ENPH 353 FINAL REPORT

April 11th, 2021
Arjun Singh
Ben Osborne
Team 10

Strategy

The competition strategy was to first focus on getting as many plates on the outer loop as possible. By focusing on the outer loop first, we could guarantee at least five points from a successful lap and a maximum of 36 additional points from the plates. The inner loop was to be tackled later if time permitted as it required avoiding a moving truck. We planned to try and implement an imitation learning approach to navigate the inner loop once we could score consistently well on the outer loop.

Software Architecture

The agent functions using two nodes, one for navigation and another for plate detection and recognition. The modular approach allowed for the navigation to be developed separately from the plate detection. A central github repository was used for the main navigation and plate detection while a separate repository was used for training the neural networks.

1. Navigation

The navigation node's purpose was to process image frames from the agents camera and publish velocity commands that the agent can execute. Our general approach was to try and design a working navigation for the outer ring using PID type control and then try and implement the navigation in the inner loop using an imitation learning approach if time permitted. In the end, a Neural Network was trained to control the agent in the inner loop but there were issues transitioning from the outer loop PID navigation to the inner loop machine learning navigation that were not resolved in time for the competition. As such only the outer ring PID navigation was used for the competition. For more information about the attempted transition from outer to inner loop see Appendix A1. Pedestrian detection and avoidance was also done by the navigation node. Finally, the navigation node was responsible for starting and stopping the competition timer by submitting messages to the scoreboard topic.

Driving

Navigating a complex environment with obstacles, roads and intersections added additional challenges when compared with a simple line following robot. The major obstacles were dealing with the intersections, edge cases caused by the environment and pedestrians. An initial implementation was done for the time trials however this version was slow and unreliable. A second implementation was done that was used for competition. This version borrowed some concepts from the first but had improved in speed and reliability.

In both implementations, three main masks were applied to the image feed in order to isolate and detect the number and position of various features of the world although in the initial implementation of navigation - which was used for the time trial, another mask was used to isolate for the parked cars to avoid the agent confusing the indent in the road with a corner.

1. Gray mask: isolates for gray road
2. White Mask: isolates for white road lines.
3. Blue Mask: isolates for the pants of the pedestrian.

The first approach was to loop over the image and compute the center of mass (CoM) of the road in the near ground (bottom quarter of the screen). Using the horizontal location of the CoM of the road in relation to the center of the screen, the proportional difference was computed and used to control the agent's angular velocity. This approach was able to navigate the corners and simple straight sections of the road suitably but dealt poorly with the intersections and additional cases encountered when navigating the environment. For example, since the agent would often confuse the indented parking spots with corners, an additional mask was required to identify and avoid the parked cars.

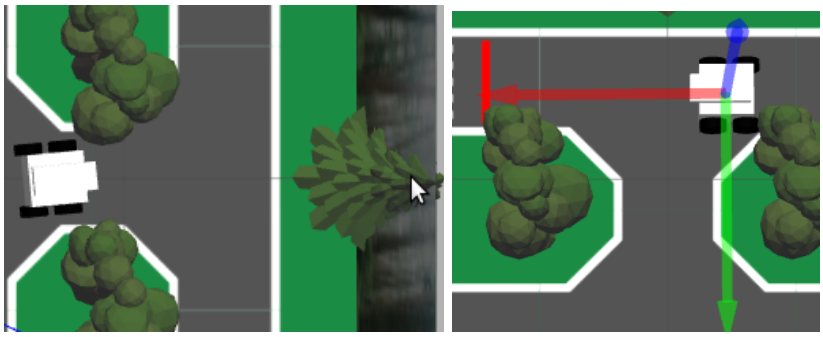


Figure 1.1: Shows starting intersection orientation (right) and a subsequent intersection orientation encountered on the outer loop (left).

When navigating the outer loop, the agent will encounter numerous T-intersections (Figure 1.1). Besides the first intersection it encounters where it must turn left, in the others the agent must go straight to avoid entering the inner loop. When using just the center of mass of the road to control the agent, it will either drive straight into the wall at the first intersection since it is symmetrical and turn into the inner circle at the subsequent intersections due to the asymmetry.

Detecting the edges of the road helped resolve this issue. If the center of mass of the road was near the center of the screen and no road edges were detected then the agent would know it was in the first intersection and that it should turn left. Additionally, if the center of mass of the road was over to the left and no right edge of the road was detected then the agent knew that it was dealing with an interaction. Once moving past the first interaction the agent would always make sure that it could detect the right side of the road as this was a consistent line around the outer path.

The implementation of navigation used in the time trials had many flaws. Mainly speed and the amount of specific cases it needed to handle. Even though the images were processed using numpy arrays, looping through images to calculate the locations of the road and roadlines is

extremely slow in python[1]. This issue was resolved by using OpenCV functions to find the contours of different features in the world such as the road and pedestrian. Once the contours are found, OpenCV provides methods to compute the various moments of the contours and from there the center of masses could be found.[2]

The large amount of specific edge cases meant that the agent could easily enter an unstable state and leave the road. This, coupled with the large processing time, meant that the agent was slow and unreliable. In the final version of navigation, the number of specific cases that had to be checked was drastically reduced. Instead of computing the center of the road and road edges for a specific section of the image, the foreground of the image was split into ten horizontal sections and the locations of the road and road edge lines for each slice were computed and saved to a list.

Next, the valid center of mass (CoM) locations for the road were extracted. A location was considered valid if a right edge of the road could be detected in the given section. Next the average of all valid locations was taken and used similarly to the first implementation to calculate the necessary angular velocity. Taking an average of CoM locations from valid horizontal slices acted as a low pass filter and eliminated the need to account for a number of special cases. For example the agent was less sensitive to the indents in the road at the parking spots. Moreover, since detecting the right side of the road determined if the CoM location for a particular slice was included in the average, the agent was biased to go straight at subsequent intersections, although it did tend to turn inward slightly. The slight turning worked to our advantage however, as after the intersection the agent was perfectly positioned to get a reading of the next parked car. By looking throughout the image feed for valid CoM locations to use, the agent was much more robust.

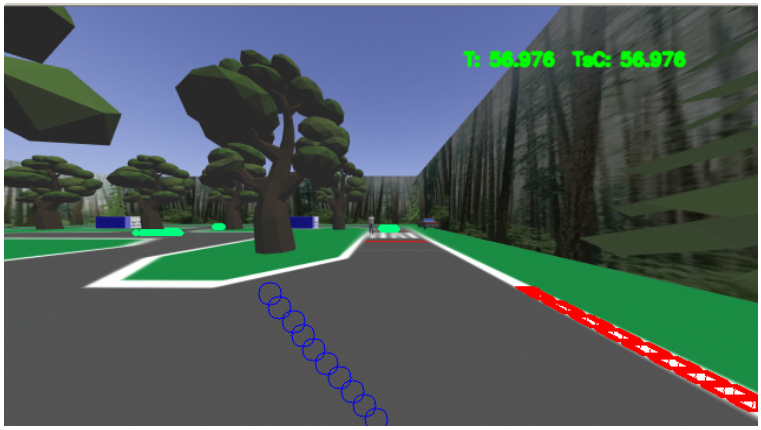


Figure 1.2: showing CoM and road edges detected in the horizontally sliced image. The light green contours show that the road has been identified in the far ground.

This implementation was faster in processing time and noticeably more reliable than the first implementation. Consistent outer loop lap times of around 20 seconds were achievable

[1] https://www.pyimagesearch.com/2017/08/28/fast-optimized-for-pixel-loops-with-opencv-and-python/
[2] https://www.pyimagesearch.com/2016/02/01/opencv-center-of-contour/

(however the velocity was lowered to allow the license plates to be more easily detected). The CoM locations and road edges also were highlighted on the debug image feed as shown in Figure 1.2.

The main issue caused by averaging CoM locations was that the agent would take wide corners and would miss the plates of the parked cars directly following the bends. At the 90 degree bends, it would take time for the average CoM of the road to be pulled to the side causing a delay in the turning. Additionally, the effect of averaging would dampen the perceived sharpness of the corner resulting in a large turning radius. To fix this, the contour of the road in the far ground was searched for in each image. If it could detect a road in the far ground (top region of the image, see Figure 1.2) then the agent knew the road continued straight. If it couldn't detect the road in the far ground then it meant that there was a sharp bend in the road. If this was the case, the speed was reduced and the gain of the proportional control was increased resulting in slower tighter corners.

Pedestrian Detection and Avoidance:

The pedestrian avoidance was a crucial function of the navigation node since hitting the pedestrian carried a steep points penalty.

The agent would search for the pedestrian each time it encountered a crosswalk. Crosswalks were identified by counting the number of white lines in the bottom horizontal slices of the image feed. The pedestrian identification was done using a mask to filter for the color of the pedestrian's pants. This was the largest region of the pedestrian that was a unique color in the simulation. Masking for the shirt for example would lead to false positives as the road lines were also white.

Next the agent would try and get the location of the edges of the road. This was done by using the location of the centers of leftmost and rightmost road lines in the bottom two horizontal slices to plot lines following the edges of the road. With the points found, line equations could be computed to describe the boundaries of the road on the image feed. Comparing the lowest point on the pedestrian contour with the line equations for the edges of the road, the agent could detect whether or not the pedestrian was crossing.

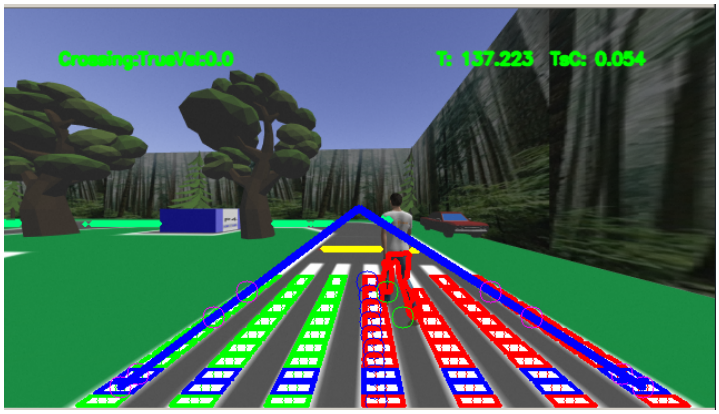


Figure 1.3: Shows the Pedestrian crossing. His location is identified in the debug image by the green circles and red outline of his pants. The multiple blue contours of the crosswalk lines indicate that the crosswalk has been found. The trajectories of the detected edges of the road are plotted in blue.

One important case that had to be considered is if the pedestrian is just about to cross when the agent encounters the crosswalk. If the agent proceeds, (since it doesn't detect the pedestrian crossing the intersection) it runs the risk of having the pedestrian side swipe it. To solve this the simulation times of the last pedestrian crossing and last pedestrian sighting were recorded. The agent would only proceed through the intersection if it had seen the pedestrian cross within the last four simulation seconds. This period of time was chosen to match the period of time between crossings. Thus, the agent would stop and wait for the pedestrian to cross before proceeding. Once proceeding the agent would use the time from the last pedestrian sighting to confirm that it was in the same intersection and therefore didn't have to stop and look for the pedestrian again. This method of pedestrian avoidance showed promising results from testing the agent by having it do continuous laps around the outer loop.

2. Plate Detection

The purpose of the plate detection node was to recognize the license plate/parking spot and publish it to the score tracker. The failure modes for the node were publishing an inaccurate reading and missing a reading. To mitigate these, A mix of computer vision techniques and neural networks were used for accuracy and robustness.

The node also consists of data collecting functionality for training the neural networks.

Node Layout

The main executable 'plate_detector.py' consisted of a subscriber to the camera topic and a publisher to the score tracker topic. A class structure was chosen for the script to allow the subscriber and publisher share data and methods, avoiding the use of global variables for faster processing time.[3]

[3] https://docs.opencv.org/master/dc/d71/tutorial_py_optimization.html

The methods used in the node are a part of Numpy, Scipy, OpenCV and PyZbar packages. All image processing was done using vectorized Numpy arrays instead of Python lists due to better performance and a greater variety of methods available to manipulate the data type.[4] OpenCV was largely used to implement computer vision protocols and PyZbar was used for data collection.

All the methods associated with neural networks were abstracted into a different class called 'CNNpredictor' contained in the 'predict.py' package for modularity and better code comprehension.

When called, the node would enter an infinite loop using Rospy.spin(), with the callback function performing all the required operations. The main executable would extract the license plate / parking spot via image processing whereas the CNNpredictor object would read the extracted images using Neural Networks. The final result would be published by the main executable.

Image processing

Before passing images to the Neural Network for predictions, They underwent several checks and transformations to mitigate any inaccurate readings. In the event a given frame fails a certain check, a flag was raised to stop the callback and return to loop.

Each frame was passed into the callback function as an argument and went through the following pipeline -

a) Masking - The raw image (Fig 2.1.1) is converted into the HSV space where two masks are applied to it. The car mask establishes a region of interest around the parked cars by identifying blue hues while the plate mask isolates the hue of the license plate (Fig 2.1.4). A series of morphological transformations are applied on each mask with specific kernel sizes to target recurring patterns of noise. An additional dilation (Fig 2.1.3) is performed on the car mask with numerous iterations and a larger kernel size to introduce an area of overlap between the masks. The intersection of both masks (Fig 2.1.4) isolated the plate in the region of interest established by the car mask.

b) Contour detection - The contour detection function[5] is run on the masks and the contour with the largest area is considered (Fig 2.2). Here a zero area indicates the absence of a plate and triggers the error flag.

c) Perspective transform - If a plate is detected, The area of the associated largest contour is checked against a threshold. A small area indicates a license plate at a distance or a clipped plate (usually not fit for prediction) and triggers the error flag.

[4] https://towardsdatascience.com/how-fast-numpy-really-is-e9111df44347
[5] https://docs.opencv.org/master/d4/d73/tutorial_py_contours_begin.html

A valid contour is used to get the features of a bounding box on which a perspective transform[6] is performed to isolate and straighten the plate.
The region (200px) above the bounding box usually contains the parking spot and is also saved.

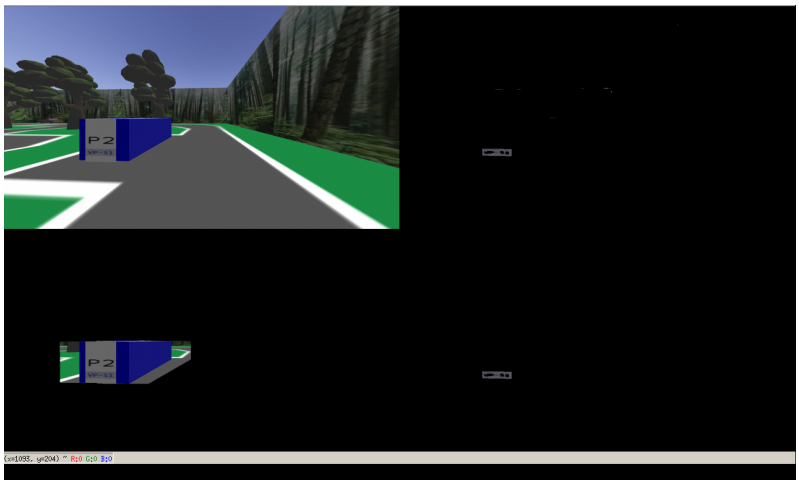


Fig 2.1.1 (top left): Raw camera feed
Fig 2.1.2 (top right): License plate mask
Fig 2.1.3 (bottom left):Car mask
Fig 2.1.4 (bottom right): Raw camera feed

d) License Plate processing - The transformed region with the license plate is then passed into the CNN predictor object where it is transformed into the right format for the NN (fig 2.3).
The plate is first masked to isolate the digits and remove noise. At this point, a plate that is only partially legible or too distorted to be accurately read could have made it past all the previous checks. A digit count implemented using numpy functions is performed to count the number of isolated white blobs corresponding to the image. If four distinct digits are detected, the image is spliced, resized and normalized for the Neural Networks.

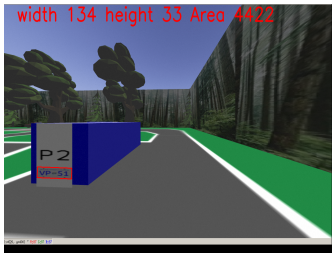


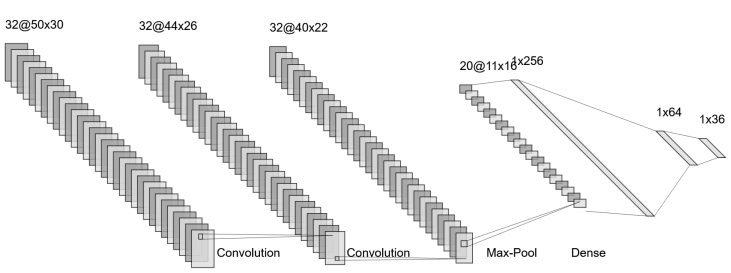
Fig 2.2 (right) : License plate contour
Fig 2.3 (left) : License plate processing

[6] https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html

e) Parking Spot processing - A mask is applied in the region containing the parking spot to isolate the parking spot digits (fig 2.4). A digit count is performed to determine if the two digits corresponding to the parking spot have been located. If the image is valid, it is resized and normalized for the neural networks.

Fig 2.4 (right) : Parking spot after masking

Neural Networks



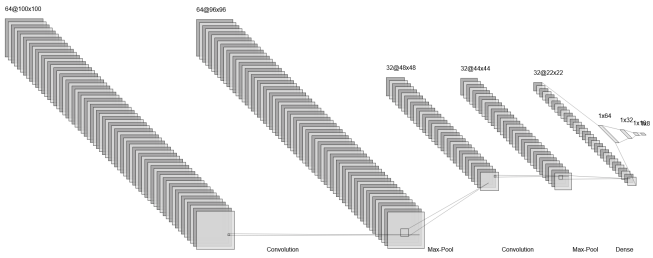


Fig 2.4.1 (Left) : License plate NN architecture
Fig 2.4.2 (Right) : Parking spot NN architecture

The node uses two classification neural networks to detect the license plate digits and the parking spots (Fig 2.4.1 and Fig 2.4.2). The initial layout was based on the referenced example[7] and iterated upon to decrease loss.

Relu activation is used for each layer except the last one which uses Softmax (as is the norm for classification NN). To prevent overfitting, drop out layers and l2 weight suppression were used after dense layers and convolutional/max-pool layers respectively. The hyperparameters

7 https://machinelearningmastery.com/handwritten-digit-recognition-using-convolutional-neural-networks-python-keras/

were tweaked based on the loss/accuracy metrics between each iteration of training. Both networks were trained using Adam optimizer to set the learning rate.

Images were significantly scaled down for faster training times and reduced memory usage. They were also normalised for faster convergence.[8]

While the initial model for both the networks were similar, the final networks ended up having distinct layouts through iterations. The major points of difference between the networks are as noted :

- Classes - The license plate NN classified images into one of 36 classes (corresponding to each digit) whereas the parking spot NN classified images into 8 possible classes (corresponding to each parking space)
- Data Set - The greater number of classes and the large variance in collected data for license plates demanded a much larger data set than the parking spot NN. Hence, the dataset for license plates was augmented using the Keras Image Generator[9]. The images also had to be scaled down to a greater extent for larger data sets to fit memory constraints.
- Dense Layers - While training , The parking spot NN converged at a much lower loss on increasing the amount of dense layers as compared to the license plate NN.

The networks were trained until a divergence was observed between training loss and validation loss indicating overfitting. The batch sizes were chosen proportional to the dataset sizes but were always < 64. The metric data and the confusion matrix for the final generation of the networks are in appendix A2.

Training

Training methods were built into the main node classes to allow reusability of code. Data collection was controlled through constants in the class set while initializing the class object. The training was done on Google collaboratory notebooks[10] [11].

To collect data for the license plates, The cars were spawned with the QR codes. The agent was manually driven around the simulation where it would apply the described image processing and save the resulting digits as labelled files. The labels for the file were automatically generated using QR codes decoded by the PyZbar library.

About 2800 images were collected for training. They were augmented using the parameters listed in A3. The dataset was augmented with classes associated with high error identified with the confusion matrix between training sessions. The final iteration of CNN converged after

[8] https://www.google.com/search?client=firefox-b-d&q=do+normalized+images+converge+faster+keras
[9] https://keras.io/api/preprocessing/image/
[10] https://colab.research.google.com/drive/1Uu3gfR4x5WvxF5D6nj-w1KpusljeYqR3?usp=sharing
[11] https://colab.research.google.com/drive/18BNIkGJ8WSuOIDbpsl4V0r45Kw6XXBS9?usp=sharing

training for 37 epochs. There would be occasional erratic readings for the digits ‘O’, ‘D’, ‘V’ and Y. These erroneous readings were mitigated through the publishing method chosen.

The data collected for the parking spots was much simpler as the spots did not show any variations in different simulation instances. The target images were also of a much higher resolution, usually giving a clear reading and minimizing bayesian errors. Images were collected in a single run and manually placed in different folders. A python script was used to label, normalize and resize each image . The final iteration of the neural network was trained for 70 epochs. The Bayesian errors were again addressed at publishing.

Publishing

While the neural networks displayed a high value of accuracy, there were still non negligible Bayesian errors. To obtain consistent readings, The node was made to collect readings until the targeted license plates were in sight. With a larger range of readings to choose from, outliers could be eliminated using the mode (Scipy statistics[12]). The node would publish the readings when the license plate went out of sight, indicated by detecting a plate region with area less than the threshold. The node was allowed to publish different license plate strings for the same parking spot. This would allow the node to correct any inaccurate readings published in a previous lap.

Results

Testing prior to competition showed promising results of 35 to 41 points scored. The agent could identify at least 5 out of 6 plates on the outer loop within two laps and complete a lap in ~ 40 seconds.

Throughout testing of the fully integrated agent, a collision with a pedestrian was never observed. However, during the competition, the agent oriented itself at a sharp angle to the crosswalk. This caused the agent to confuse the lines of the cross walk with the edges of the road and falsely conclude that the pedestrian was not crossing. The final competition score was 25 (30 points for 5 plates, 5 points for at least one lap, -10 points for pedestrian collision).

Further testing of pedestrian avoidance could have mitigated this. Based on the fact that the agent did not collect any new plate data in the 30 seconds leading up to the collision, the timer could also have been stopped after a target number of plates were collected.

The agent was able to avoid all other failure modes. The strategy chosen proved to be a good starting point for the given task. With improvements in the areas described, the strategy could show significantly better performance in future runs.

[12] https://docs.scipy.org/doc/scipy/reference/stats.html

Appendix

A1. Outer Loop to Inner Loop transition

An attempt was made to integrate an imitation learning based navigation for the inner loop with the PID based navigation for the outer loop. This required a transition to be made during the competition where the agent would have to enter the inner loop and orientate itself so that the imitation learning algorithm could take control.

The transition is triggered when either 3 minutes has passed or all the plates in the outer loop have been detected. Since the imitation learning was trained from a particular starting orientation, the agent traveling the outer loop had to enter the inner loop and stop at a particular location before allowing the imitation learning to take over.

Entering the inner loop was accomplished by increasing the gain of the proportional control causing the agent to turn inwards at the intersections. The inner loop parked cars were used to detect the proper stopping point of the agent. Upon entering the inner circle, the agent would be perpendicular to the parked cars. It would then be able to detect that it was at the end of the connecting road when the contour of the parked car reached a certain area. The agent would then stop (Figure A1.1). Once stopped, the agent would turn so that the center of mass of the parked car was on the right side of the screen thereby biasing the imitation learning to move the agent in the correct direction around the inner loop. In order to avoid a collision with the pick-up truck, the agent would only be allowed to proceed if the truck had just passed by. The truck passing was detected by waiting for the parked car contour to disappear (as the truck passes) and the reappear.

Since the transition depended on the agent entering the inner loop at the paths facing the parked cars, the agent had to keep track of its position to know when to turn inwards. This was accomplished by recording the time the agent crossed the last intersection. Testing showed that the agent was always past the first intersection at the 43 simulation second mark past the crosswalk. Only after passing this time threshold would the inner loop entry procedure begin.

This method worked semi-reliably but needed more time to be tested and for the crucial bugs to be fixed. It was therefore not used in the competition.

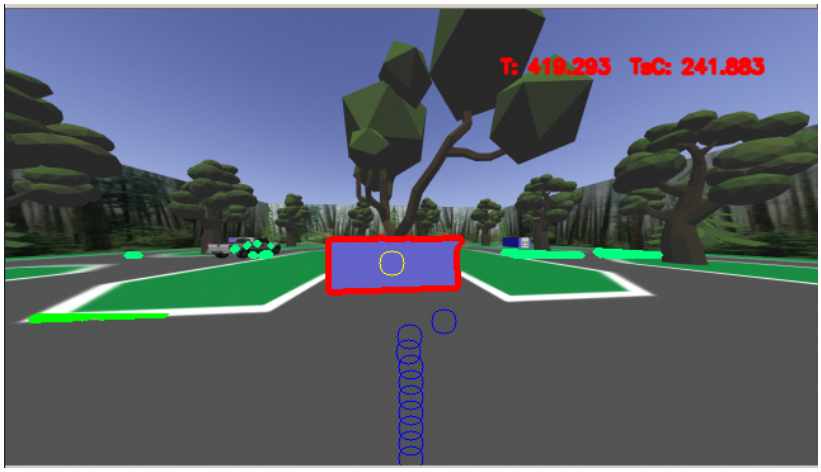


Figure A1.1: Shows result of successful transition from outer to inner loop.

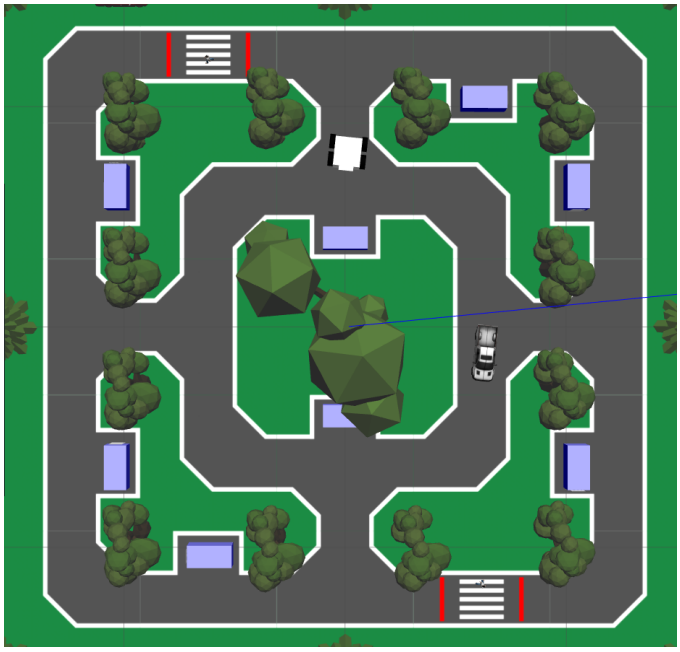
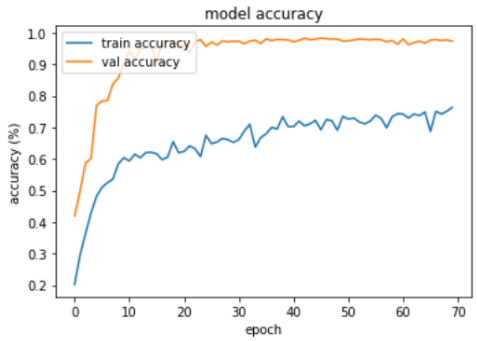
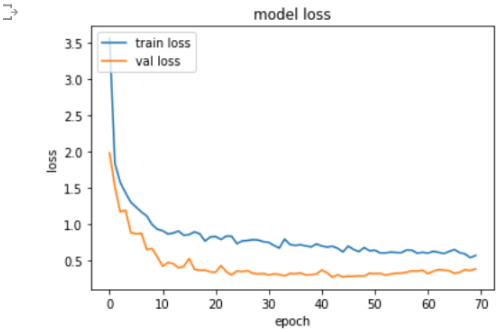


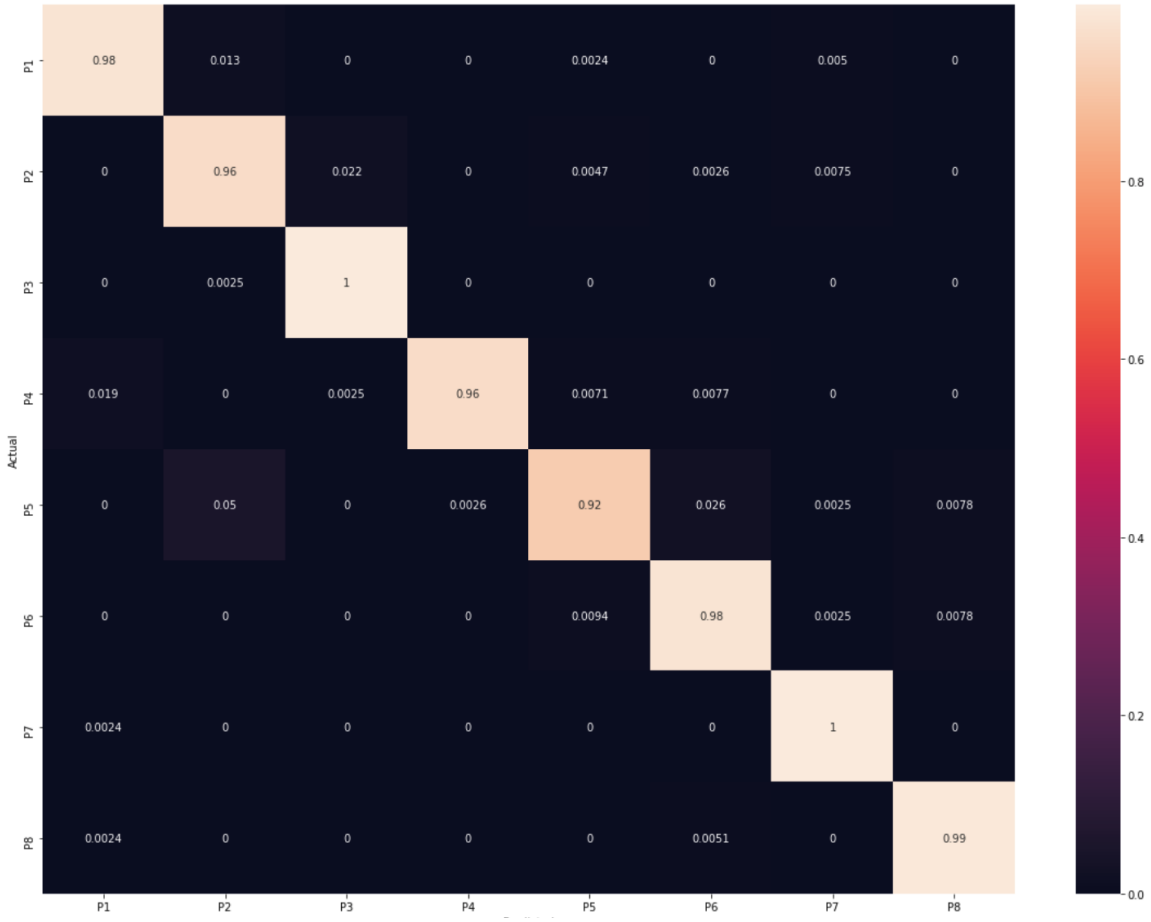
Figure A1.2: Shows the location of the agent upon transitioning into the inner loop just before the imitation learning would take over navigation.

A2. Neural Network Metrics

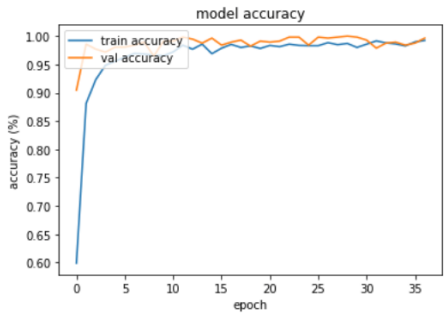
Parking Spot NN

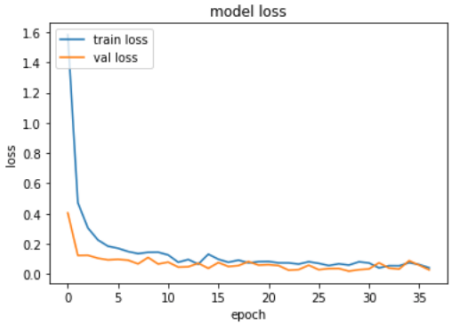


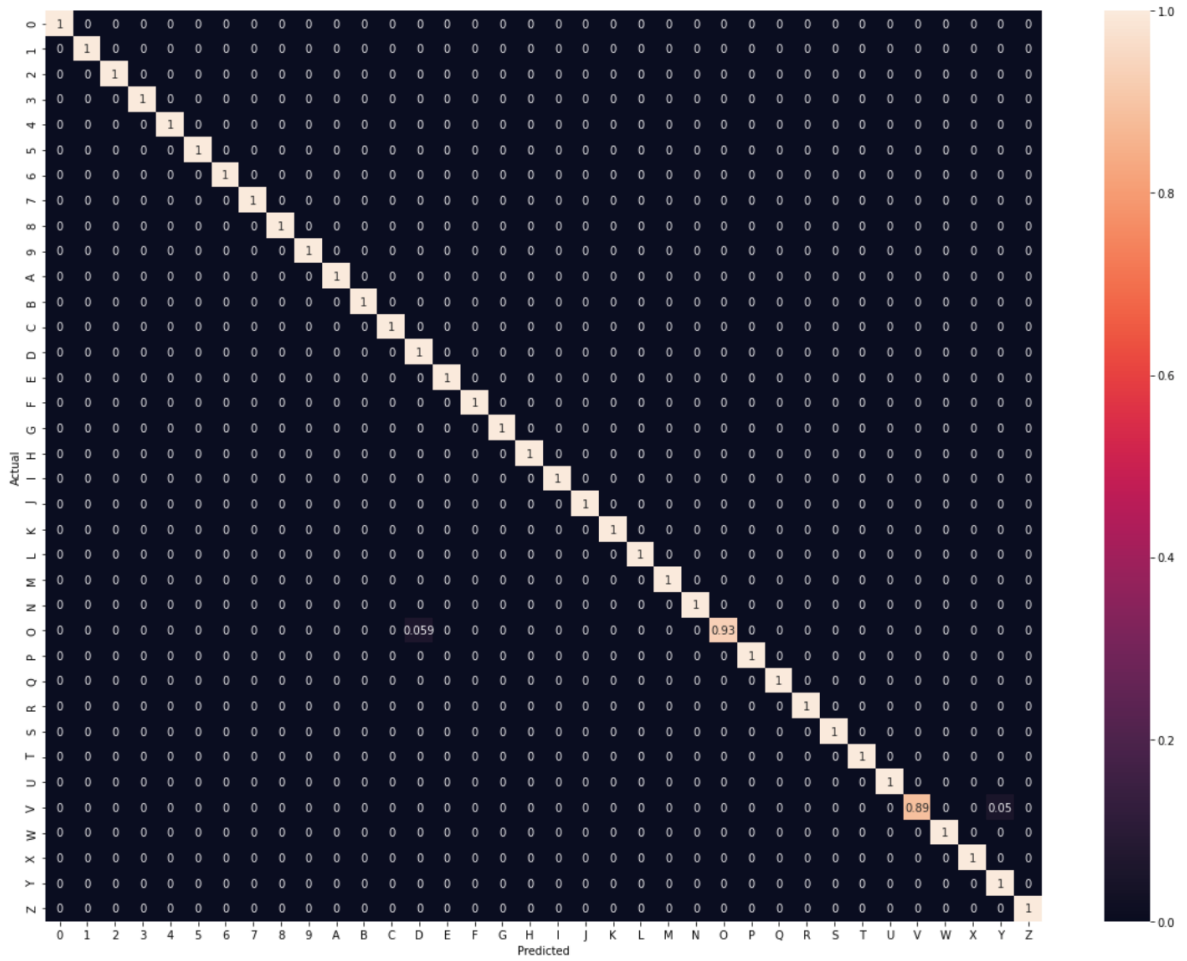




License Plate NN







A3. Image Augmentation Parameters

rotation_range	5
zoom_range	0.05
width_shift_range	0.02

horizontal_flip	False
vertical_flip	False
shear_range	3.0