ENPH 353 Final Report

Tyler Wilson and Peter Goubarev

December 18, 2022

Contents

1 Introduction

1.1 Project Background

ENPH 353 is a project-based course that teaches students how to use modern computing tools, with a focus on computer vision and machine learning. Some of the tools discussed include the Robot Operating System (ROS) and Gazebo simulator, OpenCV library, the SIFT algorithm, neural networks, and reinforcement learning. The final project for the course is a competition, in which teams of students train a robot to navigate a simulated world (depicted in figure 1), resembling the task of creating a self-driving car. The goals for the robot are to navigate the world reliably, locate parked cars (figure 2) and read their license plates, and avoid obstacles such as pedestrians and other vehicles. Points would be awarded for returning the correct license plates and completing the outer loop of the course while points would be deducted for veering off the road or colliding with obstacles.

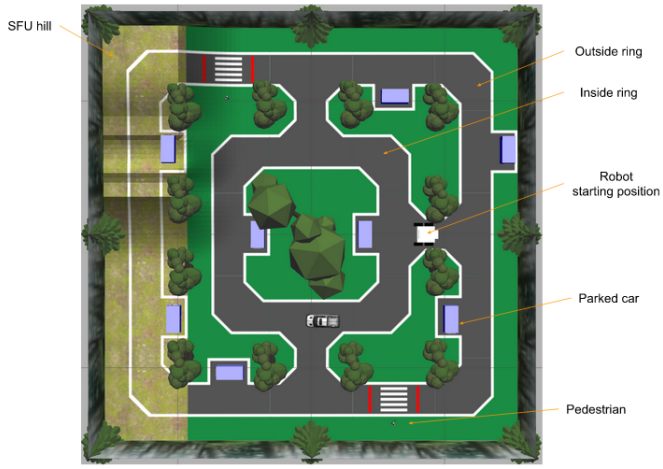


Figure 1: Simulation world and features shown in above view

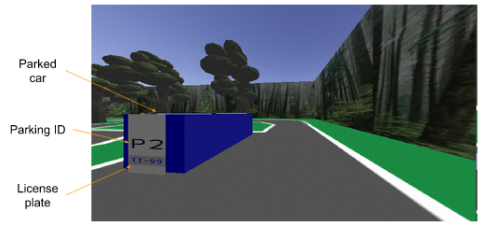


Figure 2: Simulation view of a parked car

1.2 Goals

Our team's goal was to complete all the challenges laid out in the competition, giving us the ability to get a perfect score. This included us being able to:

- Reliably complete an outer loop of the course
- Complete a loop of the inside without colliding with the car
- Identify and avoid pedestrians
- Reliably identify and return the license plates on all the vehicles

2 Methodology

2.1 Software Architecture

Figure 3 depicts our software architecture. We used a single repository for client-side code and trained our neural networks on Google Colab. The client-side code contained three nodes and

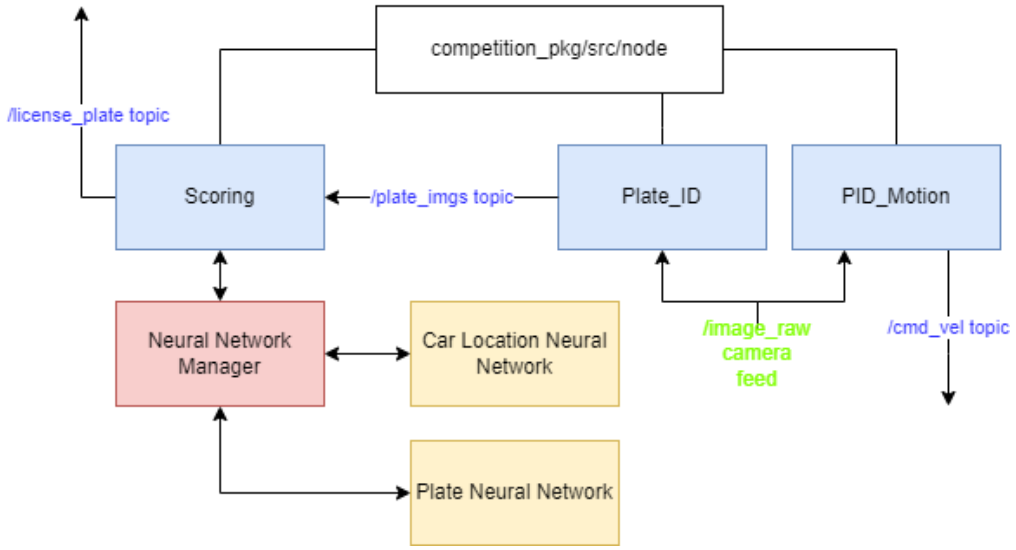


Figure 3: Software architecture

several helper classes:

- `PID_Motion` node: contained the driving algorithm and state machine
- `Plate_ID` node: contained the image processing algorithm, which obtained a nicely cropped and transformed license plate, as well as an image of the stall number. These images were stitched together and published to a ROS topic we created called `/plate_imgs`.
- `Scoring` node: started and stopped timer, passed the images from the `/plate_imgs` topic to the neural networks, and submitted plate predictions to the `/license_plate` topic.
- Neural network helper: several classes to pass data to our neural networks, obtain predictions, and translate them into a format that can be published by the scoring node.

2.2 Driving and Course Navigation

Our robot made use of multiple PID algorithms to navigate the different parts of the course. The `PID_motion` node contained a state machine which was capable of handling different components of the course.

2.2.1 Navigation of the Outer Loop

The robot was hard-coded to turn left off the start of the course and then switch over to a PID algorithm to drive the outside loop. The key challenges we overcame included navigating the hill and remaining inside the lines on the outer loop without turning into the inner loop or a parked car. Our robot followed the white road lines, because unlike the gray road surface, they were present everywhere including on the hill. We applied an HSV mask on our raw camera image in order to isolate the lines (figure 4), making use of the fact that their low saturation value was distinct.

However, given that there were two lines, this introduced an issue of determining which line to follow. To solve this, we developed three different PID methods (follow the left line, right line, or the average of the two lines) and wrote an algorithm to decide which PID method to use. The three methods followed a very similar structure in their implementation. We looked at the bottom 20% of the masked image and partitioned it into two halves (left and right). The centroid of each half was computed, giving a center for the left and right lines. The average of these two centers was used to give a third center value of both lines combined. To visualize this, we created a troubleshooting screen and plotted the relevant centroids in figure 5 in appendix A

In our PID algorithm, the robot's ideal state is to be driving straight and to be located exactly half-way between the lines. When driving using both lines, the PID error is proportional to how far apart the combined centroid of both lines and the middle of the screen are. When driving using one line, the algorithm attempts to minimize the error between where the line's centroid is and where the line *should* be. We determined where it should be by writing a function that considers the average perspective slope of the lines on a straightaway and the width of the road. The robot's turn speed is then computed with the following function:

```
def PIDcontrol(self, error, max_turn, frame_width):
    return -max_turn*error/(0.5*frame_width)
```

During testing, we noticed that the robot had some difficulty navigating the corners. To compensate for this, we implemented a ramp function that set the robot's linear speed based on the magnitude of the angular speed we published;

```
move.linear.x = max(maxSpeed - turnReduction*abs(move.angular.z),-0.05)
```

The PID algorithms all use the bottom 20% of the screen to navigate, where the lines are most visible. To determine which PID algorithm to use (left, right, or both lines), we looked at the next 10% of the screen above that and found the lines' centroids there. Then, for each line, we found the difference in their centroid's x position between the first and second partitions of the screen, and used this to estimate the line's angle. Intuitively, if the robot is driving straight, perspective makes each line be angled slightly towards the centre of the screen and we follow the average of both lines. If the left line is angled sharply left but the right one points towards the middle, there might be an intersection or parked car on the left side, so the robot navigates using the right line. Vice versa when only the right line is angled. This technique allowed us to successfully navigate the outer loop without steering into any parked cars or accidentally entering the inner loop.

2.2.2 Pedestrian Detection

To handle the pedestrians, we stopped at the red line in front of the crosswalk and waited until we detected the pedestrian crossing. To do this, we created an HSV mask that isolated the specific red colour of the stop line (figure 7) and computed the vertical centroid of the line on the screen. Once the centroid was low enough on the screen, the robot stopped and entered the crosswalk state of our state machine. In this state, the robot looked at a small square of pixels on the screen, at the height of the pedestrian's body (figure 8), and recorded the pixels' average darkness value. Once the pixel values changed by a sufficient amount, it would indicate to our robot that a pedestrian

was detected and it would exit the crosswalk state.

To ensure that the robot wouldn't stop at the red line at the far end of the crosswalk, we created a boolean value, `self.in_crosswalk`, which was set to true when the robot detected a crosswalk. When set to true, the robot does not search for red lines or stop at them, although the red line at the far end of the crosswalk is usually still visible. The boolean is set to false when that line's centroid moves far enough down the screen, indicating that we have exited the crosswalk.

```
if red_center[1] < 300:
    self.in_crosswalk = False
```

2.2.3 Navigation to the Inside Loop and Vehicle Detection

Because our team produced a reliable PID algorithm for the outer loop early on, we tackled the challenge of navigating the inner loop. We explored many methods of entering the inner loop from the outer one, and found that the most reliable way was to use the centroid of the red crosswalk stop lines again.

As our robot navigated the outer loop, it recorded the number of times it stopped for a pedestrian. We used this variable to easily control how many times our robot navigated the outer loop, and to determine when to navigate to the inner loop. Once it had stopped at enough crosswalks, the robot entered a state that made it treat the next crosswalk differently: it would drive towards it until the red line's centroid reached a low-enough position on the screen, and if the centroid was within a particular horizontal range of the screen, the state machine would instruct the robot to start moving towards the inner loop. The following code was used to accomplish this:

```
if self.pedestrian_count == 4 and self.in_crosswalk == False:
    if red_center[1] > 375 and abs(red_center[0] - image_width/2) < 200:
        self.state = "turn inside"
```

Once instructed to move inwards, the robot would follow the left line until the centroid of the red stop line disappeared, at which point the robot would stop moving. This would typically correspond to a 90 degree turn, with the robot facing the inner loop directly and with a clear view of the NPC truck. We detected the truck the same way as the pedestrian, by looking at a small square of pixels on the screen and waiting until their average value changed. Once the truck was detected, the robot would wait a short time before entering the inner loop and then follow the left line before entering the "inner loop" PID state, to avoid having a collision, and to ensure the robot would traverse the loop in the desired direction.

2.2.4 Navigation of the Inner Loop

Once we entered the inner loop, we applied an HSV mask to isolate the gray colour of the road. We computed the centroid of the road in the bottom 20% of the screen. Like in the outer loop, we varied the robot's linear speed based on our desired angular speed so that the robot did not overshoot the road on the turns. This method had the slight issue of turning into the parked cars in the center. To avoid this, we created an additional HSV mask for the blue of the cars (figure 9),

determined the blue centroid, and computed a set of angular PID values based off those centroids (figure 10). We took the PID outputs from following the road and subtracted the PID outputs from following the blue cars such that our robot would steer away from the parked cars, eliminating this issue.

The speed of our robot was tuned such that it could complete the inner loop numerous times without colliding with the NPC truck, increasing our plate-reading reliability. We determined that 30 seconds of circling the inside loop was enough to ensure a high probability of predicting the correct plates and so ended our run 30 seconds after entering the inside loop.

2.3 License Plate Detection

2.3.1 Recognition and Perspective Transform

To detect the license plates, we ran an HSV filter on the image from the camera, isolating the white colour above and below the actual license plate. The filtered image ideally contained a large white box wherever there was a parked car. We then found the contours of this masked image and picked out the largest contour, ideally corresponding to the large white box above the plate. We would then use a polygon approximation to find the corners of this largest contour. For each camera frame, we would compute the area of the largest contour and find its corners. We would only choose to analyze the image further if the area met a certain threshold value and the area of the current frame was larger than the area of the previous frame. We only considered contours that had exactly 4 corners as any contour that didn't would not correspond to the parking stall and was of no interest to us.

For the largest contours that met our criteria, we performed a perspective transform on the image given the location of the corners and produced a transformed image of the plate. We then stitched this image to an image of the stall number and published them to the `/plate_imgs` topic. See Appendix B for a visual description of our image processing method, as well as a code walkthrough.

2.3.2 License Plate Neural Network

We identified the license plate characters and stall numbers using separate neural networks. The license plate network was trained with 2000 plates in total, with a 90%/10% mix of generated/simulation data respectively.

To generate data with high fidelity relative to simulation images, we used the license plate generation script from lab 5 and applied a series of distortions. The distortion amount for each parameter was randomized for each plate. The following effects were used:

- Small X and Y displacement (resulting in edges of plate getting cut off)
- Gaussian blur
- Perspective transform
- Image darkening (by decreasing the brightness value after image was converted to HSV)
- Salt and pepper noise
- Downsampling

See Appendix C.1 for examples of generated plates and Appendix C.2 for a description of the code used to apply all of these effects.

We decided on the network architecture by performing controlled experiments where we modified one parameter at a time and re-trained the network from scratch. The parameters considered were: the learning rate, mini-batch size, the number of hidden layers, number of filters in the hidden layers, the number of dropout layers, and training dataset size. We found that the only parameters that improved training performance were the number of layers, number of filters per layer, and training dataset size. The license plate network eventually had four 2D-convolution layers, with 32, 128, 256, and 512 filters in sequence, and in between each convolution layer was a 2D MaxPooling layer. The other parameters were kept at their default values (learning rate of 1E-4, batch size of 16, one drop-out layer). See Appendix C.3 for the model summary.

We trained the license plate network as follows:

- First phase: 10 epochs, 1000 generated plates (4000 characters)
- Second phase: 10 epochs, 800 newly generated plates, 200 simulation plates (4000 characters)
- Third phase: 5 epochs, 200 generated plates that focused on characters with high losses (5, 6, 8, O, Q, Z, J).

The network achieved high training and validation accuracies of over 85% and low losses of less than 0.7 after only 10 epochs, but performed poorly when asked to predict simulated plates it had never seen. The second training phase increased its prediction performance significantly. The third training phase made the network predict the high-loss characters accurately, but perform worse overall, so we used the model from the second phase. While training the networks, we frequently plotted the confusion matrix to identify the characters missed often. Appendix C.4 contains graphs of training and validation losses, as well as the confusion matrix, from the first and second training rounds.

The network used to identify the stall number was trained with 100 images from the simulation. Some of the images we provided contained noise, the sky, or just gray or black, and we trained the network to associate the 0th index of the one-hot vector with such images, in order to control the model's behaviour when image pre-processing failed. This network had the same architecture as the license plate network, except that we removed the 512-filter convolution layer and subsequent MaxPooling layer.

2.3.3 Plate Prediction

When passed an image of a plate, we would split it into 5 separate images (one for each character and for the plate ID). For the first two characters we returned the maximum value of the character entries of the one-hot array and for the third and fourth characters we would return the maximum value of the one-hot array for just the number values. This allowed us to avoid mixing up numbers and letters while only needing to train one neural network. To guess the plate IDs we used a different neural network to predict numbers as the text and colouring was different than that of the license plates.

To optimize our predictions for the plate values, we created a variable called `self.plate_recordings` that stored the value of every guess we made for each letter of each plate. Every time we saw a new plate we would update this variable and take the most frequently occurring character for each position of the plate using the following method;

```
def getMostFrequent(self, input_list):
    return max(set(input_list), key = input_list.count)
```

This ensured that the value we submitted for the plate was the best one based on the data from multiple images. To ensure further reliability, we decided to traverse the outer loop twice before entering the inner loop to give us more images of each plate.

3 Conclusion

3.1 Competition Performance

By the end of our development, our robot was able to able to complete full loops of the course with very high reliability and return license plates with fairly high precision. This led to us executing some perfect runs during our pre-competition testing. During our competition run, we were able drive smoothly through the course and return correct values for all of the plates. However, due to some issues with our plate ID neural network not being able to filter out noise, we overwrote two of our correct guesses for the outside plates, giving us a final score of 45, corresponding to 6/8 plates returned correctly, placing us 3rd in the competition.

3.2 Explored Methods

3.2.1 RGB Filter to Detect Lines

The development of our driving algorithms went through many iterations. Initially, we took a grayscale image and used a binary threshold tuned to pick out the white lines on the road. This worked well for the road, but picked up a lot of noise from the colour of the ground on the hill. A higher threshold value was able to remove a large portion of this noise, but meant that we were not able to see the lines on the sloped part of the hill, causing our robot to go off course.

One fix we implemented was to use only the blue channel of the colour image and use a binary threshold on the same channel to pick out the white lines. Because most of the noise we picked up from the hill were contained in the red and green channels, this allowed us to better filter out noise and use a lower threshold value. This method allowed us to begin completing loops of the course, but our robot still had trouble staying within the lines on the hill. Through discussion with our classmate, Alex Koen, we learned about HSV filters which can be used to detect white by picking out colour values with a very low saturation, leaving only white. An HSV filter allowed us to almost completely filter out the noise, letting us use a far lower threshold value and so see the lines on the sloped parts of the hill. After this change, our robot became reliable on the hill.

3.2.2 SIFT License Plate Detection

Before settling on the current plate detection technique (described in Section 2.3.1), we used SIFT. Our methodology was to use SIFT to locate the large 'P' on the car and selectively choose images to analyze based on the number of keypoints that matched with the template. Given the image with the highest number of keypoints, we would crop the image around the 'P' to exclude the background, run an HSV filter on the white above and below the license plate and obtain the longest contours which we would use to generate Hough lines. Finally, we would use the endpoints of the longest horizontal Hough lines to find the corners and create a matrix for perspective transform. We replaced this method because it performed poorly and was computationally slow. For example, the road often made it into the cropped image and so became the bottom-most contour detected, and at other times, the Hough lines appeared had breaks in them, leading to skewed perspective transforms which we found unusable.

3.3 Further Development

Due to the time limitations in this project, there were some improvements to our robot that we identified but were not able to implement. We lost points in the competition because the parking stall neural network was not trained enough, resulting in it misbehaving when given noisy or poorly transformed images. We identified all of the license plates correctly, but the network overwrote two of our submissions after driving around the loop several more times and seeing poor images. While we trained our network on such images, it did not always identify them correctly. Given more time, we would have trained the network on a much larger dataset, including poor images that we did not consider, such as ones that contain part of the 'P' on the back of the car.

Another solution to this issue would be to filter out noisy and blank images before they're passed to the neural network by detecting the number of contours on the image and only qualifying images with enough contours on them.

Another area that we would've liked to explore was making better use of contours in our driving algorithm. Our driving method was implemented entirely using centroids of images. This worked quite well for us, however, making use of contours may have allowed our steering to become more robust, allowing us to not having to decide which line to follow and rather have one method that takes into account when there's a parked car off to the side. Given that our implemented driving method worked quite well, we decided not to optimize it using contours but rather focus our remaining time on developing other aspects of the competition.

A PID State Machine Visuals and Methods

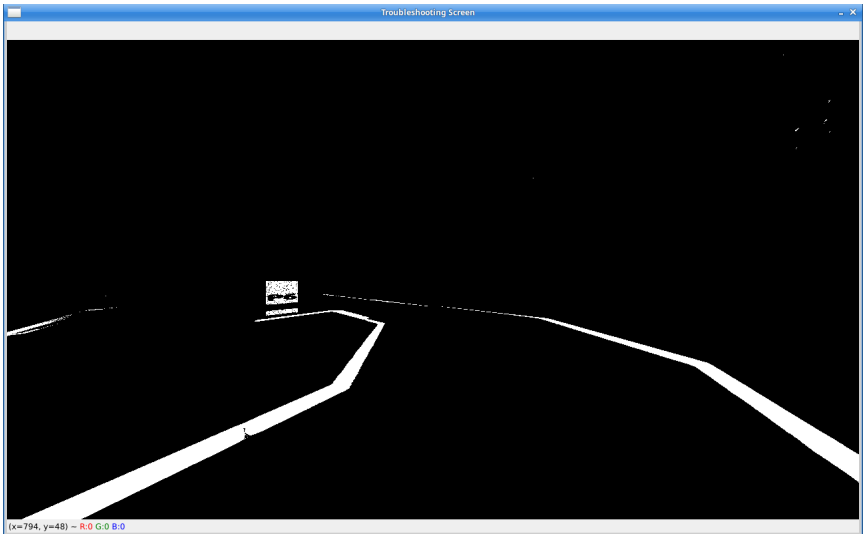


Figure 4: Output of appying the HSV mask to pick out the white lines

The HSV filter in figure 4 was computed using

```
lower = np.array([0,0,100])
upper = np.array([0,0,255])
line_mask = cv2.inRange(hsv, lower, upper)
```

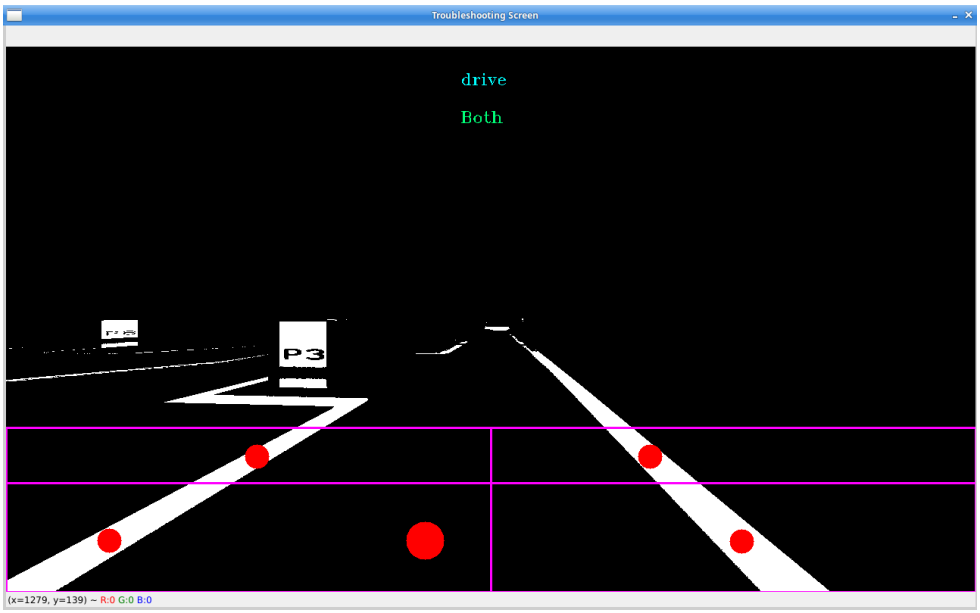
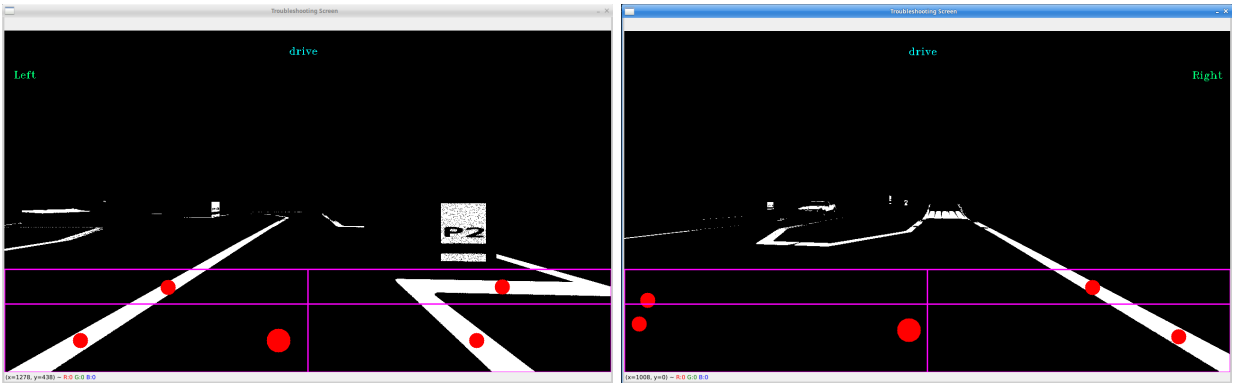


Figure 5: Image showing the troubleshooting screen in the "drive" state. The current state is shown at the top in blue. The line the robot is deciding to follow is shown at the top in green. The purple rectangles depict the partitions used to determine the top left, top right, bottom left, and bottom right centroids which are shown as small red dots. The large red dot is the average of the two lower centroids.



(a) Driving using the left line (b) Driving using the right line

Figure 6: Display of the centroids and troubleshooting screen in the cases when the robot would switch to following the left line (a) and the right line (b)

The robot would determine which line to follow using the following code:

```
# determine which line to follow and drive off that line
if delta_xL < turn_diff and delta_xR > turn_diff:
    # drive right
    move.angular.z = self.PIDcontrol(centersR[0]-eCR[0],angMax,image_width/2)
elif delta_xR < turn_diff and delta_xL > turn_diff:
```

```
    # drive left
    move.angular.z = self.PIDcontrol(centersL[0]-eCL[0],angMax,image_width/2)
else:
    # drive with both
    move.angular.z = self.PIDcontrol(centersC[0]-image_width/2,angMax,image_width)
```

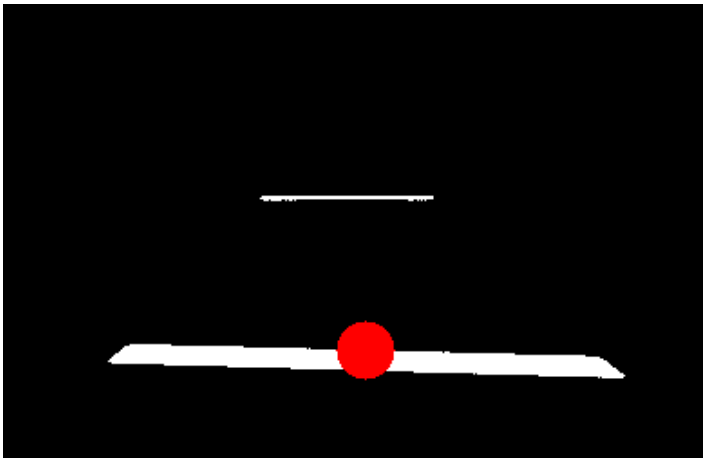


Figure 7: HSV mask and centroid for the red lines on the crosswalk

The HSV filter in figure 7 was computed using

```
red_lower = np.array([0,200,0])
red_upper = np.array([6,255,255])
red_mask = cv2.inRange(hsv,red_lower,red_upper)
```

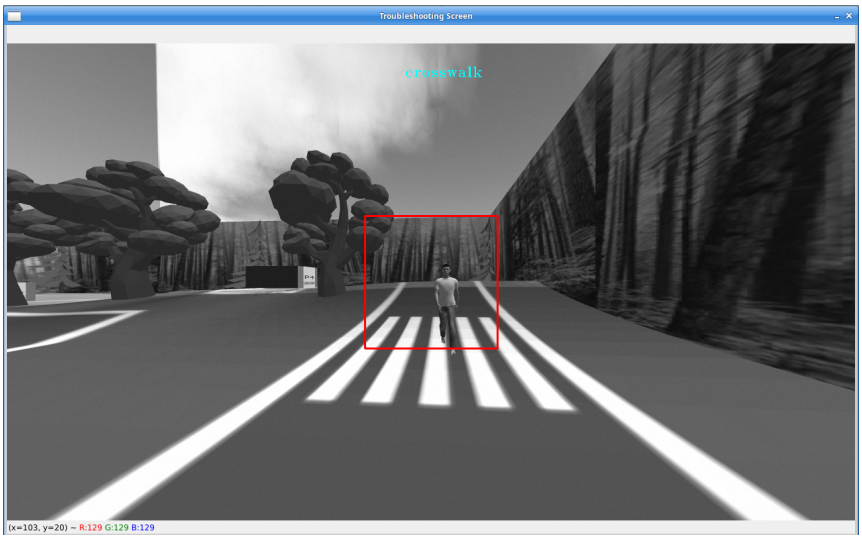
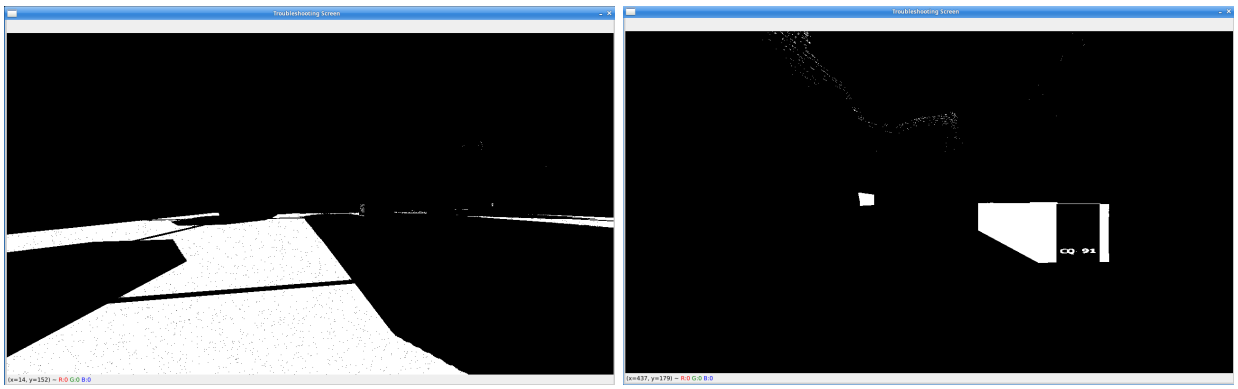


Figure 8: View of the robot in the "crosswalk" state. The average value of the image within the red box is computed and compared to previous values

To detect the pedestrian, we used the following code:

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

    # stop the car
    move.angular.z = 0
    move.linear.x = 0

    # look for the pedestrian
    error_val = 0.5
    img_val = self.averageImageValue(gray,640,360,100)
    if time.time()>self.delay_time+1:
        if abs(img_val-self.img_val_last) > error_val:
            self.pedestrian_count += 1
            self.state = "drive"
            self.detection_time = time.time()
    self.img_val_last = img_val
```

```
def averageImageValue(self, img, X, Y, size):

    cropped_img = img[Y-size:Y+size,X-size:X+size]

    # blur image to reduce noise
    cropped_img = cv2.GaussianBlur(cropped_img,(1,1),0)

    return np.mean(cropped_img)
```



(a) HSV mask for the gray of the road

(b) HSV mask for the blue parked cars

Figure 9: HSV masks applied in the “inner loop” PID state

The HSV filters in figure 9 were computed using

```
lower_road = np.array([0,0,80])
upper_road = np.array([0,0,100])
road_mask = cv2.inRange(hsv, lower_road, upper_road)

low_blue = np.array([115, 50, 50])
high_blue = np.array([130, 255, 255])
blue_mask = cv2.inRange(hsv, low_blue, high_blue)
```

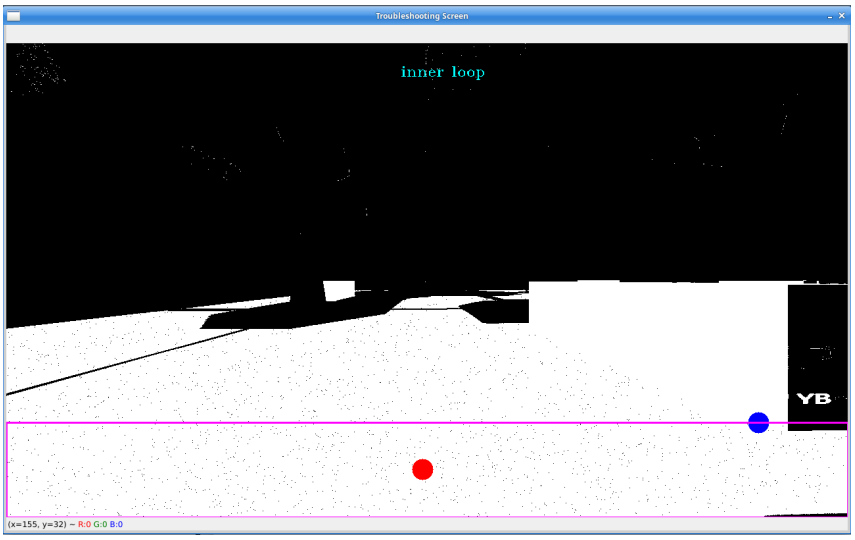
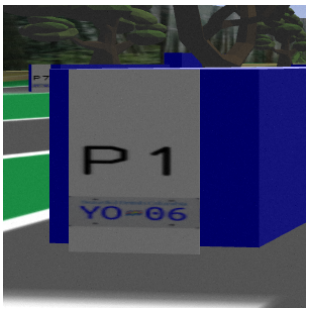


Figure 10: Image showing the road centroid and blue centroid in the "inner loop" PID state

Navigating the inner loop was accomplished with the following code:

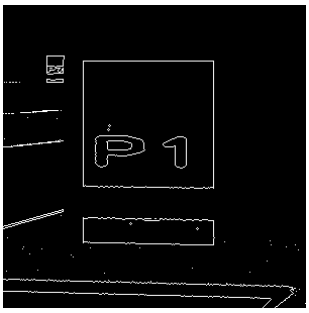
```
delay_time = time.time()-self.detection_time
# turn left after seeing the car
if delay_time > 2 and delay_time < 5:
    # drive left
    move.angular.z = self.PIDcontrol(centersL[0]-eCL[0],angMax,image_width/2)
    move.linear.x = max(maxSpeed - 0.5*turnReduction*abs(move.angular.z),0)
# drive the inner loop
elif delay_time > 5:
    road_turn = self.PIDcontrol(cent_road[0]-image_width/2-80,angMax,image_width)
    blue_turn = self.PIDcontrol(blue_cent[0]-image_width/2,0.2*angMax,image_width)
    move.angular.z = road_turn-blue_turn
    move.linear.x = max(maxSpeed - turnReduction*abs(move.angular.z),-0.05)
# wait for the car to pass by
else:
    move.linear.x = 0
    move.angular.z = 0
```

B License Plate Image Processing Steps

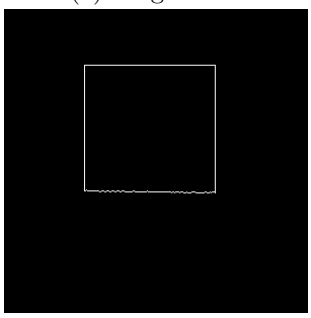


(a) Original

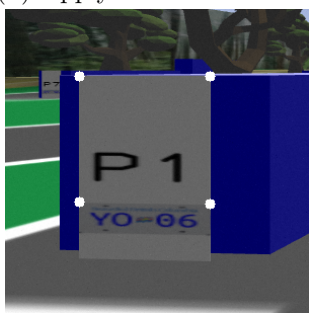
(b) Apply HSV Filter



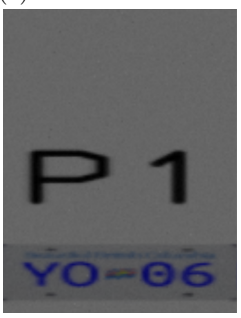
(c) Find Contours



(d) Find Longest Contour



(e) Obtain Perspective Matrix Points



(f) Apply Perspective Transform

(g) Extract plate and parking stall ID and stitch together

Figure 11: Image processing steps undertaken to prepare plates for neural network.

Given the raw image in (a), we applied the following HSV filter to get (b):

```
# convert to HSV
hsv = cv2.cvtColor(plate_img.cropped[:], cv2.COLOR_BGR2HSV)
# mask the image
lower = np.array([0,0,90])
upper = np.array([0,0,210])
hsv_mask = cv2.inRange(hsv,lower,upper)
```

We got the contours in (c) from (b) using the following method and parameters:

```
# get the contours
contours = cv2.findContours(hsv_mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)[0]
```

To get the largest contour in (d) from (c) we applied the following sort:

```
try:
    largest_contour = sorted(contours, key=cv2.contourArea, reverse=True)[0]
```

```
except IndexError:
    return (None, 0)
```

To get the corners shown in (e) from (d), we used a polygon approximation:

```
epsilon = 0.05*cv2.arcLength(largest_contour,True)
box = cv2.approxPolyDP(largest_contour, epsilon, True)
```

After sorting the corners in (e), we apply a perspective transform to achieve (f):

```
dest_pts = np.array([(0, 0), (500,0), (500, 500), (0,500)])
matrix = cv2.getPerspectiveTransform(np.float32(plate_img.corners), np.float32(
                                        dest_pts))
warped = cv2.warpPerspective(plate_img.raw_fr[:], matrix, (500,650))
```

The image in (f) is then cropped and reformatted to give (g):

```
plate_img.plate_crop = warped[500:650,:] # Necessary so that neural network gets
                                         the plate later
plate_img.car_id_crop = cv2.resize(warped[250:500,250:], (120,150))

combined_img = 255*np.ones((150,620,3), dtype=np.uint8)
combined_img[:,0:500] = plate_img.plate_crop[:]
combined_img[:,500:620] = plate_img.car_id_crop[:]

plate_img.combined = combined_img
```

C Neural Network Metrics

C.1 Generated License Plate Examples

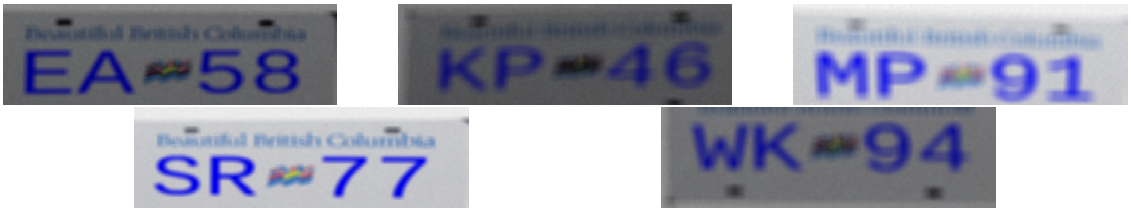


Figure 12: Examples of generated license plates, showing some randomization in x/y translation of the plate, perspective, gaussian blur, and noise.

C.2 License Plate Generation Code

We used the following code to generate our data. We start with a perfect license plate generated using the script provided with Lab 5. We also select random values for all of the effects that will be applied to the plate. The range of values was selected with trial and error.

```
    # Convert perfect license plate to cv2 image
    perfect_plate = np.array(perfect_plate_pil, dtype=np.uint8)

    #Generate random values for distortion and noise
```

```
plate_y_change = -20 + randint(0, 40)
plate_x_change = -20 + randint(0, 40)
blur_amount = randint(7, 20) # Gaussian blur amount
persp_dy = [randint(0, 15) for i in range(4)]
persp_dx = [randint(0, 15) for i in range(4)]
darken_amount = randint(0,150) # for hsv value which has range 0,255
noise_thresh = 0.25 + 0.5*random.random()
```

First, we apply the x and y translation effect. This involves cutting a slice off of the top or bottom and left or right of the image, and pasting the cropped plate onto a blank canvas (`plate`) in the correct position.

```
# X and Y translation effect
plate_downscaled = cv2.resize(perfect_plate, (500, 150))
plate = 255*np.ones((150, 500, 3), dtype=np.uint8)

x_pos = (max(0, plate_x_change), min(500, 500+plate_x_change))
x_slice = (max(0, -plate_x_change), min(500, 500-plate_x_change))
y_pos = (max(0, plate_y_change), min(150, 150+plate_y_change))
y_slice = (max(0, -plate_y_change), min(150, 150-plate_y_change))

plate[y_pos[0]:y_pos[1],x_pos[0]:x_pos[1]] = plate_downscaled[y_slice[0]:
                                             y_slice[1],x_slice[0]:x_slice[1]]
```

Next we apply a perspective transform and gaussian blur.

```
# Apply perspective distort
pts_orig = np.float32([[0,0], [500,0], [500,150], [0,150]])
pts_dist = np.float32([[0+persp_dx[0],0+persp_dy[0]],
                       [500-persp_dx[1],0+persp_dy[1]],
                       [500-persp_dx[2],150-persp_dy[2]],
                       [0+persp_dx[3],150-persp_dy[3]]])

matrix = cv2.getPerspectiveTransform(pts_dist, pts_orig)
plate = cv2.warpPerspective(plate, matrix, (500, 150))

# Apply blur
plate = cv2.blur(plate, [blur_amount, blur_amount])
```

Next we darken the image and apply salt and pepper noise. We found that the most faithful way to reproduce the variations in darkness from the simulation was to split the image into its h, s, and v layers and decrease the v value for every pixel. This is also a great time to add some salt and pepper noise, which is done by generating an image with random pixels and applying a Gaussian blur to the layer, since the noise seen in the simulation had some smoothness to it. After darkening the image and adding the noise layer, we put the h, s, and v channels back together.

```
# Split image into h,s,v
plate_hsv = cv2.cvtColor(plate, cv2.COLOR_BGR2HSV)
h, s, v = cv2.split(plate_hsv)

# Generate noise layer
noise_layer = np.zeros((v.shape[0], v.shape[1]), dtype=np.uint8)
for i in range(noise_layer.shape[0]):
    for j in range(noise_layer.shape[1]):
        rdn = random.random()
        if rdn < noise_thresh:
```

```
                noise_layer[i,j] = randint(0,20)
    noise_layer = cv2.blur(noise_layer, [4, 4])

    # Apply darkness and noise layer
    v[:] = v[:] - darken_amount + noise_layer[:]

    # Reconstruct license plate
    plate_hsv = cv2.merge([h,s,v])
    plate = cv2.cvtColor(plate_hsv, cv2.COLOR_HSV2BGR)
```

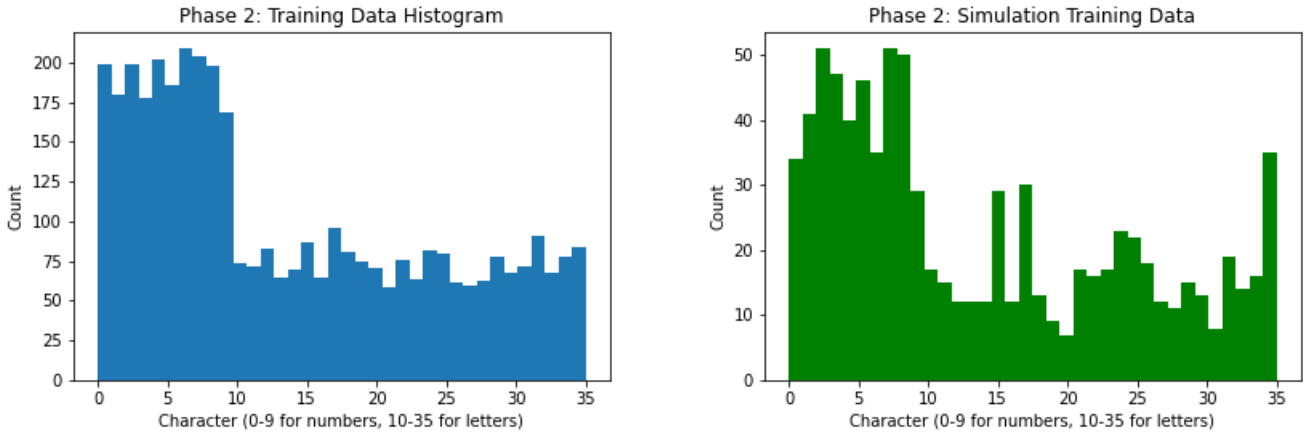
Finally, we downsample the image by a small random amount, by scaling it down and then scaling back up. The resulting effect is a bit of pixelization. The `plate` at the end of this code block gets saved and used as training data.

```
    # Pixelate
    downsized = cv2.resize(plate, (400 + randint(0, 100), 120 + randint(0, 30)))
    plate = cv2.resize(downsized, (500, 150), interpolation= cv2.INTER_NEAREST)
```

C.3 License Plate Neural Network Summary

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 148, 118, 32)      896
 max_pooling2d (MaxPooling2D  (None, 74, 59, 32)       0
 )
 conv2d_1 (Conv2D)           (None, 72, 57, 128)       36992
 max_pooling2d_1 (MaxPooling  (None, 36, 28, 128)      0
 2D)
 conv2d_2 (Conv2D)           (None, 34, 26, 256)       295168
 max_pooling2d_2 (MaxPooling  (None, 17, 13, 256)      0
 2D)
 conv2d_3 (Conv2D)           (None, 15, 11, 512)       1180160
 max_pooling2d_3 (MaxPooling  (None, 8, 6, 512)        0
 2D)
 flatten (Flatten)           (None, 24576)             0
 dropout (Dropout)           (None, 24576)             0
 dense (Dense)               (None, 512)               12583424
 dense_1 (Dense)             (None, 36)                18468

=================================================================
Total params: 14,115,108
Trainable params: 14,115,108
Non-trainable params: 0
_________________________________________________________________
```

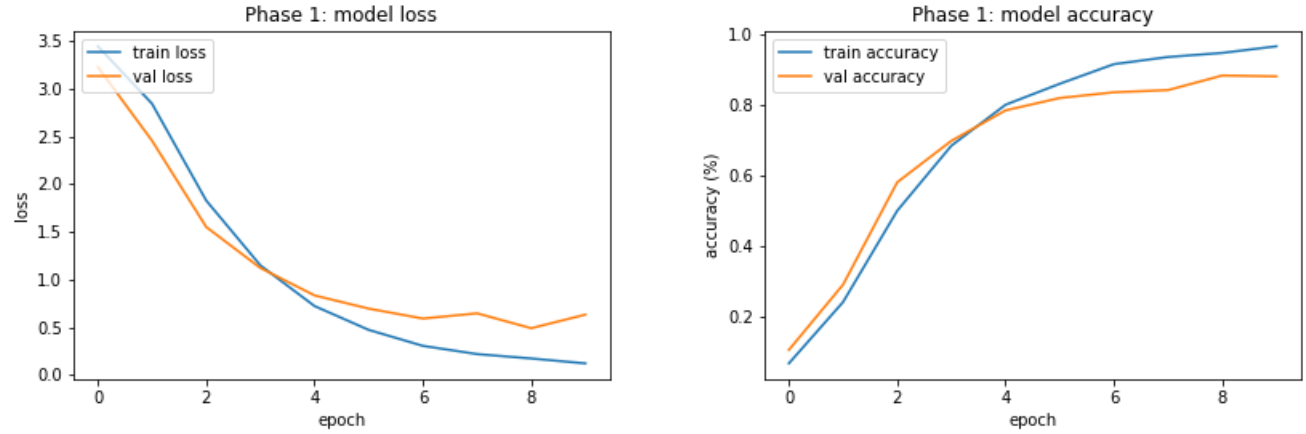
C.4 License Plate Neural Network Training Metrics



(a) (b)

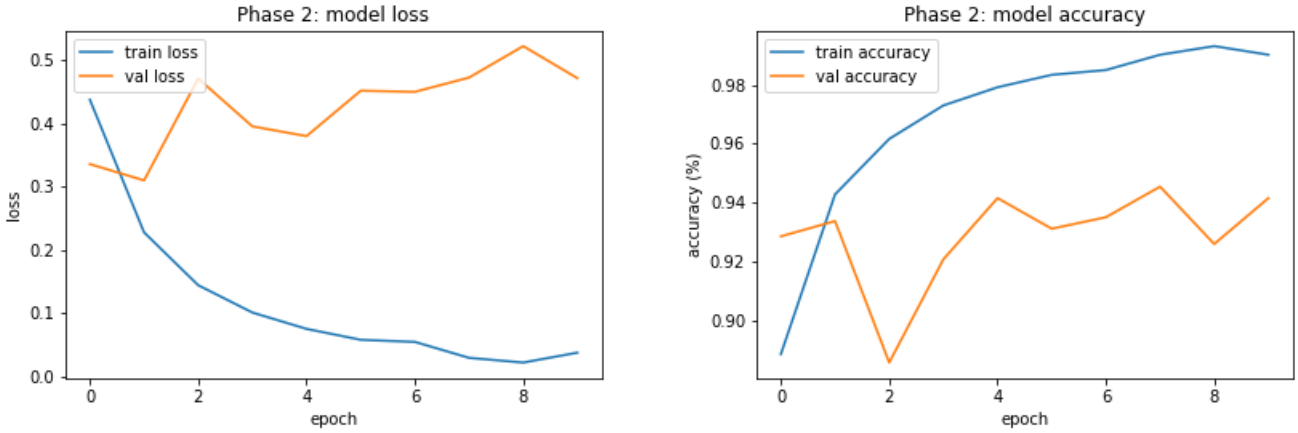
Figure 13: Histograms of data used in 2nd training phase.

Figure 13 shows histograms of the data used in the second training phase. (a) is for the entire dataset (800 generated + 200 simulation) while (b) focuses on the 200 simulation plates.



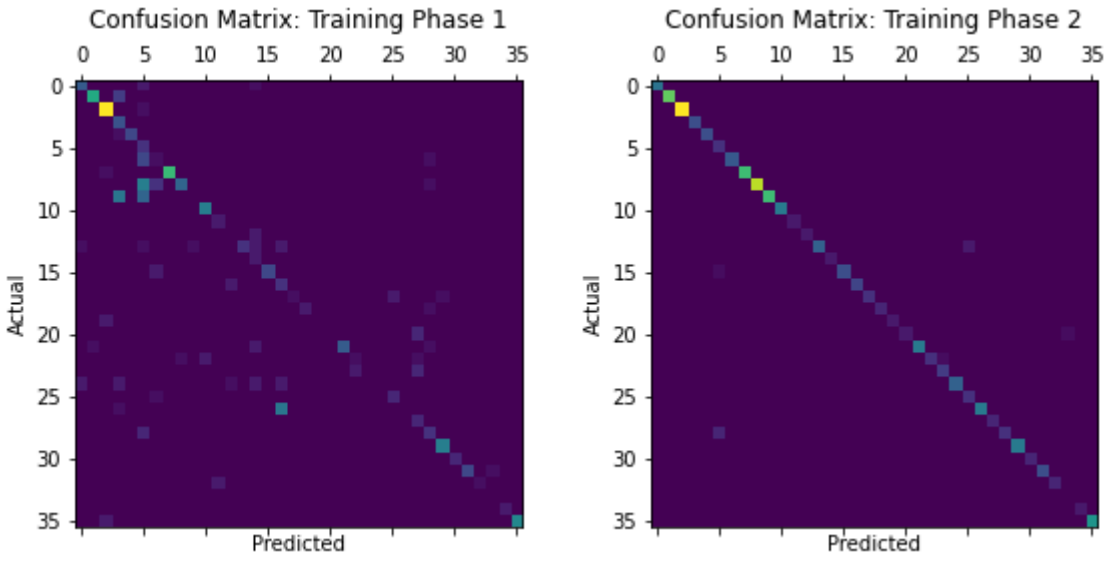
(a) (b)

Figure 14: License plate network loss and accuracy in first training phase.



(a) (b)

Figure 15: License plate network loss and accuracy in second training phase



(a) (b)

Figure 16: Confusion matrix when license plate network was tested on the same set of images that were not used for training or validation.

D Links and Media

Link to our team's GitHub repository: `https://github.com/Enph353-Team-20/controller_pkg`

Link to our license plate neural network training code: `https://colab.research.google.com/drive/1BVN6HYZAAfA3Z5OGEbo1AT4XNALMtZVJ?usp=sharing`

Link to our parking stall ID neural network training code: `https://colab.research.google.com/drive/137n_6z-Up2uz9sKW7OaxouHfhTBv7JQj?usp=sharing`

Video of our robot completing a perfect run: `https://youtu.be/k3zzYlgzKM8`