Self-Driving Robot Simulation - Final Report

ENPH 353

Sanjana Chawla and Kritika Joshi

1 Introduction

Welcome to ENPH 353, our introduction to machine learning course! We will be participating in a competition at the end of term in which our virtual robot, Blue, will navigate around an intersection attempting to read license plates and adhering to pedestrian crossing rules.

This report will primarily cover the essentials of our strategy for the course competition, including the key algorithms we used to design our project.

There were two main aspects to consider in designing our functional self-driving robot:

1. Detection and Recognition
2. Navigation

2 Detection and Recognition

In this section we will discuss the various algorithms we considered while designing our virtual robot's detection and recognition capabilities in this simulation of an intersection. We will be focusing on the chosen implementation, which was a ROS package for Darknet's YOLO r-CNN and a simple character detection algorithm. More detail about our alternative options can be found in our logbook.

2.1 Overview

Here is a brief overview of the various algorithms and topics that were considered while designing pedestrian and license plate detection and recognition.

- Looking at different detection algorithms
  - Darkflow
  - Darknet
  - Darknet for ROS
  - Image Processing using OpenCV
- Creating datasets
  - Choosing how to generate YOLO dataset as well as a license plate dataset
  - Creating a large enough and well-represented dataset
  - Data augmentation
  - Image annotation tools for YOLO model compatibility
    - Bbox Label
    - LabelImg
    - Yolo-New-Annotation Tool
- Training models on custom dataset
  - Configuring how to have Gazebo nodes communicate with Python 3-based Neural Networks
  - Modifying dataset and re-training with varying parameters to find optimal model to detect license plates and pedestrians in real-time
  - Simplifying YOLO algorithm to process faster due to zero GPU in laptop
- Developing algorithms for intended behaviour upon class recognition
  - Stopping the car and waiting for pedestrian to cross upon pedestrian detection
  - Feeding license plate picture into character detection CNN to read license plates upon license plate detection

2.2 Detection Algorithms

Initially, we had chosen to use Darkflow, a version of YOLO which had been translated over the Tensorflow framework, but after consultation with the course professor, we decided to work with darknet_ros, which allowed abstraction of converting the Python 3-based YOLO into something Gazebo (which runs on Python 2.7) could interpret. By using YOLO, we intended to let our neural network learn and recognize our main classes: pedestrians and license plates so that when the robot drove in real-time, it would be able to apply a model we train on our data to put bounding boxes on the main class items. Using these bounding boxes, we would be able to set up a node that subscribed to this information and, depending on whether the bounding box was identifying a license plate or a pedestrian, could jump to the correct class to score our robot points.

One thing we tried to be careful with was choosing which YOLO version to use. Since neither of our laptops had GPU capabilities and relied solely on our limited CPU, we used a tiny-yolo framework which took up significantly less processing power than a normal framework.

We could have just as easily chosen to mask the images using image processing, the simplest detection method based on our simplified competition environment, however we chose YOLO because it was a more elegant solution that would work even in a simulation where image masking was not possible. This means that if we created a different traffic simulation world with more complicated objects (such as cars with different colours and more complicated pedestrians) ours would be a more robust solution. Below you can find a diagram abstracting the process we used in working with YOLO.

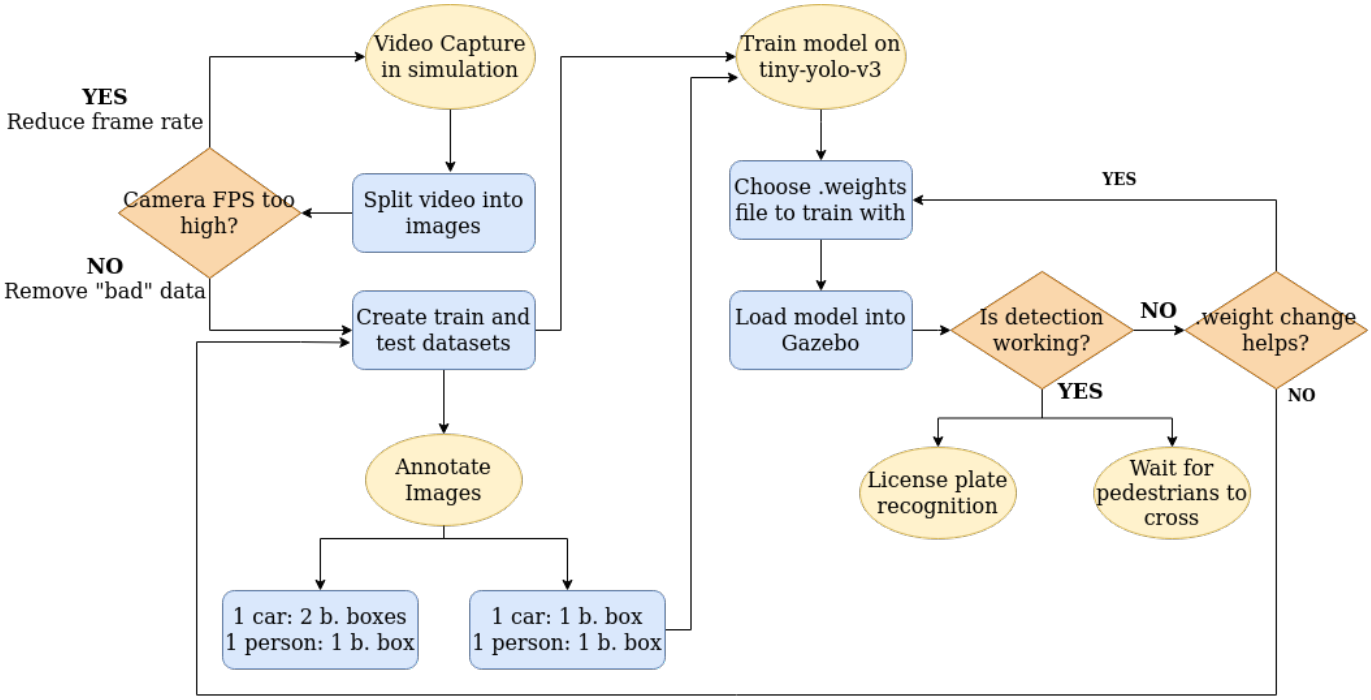


Figure 1: Detection state map.

2.3 Dataset Creation

In order to use YOLO, we would have to train it on our custom dataset, inputting our classes (pedestrian and license plates), as well tweaking parameters of the model based on the fact that we only have two classes. Dataset creation started by first taking a video of the robot driving around in the simulation. Once the video was saved, I created a Jupyter Notebook to take this video and split it up, frame-by-frame. I then analyzed the frames to determine if they were too similar, creating a lot of repetitive data. If I believed this to be the case, I went into the robot.xacro file and modified the camera link's update rate to be something smaller. Another thing I kept in mind was ensuring I had a similar amount of images that were "negative" for class detection (meaning neither of the classes were in the image) as opposed to "positive" to ensure the model was able to learn more accurately.

Once we had sets of training and test data we were comfortable with, we are ready to move on to image annotation! This is so that we tell our model where the expected bounding boxes should be per class. Similar to previous lab work, we can say that our images are our "X data" and our annotations are our "Y data". Below you can see my final method of annotation for images:

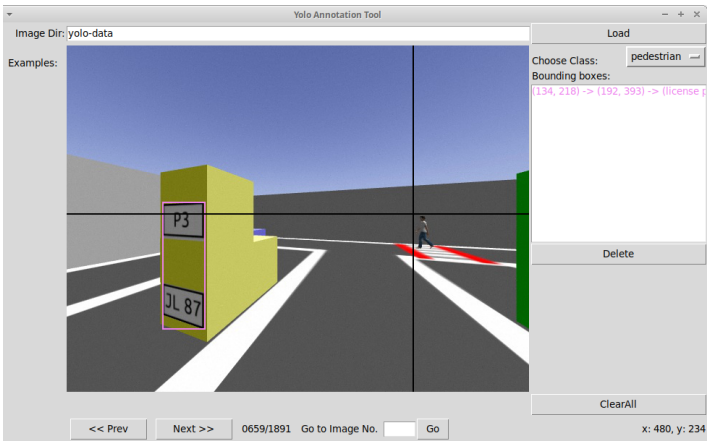


Figure 2: YOLO Annotation Tool.

2.4 Training YOLO on a Custom Dataset

The first time I annotated the images, I chose a very small sample size of about 64 images and chose to annotate such that there were 2 license plates per car and the normal one bounding box per pedestrian. However, after training the model on this data, and ensuring a high enough threshold to ensure bounding box criteria was selective enough to get useful results, we found that the bounding boxes were overall too large and still not selective enough.

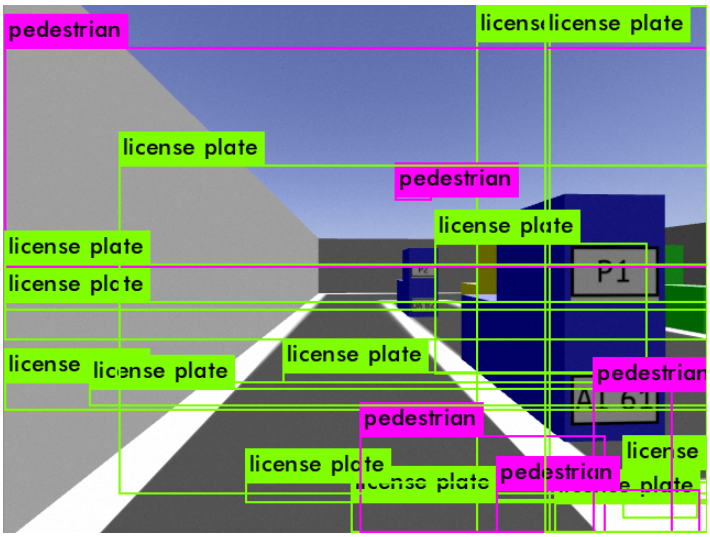


Figure 3: Training tiny-yolo-v2 on a small sample size with 2 bounding boxes per license plate.

In order to combat that, I increased the dataset to about 1900 images and manually annotated them all. When doing this, I chose to, this time, annotate one car to have a single bounding box for the 2 license plates. After running the model with this new data and experimenting with the different .weights files that were generated (a .weights file captured the model's training, and stored a checkpoint file once every 100 images that were processed) I got better results in real-time, but still, they weren't great.

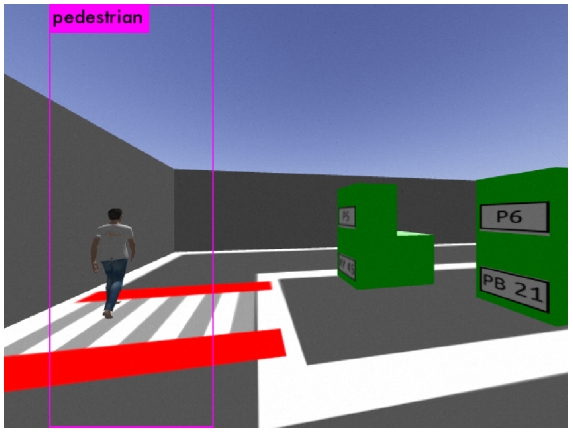
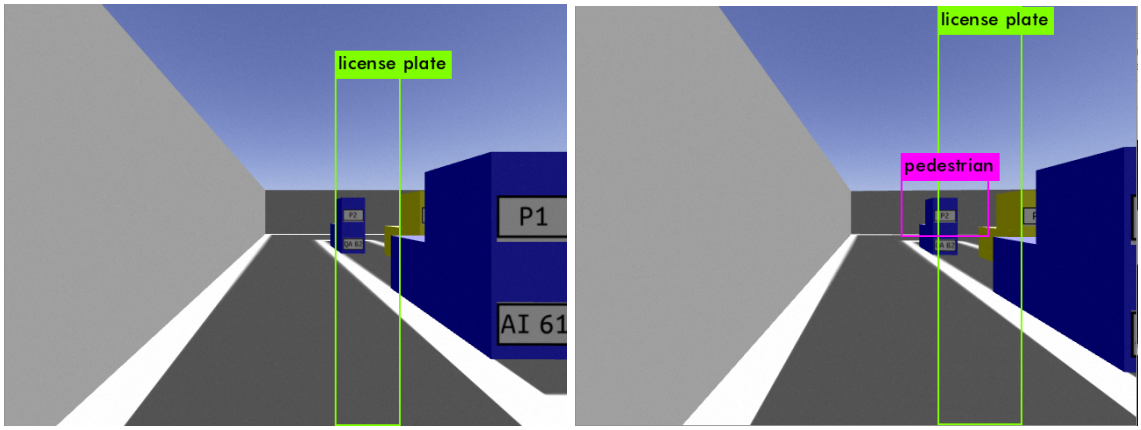


Figure 4: YOLO detection at intersection.

My final attempt involved going back through the training dataset, removing more "bad" data (such as when a pedestrian was right in front of a license plate and was only half-visible on the screen). Then I went back into the annotations and removed partially visible license plates and pedestrian bounding boxes for when the pedestrian was far away. This was so that the r-CNN would have big enough bounding boxes to be able to learn something useful from that training data. Below are the results I got after this iteration of training:



(a) At start-line.

(b) Driving towards intersection.

Figure 5: Final model characteristics.

2.5 Pedestrian Detection

For this section, we essentially needed to create a class that could subscribe to the darknet_ros bounding box node, which would give coordinates of the bounding box and which class it corresponds to. From there, if any bounding boxes correspond to the pedestrian class, the recognition class would stop and wait 2 seconds for the pedestrian to cross and then continue driving. On the right is a state map showing the decision making for this section:

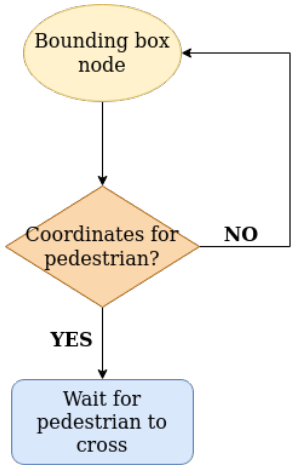
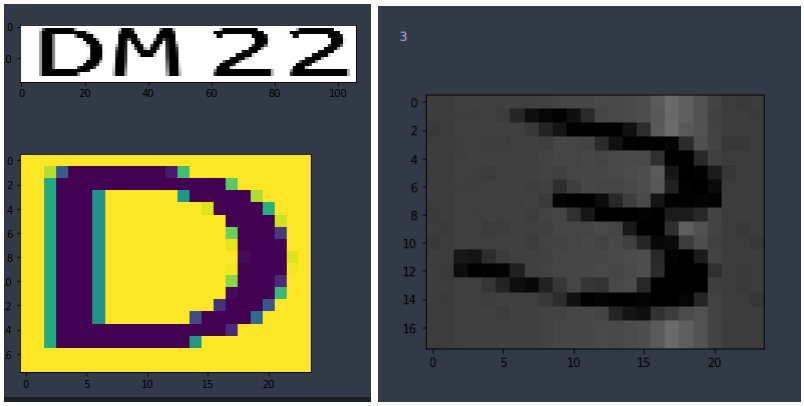


Figure 6: Pedestrian recognition.

2.6 License Plate Detection

For license plate detection, we used an approach similar to what we did for our license plate detection lab, with a few differences. We have 36 classes (26 for letters and 10 for numbers) and we want to, similar to in pedestrian recognition, use the bounding boxes to tell us where the license plates were. Here, however, once we know where the license plates are, we will crop the image into individual letters (using hard-coded dimensions for successful cropping) and run them through our character recognition model. Since it's quite possible that the given images will have characters at different angles, to make our model more robust I augmented data so

that we have characters tilted at different angles in our training set. As you can see below, despite this our model did a pretty good job of detecting this characters. A confusion matrix using a small test test is included in our logbook.



(a) Creating training data. (b) Successfully identifying a tilted 3.

Figure 7: Final character recognition model characteristics.

2.7 Results and Discussion

After extensive training on tiny-yolo-v2, we found that despite expanding our dataset, using different .weights files (increasing the validation accuracy on the model) and changing annotation criteria, our model was still laggy and not as accurate as we would have expected based on similar GitHub projects that we looked at for inspiration. We think that a few ways we could improve the accuracy on our model (that we will try after finals) are:

- Add a third class for pedestrian crossing
  - If both pedestrian b. box AND pedestrian crossing b. box detected, then only stop the car.
- Go through annotations and further restrict criteria for training model
  - Continue to monitor if this makes any changes in how the model performs.
- Run model on a laptop with GPU
  - To see if GPU makes a significant difference in processing accuracy and/or speed.
- Simplify YOLO to increase speed of real-time processing
  - Simplifying YOLO model is more viable than running on a laptop with GPU and may let us more easily identify other issues in how this was structured.

Due to time constraints we were not able to integrate our license plate and pedestrian detection decisions with our YOLO model, however we have most of the base-code figured out for the pedestrian detection and have created a robust model for character detection. We will continue to work on this project after final exams and will hopefully be able to tell you more about our model once integration is completed.

3 Navigation

3.1 Overview

We decided to use Q-Learning to enable the robot to follow the road and keep within the tracks. The goal is to keep the robot withing the white line boundaries, while also ensuring that a license plate box is within it's vision. While a common implementation to get the state array of the robot is through it's camera feed, I instead decided to subscribe to the position node to find the robot's location in the gazebo environment. Then proceeded to map its position to a masked image of the track map to determine the state array.

From the map, the state was determined through the three key pieces of information we could gather from its location. We wanted to know which section of the map the robot is in, where on the road the robot is, and which direction it is going. Our goal is for the robot to stay on the center of the road, while constantly moving forward.

3.2 Implementation

An overview of the process we used to assign the state of the robot is shown in the flow chart below. As seen, we begin by subscribing to get the ModelState of the robot. From here, we have information of the robot's x,y,z positions, as well as its orientation angle, and its twist. Only the x and y position and the angle is used in our implementation.

After we obtain the x and y position of the robot, we want to know what that means on the map of the robot's environment. To do this, I transformed those coordinates to a point on a masked image of the map. That masked image can then tell us more about the state of the robot. Specifically, we can know where on the road the robot is by assigning colors to each section of the road. We also want to know which section on the map the robot is located, and lastly its angle is also used in determining the state array.

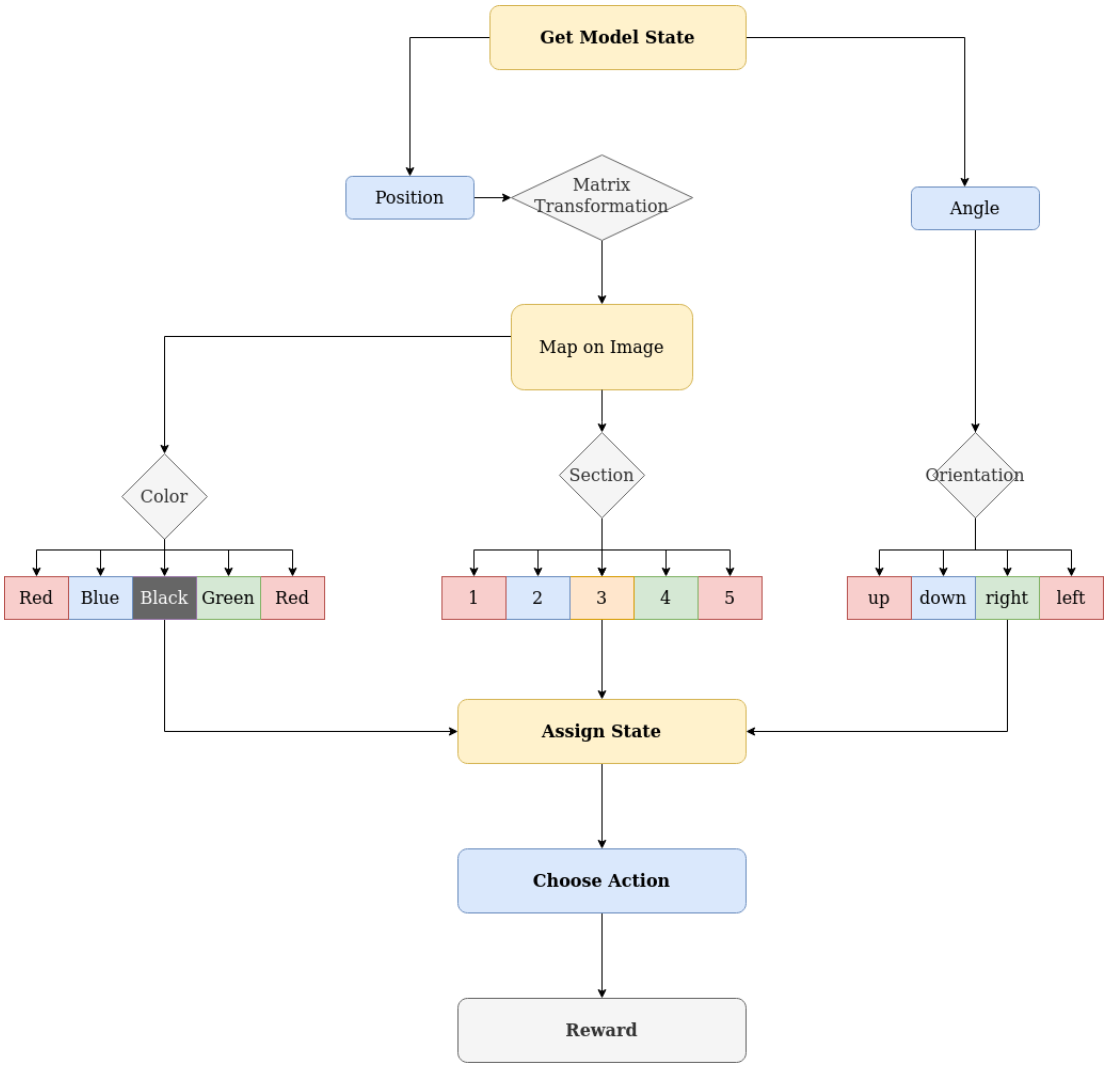


Figure 8: Navigation

3.3 Obtaining State

By subscribing to the Model State, I could obtain the x and y coordinates of the robot. The next step upon that is to transform our coordinates into a point on the image of the map, and then have that point tell us the other information we

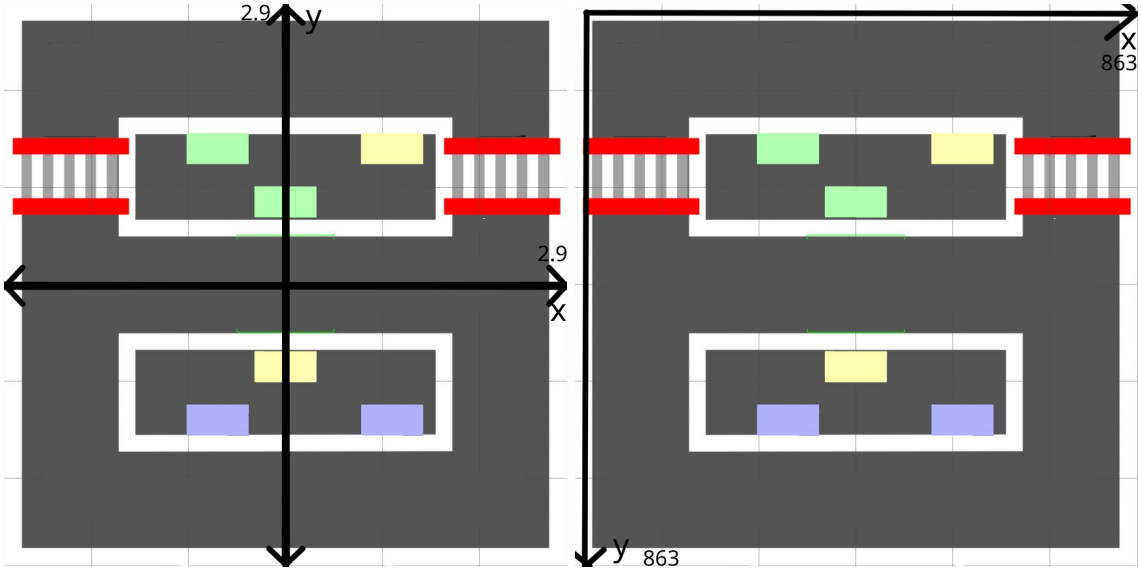
need to know through masking and image processing. Specifically, the information we use to determine the state array is:

- Color: Tells us robot's position on road (too left, center, too right)
- Section of Map: Tells us robot's position on Map
- Orientation: Tells us the direction the robot is facing

The orientation is given from the angle obtained from the ModelState, and the Color and Section are obtained by transforming the coordinates to locating pixel on a masked image of the map. I will first talk about the Transformation, and then about the Masking.

3.3.1 Transformation Matrix

Using a transformation matrix, I mapped the x and y coordinates given to us from its model state to its position on an image of the map. Ros gives its coordinates with reference to the center of the map, however when using OpenCV to perform image processing, it counts the pixels starting from the upper left corner. Also, the shape of the image in openCV is 863×863 pixels, and the map in the environment's size is 2.9×2.9



(a) Ros Coordinate System (b) OpenCV Coordinate System.

Figure 9: Transforming Coordinates.

Therefore a matrix transformation is needed to map the coordinates over to the image. this is done by:

$$\begin{pmatrix} x_{img_i} \\ y_{img_i} \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \frac{2.9 \times 863}{5.8} \tag{1}$$

Then, because the coordinates give the centroid of the robot, obtaining the location of the nose would give us a more accurate representation of the state, so:

$$x_{img} = x_{img_i} \times 4.2 \sin(angle) \tag{2}$$

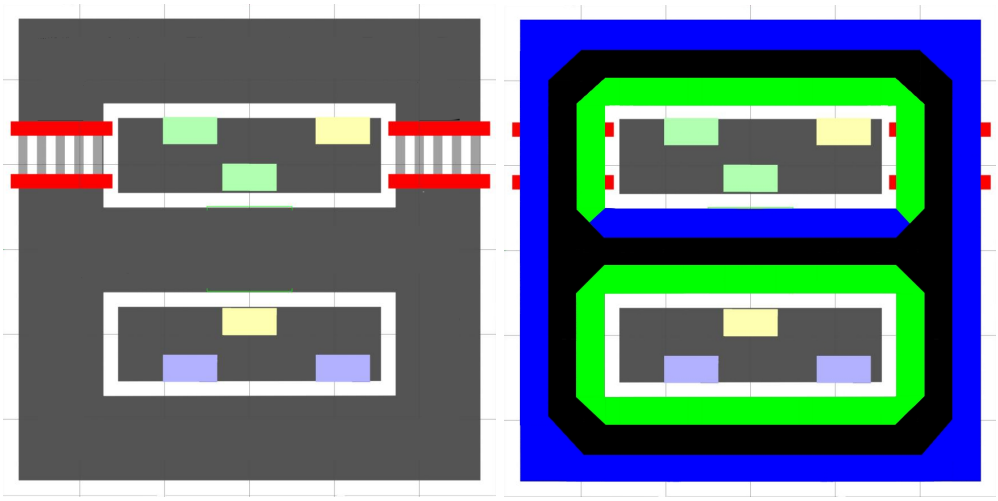
$$y_{img} = y_{img_i} \times 4.2 \cos(angle) \tag{3}$$

Where x and y are the original coordinates given by the RosServiceCall and x_{img} and y_{img} are the points on the image those coordinates map to. This maps the robot's position from its model state to one pixel on the image of the map

3.3.2 Masking and Map processing

From the map, the state was determined through the three key pieces of information we could gather from its location. We wanted the robot to stay on the center of the road while progressing forward, so the first step was to mask our image to give use where on the road the robot was. To get this information, I masked the image using a color coding scheme to tell the robot whether it was too far left, too far right, or completely off the road.

Beginning with an image of the original map, and imagining the robot at its starting position and moving forward, I masked this image by sectioning off the road before the white railings into three different categories. The blue indicating the left side, the green indicating the right, and black being the center of the road.



(a) Original Map (b) Initial Mask.

Figure 10: Map Masking to section the road.

Once we masked our image, we quickly realized that the color itself would not tell us enough about the state of the robot. As whether it is "too right" or "too left" is highly dependent on the section of the map, as well as its orientation. For example, if the robot started in its initial position, when going forward "blue" would mean "too left", however, in that same position when going backward "blue" would mean "too right" - and you would need a different action depending on this state.

This is where sectioning the map and obtaining the orientation of the robot can help us get more information to which state the color should correspond to. I sectioned the Map into five different categories, based on the possible motions of the robot. Then, along with the orientation of the robot, we can now determine our state array with more accuracy. For example, say the robot is in section 1, and its orientation is that its facing to the left and moving forward. We now know that if the color is blue, it means the robot is too far to the left of the road, and if the color is green the robot is too far to the right of the road.

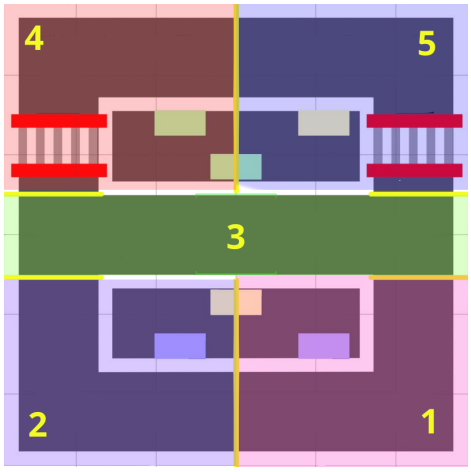


Figure 11: Sectioned Map

While training with this image however, the robot would find loopholes to where the unmasked parts of the image were. To combat this problem, I introduced a new state masked with red, which means the robot is either completely off the tracks, or close to being off the tracks. The green and the blue now only serve as indicators of its state before it turns to red and it starts to lose points.

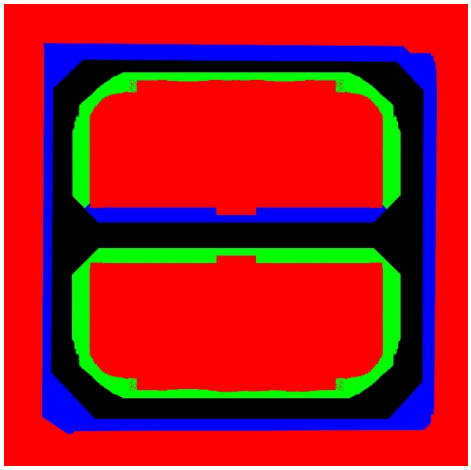


Figure 12: Final Masked Map

Then, finally we could process our state into a single 5 entry state array, giving where the robot was on the road.

off Road	Too Left	Center	Too Right
[1 0 0 0 1]	[0 1 0 0 0]	[0 0 1 0 0]	[0 0 0 1 0]

When doing this again however, I want to change the state array to:

off Road Left	Too Left	Center	Too Right	Off Road Right
[1 0 0 0 0]	[0 1 0 0 0]	[0 0 1 0 0]	[0 0 0 1 0]	[0 0 0 0 1]

Adding this change might make the robot's learning process more efficient, but adding it in would require going back to change the conditions choosing its state, once we get more time to do so.

3.4 Action and Reward system

This section required the most testing while training to see how the robot learnt to do its intended goal. Following Lab06, I decided to make the highest penalty when the robot is completely off the road, or in the red state. However, since the

state array is based off location and not camera feed, if the robot increments only by twisting in z (turning right or left) by very small amounts, it would technically stay on the center of the road and be able to accumulate points without moving right or left. To combat this problem,I made it so even when it turns, it still has a small velocity in the +x direction - this way it wouldn't be able to remain stagnant and only rotate incrementally to gain points. I also initially made the reward negative for turning, but larger than if it were to be on a red state, and the reward very high if it goes straight. The robot quickly learnt it only wants to be in the center of the road and drive forward, however, it was extremely resistant to turning when it hit corners and would get stuck at the wall before able to complete a turn.
Then, I decided that it should gain points for turning if it was able to get itself out of a red state. The final decision on the reward system is:

Turn Left	Go straight	Turn Right	Red State
+35	+50	+35	-200

3.5 Results and Discussion

After about five hours of training, the farthest the robot was able to get to during a training session was section 4 of the map. However, this only happened rarely and the most common path the robot would take is shown in the included video, where it drives up to the corner and then gets stuck. I think the reason for this is that even while turning, the robot has to move forward as well (this is to make it impossible for the robot to only accumulate points while shifting left and right and not risking a change in state) but a drawback is that it easily gets stuck in corners when it doesn't choose to turn early enough. Re-masking the image might help to give the robot a smoother turning path, and also increasing the reward for turning. Another major drawback of this implementation as a whole is that it requires a highly accurate mask on the image revealing the robot's state, since it is determined by one single pixel that the coordinates map to. I used the image processing software, Krita, to mask the image which gave some level of precision, but more time needs to go into deciding a more appropriate mask to make based on knowing where the robot is likely to get stuck, and the path we want him to take.

4 Conclusion

Due to time constraints we were not able to integrate our navigation and detection portions of the robot. We have attached individual videos of the components working to demonstrate our progress for the competition. We will continue to develop this project after final exams and will tell you more about our project once integration is completed.