ENPH 353 Autonomous Controller

Logan Underwood and Josh Bauman

ENPH 353

Dec 6th, 2023

Table of Contents

Table of Contents 2
Introduction 3
  Background 3
  Contributions 3
  Software Overview 4
Discussion 4
  Modules 4
    Sign Detection 4
    CNN Text Reading Model 5
    Single Object PID 7
    Motion Detection 7
    Line Detection 8
  Overview of Controller 8
    Paved 9
    Grass 9
    Mountain 10
    Driving Algorithms 10
    Pedestrian and Truck Handling 11
    Our Off-Road Solution 12
Conclusion 13
  Competition Performance 13
  Some Dead Ends 13
    Mountain 13
    Acceleration Curves 13
    Retraining The CNN 13
References 13

Introduction

Background

This year's rendition of the ENPH 353 robot competition was a challenge that required a robot capable of many different tasks as it traversed the world's terrain. During the traversal of the map, we were required to read the signs and put together the clues of a murder mystery. Our robot was allowed to receive information from only two inputs:

1. A timer
2. A camera mounted on its front

Condition	Points awarded / deducted
First 6 Clues	+6pts
Final 2 Clues	+8pts
Reaching Tunnel	+5pts
Maximum points	57

With this limited information we not only had to navigate our robot through the world, but we were also presented with unique challenges such as a pedestrian, another vehicle, a Baby Yoda, and a small clearance tunnel.

You will notice that 51/57 points are from reading the clues, so you must be able to read all the signs with consistency. However, to read all the signs you needed to be able to reach them. You need to be able to navigate the map's 4 distinct domains (see Figure 1):

1. Paved Section
2. Grass
3. Off-road
4. Mountain

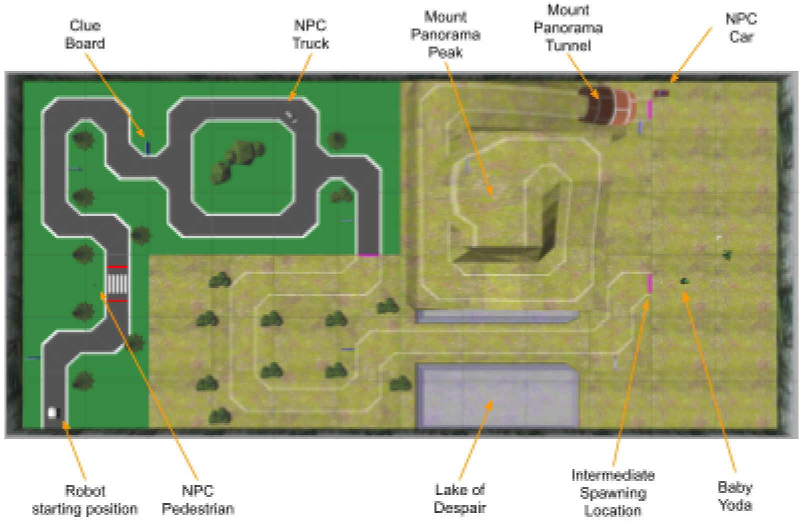


Figure 1 - Competition Map

Each section boasted its own challenges that require creative solutions to navigate and with a high level of consistency. In the paved section, some critical challenges included dealing with a moving truck, a crossing pedestrian, and choosing the correct direction to drive. The grass section brought new lighting conditions and line markings. Conversely, the off-road section required new methods of using the environment to navigate with the lack of road lines.
Similar to the grass section, the mountain gives even more difficulties because of the changing lighting and shadows. In addition to each environment-specific challenge, you must consistently know where you are on the map using reliable state-switching logic.
The best solution required the development of several abilities to extract information from the camera using tools like OpenCV and Keras (CNNs). However, the most profound challenge was integrating the tools into a robust controller capable of traversing the map with high levels of consistency.

Contributions

Logan Underwood	**Josh Bauman**
- Motion Detection (Pedestrian and Truck) - Roundabout Traversal Logic - Sign Detection Pipeline - Text Reading Model Training - Baby Yoda Section Traversal - Tunnel Entrance - Changing Sections Line Detection	- Road image masking and processing - Paved Driving Control - Grass Driving Control - Mountain Driving Control - Sign Detection Pipeline

Software Overview

ENPH353 Software Diagram.pdf

Discussion

Modules

In the below section, we will explore the different modules of our robot in detail. We will demonstrate the critical pieces that allowed us to design and build the robust systems used in our final design. Each piece played an essential role in building a successful controller.

Sign Detection

Our method for sign detection was a significant focus during the development process. Since clue detection was worth the most points, we knew this aspect needed to be rock solid. The first step in our development was data collection. For this, we drove around the map using keyboard inputs to collect raw images that we could later experiment with a Jupyter Notebook. Using this data (see example in Figure 2), we established 2 distinct qualities the clue boards could be identified by:

1. Blue Outline
2. The red element of the logo

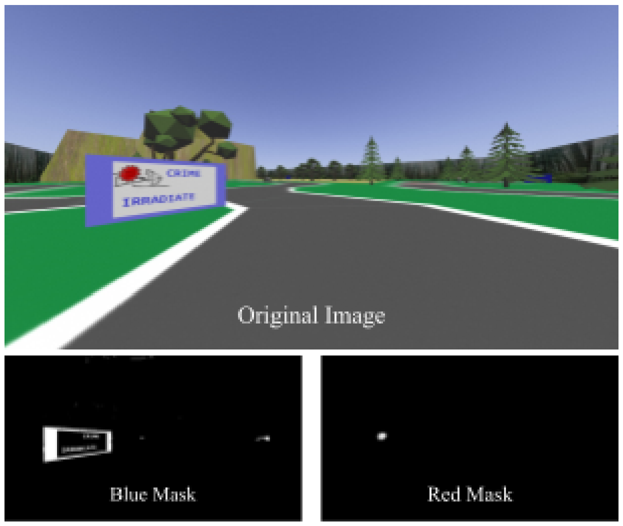


Figure 2 - Image Masks

These 2 characteristics were unique to the signs, so they could be used to accurately identify sign location in our camera feed. As we received each frame from the camera, we identified signs as all areas which had a blue contour of a minimum size and also contained a red circle inside. Once classified as a sign, we performed a perspective transform to reshape the blue outline of the sign into a consistent rectangle. We then performed another perspective transform on the white section of the sign (see Figure). This second transform was critical to preparing the image for cutting, as the first transform still left the interior white section containing the words slightly warped and rotated.

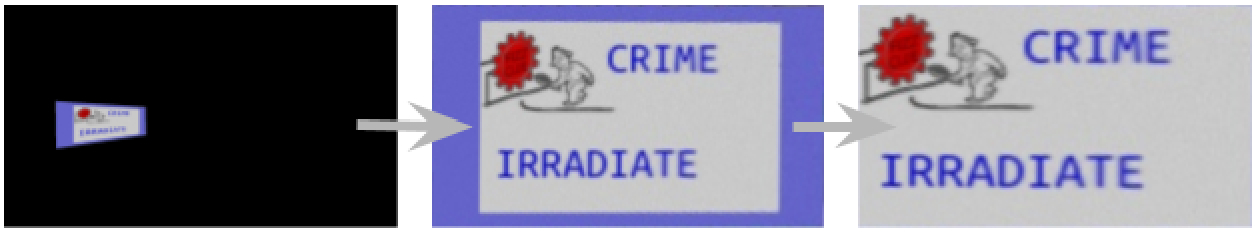


Figure 3

Since the image processing could guarantee the output to be of similar shape and orientation every time. In addition, all of the signs had the 2 words starting in the same location, and all letters had the same width. This consistency allowed us to define specific pixel ranges on the transformed image where we could find each letter. We carefully determined the values of these pixels by analyzing a variety of sign outputs to ensure that the region cutoffs would work for all letters (see section __ for more detail).
After some work developing our text-reading model, we needed to add an additional. It was critical for us to shift as much of the work away from the model to assist with data generation and training. We binarized the images and removed any blurry sections leaving only the meaningful pieces of the image behind as the input to the model.

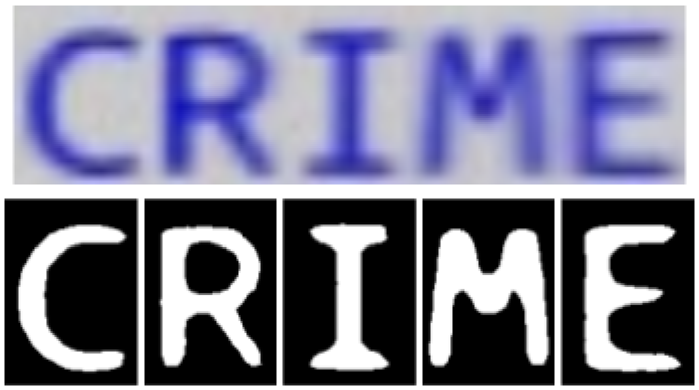


Figure 4

CNN Text Reading Model

The goals of our model were simple; it needed to be able to take images of characters extracted from the world as inputs and output correctly which character was in the image (note that this includes numbers as well). Outside of a few letters, the model was not challenging to build with the correct input data. Since the data that our model would be required to predict on during competition was not very complex (see figure _), we believed that the easiest way to train our model would be to generate our own dataset of text matching the text found on the signs. Firstly, we created a set of letters from {0,1,...,9,A,B,...,Z} (the original can be seen in figure _). After building out the set we performed a variety of operations so the input data would match the expected input from the competition. We built the dataset using the following steps:

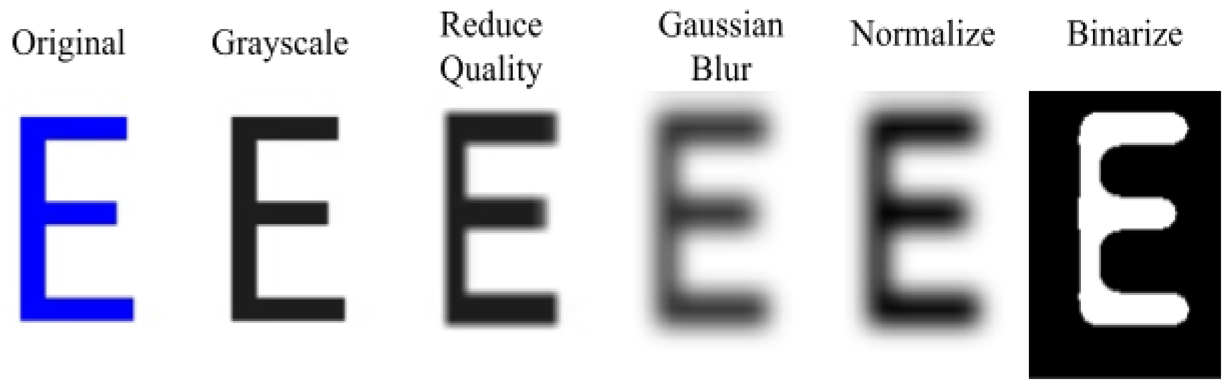


Figure 5

To improve the variance in the data set, we introduced some randomness in the amount of blur and level of image compression. The most critical element of our data was having sufficient randomness to model the input data. Since our data was entirely generated it was essential for the data engine to be tested and visualized. Running this randomized data preparation script we trained our model on 1000 of these randomized letters. On these letters, we performed another batch of preprocessing using Keras.Preprocessing.Image's ImageDataGenerator. This additional segment allowed us to perform small rescaling, rotations, shearing, size scaling, and shifting (see Figure 6).

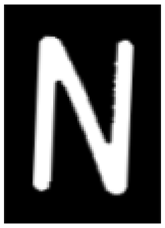


Figure 6

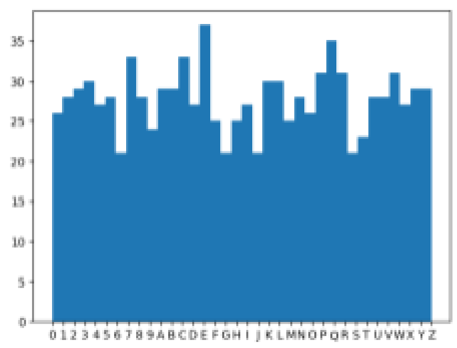
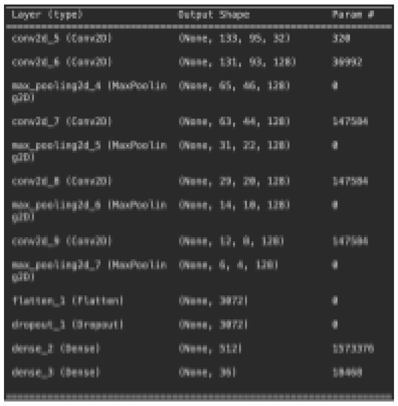


Figure 7 - Data Histogram

The next piece of the puzzle was the neural network structure. We had 2 competing priorities during testing; accuracy and forward pass speed. With those two requirements in mind, we experimented to find the smallest possible network size which converged to ~99% accuracy (see figure 8)

For our training, we settled on the following parameters:

1. **Epochs:** 15
2. **Learning Rate:** 1e-3
3. **Batch Size:** 75
4. **Optimizer:** RMSProp



```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_5 (Conv2D)            (None, 133, 95, 32)       320

conv2d_6 (Conv2D)            (None, 131, 93, 128)      36992

max_pooling2d_4 (MaxPoolin   (None, 65, 46, 128)       0
g2D)

conv2d_7 (Conv2D)            (None, 63, 44, 128)       147584

max_pooling2d_5 (MaxPoolin   (None, 31, 22, 128)       0
g2D)

conv2d_8 (Conv2D)            (None, 29, 20, 128)       147584

max_pooling2d_6 (MaxPoolin   (None, 14, 10, 128)       0
g2D)

conv2d_9 (Conv2D)            (None, 12, 8, 128)        147584

max_pooling2d_7 (MaxPoolin   (None, 6, 4, 128)         0
g2D)

flatten_1 (Flatten)          (None, 3072)              0

dropout_1 (Dropout)          (None, 3072)              0

dense_2 (Dense)              (None, 512)               1573376

dense_3 (Dense)              (None, 36)                18468

=================================================================
```

Figure 8 - Model Params

During the gradient descent, we had immediate accuracy and loss improvement tending >90% in less than 3 epochs. The final 12 cycles allow for the clean-up as we approach near perfection. In the below graphs, it can be noted that validation performance tends to jump ahead of the training set. The reason is the validation set is run at the end of the cycle so the model has improved since the first batch run during the train set. The graphs verify two important details. The model has converged on an accurate local minima and it has the model is capable of handling minor deviations in the provided images.

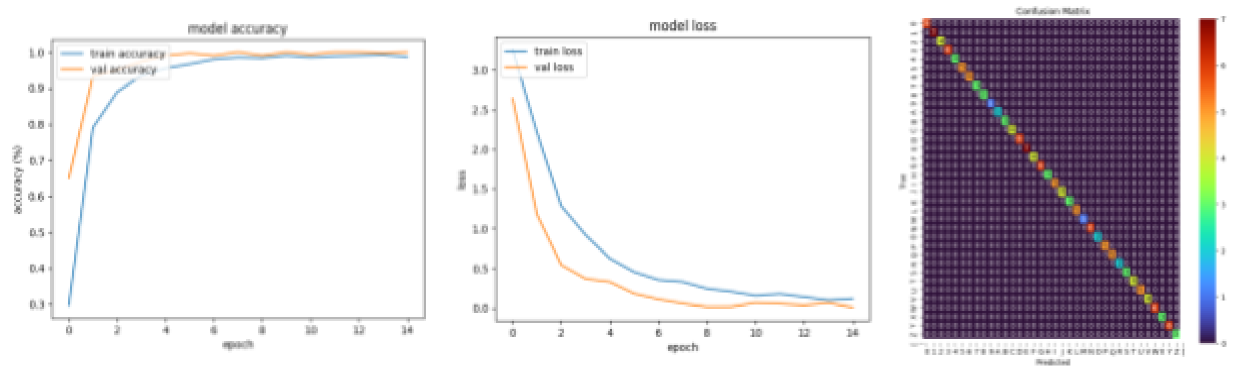


Figure 9

However, since our training and validation set were generated, our testing was another critical component of the NN development process. We ran the model on the competition surface and reached a ~99% success rate. The night before the competition, we predicted 4 words incorrectly in 23 total runs (8-word predictions per run). If we had run into more issues with our model we would have employed more data from the actual competition but since one of our first models was so effective we did not feel the need to do so.

Single Object PID

This blanket term of single object PID is defined as driving towards a single object in the frame of view. Examples include driving towards a sign, a magenta line, or the tunnel. Each specific case uses its own procedures but all share a common procedure:

1. Mask the object and find the centerline (in the x direction) of the object
2. Define a function to convert the centerline into a velocity command.

For example in the Baby Yoda section, we first use the blue sign and then the magenta line to PID towards (see more below). After finding the centre line of the desired object we used a linear error function to determine how many pixels were the difference between the centre of the magenta line and the centre of the car's camera. Although you don't need to use a linear error function (in fact we use a sigmoid for other sections), the linear function was the best for single object PID.

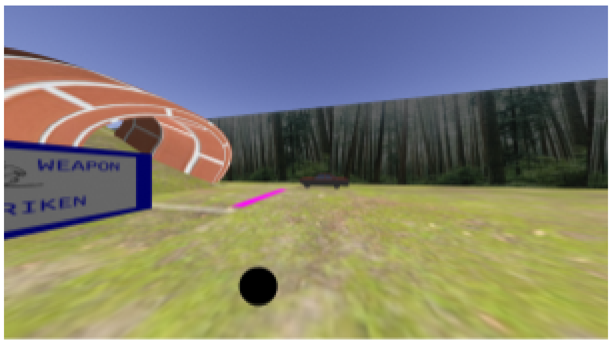


Figure 10 - Magenta Line locating

Motion Detection

There are 2 sections of the map where we used motion detection; the crosswalk and the roundabout. Like many parts of the competition, the best solution was a simple one. The lack of wind or other moving objects made a novel solution of image subtraction particularly appealing.

To detect the pedestrian or truck, the car would have to wait and collect two consecutive images. The output of the difference of

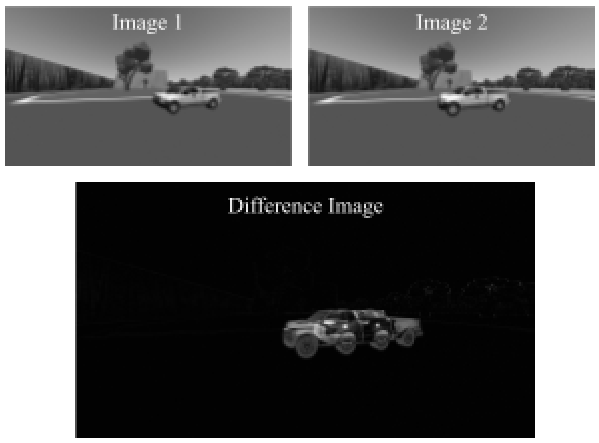


Figure 11

the pair of images would show the changes that occurred between the pair.

As seen in Figure _, when the truck passes through the frame the different image captures the change. More detail on how we used this to avoid these moving objects is discussed in the below section.

Line Detection

There are a set of line markers around the map which can be used to identify context clues about where the robot is. The major lines are the magenta ones which inform changes in regions and the red line in front of the crosswalk. To identify lines we employed a mask followed by finding a bounding rectangle for the line. The process is outlined below:

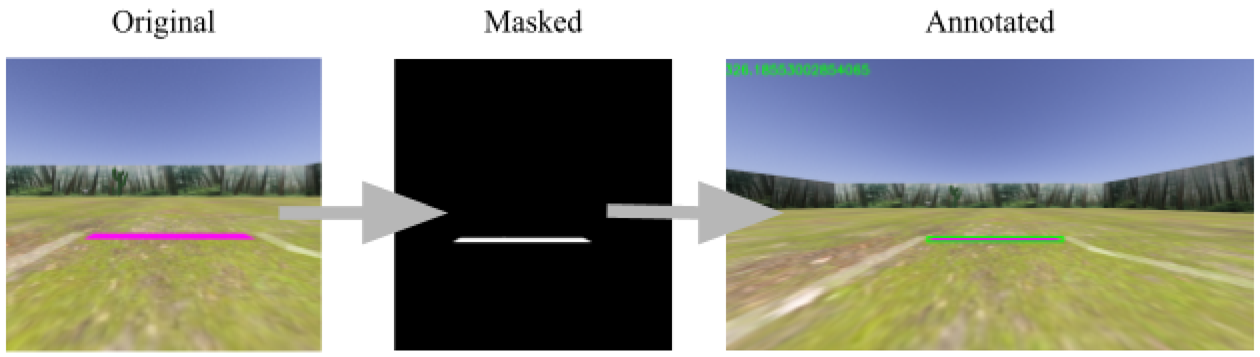


Figure 12

Note the number in the top right of the annotated image which denotes the size of the line. For different sections, we wanted different thresholds for line size to trigger a certain behaviour. For instance, at the crosswalk, we wanted to stop in a very consistent location to help assist our motion detection thresholds. The line size can be used for that purpose.

Overview of Controller

As mentioned in the background section of this paper, completing the course required our robot to independently navigate its way through 4 distinct terrains, each bringing new problems to be solved. At its core, our robot's control/driving algorithm consisted of 4 main steps.

1. Process image data to extract road lines or guiding features as best as possible
2. Locate the centre of the road / desired headway for the robot
3. Convert this location into a velocity command
4. Check for edge cases

The last 3 steps could mostly be solved the same way for each of the terrains, however, how road lines were extracted varied substantially. Our driving algorithm was dependent on our ability to process road images and extract the information we were interested in, so it makes the most sense to look into our image processing first.

Paved

Apart from the challenges involved with avoiding pedestrians and trucks, this section was the easiest to navigate through. The clean road lines and high contrast between the colours of the road lines and the rest of the robot's field of view made extracting the road lines straightforward. As the lighting in the simulation was not guaranteed to be consistent all along our path, we decided to work on our images in the HSV colour space as it would make our filters more reliable. The RGB or BGR colour space does not isolate hue from brightness; if we didn't switch to HSV, the values for the masks would be highly dependent on the brightness at a given location reducing accuracy as a result. Once our images were converted to HSV, we ran a mask thresholding script that allowed us to change mask values and view the mask applied to our image in real-time. This quickly gave us the mask values we were looking for, and we were then able to test the mask on our images, this time in the full simulation setup. and the results were just as we expected.

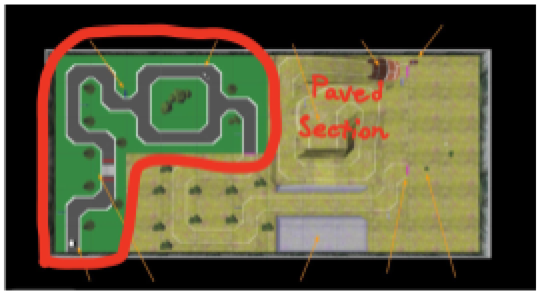


Figure 13

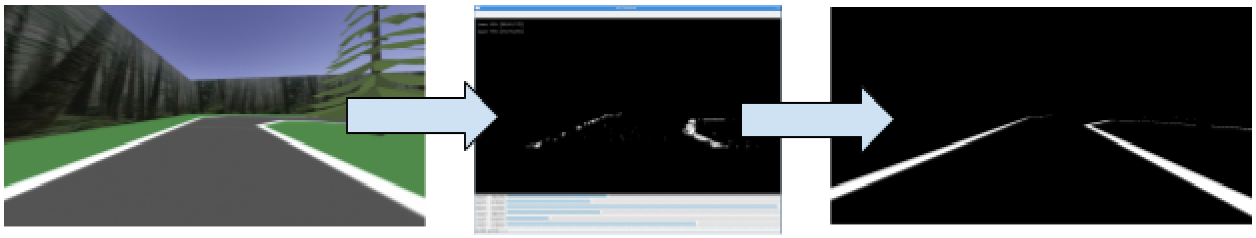


Figure 14 - Paved Road Masking Process

Grass

The grass section did not contain any moving obstacles like the paved section, however, it still proved to be tricky to navigate. The simple masking that we used for the paved section no longer gave us clean road lines to use for navigation. This is because the colour of the lines was also present throughout the grass and surroundings, so it made it impossible to only extract road lines. After applying our mask, the image was plagued with many random blotches that caused our driving algorithm to fail and steer the robot off the path. To solve this problem, we turned to a couple of useful functions provided in the OpenCV library. We started by applying an erosion kernel to the image to eat away at pixels along the borders of the many blotches in the image. The idea behind this was that the erosion kernel would be able to remove a lot of the high-frequency noise (blotches) in the image and leave it cleaner for further processing. This erosion step would also eat away at the road lines, however, this can be reversed once we get rid of the unwanted noise. We then used the findContours function in the OpenCV library to obtain all the contours in the image. These contours could then be sorted by area, and we could discard any below a certain threshold value. For the contours that did survive our filtering, we drew onto an empty copy of the image and continued with the cleaning process. Our last step was to try and replace some of the pixels on the road lines that got eroded. To do this we used a dilation kernel which can be

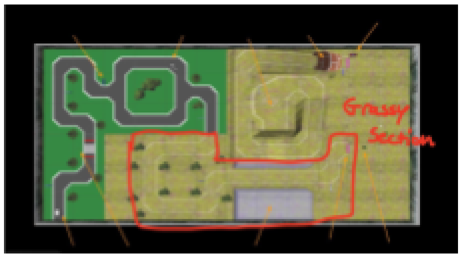


Figure 15

thought of as an inverse to the erosion kernel. After some tuning of the minimum contour area threshold, our stream of images consisted of clean road lines that were easy to follow.

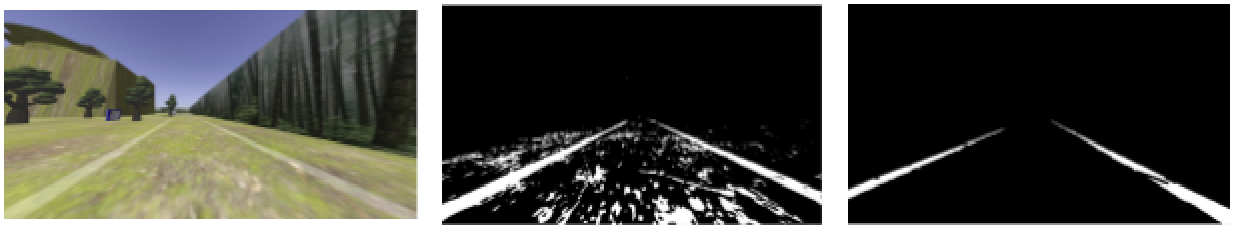


Figure 16 - Grass Road Masking Process

Mountain

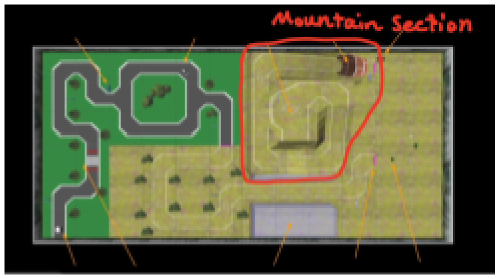


Figure 17

The mountainous section of the track ended up causing the most difficulty because of the different shadows that were cast by the sun onto the roads. Thinking this portion of the track would be no different from the grass section, we tried using the same methods of image processing. We quickly realized that was not going to work. After troubleshooting what the underlying difference in this section was, we narrowed it down to the shadows being cast on the mountain by the sun. Because of the shadows, the mask we were using for one part of the mountain was filtering out the road entirely on another part of the mountain. Our approach when solving this problem was to break the mountain up into distinct sections where we could then deploy different image filters correspondingly. This resulted in us using 4 different masks depending on how far up the mountain we were. To determine which section of the mountain we were climbing and when to switch masks, we looked at the average brightness of the pixels in the region of the image where the road existed. This served as a decent measure of how much shadowing was present and which mask to apply. Once we knew which mask to apply, the images were processed following the same procedure as the grass section.

Driving Algorithms

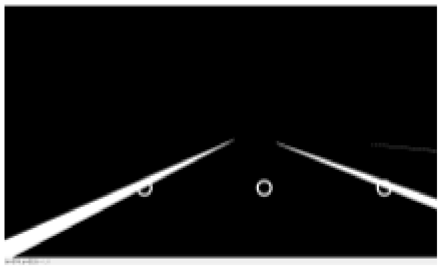


Figure 18 - Road Coordinate Visualisation

The role of our driving algorithm was to interpret the images being provided by the camera feed, and produce a velocity command such that the robot would follow the road laid out ahead. Given each processed frame, our algorithm would search a row of pixels a set distance in front of the robot to locate coordinates for the inside edges of the white road lines. The search would happen from the outside in on both sides. We then averaged these coordinates to find the centre of the road. Knowing our camera was secured to the robot, we knew the forward heading of our robot was always aligned with the centre of each camera image. Taking the difference between the road centre and the centre of each frame gave us a measurable error in pixel distance. To map this error into a velocity command, we started by passing the error value into a sigmoidal function of our choosing. To create this function we altered a standard sigmoid function in desmos until it fit our desired input/output space. An ideal input space would account for all possible error values that our robot could return. In our case, this was half the width of the image in pixels on the positive input axis and half the image width on the negative axis. As for output space, we wanted to see the output begin to saturate around our maximum possible input. Figure __ shows the actual function we landed on. Once we had this function, we scaled the output by the inverse of the function amplitude so that we obtained a value between

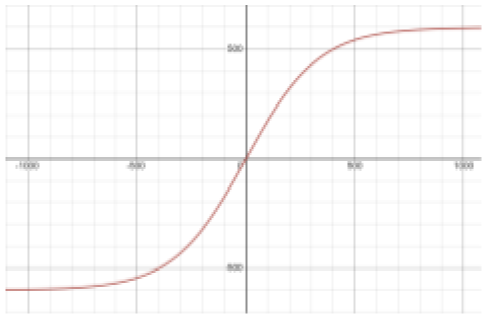


Figure 19 - Sigmoid Activation

0 and 1 for any given input error. Lastly, we defined a maximum angular velocity for the robot and used the product of this maximum and the sigmoid output as our robot's error-correcting velocity command. At first, we programmed the robot to travel at a constant linear velocity and receive angular velocity commands from our algorithm to correct errors. Seeing an opportunity for improvement, we added functionality so that our linear velocity was scaled depending on the degree to which we were turning. This meant increased speed on the straightaways and overall a faster run!

Pedestrian and Truck Handling

Above we discussed the motion detection tools, however, we did not discuss how we used the tool to pair it with our controller to avoid the objects. For both the truck and the pedestrian we studied their motion and devised a solution for each case.

For the pedestrian, we employed a simple strategy of waiting for him to cross the street before we moved on. To accomplish this task we needed to differentiate when the pedestrian was on the side versus crossing the road. Firstly, we masked the image so that the movement of the truck in the background did not clutter the image. Afterwards, we used the motion detection module. We used the size of the moving blob with carefully selected thresholds so they only triggered when the pedestrian walked across the frame and not when he was spinning off to the side.

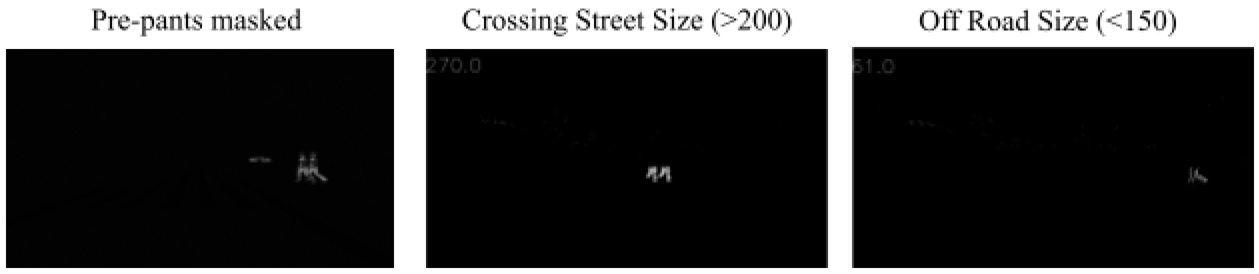


Figure 20 - Pedestrian Masking

For the truck section, we employed a more complicated approach. We divided up the screen into left versus right. To get the location of the truck we used a combination of the location of the moving pixels and their size. If the truck was on the right then we continued forward, but if it was visible on the left then we would wait.

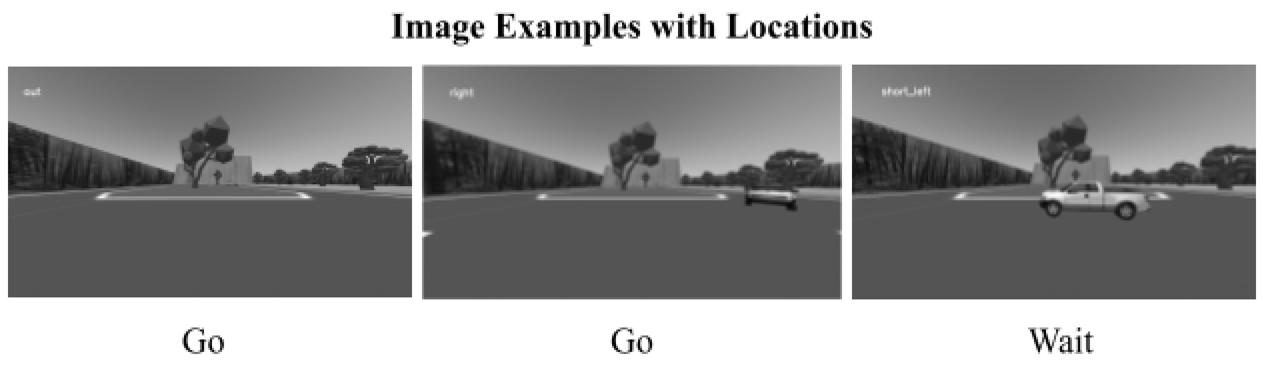


Figure 21 - Truck Decision Visualization

Our Off-Road Solution

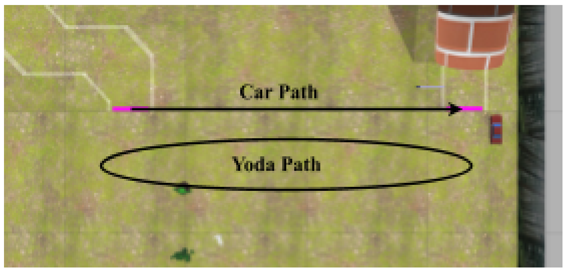


Figure 22

The off-road section was a completely different set of problems compared to any other driving section on the map. Although you do not have road lines you do have a variety of different objects to follow. The section of the track requires a complicated logic shown below. The 3 major goals of the solution were to avoid any collisions with the moving Baby Yoda, read the sign near the tunnel, and approach the tunnel in a consistent pattern.

Note that our path avoids Yoda completely by taking the line of open space from magenta line to magenta line (see Figure _). Although you cannot see the magenta line from such a far distance you can spot a small section of the blue sign. So we decided that we would follow the blue sign and then switch to following the magenta line once sufficiently close. The closer we are to the sign the more the magenta line is in view. We switch our target centre line once the blue sign reaches a certain number of pixels on our screen. Although pixel calculation was vulnerable to different approach angles, the range of outcomes of our angle is such that it was never an issue.

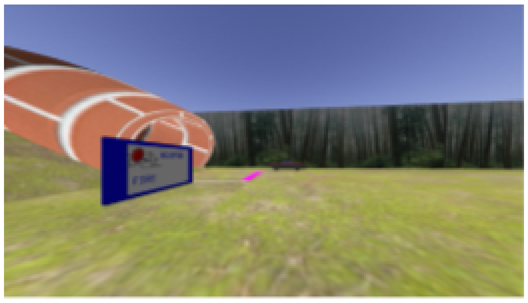


Figure 23 - Angle of Approach to Tunnel

As we approach the new magenta line we have a full view of the sign which can be easily read. After reading the sign, the only remaining task is to prepare for the tunnel approach. To accomplish this, we stop once we reach a maximum of magenta pixels in a frame (telling us we have passed the line).

The greatest challenge related to this section is knowing how to use the given information. Without road lines, there is no obvious answer for how to navigate so you must engineer a solution which utilizes the information to create the best possible solution.

Conclusion

Competition Performance

By the time the competition rolled around, our robot had become consistent to the point of completing the course while scoring maximum points. There was the occasional mistake where the robot would fall off the mountain or incorrectly guess a letter on a sign, however, these were rare. In the competition our robot placed 2nd overall, scoring maximum points with a time of 89 seconds.

Some Dead Ends

Mountain

Because of the increased difficulty of processing the images in this section, our filters were not as consistent as they were for other sections. By the time the images were being used by our driving algorithm there would still be some noise in the image. This resulted in the robot occasionally misreading the coordinates of the road line and sending a bad velocity command. In the case of the mountain, if a few of these bad commands stacked up, the robot was headed straight off the side of the cliff. While we did get to the point where this mistake was rare, it was at the sacrifice of speed. Our robot completed the mountain portion of the course in nearly the same amount of time it took to get through the rest of the course. The average time it took to reach the mountain section was 45 seconds, and our average course completion time was 84 seconds. Reflecting on this, had we had more time until the competition date, we would have tried to implement an imitation learning model for driving the mountain section that likely would have decreased our time greatly.

Acceleration Curves

In a quest for speed, one thing we had our eyes set on was some form of acceleration and deceleration curves. The car tips over with any sudden velocity changes so if you were able to input acceleration commands instead of velocity ones you could have generated faster speeds without tipping. This is particularly valuable in sections of long straightways which this map has. However, by the end of the journey the code had gotten so unstructured that implementing this would have been impossible without a major rewrite of the code base. With smoother acceleration, we could approach much higher top speeds and would have been able to shave off critical seconds without damaging our consistency.

Retraining The CNN

Like many teams, we planned on retraining our model to improve it with competition data at the end once we had collected enough data. Interestingly, when training on this new set we saw a decrease in performance. These problems were not the result of overfitting to competition data as we tested the new model on the generated data (discussed in the earlier section). Left confused, we decided to stick with the old model which was accurate enough, but a mystery remains as to why we were unable to improve our new network

References

Discussion with classmates was a critical piece of the project, lots of people had unique ideas that made for easier solutions. Listed below were some of the great ideas that others' shared with us:

1. **Kris:** Relative brightness scaling for the mountain region
2. **Itai:** Pre-masking to get rid of the truck for crosswalk image subtraction.
3. **Hunter:** Thoughts on a data collection script for sign
4. **Levi:** Running forward passes in the initialization of the state machine to increase speed
5. **Levi:** Pass in images as an array to improve forward pass speed
6. **Itai:** Neural Network quantization