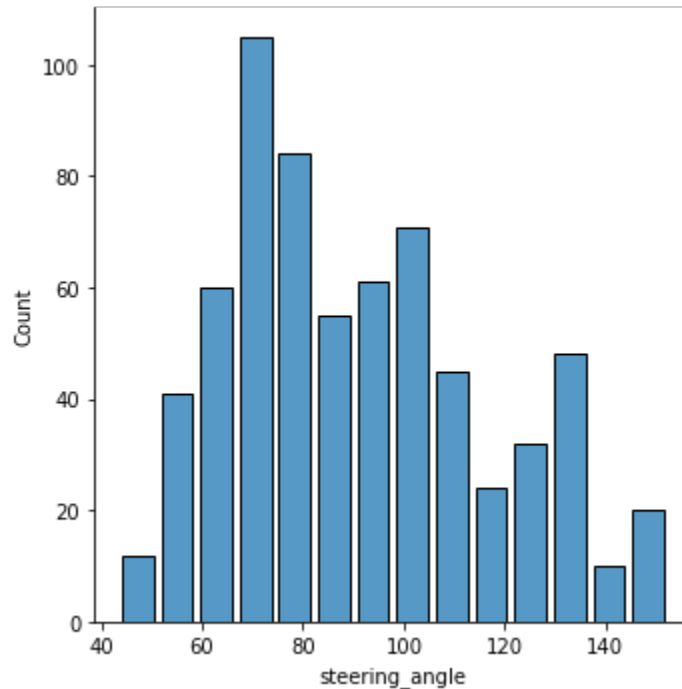


Figure 5: Distribution des angles

Le diagramme ci-dessus (Figure 5) représente la distribution des angles dans nos données. Notez que le diagramme contient des angles généralement plus petits que 90. Cela a du sens, car dans nos données d'entraînement, la voiture tournait principalement à gauche. Cela sera arrangé, car nous équilibrerons les données en retournant au hasard l'image et l'angle de braquage dans le processus de Data Augmentation.

Avec la méthode `train_test_split` de `sklearn` nous séparons les données en dataset de train et dataset de validation 80/20.

Notre jeu de données d'entraînement ne contient que 667 images. De toute évidence, cela n'est pas beaucoup pour entraîner notre modèle de Deep Learning. C'est pour cela que nous avons utilisé une technique simple, appelée Data Augmentation. Certaines des opérations d'augmentation courantes sont le zoom, le panoramique, la modification des valeurs d'exposition, le flou et le retournement de l'image. En appliquant au hasard l'une ou l'ensemble de ces 5 opérations sur les images originales, nous pouvons générer beaucoup plus de données d'entraînement à partir de nos images originales, ce qui rend notre modèle entraîné final beaucoup plus robuste. Je vais juste illustrer le zoom ci-dessous (Figure 6).

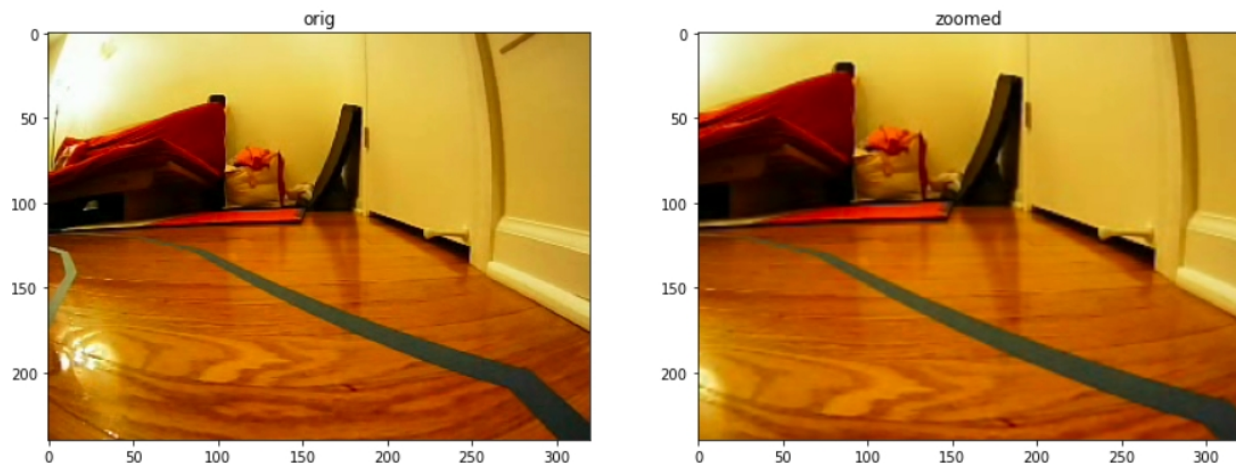


Figure 6: Exemple de Zoom sur une des images de notre dataset

Nous devons également modifier nos images dans l'espace colorimétrique et la taille acceptés par le modèle Nvidia. Premièrement, l'article de Nvidia appelle à des images d'entrée en résolution de 200x66 pixels. La moitié supérieure de l'image n'est pas pertinente pour prédire l'angle de braquage, nous allons donc simplement la rogner. Deuxièmement, le modèle demande que les images soient dans l'espace colorimétrique YUV. Nous utiliserons simplement `cv2.cvtColor()` pour faire cela. Enfin, le modèle requiert que les images soient normalisées.

Nous avons implémenté assez fidèlement l'architecture du modèle Nvidia, sauf que nous avons supprimé les couches de normalisation, comme la normalisation est faite en dehors du modèle, et ajouté deux couches dropout, pour rendre le modèle plus robuste. La fonction de perte que nous utilisons est l'erreur quadratique moyenne (MSE) parce que nous faisons un entraînement à la régression. Nous avons également utilisé la fonction d'activation ELU (Exponential Linear Unit) au lieu de la fonction d'activation habituelle ReLU (Rectified Linear Unit) car ELU n'a pas le problème de ReLU lorsque x est négatif. Lorsque nous créons le modèle et imprimons sa liste de paramètres, nous voyons bien qu'il contient environ 250 000 paramètres. C'est une bonne vérification que chaque couche de notre modèle est créée comme prévu.

Une fois le modèle entraîné, il faut l'évaluer. La première chose à faire est d'imprimer les courbes de loss pour les données d'entraînement et de validation (Figure 7). Nous pouvons remarquer que les loss d'entraînement et de validation ont diminué rapidement ensemble, puis sont restées très faibles après l'époch 8. Il ne semblait pas y avoir de problème d'overfitting.

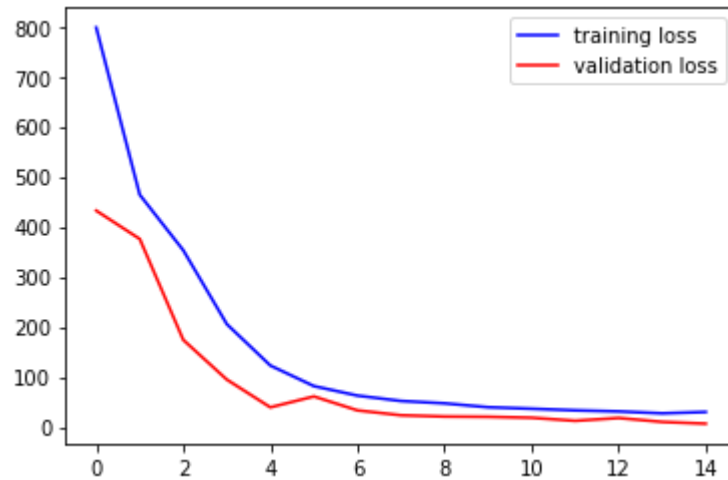


Figure 7: Courbe de loss

Une autre métrique pour voir si notre modèle a bien fonctionné est la métrique R^2 . Un R^2 proche de 100% signifie que le modèle fonctionne plutôt bien. Comme nous pouvons le voir, notre modèle a un R^2 de 97%, ce qui est plutôt bon. La MSE est de 15. Le modèle de Deep Learning a donc de meilleur résultat que XGBoost.

Pour voir si notre modèle est vraiment bon, il doit être finalement testé sur la route. Pour cela, nous devons importer notre modèle et réaliser des prédictions.

```
model = load_model('lane_navigation_model.h5')

def compute_steering_angle(self, frame):
    preprocessed = img_preprocess(frame)
    X = np.asarray([preprocessed])
    steering_angle = model.predict(X)[0]
    return steering_angle
```

Le code source complet du code de navigation à l'aide du modèle d'apprentissage profond formé peut être trouvé ici (10). Notez que le fichier contient quelques fonctions d'assistance et de test pour faciliter l'affichage et les tests.

Détection des objets sur la route

Maintenant que notre voiture peut suivre une route, nous allons l'entraîner à identifier et à répondre aux panneaux de signalisation (miniaturisés) et aux piétons en temps réel. Nous utiliserons également l'accélérateur TPU Google Edge pour réaliser les inférences.

La détection d'objets est un problème bien connu en computer vision. Il existe deux composants dans un modèle de détection d'objets, à savoir le réseau neuronal de base et le réseau neuronal de détection.

Premièrement, les réseaux neuronaux de base sont des CNN qui extraient des caractéristiques d'une image, des caractéristiques de bas niveau, telles que des lignes, des bords ou des cercles à des caractéristiques de plus haut niveau, comme un visage, une personne, un feu de signalisation ou un panneau stop. , etc. Quelques réseaux neuronaux de base bien connus sont ResNet, VGG-Net, et MobileNet, etc.

Ensuite, les réseaux neuronaux de détection sont attachés à l'extrémité d'un réseau neuronal de base et utilisés pour identifier simultanément plusieurs objets à partir d'une seule image à l'aide des caractéristiques extraites. Certains des réseaux de détection populaires sont les SSD (Détecteur Multi Box Single Shot), Faster R-CNN et YOLO, etc.

Un modèle de détection d'objet est généralement nommé comme une combinaison de son type de réseau de base et de son type de réseau de détection. Par exemple, un modèle «MobileNet SSD».

Enfin, pour les modèles de détection pré-entraînés, le nom du modèle inclurait également le type d'ensemble de données d'image sur lequel il a été formé. Quelques ensembles de données bien connus utilisés dans la formation des classificateurs et des détecteurs d'images sont l'ensemble de données COCO (environ 100 objets ménagers courants). Par exemple, Le modèle `ssd_mobilenet_v2_coco` utilise la 2ème version de MobileNet pour extraire des fonctionnalités, SSD pour détecter des objets et pré-entraîné sur l'ensemble de données COCO.

Google a publié une liste de modèles pré-entraînés avec TensorFlow (appelé Model Zoo) afin de pouvoir simplement télécharger celui qui correspond à nos besoins et l'utiliser directement dans nos projets pour inférences (12).

Cependant, nous voulons détecter nos mini panneaux de signalisation et des piétons, pas de livres ou de canapés. Mais nous ne voulons pas collecter et étiqueter des centaines de milliers d'images et passer des semaines ou des mois à construire et à entraîner un modèle de détection à partir de zéro. Ce que nous pouvons faire, c'est tirer parti de l'apprentissage par transfert (transfert learning) qui commence par les paramètres de modèle d'un modèle pré-entraîné, ne lui fournit que 50 à 100 de nos propres images et labels, et ne passe que quelques heures à entraîner des parties du réseau de neurones de détection. L'intuition est que dans un modèle pré-entraîné, les couches CNN de base sont déjà efficaces pour extraire des caractéristiques d'images, car ces modèles sont entraînés sur un grand nombre et une grande variété d'images. La distinction est que nous avons maintenant un ensemble de types d'objets (6) différent de celui des modèles pré-entraînés (~ 100–100 000 types).

Collecte de données

Nous avons 6 types d'objets, à savoir, le feu rouge, le feu vert, le panneau stop, la limite de vitesse de 50 Km/h, la limite de vitesse de 30 Km/h et quelques figurines Playmobil en tant que piétons. J'ai donc pris 63 photos similaires à celles ci-dessous (Figure 8) et placé les objets au hasard dans chaque image.



Figure 8: Exemple d'images collectées pour la détection

Ensuite, j'ai labellisé chaque image en englobant avec une boîte chaque objet de l'image. Cela a été facilité grâce à un outil gratuit, appelé Labellmg (13), qui a fait de cette tâche ardue un jeu d'enfant (Figure 9). Tout ce que j'avais à faire était de pointer Labellmg vers le dossier où les images d'entraînement étaient stockées, pour chaque image, faire glisser une boîte autour de chaque objet sur l'image et choisir un type d'objet (s'il s'agissait d'un nouveau type, je pouvais rapidement créer un nouveau type). Donc, en utilisant les raccourcis clavier, vous ne passeriez qu'environ 20 à 30 secondes par image, et il ne m'a donc fallu environ 40 minutes pour étiqueter environ 60 photos. Ensuite, je divise au hasard les images (et les fichiers xml de label que créer Labellmg) en dossiers d'entraînement et de test. Vous pouvez trouver mes données de train et test dans mon Github sous `models/object_detection/data` (14).



Figure 9: LabellImg pour labelliser les images pour la détection

Choix du modèle

Sur un Raspberry Pi, étant donné que nous avons une puissance de calcul limitée, nous devons choisir un modèle qui fonctionne à la fois relativement rapidement et avec précision. D'après mes recherches c'est le cas de MobileNet. J'ai donc choisi le modèle MobileNet v2 SSD COCO comme équilibre optimal entre vitesse et précision. De plus, pour que notre modèle fonctionne sur l'accélérateur Edge TPU, nous devons choisir le modèle quantifié COCO SSD MobileNet v2. La quantification est un moyen d'accélérer l'exécution des inférences de modèle en stockant les paramètres du modèle non pas sous forme de valeurs doubles, mais sous forme de valeur intégrale, avec très peu de dégradation de la précision de la prédiction. Le matériel Edge TPU est optimisé et ne peut exécuter que des modèles quantifiés. L'article de Mr. Suryavansh (15) approfondit le matériel et les performances d'Edge TPU, pour les personnes intéressées.

Transfert Learning

Pour la détection des différents objets, nous allons faire du transfert learning. Vu la lourdeur de l'entraînement et le manque de puissance de la machine, je suis passé par Google collab qui nous permet d'utiliser gratuitement un GPU.

Cette section est basée sur le tutoriel "Setup TensorFlow for Object Detection on Ubuntu 16.04" (16).

Le code du notebook entier peut être trouvé dans mon Github (14) sous model/object_detection_code/[tensorflow_traffic_sign_detection.ipynb](#). Le notebook étant assez long je ne vais présenter que quelques étapes clés.

La première étape clé est le téléchargement du modèle choisi (MobileNet v2 SSD COCO Quantized) et nous allons utiliser le fichier model.ckpt pour appliquer notre Transfert Learning.

La seconde étape clé est de convertir les fichiers xml des labels au format binaire (.record) pour que Tensorflow puisse les traiter rapidement.

Il faut bien sûr entraîner notre modèle, ce qui prend plus de 5 heures pour 2000 étapes. L'entraînement a pris plusieurs heures, mais travailler sur Collab m'a permis d'enregistrer les poids sur le drive directement facilitant l'entraînement après l'avoir lancé une fois.

Pendant l'entraînement, nous pouvons suivre la progression de la loss et de la précision via TensorBoard (Figure 10) (17). Nous pouvons voir pour l'ensemble de données que la perte diminuait et que la précision augmentait tout au long de l'entraînement, ce qui est un excellent signe que notre entraînement fonctionne comme prévu.

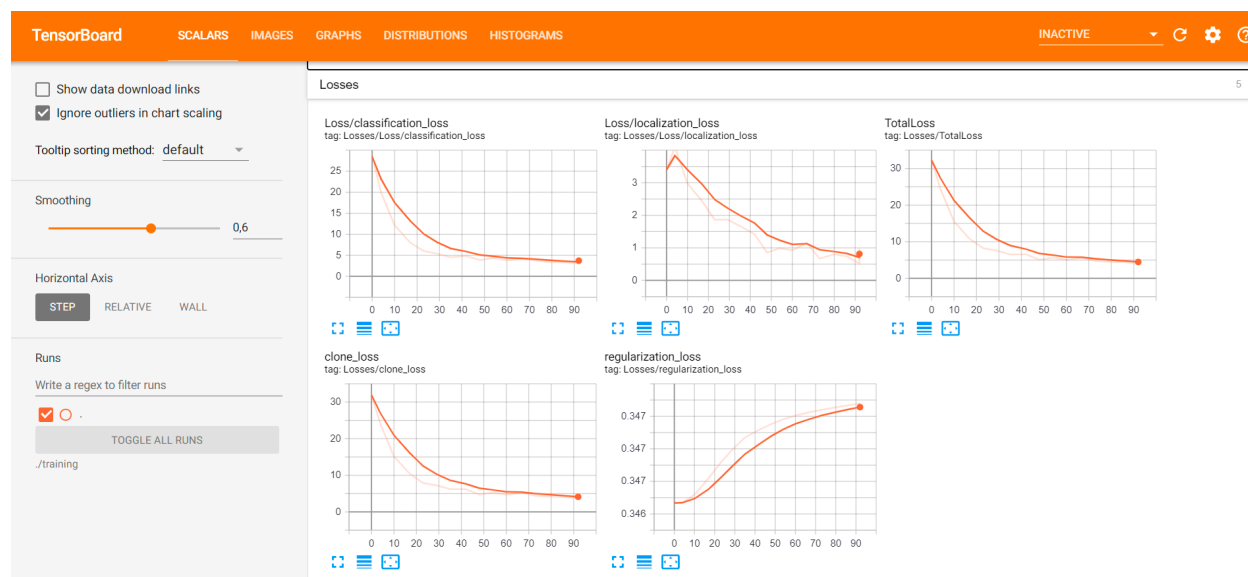


Figure 10: Distribution de la loss via Tensorboard

Après l'entraînement, nous avons testé quelques images de l'ensemble de données de test avec notre nouveau modèle (Figure 11). Comme prévu, presque tous les objets de l'image ont été identifiés avec une confiance relativement élevée. Il y avait quelques images dont les objets étaient plus éloignés et n'ont pas été détectés. C'est très bien pour notre objectif, car nous voulions uniquement détecter les objets à proximité afin de pouvoir y répondre. Les objets les plus éloignés deviendraient plus grands et plus faciles à détecter lorsque notre voiture les approchera.

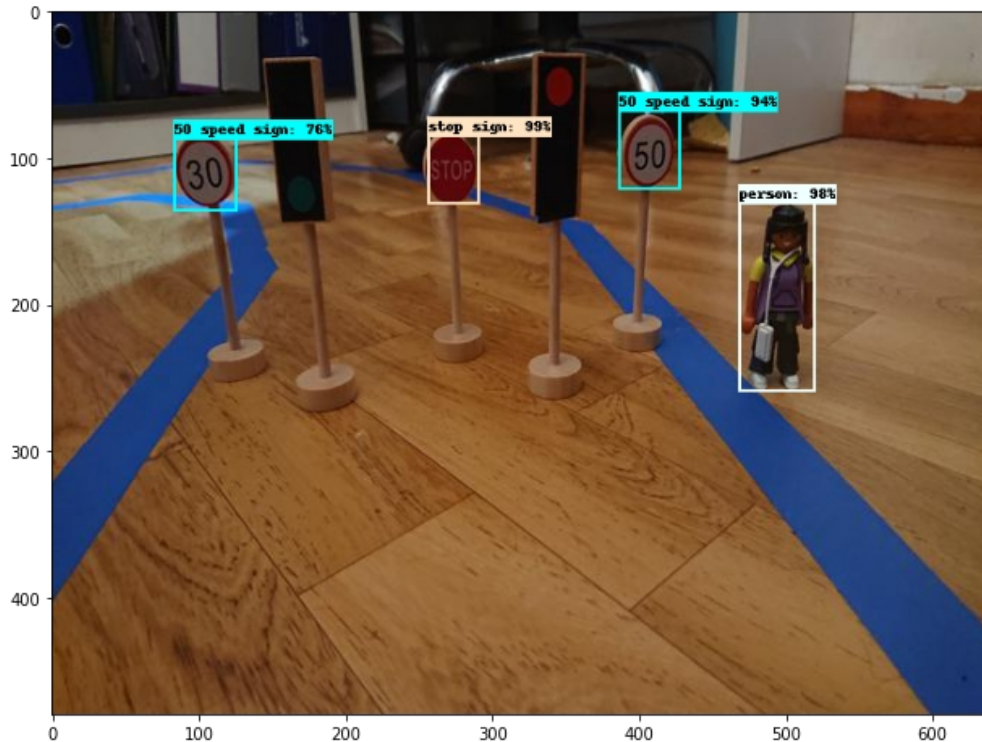


Figure 11: Image test de notre modèle

Maintenant que nous avons entraîné avec succès notre modèle, nous devons générer le graphe d'inférence. Le graphe d'inférence est le fichier utilisé par les applications qui souhaitent exécuter notre modèle. Pour ce faire, nous devons récupérer le fichier de sauvegarde avec le numéro d'étape le plus élevé. Les fichiers récapitulatifs sont généralement nommés `model.ckpt-XXXX` où `XXXX` est le numéro d'étape. A la fin de l'entraînement nous avons récupéré le fichier `model.ckpt-2000.meta`.

Nous allons ensuite créer un frozen graph TensorFlow avec des opérations compatibles pouvant être utilisées avec TensorFlow lite. Nous allons maintenant convertir le frozen graph (`tf_lite_graph.pb`) créé, au format TensorFlow Lite (`detect.tflite`). En fin de compte, nous obtenons un fichier `road_signs_quantized.tflite`, qui convient aux appareils mobiles et au processeur Raspberry Pi pour effectuer l'inférence de modèle, mais pas encore pour l'accélérateur Edge TPU.

Convertir le modèle au format Edge TPU

Possibilité de convertir le modèle au bon format avec `edge-tpu compiler` (18). Le problème est que cela ne fonctionne que sur des systèmes Linux Debian-based. Pour le faire fonctionner sur Windows je suis donc passé par docker (19).

Néanmoins j'ai rencontré un problème avec la commande de run me disant qu'il n'a pas accès au fichier pour le transformer (« Error opening file for reading »). J'ai donc décidé de passer par un docker-compose (Figure 12) qui va lire le Dockerfile du github.

```
version: '3'

services:
  myservice:
    build:
      context: C:/Users/utilisateur/Desktop/object_detection/docker-edgetpu-compiler
      dockerfile: Dockerfile
    image: result/latest
    volumes:
      - ./transform_model:/home/edgetpu
    command: tail -f /dev/null
```

Figure 12: Le fichier Docker Compose pour le edge tpu compiler

Les commandes à utiliser sont les suivantes:

- docker-compose up -d
- docker exec -it docker-edgetpu-compiler_myservice_1 bash

Une fois dans le container il suffit de lancer la commande suivante qui va nous créer le modèle (et les logs) au format compatible avec Edge TPU :

- edgetpu_compiler road_signs_quantized.tflite (Figure 13)

```
root@15c30b6d8d:/home/edgetpu: ls
road_signs_quantized.tflite
root@15c30b6d8d:/home/edgetpu: edgetpu_compiler road_signs_quantized.tflite
Edge TPU Compiler version 15.0.34023435

Model compiled successfully in 2434 ms.

Input model: road_signs_quantized.tflite
Input size: 4.57MiB
Output model: road_signs_quantized_edgetpu.tflite
Output size: 5.19MiB
On-chip memory used for caching model parameters: 5.01MiB
On-chip memory remaining for caching model parameters: 2.73MiB
Off-chip memory used for streaming uncached model parameters: 0.00B
Number of Edge TPU subgraphs: 1
Total number of operations: 99
Operation log: road_signs_quantized_edgetpu.log

Model successfully compiled but not all operations are supported by the Edge TPU. A percentage of the model will instead run on the CPU, which is slower. If possible, consider updating your model to use only op-
erations supported by the Edge TPU. For details, visit g.co/coral/model-reqs.
Number of operations that will run on Edge TPU: 98
Number of operations that will run on CPU: 1
See the operation log file for individual operation details.
```

Figure 13: Convertir le modèle au format Edge TPU

C'est le modèle converti (road_signs_quantized_edgetpu.tflite) que nous utiliserons dans notre Raspberry Pi pour que la voiture détecte les objets sur la route. La différence entre le fichier _edgetpu.tflite et le fichier .tflite normal est que, avec le fichier _edgetpu.tffile, toutes les inférences de modèle (99%) s'exécuteront sur le TPU Edge, au lieu du processeur de Pi. Pour des raisons pratiques, cela signifie que vous pouvez traiter environ 20 images de résolution 320x240 par seconde (aka. FPS, Frames Per Sec) avec l'Edge TPU, mais seulement environ 1 FPS avec le CPU Pi seul. 20 FPS est (presque) en temps réel.

Réponse à la détection

Maintenant que la voiture peut détecter et identifier les objets devant elle, nous devons toujours lui dire quoi en faire, c'est-à-dire le contrôle de mouvement. J'ai utilisé l'approche basée sur des règles ce qui signifie que nous devons dire à la voiture exactement ce qu'elle doit faire

lorsqu'elle rencontre chaque objet. Par exemple, dites à la voiture de s'arrêter si elle voit un feu rouge ou un piéton, ou conduire plus lentement si elle voit un panneau de limitation de vitesse inférieure, etc.

Puisque nous avons six types d'objets (feux rouges, feux verts, limite de 50 Km/h, limite de 30 Km/h, panneau stop, piétons), je vais illustrer comment gérer quelques types d'objets, et vous pouvez lire mon implémentation complète dans ces deux fichiers sur GitHub (14), `traffic_objects.py` et `object_on_road_processor.py`.

Les règles sont assez simples: si aucun objet n'est détecté, la voiture continue à la dernière limite de vitesse connue. Si un objet est détecté, cet objet modifiera la vitesse ou la limite de vitesse de la voiture.

Tout d'abord, nous allons définir une classe de base, `TrafficObject`, qui représente tous les panneaux de signalisation ou les piétons qui peuvent être détectés sur la route. Il contient une méthode, `set_car_state(car_state)`. Le dictionnaire `car_state` contient deux variables, à savoir `speed` et `speed_limit`, qui seront modifiées par la méthode. Il a également une méthode d'assistance, `is_close_by()`, qui vérifie si l'objet détecté est suffisamment proche. (Eh bien, comme notre seule caméra ne peut pas déterminer la distance, j'évalue la distance avec la hauteur de l'objet. Pour déterminer la distance avec précision, nous aurions besoin d'un Lidar par exemple).

```
1 class TrafficObject(object):
2
3     def set_car_state(self, car_state):
4         pass
5
6     @staticmethod
7     def is_close_by(obj, frame_height, min_height_pct=0.05):
8         # default: if a sign is 10% of the height of frame
9         obj_height = obj.bounding_box[1][1]-obj.bounding_box[0][1]
10        return obj_height / frame_height > min_height_pct
```

Les implémentations pour Red Light et Pedestrian sont alors évidentes il faut définir la vitesse de la voiture à 0 pour qu'elle s'arrête tant qu'elle détecte un piéton ou un feu rouge.


```

1  class RedTrafficLight(TrafficObject):
2      def set_car_state(self, car_state):
3          logging.debug('red light: stopping car')
4          car_state['speed'] = 0
5
6  class Pedestrian(TrafficObject):
7      def set_car_state(self, car_state):
8          logging.debug('pedestrian: stopping car')
9          car_state['speed'] = 0

```

Les limites de vitesse de 30 Km/h et 50 Km/h peuvent utiliser une seule classe SpeedLimit, qui prend speed_limit comme paramètre d'initialisation. Lorsque le signe est détecté, il faut régler la limite de vitesse de la voiture à la limite appropriée.

L'implémentation du feu vert est encore plus simple, car elle ne fait rien d'autre qu'une impression d'un feu vert est détectée. L'implémentation du panneau d'arrêt est la plus complexe, car elle doit garder une trace des états, ce qui signifie qu'elle doit se rappeler qu'elle s'est arrêtée au panneau d'arrêt depuis déjà quelques secondes, puis avancer même si les images vidéo suivantes contiennent un très grand panneau d'arrêt lorsque la voiture passe devant le panneau.

Une fois que nous avons défini le comportement de chaque panneau de signalisation, nous avons besoin d'une classe pour les lier ensemble, qui est la classe ObjectsOnRoadProcessor. Cette classe charge d'abord le modèle entraîné pour Edge TPU, puis détecte les objets dans la vidéo en direct avec le modèle et appelle enfin chaque objet de trafic pour modifier la vitesse et la limite de vitesse de la voiture. Le code complet se trouve dans le Github (14) sous objects_on_road_processor.py.

Notez que chaque TrafficObject change simplement la vitesse et speed_limit dans l'objet car_state, mais ne change pas réellement la vitesse de la voiture. C'est ObjectsOnRoadProcessor qui modifie la vitesse réelle de la voiture, après avoir détecté et traité tous les panneaux de signalisation et les piétons.

Base de données

Schéma de la base

Dans le cas de ce projet de voitures autonomes, le choix d'une base de données de type relationnelle a été fait pour mettre en avant les relations entre les données. Afin de concevoir notre base de données, j'ai utilisé l'outil DB designer (20) qui m'a permis de créer un schéma de la base de données désirées (Figure 14) et d'exporter ce schéma au format SQL pour pouvoir la créer.

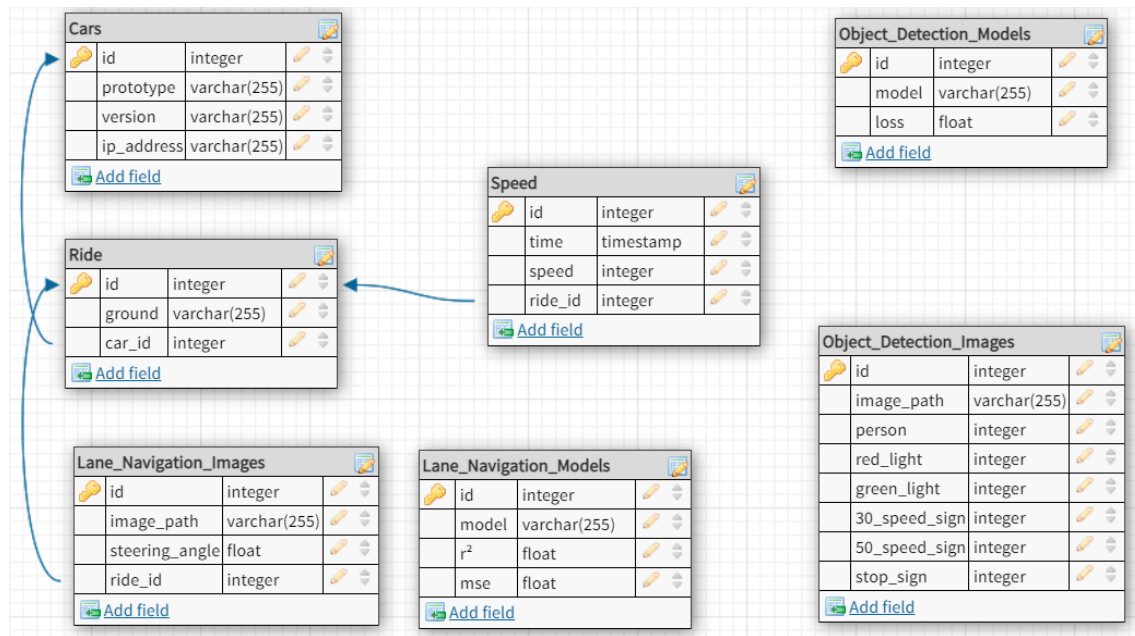


Figure 14 : Schéma de ma base de données.

La première table Cars va contenir les différents prototypes de voiture que nous testerons. Ces voitures seront associées à des trajets que l'on retrouvera dans la table Rides. Ces différents trajets seront définis par les images stockées dans la table Lane_Navigation_Images et nous enregistrerons la vitesse de la voiture lors de ces différents trajets. Nous avons ainsi établi des relations 1 to many entre ces quatre tables. En effet un trajet est en relation avec plusieurs images, une voiture est reliée à plusieurs trajets et un trajet est relié à plusieurs vitesses. La relation entre les tables est définie par la création des clés étrangères. Par exemple, la table Ride contient la colonne car_id qui correspond à la clé primaire de la table Cars.

Il s'agit des seules relations présentes dans ma base de données, en effet pour ce projet, je n'ai pas de relation 1 to 1 car je n'ai pas de cas où un id correspond à un autre id. Par exemple, entre une voiture et un trajet : si une voiture ne peut suivre qu'un seul trajet, et notre trajet ne peut pas être suivi par plusieurs voitures. Néanmoins ce n'est pas le cas, donc nous n'avons pas de relations 1 to 1.

Les table Object_Detection_Images et Lane_Navigation_Images, vont permettre de stocker les images nécessaires pour entraîner les modèles IA pour la régression et la classification/détection d'objets.

Nous aurons ensuite une table pour stocker le nom et l'évaluation des modèles IA utiliser pour la navigation (Lane_Navigation_Models) et la détection d'objet (Object_Detection_Models).

Création

La création de la base de données se fait via un docker compose (Figure 15), nous avons utilisé l'interface Adminer car elle est simple d'utilisation et d'installation:

```

version: '3.7'

services:
  database:
    image: mysql:latest
    container_name: database
    env_file:
      - .env
    networks:
      - database_net
    ports:
      - 3306:3306
    security_opt:
      - seccomp:unconfined
    volumes:
      - database_data:/var/lib/mysql
      - ./BDD/init-db:/docker-entrypoint-initdb.d
    command: --default-authentication-plugin=mysql_native_password
  adminer:
    image: dehy/adminer
    container_name: adminer
    networks:
      - database_net
    ports:
      - 8080:80

```

Figure 15: Docker compose pour créer la base de données

Depuis adminer, que l'on peut accéder au port 8000, il nous suffit d'importer le schéma SQL pour que la base de données soit créée.

Remplissage de la base de données

Le remplissage de la base de données se fait directement depuis le Jupyter Notebook, lors du chargement des images et de l'entraînement des modèles en ce qui concerne les tables d'images, de modèles et de vitesses (Figure 16). Il faut bien sûr se connecter à la base de données via create_engine de SQL Alchemy.

```
df_model.to_sql('Lane_Navigation_Models', con = engine, if_exists = 'append', chunksize = 1000)
```

Figure 16: Commande pour envoyer un dataframe pandas directement dans la base de données

Ainsi nous remplissons ces tables avec le chemin vers les images, les valeurs d'angles de braquages, les différents objets présents sur les images. Pour les modèles nous récupérons le nom du modèle et son évaluation. Pour éviter que les mêmes images soient rentrées plusieurs fois, la valeur image_path est unique. De fait, si le path existe déjà, l'image n'est pas insérée, mais la requête s'arrête net, ce qui ne permet pas d'envoyer les autres images dans la base de données. Pour cela j'ai créé cette requête (Figure 17) qui permet de vérifier si l'image existe déjà et si elle n'existe pas, alors on peut insérer dans la base de données.

```


for path in partial_paths:
    #print(path)
    connection= engine.raw_connection()
    cursor = connection.cursor()
    mci= cursor.execute(''SELECT EXISTS (SELECT 1 FROM Lane_Navigation_Images WHERE image_path=%s)''', (path,))
    result = cursor.fetchone()[0]
    print(result)
    if result == 0 :
        cursor.execute('' INSERT INTO Lane_Navigation_Images VALUES(%s,%s)''',(df.image_path, df.steering_angle))
    cursor.close()

```

Figure 17: Requête pour vérifier si l'image existe déjà avant l'insert

Les tables Cars et Rides sont remplies directement via la commande SQL similaire à l'exemple qui suit (Figure 18) que l'on peut lancer directement sur Adminer:

```

INSERT INTO  Rides (ground, car_id)
VALUES ('vinyl', '2');

```

Figure 18: Insérer des données

Sauvegarde de la base de données

Dans l'idée d'assurer la pérennité de nos données, nous allons sauvegarder notre base de données de façon régulière. Pour ce faire, plusieurs possibilités s'offrent à nous. La première consiste à utiliser la ligne de commande :

- Pour l'enregistrer depuis le container : `mysqldump -u deeppicar -p deeppicar --no-tablespaces`
- Pour l'enregistrer hors du container : `docker exec -t database mysqldump -u user -ppassword database > nom.sql`

Il est aussi possible de créer un mécanisme de sauvegarde automatique. Il existe plusieurs façons de faire. Avec Windows, par exemple, il est aussi possible de créer un script (.bat) qui pourra être lu par cmd. Ce script peut alors être lancé via le planificateur de tâches de façon qu'il se lance automatiquement (tous les lundis à midi par exemple). Dans mon cas, j'ai directement ajouté la sauvegarde de la base de données directement dans le docker compose (Figure 19), qui s'active à la date choisie selon CRON. La sauvegarde de la base de données est réalisée tous les Lundi à 12h00 et est enregistrée dans le dossier BDD/backup.

```

✓ ···mysql-cron-backup:
  ····container_name: db-backup
  ····image: fradelg/mysql-cron-backup
✓ ···depends_on:
  ····database
✓ ···volumes:
  ····./BDD/backup:/backup
✓ ···env_file:
  ····.env
  ···restart: unless-stopped
✓ ···networks:
  ····database_net
✓ networks:
✓ ···database_net:
  ···driver: bridge
  ···name: database_net
✓ volumes:
  ···database_data:

```

Figure 19: Sauvegarde de la base de données via CRON

Bien sûr ceci nous permettra en cas de soucis de récupérer notre base via ligne de commande :

- `mysql -u [user] -p [database_name] < backup.sql`

Il est aussi possible d'utiliser l'interface d'adminer pour exporter la base de données ou pour réimporter un backup sans passer par la ligne de commande.

Performance

Pour le moment toutes les requêtes réalisées sont très rapides comme nous pouvons le voir (Figure 20) en utilisant la commande EXPLAIN ANALYZE, nous voyons aussi où la requête passe du temps et pourquoi.

```

mysql> EXPLAIN ANALYZE SELECT `steering_angle` FROM Lane_Navigation_Images INNER JOIN Cars ON Lane_Navigation_Images.ride_id = Cars.id WHERE Cars.id=1;
+-----+
| EXPLAIN |
+-----+
| -> Index lookup on Lane_Navigation_Images using Lane_Navigation_Images_fk0 (ride_id=1) (cost=24.90 rows=219) (actual time=0.153..0.340 rows=219 loops=1) |
+-----+
| 1 row in set (0.00 sec) |
+-----+

```

Figure 20: Exemple de requête avec un JOIN et EXPLAIN ANALYZE

L'optimisation de la base de données n'est donc pas nécessaire pour l'instant au vu de la rapidité des requêtes. Il est possible que plus nous allons ajouter des données plus les requêtes deviendront lentes. Il sera alors peut être intéressant de créer des index sur la base de données. L'instruction CREATE INDEX est utilisée pour créer des index dans les tables. Les index sont utilisés pour récupérer les données de la base de données plus rapidement qu'autrement. Les utilisateurs ne peuvent pas voir les index, ils sont juste utilisés pour accélérer les recherches / requêtes.

API Web

Pourquoi?

La dernière étape de ce projet a été de créer une API Web. L'idée était de créer une API Web permettant de regrouper des statistiques et des données intéressantes sur notre voiture. Pour cela j'ai cherché une API Web permettant de faire de la visualisation à partir de données stockées dans une base de données SQL. En faisant quelques recherches je suis tombé sur Dash (21).

Écrit au-dessus de Flask, Plotly.js et React.js, Dash est idéal pour créer des applications de visualisation de données avec des interfaces utilisateur hautement personnalisées en Python pur. Il est particulièrement adapté à quiconque travaille avec des données en Python.

Grâce à quelques modèles simples, Dash fait abstraction de toutes les technologies et protocoles nécessaires pour créer une application Web interactive. Dash est suffisamment simple pour pouvoir lier une interface utilisateur autour de votre code Python en un après-midi.

Dash + SQL

Via un code python nous allons nous connecter à notre base de données SQL avec la librairie sqlalchemy (Figure 21). Bien sûr pour cela, il faut que nous ayons lancé notre docker compose pour que la base de données soit accessible à la connexion.

```
from dotenv import load_dotenv
from sqlalchemy import create_engine

# Get the authentication informations from env file
path='C:/Users/utilisateur/Desktop/Voiture_Autonomie/bdd/.env'

load_dotenv(dotenv_path=path)
user=os.getenv("MYSQL_USER")
password=os.getenv("MYSQL_PASSWORD")
host=os.getenv("MYSQL_HOST")
database=os.getenv("MYSQL_DATABASE")
port=os.getenv("MYSQL_PORT")

# Connect to my sql database
engine = create_engine("mysql+pymysql://{user}:{pw}@localhost/{db}"
                        .format(user=user,
                                pw=password,
                                db=database))

cnxn = engine.connect()

# Get the data for the graph by requesting sql
df_angle = pd.read_sql("SELECT steering_angle FROM Lane_Navigation_Images", cnxn) # WHERE ride_id = 1
df_lane_nav = pd.read_sql("SELECT * FROM Lane_Navigation_Models", cnxn)
df_object_detection = pd.read_sql("SELECT 'person', 'red_light', 'green_light', '30_speed_sign', '50_speed_sign', 'stop_sign' FROM Object_Detection_Images", cnxn)
df_join_car = pd.read_sql("SELECT 'steering_angle' FROM Lane_Navigation_Images INNER JOIN Cars ON Lane_Navigation_Images.ride_id = Cars.id WHERE Cars.id=1", cnxn)
df = pd.read_csv("C:/Users/utilisateur/Desktop/Voiture_Autonomie/temps_frame.csv")
df_trajet = pd.concat([df_join_car, df], axis=1)
df_speed = pd.read_sql("SELECT * FROM Speed", cnxn)
```

Figure 21 : Connexion et Requête SQL

A partir des données récupérées avec les différentes requêtes SQL, nous allons pouvoir créer des visualisations grâce à plotly. Il nous reste alors plus qu'à lancer le script python qui nous fournira un lien vers la page web:

- <http://127.0.0.1:8050/>

En suivant le lien, nous arrivons sur la page web suivante (Figure 22) qui contient les différentes visualisations plotly qui nous intéressent. Nous avons notamment la distribution des angles de braquages, la trajectoire et la vitesse de la voiture.

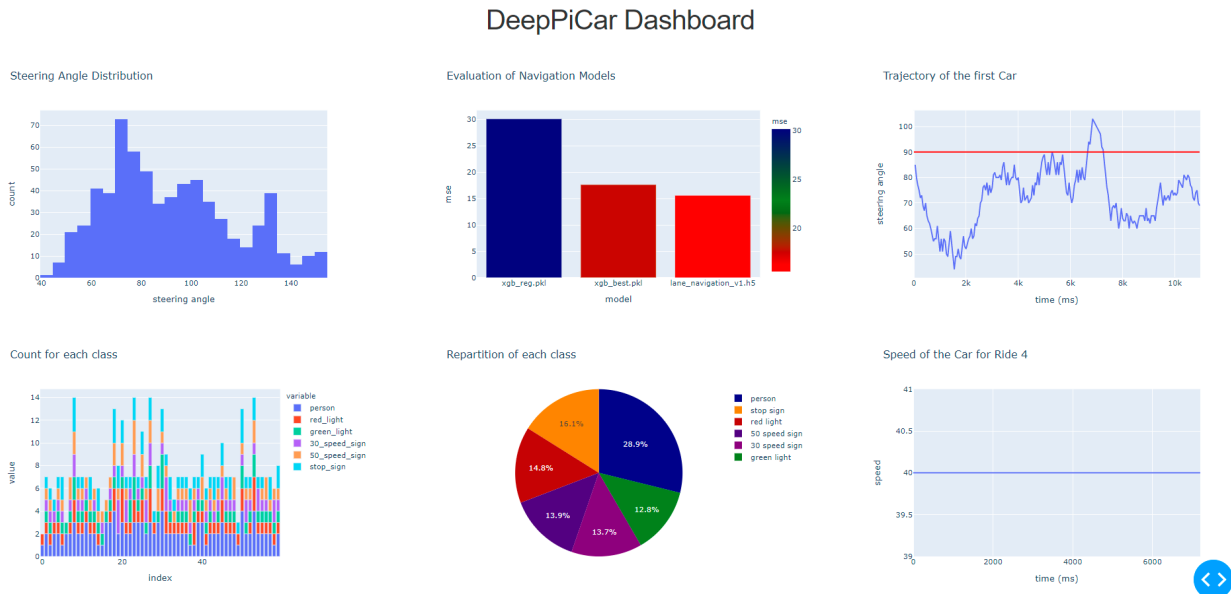


Figure 22: Dashboard de notre voiture

Conclusion

A la fin de ce projet, nous avons donc une voiture robotique autonome via deep learning qui peut suivre une route et peut s'arrêter quand elle repère un piétons ou un feu rouge. Ce n'est pas une mince affaire, car la plupart des voitures sur la route ne peuvent pas encore le faire. Nous avons pris un raccourci et utilisé un modèle de détection d'objet pré-entraîné et appliqué du transfert learning. En effet, l'apprentissage par transfert est répandu dans l'industrie de l'IA lorsque l'on ne peut pas rassembler suffisamment de données d'entraînement pour créer un modèle d'apprentissage en profondeur à partir de zéro ou que l'on n'a pas assez de puissance GPU pour entraîner des modèles pendant des semaines ou des mois.

Ce projet m'a permis de réaliser que j'aime construire des robots et les voir évoluer petit à petit. J'ai donc quelques idées pour la suite pour continuer à améliorer cette voiture. Pourquoi pas essayer via l'installation d'un micro d'avoir de la reconnaissance vocale, notamment pour la démarrer ou l'arrêter (Hello Batman !). Il serait intéressant aussi de rajouter un capteur de style lidar pour détecter les distances entre les objets et la voiture.

Bibliographie

1. <https://deepracing.com/index.html>

2. <https://fr.smartsheet.com/welcome-customers-home>
3. <https://www.sunfounder.com/>
4. <https://www.robotshop.com/media/files/content/s/suf/pdf/sunfounder-smart-video-car-for-raspberry-pi-4-3-2-b.pdf>
5. <https://projects.raspberrypi.org/en/projects/raspberry-pi-setting-up>
6. https://www.youtube.com/watch?v=IDqQIDL3LKg&list=PLQVvaa0QuDesV8WWHLLXW_avmTzHmJLv&index=4
7. <https://raspberry-pi.fr/transferer-fichiers-raspberry-ssh/>
8. <https://coral.ai/docs/accelerator/get-started/#1-install-the-edge-tpu-runtime>
9. <https://www.tensorflow.org/lite/guide/python>
10. <https://github.com/dctian/DeepPiCar/tree/master/driver/code>
11. <https://arxiv.org/abs/1604.07316>
12. [https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/](https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md)
[tf1_detection_zoo.md](https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md)
13. https://medium.com/@sanguhuynh_73086/how-to-install-labelimg-in-windows-with-anaconda-c659b27f0f
14. https://github.com/YohanCaillau/projet_chef_oeuvre
15. <https://towardsdatascience.com/google-coral-edge-tpu-board-vs-nvidia-jetson-nano-dev-board-hardware-comparison-31660a8bda88>
16. <https://teyou21.medium.com/setup-tensorflow-for-object-detection-on-ubuntu-16-04-e2485b52e32a>
17. <https://www.dlology.com/blog/quick-guide-to-run-tensorboard-in-google-colab/>
18. <https://coral.ai/docs/edgetpu/compiler/#system-requirements>
19. <https://github.com/tomassams/docker-edgetpu-compile>
20. <https://app.dbdesigner.net/dashboard>
21. <https://dash.plotly.com/>