Implementation of Autonomous Detective Robot With Imitation Learning

Ella Yan, Nora Shao

ENPH 353

Miti Isbasescu

16 December 2024

1 Introduction

1.1 Background

In this project, we developed code for an autonomous robot to compete in a simulated virtual environment. The robot, acting as a detective, navigated through the environment to identify and collect clues related to a fictional crime scenario while adhering to traffic laws. The robot also had to avoid obstacles, namely the pedestrian, the truck, and Baby Yoda. The robot's performance was evaluated based on three main criteria:

1. Its ability to quickly navigate the competition course.
2. How well it avoided any stationary or moving obstacles.
3. Its accuracy in reading clue messages off the clue boards stationed throughout the track.

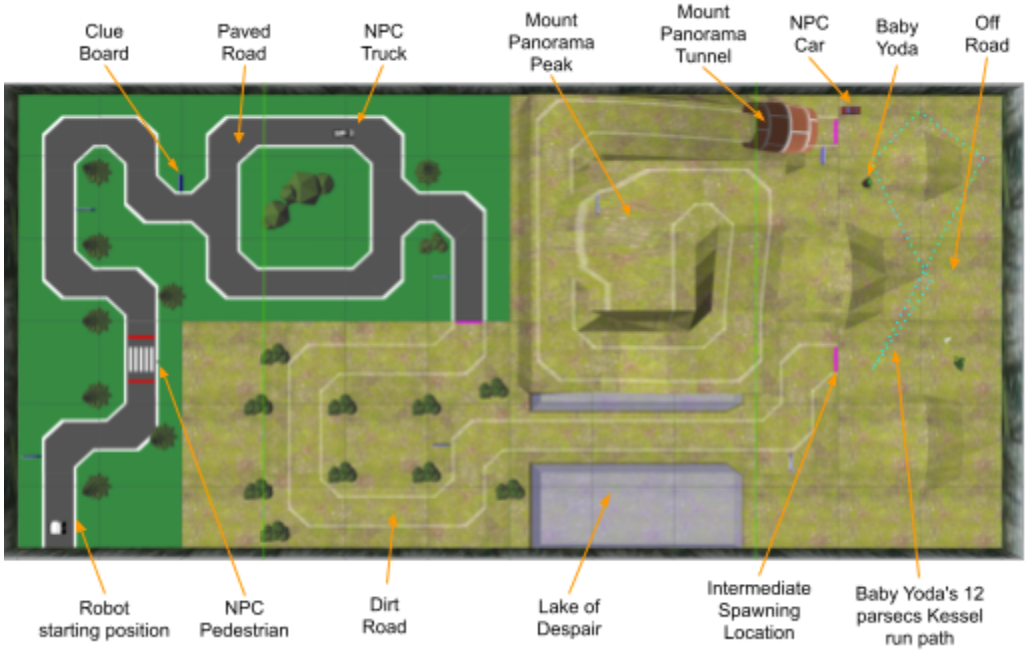


Figure 1. Overhead view of Gazebo Virtual Environment

1.2 Competition Strategy

Our goal was for the robot to consistently and reliably score the full 57 points while using machine learning (ML) for driving. Since each team was allowed only one attempt during the competition, we prioritized consistency and reliability over speed, though we optimized for speed where possible.

Our final strategy incorporated two convolutional neural networks (CNNs): one for clue reading and another for imitation learning (IL). We selected imitation learning as it provided a good balance between being challenging/interesting and feasible to implement. To increase reliability, we implemented a state machine to avoid obstacles and include failsafe protocols.

1.3 Contribution Split

Ella Yan was responsible for clue reading, which included clue board detection, letter segmentation, data generation, the CNN model, and the clue reader GUI. She was also responsible for the IL model which involved creating the imitation learner GUI, collecting training data, and training the model. Nora Shao was responsible for traffic detection and avoidance, which included the pedestrian and truck detection systems. She also set up the second camera to help clue reading and the respawning failsafe in case the robot went off course.

1.4 Overview of Software Architecture

Nearly all of our code resides in the controller_pkg ROS package. Our package contains two main nodes, one dedicated to reading clues (clue_reader) and one dedicated to driving (driving_controller). The figure below depicts a high-level overview of our two ROS nodes, the most commonly used ROS topics, and their relationships to each other.

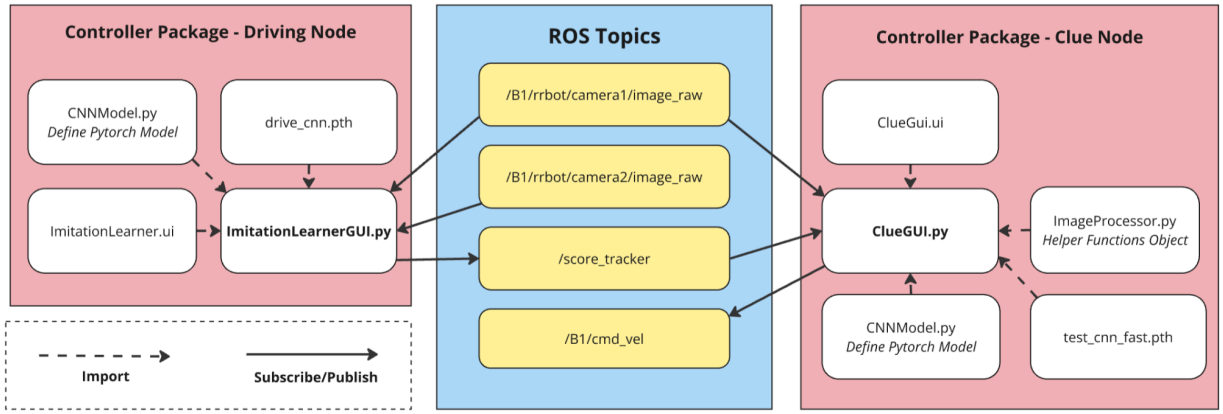


Figure 2. ROS Package Overview

The driving and clue reading nodes are initialized when their respective GUIs are run. The clue reader node imports various helper functions from *ImageProcessor.py* and loads a pre-trained CNN. It is responsible for processing camera images to identify and send valid clues to the /score_tracker topic. It is also responsible for stopping the timer once the last clue has been sent.

All of the driving logic, including state machine logic, is contained in the ImitationLearnerGUI.py file. This node is responsible for processing the camera footage from the centered front-facing camera and generating steering commands which are published to the /B1/cmd_vel topic. It also listens along to the /score_tracker topic in order to keep track of its rough location.

We set up an additional camera to support the clue board detection and reading. While testing, we discovered that the robot struggled to rea some clue boards due to awkward positioning. We added a second camera on the top left of the robot, angled 45° from the robot's horizontal. In the image callback, the robot would then select the camera input with the largest blue contour to read the clues from.

2 Clue Board Recognition

2.1 Clue Board Detection

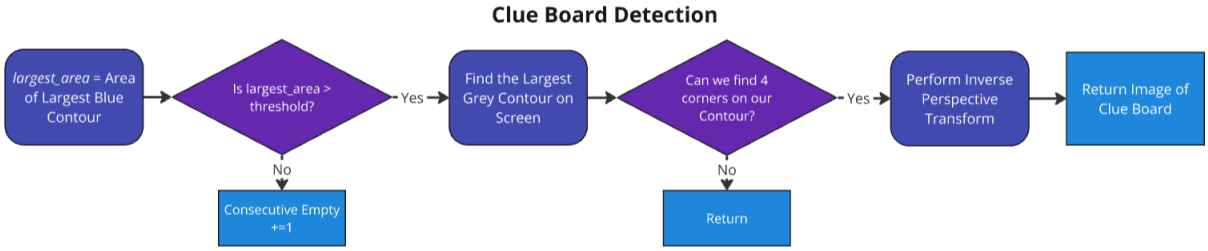


Figure 3. Clue Board Detection Algorithm

Our clue board detection algorithm uses a gray HSV mask and the morphological closing operation to make out the largest gray contour on screen. It then tries to find 4 corners and then uses that to perform an inverse perspective transform. The result is a nice clean head-on image of our clue board.

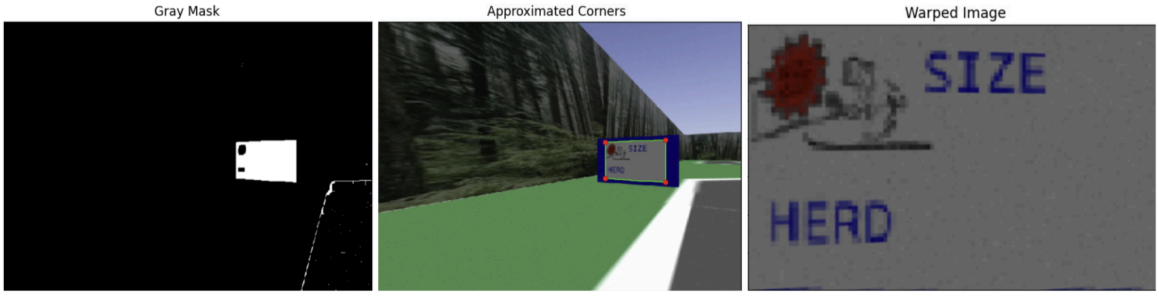


Figure 4. Clue Board Detection Masks

Additionally, we initially use a blue mask to check whether we are close enough to the clue board before even looking for gray contours. This is to prevent two issues, the first being that images taken from far away tend to be fuzzy so the letters tend to bleed into each other and the other being to prevent other gray objects from being detected as the largest rectangular gray contour.

We initially tried SIFT but found that it was extremely computationally expensive. We also originally attempted to locate blue rectangles instead of gray but found that this would cause issues when one was extremely close to the clue board. The edges of the blue sign would begin to exit the frame, destabilizing the algorithm.

2.2 Letter Segmentation

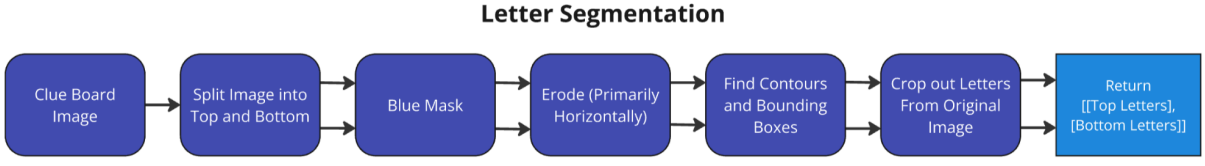


Figure 5. Letter Segmentation Algorithm

After the clean clue board was detected, the code would then attempt to extract the clue context and the clue itself. This was done by first separating the top and bottom half of the image. The top and bottom

image would each get its own mask of blue pixels. The algorithm would then locate contours and draw bounding boxes around said contours. The original top and bottom images would then be cropped along these bounding boxes. These sub-images would be turned gray and scaled to be 25 pixels wide and 35 pixels tall.

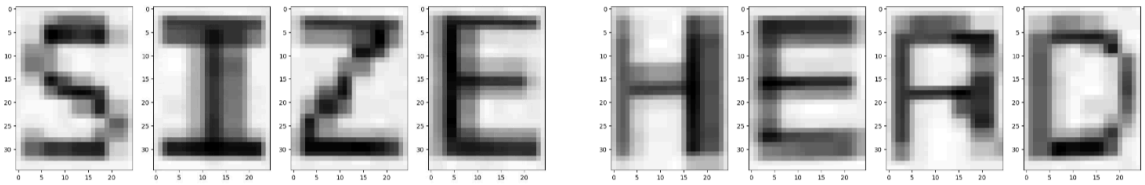


Figure 6. Cut Subimages From a Distant Clue Board

2.3 CNN Model

Our clue reading CNN model takes 25x35 grayscale images. It contains three convolution layers with ReLU activation functions. Each convolution layer is followed by a pooling layer. The output is then fed to two fully connected layers. The final layer contains 36 output neurons, each corresponding to a different possible character. Dropout is applied after the first fully connected layer to reduce overfitting.

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 25, 35]	320
MaxPool2d-2	[-1, 32, 12, 17]	0
Conv2d-3	[-1, 64, 12, 17]	18,496
MaxPool2d-4	[-1, 64, 6, 8]	0
Conv2d-5	[-1, 128, 6, 8]	73,856
MaxPool2d-6	[-1, 128, 3, 4]	0
Linear-7	[-1, 200]	307,400
Dropout-8	[-1, 200]	0
Linear-9	[-1, 36]	7,236

Figure 7. Clue Reader CNN Model Architecture

2.3 Data Generation and Training

We trained our model on artificially generated data for the sake of simplicity. We developed a function that would generate 25x35 images of letters using the UbuntuMono-R font. It would then apply various transformations to make our dataset more representative of the real data. The letters were gently rotated, violently scaled, brightened/darkened, randomly wobbled, and blurred (primarily horizontally). The images also had both small-scale Gaussian noise and large-scale noise applied. In the end, our training dataset had 12568 images and our validation dataset had 8505 images.

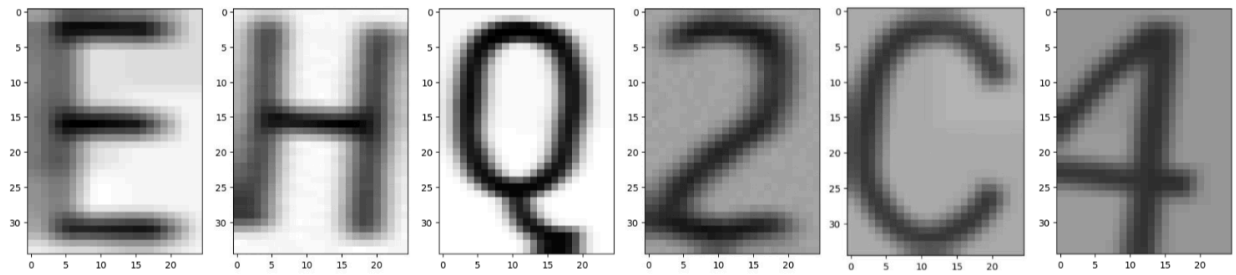


Figure 8. Training Data Samples

As this was a classification problem, we used Cross Entropy Loss as our criterion and Stochastic Gradient Descent as our optimizer. Our learning rate was 0.001, our momentum was 0.9, and our weight decay was 0.0001. We typically trained for about 10 epochs at a time. Our final validation accuracy was 99.80%.

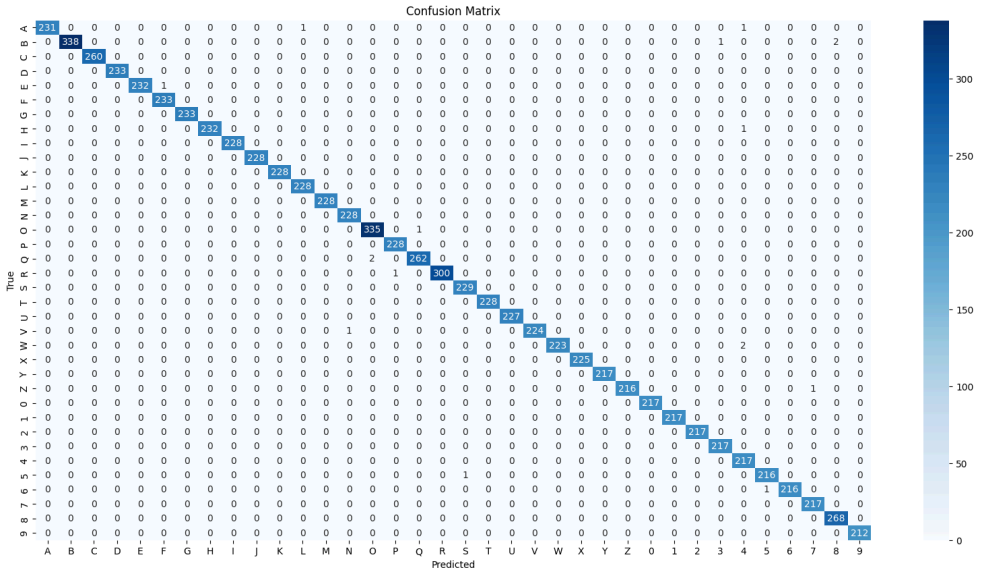


Figure 9. Clue Reader Confusion Matrix

2.4 Integration

All of the aforementioned image processing algorithms are called by the clue GUI. Upon image callback, the GUI looks for a clue board. If found, the GUI then attempts to cut out the letters in the top and bottom of the clue. The program then uses the CNN to predict the context and the clue. If the context detected matches one of the possible contexts (SIZE, MOTIVE, WEAPON, PLACE..), the clue is scored as a possible guess. The GUI continues to accumulate guesses until either there have been 5 consecutive images with no clue board, or one guess has been repeated 20 times. The GUI then publishes the most popular guess to the score tracker topic. See Appendix 6.5 for a flow chart.

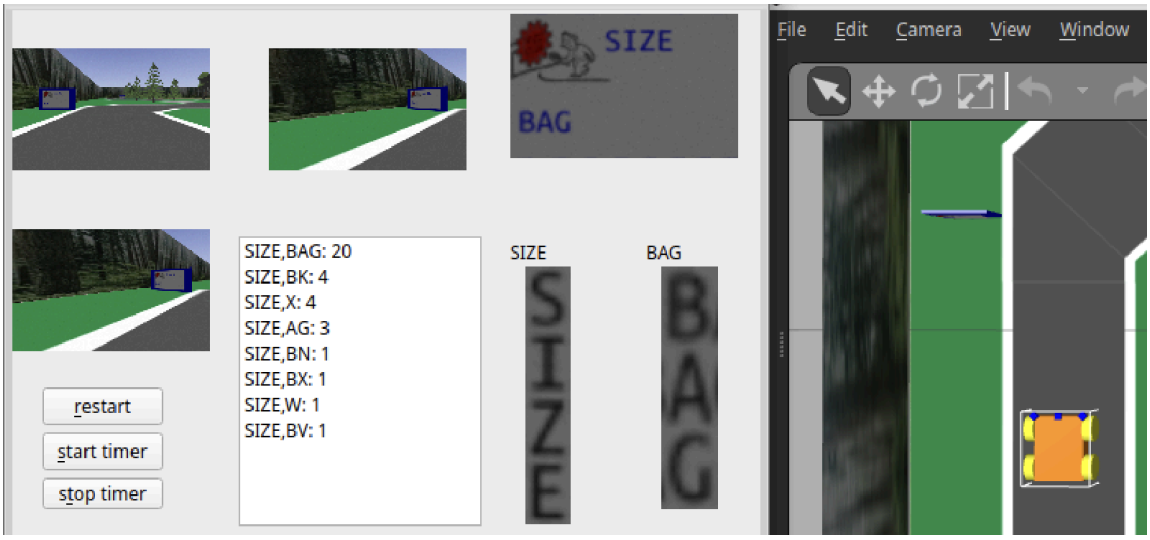


Figure 10. Custom Clue Reader GUI Making Predictions

3.1 Driving - Imitation learning

3.1 Data Collection

We began our IL journey by developing a robust robot controller and data logger. Since the model learns solely from the examples it is provided, we quickly realized that the erratic and disjointed movements generated by teleop_twist commands would not yield the high-quality data necessary for training our model well.

Our custom controller is integrated into ImitationLearnerGUI.py. The GUI contains a rectangle labeled “Steer Here!”. When clicked, it detects the position of your mouse and maps that to a linear and angular velocity command. The mapping is exponential so the middle of the rectangle is not very sensitive, but allows for finer adjustments whereas the outer edges of the rectangle are more sensitive and allow for much larger adjustments to velocity. The resulting controller is very smooth and easy to use.

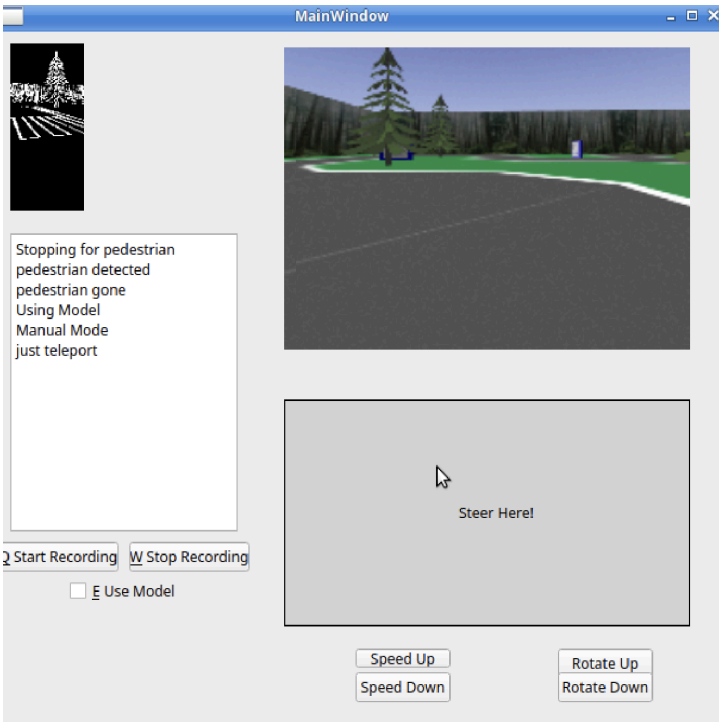


Figure 11. Imitation Learner GUI - Going Left and Forward

The GUI has a button that clears any previously built-up data and tells the GUI to start logging data. When collecting data, the GUI records the current image, current linear velocity, and current angular velocity and appends them to a list. Any data points where the robot is perfectly still discarded. Once the user hits stop recording, a popup appears, allowing the user to save the data as a gzipped pickle file.

We found that when it came to data, quality beat quantity. There are only a few full runs in the final dataset. Most of the footage is of us driving slowly past the clues in a manner such that they are visible to the camera, or of small tricky segments of the track. We also froze the truck in the simulation and collected data of us avoiding and waiting for the truck to pass, or of us following it at a safe distance. We did something similar with Baby Yoda where we drove up to the off-road segment and waited until Baby

Yoda appeared on screen. We then followed him across the off road segment. This proved to be the most reliable out of all the offroad strategies we tried.

In order to save space and increase speed, the images used by the driving algorithm had a much lower resolution than the original image as they were scaled to a size of 150x200.

3.2 CNN Model

The model takes 150x200 colour images. It consists of three convolution layers each followed by a ReLU activation function. The model pools after the last convolution layer. The flattened output of that layer is then passed to a shared fully connected layer. The model then splits into two fully connected layers: one for linear velocity and one for angular velocity, each with one output neuron. The output neuron for linear velocity and angular velocity will each output a float representing linear and angular velocity commands.

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 75, 100]	2,432
Conv2d-2	[-1, 64, 38, 50]	51,264
Conv2d-3	[-1, 128, 19, 25]	73,856
AvgPool2d-4	[-1, 128, 3, 5]	0
Linear-5	[-1, 400]	768,400
Linear-6	[-1, 1]	401
Linear-7	[-1, 1]	401

Figure 12. Imitation Learning CNN Model Architecture

We used Mean Square Error (MSE) as our criterion for our IL model. It made sense to use MSE here as this was a regression task rather than a classification task. We again used SGD as our optimizer with a learning rate of 0.001. We didn't use a validation dataset as collecting data was extremely time consuming and wouldn't be that helpful in determining whether the model was actually learning good habits or not

This model was trained using Apple's Metal Performance Shaders (MPS). This was significantly faster than training on a CPU and allowed us to continue training despite running out of time on Google Colab.

3.3 Testing

The Imitation Learner GUI has a checkbox that allows one to quickly switch between using the IL model or manual driving. After training, we would deploy the model in Gazebo and observe the model's performance in driving. Any time it would go off course or do something undesirable, we would switch to manual mode and log more data. Afterwards, we would retrain and repeat.

4 State Machine

The robot's movement was controlled by a finite state machine. In general, when it was just following the path, the robot's actions were determined by the IL model, which acted as the default state. However, there were a few dynamic obstacles that the model struggled to reliably so when flags indicating these obstacles were triggered, the robot would switch to their respective handling states.

The first obstacle that necessitated a state switch was the pedestrian that periodically crossed the sidewalk and could hit the robot, pushing it off the path. This is an especially tricky obstacle for IL to avoid as it requires temporal reasoning, something a basic CNN model is not capable of. The second obstacle was

the truck in the roundabout that could also hit the robot. The final obstacle occurs when the robot reaches the offroad area guarded by Baby Yoda's 12-parsec Kessel Run.

4.1 Pedestrian Avoidance

The pedestrian handling algorithm had two main components: determining when the robot was at the sidewalk and when the pedestrian had crossed the crosswalk.

The crosswalk detection was implemented by using a red mask to check for the red boundaries of the crosswalk. When a sufficiently large contour was found in the bottom third of the robot's camera frame, the robot would pause the IL driving controller and wait for the pedestrian.

The pedestrian detection was implemented with a movement mask on the left third of the camera input so the pedestrian would be the only large moving contour in the frame (note the truck moving on the right in Figure 13). To minimize noise, we converted the fray to grayscale, then a sufficiently large difference in pixel intensity (thresholded by number of pixels passed in as a size argument to the function) found between two successive frames would indicate the pedestrian was crossing the street. This would return the robot to the IL driving state.

Figure 13. Pedestrian Detection View in GUI (Movement Mask Top Left)

4.2 Roundabout & Truck Avoidance

Similar to the pedestrian detection system, to ensure the truck did not collide with the robot in the roundabout, the robot switched into a 'truck detection' state when it encountered the roundabout. Unlike the crosswalk's red borders, the roundabout had no distinctive markers we could use to detect it. Instead, we used the clue board right at the entrance of the roundabout as the marker; when the third clue guess was published to the score tracker topic, the robot would also pause the IL driving and switch to the truck detection state.

We identified two optimal locations for the truck to be where it was safe for the robot to enter the roundabout. Condition 1 was met when the truck has just passed the entrance of the roundabout, the robot could enter and just follow behind the truck. Condition 2 was met when the truck is on the far side of the roundabout; there would be enough distance between the robot and truck that it was unlikely the truck could catch up to the robot before the robot exited the roundabout on the other side. When either condition was met, the robot would return to the IL-controlled driving state and enter the roundabout.

To identify Condition 1, we implemented the same movement mask described in Section 4.1, but with modified parameters. As shown in Figure 14, the movement detection function was applied only to the

left half of the camera frame to confirm that the truck had already passed the robot. Additionally, we increased the pixel threshold to account for the truck's large size compared to the pedestrian.

To identify Condition 2, we applied the movement mask to the entire frame. To ensure the truck was passing on the far side of the roundabout rather than directly in front of the robot, we set both an upper and lower bound on the size of moving pixels in the movement mask to ensure the truck was smaller and in the distance.

Figure 14. Truck Detection in GUI (Condition 1)

4.3 Baby Yoda Avoidance

Ultimately, we found that the most reliable and controlled method of crossing the offroad section was to follow baby Yoda. The IL model was partially trained to wait and follow Baby Yoda, but it was not as reliable as we required.

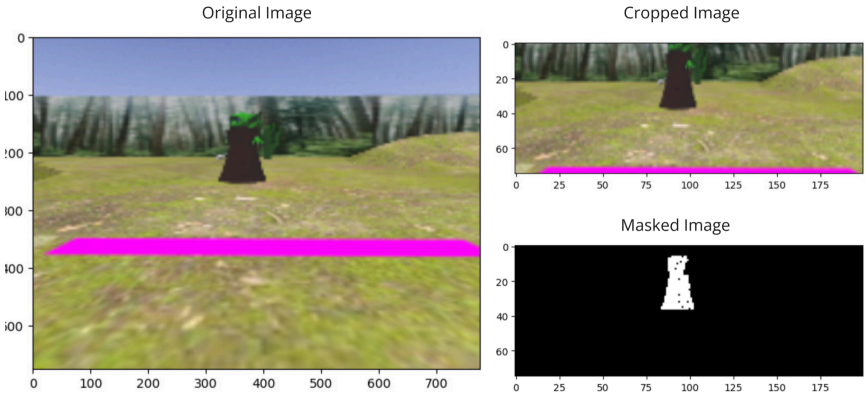


Figure 15. Baby Yoda Masks

The logic involved first waiting for clue six to have been sent. The code then began to look for a magenta line in the bottom half of the frame. If the line was big enough, the code would then turn off the model and wait for baby Yoda to appear. Baby Yoda would then be detected by masking for dark brown, and seeing if the largest contour was larger than a certain threshold. If baby Yoda was spotted, the model would turn back on and follow baby Yoda as it was trained to do so.

4.4 Failsafe Mechanism

While we tested the driving algorithm for the robot, we recognized the importance of having a failsafe in case the primary driving controller suddenly stopped working. We identified two potential flag conditions

that could necessitate a switch from primary driving to the backup algorithm: the robot driving off the path and a long duration between clues being read. To avoid developing any new functionality to differentiate between the robot being on and off the path, we settled on the latter flag.

We timed the duration between each of the clues being read and sent to the score tracker and found that the longest duration between two successive clues between sent was roughly 60 seconds. Thus, we set the robot to switch to the backup algorithm if the time since the last clue being read exceeded 85 seconds.

Once the condition was met, the robot would teleport to the spawn location at the foot of Mount Panorama and return to the IL-controlled driving. This way, although we would lose 2 points for the respawn, the robot could still gain the last 14 points from reaching Mount Panorama Peak.

4 Results

4.1 Competition Results

We scored 51 points during the competition. Points-wise we would have tied for second but factoring in time, we placed 5th overall. While we didn't win, we are proud of being the only team to get ML driving working consistently at competition. Most of the other teams either used PID line following or a drone. We would like to think that we successfully captured the spirit of the course by tackling the imitation learning challenge.

4.2 Room for Improvement

During the competition, we submitted an incorrect guess for the clue "HIGH NOON" which cost us 6 points. Our theory is that since the double Os are very wide, they somewhat bled into each other, causing them to be cut as one letter. This issue could have been addressed by leveraging the properties of the monospace font. For instance, we could have checked for any "letter" that had an abnormally wide aspect ratio and then split those in half. Alternatively, we could have segmented the word into individual letters at fixed positions. While we attempted to implement this solution, we did not have enough time to fully test it.

Our code also ran fairly slowly. To mitigate this, we initially optimized the clue board detection process by switching from SIFT to grayscale contour detection. Additionally, we reduced the size of our clue-reading CNN to 6% of its original size by shrinking the fully connected layers. These changes kept the frame rate acceptable—until we added a second camera. The second camera significantly slowed down the system. Once all GUIs and the score tracker were running, our frame rate dropped further. To compensate, we reduced the robot's speed to ~85% of its original value, allowing more time for processing and updating linear and angular velocities. However, slowing the robot down only further decreased our chances of winning the tie-breaker. Ideally, we would have tested adjusting the simulation speed or spent more time optimizing the code, particularly the lengthy image callback functions.

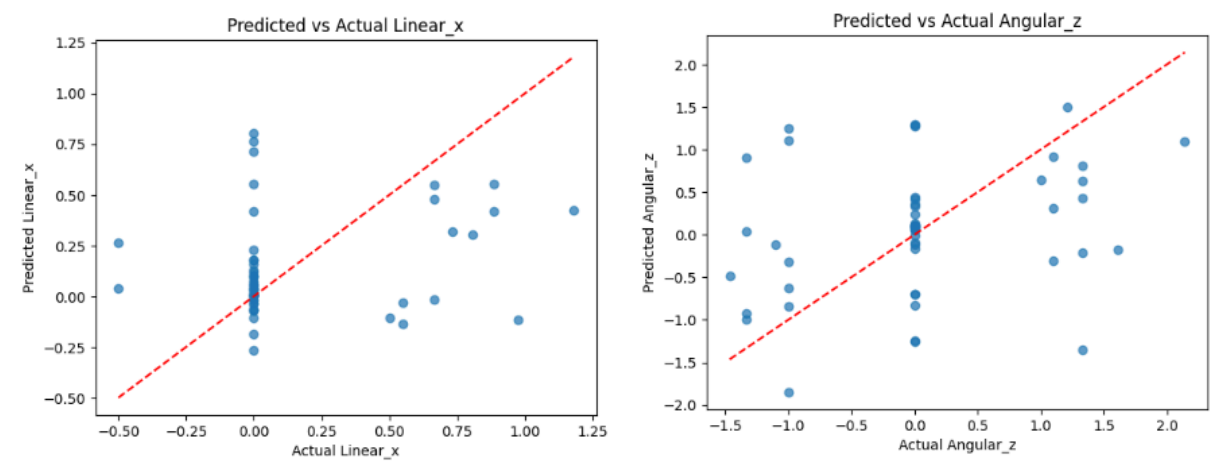
5 References

We spoke to Amjad Yaghi and Michael Koo about alternatives to SIFT clue board detection. They both gave me the idea to use rectangle detection instead. We also spoke to Ronald Nelson quite a bit early into the project about various strategies and clue board detection. He also suggested the idea of integrating a custom controller with the robot's movements due to the awkwardness and difficulty of using teleop_twist commands. Steven Xu from the year above also gave us the idea to follow Baby Yoda to get to the tunnel from the beginning of the off road area.

6 Appendix

6.1 Alternative Driving Algorithms

We started trying to train an IL model on the teleop_twist keyboard commands, but found that the model performed terribly; the difference between the model's predicted actions and the actual actions we made for a specific camera can be seen in the figures below. We realized that while part of the problem was that the CNN for the imitation model was not properly tuned yet, the quality of training data was also a major issue. For example, given the same camera input, in each different training iteration, it is highly like a human controller could be moving forward or turning left or turning right depending on whether their plan changed or they made a mistake with the keyboard commands and had to correct their actions. This motivated us to later implement another controller.

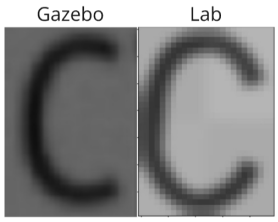


IL Model's Predicted Velocity Commands vs Actual Velocity Commands

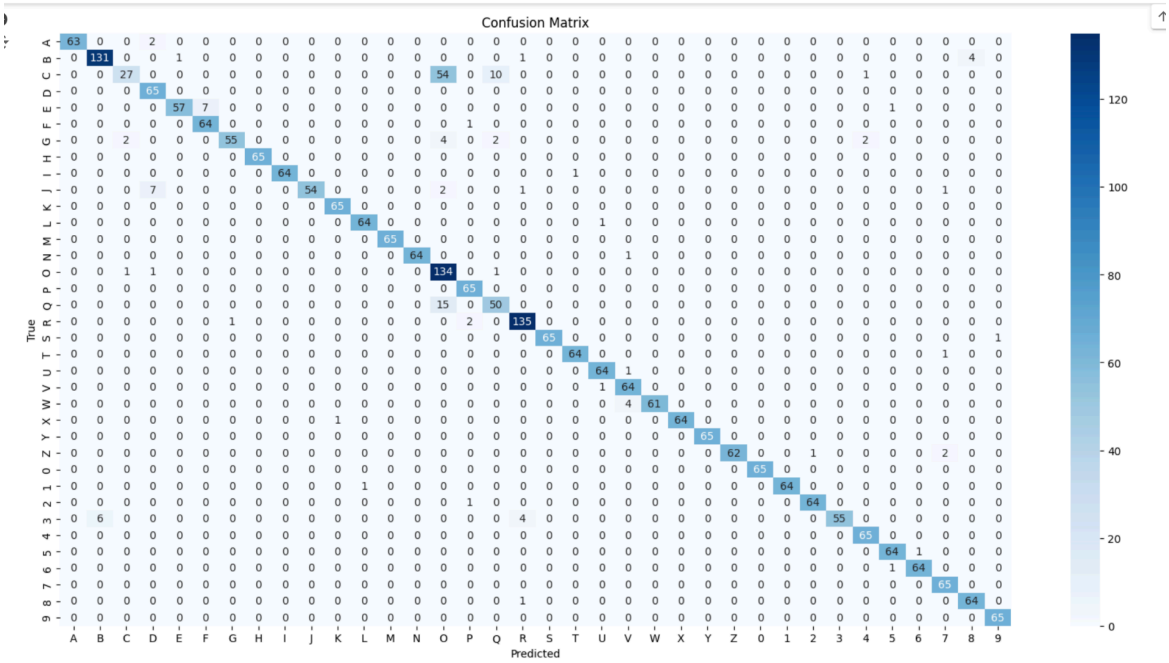
We also developed PID-controlled driving as a backup to the imitation learning model. It used a binary mask and contour detection to follow the bounded road, and similar to our current implementation, used a state machine to switch between normal path following and obstacle avoidance. However, the robot also had to switch between PID states when it moved off the dark paved road onto the grass because of the difference in colour changing the thresholding values for the mask.

6.2 Clue Reader Mishaps #1

When we tested our first iteration of the clue reader CNN, we noticed that it failed at detecting the letter "C". We realized that this was because the font from the labs and the font in the Gazebo simulation are different.



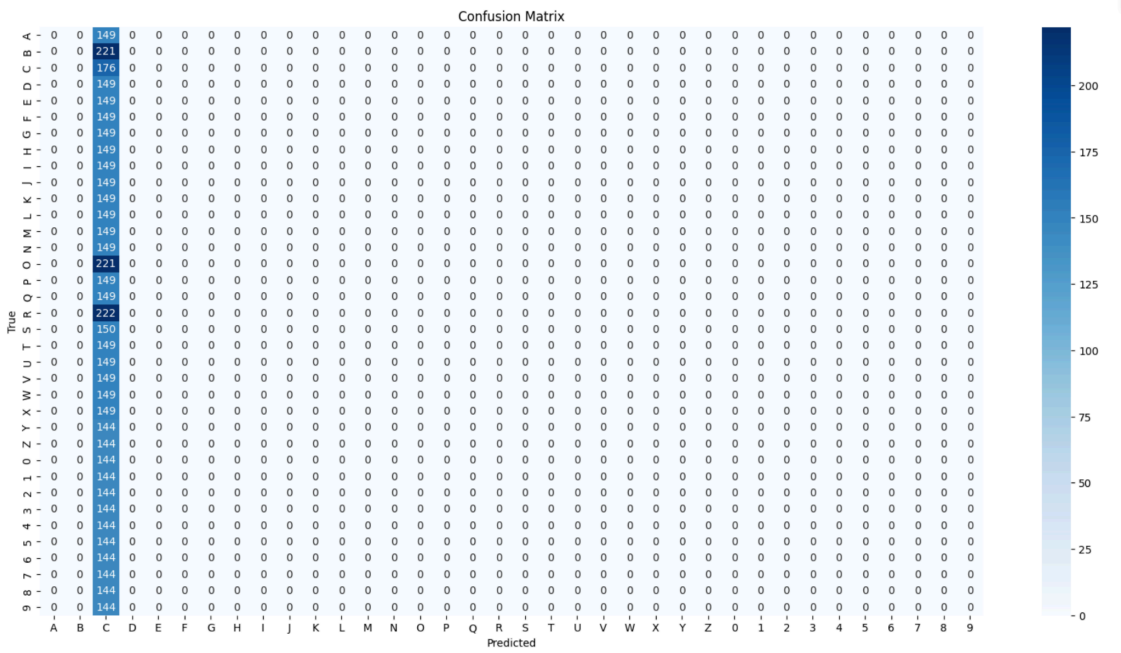
My confusion matrix was perfect until we added the gazebo font to our validation set. When we regenerated my confusion matrix (without retraining), we found that the model was significantly worse at guessing the letter “C” than we had initially thought.



Confusion Matrix After Adding Gazebo C’s

6.3 Clue Reader Mishaps #2

While trying to optimize our clue reading pipeline, we took a look at the size of our clue reading CNN. We found it odd that our CNN model was about 25Mb in size. After looking at how we defined the model, we found that our model was so big because we had 4096 neurons in just one of the fully collected layers. We decreased this number to 200 and the model size lowered to be approximately 1.6Mb with no adverse effect on accuracy. However, when we tried to further remove neurons, our confusion matrix would end up like this:



Confusion Matrix after Decreasing Neurons in Fully Connected Layers

We also used ChatGPT to calculate how many parameters we had originally compared to how many we had after the reduction.

1. **Original Configuration**:
   - Conv layers: $92,672$
   - FC layers: $6,447,140$
   - **Total**:

$92,672 + 6,447,140 = 6,539,812$

2. **Updated Configuration**:
   - Conv layers: $92,672$
   - FC layers: $314,636$
   - **Total**:

$92,672 + 314,636 = 407,308$

6.4 Model Architectures - Code

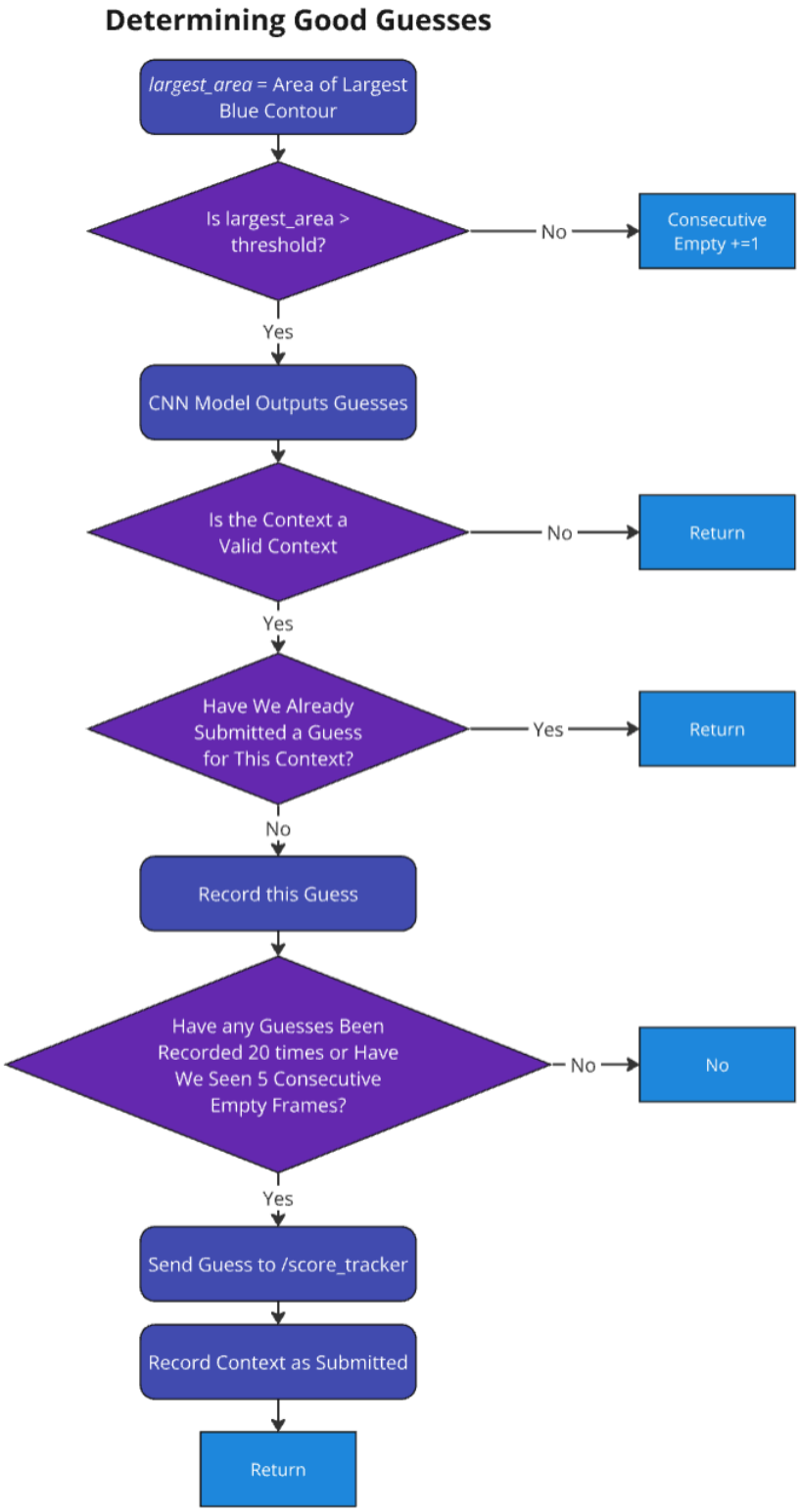
```python
class CNNModelFast(nn.Module):
    def __init__(self):
        super().__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding=1)

        # Fully connected layers
        self.fc1 = nn.Linear(128 * 4 * 3, 200)  # Adjusted for the output of the convolutional layers
        self.fc2 = nn.Linear(200, 36)  # Assuming 36 output classes (e.g., A-Z, 0-9)

        # Max Pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Dropout to prevent overfitting
        self.dropout = nn.Dropout(p=0.5)

    def forward(self, x):
        #x = x.permute(0, 3, 1, 2) #put channnel in the right place
        x = x.unsqueeze(1)
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        #print(x.shape)

        # Flatten the tensor for fully connected layers
        x = torch.flatten(x, 1)  # Flatten all dimensions except batch

        # Fully connected layers
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.fc2(x) # Output layer
        #no softmax cuz its in crossentropyloss already
        return x
```

Clue Reader CNN Model Code

```python
class DriveCNN(nn.Module):
    def __init__(self):
        super(DriveCNN, self).__init__()
        # Shared convolutional layers
        self.conv1 = nn.Conv2d(3, 32, kernel_size=5, stride=2, padding=2)  # Output: [32, 75, 100]
        self.conv2 = nn.Conv2d(32, 64, kernel_size=5, stride=2, padding=2)  # Output: [64, 38, 50]
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1)  # Output: [128, 19, 25]

        # Replace Adaptive Pooling with fixed pooling
        self.pool = nn.AvgPool2d(kernel_size=5, stride=5)  # Output: [128, 4, 5]

        # Update linear layer dimensions
        self.fc_shared = nn.Linear(128 * 3 * 5, 400)
        self.fc_linear = nn.Linear(400, 1)  # Outputs for linear velocity
        self.fc_angular = nn.Linear(400, 1)  # Outputs for angular velocity

    def forward(self, x):
        x = x.permute(0, 3, 1, 2)  # [B, H, W, C] to [B, C, H, W]
        #print(x.shape)
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        #print(x.shape)
        x = torch.flatten(x, 1)

        # Shared fully connected layer
        x = F.relu(self.fc_shared(x))

        # Separate outputs
        linear_out = self.fc_linear(x)
        angular_out = self.fc_angular(x)

        return linear_out, angular_out
```

Imitation Learning CNN Model Code

6.5 Determining a Good Clue Guess - Flow Chart



Logic For Determining a Good Guess