Sensory Group: Recognition

Dipalma Alessandro, Petix Marco, Ristori Alessandro, Rizzo Simone, Silvestri Federico
Master Degree in Computer science.
a.dipalma9@studenti.unipi.it, m.petix@studenti.unipi.it, a.ristori5@studenti.unipi.it
s.rizzo14@studenti.unipi.it, f.silvestri10@studenti.unipi.it.
Smart Applications, Academic Year: 2021/2022
Date: 15/01/2022
`https://github.com/unipi-smartapp-2021/sensory-cone-detection`

Contents

1 Task

Cone detection is an extremely fundamental task for an autonomous driving system since the car must move in accordance with what it receives from both networks in the pipeline. For this reason we thought that it was essential to explain what models we decided to work on and how to perform training and inference with it.

As we can see from the pipeline, there are two dataflows: one for the stereocamera and one for the LiDAR. We will document the creation of the two datasets and, when present, we will describe the preprocessing and postprocessing steps surrounding the interaction of the sensors data with the two models responsible of the cone detection.

Even though we obtained satisfying results, we will show some issues that arose during the development and, when possible, their potential solutions by future improvements that we could not implement due to time constraints.

2 Implementation

In this section we'll talk about why we chose a specific model and how to work with it, then we'll show our workflow with both sensors.

2.1 YOLOv5

We'll start with a brief introduction to the model we utilized for the cone detection process implemented for both the stereocamera and LiDAR sensors.

YOLOv5, being the state-of-the-art model for object detection tasks and maintained by a very active community, was our main choice for the stereocamera's workflow. The LiDAR's cone detection workflow, instead, didn't previously include a model, the e-team implemented the task through outdated methods like image segmentation, we therefore decided to make use of YOLOv5 again in this regard.

The chosen model was a small pretrained version of YOLOv5, higly suggested by the YOLOv5 team itself; said version better satisfied our constraints associated to the real-time environment of the task due to its good tradeoff between inference speed and performances on the simulator. Figure 1 shows a comparison between the various YOLOv5 models, our choice was also based on what we've seen from that graph.

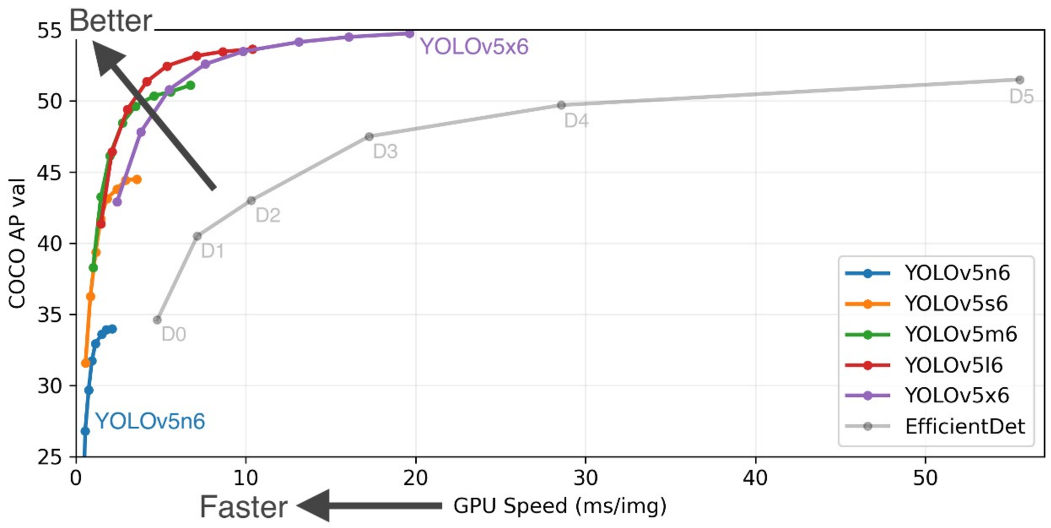


Figure 1: Comparison between some YOLOv5 models.

2.2 Tutorial for the model

Before working with the project, remember to clone the sensory group repository.

```
git clone --recursive https://github.com/unipi-smartapp-2021/sensory-cone-detection
```

Clone the official yolov5 repository too, or just use the model from the *sensory/scripts/yolov5* directory.[1]

```
git clone https://github.com/ultralytics/yolov5
```

Install the model requirements for working correctly.

```
pip install -r yolov5/requirements.txt
```

Managing the dataset Upload the dataset in the apposite directories following what is suggested in the official documentation from the YOLO team, section 1.3, or use *manage_dataset.py* to let it do the entire work.

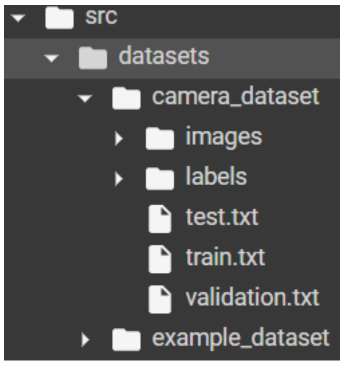
```
python manage_dataset.py --dataset dataset --path path_to_your_dataset_directory
```

For the previous command you can use the following arguments:

- **--dataset**: where the non-split dataset is located.
- **--path**: where to save the split dataset.

[1]**Warning**: the local YOLOv5 model may not work if it is not up to date with the latest version.

- **--copy**: copy the records, if not defined it will move each record from the source directory to the target one.
- **--grayscale**: convert the images into grayscale ones.
- **--split**: split the dataset into training, validation and test sets (default is 80%, 10% and 10%).



```
src
  datasets
    camera_dataset
      images
      labels
      test.txt
      train.txt
      validation.txt
    example_dataset
```

Figure 2: The src directory after running the *manage_dataset.py* command.

Each image has one text file with a single line for each bounding box. The format of each row is

$$\langle class\_id, center\_x, center\_y, width, height \rangle \tag{1}$$

where fields are space delimited, and the coordinates are normalized from zero to one.

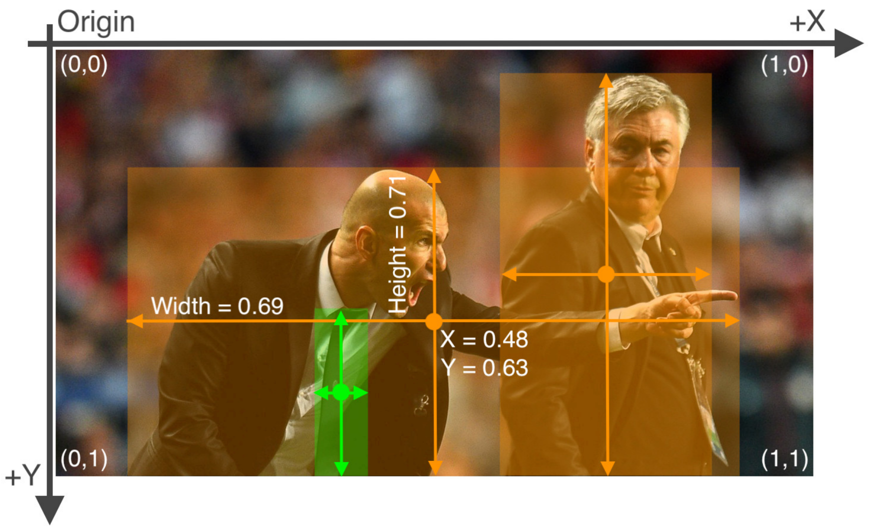


Figure 3: How YOLOv5 fields work.

The datasets used during the development resulted too big to be uploaded on GitHub, therefore we opted for hosting the datasets on a Google Drive shared by the group. We encourage to build your own dataset if you're interested in implementing the model.

After splitting the dataset into the usual train, validation and test splits, the latter being not strictly mandatory for YOLOv5, three more files, naturally created by *manage_dataset.py*, should be manually created if lacking. They are, respectively: *train.txt*, *validation.txt* and *test.txt*, and contain the paths to the respective splits of the dataset within them.

Setting up the YAML file Modify the *.yaml* file to your needs: this is required to work with YOLOv5 on a custom dataset. The following code refers to the *.yaml* file required by YOLOv5 to work with the images coming from the stereocamera.

```
# Train/val/test sets as 1) dir: path/to/imgs, 2) file: path/to/imgs.txt,
# or 3) list: [path/to/imgs1, path/to/imgs2, ..]
path: [path_to_your_camera_dataset_directory] # dataset root dir
train: train.txt # train images (relative to 'path')
val: validation.txt # val images (relative to 'path')
test: test.txt # test images (optional)
# Classes
nc: 4 # number of classes
names: ['big','orange','yellow','blue'] # class names
```

Importing the model It's recommended by the YOLOv5 team to work with a pretrained model and not building it from scratch due to how this would require to configure it entirely.

The weights of the various pretrained models (base, small, large etc.) can be found on the release page from the YOLOv5 team. The following is an example of importing one of them:

```
import requests
url = 'https://github.com/ultralytics/yolov5/releases/download/v6.0/yolov5s.pt'
r = requests.get(url, allow_redirects=True)
open('yolov5s.pt', 'wb').write(r.content)
```

Training To train the model is strongly suggested to follow this guide. The main arguments of the *train.py* method are:

- **--img**, image size.
- **--batch**, batch size.
- **--epochs**, number of epochs.
- **--data**, the yaml file.
- **--weights**, the model weights.
- **--cache**, to save a cache file of the train and validation sets.

- **--project**, where to save the results.

```
python yolov5/train.py --img 640 --batch 16 --epochs 10
    \ --data path/to/your_yaml.yaml --weights yolov5s.pt
```

Use **--cache** to cache the train and validation sets before starting the training, also, use **--project** to specify where to save the results. The directory used to store the results will also contain the last and best weights for the model.

Evaluation The evaluation on labeled data, either on the validation set or the test one, as specified on the *.yaml* file, is done by using *val.py*.

```
python yolov5/val.py --weights weights_path.pt --img 640 --conf 0.5
    \ --data yaml_path.yaml --task test
```

At the end of the training, the freshly computed weights should be in */runs/train/exp/weights*, otherwise, **--weights** needs to receive the path of the weights to be used during evaluation.

The *val.py* argument **--task** defines the split to be used for the evaluation of the model depending on the values *test* or *val*.

Inference *detect.py* is instead the method used when dealing with non-labeled data, it is therefore used during the various driverless competitions.

```
python yolov5/detect.py --weights best.pt --img 640
    \ --conf 0.5 --source path/to/your_image.png
```

For both *val.py* and *detect.py*, the results are stored by using **--project**. Both methods support the following arguments:

- **--save-txt**, to save the labels (useful to locate the cones);
- **--nosave**, to stop the model from saving the labeled image;
- **--exist-ok**, to store the results for all of the images within the same directory;
- **--project**, to define where to save the results;
- **--hide-labels**, to not show the labels on the detected cones;
- **--hide-conf**, to not show the confidence on each detected cone.

Use the model from the PyTorch hub It is also possible to load the trained model from PyTorch Hub and directly use it to perform inference on custom data[2]. The following snippet displays how to perform such endeavour:

[2]More details on this guide.

```
import torch
model = torch.hub.load('ultralytics/yolov5', 'custom', path='your_weights')
results = model('path_to_image')
```

Such method, however, was not used on the simulator since it was not clear if an internet connection would be available on the actual car. Ultimately, we decided to load the model from a local directory in order to overcome the aforementioned problem and speedup the process.

2.3 StereoCamera Object Detection

Dataset creation The dataset used for the stereocamera is a polished version of the FSOCO dataset ”legally” provided by a previous member of the E-TEAM; since it had a satisfiable size, we avoided labeling the data coming from the simulator by ourselves using RoboFlow. [3]

The dataset contains 8673 records which are split into training, validation and test sets according to an 80%-10%-10% split, those are then saved within a the *src/datasets* folder.

Training To train the model, we started by modifying the respective *.yaml file* to our needs, the ”path” row was changed to contain the path of the directory previously created by the *manage_dataset.py* method. The model was trained for 10 epochs on Google Colab GPUs, all the results can be found in section 3. Additional training, after the aforementioned 10 epochs, did not provide meaningful improvement to the model performance.

Inference We provide an example of the inference process for the stereocamera.

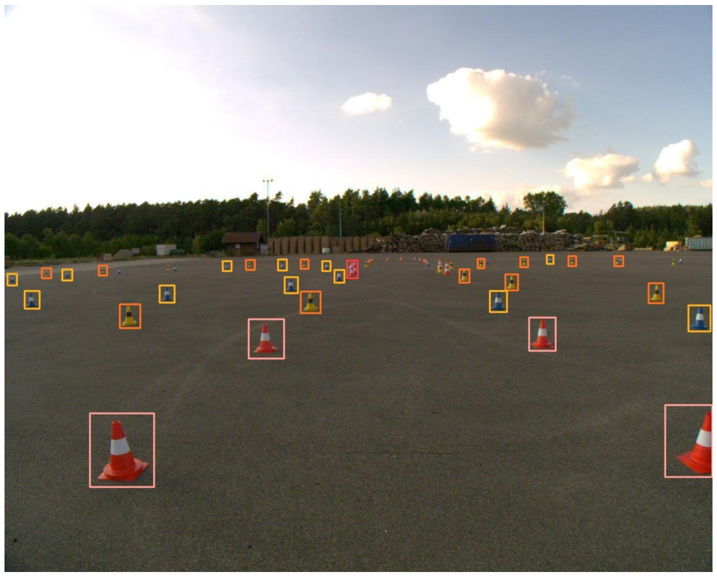


Figure 4: An example of inference on an image from the FSOCO dataset.

[3]RoboFlow is a tool suggested by the YOLOv5 team for the labelling of a dataset.

2.4 LiDAR Object Detection

Dataset creation The dataset for the LiDAR was created from the data provided by simulated sensor within the environment created by the Carla Simulator. The pointclouds used in this process correspond to those read by the sensor in about ten seconds. A ROSbag containing this data was extracted and used as an argument for the method *convert_rosbag_to_pcd.py*

- **--topic**, message topic default is "/carla/ego_vehicle/lidar";
- **--rosbag**, *dataset_bag.bag* path;
- **--path**, where the pointclouds are going to be stored;
- **--save**, save with built-in method.

PointCloud (PC) A PointCloud (PC) is a list of points in a Cartesian space $\langle x, y, z\rangle$. We can see in Figure 5 a visualization of the PC obtained from the simulator. Our purpose is to use this data to train a model able to accomplish an object recognition task, such work can be done in:

- 3D Cartesian space by using directly the point cloud.
- 2D by converting the PC to an image.

In our case, we have chosen to perform object recognition by flattening the PC in 2D. This was the best way to reach our goal given the time constraints and because in any case the model must be light due to the hardware constraints of the machine.

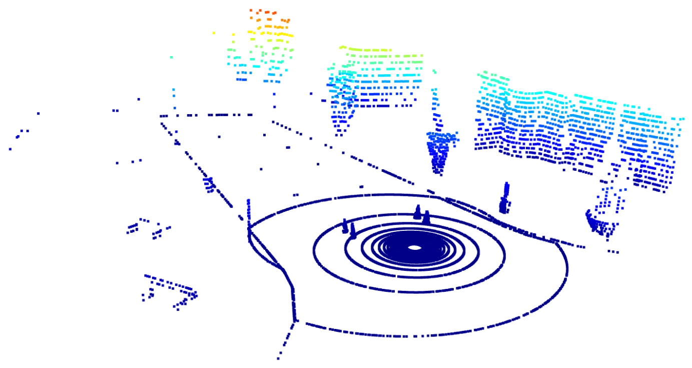


Figure 5: An example of PointCloud obtained from the simulator.

PC to image conversion To represent all the PC information in a grayscale image, we proceeded in the following way:

- Convert the points from 3D Cartesian coordinates (x, y, z) to spherical coordinates, representing each point as distance from the centre r, the longitude θ and latitude ϕ;

- Define the output image resolution (img_width, img_heigth) in pixels; we found that a good choice was $(480, 480)$, since the YOLOv5 model would not have to rescale anything;
- Map the spherical coordinates in the image space, via a simple min-max scaling to bring r in $(0, 255)$, θ in $(0, image\_width)$, ϕ in $(0, image\_height)$;
- Discretize the scaled values (since we are building a bitmap image);
- Build the image by placing on the x and y axes the θ and ϕ values, and represent r as the pixel color.

Using this representation, we estimated a 5% compression rate. As an example, shown in Figure 6, we show the final image which has 16544 points of the 17311 of the starting PC.

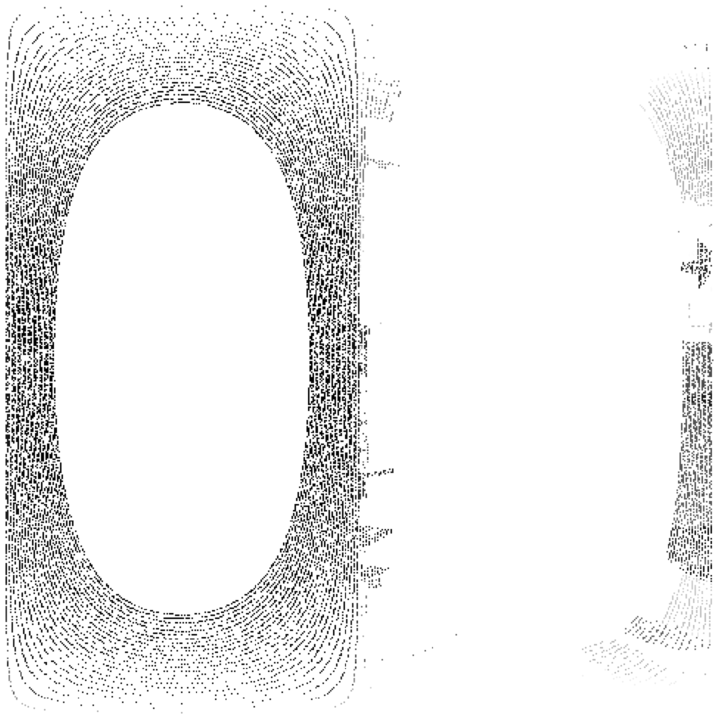
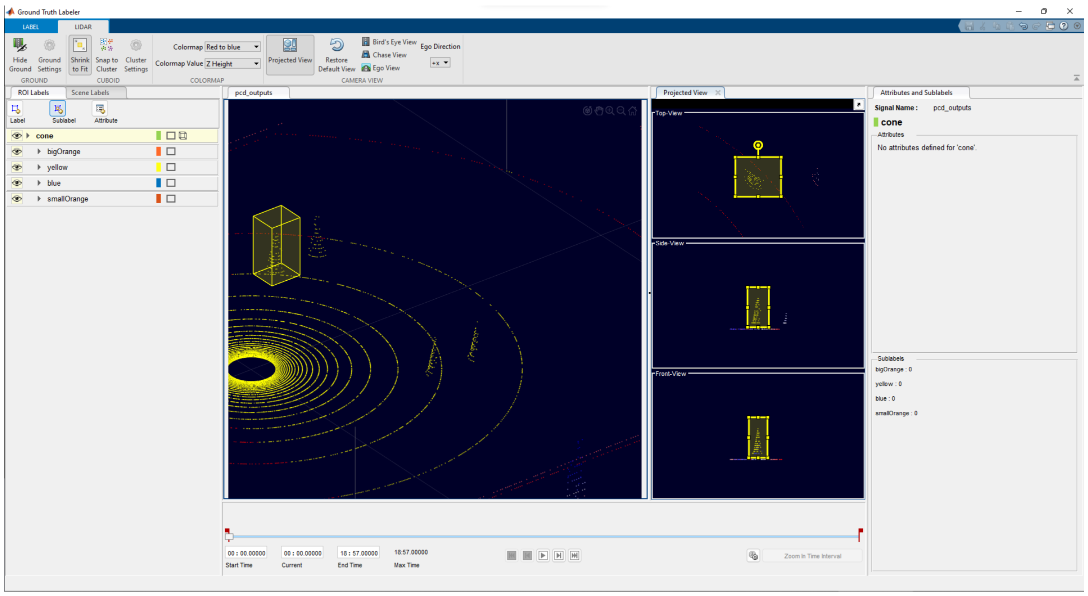


Figure 6: An example of Point Cloud converted as image.

Labeling Due to time and hardware issues in having the simulator available on our machines, we performed a manual labeling of the PCs using the *groundTruthLabeler* available in Matlab. 600 PC were labeled by drawing bounding boxes around each recognizable cone in the scene. Figure 7 shows a snapshot of this process.



Colormap: Red to blue
Colormap Value: Z Height
Ego Direction: +x

ROI Labels
cone
bigOrange
yellow
blue
smallOrange

Signal Name : pcd_outputs

cone

No attributes defined for 'cone'.

bigOrange : 0

yellow : 0

blue : 0

smallOrange : 0

00 : 00.00000	00 : 00.00000	18 : 57.00000	18:57.00000
Start Time	Current	End Time	Max Time

Figure 7: Labeling on point clouds

Once we got the 3D cone-bounding boxes for each PC we used *draw_2d_box.py* to generate the real dataset, this script takes as input all the point clouds with attached file containing the 3D bounding boxes. We converted each PC into an image by also converting the bounding boxes into 2D boxes and thus saving the image with the labels in the YOLO format. We implemented also a parametrized thresholding filter to filter out the boxed cones depending on the density of points contained within them.

This allowed us to generate different datasets with increasing minimum point density for the cones, and observe which was the optimal tradeoff for a good model generalization.

Training The chosen model was YOLOv5 so the training was the same as we did for the stereocamera, but this time the *.yaml* file declares that there exist only one class ("cone"). 600 images were then split as usual into training, validation and test sets. The model was trained for 200 epochs, since there were no major improvements after that amount, instead, performances were getting worse. The results are somewhat worse than the stereocamera since there were many false positives, don't worry, we will talk about this during the results section.

Inference Just as we did for the stereocamera, we provide an example of how the inference process works with the LiDAR dataset.

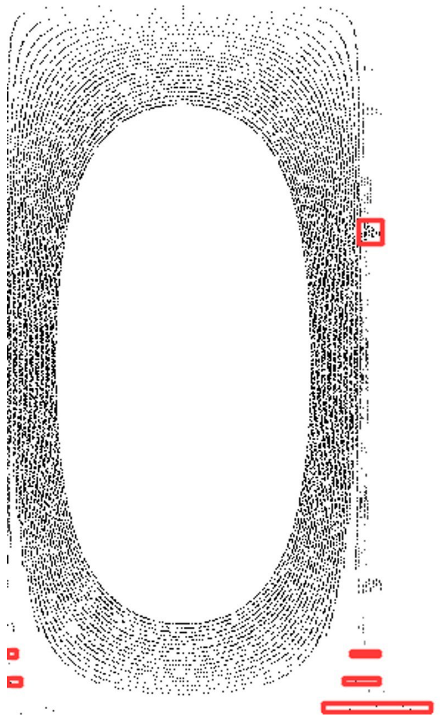


Figure 8: An example of inference on an image from the LiDAR dataset.

Post processing As explained before, the *.png* format of the spherical images is necessary in order implement the cone detection process through the YOLOv5 model. The format required by the SLAM process, however, is the *.pcd* one. For this reason, after identifying the relative position of the cones for each image, we perform a post-process operation to get the 3D coordinates of the cones in the original Cartesian space. The YOLOv5 model outputs the cones bounding boxes in image-coordinates (x_{min}, y_{min}, x_{max}, y_{max}). These boxes will contain all the points for the cones plus those for the foreground and background elements in that box. Therefore, we need to filter out these undesired points and end up with only those that describe the detected cones. We addressed this issue by identifying the clusters of cone points via outlier detection.

Such process, performed on each box in the model output, is described by the following steps:

1. convert the box boundaries, currently expressed in spherical coordinates, to the original non-discretized PC space;

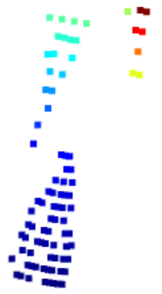
2. perform the outlier detection over r. This is done by assuming that each object in the scene has its points in a small region of the r axis and such assumption works particularly well in the case of small objects like cones;

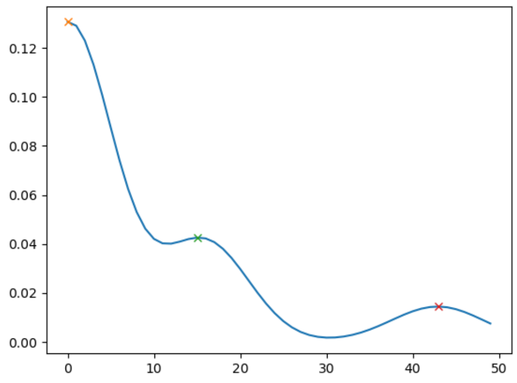
3. select the first peak, which is assumed to be the nearest to the car, as the cone cluster. Since the cone has been detected, there is likely no object obstructing its view;

4. filter out all the points not belonging to the cone cluster.

If our task consisted only in avoiding to hit the cones, this solution should work fine because. If another object, e.g. a sidewalk, were standing in front of the cone, we would still want to avoid it.

The underestimation of the cone distance can present some issues during the construction of the circuit map. This however can be corrected by subsequent recordings of the same cone during the run.


(a) Scene after outlier detection



(b) Density plot

Figure 9: Scene view and relative density plot with peaks on the r axis. Colorscale in the view represents the height from the ground.

2.5 Sensor fusion

Sensor fusion deals with combining the output of the models responsible for both the stereo-camera and the LiDAR at inference time. The goal of this process is to match the cones class (color and size) with the accurate position given by the LiDAR sensor.

- The stereo camera, besides the RGB image, produces also a grayscale depthmap image, the gray level represents the distance of the object from the camera. Since in presence of an almost exact alignment between the RGB image and said depthmap, we can put the stereocamera model boxes directly on the latter.

- The depthmap is then transformed into a point cloud (fig.11). In order to keep the cone label mapping, we perform the same outlier detection process performed for the LiDAR model postprocessing.

- Then we need to map the depthmap-based point cloud space to the LiDAR point cloud space. We achieved this by labeling a small dataset of 30 points and using it to train a Support Vector Regressor to perform the mapping. The manual labeling was performed

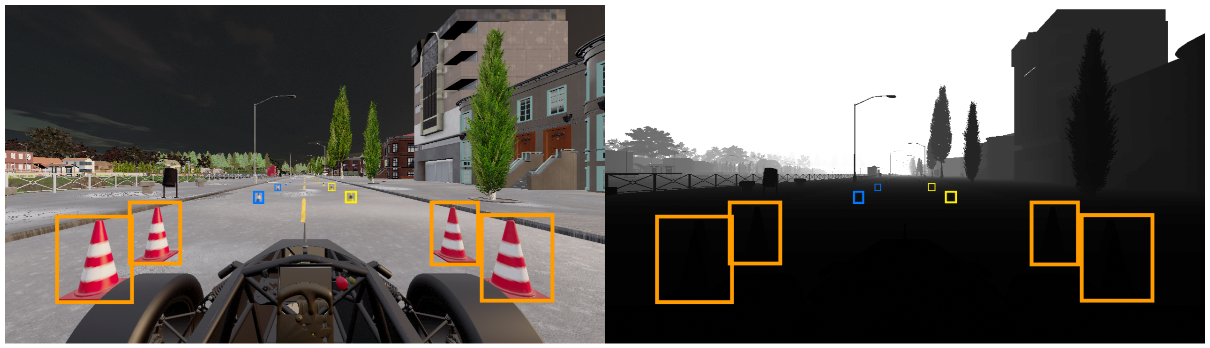

(a) RGB image (b) Depth Image

Figure 10: RGB and depthmap at the same time instant.

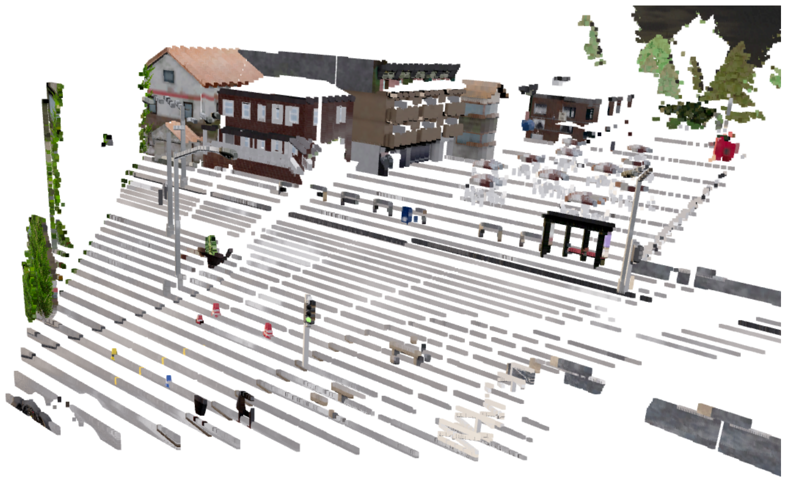
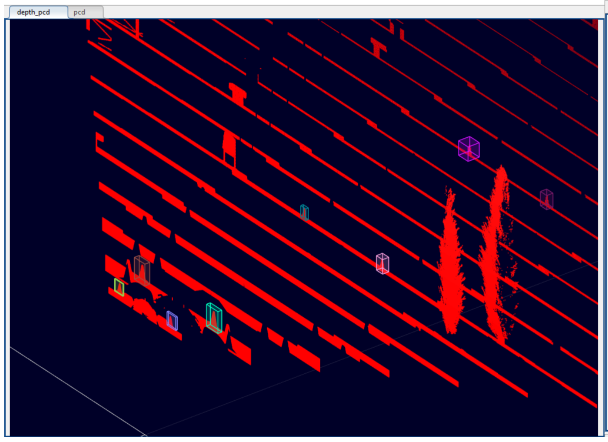
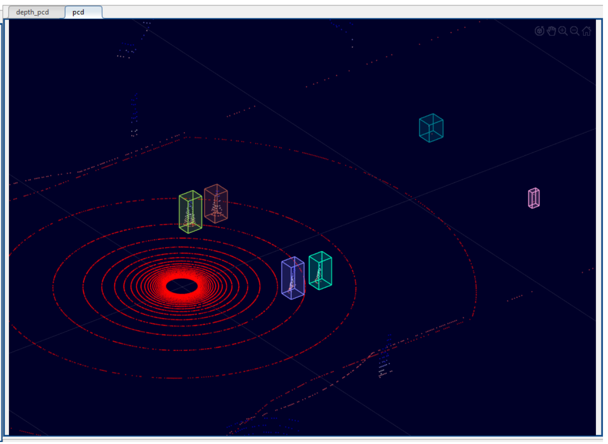


Figure 11: Pointcloud generated from depthmap and rgb image.



(a) Depth PC



(b) Lidar PC

Figure 12: Parallel manual labeling of depth and lidar PC of the same time instant

with the same tools of the LiDAR cone datasets.

We now need to account for just two remaining set of points in the LiDAR space: the centers of the cones detected by the LiDAR model, and the centers of the cones from the stereocamera model that are associated with the cone class and have translated into LiDAR space by the SVR.

- To match these two sets, we perform a sort of mutually exclusive 1-NN based on the distance of the points from one another. We end up with the a label being associated with each of the LiDAR model cones centers (fig 13).

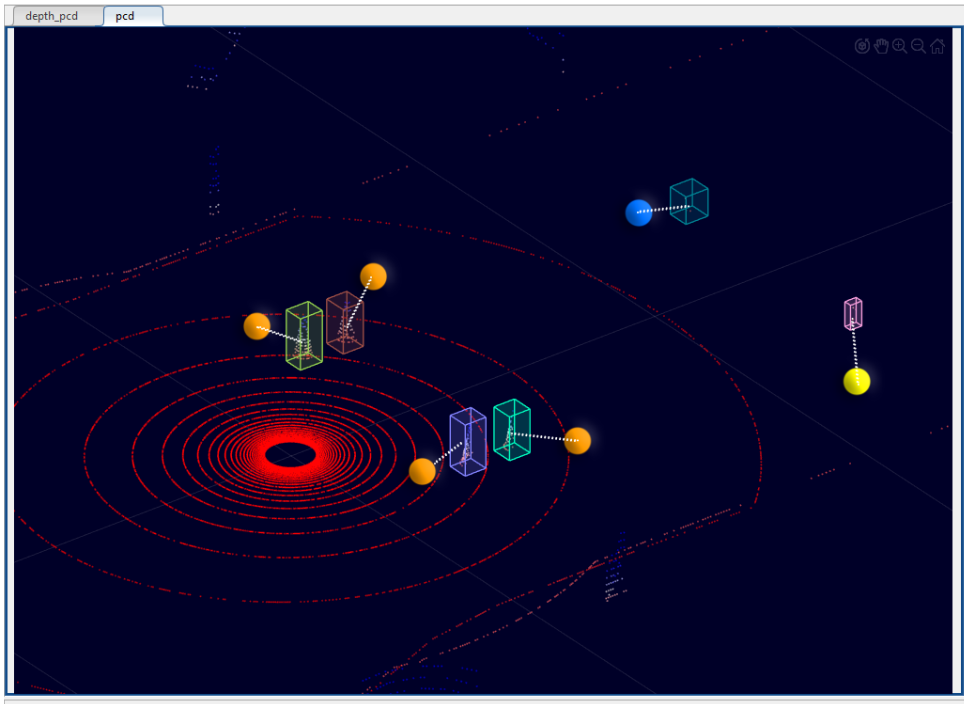


Figure 13: Association of the SVR output with the lidar cone centres.

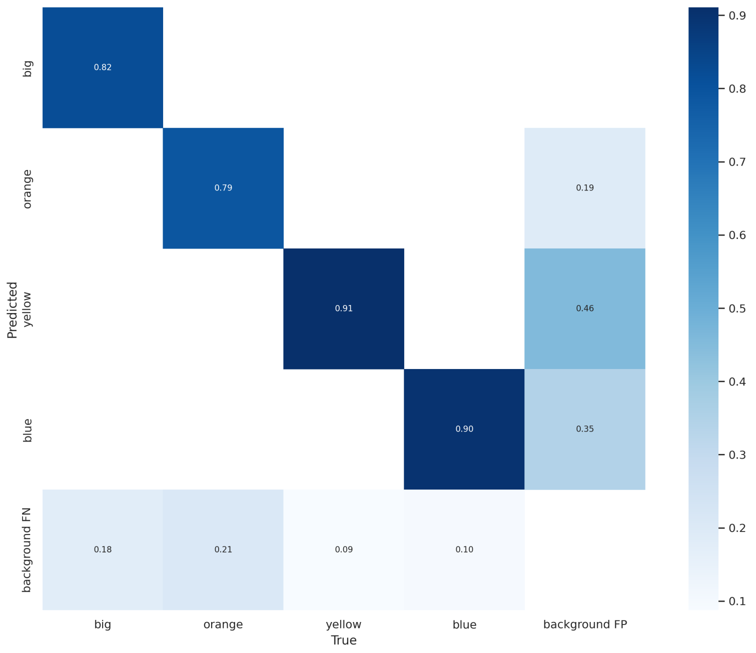
Finally, the output of this process is used by the SLAM process to map the entire track and to position the car within it.

3 Results

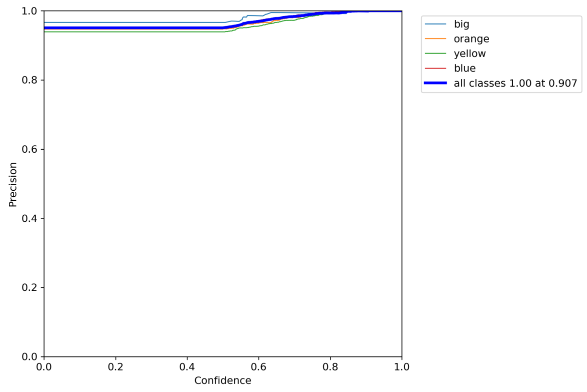
In this section we will discuss about the results we have achieved with the two models previously described. Said results do not represent the final performance of the overall cone detection process due to them being computed before the sensor fusion step, developed and described, in another document, by the **Sensory Data: Sensor Fusion** group.

As we will see, we would like to point out that the results from the LiDAR cone detection model are not as good as those from the stereocamera's one. The reason for such behaviour is probably associated with the scarcity of available pointcloud data.

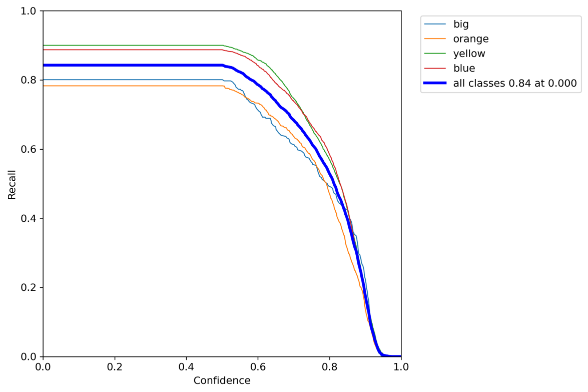
3.1 Stereocamera



(a) Confusion matrix



(b) Precision curve



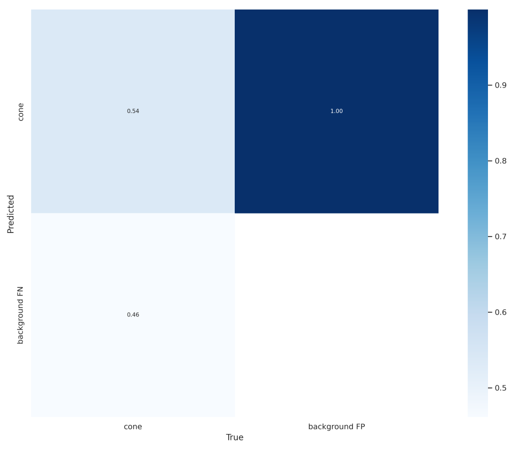
(c) Recall curve

Figure 14: Results for the stereocamera model test

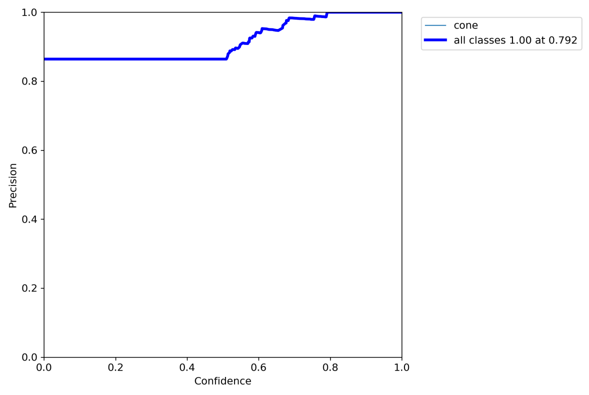
As expected from a state-of-art model for object detection like YOLOv5, the results are great. There may be some issues with false positives since the recall is not really high, but those should be solved during the sensor fusion process, once aligned with the results coming from the lidar's model.

Stereocamera cone detection test				
Class	**Labels**	**Precision**	**Recall**	**mAP@.5**
all	6427	0.951	0.843	0.911
big	286	0.966	0.801	0.893
orange	1276	0.947	0.783	0.878
yellow	2381	0.939	0.9	0.937
blue	2484	0.95	0.887	0.935

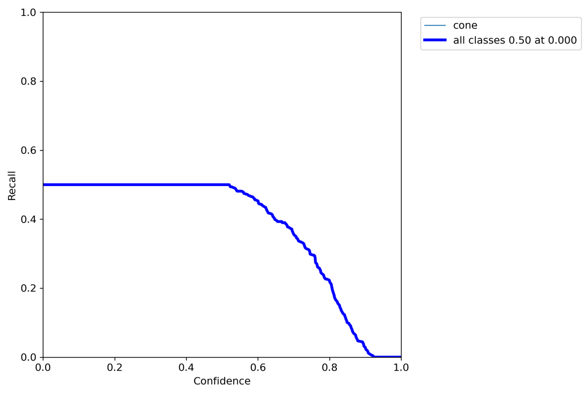
3.2 LiDAR



(a) Confusion matrix



(b) Precision curve



(c) Recall curve

Figure 15: Results for the lidar model test

The results of the cone detection performed on the LiDAR readings are less exciting than those obtained for the stereo camera. This is also associated with the fact that YOLOv5 is not explicitly designed to work with images in spherical coordinates. The other solutions in the field of object detection for pointclouds, such as VoxelNet or PointNet, however, were not prosecutable due to the bad maintenance of these libraries or the lack of actual sufficiently recent releases of the same.

Lidar cone detection test				
Class	**Labels**	**Precision**	**Recall**	**mAP@.5**
all	318	0.888	0.5	0.706

Despite these conditions, we still believe the inclusion of a network such as YOLOv5 within the pipeline responsible for the cone detection of LiDAR, as a step forward compared to the workflow previously implemented by the E-Team.

3.3 Issues

As shown on Figure 15, the LiDAR's model presented a less than exciting recall, which implies a high amount of false negatives. This is due to the richness of the dataset in cones represented by a poor amount of points. While the nearest cones presented an high density of points, the ones farthest from the sensor were represented by just a couple of points. The rationale behind this labeling behaviour, accounting for cones at every detectable distance, was to discourage a myopic attitude in the model, pushing it to identify cones at a greater distance from the sensor. This resulted into a tradeoff between the model's classification performance and maximum range of detection.

Another issue faced by both models is the perception of some false positives. These, usually resulted in lamp lights and other road parts labeled as cones,

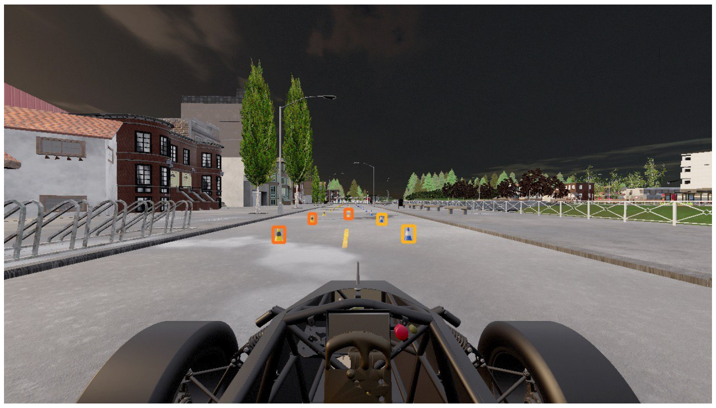


Figure 16: An example of false positive from the stereocamera simulator.

Another issue is with big orange cones labeled as small orange ones, that's something that we could not solve since the former are poorly represented over the stereocamera dataset.

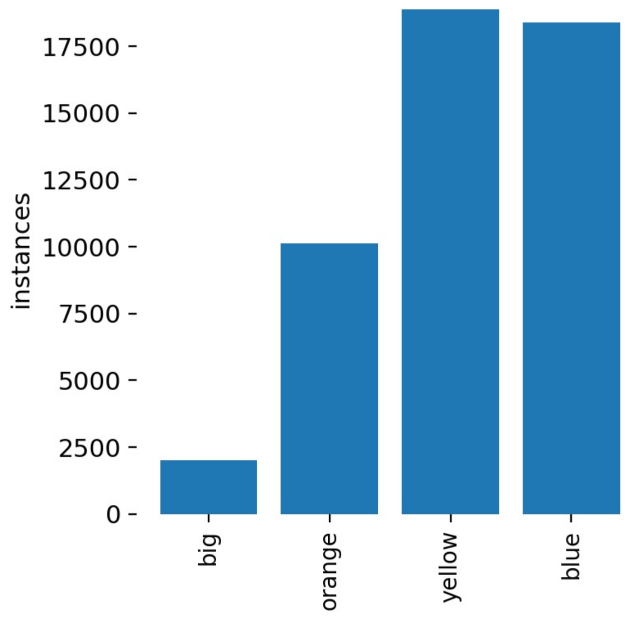


Figure 17: Distribution of the cones in the stereocamera dataset.

4 Future work

Altough increasing the size of the datasets used for both models would greatly improve their performance, the time constraints imposed by our course schedule were not benevolent in this regard.

Mainly concerning the development of the LiDAR's model: the manual labeling of the cones resulted quite demanding for our small team, a worthwhile endeavour would be researching tools so to automate this process.

Additionally, another alternative to our current workflow would include a model able to directly work with data in the point cloud format. Models like VoxelNet[2] and PointNet[1], while being theoretically able to scratch this itch, were not considered as viable alternatives to YOLOv5 due to their poor maintenance status. Regretfully, state-of-the-art models like those do not have any official public repositories yet and the ones available have not been updated for years making their codebases messy and rich with issues. We suggest to the e-team, as a future development, to build a new point cloud-friendly network from scratch.

Another note for the LiDAR: we found that the post-processing is the most time consuming task in the entire pipeline. Again, working directly with point clouds would save us from the many conversions and thus solve this issue.

References

[1] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation, 2017.

[2] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection, 2017.