ENPH 353 - Final Report

Adam Crespi, Jordan Werstiuk

December 5, 2024

Contents

1 Introduction

1.1 Background

The ENPH 353 2024 competition, "Fizz Detective," challenged teams to create autonomous robots capable of navigating a simulated world to solve a crime. Teams had to identify clues while following traffic laws, avoiding penalties, and maximizing their score in a four-minute round.

As a team, we had three main goals, listed here in order of importance:

1. Score maximum points and complete the course as quickly as possible.
2. Expand our knowledge and experience in machine learning and simulations.
3. Push the limits of this competition to see what is possible given the freedom allowed this year.

The only requirements for our solution were that we be content with our final creation and that we put in our best effort given the time we have available.

For the competition, we chose to use a drone instead of a ground vehicle. This decision was strategic. A drone allowed us to bypass the pedestrian, the Ford truck, the hills between clues 6 and 7, as well as baby Yoda, which simplifies navigation significantly. We could also simply fly up to clue 8 on top of the mountain, instead of driving the mountain road. The competition rules also gave drones the freedom to fly outside road boundaries, which is a huge advantage. We were also curious about drones, and aimed to expand our knowledge of flight control and aerial robotics.

1.2 Contribution Split

In our team, we divided the responsibilities to ensure maximum efficiency and parallel progress. Adam took charge of the hector-quadrotor implementation, driving, and clue board centering. Jordan focused on implementing the clue board text extraction system and developing and training the CNN.

1.3 Software Architecture

The software architecture for our project is organized into a modular structure to keep things organized, as well as to avoid frequent GitHub conflicts by allowing us to each mainly be working in separate repositories.

We had two repositories, one containing the competition, drone, and user interface, while the other was the controller package.

1.3.1 User Directory and ROS Workspace

At the root of the system lies the user directory, which houses the ROS workspace (`ros_ws`). Inside the workspace, the `src` folder serves as the container for all repositories and source code related to the competition. This is standard for ROS projects.

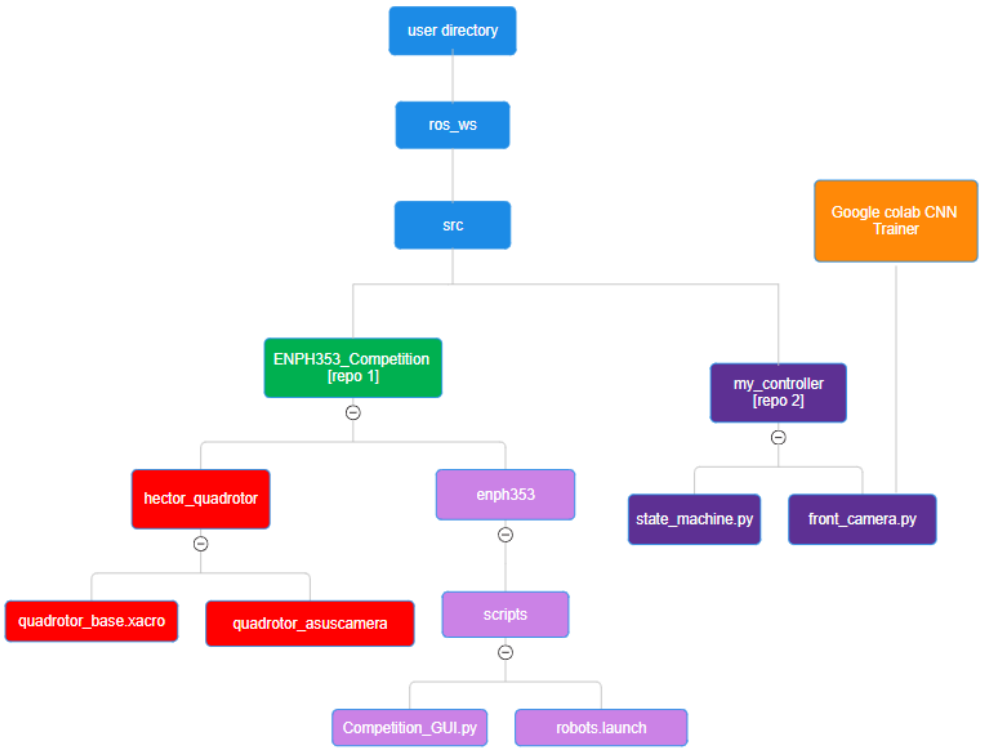

Figure 1: Simplified Software Structure Diagram

1.3.2 ENPH353_Competition Repository

The `ENPH353_Competition` repository contains the essential competition environment and configuration files. Within this repository, two key directories are included:

- **hector_quadrotor**:
  - `quadrotor_base.xacro`: The base URDF file for the quadrotor model.
  - `quadrotor_asuscamera`: Macro for creating the quadrotor's cameras
- **enph353**:
  - `scripts`: Contains tools and utilities, including:
    - `Competition_GUI.py`: A graphical user interface for launching and controlling gazebo and scripts.
    - `robots.launch`: The main launch file for initializing the robot and simulation environment.

The `hector_quadrotor` package is a large set of folders and xacro files that took a while to understand. From this package, we used the `base.xacro`, as well as the ASUS camera. We purposely avoided using the `hector_quadrotor` built-in control files and IMU callbacks, as these relied heavily on global coordinates, ground truth, and data that is illegal for this competition. We will touch more on this in section 3.3.1.

1.3.3 my_controller Repository

The `my_controller` repository contains the custom logic and algorithms developed by our team. It is responsible for controlling the drone and processing data:

- `state_machine.py`: Implements the state machine logic to govern the drone's behavior during the competition. This script includes the state-machine itself, as well as algorithms for centering clue boards and aligning with the white line at board 4
- `front_camera.py`: Handles clue board extraction as well as inference with the CNN.
- `score_tracker.py`: Sends timer start/end messages and clue messages to the score tracker.

1.3.4 Google Colab Image Processor and CNN Trainer

A separate module was implemented in Google Colab. One file was used to develop the image processing method and another to train the CNN. The trained models were then integrated into the `my_controller` repository for real-time deployment during the competition.

2 Discussion

2.1 Driving

2.1.1 Closed Loop Hover-Control

The most challenging part of a drone is the flight control. A large part of this is the hover control, which is a system that balances thrust with gravity to ensure the drone stays at a constant height. For my first implementation of this, I simply constantly push a twist command setting the linear.z acceleration to a constant $+9.81\,\mathrm{m/s^2}$ to balance gravity. See this in Appendix 1.

And although this hover control works fine when the drone is stationary, it completely stops working when you need to go forwards or backwards.

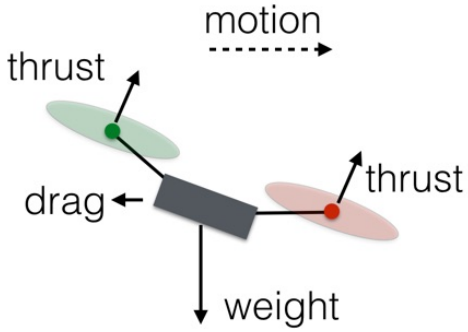

Figure 2: Hover Control

Issues arise because drone propellors only provide a vertical thrust, and thus to move longitudinally, the drone must tilt forward. Without getting too much into flight dynamics, this means the propellors on the front side must slow down relative to rear propellors. The hector-quadrotor package takes care of some of these details, but still we must change our imu-callback hover function. A constant upwards thrust will not allow us to tilt properly for movement.

In our function, we introduce a variable that turns off hover control when trying to move forward or backward. Thrust adjustment for maintaining hover when moving was then added directly into the state machine.

2.1.2 Hard-coded Movement

The idea behind our movement strategy is to move to the general area of the next clue board, before centering our drone on each clue board.

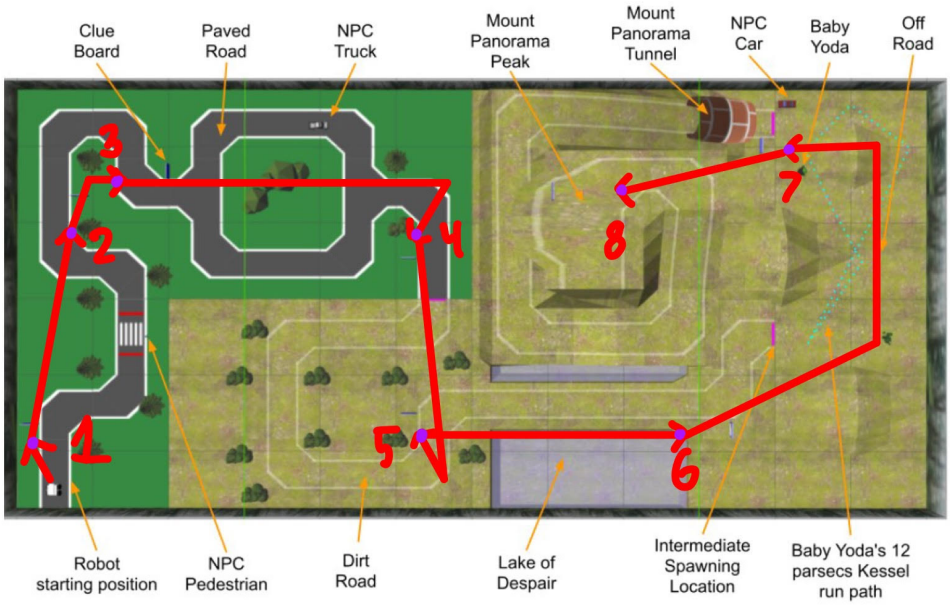

Figure 3: Navigation Route

By distance traveled, this route is much faster than driving along the road and in theory this should have given us a major time advantage. However, the lack of deterministic drone movement meant that following this route was much harder than anticipated.

The state machine was implemented with 6 different states: Navigation, board alignment, hold, align white line, hold white line and next location. The machine rotates through navigating to the next area, using PID to align to board, before holding for a quality CNN image. For sign 4, we also align with a white line to reduce error.

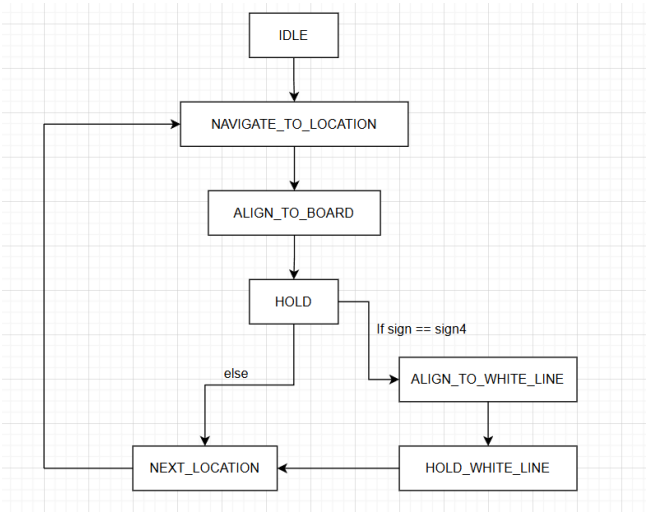

Figure 4: State Machine Diagram

Getting back to the challenges this strategy faced, after each clue board, we accumulated more and more error in our absolute position. This meant that rather than flying to the exact spot we wanted, our drone would end up in a wider and wider randomized area. Say that as an estimation, each clue board get introduced 5% error in our absolute position from drift introduced from our IMU and unpredictable sign PID, by board 4 and 5 we are hitting around 25% error!

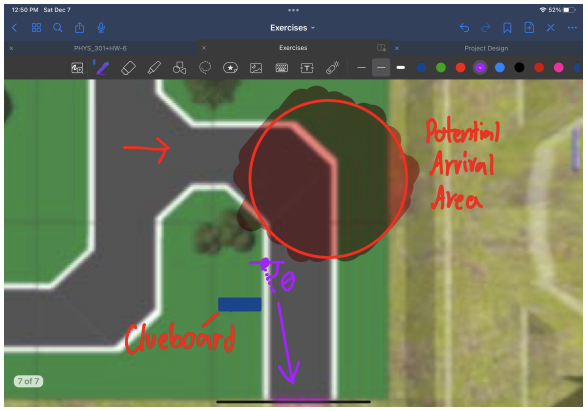

Figure 5: Clue Board 4

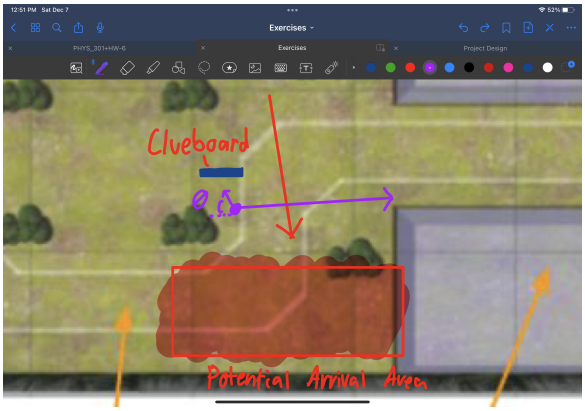

Figure 6: Clue Board 5

The accumulated error is illustrated in figures 3 and 4. Despite 5-10 hours of tuning, I could not reduce the size of these 'error circles'. With tuning, I got an around 75% run success rate, but could not reduce this further. Also note that this error was especially frustrating for boards 4 and 5, as after each board the drone need to fly straight for around 4 meters. This means that a small error angle of $\theta = 5°$, leads to 0.35m of error, and θ was often greater than 5°.

$$4\tan(\theta) = 0.35m, \quad \text{where } \theta = 5°$$

My first solution to reducing drift accumulation involved aligning with the white line at clueboard 4. In theory, aligned with the white road line here would allow us to "reset" our accumulated error. See more on this in section 3.3.2. I struggled in getting the line alignment to work consistently, and it didn't end up helping reduce error too much in the final runs.

2.1.3 Locking in on Clue Boards

The general process for locking in on clue boards is as follows.

1. Apply a blue mask to the image
2. Identify sections of grouped blue pixels of size over set threshold
3. Find center of camera image and center of blue-pixel group
4. Rotate left or right to align central y axis of camera with y coordinate of clueboard center
5. Move forward or backwards to align central x axis of camera with x coordinate of clueboard center

Figure 7: Blue Mask of Clueboard

The center of the clueboard can be seen as green with the image x and y axis as red lines. The purple rectangle is our alignment threshold. As we don't need the sign perfectly aligned, if the green circle is within the threshold, alignment is considered met.

Initially, our drone was flying significantly higher and using a separate camera with a much narrower FOV as a "zoom lens" to get quality sign pictures. We would use our wide FOV camera to center the clueboard before using our zoom camera to get a quality picture. However closed to competition, we changed to flying lower and thus a single camera was used.

2.2 Clue Plate Recognition

2.2.1 Recognizing a Clue Board

While initially the area of the largest blue contour was compared to a threshold value to determine if a clue board was visible enough, the final version of the design implemented a more effective strategy. From Section 2.1.3 Locking in on Clue Boards, the code knows exactly when it has finished locking in a clue board, which is when the drone has the best view of it. It is precisely at this moment that the image processing should begin. A new ROS topic delivers this information in the form of an integer corresponding to the sign number from the drone control code to the image processing code. Only for the immediate next image received from the camera by `front_camera.py` will processing begin, then a flag will be set to stop processing of images until a new integer is sent across the ROS topic.

2.2.2 Extracting Separate Words

To extract only the relevant data from the image, a binary mask is applied to specifically isolate the blue border and words on the clue boards. The largest contour is then identified and used to crop the image to the size of the clue board.

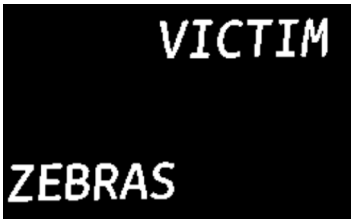

Figure 8: Cropped Binary Mask

Individual words are extracted by analyzing the variance within the masked region. A sliding window method is used to compute the variance map of the image, which highlights high-contrast areas corresponding to word regions. The variance map is thresholded to identify potential word boundaries, and connected components are extracted based on their size and shape.

The system then applies additional size constraints on the connected components and sorts the detected words by their vertical and horizontal positions to maintain the correct order.

Figure 9: Ordered Extracted Words

2.2.3 Extracting Separate Letters

Once words are isolated, the system processes each word region to extract individual letters. Contours are identified within each word, sorted left-to-right, and filtered based on height and

width thresholds to ensure only valid letter regions are retained.

The extracted letter regions are resized to a fixed dimension, padded with a consistent border, and cleaned using connected component analysis to remove noise. At this point there was never any noise left, but it didn't hurt. Each letter is subsequently passed to the classification models for prediction, and sent to the score tracker with the help of `score_tracker.py`.

Figure 10: Ordered Extracted Letter

2.3 Convolutional Neural Network

2.3.1 Model Architecture

The neural network used for character recognition is a Convolutional Neural Network (CNN) with the following architecture:

- **Input Layer**: 50×44 grayscale images with a single channel.
- **Convolutional Layer 1**: 32 filters of size 3×3, activation function ReLU.
- **Max-Pooling Layer 1**: Pool size 2×2, reducing spatial dimensions.
- **Convolutional Layer 2**: 64 filters of size 3×3, activation function ReLU.
- **Max-Pooling Layer 2**: Pool size 2×2, once again reducing spatial dimensions.
- **Flatten Layer**: Converts feature maps into a 1D vector.
- **Dense Layer**: 128 neurons with ReLU activation and a dropout rate of 0.5 to reduce overfitting.
- **Output Layer**: 36 neurons (one for each class), activation function Softmax for classification.

The architecture was chosen to be very similar to what was showed as an example in lecture for character recognition. The main difference is the use of two convolutional layers instead of three, chosen because of the smaller dimensions of the input images.

2.3.2 Preparing Data

The data used by the models that were ultimately deployed in the final version was manually collected. This was done by using the teleop keyboard to fly the drone around the environment, while downloading images from the rqt image view. The drone was positioned to be in locations slightly offset from the signs to incorporate variability into the data. Around 20 images were

collected. To ensure a (mostly) flat distribution of letters/numbers, the clues were modified manually instead of loading them in from the external URL.

The individual letter images were then extracted from the collected images using OpenCV in the image processing Colab file (outlined in Section 2.2.2 and 2.2.3).

The letter images were then randomly augmented:

- 25% chance of Gaussian blurring with kernel size 3 or 5
- 25% chance of rotation between -25 and 25 degrees (minimum of 6 degrees)
- 25% chance of zooming in or out between 0 and 20
- 25% chance of shifting 3-6 pixels in any direction

Next, the images were manually labeled with a letter or a number.

Figure 11: Augmented Image Ex 1

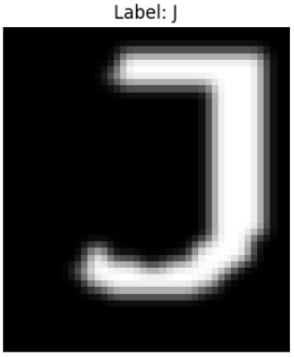

Figure 12: Augmented Image Ex 2

The images were also normalized to the range of [0,1] by dividing by 255.

2.3.3 Training Parameters

The model was trained using the Adam optimizer, as it was found to learn much faster without creating any worse of a model compared to categorical cross-entropy.

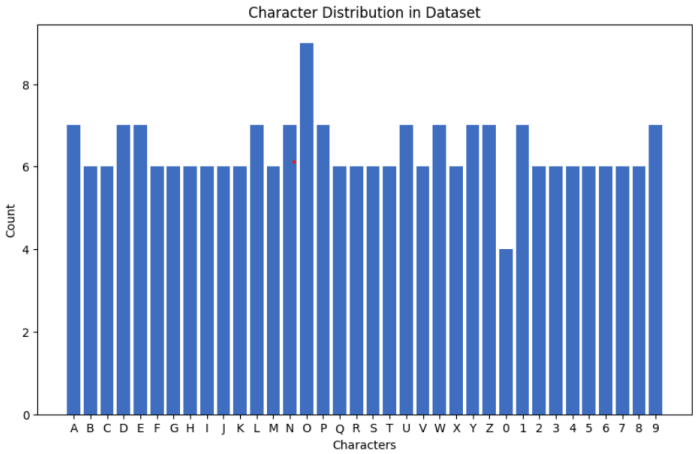

Figure 13: Training Data Histogram

The following parameters were used:

- **Learning Rate**: Adaptive (default for Adam).
- **Batch Size**: 6 samples per update.
- **Epochs**: 70 epochs with early stopping to avoid overfitting.
- **Validation Split**: 20% of the training data used for validation.

2.3.4 Testing Performance

While the model only performed with 80% accuracy on the validation data, since this data was augmented, it wasn't reflective of the models performance on what it would see in the actual competition.

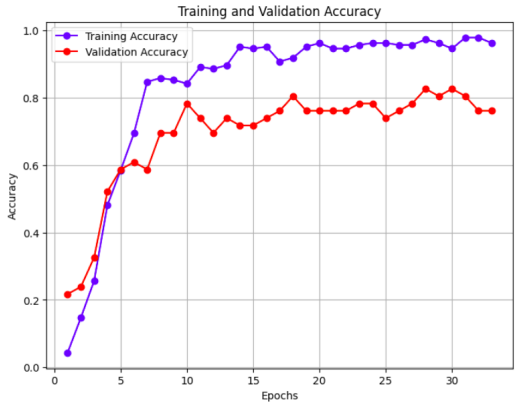

Figure 14: Accuracies vs. Epoch

The confusion matrix gave important insight into the performance of the model and the letters which it struggled with most (see the next section).

2.3.5 Combining Models

A second model was created exactly like the first, except using the unaugmented version of the same training data. Both models made mistakes on a small amount of letters, however they made mistakes on different letters. Therefore, using the prediction of the more confident model for a given input image can reduce the number of overall errors seen.

Additional error analysis was performed once everything was fully implemented and running in ROS, simply by analyzing the output in the score tracker. Certain letters were still predicted incorrectly occasionally, however there was not enough time to finish implementing the improvements that we began on before the competition.

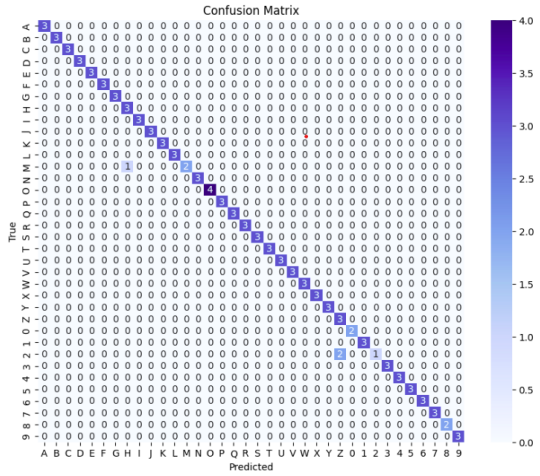

Figure 15: Confusion Matrix of Augmented Model

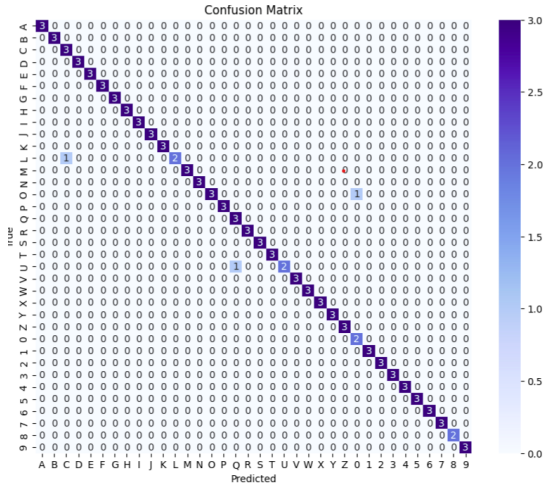

Figure 16: Confusion Matrix of Unaugmented Model

2.4 ROS Publishers and Subscribers

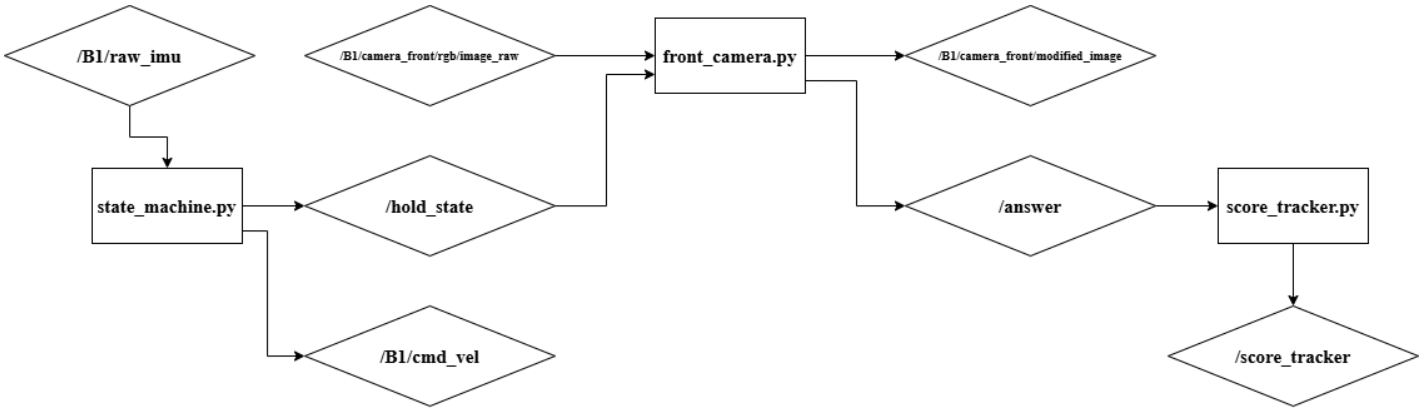

Figure 17: ROS Flow Diagram

The diagram above shows the interactions between the three relevant files of our controller package.

The `/B1/camera_front/modified_image` topic was mainly used for debugging and displaying within the user interface.

The `/B1/raw_imu` topic was part of the hector-quadrotor package.

3 Conclusion

3.1 Competition Performance

The competition performance didn't quite go as well as planned. Despite an around 75% success rate in practice runs, for our only run in competition the movement failed. After the first clue board the drone immediately went way off course, which was an issue that had never happened before. Despite the failure, in subsequent runs our CNN worked almost perfectly, as expected.

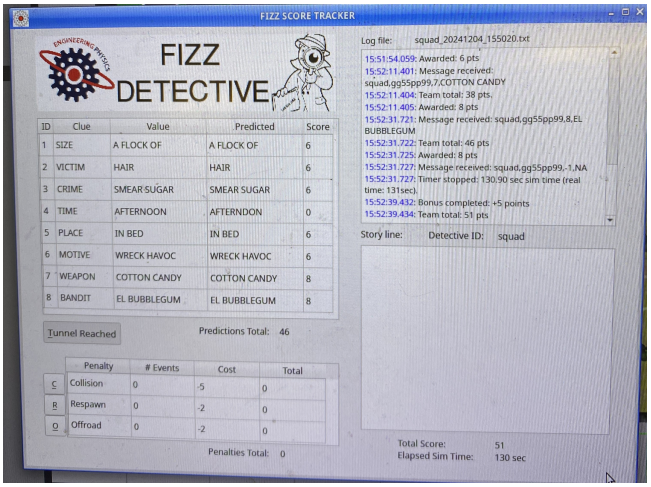

Figure 18: Scoreboard for the successful run we submitted as video

3.2 Other Attempted Methods

3.2.1 Clipping Manipulation

When playing around with camera settings in ROS. We discovered a setting called camera "clipping" which allows you to set the closest distance at which objects are visible to the camera. This could have been useful for our drone, as cameras with different clipping settings could allow us to dynamically adjust our render-able range, and essentially see through trees and obstacles. However this is also overpowered and after talking to Miti, he amended the rules to ban it.

3.2.2 Downwards Facing Camera

Originally for navigation, we planned to follow the road lines with a downwards facing camera. With this camera setup, we had a PID algorithm implemented that navigated the course with the drone centered on the road. This worked decently and we were able to navigate to clue board 4. However the PID movement was very slow and we wanted to take full advantage of having a drone, which lead us to transition into hard-coding.

3.3 What We Would Have Done Differently

3.3.1 Dead Reckoning and Determinism

The issues we faced with drone control and unpredictable movement stems from the fact dead reckoning is unpredictable. Using the drones build in IMU presents a big challenge, as over time drift and error accumulates heavily. We spent a large number of hours tuning and it was nearly impossible to get consistent runs. Subscribing to the built in Gazebo ground truth drone IMU would have made our lives way easier and solved these issues, but it is not allowed in ENPH 353.

In retrospect, either avoiding a drone entirely or using many waypoints would have been a better approach. Rylan and Cohen were a great example of how we could have done this. They faced similar challenges to us in drone control, and instead of spending many hours to try and tune perfectly, they added key points at each sign that allowed the drone to correct its positioning. This slowed them down a lot, but allowed them to achieve consistent runs.

3.3.2 Simpler can be better

As much fun as it was using a drone for this competition, it was not a great decision in retrospect. It took a crazy amount of time to integrate the hector-quadrotor into our workspace properly. Although I definitely learned a lot about ROS, we could have just been driving with the car immediately. From a project management perspective, spending 10 hours perfectly tuning PID for the car would have been a much better use of our time.

3.3.3 Machine Learning Driving

Looking back on this course, it would have been a better learning experience had we chose to use machine learning for vehicle navigation. Although the state machine and drone navigation was interesting, in labs prior we learned about reinforcement learning and imitation learning. Applying these techniques would have maybe been more successful and would definitely have been better for learning.

3.3.4 Faster Image Processing

While it wasn't an issue, the clue board pre-processing (ie. extracting the letter images) took multiple seconds. Ideally this would have been faster, and if we were to redo this part of the code, we would determine which parts take the most amount of time (we believe it is finding the variance), and determine alternative methods of accomplishing the same thing. There were most definitely parts of the image processing that were redundant or even unnecessary, however we kept those parts because we did not want to change something that was already working, given the limited time we had.

3.3.5 CNN Accuracy

Having a CNN that made no mistakes would eliminate the need to get two predictions for each letter, increasing the speed, and also increasing the accuracy. We were not sure how hard it would be to improve our models accuracy from the high 90% range to 100%, which is why we compared confidences of two models. Given more time, we would spend it experimenting with the training data and parameters, as well as the model architecture, to see if 100% accuracy could be reached.

4 References

4.1 Discussions with Chase and Josh

Throughout working on the final competition, we had conversations with Chase and Josh regarding our drone movement. These discussions led to a few insights that changed our movement techniques.

- Flying directly between clue boards: Before this, we were trying to use a downwards facing camera for road following
- Reducing Flying Height: Originally we were flying 2-3 meters above the ground and using a zoom camera to zoom in for clue board reading, but by reducing flying height we were able to get more consistently high quality clue board pictures for our CNN.
- Comparing model confidences: Beforehand, we had one model that gave our predictions, which we changed to be two models, and the prediction with the higher confidence was taken.

4.2 hector-quadrotor

Major thank you to Aarón Juárez who wrote the GitHub repository hector-quadrotor ported to ROS-Noetic with Gazebo 11. This made our lives much easier for the drone xacro creation and camera creation. See the full repo here. Also special thanks to Cohen and Rylan who told us about this package in the first place.

5 Appendices

5.1 Appendix 1 : Original Hover Control imu-callback Function

```
def imu_callback(self, msg):
    """Adjust hover thrust based on IMU feedback."""
    global THRUST, HOVER_CONTROL_ENABLED
    self.latest_accel_z = msg.linear_acceleration.z

    if HOVER_CONTROL_ENABLED:
        error = TARGET_ACCEL_Z - self.latest_accel_z
        THRUST += GAIN * error
        THRUST = max(0, min(THRUST, 1))  # Clamp thrust
```

5.2 Appendix 2 : Full State Machine

```
     def run(self):
        """Main state machine loop."""
        rate = rospy.Rate(10)  # 10 Hz

        while not rospy.is_shutdown():
            if self.state == "IDLE":
                rospy.loginfo("State: IDLE")
                if self.current_location_index < len(self.locations):
                    self.state = "NAVIGATE_TO_LOCATION"

            elif self.state == "NAVIGATE_TO_LOCATION":
                rospy.loginfo("State: NAVIGATE_TO_LOCATION")
                location = self.locations[self.current_location_index]
                self.navigate_to_location(location)

                if location == "sign4":
                    rospy.loginfo("Completed navigation to sign4. Aligning to blue board.")
                    self.state = "ALIGN_TO_BOARD"
                else:
                    self.state = "ALIGN_TO_BOARD"

            elif self.state == "ALIGN_TO_BOARD":
                rospy.loginfo("State: ALIGN_TO_BOARD")
                if self.align_to_blue():
                    if self.locations[self.current_location_index] == "sign4":
                        rospy.loginfo("Fully aligned to sign4. Moving to HOLD state.")
                        self.state = "HOLD_WHITE_LINE"  # Transition to holding for white line alignment after sign4
                        current_sign_number = self.current_location_index + 1  # Convert to 1-based index
                        self.hold_state_pub.publish(current_sign_number)

                        rospy.loginfo("Published True to /hold_state. Entering hold duration.")

                        rospy.sleep(SLEEPTIME)  # Wait for defined sleep time after aligning to other locations

                        self.hold_state_pub.publish(0)
                        rospy.loginfo("Published False to /hold_state. Hold complete. Moving to NEXT_LOCATION.")
                    else:
                        self.state = "HOLD"

            elif self.state == "HOLD_WHITE_LINE":
                rospy.loginfo("State: HOLD_WHITE_LINE")
                rospy.sleep(SLEEPTIME)  # Wait for defined sleep time after aligning to sign4
                rospy.loginfo("Hold after sign4 complete. Aligning to white line.")
                self.state = "ALIGN_TO_WHITE_LINE"

            elif self.state == "ALIGN_TO_WHITE_LINE":
                rospy.loginfo("State: ALIGN_TO_WHITE_LINE")
                if self.align_to_white_line():
                    rospy.loginfo("White line alignment complete. Moving to NEXT_LOCATION.")
                    self.state = "NEXT_LOCATION"

            elif self.state == "HOLD":
```

```
        rospy.loginfo("State: HOLD")

        rospy.sleep(SLEEPTIME - 1.5)

        current_sign_number = self.current_location_index + 1  # Convert to 1-based index
        self.hold_state_pub.publish(current_sign_number)

        rospy.loginfo("Published True to /hold_state. Entering hold duration.")

        rospy.sleep(SLEEPTIME)  # Wait for defined sleep time after aligning to other locations

        self.hold_state_pub.publish(0)
        rospy.loginfo("Published False to /hold_state. Hold complete. Moving to NEXT_LOCATION.")

        rospy.loginfo("Hold complete. Moving to NEXT_LOCATION.")
        self.state = "NEXT_LOCATION"

    elif self.state == "NEXT_LOCATION":
        rospy.loginfo("State: NEXT_LOCATION")
        self.current_location_index += 1
        if self.current_location_index >= len(self.locations):
            rospy.loginfo("All locations visited. Returning to IDLE.")
            self.state = "IDLE"
        else:
            self.state = "NAVIGATE_TO_LOCATION"

    rate.sleep()
```

5.3 Appendix 3 : Select ChatGPT Prompts

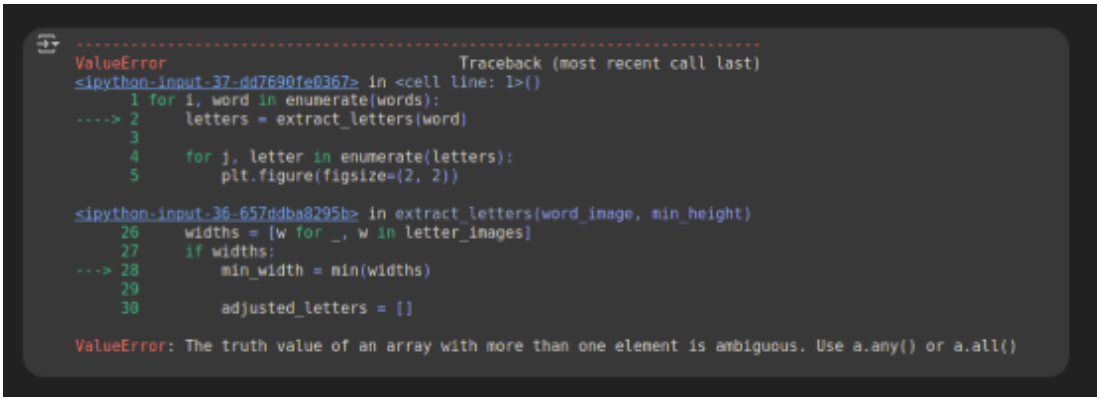

Figure 19: ChatGPT prompt displaying an error

ChatGPT was very helpful for diagnosing errors. With this, we were able to spend more time working on the important parts of the competition.

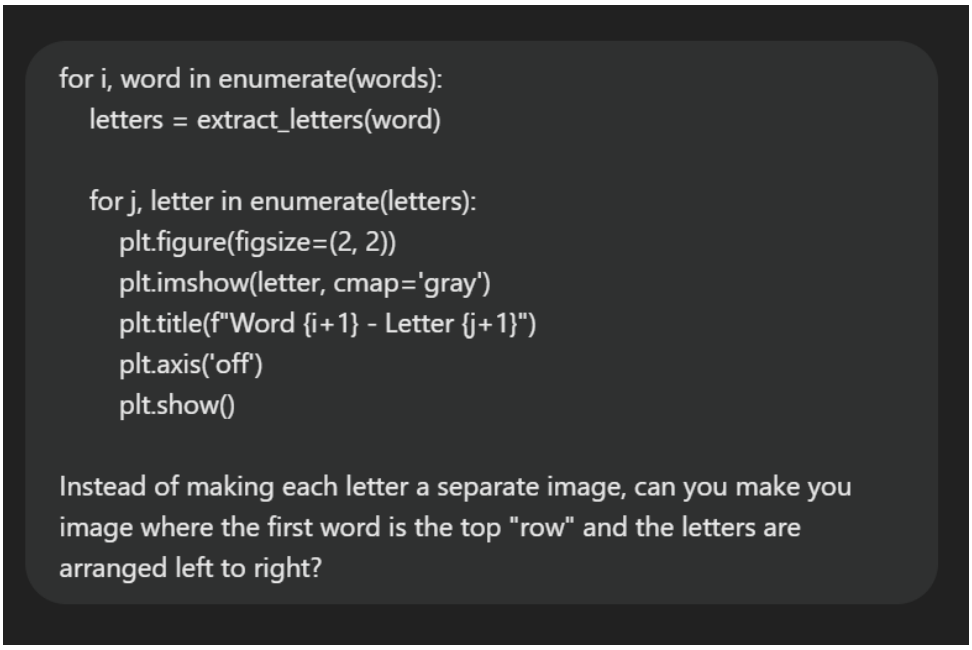

Figure 20: ChatGPT prompt outlining a change to a code section

ChatGPT also helped with making certain changes to code (or write code). This was used occasionally in specific situations, such as when we were short on time, or when we were lost as to how to implement something.

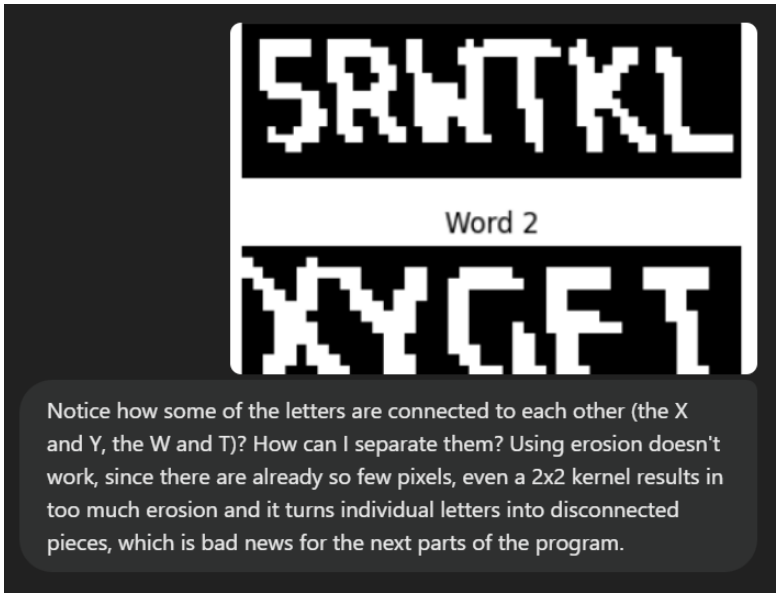

Figure 21: ChatGPT prompt asking for ideas

Lastly, ChatGPT helped sometimes with advice or ideas on things to try.