AI & ROBOTICS

Stop sign detection project

LIENS UTILES

DESCRIPTION	LINK
Github - Object Detection Robotics	LINK

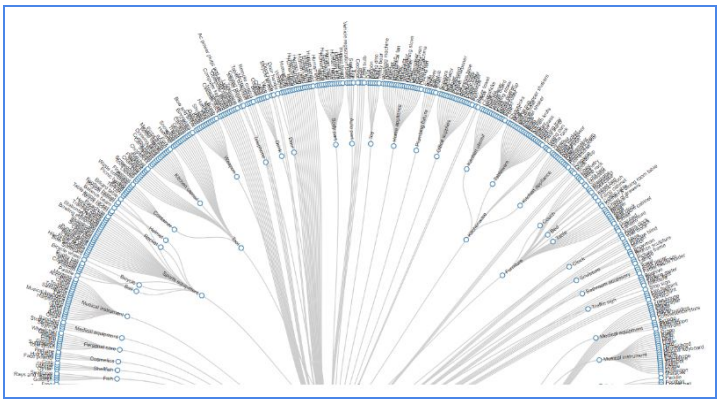
SOMMAIRE

INTRODUCTION

Lors de ce projet nous avons mis en place des algorithmes capables de **détecter des panneaux de stop.** La détection des panneaux est un élément important de la robotique et du véhicule autonome. En plus de **détecter** un panneau de stop, nous avons par la même occasion **entraîné** notre modèle, afin qu'il puisse reconnaître d'autres éléments comme : des **voitures** et des **humains.**

RÉCUPÉRATION DE LA DONNÉE

Open Image Dataset V4 API



Nous avons utilisé l'API **Open Image Dataset v4** *(OIDv4)* pour récupérer quasiment tous nos jeux de données. C'est un dataset open source qui est composé de **600 classes** *(~1.7 Millions d'images)* pour la détection d'objet et ~20000 classes pour de la classification d'images *(labels fournis avec les classes).*

Nous avons donc choisi d'y récupérer les classes suivantes : **Stop_sign**, **Car** & **Person.**
Une fois le git repository récupéré et l'environnement adéquat créé, voici la commande que nous avons utilisé pour récupérer l'un de nos datasets : **Python main.py --classes Person --type_csv train --limit 2500** *(cf. l'image du terminal ci-dessous).*

Pour récupérer l'ensemble des données des différentes classes, il est possible de mettre des classes supplémentaires en ajoutant l'option **--classes** *(attention à la casse)*, le dataset est déjà **split en trois catégories** *(train, validation & test)* et il est possible de prendre tout ou une parti du dataset.

Voici la commande pour récupérer tous les datasets en une seule fois :
python main.py downloader --classes Stop_sign Car Person --type_csv all

COCO Dataset API

Nous avons essayé de diversifier nos données en nous tournant sur le célèbre **COCO dataset** qui dispose de **80 classes** *(~330 000 images)* de détections d'images *(fourni avec les annotations).* Le dataset étant moins bien organisé que l'OIDv4, nous avons choisi d'utiliser la **library pycocotools** pour chercher dans le fichier **instances_annotation2017.json** *(provenant des annotations)* tous les éléments avec la classe "stop sign". Pour chaque photo avec la bonne classe, nous obtenons une **url** qui nous redirige vers une image, que nous **téléchargeons en local.** Nous avons principalement utilisé ce script pour récupérer plus d'images de panneaux stop, car nous n'en avions pas assez avec OIDv4.

Pour la récupération des **modèles pré-entraînés** et leur entraînement avec nos custom datasets, nous sommes passés par le tutoriel de Tensorflow :
https://tensorflow-object-detection-api-tutorial.readthedocs.io

TRAITEMENT DE LA DONNÉE

Avant d'utiliser le data preprocessing intégré dans le pipeline des modèles pré-entraînés, nous avons fait du **data preprocessing "maison"** : rename des classes, resize des images, resize des labels & création d'un csv avec toutes nos images, leurs caractéristiques propres associé à leur annotations *(bounding boxes).*

Resize images

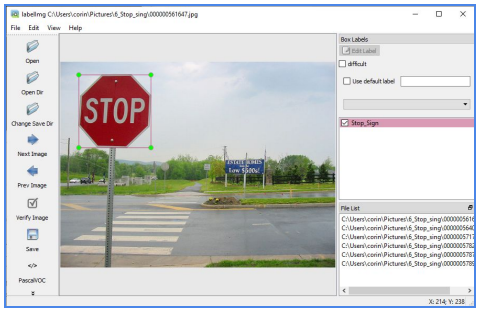
Pour que l'entraînement de notre modèle ne soit pas trop long, nous avons décidé de **redimensionner nos images,** pour cela nous avons créé un script pour mettre nos images en **640x640 pixels.** Nous avons par la suite redimensionné les annotations *(les bounding boxes)* des images *(panneaux stop, voitures, etc ...),* afin qu'elle soit à la bonne dimension pour correspondre avec nos nouvelles tailles d'images.

Le Script est **scalable**, si jamais nous voulons rajouter d'autres objets, il les normalisera alors en 640x640.

Labellisation manuelle

Pour certaines classes, nous n'avions pas assez de données donc nous avons donc dû croiser différents datasets *(OIDv4 & COCO).* Par exemple, pour les panneaux stop, nous avons dû récupérer à l'aide d'un script uniquement les panneaux stop du dataset COCO, puis nous avons défini les bounding boxes grâce à labelImg.

Les annotations n'étaient plus du même type *(texte pour OIDv4 dataset et XML pour labelImg).* Nous avions environ 400 images annotées grâce au dataset d'OIDv4 et 600 images annotées manuellement, afin d'obtenir un total de **1000 images annotées.**



Label Img

DATA AUGMENTATION

L'un des gros problèmes de la computer vision est le **manque de données flagrant.** Pour chaque problématique, nous ne disposons pas encore d'assez de dataset avec **suffisamment** de données. Nous faisons alors recours à ce que l'on appelle la **Data Augmentation.** Avec un nombre d'images finis, nous sommes capables d'en **créer de nouvelles à la volée** pendant l'entraînement du modèle.

Do It Yourself

Pour obtenir plus d'images dans notre dataset, nous avons simplifié la tâche. Pour augmenter le nombre d'éléments, nous avons utilisé des **modifications de teintes** sur les images en ajoutant des **filtres.** Dans notre cas, nous n'avions pas besoin de changer les bounding boxes à nouveau. Il est rare, voire peu probable, de rencontrer des voitures, des humains ou des panneaux stop **à l'envers** ou à plus de **15° d'angles.** Pour cela nous avons créé un script, afin de convertir nos images en **nuances de gris** *(greyscale)* et une version **légèrement flouté** de nos images à l'aide d'un filtre blur.

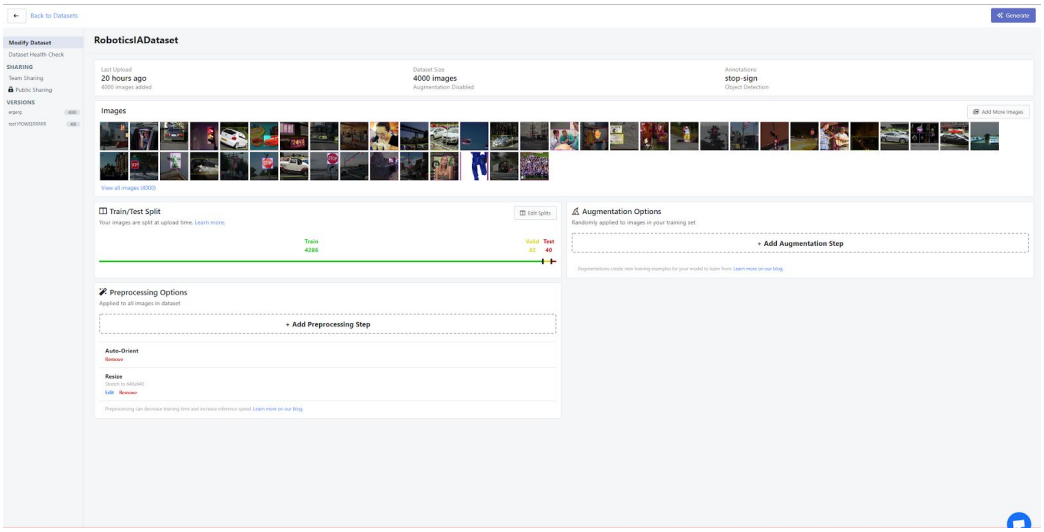
Standard

Flou

Inversé

Gris

Site tier



Nous avons utilisé le site **Roboflow.com** pour unifier les annotations et générer un fichier **TF.records** & **labelmap.pbtxt**

ENTRAÎNEMENT DES MODÈLES

La pipeline d'entraînement

Il suffit de rédiger un **fichier de configuration** pour la quasi-totalité du pipeline: du **pre-processing** du dataset au post-processing en passant par la **sélection du modèle** à entraîner et les **hyperparamètres** utilisés, voici un extrait :

```
ssd {
  num_classes: 4
  image_resizer {
    fixed_shape_resizer {
      height: 640
      width: 640
    }
  }
  feature_extractor {
    type: "ssd_resnet50_v1_fpn_keras"
    conv_hyperparams {
      regularizer {
        l2_regularizer {
          weight: 0.00039999998989515007
        }
      }
```

Critères de sélection

Les deux critères utilisés pour sélectionner les modèles à entraîner dans notre cas ont été :

- Une **bonne vitesse d'exécution** pour l'inférence. Nous avons arbitrairement défini le seuil optimal à 50ms et maximal à 100ms.
- De bonnes **performances de détection** en utilisant la métrique **mAP** *(mean Average Precision).*

Ces deux critères s'opposent en général et demandent de faire un **compromis**. Nous nous sommes basés sur les modèles exposés dans ce tableau pour choisir lesquels entraîner.

Entrainement

Nous avons choisi **5 modèles** pour faire de la classification sur les panneaux, les personnes *(piétons)* et les voitures *(ou autres véhicules présents sur la route).* Voici les informations récoltées avant entraînement :

Nom	Temps d'execution (ms)	Performance (mAP)	Description
EfficientDet D0 (512x512)	39	33.6	Architecture de détection d'objets conçue pour améliorer l'efficacité des modèles en vision par ordinateur, elle s'appuie sur EfficientNet, un

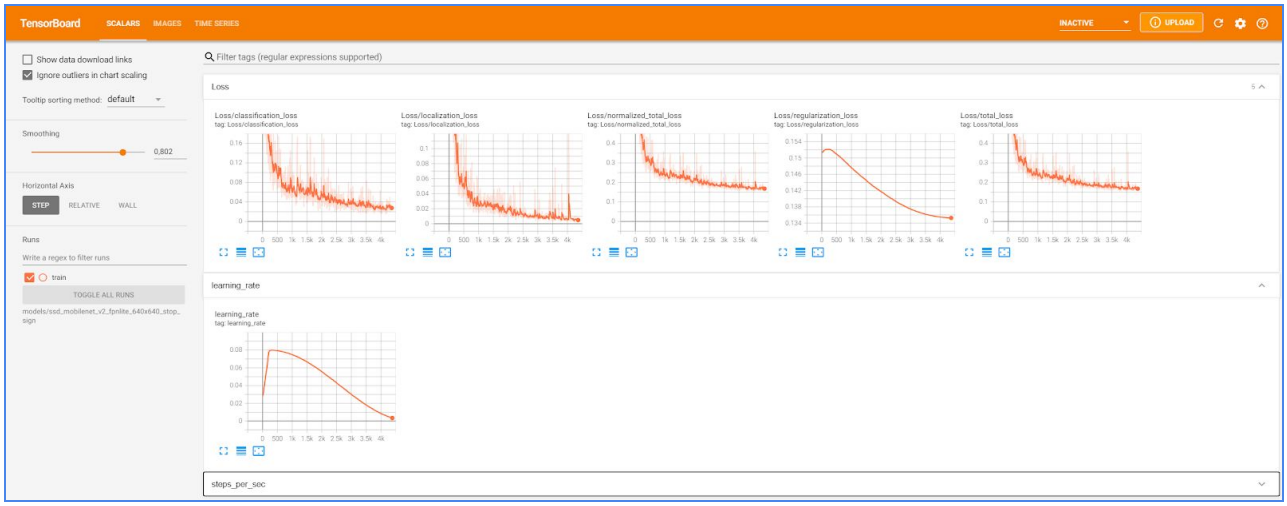
EfficientDet D1 (640x640)	54	38.4	réseau neuronal convolutif (CNN) qui est préformé sur la base de données d'images d'ImageNet pour la classification et qui propose plusieurs niveaux de complexités allant de D0 à D7.
SSD ResNet50 V1 FPN (640x640)	46	34.3	Abréviation de Residual Network, ce CNN est composé de 50 couches et comporte des sauts de connexion entre certaines couches.
SSD MobileNet V2 FPNLite (320x320)	22	22.2	Lancée en 2017, cette architecture a la particularité d'être légère par rapport à ses concurrents et en fait un candidat de choix pour les systèmes embarqués.
SSD MobileNet V2 FPNLite (640x640)	39	28.2	

```
INFO:tensorflow:Step 16700 per-step time 0.955s loss=0.313
I0124 17:30:24.253421  8784 model_lib_v2.py:648] Step 16700 per-step time 0.955s loss=0.313
INFO:tensorflow:Step 16800 per-step time 0.970s loss=0.330
I0124 17:32:00.475121  8784 model_lib_v2.py:648] Step 16800 per-step time 0.970s loss=0.330
INFO:tensorflow:Step 16900 per-step time 0.935s loss=0.349
I0124 17:33:36.677199  8784 model_lib_v2.py:648] Step 16900 per-step time 0.935s loss=0.349
INFO:tensorflow:Step 17000 per-step time 0.941s loss=0.280
I0124 17:35:12.460366  8784 model_lib_v2.py:648] Step 17000 per-step time 0.941s loss=0.280
INFO:tensorflow:Step 17100 per-step time 0.977s loss=0.325
I0124 17:36:49.001514  8784 model_lib_v2.py:648] Step 17100 per-step time 0.977s loss=0.325
INFO:tensorflow:Step 17200 per-step time 0.980s loss=0.277
I0124 17:38:25.039209  8784 model_lib_v2.py:648] Step 17200 per-step time 0.980s loss=0.277
INFO:tensorflow:Step 17300 per-step time 0.986s loss=0.258
I0124 17:40:00.708871  8784 model_lib_v2.py:648] Step 17300 per-step time 0.986s loss=0.258
INFO:tensorflow:Step 17400 per-step time 0.941s loss=0.270
I0124 17:41:36.906490  8784 model_lib_v2.py:648] Step 17400 per-step time 0.941s loss=0.270
INFO:tensorflow:Step 17500 per-step time 0.915s loss=0.282
I0124 17:43:12.795427  8784 model_lib_v2.py:648] Step 17500 per-step time 0.915s loss=0.282
INFO:tensorflow:Step 17600 per-step time 0.974s loss=0.265
I0124 17:44:48.429663  8784 model_lib_v2.py:648] Step 17600 per-step time 0.974s loss=0.265
INFO:tensorflow:Step 17700 per-step time 0.958s loss=0.255
I0124 17:46:24.267884  8784 model_lib_v2.py:648] Step 17700 per-step time 0.958s loss=0.255
INFO:tensorflow:Step 17800 per-step time 0.947s loss=0.235
I0124 17:48:00.759961  8784 model_lib_v2.py:648] Step 17800 per-step time 0.947s loss=0.235
INFO:tensorflow:Step 17900 per-step time 0.929s loss=0.336
I0124 17:49:36.455954  8784 model_lib_v2.py:648] Step 17900 per-step time 0.929s loss=0.336
INFO:tensorflow:Step 18000 per-step time 0.988s loss=0.295
I0124 17:51:12.639859  8784 model_lib_v2.py:648] Step 18000 per-step time 0.988s loss=0.295
```

Exemple des logs en temps réel générés lors d'un entraînement.

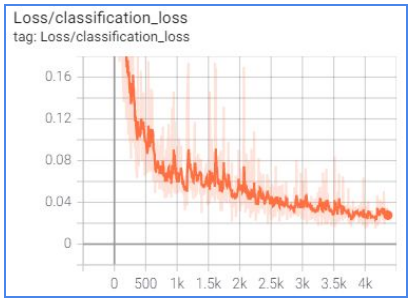
Tensorboard

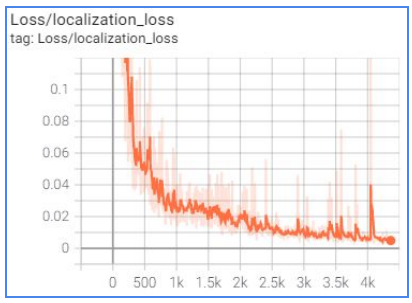
Pour suivre et confirmer le bon entraînement des modèles, nous avons utilisé **TensorBoard** *(suivi des fonctions de coût, des metrics, exemples d'images, etc.)* dont voici un exemple ci-dessous :

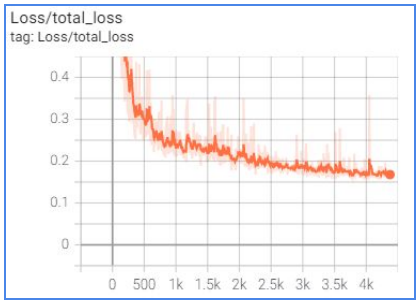


Fonction de coût

Pour la **fonction de coût**, nous précisons qu'il s'agit d'une **combinaison** de deux fonctions : l'une concernant la **localisation de la bounding box** autour d'un objet et l'autre concernant la bonne **classification** dudit objet.

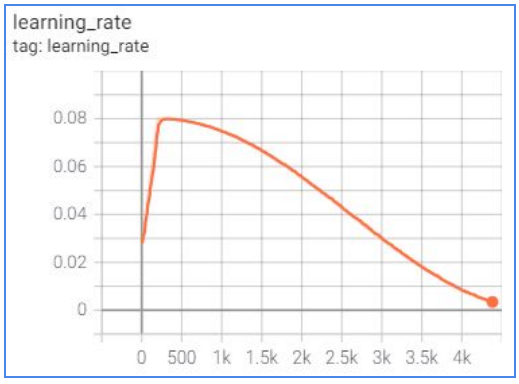






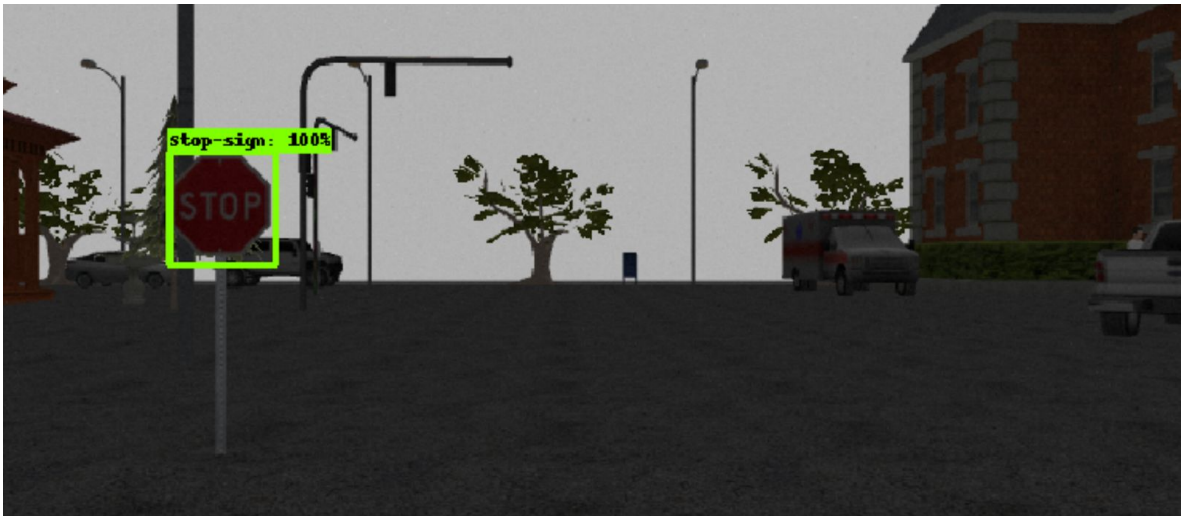
Evolution du taux d'apprentissage

Configuré dans le pipeline, on peut voir que le **taux d'apprentissage** (learning rate) évolue au fur et à mesure de l'entraînement. On **commence par l'augmenter** pour le sortir d'un potentiel extremum local, **puis** on le **rediminue** pour améliorer l'entraînement et éviter de stagner autour d'un autre extremum. C'est une pratique courante dans l'entraînement de modèles de Deep Learning.



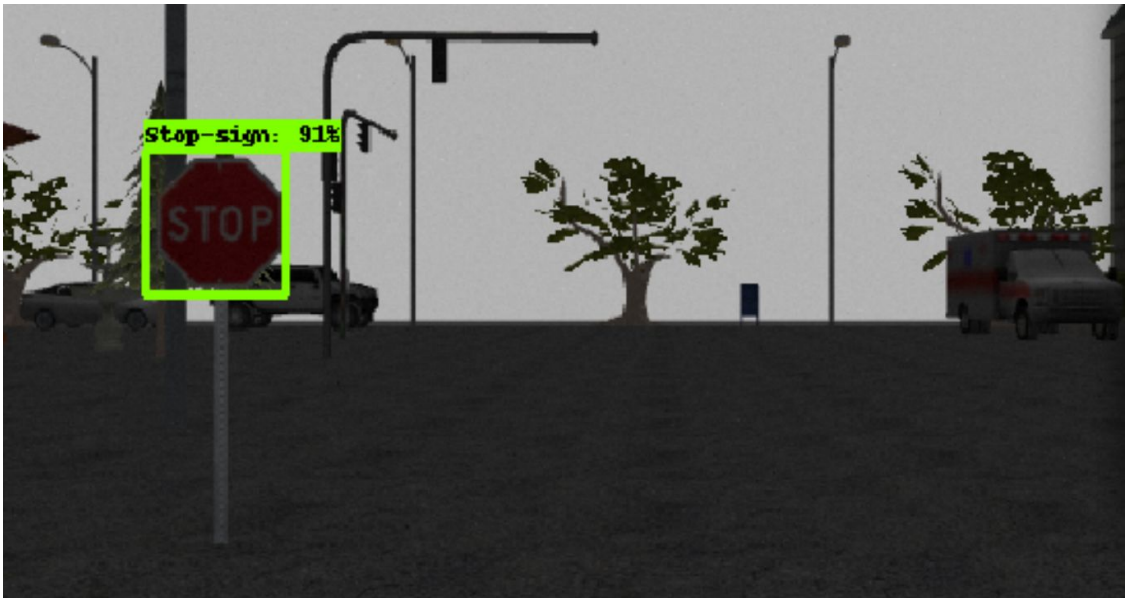
Choix du modèle

Suite à des erreurs non résolues sur le **script d'évaluation**, nous avons décidé de tester et comparer nos modèles directement via le roomba. En temps normal, il serait plus pertinent de les comparer sur la base d'une ou plusieurs **métriques simples.** Ici nous avons dû nous contenter des **scores de confiance** sur les panneaux dans l'environnement de ROS.



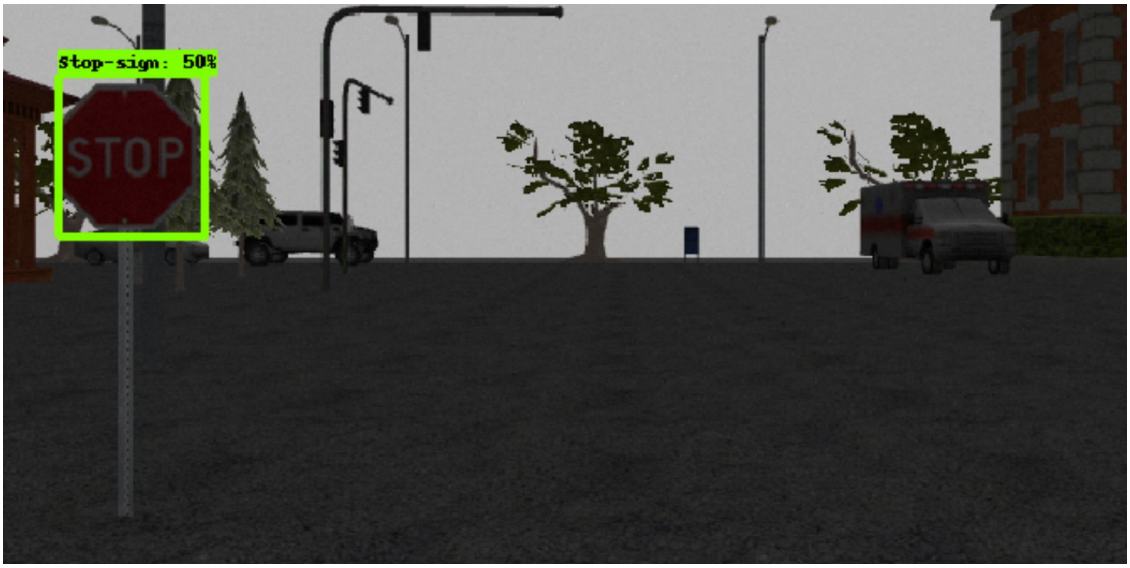
Took 0.35497426986694336 seconds

MobileNet V2 *(~0.35s/img)*



Took 0.9602630138397217 seconds

EfficientDet D1 *(~0.9s/img)*

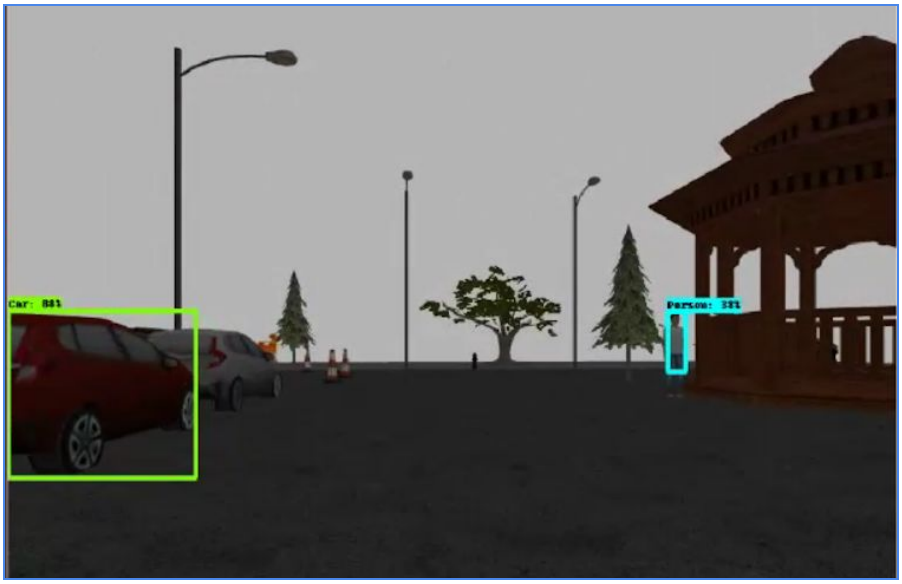


ResNet 50 *(~0.55s/img)*

Nous avons donc choisi le modèle **MobileNet v2,** car il avait les meilleurs scores et détectait plus souvent les panneaux en faisant moins d'erreurs *(faux positifs et faux négatifs).*

Détection de plusieurs classes

Nous avons également commencé des entraînements pour détecter **d'autres classes** *(personne, voiture).* Sans avoir de modèle vraiment efficace, nous avons eu des **résultats très encourageants.**



EXPORTATION & INTÉGRATION DU MODÈLE DANS ROS

Une fois le modèle entraîné, nous avons utilisé le script fourni par **l'API de Object Detection**, il permet **d'exporter** dans un dossier le modèle avec l'extension **.pb**

Nous avons suivi le tuto dans lequel on nous explique comment **charger le modèle** en mémoire et faire des **inférences.** Nous avons appliqué la procédure dans **l'*init*** de la **classe *Vision*** et nous l'avons stockée dans une de ses propriétés.

Par la suite, nous avons rajouté les labels des classes à détecter via un fichier .pbtxt.
La méthode *img_cb* est alors appelée dès qu'une **image est prise par la caméra du robot.** Il suffit de transformer l'image et de l'envoyer dans le modèle qui renvoie les positions des **Bounding Boxes** et les **labels** des objets détectés, que l'on ajoute à l'image.

CONCLUSION

Pour résumer nous avons pu :

- Récupérer de la donnée depuis plusieurs sources externes.
- En labelliser en plus pour équilibrer les classes.
- Utiliser de la Data Augmentation pour simuler une plus grande variété de données.
- Définir des critères de sélection pertinents pour notre problème.
- Entraîner des modèles via des scripts et des pipelines pour simplifier la procédure.
- Surveiller l'entraînement via TensorBoard.
- Vérifier les modèles entraînés sur de la vraie donnée (générée par le robot).
- Intégrer le modèle du MobileNet v2 à l'environnement du robot.
- Faire tourner la simulation et vérifier que le robot suit le chemin et détecte les panneaux en direct.
- Aller plus loin en intégrant plus de classes à détecter au modèle.