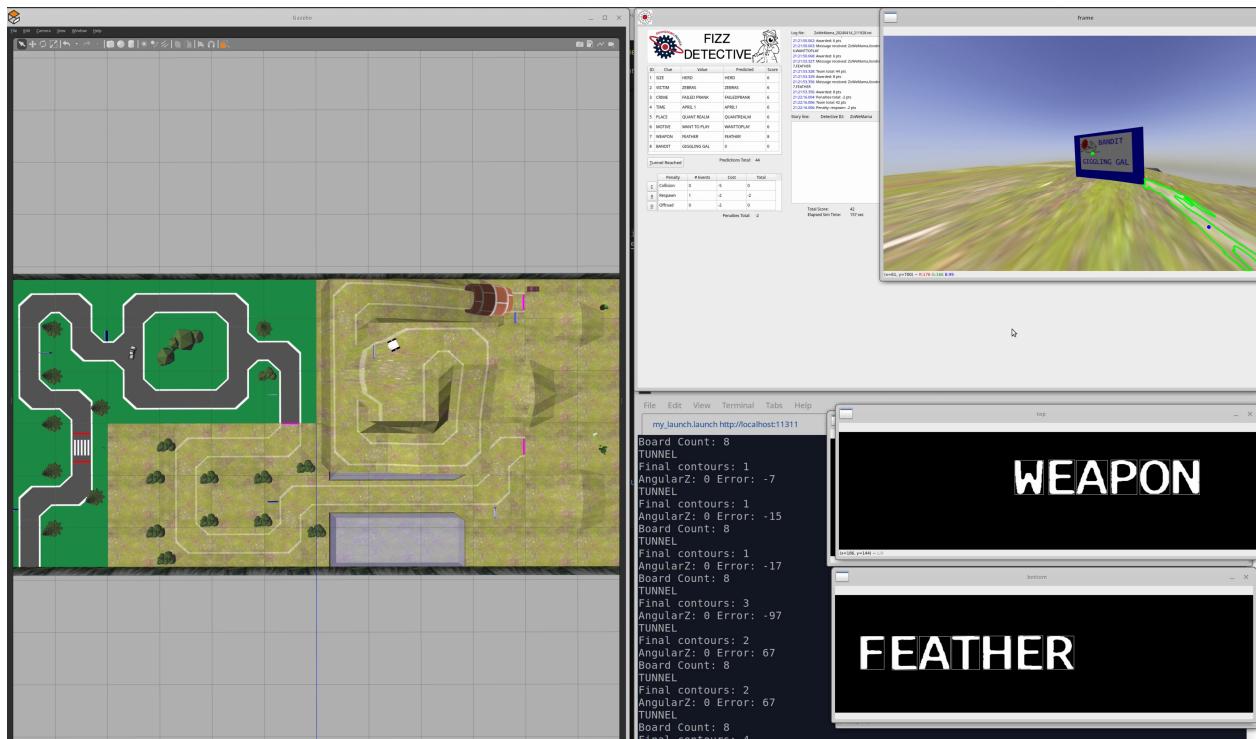


ENPH 353 Final Report

Andrew Du, Lisa Shi

Team 6

Submitted



<https://github.com/AndrewD0/Zoo-Wee-Mama.git>

Table of Contents

1	Introduction	1
1.1	Competition Overview	1
1.1.1	Competition Rules	2
1.1.2	Contribution Split	2
1.2	System Architecture Overview	3
2	Driving Algorithm	4
2.1	PID Driving	4
2.1.1	Road	4
2.1.2	Grass	4
2.1.3	P-Controller	5
2.1.4	Mountain	5
2.2	Roundabout and Truck	6
3	Clue Detection	6
3.1	Character Recognition Neural Network	6
3.1.1	Training Data	7
3.1.2	Training Result	7
3.2	Clue Board Detection	8
4	Barrier Detection	9
4.1	Pedestrian	9
4.2	Baby Yoda and Tunnel Reaching	9
5	Conclusion	10
5.1	Future Improvements	10
Appendix		11
A	Neural Network Model Summary	11
B	Character Generation	11
C	State Tracker	12
C.1	Clue Counter	12
C.2	Pink and Red Line Crossing	12
C.3	Roundabout	13
D	Use of ChatGPT	13

1 Introduction

1.1 Competition Overview

The ENPH 353 Competition is conducted within a simulated environment comprising three primary areas: road, grass, and mountain, along with various obstacles such as pedestrians, trucks, roundabouts, Baby Yoda, and a tunnel.

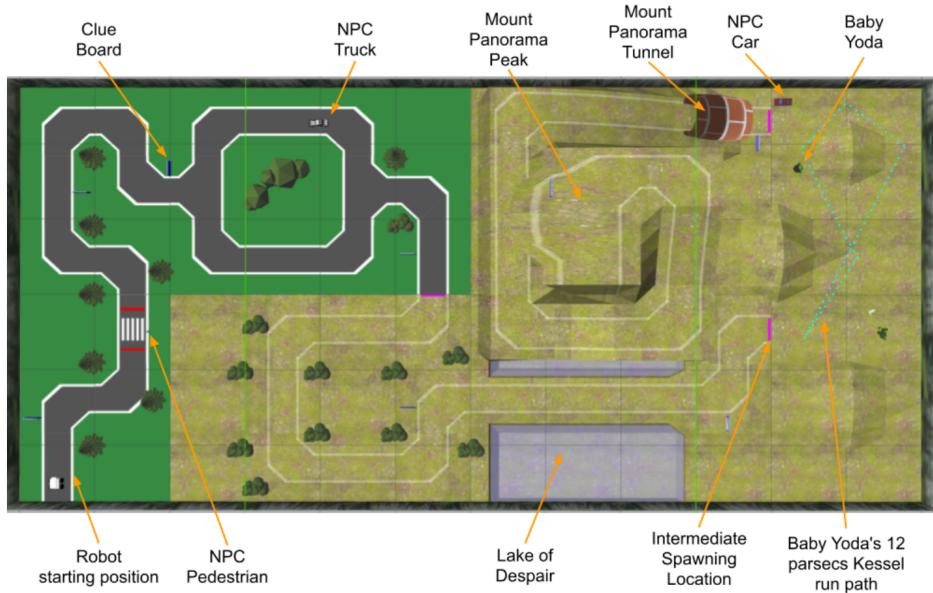


Figure 1: Screenshot of the competition surface.

The competition features eight clue boards positioned at different locations across the environment. The objective is to develop an autonomous robot capable of navigating through the terrain, reading the clues presented on the boards, and solving a detective story. This is achieved using the Robot Operating System (ROS) in conjunction with the OpenCV library.

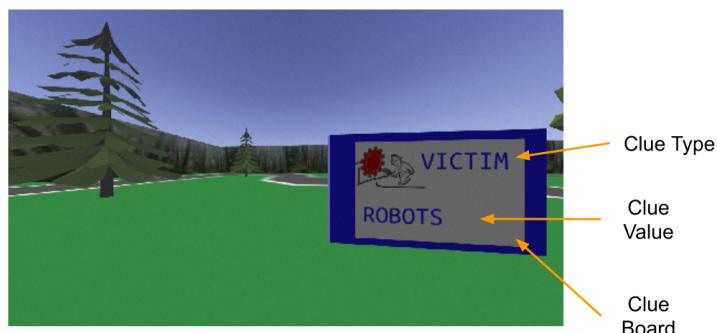


Figure 2: Robot camera view of a clue board.

1.1.1 Competition Rules

Each team is allotted one four-minute round of Gazebo simulation time. The scoring is based on correctly predicting clue values and adhering to traffic rules, with penalties for violations.

Teams earn points by submitting correctly predicted clue values to the score tracker board. The first six clues are worth 6 points each, while the last two are worth 8 points each. Penalties include 2 points for teleporting at pink lines or having more than two wheels off-road, and 5 points for collisions with trucks, pedestrians, or Baby Yoda.

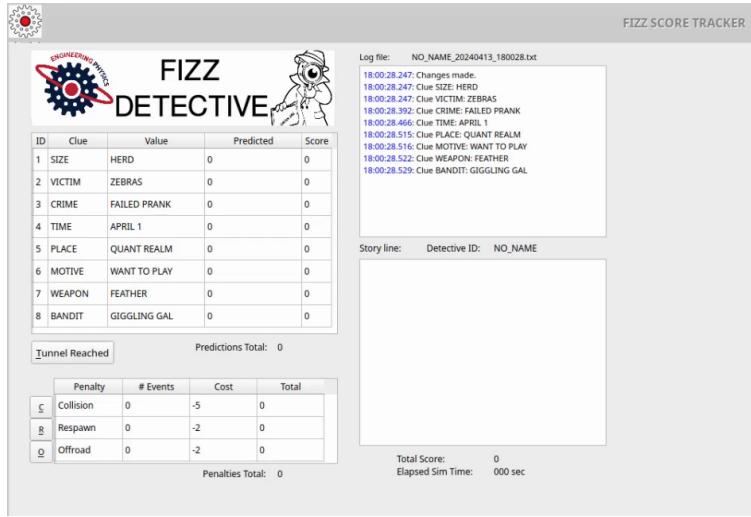


Figure 3: Screenshot of the score tracker board.

Robots are limited to subscribing to four topics: camera, clock, velocity, and model state. However, they can follow additional topics of their own design. A 5-point bonus is awarded if a robot reaches the tunnel without respawning. Pre-trained neural networks are not allowed, except for You Only Look Once (YOLO).

Teams must send messages to the score tracker board to start and stop the timer. In case of tied scores, the team with the shortest lap time will rank higher.

1.1.2 Contribution Split

Andrew worked on implementing the PID driving algorithm, pedestrian detection, and the implementation of a neural network. Lisa focused on character recognition, which includes implementing clue board detection methods, character trimming techniques, and neural network integration with the overall robot ROS system. Both Andrew and Lisa worked on methods to navigate the roundabout, as well as generating data for neural network training.

1.2 System Architecture Overview

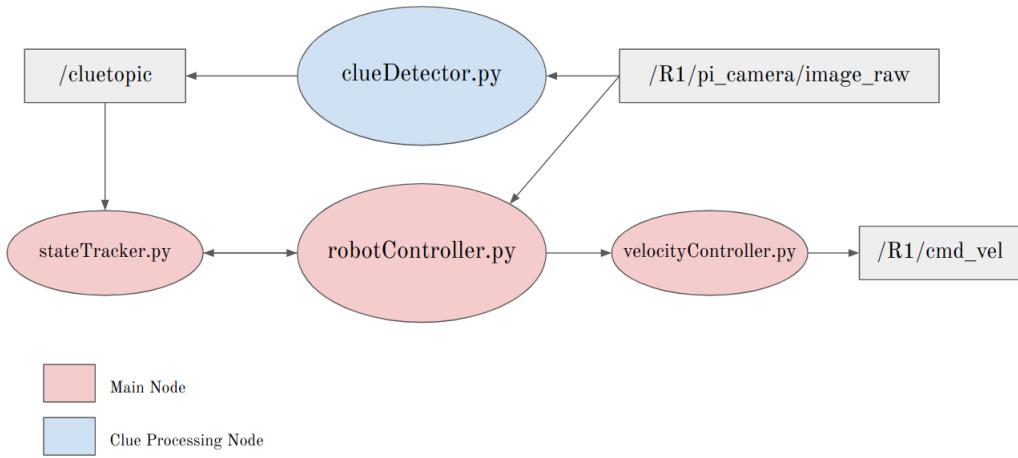


Figure 4: System node connectivity.

For this project, three new nodes were implemented: `clueDetector`, `robotController`, and a topic `\cluetopic`.

`clueDetector` is used for capturing and processing signs, as well as passing them into a CNN model for character prediction. As it is run in a separate thread, the robot can perform other tasks while it is processing signs.

`robotController` is the main node of our robot and is responsible for the driving, pedestrian detection, and state tracking to determine which point of the track the robot is currently at. `robotController` primarily uses images received from the `/R1/pi_camera/image_raw` topic to implement its functionalities.

For `robotController`, we use two helper classes: `stateTracker` and `velocityController`. `stateTracker` implements a finite state machine that will constantly update as images are received from the robot. `velocityController` is responsible for driving of the robot, which includes PID control and publishing messages to the `/R1/cmd/vel` topic. An overview of the `stateTracker` class is provided in Appendix C.

To enable communication between `robotController` and `clueDetector`, we implement a new topic called `\cluetopic`. This will be responsible for communicating data such as the number of clues detected by `clueDetector`.

2 Driving Algorithm

2.1 PID Driving

For driving, our robot relies on implementing PID control. As most of the track has painted road lines, our strategy involves masking the environment to find these road lines, then calculating an error value by comparing the centroids of the road lines with the center of the frame.

2.1.1 Road

When on the road, we mask for white by applying `cv2.threshold()` on a gray-scaled image of our environment. Then, we find all the contours in this image. To find only the white road lines, we go through the list of contours found and filter based on area. Then, we filter again for contours that are touching either the left, right, or bottom of the frame. After this process, we confidently and without error always obtain the two road lines as our final contours to be used.

2.1.2 Grass

When on grass, we still follow the same filtering methods as when on the road while adding a few more filtering parameters.

Firstly, we apply a mask to a HSV image of our frame and filter for red, green and blue. We then erode our image to lessen smaller contours and ensure that the road lines can be easily picked out. Then, as similar to the road, we find all the contours in our mask and filter based on area and if the contours are touching either the left, right, or bottom of the frame.

An additional filtering parameter we add is if the contours are above a certain threshold from the top of the frame. This way, we can guarantee that the contours we are filtering for are the road lines; they meet a certain area threshold, are touching either the left, right, or bottom of the frame, as well as having enough vertical length to meet a certain threshold from the top of the frame.

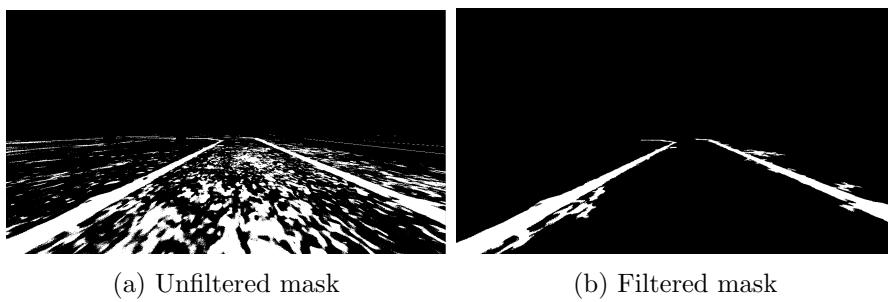


Figure 5: Grass line contours.

2.1.3 P-Controller

For driving, we implement a P-controller based on the comparing the centroid coordinates of the road lines with the center of the frame. Moreover, we compare if we can see two lines, or only one line.

In the case where two road lines are visible, we simply calculate the average x-coordinate between the two centroid coordinates of the road lines and compare it with the coordinates of the center of the frame. The green dot represents the center of the frame, the blue dot represents the center of the two centroids, and the red dots represent the centroids individually.

We then calculate the error through taking the difference between the x-coordinate of the center of the frame with the x-coordinate of the average centroid. This error is then scaled by a proportional constant and set as the robot's angular.z to be published to `cmd/vel`. Therefore, the robot will try to turn in a way such that the center of the frame will try to align with the center of the average centroid for the two road lines.

When only one road line is visible, we compare the coordinates of the center of the frame with the centroid coordinate of the single road line while using the same error calculation described above. The only difference is that when calculating error, we take the difference of the x-coordinate of the average centroid with the x-coordinate of the center of the frame. In this case, the robot will now turn in a way such that the centroid of the road line "turns into" the center of the frame.

At times, in order to ensure more reliable clue detection, we "bias" the robot in such a way that it tries to align itself with the left or right side of the road. This is implemented through adding a constant value in our error calculation.

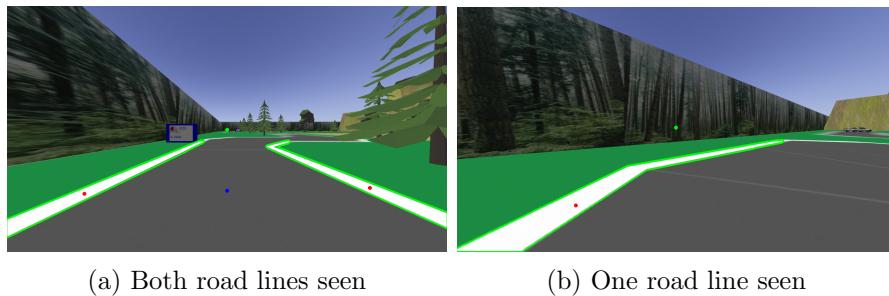


Figure 6: Road line contours.

2.1.4 Mountain

When on the mountain, we employ a special case of line following where we exclusively use one road line instead of two. To ensure that we read the last clue board at the top of the mountain, we decided that it would be best to follow the right road line on the mountain, as that leads directly towards the board.

Firstly, we apply a mask on a HSV frame of our environment to filter for the soil of the mountain. This exact values to filter for required a process of trial and error. Next, we find the contours on our mask and apply a series of filters to extract the final road contour to follow. Firstly, we divide the mask such that we only take contours that are located on the right side of our frame.

Next, we filter for area and check if each contour is a certain distance away from the right side of the frame. Moreover, we also apply `cv2.dilate()` to thicken the contours present in the frame. Once all these filtering methods are applied, we follow the contour with the largest area from the list of filtered contours. We calculate our error based on the distance between the center of the frame and the center of the centroid of our final contour.

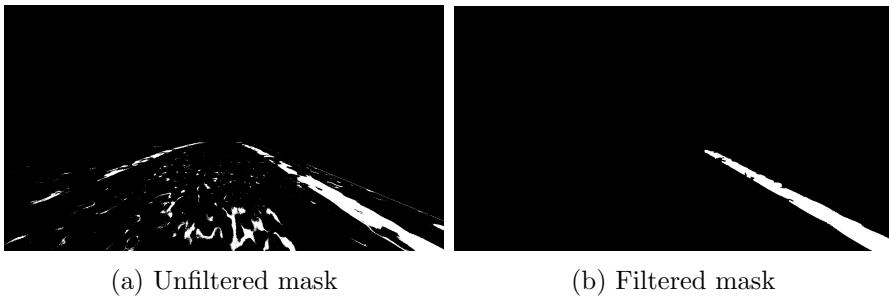


Figure 7: Mountain line contours.

2.2 Roundabout and Truck

Instead of directly detecting the truck’s location within the roundabout, our strategy involves driving along the white edge on the right side with two wheels off-road. By implementing this method, the robot will never interact with the truck inside the roundabout. After exiting the roundabout, the robot will continue driving with two wheels off-road until detecting the next clue board. Upon detecting the board, it will switch back to the “ROAD” state and navigate inside the white lines as usual. A full state diagram is shown in Appendix C.3.

3 Clue Detection

3.1 Character Recognition Neural Network

A Convolutional Neural Network (CNN) was developed and trained using TensorFlow and the Keras training framework. This CNN acts as a character classifier, taking images of individual characters as input and outputting the corresponding digit or letter. One-hot encoding is employed to label the 36 characters, converting each character label into a 1×36 vector. This vector corresponds to the 36 neurons on the last output layer of the CNN.

We use one-hot encoding to transform character labels into single-dimensional vectors, activating specific neurons based on the detected character. This output is then passed through

a categorical cross-entropy loss function to determine the probability of the input image corresponding to a specific class. The class with the highest probability is considered the prediction result.

3.1.1 Training Data

Training data was generated using the same font as the clue values, with white characters on a black background. Due to the blurriness, boldness, and stretching of characters in the robot's camera view after clue detection, a function `transform()` was created to create characters similar to what the robot sees in its environment.

The `transform()` function involves applying successive iterations of a noise map combined with stretching, then a reshaping of the image to 120×100 pixels. The full details of this function are available in Appendix B.

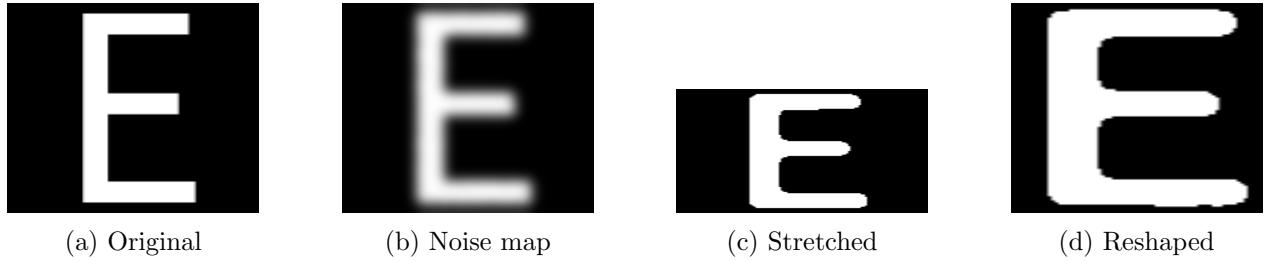


Figure 8: Training data before and after transformations.

3.1.2 Training Result

The CNN model was trained with 80 epochs and a learning rate of $0.75e - 4$. For training, 3600 samples were used (100 images per character). For validation, 900 samples were used (25 images per character). In the final epoch, the training loss reached $3.3114e - 11$, the validation loss was 0.0000, and both accuracies were 1.0000. See Appendix A for the model summary and confusion matrix.

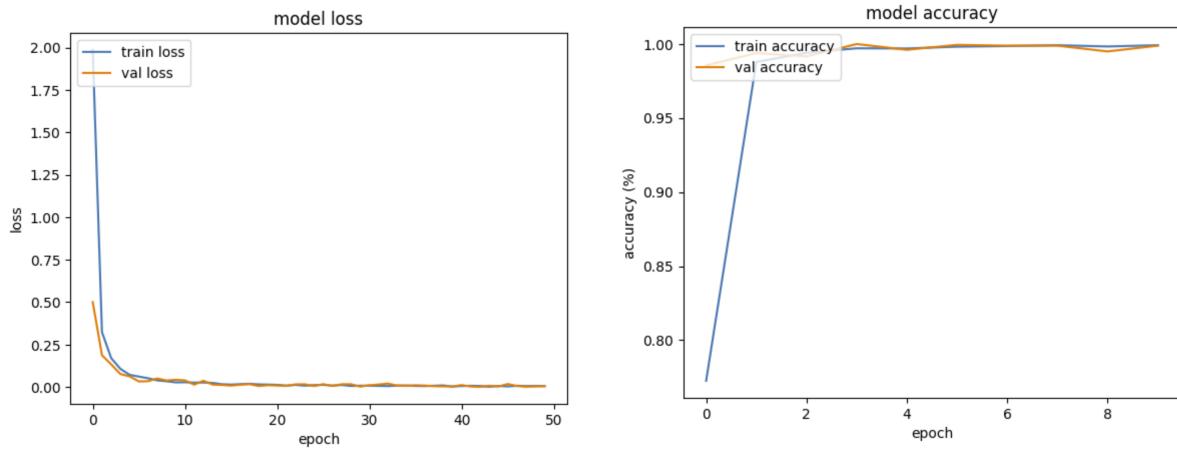


Figure 9: Model loss and accuracy.

3.2 Clue Board Detection

To detect the clue boards, two HSV thresholds are determined for the blue color. Using `cv2.inRange()`, the blue clue board, sky, and characters are converted to white, while everything else is converted to black. Another threshold range for sky blue, combined with `cv2.bitwise_and()` and `cv2.bitwise_not()`, is applied to filter out the sky. The `cv2.findContours()` function is utilized to locate rectangular clue boards, considering constraints such as area, aspect ratio, and the number of vertices. Similar constraints are again employed to isolate the white board region. Subsequently, `cv2.PerspectiveTransform()` is employed to transform the stretched image.



Figure 10: Clue board filtering.

Before trimming the characters, `cv2.erode()` is used to reduce blurriness, ensuring that each character is not connected to others. To maintain the order of characters for input into the CNN, the detected images are divided into top and bottom sections. Subsequently, `cv2.boundingRect()` is utilized to crop each individual character. These cropped characters are then saved as numpy arrays in a list, with each array being sequentially inputted into a CNN model to predict the clue values.

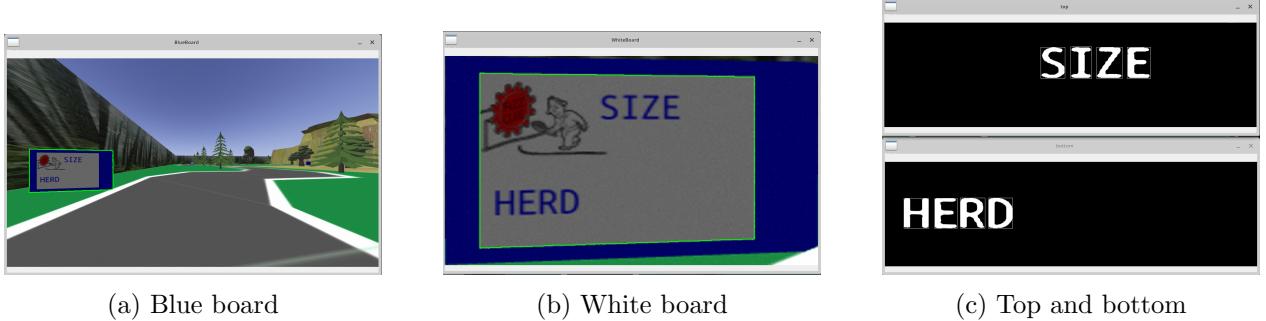


Figure 11: Clue board detection.

4 Barrier Detection

4.1 Pedestrian

For detecting and avoiding the pedestrian at the crosswalk, we rely on using their motion. To implement this, the robot first stops when the red line in front of the crosswalk is detected. Next, the previous frame of the camera feed is continuously stored and subtracted from the current frame. We implement this using `cv2.absdiff()` with the current and past frame.

With this implementation, everything from the current and previous frame are subtracted from each other except the parts of the frames that contain motion. Therefore, we can reliably isolate only the parts of the frame that are moving.



Figure 12: "Frame delta" at the crosswalk.

Next, we apply a mask to this "frame delta" by applying a binary threshold. As everything is black except the parts of the frame that contain motion, only the pedestrian is now highlighted in white. Furthermore, we dilate this mask to ensure it is easier to detect the pedestrian.

Afterwards, we crop the mask such that we are only looking at a section of the frame that corresponds to the center of the crosswalk. This cropped out section will always be black and will turn white as soon as the pedestrian crosses through the center. When this occurs, we change the state of our robot back to line following.

A function `pedestrianCrossed()` with the previous and current frame as arguments is used to implement the methods described above to detect the pedestrian.

4.2 Baby Yoda and Tunnel Reaching

The strategy implemented involved creating a tunnel mask to isolate the tunnel and enable the robot to follow its centroid after reading the sixth clue board. As the tunnel was stood out from the rest of the environment, it was relatively straightforward to implement a mask and have the robot use it.

As such, we were able to completely avoid the path of Baby Yoda. Therefore, no mechanisms were needed to implement Baby Yoda avoidance.

5 Conclusion

5.1 Future Improvements

In conclusion, there are areas where our robot can improve to achieve better performance. Firstly, it is crucial to enhance the accuracy and reliability of the clue detection algorithm. Although we applied `cv2.erode()` to each clue board image and used `boundingRect()` to identify characters, we still encountered cases where two or more characters were connected, leading to inaccurate predictions. One potential solution is manual trimming of each character by assigning a designated box based on their specific locations on the board. This approach ensures that even connected characters are separated for individual recognition. It's important to avoid inputting blank images into the CNN model, as it was trained without blank spaces.

Secondly, further experimentation with threshold values could lead to improved driving performance. While our current approach of following contours obtained through applied thresholds on the camera view and filters on the contours works acceptably, conducting additional tests to identify an optimal threshold value could result in smoother grass following.

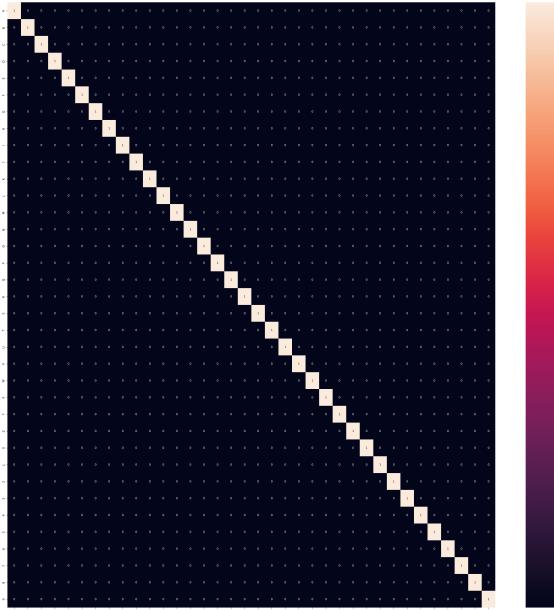
Appendix

A Neural Network Model Summary

Below is the neural network model summary and the corresponding confusion matrix with 100% accuracy in recognizing all 36 labels.

Model: "sequential"		
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 98, 118, 32)	320
max_pooling2d (MaxPooling2D)	(None, 49, 59, 32)	0
conv2d_1 (Conv2D)	(None, 47, 57, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 23, 28, 64)	0
conv2d_2 (Conv2D)	(None, 21, 26, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 10, 13, 128)	0
conv2d_3 (Conv2D)	(None, 8, 11, 128)	147584
max_pooling2d_3 (MaxPooling2D)	(None, 4, 5, 128)	0
flatten (Flatten)	(None, 2560)	0
dropout (Dropout)	(None, 2560)	0
dense (Dense)	(None, 512)	1311232
dense_1 (Dense)	(None, 36)	18468
=====		
Total params: 1.569.956		

(a) Model summary



(b) Confusion matrix

Figure 13: Model summary and confusion matrix.

Multiple neural network models were trained with altering ratios of training and validation data. Moreover, in addition to the generated characters, samples were also included directly from the track through processing characters on the clue boards in the competition environment.

B Character Generation

To generate data for training the CNN, several image processing functions are applied to manipulate the original images. First, a noise map is generated using a normal distribution with specified mean and standard deviation. This noise map is then combined with the original image using the `cv2.bitwise_and()` function to create a noise effect. Subsequently, Gaussian blur is applied to the image. The image is then stretched using `cv2.resize()` with a stretch factor of 1.7. Finally, the image is cropped to a dimension of 120x100 pixels. By experimenting with different values for the Gaussian blur kernel size, noise map iterations, and stretch factor, a test image is generated that closely resembles the actual images found on the competition surface.



Figure 14: Generated image of ‘S’ (left) and image from the competition surface (right).

C State Tracker

`stateTracker` is the class responsible for the implementation of a finite state machine, allowing the robot to change driving and clue detection techniques depending on which point of the track it is at. The robot has 6 states: “ROAD”, “PEDESTRIAN”, “ROUNDABOUT”, “GRASS”, “YODA”, and “TUNNEL”.

C.1 Clue Counter

The topic `/cluetopic` is used by `clueDetector` which publishes the number of detected clue boards. Subsequently, `stateTracker` subscribes to `/cluetopic` to obtain this information. Utilizing this data, `stateTracker` can then transition between states or adjust parameters such as P-control variables, biases, etc. dynamically across different sections of the competition environment. This adaptive strategy aims to enhance line following accuracy and optimize clue reading capabilities.

C.2 Pink and Red Line Crossing

Frames are constantly sent from `robotController` to `stateTracker` for analysis. Most importantly, we constantly check for if the robot has crossed a red or pink line using a mask; this will determine when the robot switches states. Mainly, the pink is used to transition from the “ROAD” state to the “GRASS” state.

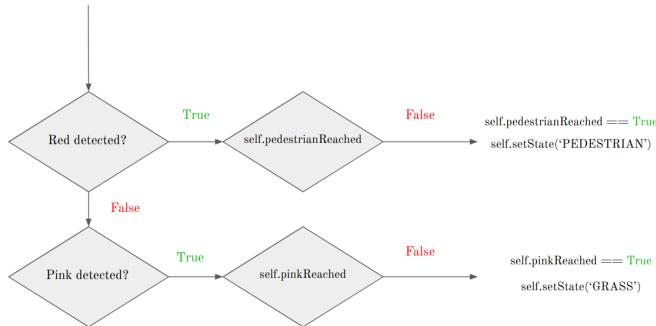


Figure 15: Using red and pink lines for state-switching.

C.3 Roundabout

Upon reaching the third clue board, the robot will initiate a full stop, execute a right turn at an angular speed of 1.05 for 1 second, and continue straight for approximately another 1 second. Subsequently, the robot will transition to the “ROUNDABOUT” state, following the white line within the roundabout and allowing the robot to exit naturally.

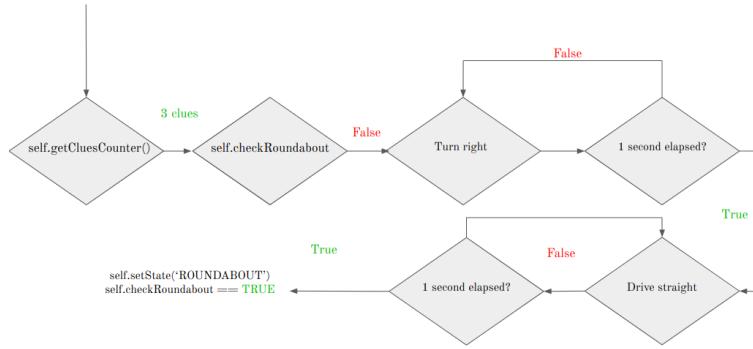


Figure 16: Roundabout process.

D Use of ChatGPT

ChatGPT was used to provide examples of how to write certain methods that we had thought of but were unfamiliar with their implementation. We also used ChatGPT to help debug issues that we were having with the code. Moreover, ChatGPT was also used to help maintain code tidiness and documentation.

 You

IF the distance is less than a certain value add to a new list of contours

 ChatGPT

To add contours to a new list if their distance to the top of the frame is less than a certain value, you can modify the code to include a condition that checks the distance and adds the contour to the new list accordingly. Here's how you can do it:

```
python  Copy code

import cv2

# Assuming 'lengthContours' is a list of contours
new_contours = []
certain_value = 20 # Adjust this value as needed

# Iterate through each contour
for contour in lengthContours:
    # Get the bounding rectangle of the contour
    x, y, _, _ = cv2.boundingRect(contour)

    # Calculate the distance to the top of the frame
    distance_to_top = y

    # Check if the distance is less than the certain value
    if distance_to_top < certain_value:
        # Add the contour to the new list
        new_contours.append(contour)

print("Number of contours added to the new list:", len(new_contours))
```

Figure 17: A snippet of a conversation with ChatGPT regarding contour filtering.

 You

why are my Ys identified as Vs with my cnn model

 ChatGPT

If your Convolutional Neural Network (CNN) model is misidentifying Ys as Vs, there could be several reasons for this issue:

1. **Training Data Imbalance:** Your training data might not have enough examples of correctly labeled Ys, leading the model to learn a biased representation.
2. **Data Augmentation Issues:** If you're using data augmentation techniques, they might be distorting the Ys in a way that makes them resemble Vs more closely.
3. **Model Architecture:** The architecture of your CNN might not be suitable for capturing the nuances between Ys and Vs effectively. You might need to adjust the number of layers, filters, or use a different architecture altogether.
4. **Overfitting:** Your model might be overfitting to the training data, meaning it's learning to recognize specific examples rather than general patterns. Regularization techniques like dropout or weight decay could help mitigate this.
5. **Preprocessing:** The preprocessing steps applied to your data could be affecting the distinction between Ys and Vs. Ensure that your preprocessing steps are not inadvertently altering the shapes or features of these characters.
6. **Noise or Distortions in Data:** If the Ys and Vs in your dataset are visually similar or distorted in a way that makes them hard to distinguish, the model might struggle to differentiate between them.
7. **Hyperparameters:** Experiment with different hyperparameters such as learning rate, batch size, or optimizer choice to see if they affect the model's ability to correctly classify Ys and Vs.

To troubleshoot this issue, try analyzing misclassified examples, adjusting your model architecture and hyperparameters, augmenting your dataset, or improving preprocessing techniques.



Figure 18: A snippet of a conversation with ChatGPT regarding neural network accuracy.