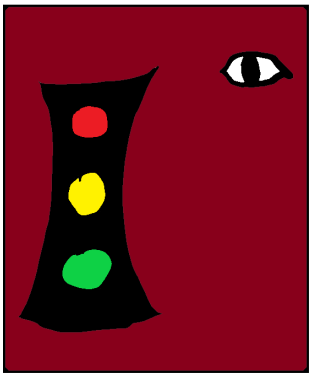
2A1: Traffic Light Detection and Tracking



Critical Design Review

2A1

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

Department of Computer Science
Texas A&M University

03/30/2023

Table of Contents

1 Introduction

1.1 General scope and problem background

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 confidence in recognition results and is quick and responsive.

We will simulate this task of traffic recognition in an ROS node system. Our inputs are ROS Bag file datasets, consisting of files that represent camera feeds that depict scenes of driving in urban environments. Our outputs will be in ROS Bag file format and will yield camera frames with all displayed traffic lights automatically labeled with relevant information such as signal color. We have decided that the YOLOv8 detection model and the DeepSORT algorithm will consist of our machine learning framework. The framework will be trained on the provided LISA Traffic Light Dataset and the framework will be incorporated into an ROS node system. Our machine learning framework will assign the job of traffic light detection and tracking to the YOLOv8 model and DeepSORT algorithm, respectively. The reason why we want to use the YOLOv8 detection model is because it is the fastest model for object detection in individual camera frames. Additionally, we choose to utilize the DeepSORT algorithm because it is regarded as a highly effective multi-object tracking algorithm that is compatible with YOLOv8. DeepSORT will aid YOLOv8 in traffic light recognition because object tracking will help us avoid cases where the sole use of YOLO would stop detection of traffic lights hidden by surrounding elements or traffic lights that are blinking. YOLOv8 will output bounding boxes locations that DeepSORT will use to keep track of detected traffic lights in all frames until the object moves out of frame. We will encompass our machine learning model and our ROS bag files within two separate ROS nodes. The ROS nodes will communicate with each other in a publisher-subscriber ROS model. The publisher node will be the ROS Bag file node and it will send its camera footage to the machine learning node, which is a subscriber node. The machine learning node will also be a publisher node that sends ROS Bag file results containing frames with bounding box detections to other subscriber nodes.

Overall, our goal is to develop a model that can detect, track, and analyze the state of traffic lights. The output should be produced in the correct format and responsive enough to give the other components of the autonomous driving system time to react to stimuli. Traffic light detection is an essential component of the autonomous vehicle decision making pipeline, and can be considered the entry point. If this implementation functions correctly, then the result is a safer and better-performing self driving vehicle. Another objective for our team is to become better acquainted with machine learning and practical software such as ROS. Also, we hope to become better team members and develop enhanced collaborative skills.

1.2 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.

1.3 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 DeepSort, 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 DeepSort 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.

1.4 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 output to ONNX format as an array of traffic light data.

1.5 Validation and testing procedures

Our project requires various forms of validation and testing to ensure that the system is functioning as intended. For model training, we can utilize YOLO training output statistics, graphs, and charts to get initial comparisons of model performance on testing image sets. For detection inference speed, the program will output in the console the speed of inference for a given frame and allow us to measure model detection speed. On top of these metrics, we should observe performance on various different test scenes in the form of rosbag files to emulate camera feed input.

The tracker is the second component of the system which takes detections and continually updates object IDs, estimating object positioning. It is important to validate and test this portion of the system by ensuring that there are minimal unnecessary object ID switches, position estimates are reasonable, and that it is able to deal with a reasonable amount of occlusion and movement to accommodate for different scenarios.

Our final demo and ultimate validation/test will be to run our program on a provided test video. The program should detect and classify traffic lights accurately, assign IDs to these objects and track them well, and perform any additional light relevance filtering and flashing detection we have implemented. It should also do this with a good deal of speed and output the data in the correct format.

2 Proposed design

2.1 Updates to the proposal design

In lieu of recent findings, we have learned that Ultralytics has a tracker. Ultralytics is responsible for the YOLOv8 image detection we are currently using. Previously, we had decided to use YOLOv8 for detection and DeepSort for tracking, as this combination has been shown to be effective based on our research. However, Ultralytics tracking exists as a simpler alternative, though further queries into its performance are ongoing.

We have also discovered Norfair, which is another tracking library. Its capabilities are similar to that of DeepSORT. Its primary advantage over DeepSORT is that it has been trained to track objects when the camera itself is in motion, and we find this feature particularly useful for our needs in a moving environment. Whereas with DeepSORT, we would have to train a feature extractor on our traffic light objects to yield similar functionalities. In our testing, this tracker has shown promising results when comparing its performance to DeepSORT.

We had previously declared our budget to be $0. That is, we didn’t plan to spend any money, since this project is almost entirely based on open source software. Now that we have had the opportunity to work on the project in more detail, reasons to spend a budget have come to light. Namely, we had considered purchasing an SSD (solid state drive) in order to boot and train our model on a more powerful computer for better performance. However, we have decided against this, as we have found that we are more than capable of training on our own hardware.

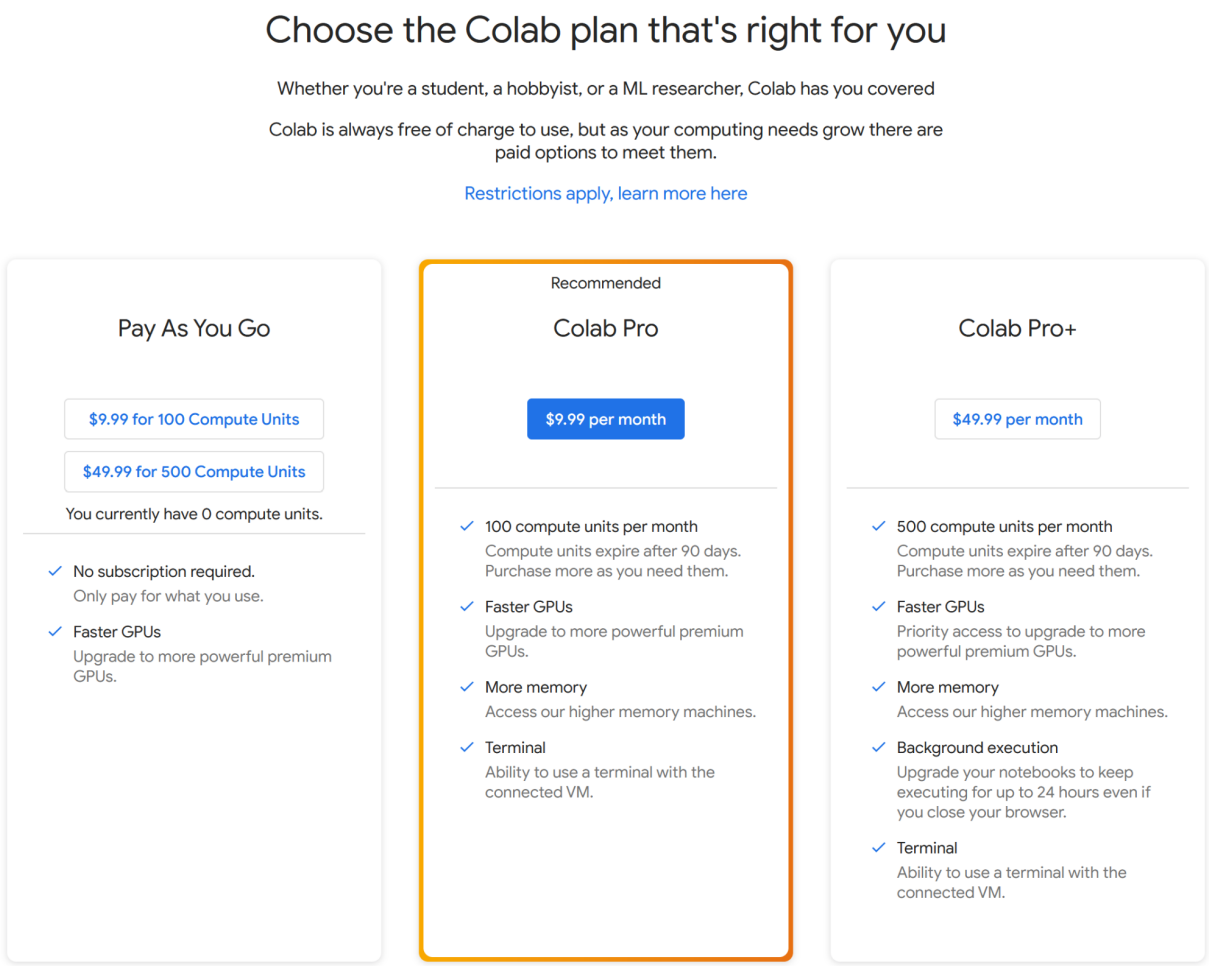


Figure 1: Collab monthly subscriptions offered by google

We have made extensive use of Google's Colab software for testing and training our model. We have not had to spend any money, as Google allows Collab to be used for free. However, this is not without some limitations. When Google's servers are busy, Collab's performance slows to a crawl. Something worth noting is that Google offers paid monthly plans that would allow us to avoid this issue. Figure 1 clearing outlines the prices and benefits of the offered plans. While we have considered allocating a budget toward this, we believe that it is unnecessary for our purposes. We have found that the servers are significantly less busy in the evenings and at night. As such, we now tend to use Collab during those times exclusively.

2.2 System description

The project will consist of a YOLOv8-DeepSORT 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 DeepSORT 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-DeepSORT 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-DeepSORT 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-DeepSORT 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, DeepSORT 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 DeepSORT 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. DeepSORT outputs like object ID, time since the last updating frame, and dimensions of its associated bounding boxes will be the inputs of the layer's logic. Time since the last updating frame is the most important measurement to detect blinking lights because it can indicate the amount of time since a traffic signal was detected. Finally, the project's architecture is depicted in Figure 2.

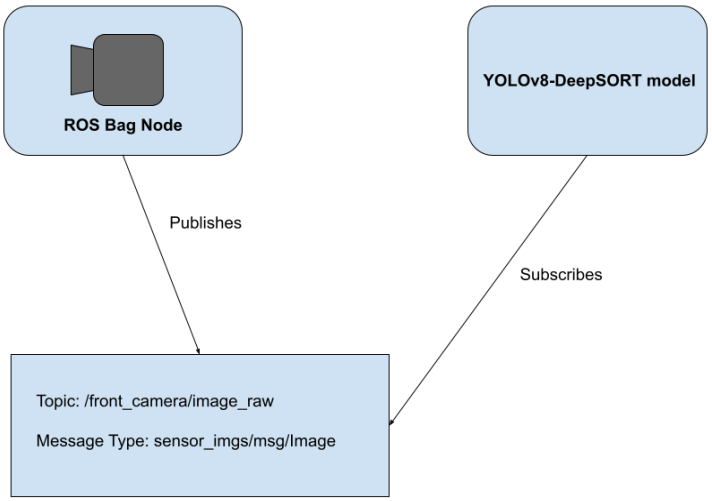


Figure 2: High-level project architecture

2.3 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 figures 3 and 4 below contain information on the current dataset state.

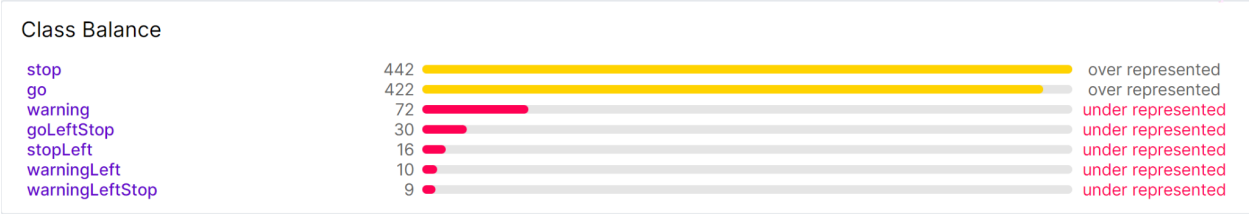


Figure 3: Dataset Class Balance

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 slightly worse after converting from PyTorch to ONNX. The results were still sufficient, but it is noteworthy and can be seen in the preliminary results section below.

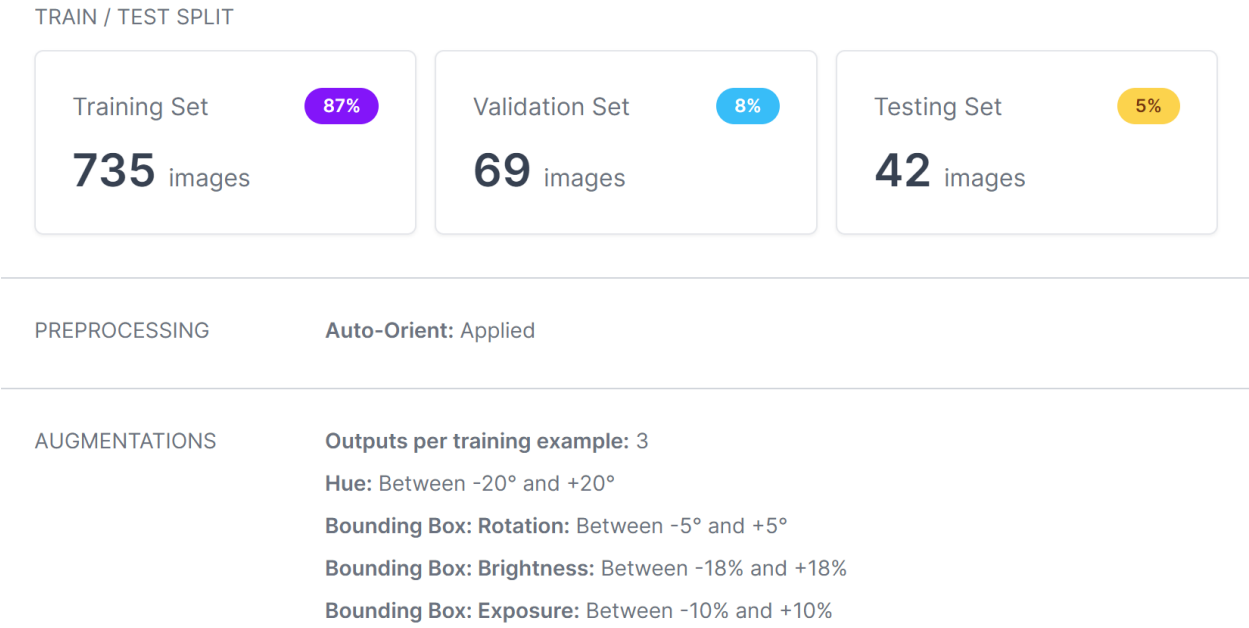


Figure 4: Dataset Overview

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 5 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 $S \times S$ 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.

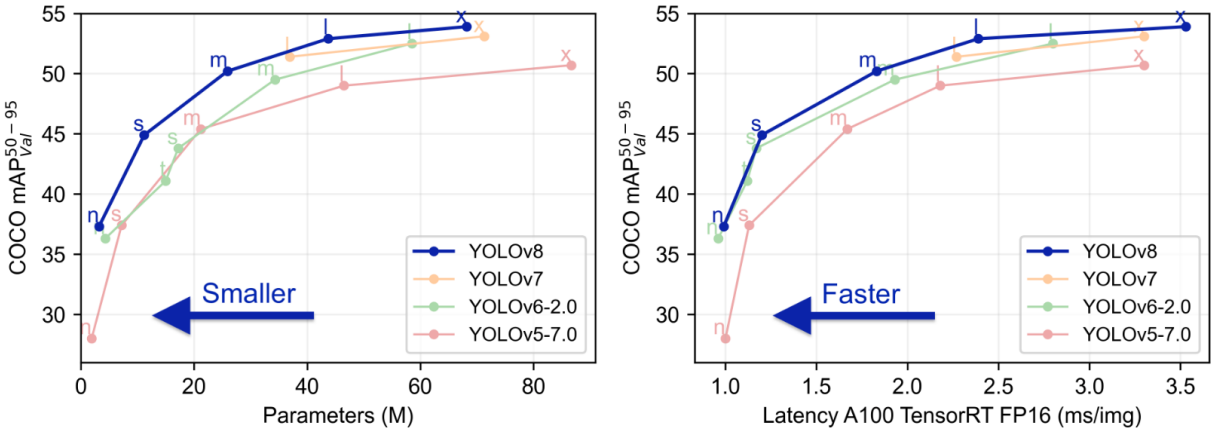


Figure 5: YOLO Performance Comparison

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, one possible solution we have discussed is to group bounding boxes by area and only keep those that are in the group with the largest average area since the closest bounding boxes are most likely to be the most relevant in this case.

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 yellow, as this is commonly found on roadways. We plan on making use of Norfair output information like object ID, the hit_counter attribute, and bounding box dimensions. 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 counting how many times the number has increased or decreased. If the count is greater than 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 6.

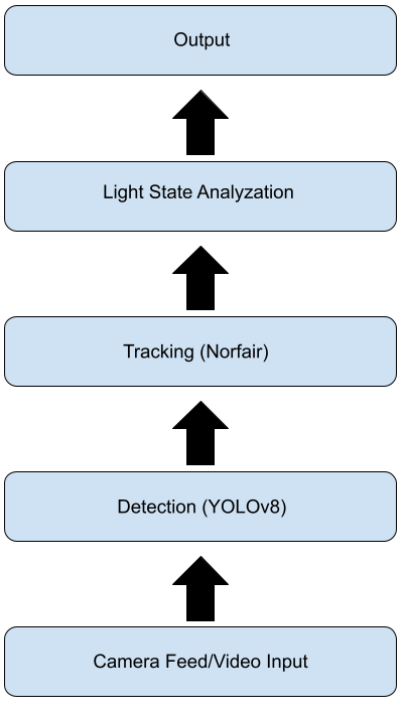


Figure 6: Stages of project's algorithm

A Python program module will utilize the YOLOv8 detection module that is loaded from a Pytorch 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. 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 Project management

Aaryan Shenoy has worked as a software developer intern for the Huntsman Corporation and General Motors, where he has gained experience in project management and system engineering. He has also taken coursework in Distributed Systems and Database Systems. Due to this software engineering experience, he will have the role of system design. Aaryan created the team’s Docker container, added project dependencies with Max, and integrated tracking capabilities with Clayton. His role is to propose ideas and implement strategies related to the architecture design of the ROS and machine learning framework.

Morgan Roberts worked as an intern at Upshur Rural Electric Cooperative and led several collaborative coding projects in the IT department. Morgan aspires to take the leadership and collaborative experience gained from these opportunities and utilize them as a member of the team. As a team leader, Morgan aims to help with all parts of the project.

Xiaohu (Max) Huang worked as an intern at FlightSafety International and has experience with satellite image processing. He also has some knowledge of computer vision and machine learning from his coursework. Due to these experiences, he will have the responsibility of software design. Working jointly with Aaryan, he helped with the project Docker container, loaded project dependencies to the container and debugged problems with the build process.

Clayton Gowan has worked at Capital Technology Group as a software development intern. He also has some basic knowledge of computer vision from working with the TAMU RoboMaster Robotics team. Due to his past software development experience and familiarity with computer vision concepts, he will be assigned the responsibility of software design and model training. Clayton was not originally the one assigned to the model training role, but took an interest in it and decided to take on the role. Up to this point, he has worked on the project dataset, detection model, and tracker.

Robert Madriaga has a relevant academic background with machine learning and artificial intelligence from course experience at Texas A&M University, which they can apply to this project. They also have experience working with robotics from the TAMU RoboMaster team. For these reasons they will be in charge of technical writing and dataset management. Robert has continued to work on the project's technical writing and dataset management, working cooperatively with team members to help with uploading and labeling images of the large dataset. Robert also has begun working on ROS2 integration of the model.

With regard to management techniques, the team has adopted several as time has passed. Initially, we were in agreement to meet in person on Tuesdays and Thursdays in order to hold each other accountable. However, we found that not everyone is necessarily needed at these meetings. While we do mostly hold meetings on these days, we have also added informal meetings in the interim when team members need to collaborate on something specific. We have also split into pairs for pair programming. For example, when we had issues with our Docker container and Github synchronization, Aaryan and Robert helped each other to solve the problem. Finally, we have begun to use Trello as a way to track and communicate the completion of progress. Trello is a workflow management tool that allows teams to visually track progress via “boards” which allow for quick and easy organization of tasks. Overall, these techniques have allowed us to improve efficiency.

3.1 Updated implementation schedule

The gantt chart in figure 7 has been updated to reflect the current adjusted implementation schedule based on the state of development. The model training/data collection process has taken longer than originally expected, but should be near to wrapping up as of now. We shifted the tracking and flashing detection portions back, and are beginning to more seriously tune the tracking module while also looking at the next steps of flashing detection, light relevance filtering, and general output formats to adhere to the expected outputs.

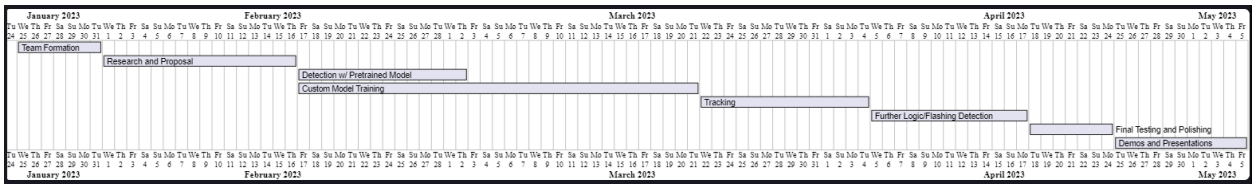


Figure 7: Gantt Chart

The Gantt chart reflects the primary critical path of setting up the ROS node network, getting detection and tracking working, and then adding on other features such as flashing detection followed by testing. Within each of these primary tasks are subtasks such as tuning the tracker or getting the tracker output drawn on the screen for testing. The tasks follow a natural dependency flow in that tracking depends on detection, and further logic and filtering steps depend on tracking.

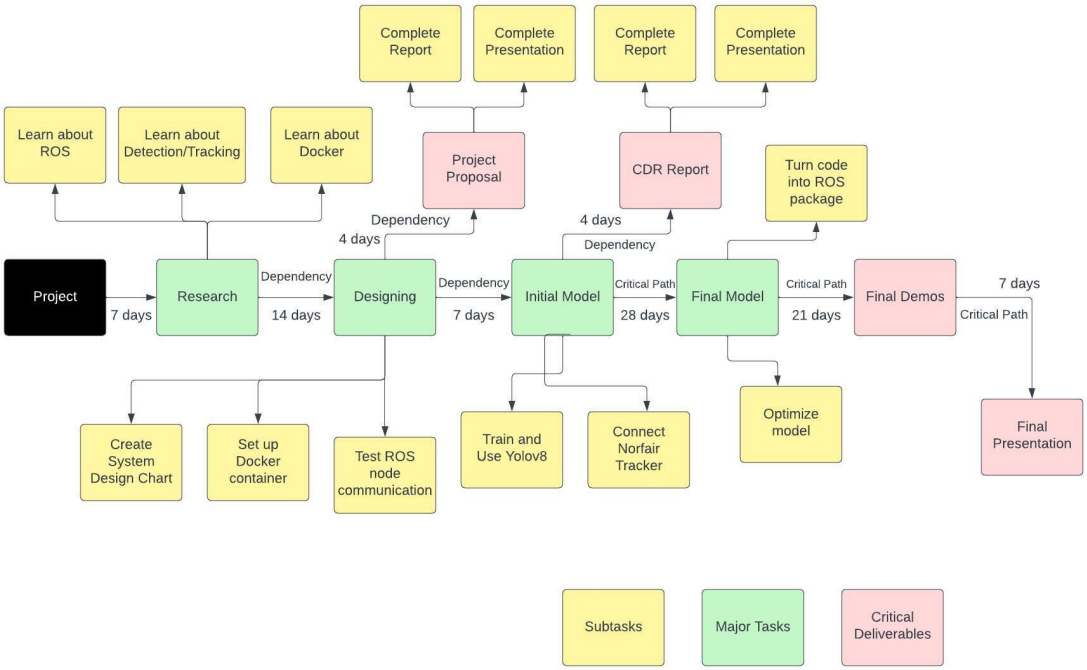


Figure 8: Pert Chart

As shown in Figure 8, the PERT chart for our project provides a breakdown of the project stages into critical deliverables, major tasks, and subtasks. The major tasks are those goals to be completed in order

to submit critical deliverables like the CDR and Final Report. These major tasks serve as dependencies for the critical deliverables. The subtasks identify the steps we planned to take in order to finish the major tasks. The critical path for the project will be the creation of the initial detection model, optimization to turn into a final detection model for traffic lights, final demos to the professors, and submission of final presentation materials. The span of these tasks of 56 days is the minimum amount of time needed to complete the project.

3.2 Updated validation and testing procedures

For our detection model, the primary means of quantitative validation and testing have been the output statistics from training. 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. The figures 9-11 below include statistics and graphs output from training a model.

```
      Epoch    GPU_mem   box_loss   cls_loss   dfl_loss  Instances       Size
      50/50      13.8G     0.5782     0.3309     0.7987         46       1280: 100% 46/46 [01:10<00:00,  1.53s/it]
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 3/3 [00:09<00:00,  3.21s/it]
                   all         69        176      0.873      0.908      0.969      0.756

50 epochs completed in 2.149 hours.
Optimizer stripped from runs/detect/train/weights/last.pt, 22.7MB
Optimizer stripped from runs/detect/train/weights/best.pt, 22.7MB

Validating runs/detect/train/weights/best.pt...
Ultralytics YOLOv8.0.53 🚀 Python-3.9.16 torch-1.13.1+cu116 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 168 layers, 11128293 parameters, 0 gradients, 28.5 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 3/3 [00:08<00:00,  2.91s/it]
                   all         69        176      0.915      0.884      0.965      0.755
                    go         69         75          1      0.775      0.958      0.661
           goLeftStop         69          4      0.914          1      0.995      0.895
                  stop         69         71      0.944      0.718       0.93      0.593
              stopLeft         69          1      0.619          1      0.995      0.895
               warning         69         20      0.996        0.8      0.885       0.73
           warningLeft         69          2      0.933          1      0.995      0.798
      warningLeftStop         69          3          1      0.895      0.995      0.708
Speed: 6.2ms preprocess, 28.2ms inference, 0.0ms loss, 2.0ms postprocess per image
Results saved to runs/detect/train
```

Figure 9: Model Summary Output

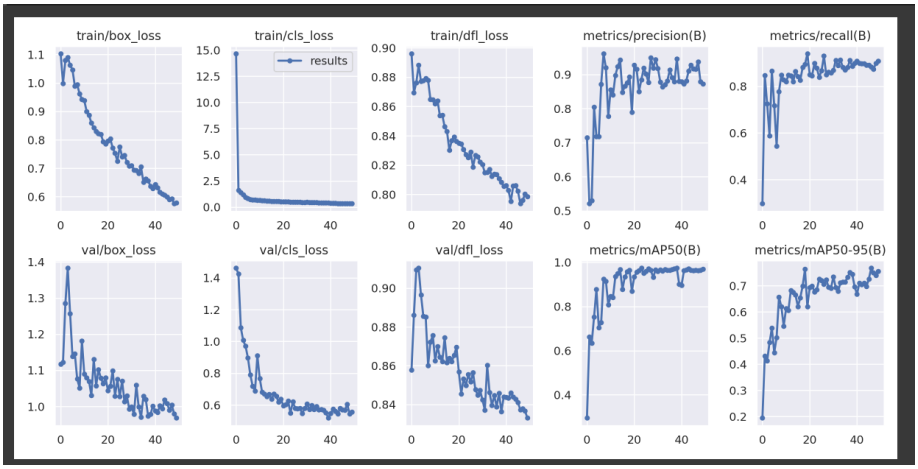


Figure 10: Model Training Graphs

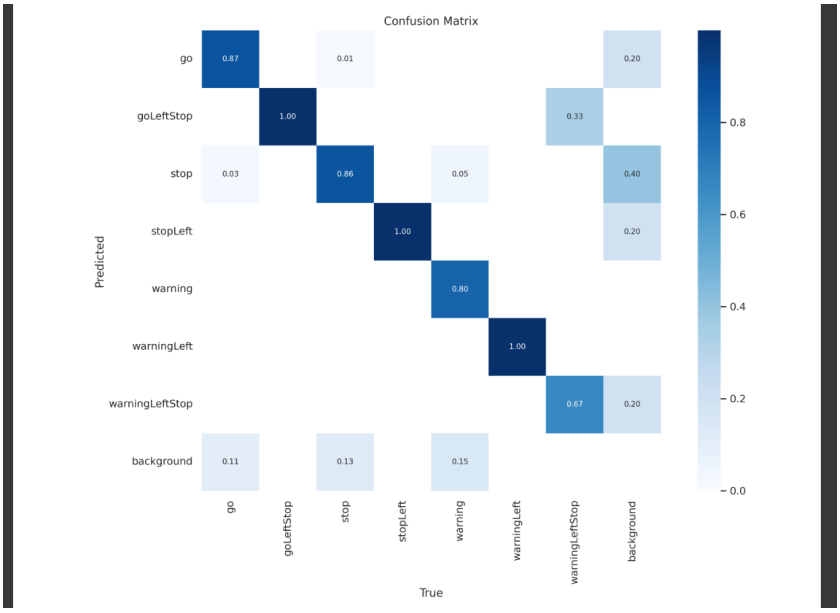


Figure 11: Confusion matrix

For speed, our goal is to achieve inference every one tenth of a second. This is something we can measure from YOLO output, but will be difficult to truly test how it will perform on competition hardware, since we are currently using our laptop CPUS which are likely slower. We can currently record/observe inference speed in milliseconds and hope to get this number as low as possible.

Ultimately, we will test and validate our system by running with input from a rosbag file or video feed of some kind on a test scene. We will seek to ensure that detections are accurate and the tracker handles IDs and predicts objects reasonably. Additionally, any other light relevance filters and flashing/state detections should be tested in bag files as well. The final demo will consist of showing the program running on one of these provided test videos.

3.3 Updated division of labor and responsibilities

The main deliverables for this project are the creation of a fast detection model for traffic lights, modularization of detection and tracking inside the model, and wrapping our project's codebase into a ROS package. At the beginning of the project, the team divided up preliminary work into teams of two, where Max and Aaryan focused on setting up Docker container configuration, while Morgan, Rob, and Clayton trained the YOLOv8 model with a Roboflow image dataset. Both preliminary goals were completed by March 10th.

Following this stage, the team strategized how to connect the detection and tracking capabilities in the project model. We decided to additionally utilize the Norfair tracking library due to favorable results in our demos. Now, the remaining deliverables are the optimization of our model to increase its inference speed and the inclusion of the project implementation into a ROS package. In order to accomplish our goals, we will divide up the team effort with Morgan and Clayton improving model efficiency, Max and Robert finalizing the ROS package, and Aaryan working on both teams due to past contributions on the two goals.

The team's work primarily occurs in face-to-face and online Discord meetings on Tuesdays, Thursdays, and Sundays. The subteams work with each other during the meetings after our standup sessions to finish

the tasks for that day and plan for the next meeting. The subteams aim to accomplish their goals specified on the team's Gantt chart and Trello board. While the Gantt chart serves as a guideline for when the subteam's overarching goal should be completed, the Trello boards are utilized to assign deadlines for individual members on a weekly basis in Tuesday meetings. In Tuesday meetings, the team also rotates which member is responsible for writing the team's weekly report. In order to fulfill the remaining requirements, the team has identified deliverables and due dates for each team member, which is specified down below.

Example of Schedule of Tasks for 03/26/23 - 04/29/2023

03/26/2023 – 04/01/2023

- All - Work on CDR Report
- All - Work on CDR Presentation
- All - meet in person on Tuesday and Thursday
- Morgan - record minutes from meeting
- Morgan - submit weekly report

04/02/2023 – 04/08/2023

- All - meet in person on Tuesday and Thursday
- Clayton - Work on tuning tracker parameters
- Aaryan - Work on tuning tracker parameters
- Max - Work on ROS package, testing
- Robert - Work on implementing ROS messages
- Aaryan - record minutes from meeting
- Morgan - submit weekly report

04/09/2023 – 04/15/2023

- All - meet in person on Tuesday and Thursday
- Clayton - Traffic light relevance filtering
- Aaryan - Flashing traffic light detection
- Max - Work on ROS package, testing
- Robert - Traffic light relevance filtering
- Robert - record minutes from meeting
- Morgan - submit weekly report

04/16/2023 – 04/22/2023

- All - meet in person on Tuesday and Thursday
- Clayton - Traffic light relevance filtering
- Aaryan - Flashing traffic light detection
- Max - Testing
- Robert - Traffic light relevance filtering
- Max - record minutes from meeting
- Morgan - submit weekly report
- All - Perform final testing of components and systems as a whole

04/23/2023 – 04/29/2023

- All - meet in person on Tuesday and Thursday
- All - Perform any final testing and polishing and prepare for final presentations
- Clayton - record minutes from meeting
- Morgan - submit weekly report

4 Preliminary results

Currently we are able to run the detection model and detect seven classes of traffic lights. Running the best19 ONNX model in our github, we are able to run at a somewhat reasonable inference speed on our machines. Current inference speed ranges vary, but on a Surface Book 2 15 inch, the current range is around 250 ms with several other tabs and programs open on the computer. The image is currently being cropped from 2048x2048 to 1024x768. Our goal is to get this inference speed as low as possible. It is difficult to know how our model will perform on competition hardware, as we do not have access to test it at this time.

We also have Norfair tracking running in conjunction with our detection model. The tracking appears to be fast and stable with minimal ID switches, but we would like to continue tuning the tracker with the various parameters available to get the best results. Several figures have been included below to show visualizations of our current program output.

Figure 12: Detection Model Example Scene 1

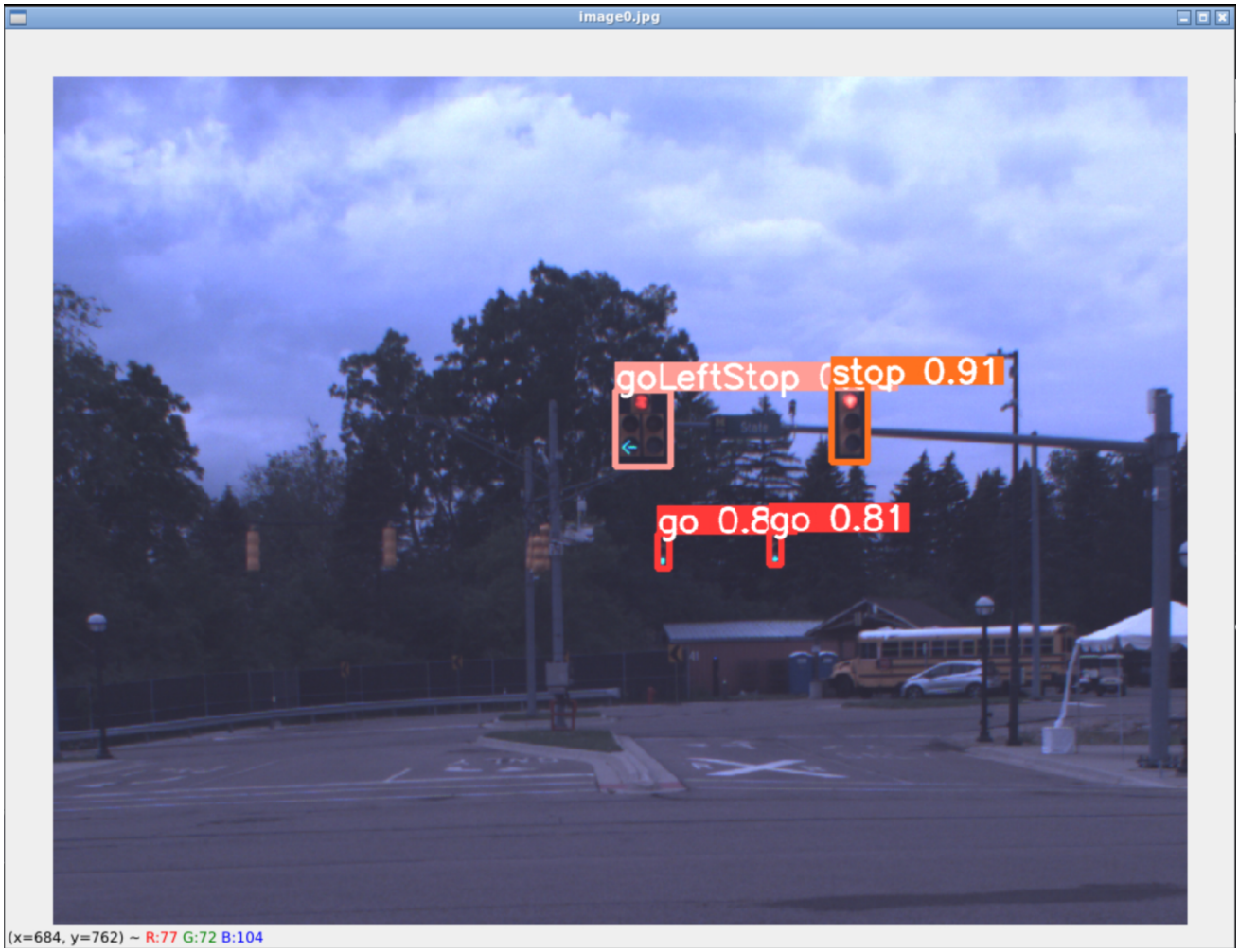


Figure 13: Detection Model Example Scene 2

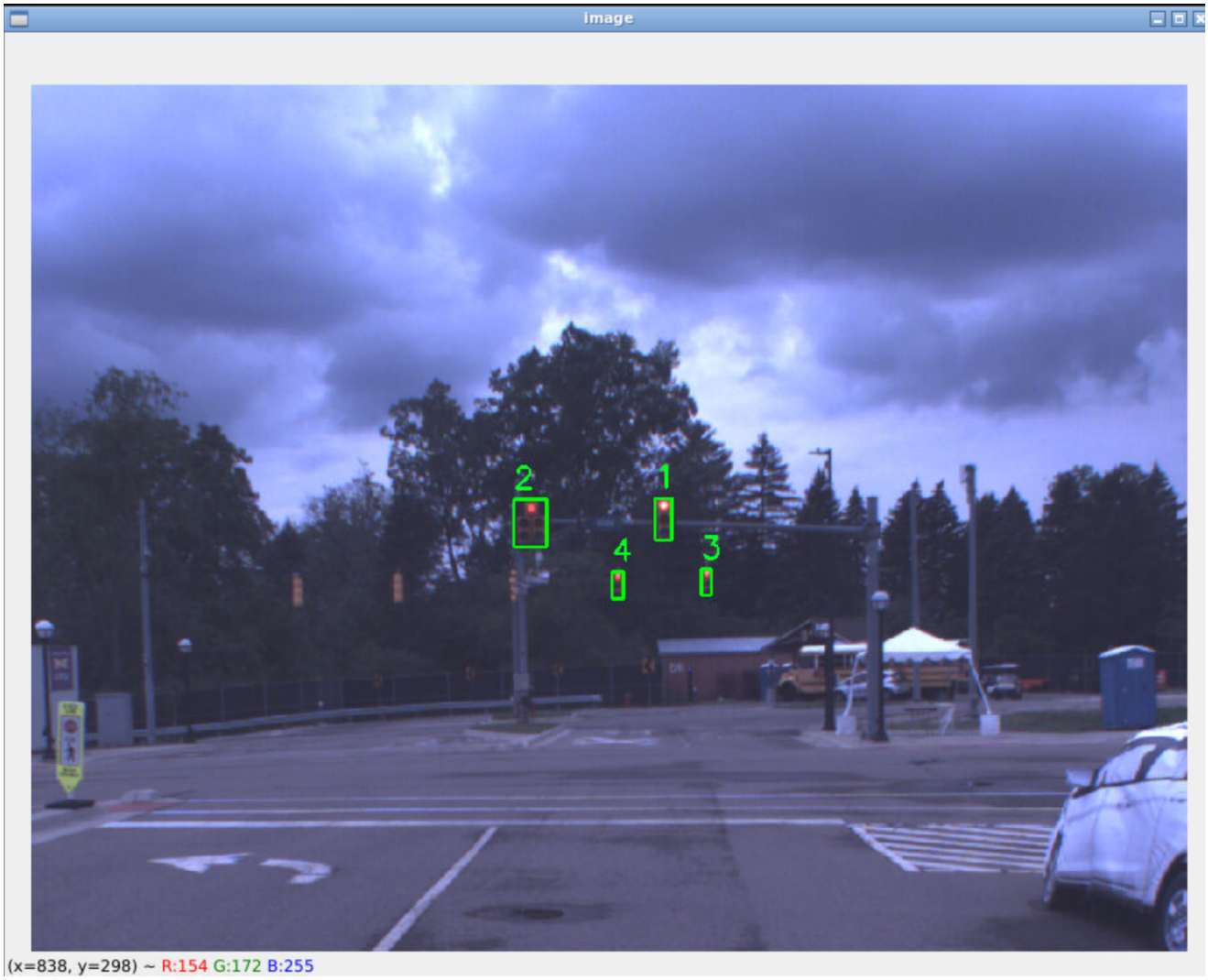


Figure 14: Tracking (Norfair) Example Scene

Figure 15: YOLO Inference Speed