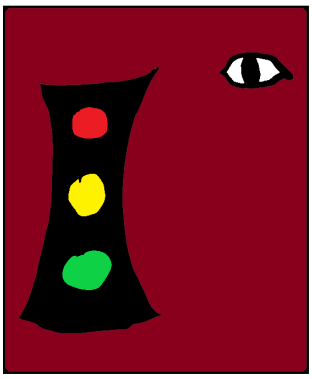
Traffic Light Detection and Tracking



Final Report

2A1: Traffic Light Detection and Tracking

Aaryan Shenoy
Clayton Gowan
Morgan Roberts
Xiaohu (Max) Huang
Robert Madriaga

Department of Computer Science
Texas A&M University

05/05/2023

Table of Contents

1 Executive summary (1-2 pages; 5 points)

Traffic lights are a ubiquitous tool for the drivers of vehicles all over the world. They offer a method by which to regulate and sustain the flow of traffic, so their importance cannot be overstated. As we move into the age of automation, it has become clear that traffic light detection will be a vital ingredient for the success of autonomous vehicles. After all, in order for self-driving cars to function properly, they need to be able to detect, react, and behave as a rational driver would in all situations, and this conduct is not possible without auxiliary knowledge of the state of traffic lights. Indeed, as it currently stands, there already exist many implementations of traffic light detection. Most if not all of these require camera data, and some supplement with data from other sources such as lidar and local maps. Unfortunately, lidar is only effective at close range, so its use with traffic light detection is naturally limited. The primary solution is to use machine learning algorithms on the camera data to detect and classify traffic lights. However, driving is an environment in which reaction time and accuracy are second to none in terms of importance, and a neural network needs to be as responsive and correct about its predictions as possible. Current implementations are often either too slow or inaccurate to be considered a viable solution. Therefore, we seek to develop a method of traffic light detection, tracking, and classification that prioritizes speed in recognition results and is confident and accurate.

Our final design utilizes a ROS node which subscribes to a camera feed node. Our node then houses our primary system functionality which operates on the input received from the camera feed and then publishes bounding box information in a ROS message format as specified. Our system consists of a detection module and tracking module, with further logic performed on detections or tracked objects. For detection, we are using a trained YOLOv8 model on a dataset collected from rosbag files with previous competition footage in mcity. Our detections are then processed through a filter that maintains only the relevant traffic lights and passed to our tracker after being converted to the correct format. The tracker we use is Norfair, which is an open-source Python library for multi-object tracking. With the tracker output, we then are able to perform further logic to analyze the state of the lights and determine if they are flashing. Finally, we publish the lights being tracked along with their detection confidence scores, the type of the light, and the coordinates of the bounding boxes.

Our final system is able to accurately detect and track traffic lights on bag files we tested. The model detects and classifies traffic lights, the relevance filter keeps only the detections at the intersection immediately relevant, and the tracker is able to track multiple objects across multiple frames. Additionally, we are able to detect if lights are flashing. The system functions as expected in most of the scenarios we tested, but there were a few instances where the model got confused on lesser common light types due to a lack of training data. Having a more robust model stemming from a more diverse dataset is the main improvement we would make to the system.

The team met multiple times per week and utilized a hybrid waterfall-agile approach for development. This allowed us to hold each other accountable and work together throughout the week. Team members frequently collaborated on components and developed ideas together. The team leader organized meetings and made sure tasks were distributed. Overall, our team was able to work within the given deadlines and requirements.

2 Project background (2-4 pages; 5 points)

2.1 Needs statement

Self-driving vehicles require a means by which to detect traffic lights. Many detection systems use cameras and neural networks to detect and recognize traffic lights. However, in an environment where reaction times are integral to the safety of the passengers, these algorithms are far too slow to be of practical use. We need to develop a means to detect traffic light colors and types and track these detections which prioritizes speed while still maintaining a high level of accuracy.

2.2 Goal and objectives

The goal of this project is to design a neural network which is able to take an image and accurately detect any traffic lights along with their color, type, and location as fast as possible. The data will be output as an array of these values. Ideally, this process should be extremely fast and responsive. Our first objective is to get familiar with the tools and libraries, such as ROS, YOLOv8, and Norfair, as we plan to use these extensively. Next, we aim to test detection of traffic lights using YOLOv8 and downloading a pre-trained model to test the functionality. We will then use ROS and YOLOv8 to train our own model and derive weights for it. Next, we will configure Norfair to track lights across frames to reduce workload and frequency of the model. It is important that we are able to detect special cases such as flashing lights, as the state evolves over time, and that will be our next objective. Our final objective is to combine our previous objectives together into a pipeline using ROS and to polish our design to increase efficiency and accuracy. It should be noted that we have no plans to make purchases for this project, as we are designing only software and will use our own hardware for testing and development.

2.3 Design constraints and feasibility

In regard to constraints that affect the practicality of our project, there are several that must be addressed. There exists an inherent time constraint, as we have set due dates. We also must spend time training the model, and this limits the amount of time that we are able to spend perfecting the project. It should be noted that not every member of the team is familiar with machine learning, which plays an important role in this project. What's more, the entire team has little familiarity with the active software such as ROS, DeepSort, and YOLOv8, and this imposes a technical constraint on the project. Our team is working on laptops with very apparent hardware constraints. Factors such as the GPU and CPU can directly affect the speed of the model, and this should be taken into account. The model must also meet certain requirements, such as a desired level of confidence. Our implementation must also use an ONNX format model and output as an array of traffic light data.

Autonomous vehicles have the potential for great societal benefits in the way of convenience and quality of life. If implemented in a widespread, robust and correct manner, they could result in more optimal travel times and offer increased accessibility to transportation. On the other hand, there are concerns about privacy due to the monitoring equipment required to make the vehicles function.

There are also numerous safety concerns when it comes to autonomous vehicles. These systems must be designed with extreme precaution, as any error in the system could result in the loss of human life and/or damage to property. Simple errors on both the software and hardware sides could easily be catastrophic. However, autonomous vehicles also have the potential to offer many safety benefits. A wide or full 360 degree field of vision working in conjunction with fast computational algorithms theoretically gives a system that is operating under safe conditions the ability to detect and respond to situations faster than humans can and with more consistency.

Concerning the environment, optimal autonomous vehicles could potentially reduce emissions by creating shorter travel times overall. Additionally, most of these vehicles are electric by nature. On the contrary, an inefficient navigation algorithm could result in more pollution/emissions due to longer travel times.

During the semester, we realized one major constraint was storage space on our laptops. The bag files with test footage were often very large and team members did not have enough storage space to test with them. This is one factor that led to us purchasing an external SSD to boot into another computer for testing. A lack of storage space proved to be a significant constraint that had us quite limited in our testing until late in the semester.

2.4 Literature and technical survey

Project 1: Traffic Lights Detection and Recognition Method Based on the Improved YOLOv4 Algorithm [1]

The aim of this project was to optimize the YOLOv4 detection model in order to improve its ability to identify small objects and to refine its tracking precision. Modifications of feature extraction network layers and calculations of uncertainty of bounding box coordinate measurements were utilized to enhance the model's general accuracy and detection of small objects. Experiments were performed on LISA and LaRa traffic light datasets to analyze the performance of the study's algorithm along with 5 other neural networks. The study's YOLOv4 algorithm resulted in the highest performance based on AUC and mAP metrics. The authors concluded that both its network modifications and uncertainty calculations improved detection accuracy; however, the network modifications increased computations in their algorithm that resulted in negligible increase in runtime. Authors proposed that research into trajectory predictions of detected traffic lights be conducted as an extension to their study. This study is relevant since it overcomes a weakness of the YOLO model and highlights in its extension proposal that object tracking should be incorporated into YOLO detection. The proposed design attempted to exploit the potential of object tracking in our machine learning model. *We opted to use YOLOv8 instead, and didn't use this for tracking. This particular project had little impact on our end product.*

Project 2: A YOLO Based Approach for Traffic Light Recognition for ADAS Systems [2]

This project aimed to develop a means to detect and classify traffic lights in advanced driver assistance systems in order to alert drivers when they may cross a red light thus helping to avoid potentially deadly situations. This was accomplished by training a model using the YOLOv4 model on the LISA dataset which includes images of traffic lights from the streets of California. First, the DNN model was trained to identify traffic lights given a whole frame of input. The output here would then be sent to another module to estimate distance and decide whether or not to notify the driver. In addition, there is another step which filters out traffic lights that are not in the driver's path so the system will not alert the driver to irrelevant traffic lights. Three classes, green, yellow, and red were trained in the model and YOLO was found to be the most reliable algorithm. This is partly due to its ability to detect without any backpropagation. The alert system was done by estimating the rate of change of distance to the light and combining that information with data on whether the driver was slowing down or not and the actual distance to the light. Every other frame was processed instead of every frame to improve performance. The system had an average framerate of 13.4 with relatively high average precision results in the 90 percentage range, except for yellow in the 80s due to low training data. They stress that the accuracy of their model can be improved by including more images of yellow and horizontal lights which they were lacking. The researchers also stressed the importance of frame processing before detection in finding an optimal resolution. They settled on 608x608 since darknet requires a multiple of 4 and it struck a balance between speed and accuracy. It is important to choose a resolution that is not too high or too low. *We implemented something similar with YOLOv8, so this had some impact on our final product.*

Project/Research 3: Traffic Light Recognition — A Visual Guide [3]

This project describes the traffic light detection problem and shows techniques that can be used for efficient real time solutions.

The traffic light detection problem can be broken down into 3 parts:

- Identifying Regions of Interest
- Training a Classifier
- Tracking and Optimization

And there are several techniques that can be used for an efficient solution:

- Sliding Windows - slow brute force method that isn't particularly useful
- Cropped Sliding Windows - same as above but with regions of interest cropped out, faster
- Color Thresholding - fastest method, process image to show only red green and yellow
- BLOB analysis - pixel clusters obtained after color thresholding can be analyzed further by their shapes and sizes

We did cut the image size down, since most lights will be in the top half. We trimmed the set of lights down based on the area of the hitbox. We wanted to only get lights that are closer to the camera, as these are more important.

Project 4: HDTLR: A CNN based Hierarchical Detector for Traffic Lights [4]

This project introduces HDTLR, a convolutional neural network based on DeepTLR which uses a hierarchy of object classes to classify an object. HDTLR is compatible with many different common feature extraction networks such as AlexNET, and is capable of handling different input sizes after training. The last layers are a multitask network split between region classification to create a probability map for each class and the second task being bounding box regression for each region similar to DeepTLR. The Softmax layer from DeepTLR is modified to use a tree that defines the hierarchy of objects, which determines the probability of each object's classification. The final decision is made from a combination of hierarchy, DeepTLR clustering. *This paper had little impact on our product. We used neither HDTLR nor DEEPTLR.*

Project 5: A deep learning approach to traffic lights: Detection, tracking, and classification [5]

This paper emphasizes the need for a system which can detect traffic light data without the need for map-based information and lidar data. The paper details an approach using deep learning, stereo vision, and vehicle odometry to detect, track, and classify traffic lights. Since the authors imposed limitations on their solution that coincide with ours, their proposed pipeline is something that we can use to similar effect. This paper helped to inform our decision to use YOLO for detection and classification, but to find another tool for tracking. Our proposed solution/design is very similar to some of these projects discussed above particularly in the use of YOLO for detection. This is a very popular algorithm for these kinds of projects and has been shown to be very fast. Our implementation is seeking to use a newer, possibly faster, version of YOLO than those used in some of the projects discussed above. Additionally, the use of a tracker in project 3 is also similar to our proposed design and need. We are seeking to use a multi-object tracking that not only uses Kalman filters and other motion-based algorithms but also works with deep learning to track objects. *Our end product heavily mirrors the implementation recommended by this paper.*

2.5 Evaluation of alternative solutions

Alternative solution 1
With less frequency, apply filters to the image to get a grayscale image. Use the large YOLOv8 model on the entire grayscale to detect hitboxes. Feed hitboxes into deepsort tracker to track hitboxes across frames and detect state data such as blinking. Feed hitboxes into the small YOLOv8 tracker to get the color of light and the type data.

We found that a reduced frame rate significantly decreased the accuracy of the tracker, so this solution is unlikely to be feasible. We chose not to opt for it as a result.

Alternative solution 2 – Tracking Techniques
One of the project requirements is to not only detect the traffic lights, but track them across frames as well. Tracking is different from detection in that instead of simply identifying an object on a single frame, tracking algorithms seek to predict the position/trajectory of an object across consecutive frames and may help account for dropped detection frames, occlusion, etc. There are different routes one can take in order to track objects. One is to use single-object tracking such as those included in the cv2 library including MOSSE, KCF, and more. These algorithms are simple to implement and can be very fast and effective, but are not as useful when multiple objects or traffic lights need to be tracked at once. This brings us to multi-object tracking algorithms such as SORT and DeepSORT. These algorithms are similar, but DeepSORT offers some key improvements over SORT. SORT does an acceptable job, but suffers from frequent id switches and cannot handle occlusion very well. DeepSORT seeks to improve on this by not only taking into account object motion but appearance as well. For these reasons, we will aim to implement DeepSORT. [6]

We switched to Deepsort for our tracker, but eventually found a tracker called Norfair, which was designed with a moving camera in mind. Since the camera moves with the car, it made sense to use this tracker.

Alternative solution 3 – Optimizing YOLO models
The YOLOv4 detection model has been optimized in a study to improve the model's to detect smaller objects [1]. Two strategies of tweaking YOLO's convolutional neural network and applying mathematical models to calculate the uncertainty of bounding box location calculations were applied. The first strategy involved technical skill of modifying network layers through splicing and upsampling, while the second strategy implemented complex statistical knowledge of Gaussian distributions. These strategies were proven to enhance the detection accuracy of small and general targets based on concrete metrics like mean average precision (mAP) and area under Precision-Recall curve (AUC). However, implementing this solution requires expertise in machine learning and statistics that our team does not have. Additionally, it is not known whether this strategy could be applicable to a newer model version like YOLOv8.

This solution was not explored, as much of the manual heuristics have been simplified by trackers such as DeepSort and Norfair.

Alternative solution 4 – Mini-YOLOv3
Mini-YOLOv3 is an object detector that is optimized for embedded applications, it has slightly reduced detection accuracy but considerably less demanding on the hardware platform [7]. The parameter size of Mini-YOLOv3 is only 23% of YOLOv3 and achieves comparable detection accuracy as YOLOv3 but only requires 1/2 detect time. This may allow for better performance on the non specialized hardware we have access to (laptops), but the reduced detection accuracy may lead to lower detection reliability of the system.

We found that we could run the detection and tracking every frame while keeping our response time below the 100ms threshold, so this solution was not necessary.

Alternative solution 5 – HDTLR (Hierarchical DeepTLR model)
In the paper by M.Bach et al., they describe how their model uses an alternative CNN approach to that of YOLO based on DeepTLR, and its capabilities to quickly identify traffic lights and their associated lanes [8]. Using their described approach we would need to choose a separate classification network such as AlexNet, and modify the DeepTLR Softmax function to use a hierarchy tree. Some of the advantages of

this approach would be that HDTLR would be able to work with different input sizes from what it was trained on. Giving more flexibility but being more complex than the more commonly used YOLOv8.

YOLOv8 fulfilled our requirements just as well as HDTLR and provided a simpler approach.

3 Final design (5-10 pages; 5 points)

3.1 System description

Please provide a description of the overall system. This description should include a high-level block diagram and a functional description of the different parts and interfaces in your final system. If any of these were revised after the CDR, please document the changes and provide a justification.

The project will consist of a YOLOv8-Norfair machine learning framework that is contained within an ROS2 node network. In the machine learning framework, the YOLOv8 model detects traffic light boxes in individual camera frames, while the Norfair algorithm tracks YOLOv8 detections across multiple camera frames to improve efficiency. This framework is housed within an ROS2 node, which is an abstraction of a process. This node is a member of our project's ROS2 node network, where camera frame images and traffic light detection results will be continuously transmitted.

The project's system is based on an ROS2 node network architecture. Various processes such as the camera frame relayer and the team's YOLOv8-Norfair program are represented as a ROS2 node. Each node is able to communicate with each other through interprocess communication; hence, our project's network transfers camera frame images from its respective source node to the YOLOv8-Norfair node for traffic light detection. Through the ROS2 library, we configured node communication to follow a publisher-subscriber approach. A node can either be a "publisher" that sends messages of a certain type to a channel, also known as a "topic", or it can be a "subscriber" that receives a message from its respective topic. This communication approach allows us to maintain streams of input and output data for continuous processing of camera feeds that require traffic light detection.

Our system's network will contain a ROS2 publisher node that sends camera feed images to a topic known as "/front_camera/image_raw". The node will utilize a ROS bag file that can be converted into image frames that depict scenes of driving in urban areas. This type of file is necessary for the node because it contains ROS2 message data that is compatible for ROS2 publisher-subscriber node communication. This publisher node will act as our project's input for traffic light detection and tracking.

Next, a ROS2 subscriber node will link to the camera feed publisher node by subscribing to the "/front_camera/image_raw" topic. This subscriber node will contain our project's YOLOv8-Norfair detection program. This program is written in Python and employs the Ultralytics YOLO, OpenCV, and the ROS Client library. The program will analyze the image frames from its subscribed topic and output those same frames labeled with bounding box detections of traffic lights that specify their signal types such as red, green, yellow, blinking, left turn, etc. Although YOLOv8 will detect traffic lights of different signals, Norfair aids the model in order to avoid common problems of the former model such as inability to detect occluded traffic lights. In the machine learning node, the YOLOv8 model will create boxes that enclosed the area of detected traffic lights, while the Norfair algorithm will associate YOLOv8's boxes with the algorithm's own predictions to yield final detections onto the image frames.

The system's final layer will execute computations to identify states of traffic lights other than their signal colors. This logic will be contained within the Python program of the machine learning node. The main traffic light state that the layer should spot are flashing yellow lights, which are prevalent in roads. Norfair outputs like object ID, number of frames, and dimensions of its associated bounding boxes will be the inputs of the layer's logic. The number of frames is a very important measurement to detect blinking lights because it can indicate the amount of time that a traffic signal was detected if it is compared to a recorded number of missed detections. Finally, the project's architecture is depicted in Figure 1.

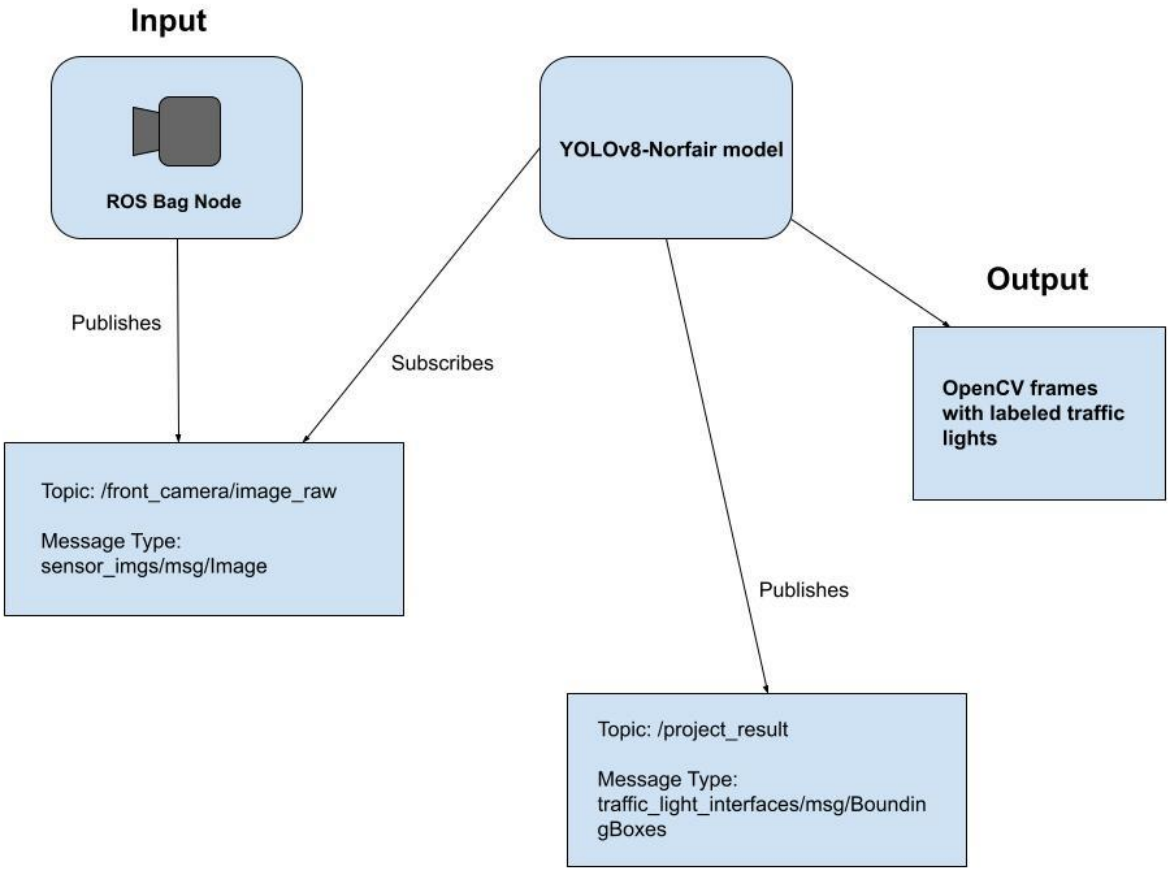


Figure 1: High-level project architecture

3.2 Complete module-wise specifications

The first module of the system is the detection module. This has been the main focus of the project up to this point, as the detection component serves as the foundation for the project task as a whole. The detection model is trained using YOLOv8 in Google Colab with the free GPUs offered and is trained on a dataset hosted and managed in Roboflow. The most recent and robust version of the model so far was trained based on the yolov8s (small) pre-trained weights with imgsz set to 1280 on 50 epochs and completed training in between 2 to 3 hours.

The dataset consists of images gathered by the team from rosbag file screenshots primarily from previous year competition footage at mcity, as this is the specific operating environment the model should perform well in. The most recent and robust model iteration was trained on a dataset consisting of 358 images with additional images added through augmentations in Roboflow. The augmented dataset consisted of 735 training images, 69 validation images, and 42 testing images. See the figure below for additional details on the pre-processing and augmentation steps used.

One challenge with collecting data has been finding an adequate number of images/instances of less common light types. Red and green lights are by far the most common light types, with yellow being significantly less common and turn light variations being rare in the bag files. This also presents additional challenges for finding good videos/images to test the model on more rare light types. We will

continue to seek out data for more traffic light variations. The implementation notes section contains more information on the current dataset state.

Currently, the model is being fed not the full image frame for detection, but a portion of it. This results in increased speed as there are less pixels to process. One interesting finding so far has been that the model performs very slightly worse after converting from PyTorch to ONNX. The results were still sufficient, but it is noteworthy.

Our reason for choosing YOLOv8 over other detection network methods is that it has been shown to be faster and better performing than older YOLO iterations as can be seen in the below graphs in figure 2 from the Ultralytics github page. In addition, YOLO is known for being quite good in terms of inference speed and accuracy balance. The model will take an input frame and divide it up into *S x S* grid cells. Each of these cells has the ability to mark the image with *B* bounding boxes. *S* and *B* are pre-set numbers for the model's configuration, while a bounding box is a rectangle that attempts to fully encircle a detected object in the image. YOLO requires a grid cell to detect an object if the object's center is within the cell. For a bounding box, the probability that the box contains a relevant object, location coordinates of the box's center, dimensions of the bounding box, and the probabilities that the associated object belongs to each respective class that we are trying to detect. In order to get rid of redundant bounding boxes out of the initial *B* boxes, ratios of the intersection area of two bounding boxes over their combined area are calculated and boxes with high resulting values are kept. Finally, a step called non-max suppression selects boxes with the highest probabilities that they are an object out of the remaining boxes.

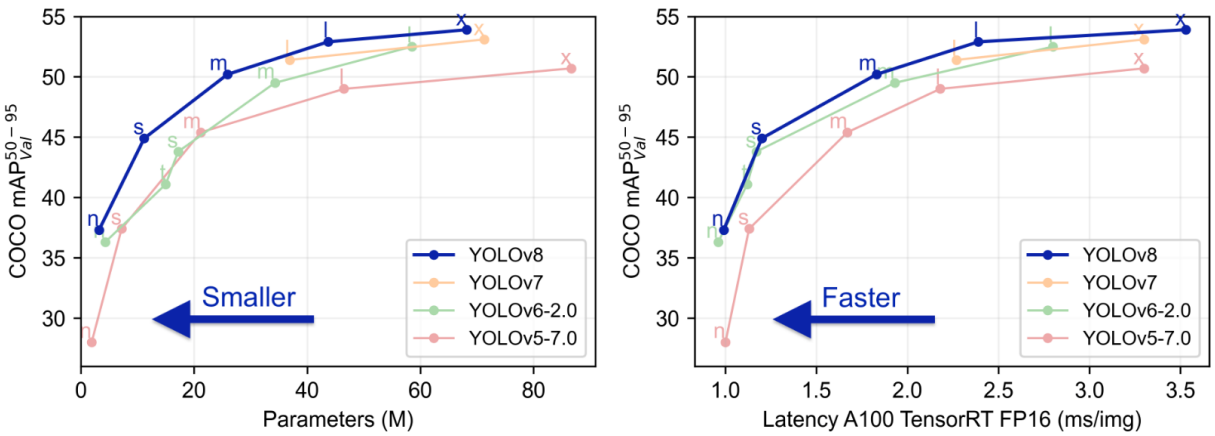


Figure 2: YOLO Performance Comparison

The second primary system module is the tracking module, which takes input from the detection model and assigns IDs such that we can continually track objects across multiple frames. The tracker that is currently implemented is the Norfair tracker. Norfair is a python multi object tracking (MOT) library that seeks to offer an easy-to-integrate MOT solution. This tracker takes as input YOLO detections converted to Norfair detection format and is updated every frame. The tracker works by estimating the future position of objects based on previous detections and matching them to newly detected objects. This matching relies on a distance function, which can be specified in the code, and appearance learning of the detected traffic lights. Norfair also supports tracking in footage with moving cameras through mathematical computations that account for camera tilts and rotations. There are also several other

tunable tracking parameters that we can continue to tweak, such as initialization delay, distance threshold, and more.

There are several reasons for using Norfair at this point over alternative solutions such as the previously planned and discussed DeepSORT. First, while we did integrate DeepSORT, we would need to train our own feature extractor which involves a process where little documentation and resources could be found and would perhaps take more time than we have. Second, Norfair is fast with speed really only limited by the detection network speed while still maintaining good results. Third, Norfair is much more user friendly and easy to integrate than DeepSORT and offers many tunable options.

In dealing with the output from the tracking module, we only want to output relevant traffic lights that we are tracking. The model may detect traffic lights far away or behind the relevant set of traffic lights, but for competition output and actually reacting to the lights we want to only deal with those that are relevant. To accomplish this, we chose to go with a simple relevance filtering algorithm that takes our yolo detections as input and disregards any detections that have bounding box areas which are less than 40 percent of the current maximum detection bounding box area.

The final important functionality of the tracking module will be the labeling of flashing lights in the output feed. The primary additional state we aim to detect is flashing, as this is commonly found on roadways. To accomplish this, we made use of Norfair output information like object ID, the hit_counter attribute, and tracked object age. The hit_counter attribute is a metric that indicates the frequency of a tracked traffic light's match to a YOLOv8 detection and is recorded by Norfair in its TrackedObject class. We can utilize this integer attribute by keeping track of the number of missed detections relative to the lifespan of the object. If the ratio of missed detections to age is within some specified threshold, then we can label that tracked signal as a flashing light. In summary, our system's algorithm can be depicted in stages in Figure 3.

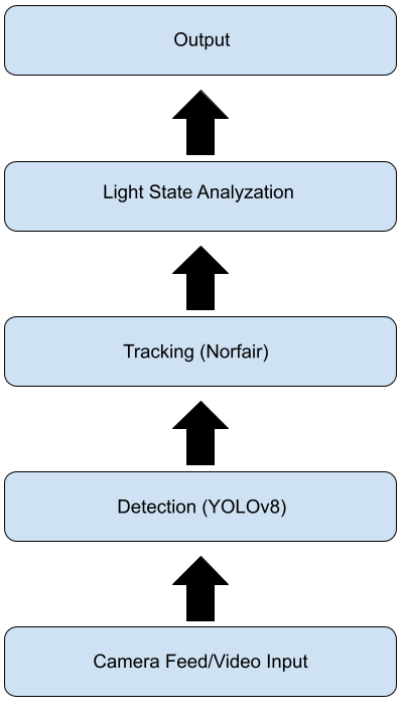


Figure 3: Stages of project's algorithm

A Python program module will utilize the YOLOv8 detection module that is loaded from a Pytorch or ONNX file in order to locate and label traffic light signals in camera footage provided by ROS bag files. Additionally, the Python program module will create the ROS publisher and subscriber nodes that form into our system's ROS network. This module imports the OpenCV library, Python ROS Client Library, NumPy, CvBridge, sys module, os module, random module, sensor_msgs package, and the Ultralytics package.

At the beginning of the program module, it instantiates a YOLOv8 and CvBridge object. Then, it defines a function named 'callback_Img' that an ROS subscriber node will later invoke. The 'callback_Img' function's input is of an Image message type, which is a data structure that represents a frame from our ROS bag files. The function converts this object into an OpenCV image object. This OpenCV image is processed by YOLOv8's predict() function call to produce a list of bounding boxes coordinates and detection probabilities. Within the predict() function call, an additional parameter of 'show=True' is passed in order to output an image window that shows the processed image frame with YOLO's bounding boxes superimposed onto it. Additionally, opencv is used to draw tracker output in another window. After the definition of 'callback_Img', we start up the ROS2 communication layer with a Python ROS Client Library function call. Next, we create a ROS2 subscription node that subscribes to a '/front_camera/image_raw' topic that contains the Image-type messages. This node also calls the 'callback_Img' function whenever it receives an Image message from its subscribed topic. Finally, we set our program to continuously check for received Images so that the subscriber node knows when to call the 'callback_Img' function.

Our system is hosted on a noVNC module, where we can run our ROS2 node network from a command line interface. A Docker container was created to host a Virtual Network Computing (VNC) server, along with other application dependencies. A noVNC web client will access the GUI of the Docker application through the VNC server. As a result, this forms the noVNC module. Inside the module, we run the command 'source /root/rootfs/foxy_setup.sh; ros2 bag play -r 10 -s rosbag_v2 --loop rootfs/filename.bag' on one terminal and the command 'source /root/rootfs/foxy_setup.sh; (ros2 run rqt_image_view rqt_image_view &); python3 /root/rootfs/image_subscriber-ros2.py' on another terminal. The first command will publish image frames from the bag file specified to the topic called '/front_camera/image_raw'. The second node will set up a subscriber node through the 'image_subscriber-ros2.py' file, which will receive the image frames stored in the '/front_camera/image_raw' topic and detect traffic lights.

3.3 Approach for design validation

How did you test your system to ensure that it did what it was designed to do? If these validation procedures were revised during the semester, please document the changes and provide a justification.

For our detection model, the primary means of quantitative validation and testing have been the output statistics from training and running on the test set. These metrics include mean average precision (mAP), recall, and others which are helpful for evaluating the initial performance of the model based on the testing and validation sets of the dataset. In addition, the inference speed of detection is outputted when running the program so it can be observed which models are faster. Another important testing and validation step is to observe the model performance on different rosbag files and scenes to see how the model handles different light types and scenarios. See the experimental results section for more information and graphs/statistics for the model.

For speed, our goal is to achieve inference every one tenth of a second or 100 milliseconds. This is something we can measure from YOLO output, but it is difficult to truly test how it will perform on

competition hardware, since we have been testing on our laptop CPUs and the lab computer. We recorded inference speed in milliseconds and sought to get this number as low as possible.

Figure 4 below shows terminal output containing our inference speed in milliseconds when not screen recording on the lab computer running off of a bootable linux hard drive. It can be seen that these three detections have an average inference speed of about 91.17 ms. We found that this average time was overall representative of the inference times we were getting in general on the lab computer when testing. This indicates that we are meeting our initial goal of running inference in less than one tenth of a second or 100 ms.

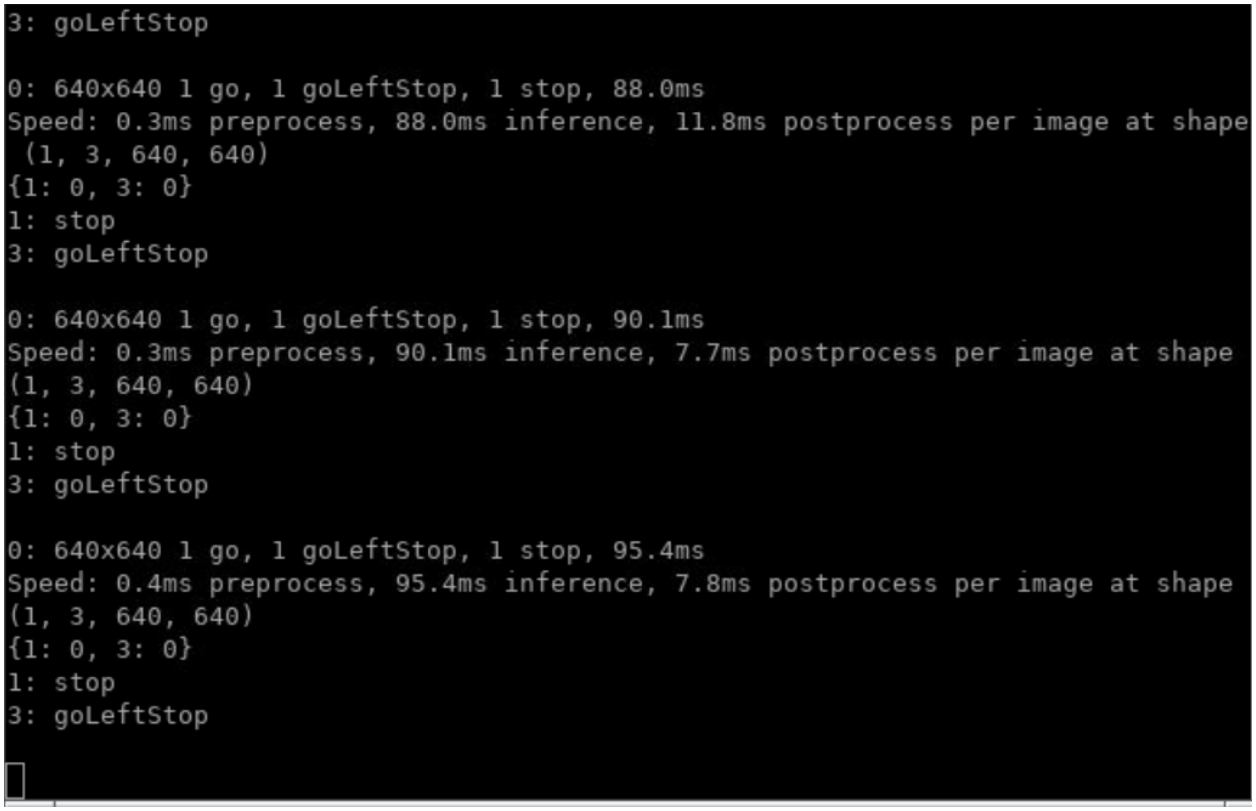


Figure 4: Inference Speed Demonstration

Ultimately, the primary testing and validation procedure involved running the complete product on a variety of rosbag files to test how the system performs in different scenarios. When running these files, we were checking to see if detections were accurate and if the tracker was handling IDs and motion reasonably well. We also checked the performance of our relevance filter and flashing light detection algorithms and tuned parameters as needed. The final demo video shows the finished product working on a complete mcity3 bag file. For further discussions on our results, see the experimental results section below.

4 Implementation notes (10-20 pages; 30 points)

Docker Setup:

The team set up our Docker container to host the project system using the Github repository from https://github.com/tamu-edu-students/ROS-Docker-Intro. Inside the root local directory of this repo, we ran the command ‘make build’ in order to create a Docker image called i0x0i/foxy-noetic. Next, we ran the command ‘make init’ so that a Docker container would run as an instance of the Docker image. Anytime that we need to restart the Docker container, we execute the ‘make start’ command.

We used docker technology to make sure that everyone has the same environment and the necessary dependencies for the project setup across multiple different devices.

To run the project on a new device, a docker container is built on the device from a docker image with the associated Dockerfile. The Dockerfile contains the installation commands for the packages needed.

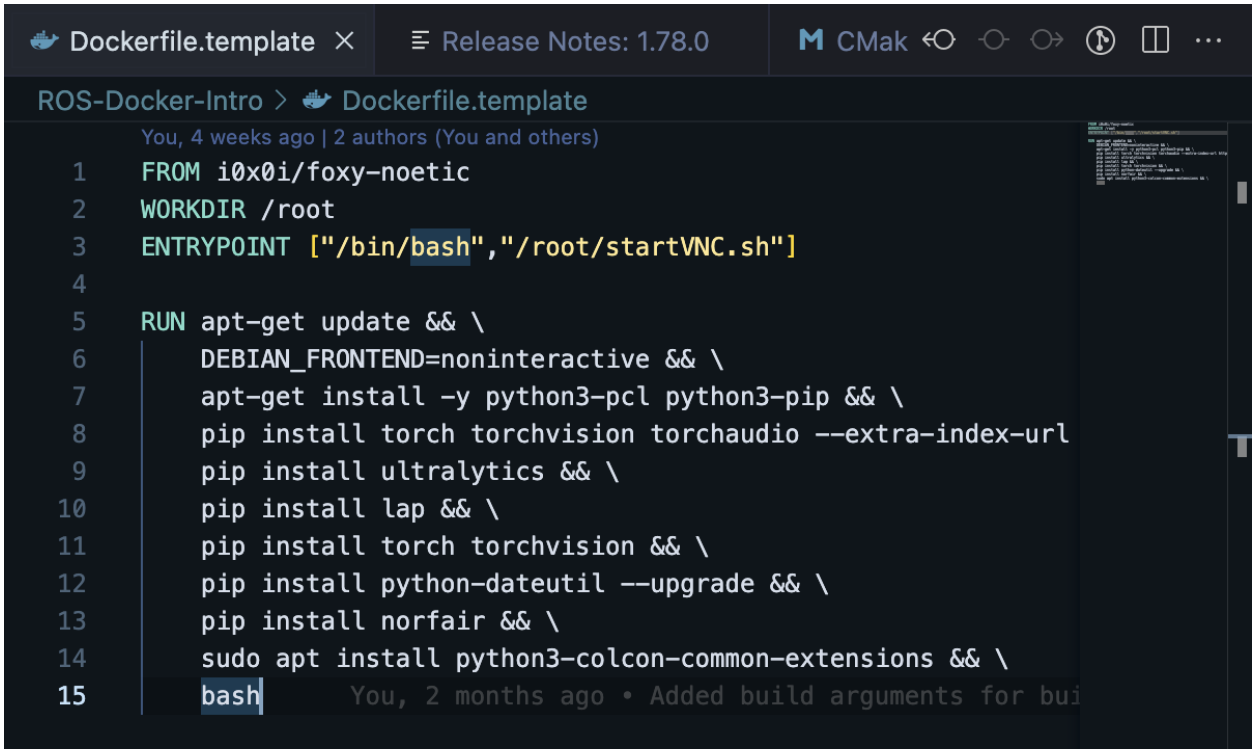


Figure 5: Dockerfile

“From” specifies the base image our custom image is built on.

After specifying the base image, we install the python libraries required for the project, and colcon build tools to build and run the project. We end the installation commands with “bash” to open up the bash shell for use in the container. This will allow us to use bash instead of the default shell and hold the container open.

We also have a makefile with some preset commands that can be used for various docker operations:

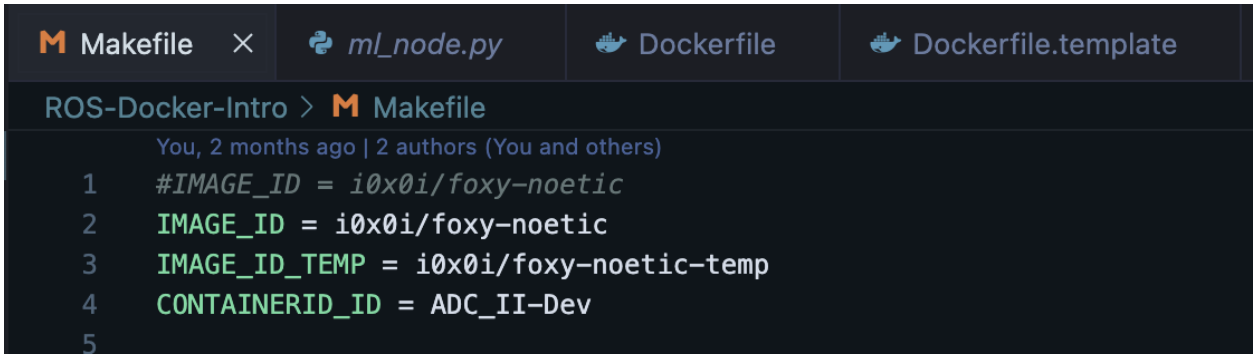


Figure 6: Makefile header

In the makefile, the image ID, container_ID can be specified by changing the values in the header.

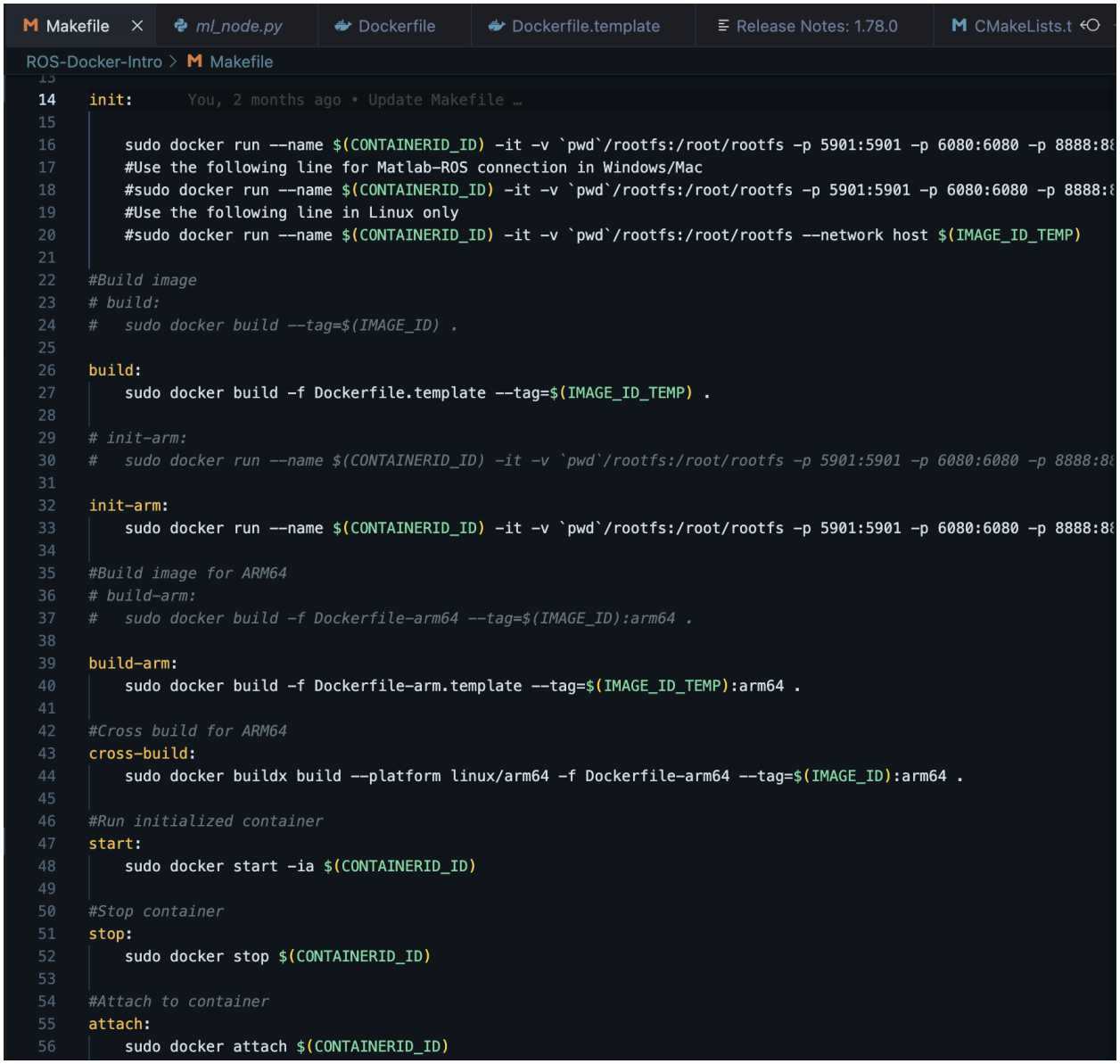


Figure 7: Makefile commands

“make build” builds the docker image with the associated dockerfile with the specified image ID

“make init” initializes a docker container with the built image

“make start” / “make stop” starts and stops the current initialized container

There is also an arm variant for the init and build commands, for running on arm-based devices such as Apple M1 computers. For running on arm devices use the arm commands instead.

To run these commands, open a terminal window inside the directory where the makefile is located.

Some additional commands available in the makefile:

```
58  #Start an CLI to container
59  terminal:
60      sudo docker exec -it $(CONTAINERID_ID) bash
61
62  #Remove container
63  rm:
64      sudo docker rm $(CONTAINERID_ID)
65
66  #Remove image
67  rmi:
68      sudo docker rmi $(IMAGE_ID)
69
70  #Show container status
71  status:
72      sudo docker ps -a
73
74  #Initialize a plain ubuntu 20.04 container
75  plain:
76      sudo docker run -it --entrypoint "/bin/bash" ubuntu:20.04
77
```

Figure 8: Makefile commands continued

Once the command “build” is run, there should be an image with the image id specified under the list of images in the docker app. The image created should then be run with the “init” command for the first time inside the makefile directory.

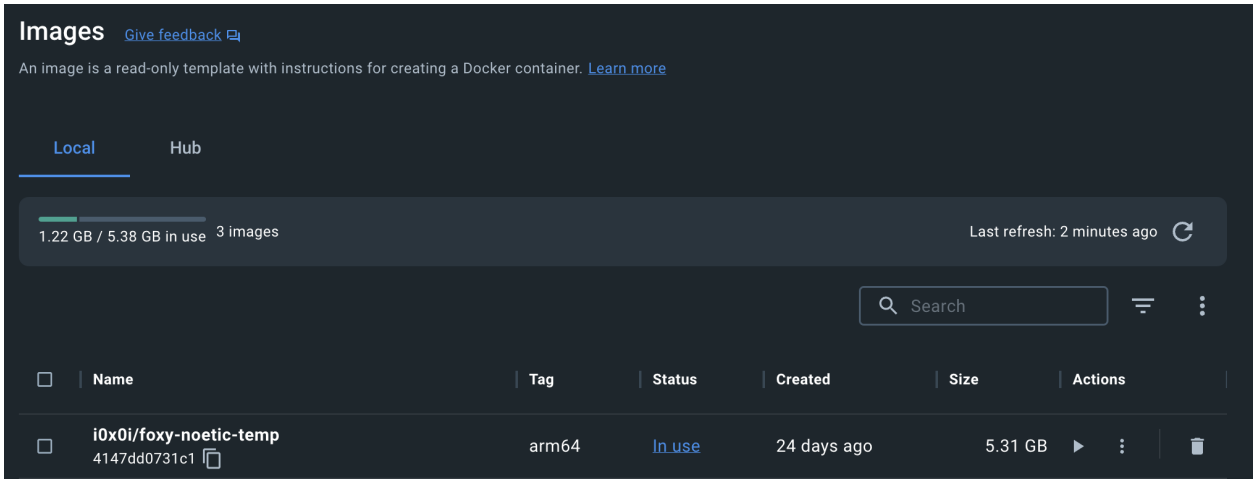


Figure 9: Docker images menu

After running with the “init” command once, the image can be run from the docker menu instead or with “make start”.

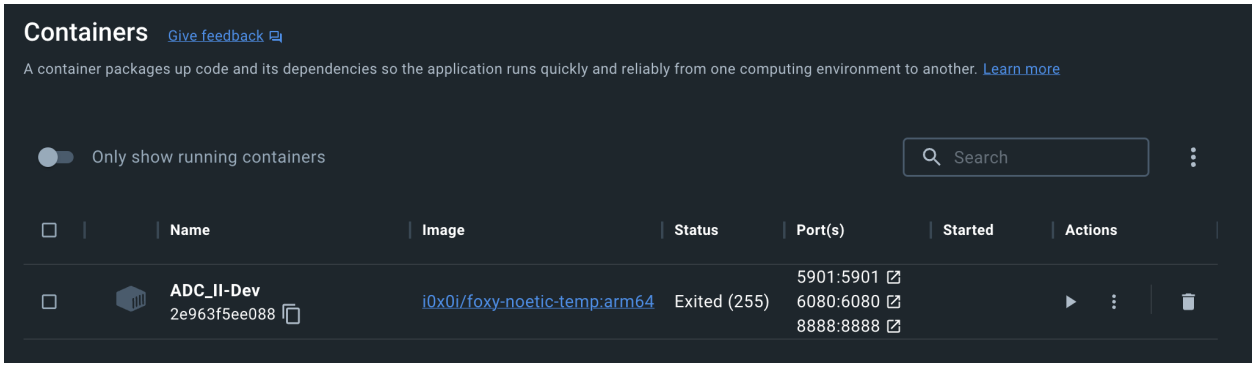


Figure 10: Docker containers menu

To stop the running container, use the “make stop” makefile command, or use the stop button in the docker containers menu.

This is what the terminal should look like after running “make start”:

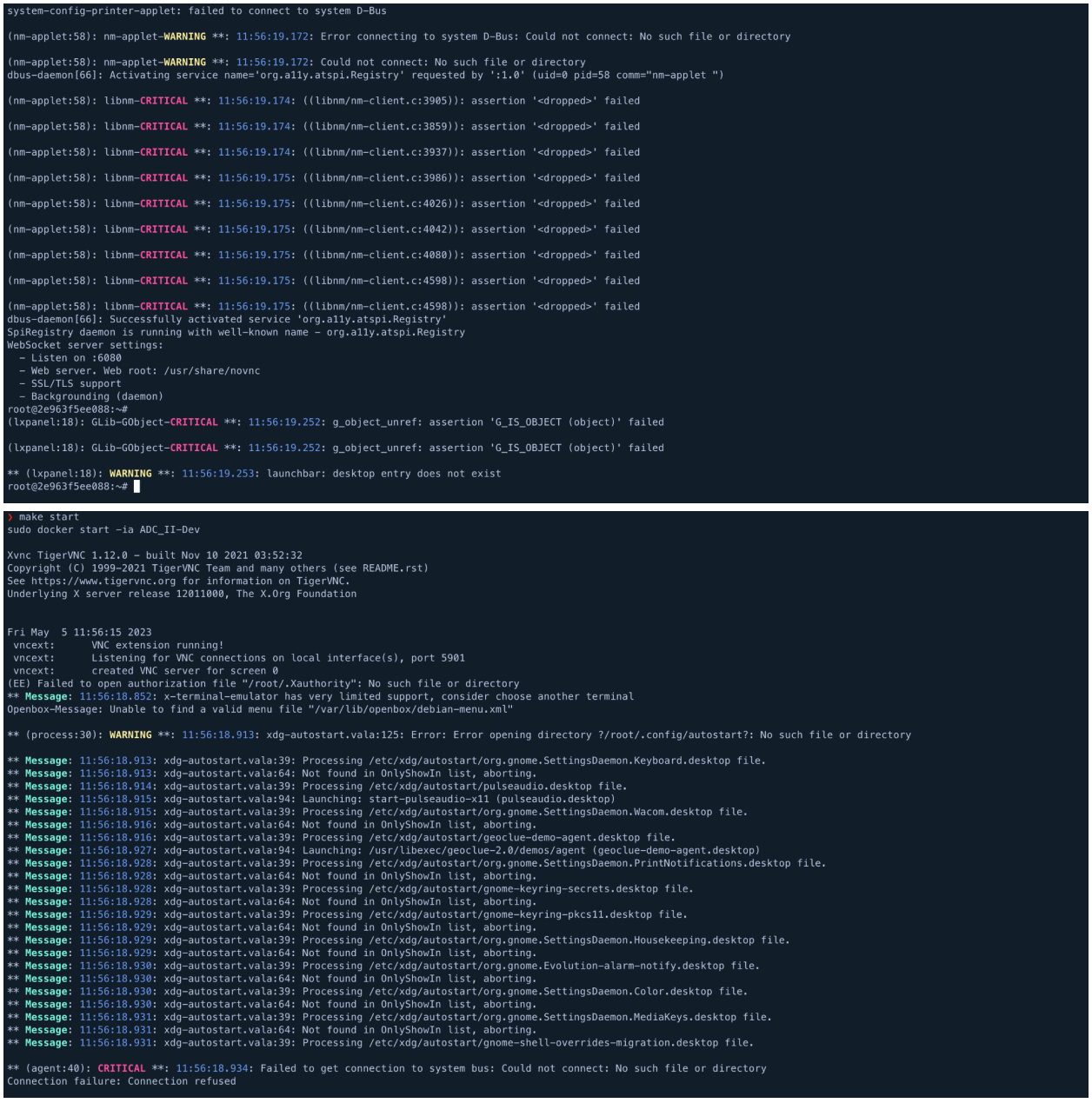


Figure 11: Terminal output for make start

The warnings can be safely ignored as they relate to the system bus which we are not using when testing locally.

Model/Dataset:

The yolov8 detection model is trained on a dataset gathered from provided rosbag files. The dataset is hosted in Roboflow, which allows users to upload images and annotate them. Additionally, Roboflow provides for different augmentation steps to help make the dataset more complete. These augmentations took our 358 gathered images and compiled them into a set of 735 training images, 69 validation images, and 42 testing images. The link to the dataset can be found on the github page.

There are a variety of preprocessing steps and augmentations available, and the settings we chose are included in figure 12 below. We tested several configurations such as downsizing images and doing mosaic augmentations, but found that those settings decreased model performance. In creating a dataset, ideally there should be more images overall in different scenarios and also a better balance of classes. There was not a lot of data to be gathered for more rare light types, which can be seen in the class balance section in figure 13. Having more images and a more even distribution of classes should provide more optimal performance across more varied scenarios. Additionally, it is encouraged to experiment with preprocessing and augmentation settings in seeking the best model performance.

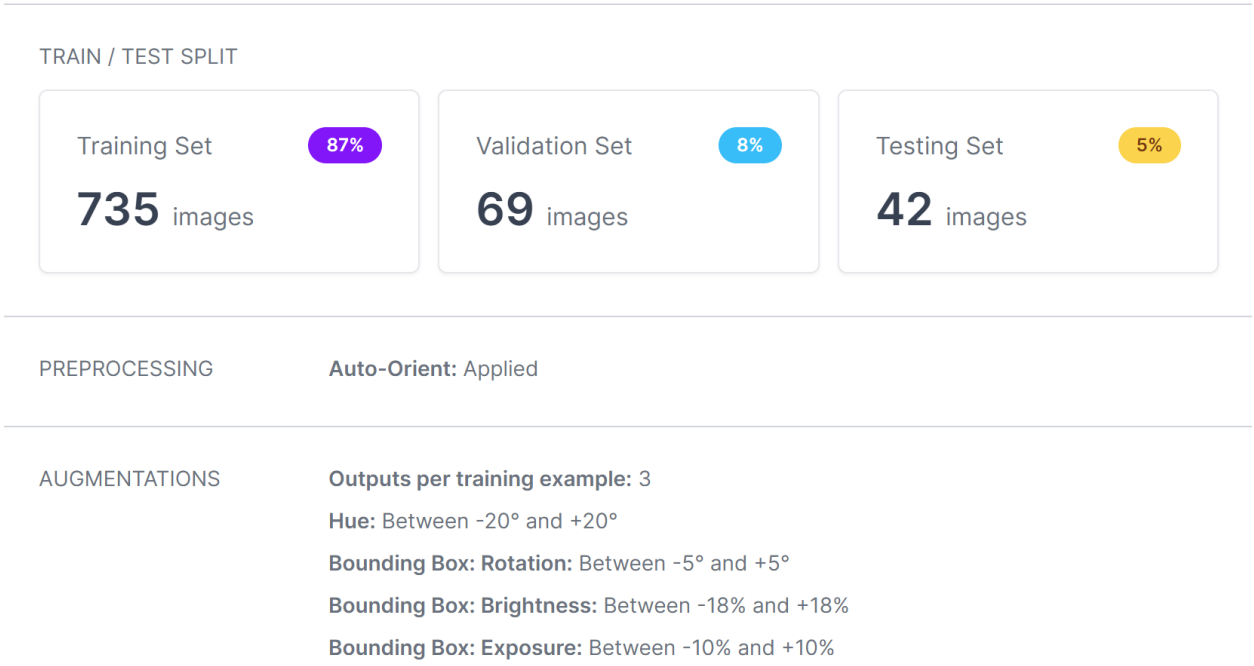


Figure 12: Dataset Split and Preprocessing and Augmentation Info

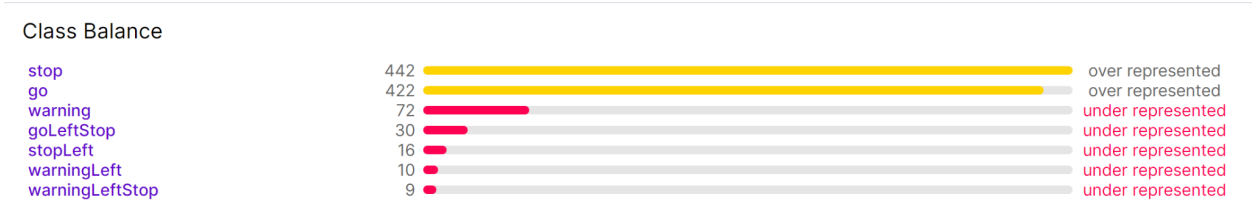


Figure 13: Class Balance

To train the models, we used the standard process for yolov8 model training from a roboflow dataset in Google Colab. The model was trained for 50 epochs on an image size of 1280 based on the yolov8s weights. The higher image size resulted in longer training times, but appeared to increase performance. It is also encouraged to experiment with different epoch amounts and image sizes in training.

Tracker:

In order to track multiple traffic light objects at once across frames we chose the Norfair multi object tracker. This tracker comes in the form of a Python library, is easy to integrate, and is quite tunable. In order to make use of the tracker, it is first necessary to convert our YOLO model detections to the format that Norfair uses for detections. Once this is complete, the detections can be continually passed to the tracker for updating.

The Norfair tracker has several tunable parameters which can be adjusted to determine behavior of the tracker. For our purposes, the first notable parameter is distance_threshold. This value determines the maximum distance to determine a match between a detection and a currently tracked object. In our case, we are using the standard euclidean distance function. If this value is too low, then fast movements or choppy video feed could result in missed matches and a new tracked object being created in place of a previous one where there should have only been the original object. If this value is too high, then detections of similar objects relatively far away could be misinterpreted as a match of an already existing tracked object and result in new objects not being made where they should be. We settled on a value of 550, but it is possible that this value could need tweaking in different scenarios.

The second notable parameter which could be adjusted is the hit_counter_max. Each tracked object has a hit counter which goes down by one each time a detection isn't matched to the object and goes up by one each time there is a match. If the counter goes below zero, the object gets destroyed. This value essentially determines how long an object can remain without any matches, not taking into account any ReID that may be implemented. The hit counter is important to be able to compensate for missed detections. It is important to not have this value too low so objects may get destroyed prematurely but it is also important not to have this too high to prevent objects from lingering. Due to the nature of the hit counter, there will be a few frames where an old traffic light detection object will remain/overlap with a newly detected traffic light object while the counter of the old object decreases. One scenario where the hit counter max may need to be adjusted is for flashing lights depending on the light and the frequency of flashes. For some scenarios a value of 4 was sufficient, but in others 8 was needed to detect flashing lights.

Additional Features:

There are some additional features that have been added to the program which can be adjusted as needed. The first of these is frame cropping. We have been testing on the default 2048x2048 bag files, but are using an area of the image from 256 to 1024 on the y axis, and 512 to 1536 on the x axis, assuming (0,0) is the top left of the image. This removes the bottom half of the image, some of the very top, and some from both sides to not only speed up performance by having less pixels to process but also to focus only on the most relevant areas of the image. The segment of the image to crop to can be adjusted based on the desired viewing area. Related to cropping, when publishing our messages, we add 256 to the y value coordinates and 512 to the x value coordinates to transform them back to their correct positions in the uncropped image. Due to our image cropping, if there is ever a scenario when the full uncropped image is being viewed, there may be times where objects on the edges of the frame may be undetected if they are outside of the cropped area actually being processed.

The second additional feature is a filter for detections which seeks to pass only the relevant traffic light detections to the tracker. For example, we do not want to track traffic lights at the intersection beyond the one we are at, as those are not the ones determining our movement. To accomplish this, we employ a simple filter that removes any detection box that has an area less than less than 40 percent of the maximum for all currently detected boxes. We found this number to provide good results in a simple solution, as the largest traffic lights are typically the ones that are most relevant. This threshold area value can be adjusted as needed as well. In the code, there is also another version of the relevance filter not currently in use that uses the interquartile range of areas or some other user defined range to group the bounding boxes into sets and only preserves the set with the highest areas. We found this to be too restrictive in our testing, but the idea could be potentially useful for some applications.

```
123     tracked_objects = tracker.update(detections=detections)
124     if (not idsToLastHitCounterValue and not idsToMissedDetections):  # if hashmaps are not initialized
125         for track in tracked_objects:
126             idsToLastHitCounterValue[track.id] = track.hit_counter
127             idsToMissedDetections[track.id] = 0
128     else:
129         idsToDelete = set()    # Norfair will delete an id if the tracked object goes out of frame, so we need to records these ids
130         for id in idsToLastHitCounterValue.keys():
131             idsToDelete.add(id)
132 
133         for track in tracked_objects:
134             if idsToLastHitCounterValue.get(track.id) is not None and idsToLastHitCounterValue[track.id] > track.hit_counter:   # If current hit_counter is less than past, it indicates
135                 idsToMissedDetections[track.id] += 1
136             elif idsToLastHitCounterValue.get(track.id) is None:
137                 idsToLastHitCounterValue[track.id] = track.hit_counter
138                 idsToMissedDetections[track.id] = 0
139             idsToDelete.discard(track.id)
140 
141         for id in idsToDelete:        # Delete ids from hashmap
142             idsToLastHitCounterValue.pop(id)
143             idsToMissedDetections.pop(id)
144     # draw_tracked_objects(cv2_img[256:1024, 512:1536], tracked_objects)
145     resultPubMsg = BoundingBoxes()
146     print(idsToMissedDetections)
```

Figure 14: First Part of Flashing Light Algorithm

```
147     for track in tracked_objects:
148         x1 = int(track.estimate[0][0])
149         y1 = int(track.estimate[0][1])
150         x2 = int(track.estimate[1][0])
151         y2 = int(track.estimate[1][1])
152         norfairLabel = track.last_detection.label
153         cv2.rectangle(cv2_img[256:1024, 512:1536], (x1, y1), (x2, y2), (0,255,0), 2)
154         boundingBoxMsg = BoundingBox()
155         boundingBoxMsg.probability = float(track.last_detection.scores[0])
156         boundingBoxMsg.xmin = x1 + 512
157         boundingBoxMsg.ymin = y1 + 256
158         boundingBoxMsg.xmax = x2 + 512
159         boundingBoxMsg.ymax = y2 + 256
160         boundingBoxMsg.id = track.id
161         ratio = idsToMissedDetections[track.id] / track.age
162         if ((track.age > 3 and ratio > 0.125 and ratio < 0.7) and not (norfairLabel == 0)):
163             cv2.putText(cv2_img[256:1024, 512:1536], str(track.id) + ': blinking ' + classes[norfairLabel], (x1, y1-10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (36,255,12), 2)
164             # print(str(track.id) + ': blinking ' + classes[norfairLabel])
165             if (norfairLabel == 1):
166                 boundingBoxMsg.group.append(15)
167                 boundingBoxMsg.group.append(3)
168             elif (norfairLabel == 6):
169                 boundingBoxMsg.group.append(13)
170                 boundingBoxMsg.group.append(3)
171             else:
172                 boundingBoxMsg.group.append(norfairLabeltoGroupField[norfairLabel])
173         else:
174             cv2.putText(cv2_img[256:1024, 512:1536], str(track.id) + ': ' + classes[norfairLabel], (x1, y1-10), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (36,255,12), 2)
175             # print(str(track.id) + ': ' + classes[norfairLabel])
176             if (norfairLabel == 1):
177                 boundingBoxMsg.group.append(15)
178                 boundingBoxMsg.group.append(6)
179             elif (norfairLabel == 6):
180                 boundingBoxMsg.group.append(0)
181                 boundingBoxMsg.group.append(6)
182             else:
183                 boundingBoxMsg.group.append(norfairLabeltoBlinkingGroupField[norfairLabel])
184         resultPubMsg.bounding_boxes.append(boundingBoxMsg)
185
```

Figure 15: Second Part of Flashing Light Algorithm

The third notable additional feature is flashing light detection. The algorithm runs for each Norfair tracked object in the callback_Img function as shown in Figure 14 and 15. The first part of the algorithm is located at the callback_Img function and utilizes two Python dictionaries called idsToLastHitCounterValue and idsToMissedDetections. These two dictionaries are actually declared outside the callback_Img function. If the dictionaries are empty or any tracked object was not stored in the dictionaries, then all the hit counter values of each tracked object are stored and their number of missed detections are set to 0. If any tracked object's hit counter value has decreased, its missed detection amount goes up. All objects that are stored in the dictionaries but are not tracked anymore by Norfair are deleted from the hashmaps in line 141. Next, the second part of the algorithm runs in a for-loop through the tracked objects. At line 152, we record the traffic light type for later use. Our algorithm makes use of the tracked object's hit counter and age at line 161 and determines the ratio of missed detections to age. This ratio is what allows us to determine if an object or light is flashing. In our testing, we found that ratios of missed detections to age in the range from 0.125 to 0.7 were sufficient overall. This also assumes a minimum age of three, meaning the tracked object has been around for at least three frames. The minimum age value is to prevent newly created objects that may miss one detection from being mistaken as a flashing object too early. After these values are met at line 162, the algorithm generates text that states the light is blinking. The next if-conditions select which classification class ID numbers to send into a custom ROS message for this specific detection box. This ratio and minimum age are tunable and may need to be adjusted in different scenarios. For example, in one bag file, the minimum ratio needed to be reduced to 0.125 from the value we originally had of 0.35, as the frequency of flashing for blinking yellow turn in this case was less than in another bag file we tested. The hit counter max also plays a crucial role in flashing detection, as the max value should be high enough to allow the tracked object to last long enough between flashing intervals in order for the algorithm to determine the object is indeed blinking. However, the hit counter max should not be set too high to reduce lingering bounding boxes in other scenarios. These tunable parameters make the flashing detection algorithm somewhat of a balancing act between ratios, minimum age, and hit counter max. Ideally, we would like for this flashing detection algorithm to be more plug-and-play without the possible need to adjust for different situations. Additionally, any lacking areas in the model where there may be dropped detections could lead to false

positives for flashing lights. This algorithm works, but adjustments may need to be made depending on the scenario.

Figure 16: ROS workspace directory

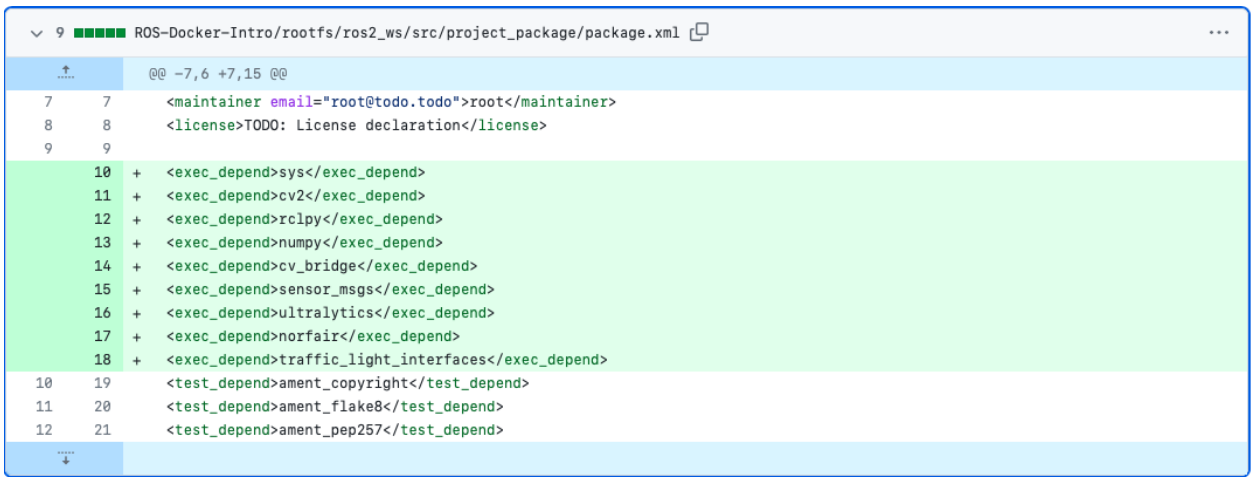


Figure 17: Project dependencies in package.xml

The fourth notable feature is the system's output of custom ROS messages called BoundingBox and BoundingBoxes. This deliverable was completed because our project code would be able to integrate with the ROS system. The system's machine learning framework node can act as a publisher of the BoundingBoxes messages. In order to accomplish this, two ROS packages named traffic_light_interfaces and project_result were set up. A project workspace was created with the command 'mkdir -p /root/rootfs/ros2_ws/src'. Within the ros2_ws directory, the command 'rosdep install -i --from-path src --rosdistro foxy -y' and 'colcon build'. The two packages were created with the command 'ros2 pkg create --build-type ament_python <package_name>. The packages were built in the ros2_ws directory with the command 'colcon build --packages-select <package_name>'. In the project_result package, the project's code is stored inside a project_result subdirectory. Within the project_result package, a file

called package.xml was changed to list the code's dependencies by using exec_depend tags as shown in Figure 17. In the traffic_light_interfaces package, the lines 'find_package(builtin_interfaces REQUIRED)' and 'DEPENDENCIES builtin_interfaces' were added to CMakeLists.txt. The .msg files used for the bounding box detections were based on the previous years msg files here. One note is that the previous years msg formats were designed for ROS1 and used the 'Header' message type which used to be a built-in interface for ROS but is no longer standard for ROS2. To get around this we recreated the 'Header' msg with our own implementation 'Header.msg' which consists of two fields 'builtin_interfaces/Time stamp' and 'string frame_id' . This change resolved errors caused by the lack of a build in 'Header' msg type in ROS2

The code was able to utilize the BoundingBox and BoundingBoxes format because traffic_light_interfaces was one of the dependencies of project_result. The code initialized a BoundingBox message for each tracked object in the callback_Img function. After the Yolo model makes its detections during the 'callback_Img()' , low scoring detections are removed,and then passed to the tracker, the code then iterates over each 'track' object in the 'tracked_objects' object. For each 'track' object in 'tracked_objects' the edges of the bounding box are extracted into x1,x2,y1,y2 values. A new 'BoundingBox()' ROS msg object is created and its 'xmin','ymin','xmax',ymax' fields are filled with 'x1+512','y1+256','x2+512','y2+256' respectively. The x values have 512 and the y values have 256 added to them before being set to the BoundingBox to compensate for the image offset. The BoundingBox probability field is filled with a float value of the 'track''s last detected score. The 'id' field is assigned to the 'track' object id. Within the if-statements of Figure 15, numbers that designate the light type are appended to the 'group' array field of the BoundingBox message. Next, the BoundingBox message is appended to a field of a BoundingBoxes message named 'resultPubMsg' that was declared on line 145. We utilize two fields of the Image message parameter of the callback_Img function called 'header.stamp' and 'header.frame_id'. The content of the two fields are initialized to the same fields within the BoundingBoxes message. Finally, the 'resultPubMsg' message is published from our machine learning framework node.

```
222  rclpy.init()
223  colorImgSub = rclpy.node.Node('image_subscriber')
224  colorImgSub.create_subscription(Image, '/front_camera/image_raw', callback_Img, 1000)
225  resultPub = rclpy.node.Node('result_publisher').create_publisher(BoundingBoxes, '/project_result', 10)
226  # grayImgPub = rclpy.node.Node('image_publisher').create_publisher(Image, '/img_gray', 10)
227  rclpy.spin(colorImgSub)
228
```

Figure 18: Node Setup Code

The fifth and final part of our tracker code is the ROS node network integration. This starts at line 222 of 'ml_node.py' with the 'rclpy.init()' which initializes ROS2 for the context of the current python file. This is needed before any nodes can be created or manipulated. On the following line a new ROS2 node is created called 'image_subsciber' and assigned to 'cologImgSub'. This will be used next on line 224, 'image_subscriber' is instantiated as a subscriber node to an image topic which the feed from the rosbag files is being played to, and with the callback function 'callback_Img'. On line 225 is when the publisher node component of our tracker, 'result_publisher' is created as 'resultPub' and instantiated to the '/project_result' topic as output. This object will be used by the callback function 'callback_Img' to publish the detection results after converting them to a 'BoundingBoxes', which is a list of multiple 'BoundingBox' objects. By having the subscriber node callback function use a publisher node object, this allows our tracker to be a subscriber and publisher within the ROS node network. Finally on line 227 'rclpy.spin(colorImgSub)' will cause code to indefinitely run the 'callback_Img' function and block until the context is terminated.

5 Experimental results (5-10 pages; 20 points)

For our project, the means of quantitative validation and testing have been the statistics from training and the outputs from live runs. The training metrics of our YOLOv8 detection model includes mean average precision (mAP), recall, and others. These are helpful for evaluating the initial performance of the model

based on the testing and validation sets of the dataset. In addition, the inference speed of detection is outputted when running the program so it can be observed which models are faster. The team set an inference speed benchmark of ~100 milliseconds per frame. Another important testing and validation step is to observe the model performance on different ROS bag files and scenes to see how the model handles different light types and scenarios. The figures 19-21 below include statistics and graphs output from training a model.

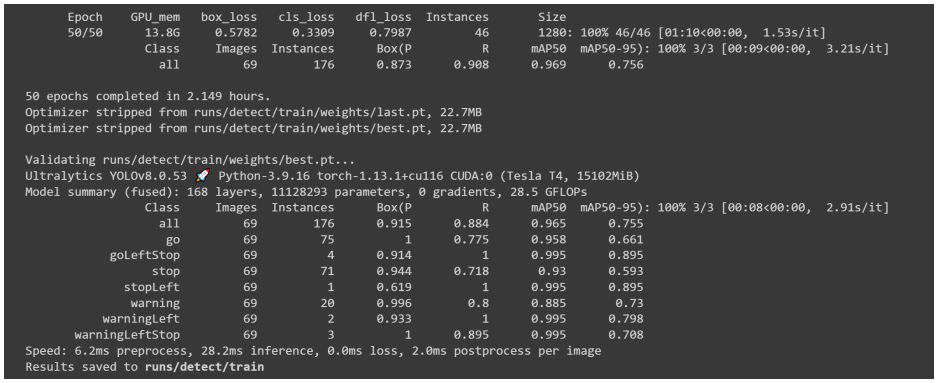


Figure 19: Model Summary Output

The figure above displays important training metrics in the columns Box(P, R, mAP50, and mAP50-95). These columns correspond to the metrics of precision, recall, mAP50, mAP50-95. These four measurements are critical towards evaluating the accuracy of our traffic light detection model. In order to calculate the numbers shown in Figure 19, we ran the command '!yolo task=detect mode=val model=/content/runs/detect/train/weights/best.pt data={dataset.location}/data.yaml' in Google Colab. The command takes the areas of each predicted bounding box from our model and compares them to the ground-truth bounding box areas. The ratio of the area of overlap between these two boxes to their area of union is calculated to indicate the Intersection over Union (IoU) values. A certain IoU threshold ranging from 0 to 1 will be configured and will be the minimum value needed to conclude a bounding box detection is a true positive. The precision measurement is equivalent to the number of true positives detected over the sum of the true positives and false positives. On the other hand, recall is equal to the number of true positives over the amount of true positives and false negatives. There is an inherent tradeoff between these two metrics as we increase the model's confidence score threshold (different from IoU threshold). As the threshold increases, recall decreases and precision increases. For each classification class, recall and precision value pairs were graphed on a 2D grid by Colab and the average precision value was collected. This value is known as mAP (mean average precision). The mAP50 metric is the mAP value for an IoU threshold set to 0.5, while the mAP50-95 describes the average of mAP values across IoU thresholds from 0.5 to 0.95.

Data from Figure 19 produces strong evidence that our model was highly accurate. The precision and recall values of all our classification classes were consistently high, as they were close to 1. Overall, the average precision value was 0.911 and the average recall value was 0.884. This conveys the model has a slightly higher ability to accurately detect positives than the ability to detect all positives, so it will leave out a miniscule amount of positives as false negatives. Both values could not be 1 for any classes because there is always a tradeoff between them, as explained in the last paragraph. The mAP50 values indicated the model's high validity with thresholds above 0.5 since those values were also close to 1.The average mAP50 value was 0.965, while the average mAP50-95 value was 0.755. The latter value could be

increased by providing more annotated images in underrepresented classes in our Roboflow repository. These high accuracy values suggest that our model is a valuable tool for classifying traffic signal light types.

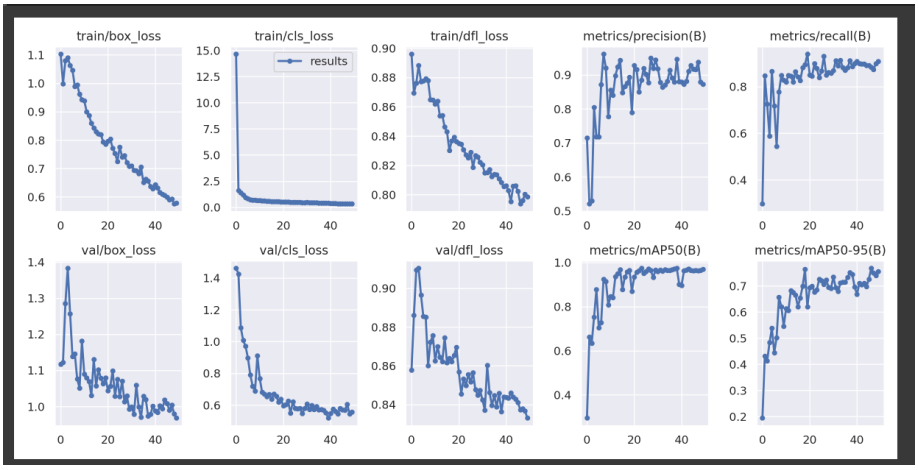


Figure 20: Model Training Graphs

The graphs from Figure 20 depict the trend of increasing accuracy of the project's model over a course of 50 training epochs. An epoch is a period of time where the model completes one pass over its training data. The set of 6 leftmost graphs refer to a decrease of the model's detection inaccuracy over the training period. The 3 top graphs represent data collected when running the model on training images to detect traffic signal types, while the bottom 3 graphs show data collected when detecting the signals on validation set images. The box_loss graphs show the difference between predicted detections versus the ground-truth detections, while the cls_loss graphs show the distinctiveness between predicted class labeling vs ground-truth class labeling. The 4 rightmost graphs exhibit the increase of the model's accuracy measurements over the training period. At the beginning of the training, the value trends fluctuate, but over time, they all stabilize to an increasing trend. At the end, the four metrics that were discussed in the previous paragraph all reach values near 1, indicating the success of the model's training.

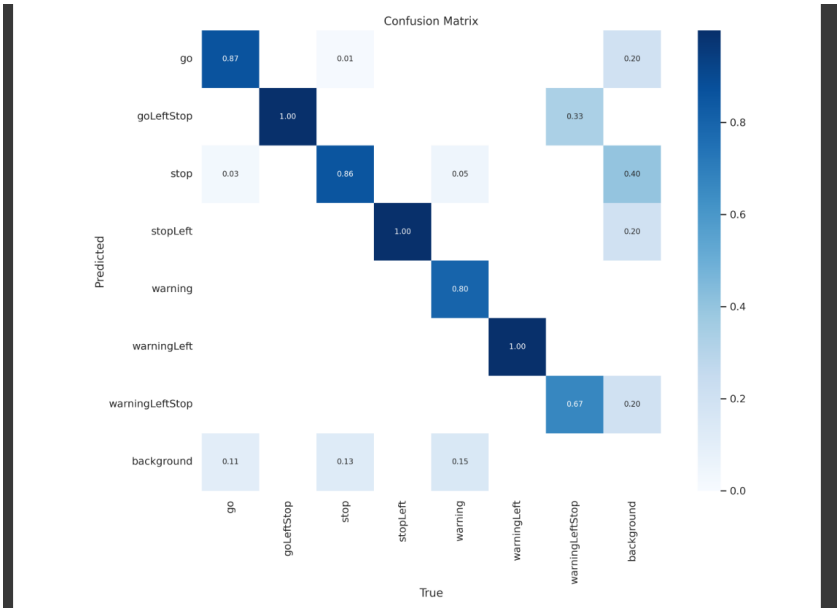


Figure 21: Confusion matrix

The Colab command also resulted in a confusion matrix for the project's model. A confusion matrix is a grid that summarizes the ratio of correct detections across the various traffic signal classes of our model. The x-axis of the matrix comprises the ground-truth occurrence of each classification class, while the y-axis of the matrix constitutes the predicted detections of each classification class. The diagonal of the matrix represents correct detections of traffic light types, while other grid cells contain the ratio of incorrect predictions of the associated y-axis class over the total number of ground-truth occurrences of the corresponding x-axis class. The bottom right cell related to the background class is not supposed to be highlighted. Most of the matrix's diagonal is captured with all ratios nearing 1. With this grid, we can identify which traffic light types need more annotated data in Roboflow to improve the accuracy of our detections. A reason why we ended up with such inaccuracies is the bottleneck of downloading large ROS bag files on personal computers with limited storage. This bottleneck prevented us from updating the model with images from these files, as we were approaching the end of the semester.

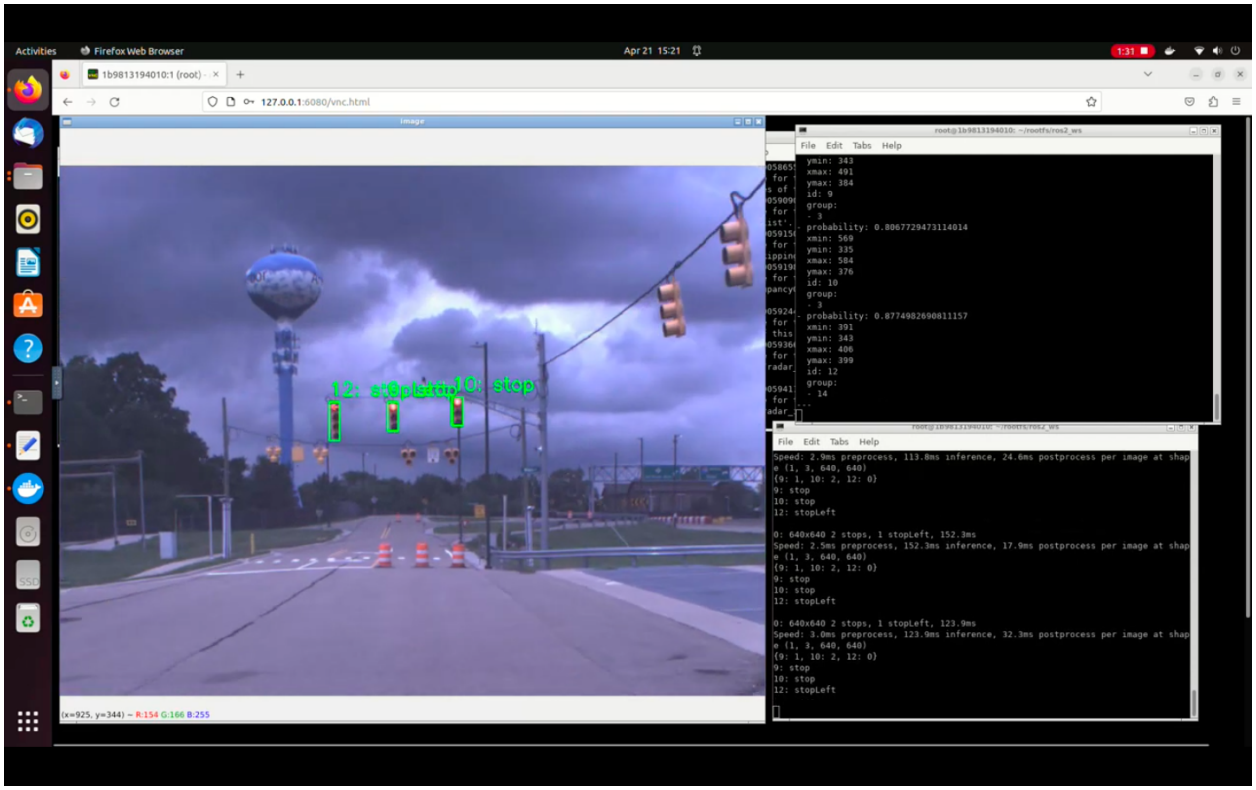


Figure 22: NoVNC view of Model Output

The screenshot of the noVNC interface in Figure 22 depicts the result of running our project's system. On the left half, our machine learning model draws tracked boxes from the Norfair module on converted OpenCV frames upon detected traffic lights. This window shows the model detects lights of two types: stopLeft and stop. This frame shows just a sample of the many classification classes of traffic lights that our machine learning model can detect. On the other half, the contents of our custom BoundingBoxes ROS messages and the YOLOv8 detections are generated on the top and bottom windows, respectively. In the top window, the probability fields of the BoundingBoxes ROS messages consistently show the high confidence scores of detections that are close to the ideal value of 1. This provides us with another metric that reinforces the conclusion that our traffic light detection model has a high level of validity. The fields of xmin to ymax in the BoundingBoxes messages define the coordinates of the detection boxes. The id field contains a unique number for each box and the group field provides a number that maps to the box's traffic light type. In the screenshot, a group field of 3 means the light is a stop signal while a field value of 14 means the light is a stopLeft signal. Next, the bottom window results in YOLOv8 detection messages. The first line of the message contains the image dimensions, amount of signal types, and the model's inference speed for its associated frame. The screenshot shows that the inference speeds are on average 130 milliseconds, which appears to not meet our speed benchmark. However, this is due to the slower performance from the screen recording at the time the screenshot was taken, as shown by the red logo on the top right. The second line elaborates on the speed measurements of the frame detection and the third line (used for debugging purposes) specifies the contents of a hashmap that tracks the missed detections of each traffic light. The remaining lines detail the traffic light types of each box.

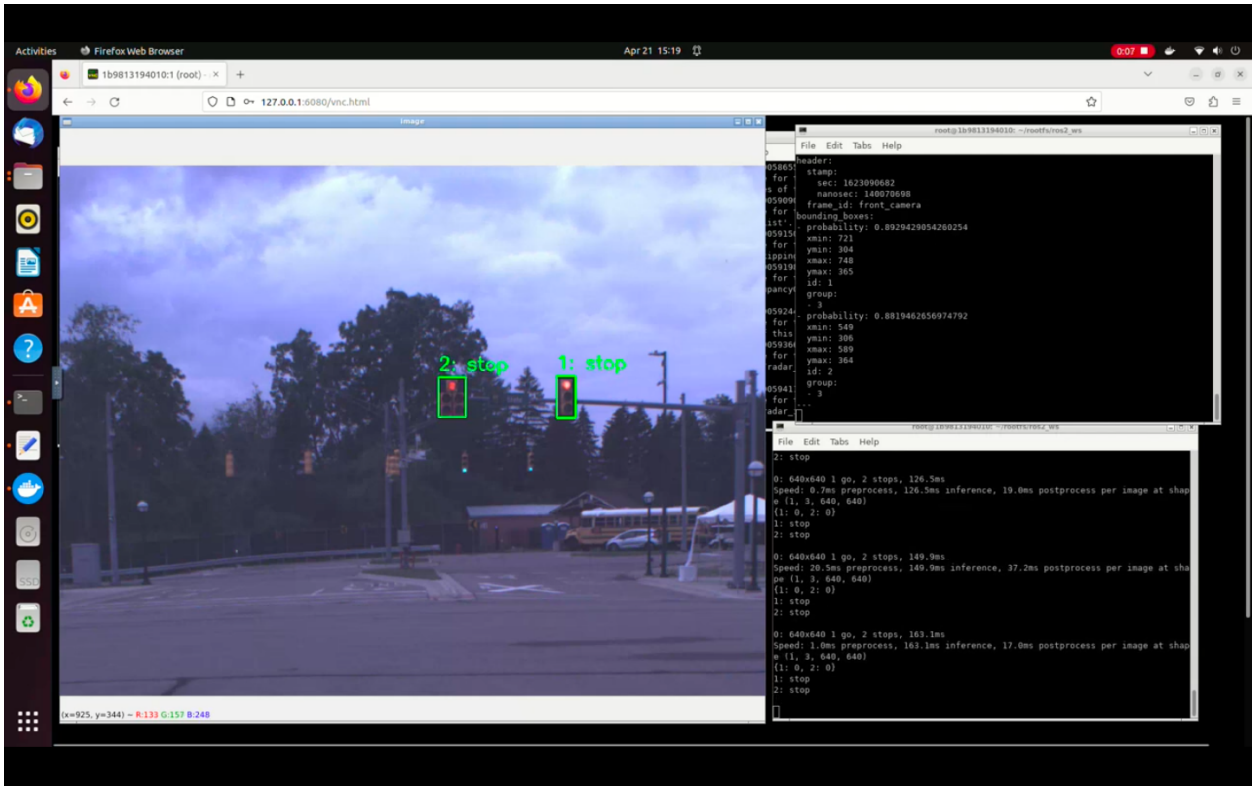


Figure 23: Distance Filtering Algorithm at Work

Figure 23 provides a frame that illustrates a result of our project's distance filtering algorithm. In the frame, there are four potential traffic lights that the model can draw boxes upon, which are the two marked red lights in the foreground and the two green lights in the background. Our algorithm ensured that only the foreground red lights would be sent from the YOLOv8 model to the Norfair tracker. This is because the algorithm noticed that the size of the green light bounding boxes did not meet the threshold of being at least 40% of the size of the largest red light boxes. Further evidence of our algorithm at work is displayed in the bottom right window containing the YOLOv8 detection transcripts. All three frames report that the YOLOv8 model detected one go light and two stop lights. The other green light was most likely not detected because of its small size compared to the lights shown to the YOLOv8 model during training. Next, the green light was ignored after processing by the distance filtering algorithm.

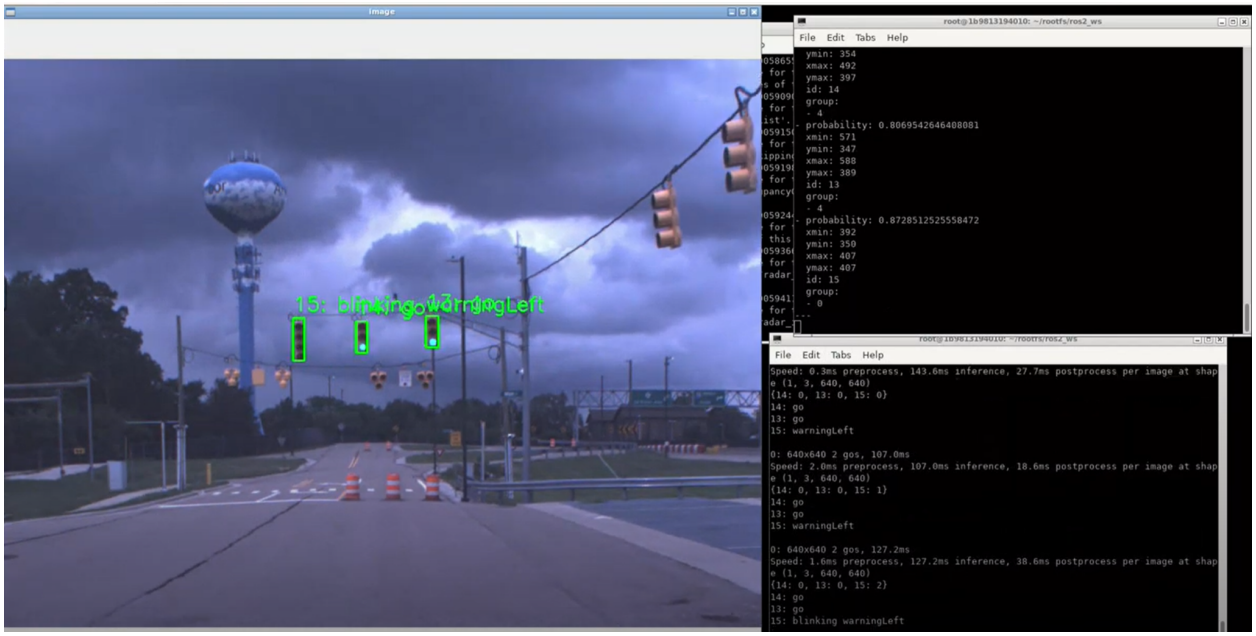


Figure 24: Blinking Light Detection

Figure 24 shows an example of blinking yellow light detection working on a blinking warning left turn arrow light. It can be seen from the text on the image screen at the left side and from the bottom right terminal output that the state of the warning left light has been changed to blinking. The algorithm was able to recognize the traffic light was flashing because of our use of the Norfair tracker. The tracker allowed the team to record the amount of missed YOLOv8 detections across multiple frames. A hashmap storing missed detections amounts to the unique traffic light IDs is also shown in the bottom right terminal window at the fourth line of the YOLOv8 messages. Our project code was able to utilize the hashmap and calculate a ratio of missed detections to the age of the traffic light. Because the warning light had two missed detections, its ratio was within our algorithm's threshold range and the system concluded that a blinking warning light was present.

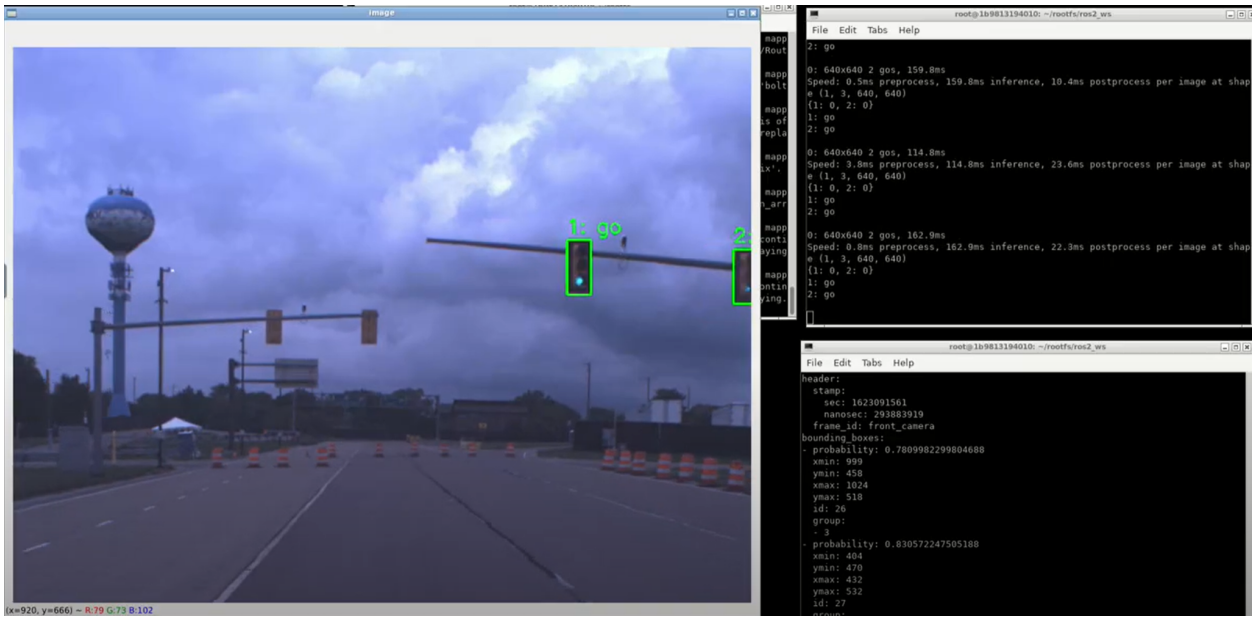


Figure 25: Tracker at Work, Forward Motion

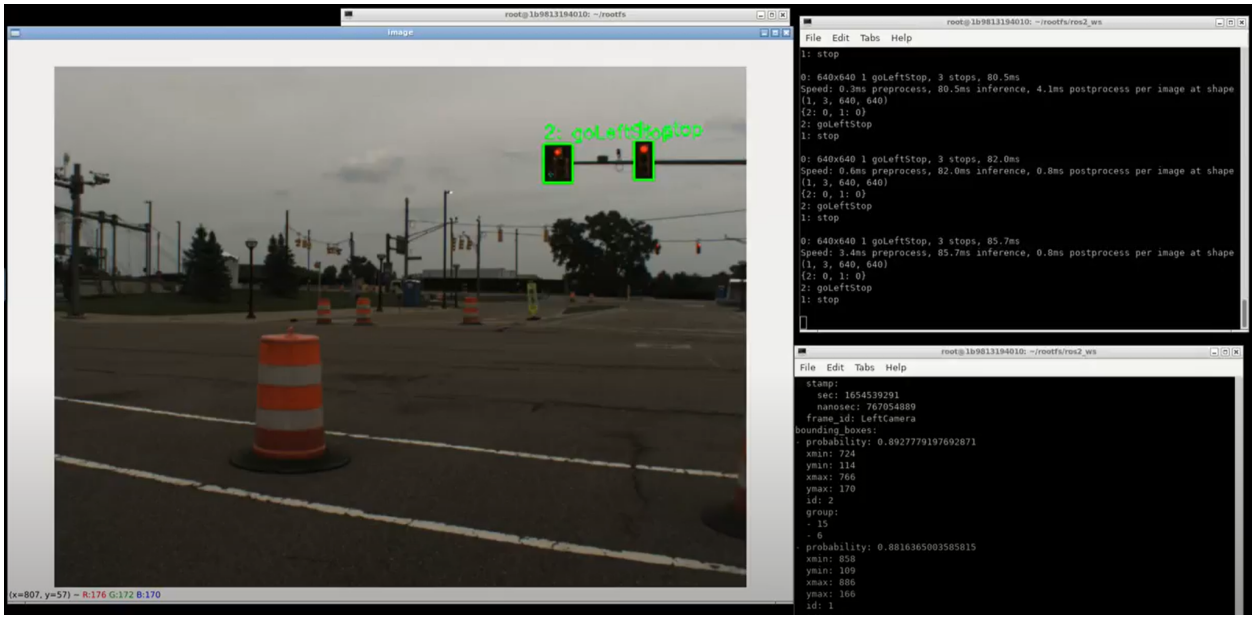


Figure 26: Different Lighting Scenario

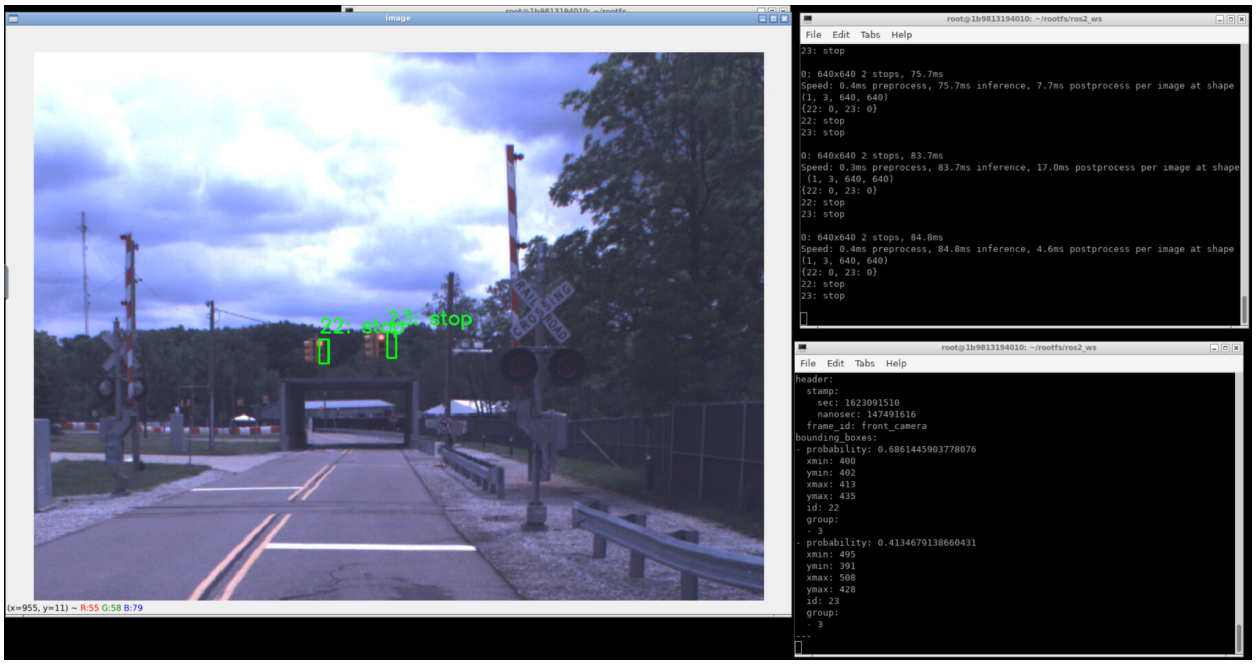


Figure 26: Tracker Bounding Box Drift/Adjustments

Figure 25 demonstrates a scene in which there is forward motion and shows the Norfair tracker accurately keeping the bounding box in place on the traffic lights. Figure 26 shows the program running in a different lighting scenario detecting a stopLeft and stop. This serves as an indication that the project's code can operate in various settings that an autonomous vehicle can drive in. The distance filtering algorithm is also at work here, ignoring the lights it detects in the background. Figure 27 demonstrates some bounding box drift from the tracker that may be seen at times as the tracker attempts to predict the motion of traffic lights. This box offset is normally corrected over a series of frames if it occurs, but may

be seen drifting more if there are framerate dips or sharp turns. Framerate dips can especially cause the tracker to have issues, as the tracker trajectory predictions cannot account for the unnatural movement.

Figure 28: Model Confusion

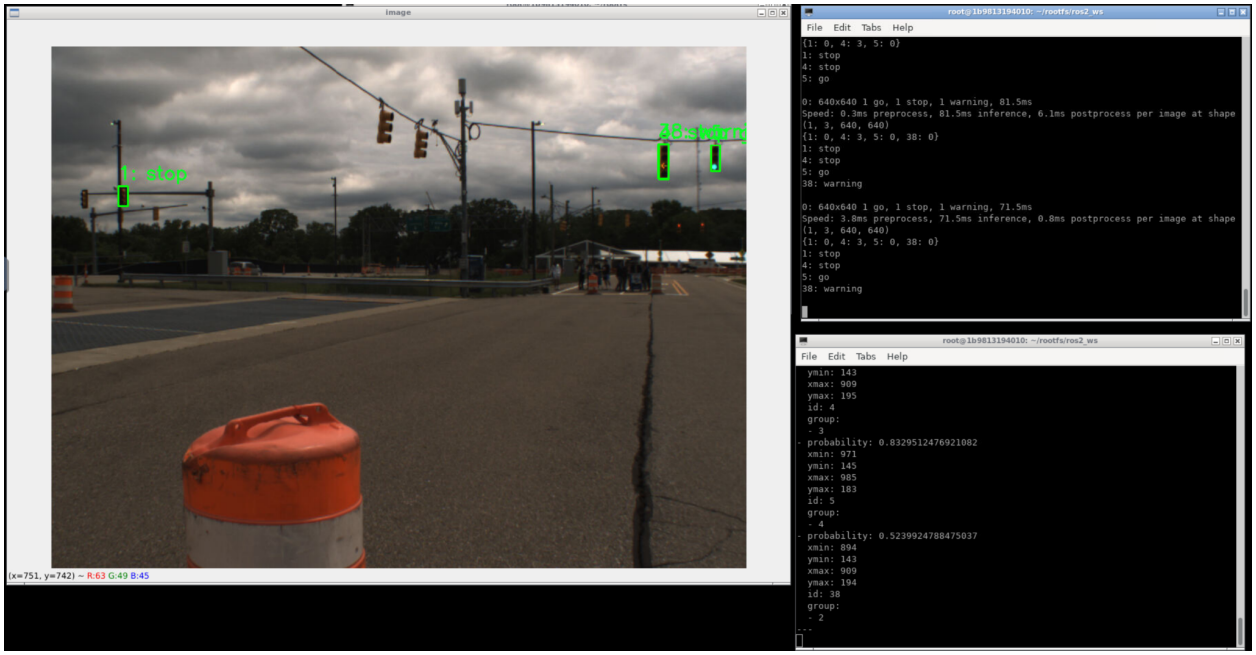


Figure 29: Model Confusion

The figures 28 and 29 above demonstrate scenarios in which our detection model gets confused. This is likely due to a lack of training data as our data was collected only from prior rosbag files. In the first image, there is a warningLeftStop dog house light which was a rare find in the bag files and is underrepresented in our dataset. Due to this, our model thinks it is both a stop and a warning. In the

second image, the warningLeft arrow light appears more red than yellow to our eyes, so it is possible that the model thinks it is a stop and a warning due to this and a lack of training data in more diverse lighting conditions.

6 User's Manuals (5-10 pages; 20 points)

Requirements:

Internet access is necessary to download software and install necessary packages for the Docker container

Minimum OS requirements is having access to a terminal with the "make" command which is necessary for building. Such as Linux systems, OSX, and WSL on Windows.

The Necessary software to be installed is the appropriate Docker Engine for your OS, which can be installed with instructions on their website.

Note for WSL users. After installing Docker Desktop for windows, WSL integration must be enabled in settings.

1. Click on The gear icon in the top right corner.
2. Click on the resources tab and select WSL integration.
3. Check Enable integration with my default WSL distro and enable the additional distros toggle for the Linux distros you plan on using.
4. Click Apply and Restart to save and enable the changes.

Installation:

Setting up the Repository

Clone the github repository onto your machine using your preferred method, either git-cli or github desktop.

Note. Long path names must be enabled in git settings before cloning the repository

```
$ git clone https://github.com/tamu-edu-students/CSCE482_23S-2A1.git
```

Figure 30: git-cli command to clone the repository

Setting up the Docker Image

Once the directory is cloned onto your machine. Open the "CSCE482_23S-2A1" directory in a terminal. Next navigate to the "ROS-Docker-Intro" directory.

```
bluyo@LAPTOP-TI4H0B7H:/mnt/c/capstone/CSCE482_23S-2A1/ROS-Docker-Intro$
```

Figure 31: The terminal prompt should look something like this

To setup the Docker image run the following command: *make build*

This will download the necessary files and dependencies for the Docker container

After it has finished run the command: *make init*

To finish the installation process and enter the docker container and have access to the terminal.

To access the noVNC GUI open this url in your web browser http://127.0.0.1:6080/vnc.html. Click connect and when prompted for a password enter "password" without quotation marks to gain access to the GUI.

```
Starting applications specified in /root/.xinit
Log file is /root/.vnc/ecaa12b4527c:1.log

Use xtigervncviewer -SecurityTypes VncAuth -passwd /root/.vnc/passwd :1 to connect to the VNC server.

WebSocket server settings:
  - Listen on :6080
  - Web server. Web root: /usr/share/novnc
  - SSL/TLS support
  - Backgrounding (daemon)
root@ecaa12b4527c:~#
```

Figure 32: Example output after entering the docker container through your terminal

Setup of the docker container is now complete additionally

- To exit the Docker container from you terminal run the command: *exit*
- To stop the docker container run the command: *make stop*
- To start the container again and access its shell run the command: *make start*

rootfs and Adding .bag Files

The rootfs directory located in the project repository at "CSCE482_23S-2A1\ROS-Docker-Intro\rootfs" is a shared folded that is mounted in the Docker container so any files you put in here from outside the container will appear in the "~/rootfs/" directory and vice versa allowing you to share files.

To run the tracker you need to download a ROS2 .bag file. An example file 16-mcity1.bag can be downloaded here.

Once you have downloaded a ROS2 .bag file, move it to the "CSCE482_23S-2A1\ROS-Docker-Intro\rootfs" directory outside of the container.

noVNC

To execute the multiple commands necessary to run the Tracker software it is needed to open new terminal emulators in the noVNC GUI.

To do this from the noVNC GUI right click on any empty black space to open up the menu.

Click "Terminal Emulator" to launch a new terminal instance. Do this when needing a new noVNC terminal.

Running the Tracker

Video guide of the process of setting up the tracker from a freshly started Docker container using the original terminal and noVNC is provided: TRACKER SETUP DEMO VIDEO

1. **Building ROS Packages**

The first step to run the tracker from a newly started container instance is to build all the ROS packages that contain the tracking code and output messages.

To do so from a terminal in the container navigate to "~/rootfs/ros2_ws" and run the command: *bash buildPkg.sh*

This uses colcon to build the project_package and traffic_light interfaces packages. This step is only necessary when using a freshly started Docker container or after modifying any of the packages files.

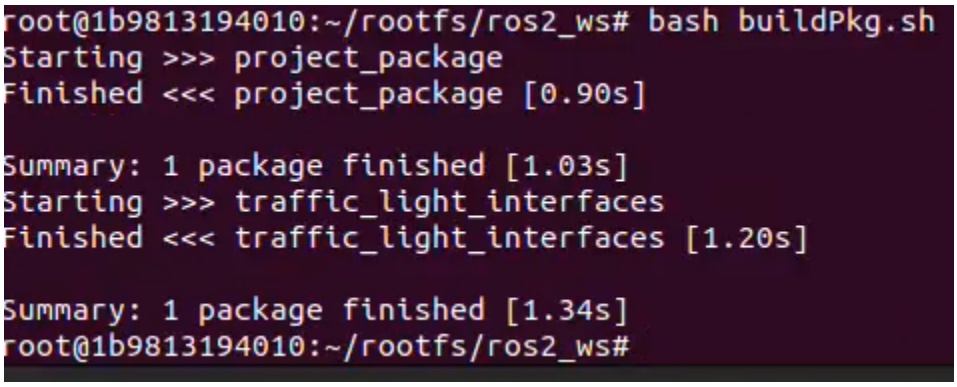


Figure 33: Command and example output

2. **Running a .bag File**

After finishing building the packages, the container is ready to run a ROS2 .bag file.

To do so open a new terminal in the noVNC GUI and navigate to the "~/rootfs/" folder where you should have put your .bag file.

Now run the command: *bash runBag.sh* (NAME OF BAG FILE)

This will play the bag file that is passed as the argument to runBag.sh

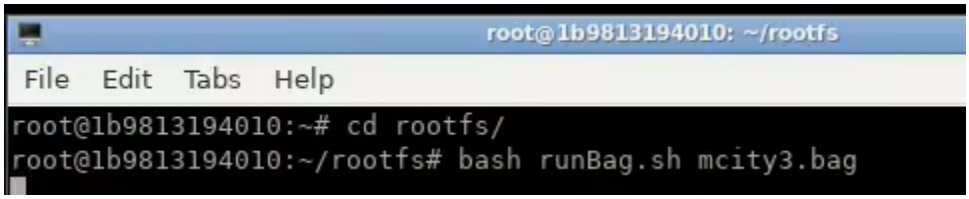


Figure 34: Command to play mcity3.bag, to run a different .bag file replace mcity3 with the name of your .bag

3. **Running the Tracker**

Once a .bag file is playing to run the tracker on that bag file open a new noVNC terminal and navigate to "~/rootfs/ros2_ws"

Now run the command: *bash runML.sh*

This will run Tracker and open a window showing the ROS bag feed and detections with their labels. Additionally in the terminal that ran the command will be outputting the trackers detection results and speeds for each iteration of the tracker

```
root@1b9813194010:~/rootfs/ros2_ws# bash runML.sh
WARNING ⚠️ Unable to automatically guess model task, assuming 'task=detect'.
 Explicitly define task for your model, i.e. 'task=detect', 'segment', 'classify
', or 'pose'.
Loading /root/rootfs/models/best19.onnx for ONNX Runtime inference...

0: 640x640 2 gos, 2 stops, 107.7ms
Speed: 2.5ms preprocess, 107.7ms inference, 24.6ms postprocess per image at shap
e (1, 3, 640, 640)
{}

0: 640x640 1 go, 2 stops, 122.4ms
Speed: 1.2ms preprocess, 122.4ms inference, 30.7ms postprocess per image at shap
e (1, 3, 640, 640)
{2: 0, 1: 0}
2: stop
1: stop
```

Figure 35: Command and example output

Figure 36: Window of ROS .bag feed with detection boxes

4. Displaying ROS2 Output

To show the published ROS messages by the tracker open a new terminal window while the .bag is playing and the tracker is running, and navigate to "~/rootfs/ros2_ws"

Now run the command: *bash runRes.sh*

This will display the messages being published to the "/project_result" topic

```
root@1b9813194010:~/rootfs/ros2_ws# bash runRes.sh
```

Figure 37: Command

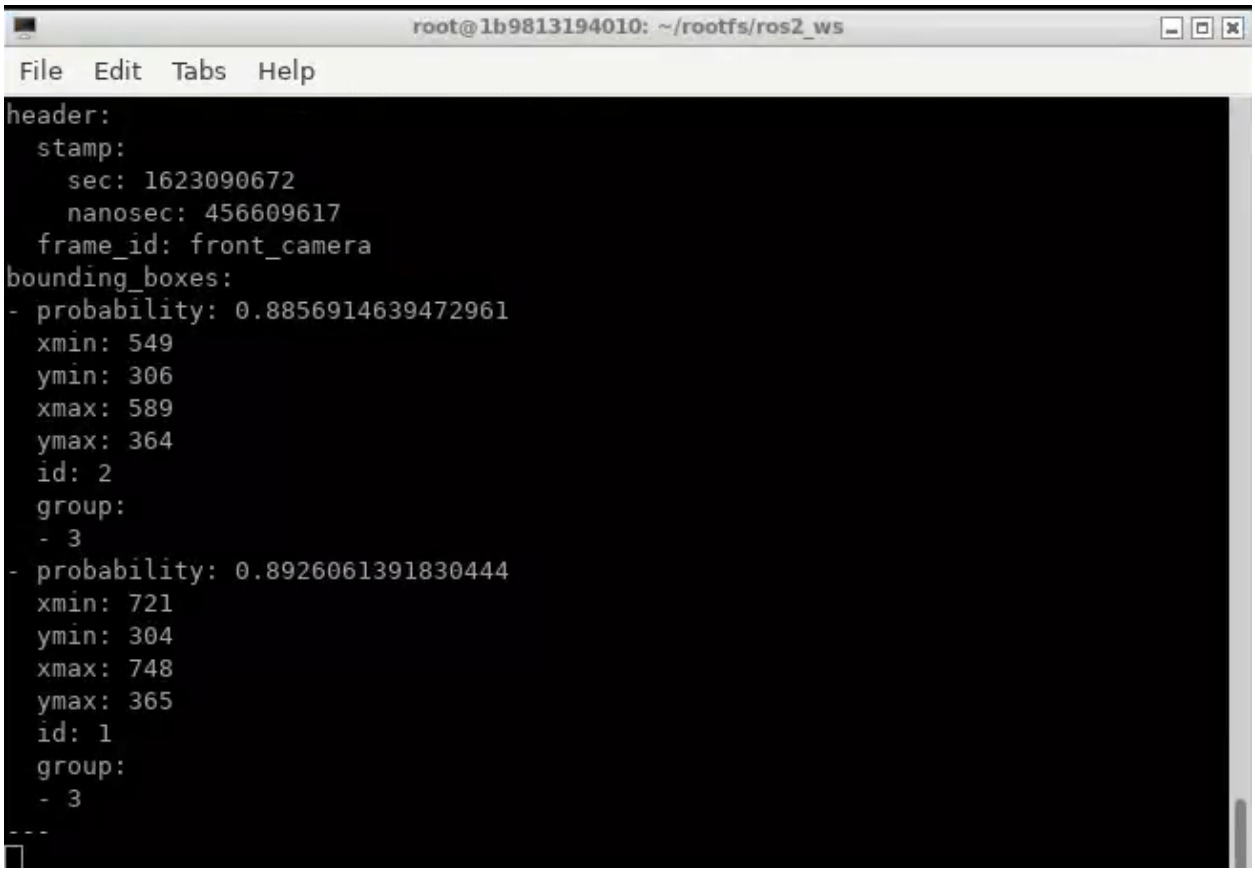


Figure 38: Example output

To stop any process running, navigate to the terminal that is running the process and press "ctrl+C" to exit.

https://drive.google.com/file/d/1q5UQj1fNsy59Akmthr6VYRco6zlQNHlq/view?usp=sharing

7 Course debriefing (2-4 pages; 10 points)

For the development of the project, the team employed a hybrid approach which combined both the waterfall and agile methodologies. Thanks to the project proposal which we had completed prior to the start of development, we had a fairly clear understanding of the project and what was necessary for its completion. Thus, it made sense to employ the waterfall approach. We used the benefit of the additional structure to our advantage. Eventually, we had a product which mostly adhered to the ideology in the proposal. Next, thanks to feedback and criticism from others, we gained insight into what the project was lacking. We then began to implement small and manageable changes to the project in chunks. This adheres much more closely to the Agile approach to development. We abided by this methodology for the

remainder of development. In retrospect, we feel that this hybrid approach allowed us to ensure a high quality product.

Our team management ideology primarily focused on the importance of communication, collaboration, and accountability. To ensure the flow of development and to promote collaboration, the team usually met three times a week. We believed that the best way to hold each other accountable was to have in-person meetings, so we met in-person on Tuesdays and Thursdays during the time slot for our capstone class. This was a natural course of action for us. We were required to meet in person at 8:30 am on Tuesdays for our weekly project updates with the instructor. Since we needed to meet regardless, we opted to extend the meeting past the project update to the full length of class, which ends at 11 am. The same applies to Thursday, though we opted to meet at 9:30 am, giving us a slightly smaller meeting window. Finally, we chose to meet on Sundays over discord. This meeting would allow us to share recent updates prior to our scheduled checkup on Tuesday, as well as to update the weekly report with recent findings and updates.

Figure 39: a screenshot of our discord server for 2A1 development

As shown in Figure 39, we opted to use discord as our primary channel of communication. This allowed us to talk over a voice channel during our Sunday meetings, and we used a text channel to communicate frequently via text.

If we were to redo this project knowing what we know now, there are many things we would have done differently. The team believes that we should have put more emphasis on the preferred method of grading for this project. This project uses a ratio of git commits to gauge the level of participation from team members. Our commits were not even distributed due to circumstances pertaining primarily to who worked on what. For example, much effort was put into fixing and debugging our ROS/Docker instance, but this is poorly reflected in terms of git commits. What's more, we had plans to use a Kanban board, which provides a visual aid for task assignment and management. We created one such board via Trello but seldom used it. We wish we had put it to better use, as it also shows a greater level of engagement from various team members. For productivity, we sometimes implemented pair programming. We found

this to be very efficient for reducing bugs and promoting efficiency, but we did not utilize as much as we should have. In regard to things we felt we did well, we believe we sufficiently researched how to implement deliverables. What's more, we also trained our members on critical components such as ROS, Docker, and Roboflow.

In regard to safety and ethical concerns, our product must demonstrate safety to be trusted for use by the public and become a viable product. Regulatory bodies may have doubts about safety and reliability of self-driving vehicles. We need to properly communicate capabilities and limitations to regulators and the public. and ensure that the overall product is safe for use before delivering. Our team took measures to give metrics associated with speed and accuracy and to understand the model so that it is not a "black box". We unanimously wish we had provided a more robust data set, as we felt the scope of ours was far too small and specific. With more varied testing and training data, the system would likely be more robust as a whole.

We tested our product on several test sets. The model works mostly as proposed, but is not without its own glaring flaws. The model has not been tested on situations which include less common traffic light types, as there are many for which we have very little data. We wish we had tested on more varied bag files sooner and perhaps added training data as we saw fit. We needed to boot off of an external hard drive for this, so we wish we had purchased it sooner. We were lacking in our testing ability until very late in the project as a result.

8 Budgets (2-4 pages; 5 points)

The only item that was purchased was an external SSD to use as a bootable drive for testing on the lab computer. Originally, we did not plan on spending any money on this project despite having 500 dollars available. However, based on our experiences with testing on our own laptops and personal storage limitations for rosbag files, we decided it would be beneficial to purchase an external SSD for testing and bag file storage. This did prove to be a good purchase, as it allowed us to store large rosbag files for testing and achieve far better performance on the more powerful lab computers in a native linux environment. This purchase should not point to increases in mass production cost, as the SSD was not a crucial component to the system itself but was only used for testing. The overall cost for mass production, not including any theoretical costs of running a company or hiring employees, would be minimal or zero as the software could be made open-source and downloadable for free.

Item	Cost
External SSD	$86.59

References

[1] Q. Wang, Q. Zhang, X. Liang, Y. Wang, C. Zhou, and V. I. Mikulovich, "Traffic Lights Detection and Recognition Method Based on the Improved YOLOv4 Algorithm," *Sensors*, vol. 22, no. 1, p. 200, Dec. 2021, doi: 10.3390/s22010200.

[2] M. Mostafa and M. Ghantous, "A YOLO Based Approach for Traffic Light Recognition for ADAS Systems," 2022 2nd International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC), Cairo, Egypt, 2022, pp. 225-229, doi: 10.1109/MIUCC55081.2022.9781682.

[3] K. Alkiek, "Traffic light recognition - A visual guide," *Medium*, 02-Oct-2018. [Online]. Available: https://medium.com/@kenan.r.alkiek/https-medium-com-kenan-r-alkiek-traffic-light-recognition-505d6ab913b1. [Accessed: 16-Feb-2023].

[4] M. Bach, D. Stumper and K. Dietmayer, "Deep Convolutional Traffic Light Recognition for Automated Driving," 2018 21st International Conference on Intelligent Transportation Systems (ITSC), Maui, HI, USA, 2018, pp. 851-858, doi: 10.1109/ITSC.2018.8569522.

[5] K. Behrendt, L. Novak and R. Botros, "A deep learning approach to traffic lights: Detection, tracking, and classification," 2017 IEEE International Conference on Robotics and Automation (ICRA), Singapore, 2017, pp. 1370-1377, doi: 10.1109/ICRA.2017.7989163.

[6] S. R. Maiya, “DeepSORT: Deep learning to track custom objects in a video,” Nanonets AI & Machine Learning Blog, 24-Apr-2020. [Online]. Available: https://nanonets.com/blog/object-tracking-deepsort/. [Accessed: 10-Feb-2023].

[7] Q. -C. Mao, H. -M. Sun, Y. -B. Liu and R. -S. Jia, "Mini-YOLOv3: Real-Time Object Detector for Embedded Applications," in IEEE Access, vol. 7, pp. 133529-133538, 2019, doi: 10.1109/ACCESS.2019.2941547.

[8] M. Bach, D. Stumper and K. Dietmayer, "Deep Convolutional Traffic Light Recognition for Automated Driving," 2018 21st International Conference on Intelligent Transportation Systems (ITSC), Maui, HI, USA, 2018, pp. 851-858, doi: 10.1109/ITSC.2018.8569522.