‘You Only Look Once’ Object Detection

Group 6 Lab Report

S23_CO-548-A RIS Project

Constructor University Bremen

Group Members:

Joaquin Omar Arias

Mohamed Mourad Ouazghire

Era Gërbeshi

Naomi Mukuhi

Instructor: Prof. Dr. Amr Alanwar Abdelhafez

Date of submission: 26-05-2023

ABSTRACT

The objective of our project is to explore an real-object detection algorithm that would be implemented in 'Duckietown' which mimics the real-world environment of self-driving cars. The object detection algorithm that we focused on is the renowned YOLO v3(You Only Look Once) algorithm and in this paper, we go through related work, the steps we took as well as the output of our project including its precision and .

Keywords: Object detection, YOLOv3

I. INTRODUCTION

Duckietowns are the urban environments: roads, constructed from exercise mats and tape, and the signage that duckiebots, which are low cost mobile robots, use to navigate around. As it is for real-world cities, it is important for the health and safety of residents of Duckietown to navigate safely in the city. To achieve this, the duckiebots need to be able to identify other road users (like ducks and other duckiebots) as well as road signs (such as traffic signs and stop signs) accurately.

II. RELATED WORK

Object detection can be performed using either traditional image processing techniques or modern deep learning networks. Image processing techniques generally don't require historical data for training and are unsupervised in nature. OpenCV is a popular tool for image processing tasks. Hence, those tasks do not require annotated images, where humans labelled data manually (for supervised training). However, these techniques are restricted to multiple factors, such as complex scenarios (without unicolor background), occlusion (partially hidden objects), illumination and shadows, and clutter effect. Deep Learning methods generally depend on supervised or unsupervised learning, with supervised methods being the standard in computer vision tasks. The performance is limited by the computation power of GPUs, which is rapidly increasing year by year.

Types of Object Detection

(Boesch, 2021) State-of-the-art object detection methods can be categorised into two main types: One-stage vs. two-stage object detectors.

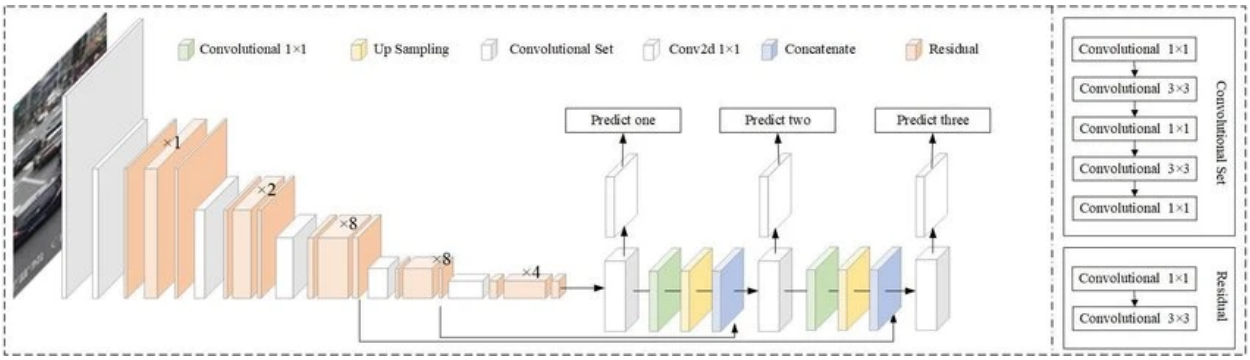
An object detector solves two subsequent tasks: finds an arbitrary number of objects (possibly even zero), and classifies every single object and estimates its size with a bounding box.

One stage object detectors: They predict bounding boxes over the images without the region proposal step. This process consumes less time and can therefore be used in real-time applications. One-stage object detectors prioritise inference speed and are super fast but not as good at recognizing irregularly shaped objects or a group of small objects. The most popular one-stage detectors include the YOLO, SSD, and RetinaNet.

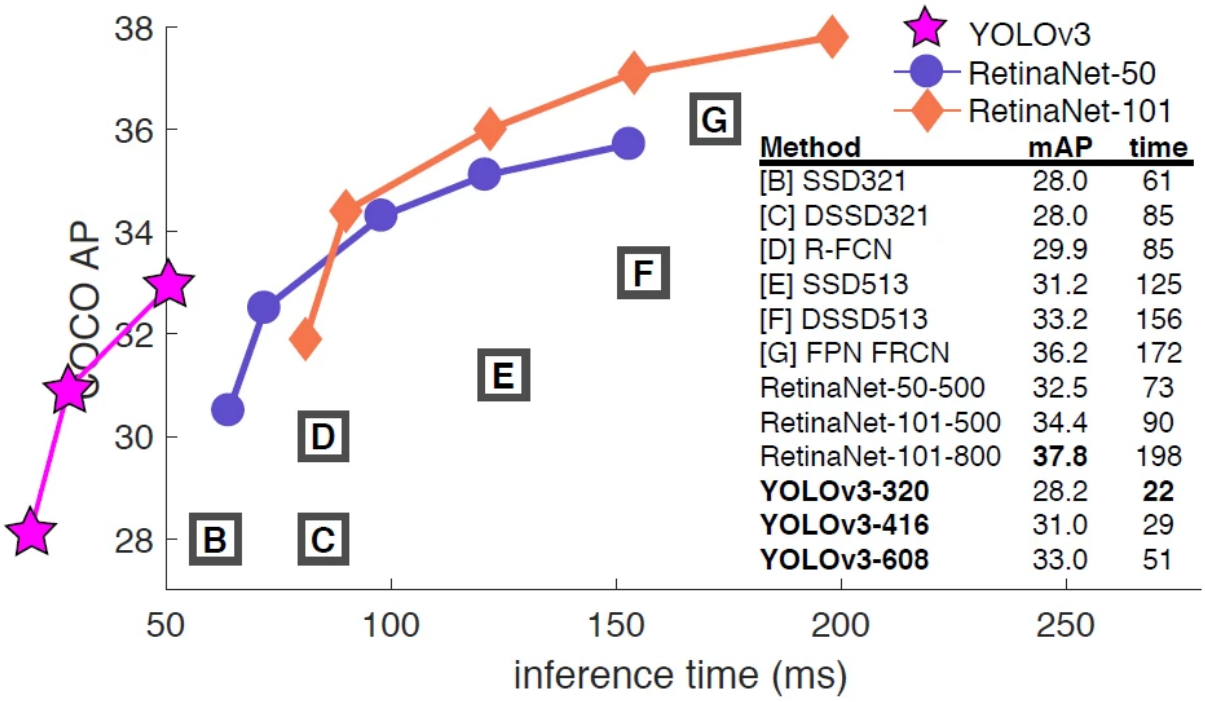
Two stage detectors: These detectors, the two-stage architecture involves object region proposal with conventional Computer Vision methods or deep networks, followed by object classification based on features extracted from the proposed region with bounding-box regression. Two-stage methods achieve the highest detection accuracy but are typically slower because of the many inference steps per image. Various examples include region convolutional neural network (RCNN), with evolutions Faster R-CNN or Mask R-CNN and the latest evolution is the granulated RCNN (G-RCNN).

YOLO v3

(Meel, n.d.)As typical for object detectors, the features learned by the convolutional layers are passed onto a classifier which makes the detection prediction. In YOLO, the prediction is based on a convolutional layer that uses 1×1 convolutions, hence the name.



CNNs are classifier-based systems that can process input images as structured arrays of data and recognize patterns between them (view image below). YOLO has the advantage of being much faster than other networks and still maintains accuracy. High-scoring regions are noted as positive detections of whatever class they most closely identify with. For example, in a live feed of traffic, YOLO can be used to detect different kinds of vehicles depending on which regions of the video score highly in comparison to predefined classes of vehicles. Other than running significantly faster than other detection methods with comparable performance, you can easily trade-off between speed and accuracy simply by changing the model's size, without the need for model retraining.



Method	mAP	time
[B] SSD321	28.0	61
[C] DSSD321	28.0	85
[D] R-FCN	29.9	85
[E] SSD513	31.2	125
[F] DSSD513	33.2	156
[G] FPN FRCN	36.2	172
RetinaNet-50-500	32.5	73
RetinaNet-101-500	34.4	90
RetinaNet-101-800	**37.8**	198
YOLOv3-320	28.2	**22**
YOLOv3-416	31.0	29
YOLOv3-608	33.0	51

III. APPROACH

Creating dataset

Machine learning models are designed to learn from data and make predictions or decisions based on that data. Therefore we need an image dataset of images of duckies, duckiebots and road signs that will be used to train and test the YOLO model. We took a video through the duckie's camera as it went around duckietown so as to use the frames as a dataset.

We also used existing datasets from different resources (tag the references) so as to improve the performance of the object detector and have more accurate results. Since some of the images were blurry, we used a script with a Laplacian filter to separate the non-blurry images from the blurry images.

```
# Go through all images in data folder
for imageFile in images:
    print('Processing image {}'.format(imageFile))

    image = cv.imread(str(imageFile))
    gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    fm = cv.Laplacian(gray, cv.CV_64F).var()

    # blurry
    if fm < threshold:
        cv.imwrite(str(blurry_folder.joinpath(imageFile.name)), image)
    # not blurry
    else:
        cv.imwrite(str(good_folder.joinpath(imageFile.name)), image)
```

The Laplacian is a scalar differential operator that can be applied to a scalar function [2] and is commonly used in image processing and computer vision to detect edges and other features in an image [1]. In the code above, the Laplacian is applied to the grayscale version of the image (`(cv.Laplacian(gray, cv.CV_64F)`) and its variance is calculated. This value is used as a measure of the "blurriness" of the image.

Labelling the Image

In order to train the object detector, the datasets need to be labelled with the relevant classes, which in this case is the duckiebot, duckie, road and stop signs. To do so, we used an open source data labelling tool called LabelImg. (*Heartexlabs/labelImg*, n.d.) After defining the list of classes, we labelled each image in the dataset and a corresponding text file was saved together with the image.

The label for the image is specified as follows:

```
2 0.4  0.22 0.02 0.1
3 0.75 0.22 0.03 0.1
1 0.99 0.44 0.03 0.07
```

The 5 elements are labelled as follows:

1. Class ID (in the same order as in the file containing the class names). In our case, Class 0 is a bot, class 1 is duckie and so on.
2. x coordinate for the centre of the object in the image

3. y coordinate for the centre of the object in the image
4. object width
5. object height

Preparing the dataset

To proceed, we had to set up a directory structure consisting of two sub-directories: ‘frames’ which contained the images and ‘labels’ which stored all the corresponding labels that were previously set using the LabelImg software mentioned earlier. We also ran the script “create_dataset.py” which we were able to effortlessly build the directories needed to accommodate, organise and manage the training process. The whole dataset was splitted into three sets: training, testing and validation. The training set contains 85% of the images and labels, and the testing and validation sets contain 7.5% respectively.

Preparing the YOLO config files

After the datasets had been created, we forked the Darknet repository[1] and added the dataset to the forked version[2]. Here, we created the YOLO configuration files which included the file containing the class names, the files where the training and validation datasets are, as well as the files specifying what neural network architecture to use, which was tiny YOLOv3 in our case. We decided to go for the tiny version because it is faster and smaller in size, which is good for using it with Raspberry Pi or Jetson Nano (Rosebrock, 2020). We also created the 'duckie_backup' folder, for further manipulation.

Training

To start training, we cloned the forked repository in a Google Colab notebook[3]. Then we set up the CUDA Toolkit to have Nvidia's GPU acceleration for efficient training and also downloaded the pre-trained weights from the Darknet website. After that, we started the training by running the command:

```
!export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-9.2/lib64 &&
./darknet detector train cfg/duckie.data cfg/yolov3-tiny-duckie.cfg
darknet53.conv.74
```

The 'duckie.data' file contains the number of classes, path to a text file with the paths of all training images, path to a text file with the paths of all validation images, path to the 'duckie.names' file (which contains the classes), and the backup folder.

After the training was complete, the weights were exported to be used in the testing stage.

[1] https://github.com/pjreddie/darknet.git
[2] https://github.com/jariasn/darknet.git
[3] https://colab.research.google.com/drive/1I-9CTdz5QGuTFAqrRQkeHUr1-OkS653l?usp=sharing

Testing

After the training, the weights were imported in other Google Colab notebook[4] that we created designed for testing our model. Here the CUDA environment was set up, as in the training notebook. To see how it worked, the object detection was performed for a single image by running the following command:

```
!export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/cuda-9.2/lib64 &&
./darknet detector test cfg/duckie.data cfg/yolov3-tiny-duckie.cfg
duckie_backup/tiny-yolo-duckie.weights datasets/testset/202_frame1184.jpg
-thresh .50
```

A threshold of 0.50 was used, which means that YOLO only displays objects detected with a confidence of .50 or higher.

The output consisted of the class detected and the confidence percentage. However, for further model evaluation, the bounding boxes information was needed. Darknet framework does not show this information on the output so the source code of darknet had to be modified. For this, the line

```
printf("Bounding Box: Left=%d, Top=%d, Right=%d, Bottom=%d\n", left, top,
right, bot);
```

was added in line 292 of the 'image.c' file.

In order to evaluate the model, a file with the ground truth boxes (which are the actual boxes that we created during labelling) for each class in each image. For this, the labels from the training dataset were taken and processed into a new file that contained the ground truth boxes in the following format:

```
image_1_path
class_1
(x1, y1, x2, y2)
class_2
(x1, y1, x2, y2)
(x1, y1, x2, y2)
```

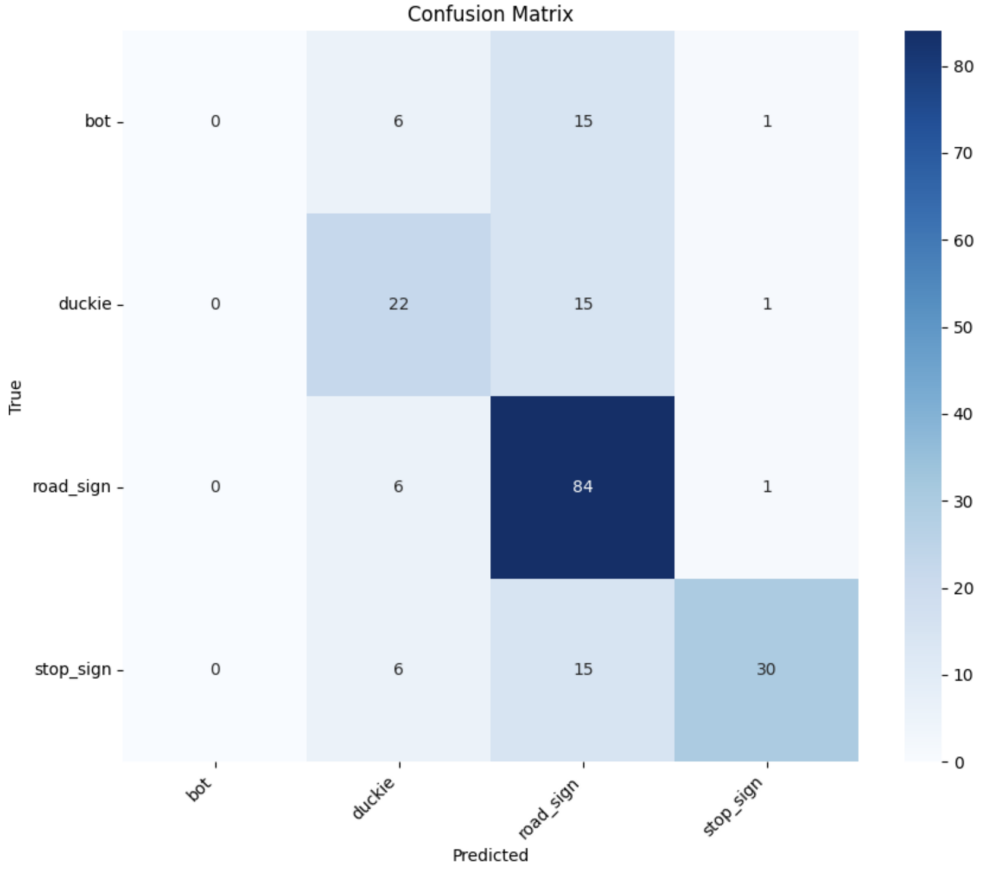
After performing object detection in all the images from the training dataset, it was possible to create a dictionary containing the predicted bounding boxes in the same format. This data could be processed in order to determine some metrics such as the confusion matrix, which is further explained in the next sections of the present paper.

[4] https://colab.research.google.com/drive/1KTiGe3cxPZeqqvHbAlBw8Iej71HOL5aj?usp=sharing

IV. RESULTS

With the two pieces of information for the testing set (the predicted bounding boxes and the ground truth bounding boxes) it was possible to determine the number of True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). The TP are predictions that match with the ground truth, the TN mean that the model does not predict the label and is not part of the ground truth, FP are those predictions that include a label that is not part of the ground truth, and the FN mean that the model does not predict a label but it is part of the ground truth.

After comparing the bounding boxes from each image and each class by running some python scripts that can be found in the testing Colab notebook, we were able to obtain the following confusion matrix:

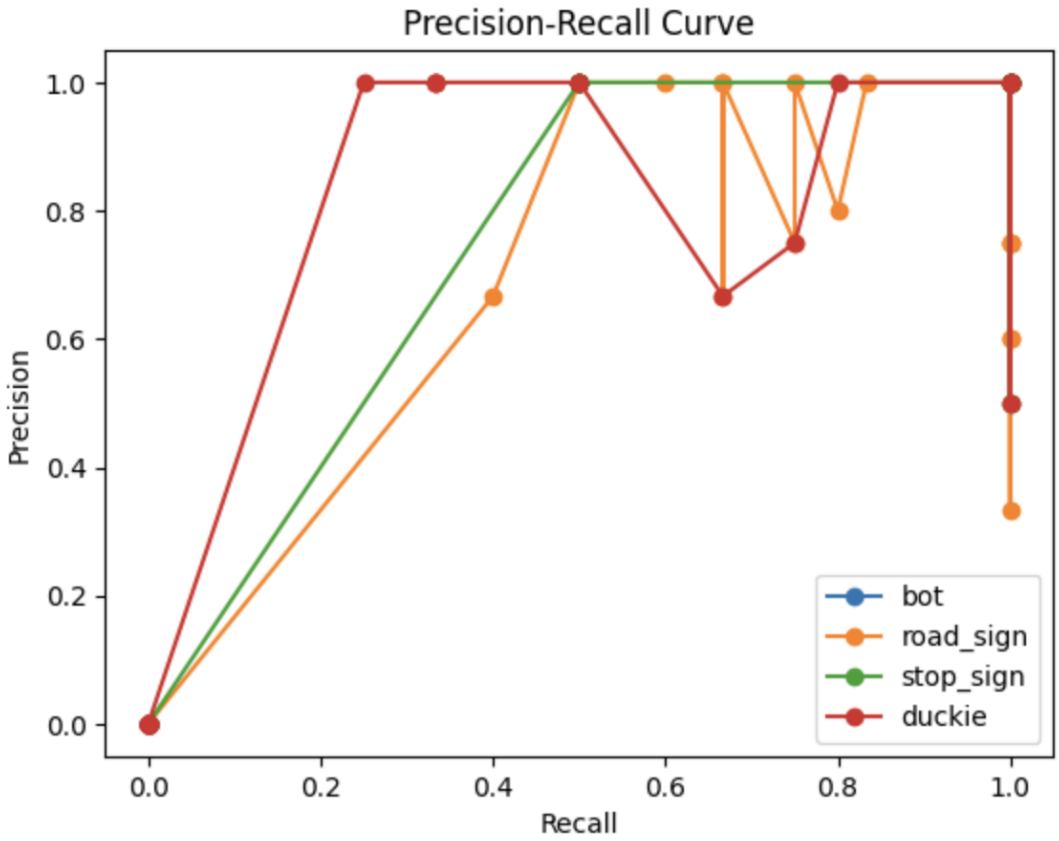


The precision and recall were also calculated using the formula:

$$Precision = \frac{TP}{TP+FP}$$

$$Recall = \frac{TP}{TP+FN}$$

And the following Precision-Recall curve was plotted using matplotlib:



To determine the average precision per class, the composite trapezoidal rule was used to calculate the area under the curve of each class. For this, the function np.trapz() was used, and the following values were obtained:

Table 1: Average Precision (AP) per Class

Class	**Approx. Area Under the Curve (AP)**
bot	0.00
duckie	0.77
road_sign	0.65
stop_sign	0.75

The Mean Average Precision (mAP) was calculated using the following equation:

$$mAP = \frac{1}{N}\sum_{i=1}^{N} AP_i$$

So,

$$mAP = \frac{1}{4}\sum_{i=0}^{3} AP_i$$

$$mAP = 0.54$$

The Mean of confidence per class was also calculated:
road_sign: 88.16%
stop_sign: 82.61%
duckie: 81.69%
bot: 95.03%

The interpretation of these results is discussed in the following section.

V. DISCUSSION

When analysing and assessing the performance of the trained model The Mean Average Precision was utilised to gather more insight into the model's behaviour and effectiveness. The Mean Average Precision (mAP) is a commonly used metric for evaluating object detection models such as Tiny YOLOv3. It provides a comprehensive assessment of the model's performance by considering several sub-metrics, including the confusion matrix, intersection over union (IoU), precision, and recall.

The confusion matrix is a tabulation that summarises the model's predictions against the ground truth labels. As mentioned in the Results part, it consists of four attributes: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). TP represents the instances where the model correctly predicted a label that matches the ground truth. TN refers to cases where the model correctly identified that a label was not present in the ground truth. FP indicates instances where the model incorrectly predicted a label that is not part of the ground truth (Type I Error), while FN represents cases where the model failed to predict a label that is actually present in the ground truth (Type II Error). In our case the "bot" class was the one that showed only False Negatives (FNs), indicating that the model failed to detect any instances of this class. This was quite a challenge since it suggested that the model had difficulty accurately identifying objects belonging to the "bot" class, leading to missed detections. However, this was influenced by the limitations of a relatively small dataset. In the training images, there was an insufficient representation of bots compared to other elements. Moreover, the dataset included various types

of bots, which presented a challenge for the model in recognizing them as objects belonging to the same class.

Intersection over Union (IoU) is a measure of the overlap between the predicted bounding box (which are the actual boxes that we created during labelling of the data in the datasets) coordinates and the ground truth box. It quantifies how closely the predicted bounding box resembles the ground truth box. In the evaluation of our model, the Intersection over Union (IoU) threshold of 0.5 was used to determine whether a predicted bounding box is considered a match with the ground truth box. This threshold ensured us that the predicted bounding box had a significant overlap with the ground truth box, indicating a close alignment.

To further analyse the model's performance, precision and recall were calculated. Precision measures the ability of the model to correctly identify true positives (TP) out of all positive predictions (TP + FP). Recall measures the model's ability to find true positives (TP) out of all instances that should have been predicted as positive (TP + FN). It quantifies the model's capability to detect objects accurately. By calculating precision and recall, it was possible to construct a precision-recall curve, which gave us more insights into the trade-off between precision and recall at different decision thresholds.

To calculate the mAP, average precision (AP) values were calculated for various IoU thresholds across a range of recall values from 0 to 1. AP represents the precision achieved at each recall level. The mean of these AP values was then calculated to obtain the final mAP score.

In conclusion, mAP is a comprehensive metric that takes into account various sub-metrics such as the confusion matrix, IoU, precision, and recall to assess the performance of object detection models like Tiny YOLO v3.

Furthermore, in order for us to deploy the trained model on the Duckiebot, we tried to incorporate the Darknet framework into the Robot Operating System (ROS) environment. Darknet, an open source neural network framework written in C and CUDA, provided us with a powerful tool for object detection tasks. With its efficient performance and support for both CPU and GPU computation, Darknet seemed like a promising choice for integrating our model into the Duckiebot ecosystem.

To facilitate the integration, we created custom ROS packages for the camera module and Darknet framework. These packages were designed to enable seamless communication between the camera module and the Darknet framework, allowing for real-time object detection on the Duckiebot platform.

However, despite our efforts, we encountered several challenges that stopped us from successfully implementing Darknet on our Duckiebot. One major hurdle was the presence of system issues, which affected the compatibility and functionality of Darknet within the ROS environment. These issues required extensive troubleshooting and debugging, but unfortunately, they remained unresolved within the scope of this project.

Additionally, connecting the Duckiebot with our ROS environment posed another obstacle. Due to technical constraints and limitations in the available resources, we were unable to establish a stable and reliable connection that would allow the model to effectively communicate with the Duckiebot hardware.

As a result, the integration of Darknet into the Duckiebot ecosystem could not be accomplished within the scope of this project.

VI. CONCLUSION & RECOMMENDATIONS

Generally, throughout this project a few key steps and milestones were achieved. One of those was finding a variety of datasets containing images of the duckietown in different lighting and different setups, and also obtaining our own dataset by the utilisation of the duckiebot camera, from which we managed to get 507 images, which then were filtered for blurriness and separated into the respective folders. All of this was done using the LabeImg software, which made our work more concise and ordered. Afterwards, we managed to successfully train, test and validate our model, together with evaluating it through The Mean Average Precision and its sub-metrics. However, when trying to transfer our trained model into the duckiebot, we encountered a few hurdles such as the system misalignment between our ROS and the Darknet framework, wiring issues, and connection problems when aiming to enter the Jetson Nanon board.

Even though we could not effectively integrate our model into the bot, we believe that it can be possible if the Darknet ROS package is customised to recognize the duckiebot's camera, and if the connection with the Jatson Nano is established by trying the command line "export ROS_MASTER_URI=http://[name].loca:11311" on each one of the terminals of the bot. Another way to upgrade the model, and the whole system in general, would be to also move towards a more advanced version of YOLO such as version 5 or version 7, and/ or try more efficient algorithms and approaches.

VII. INDIVIDUAL RESPONSIBILITIES

- Joaquin Arias:
    - Training Process and Data Processing
    - Labelling
- Mohamed Mourad Ouazghire
    - LabelImg/Darknet set up
    - Dataset Images Filtering
- Era Gërbeshi
    - Documentation, Video Editing
    - ROS environment set-up
- Naomi Mukuhi
    - Dataset preparation and labelling
    - Documentation(report)

References

Boesch, G. (2021, july). *Object Detection in 2023: The Definitive Guide - viso.ai*. Viso Suite. Retrieved May 21, 2023, from https://viso.ai/deep-learning/object-detection/

Duckietown. (n.d.). *duckietown/logs: Duckietown logs display*. GitHub. Retrieved 2023, from https://github.com/duckietown/logs

duckietown_dataset Computer Vision Project. (n.d.). Roboflow. Retrieved 2023, from https://universe.roboflow.com/cvduckies-ahtmc/duckietown_dataset

heartexlabs/labelImg. (n.d.). GitHub. Retrieved May 23, 2023, from https://github.com/heartexlabs/labelImg

Meel, V. (n.d.). *YOLOv3: Real-Time Object Detection Algorithm (Guide) - viso.ai*. Viso Suite. Retrieved May 23, 2023, from https://viso.ai/deep-learning/yolov3-overview/

Object Detection and Imitation Learning in Duckietown. (n.d.). MinYoung Chang. Retrieved May 26, 2023, from https://minyoung.info/documents/yolo_report.pdf

Piyathilaka, L. (2020). *Week6 : Train the YOLO object detector*. GitHub. https://github.com/lasithaya/Robotics-lasi/wiki/Week6-:-Train-the-YOLO-object-detector

Project Overview. (n.d.). Project Overview. Retrieved May 23, 2023, from https://universe.roboflow.com/cvduckies-ahtmc/duckietown_dataset

Redmon, J. (n.d.). *Darknet: Open Source Neural Networks in C*. Joseph Redmon. Retrieved 2023, from https://pjreddie.com/darknet/

Redmon, J. (n.d.). *pjreddie/darknet: Convolutional Neural Networks*. GitHub. Retrieved 2023, from https://github.com/pjreddie/darknet

Rosebrock, A. (2020, January 27). *YOLO and Tiny-YOLO object detection on the Raspberry Pi and Movidius NCS*. PyImageSearch. Retrieved May 26, 2023, from https://pyimagesearch.com/2020/01/27/yolo-and-tiny-yolo-object-detection-on-the-raspberry-pi-and-movidius-ncs/

Tran, M. (2022, March 9). *(PDF) Analysis of Object Detection Models on Duckietown Robot Based on YOLOv5 Architectures*. ResearchGate. Retrieved 2023, from https://www.researchgate.net/publication/359084405_Analysis_of_Object_Detection_Models_on_Duckietown_Robot_Based_on_YOLOv5_Architectures