Autonomous License Plate Recognition Robot
Final Report

Paul Moolan, 72271810
Steven Brown, 90169772
December 5, 2021
ENPH 353

Table of Contents

1. Overview

For our 3rd year ENPH project, the course focus was an introduction to computer vision and machine learning techniques which would then be applied to an end of course simulated competition. In the process of developing these skills we predominantly worked in Linux based ROS Melodic/Gazebo as a simulation ground for our robot to navigate through, as seen in Figure 1, and utilized Google Colab to gain familiarity with python libraries such as OpenCV for computer vision and Tensorflow to design and train neural networks.

In regards to the competition the key objectives were to develop an autonomous robot which could navigate roads and detect/read license plates on parked vehicles, with obstacles including pedestrian crosswalks and other vehicles to avoid. The measures of success in the competition were plates registered and time taken to complete the course. With 6 points for outer loop plates, 8 points for inner loop plates and a cap of 4 minutes for simulation duration with points taking precedence in the final result. Note also that any traffic infractions resulted in points lost.

In achieving these objectives we were limited to using an onboard camera feed and system timer, plus dictated movement commands to a velocity topic and sent plate predictions to a license plate topic.

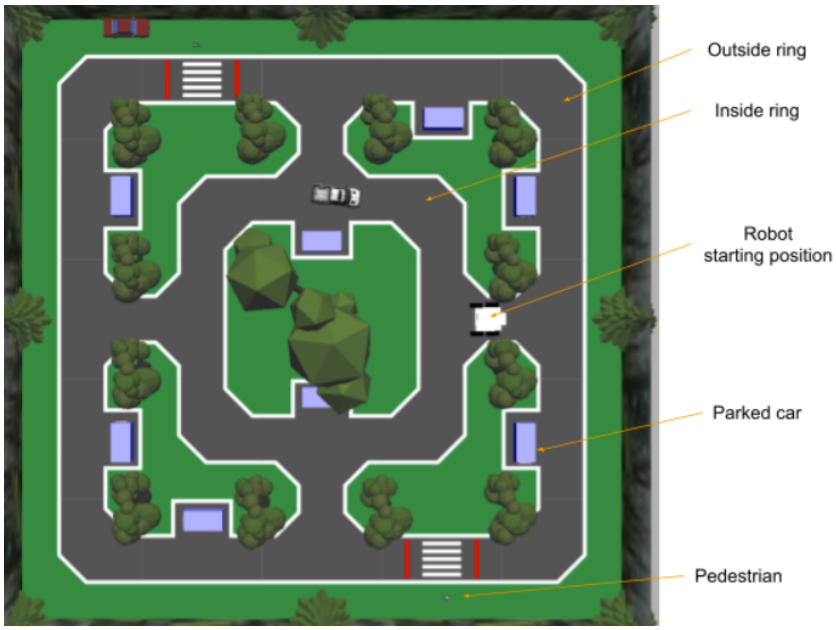


Figure 1: Top View of Simulation track in ROS

2. Software Architecture

The template for our software layout can be seen below in Figure 2, with the key repositories being *competition_pkg, controller_pkg and cnn_trainer*. The first repository, *competition_pkg* is the given competition repository which contains the robot configuration files, track textures, NPC control and launch files to run the simulation in Gazebo.

Of the created folders, the second repository is our *controller_pkg* which contains all logic controls for our robot to navigate the simulated world and detect plus process license plates. More specifically it contains the code to initialize subscriber and publisher topics to the onboard camera feed and license plate reader respectively. It further contains the logic to handle pedestrian crosswalks and other vehicles. In our design we only required the usage of the controller.py file to store these control flows and stored all launch scripts in the adjacent launch folder. The last repository is our *cnn_trainer* folder which contains a portion of our training data plates, our final CNN model used in the competition and the Colab python notebook which was responsible for training and designing the CNN model.

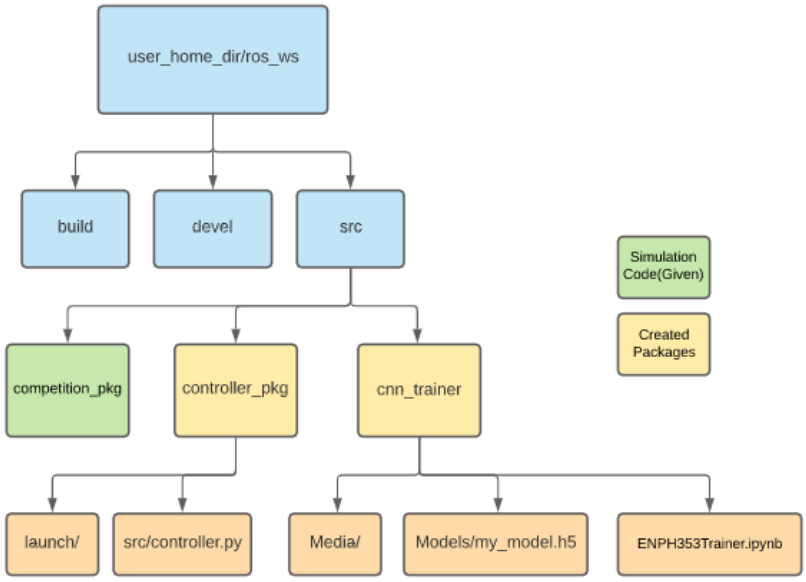


Figure 2: Software Structure/Organization

3. Robot Control Logic

In controlling the flow of our code to acclimate to the specific phases of the competition, we developed a system of detecting specific phases and the ability to switch between them as the simulation progressed. The first phase was to get the robot from its start position to enter the outer loop in position to transition to PID navigation. This was done using manual commands to drive forward and execute a 90 degree turn.

The next phase as aforementioned was to drive around the outer loop which was made possible by creating a state variable so that the robot knew that it was currently in the outer loop. Within this phase and all later phases there was the addition of a control statement to detect if a license plate was in the camera feed and to determine if it was viable to send a prediction to the license plate topic. This plate detection will be discussed in more detail in later sections, but with the ability to check if an image contained a plate, we were able to reduce the computational power that other algorithms such as SIFT would have required to actively search for a plate in every frame.

Once the outer loop was fully traversed as indicated by obtaining the 6th plate, the state was then switched to the inner loop, which was entered through hard coded commands to position the bot at the entrance to the inner loop. From here the control logic changed to use the inner loop PID drive, refer to Section:Driving. After completing the inner loop the robot was then instructed to stop, ending the simulation. The switching of state will be discussed gradually in later sections as the specific mechanisms of the detection of plates and driving are explained.

Additionally, our code included additional control logic to account for obstacles such as pedestrians and vehicles, which much like the other controls involved detection and hard coded commands to resolve obstacles.

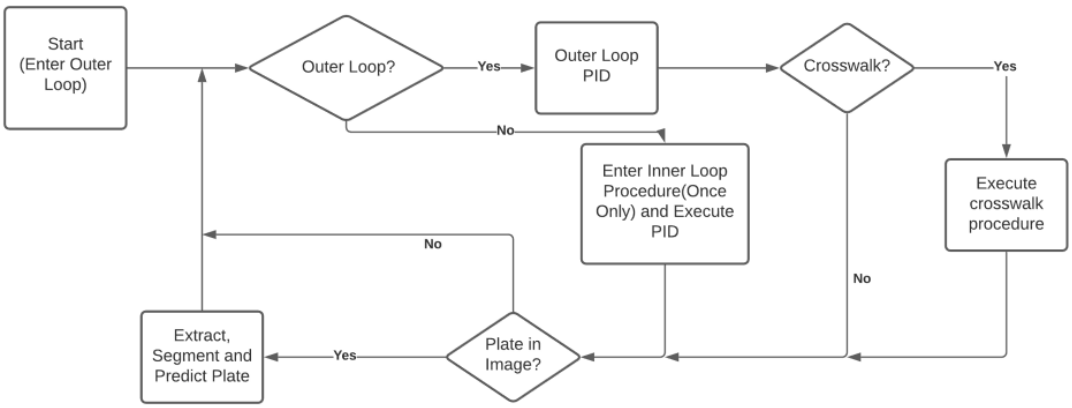


Figure 3: Basic Control Flow of controller.py

4. Autonomous Drive

Our robot agent was required to autonomously interact with its environment and the first challenge that it faced was gaining the ability to navigate the course, maintaining itself within road limits. The main options we considered to achieve this included reinforcement learning and PID control, of these we had prior success with PID and problems with reinforcement learning. Therefore the choice of using PID was clear for us as it was simpler, easier to debug and quick to implement. Once PID was chosen it was further decided that it would be necessary to differentiate PID in the outer loop versus in the inner loop, due to inconsistencies in lane lines and turn options for each loop.

4.1 Outer Loop

The two key road features available for navigation were white lane lines and gray road, in our analysis of these features we decided that the outermost lane line of the outer loop would serve best as it was continuous and contained no biasing elements. Biasing elements referring to road elements that would decenter the PID navigation, such as blocks of gray road in parking spots and inner loop entrances, i.e. why the gray road was the sub-optimal option.

The processing of a frame to execute PID is seen in figure 4, with the final result yielding the centroid of the outermost lane line and the PID calculation requiring the x-coordinate only. From this Xcm the error

was calculated as the robot moved and we only utilized the proportional term of PID to correct. The control mechanism for movement was separated into a linear velocity term $\propto$ C - D*error and an angular velocity term $\propto$ E*error, the C and D terms allow for slower speeds when sharp turns occur and high speeds on straights.

Another feature which greatly enhanced the accuracy of our PID was the addition of a memory term to remember the last direction of error, negative error implying a counterclockwise turn and positive error implying a clockwise term. This prevented the robot from ever losing the lane line, which increased the speed potential of our PID drive by around 2-3 times, allowing for an approximate 30 second outer loop time.

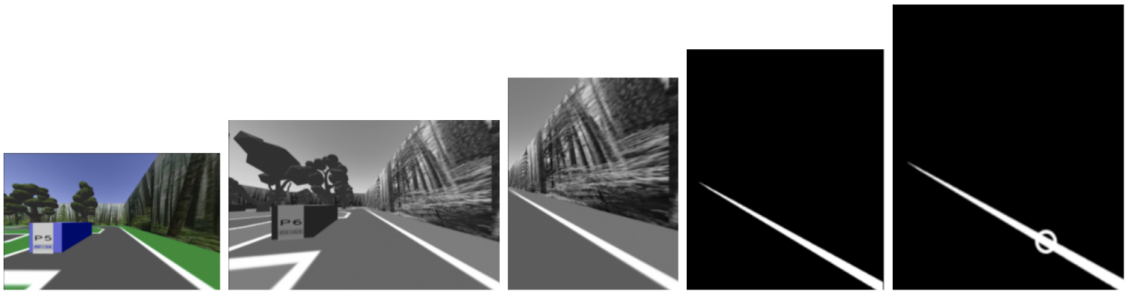


Figure 4: BGR, Grayscale, Cropped, Binary mask of lane line, Center of mass of lane line.

4.2 Inner Loop

Once the outer loop had been traversed, the switch to enter the inner loop and change PID was to pass the 6th parked car and manually turn into the entrance of the inner loop. At the entrance to the inner loop it was necessary to detect the truck before moving into the loop, which is discussed in the next section. The PID to navigate the inner loop was not able to rely on the lane lines to follow as they contained discontinuities on both sides, however we can rely on them as bounds to stay within the loop. The logic utilized was simply to travel straight when no white lane lines were in a cropped segment of the screen, and to follow simple PID control logic when there was a lane line in the cropped segment. This resulted in behaviour which resembled a sort of bouncing of the lane lines to stay within the inner loop. This was relatively easy to implement and worked decently well, but did risk missing a lane line and "bouncing" out of the loop. However this was preferable to smooth turns, as discrete turns in this bouncing sense allowed for better vantage points towards inner loop license plates as found in our testing, which is one of the key reasons for its usage in our final control logic.

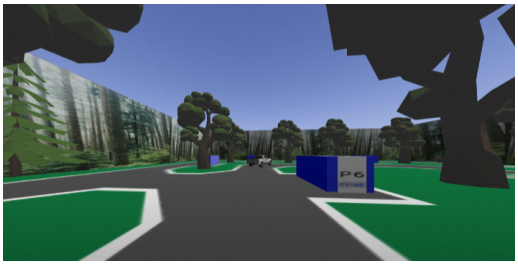


Figure 5: Inner Loop with both lane line discontinuities

5. Obstacles

The first obstacle we focused on detecting was the crosswalk located on the outer loop. From there, we looked at detecting the pedestrian crossing the crosswalk. The same logic was applied for both the outer loop crosswalks. Then, we looked at the detection of the inner loop truck after completing a full run of the outer loop. Detection of these obstacles used a variety of color masks and cropped bounds that are further described in the following sections. Note that the color masks discussed in this section are listed in section 9.2.

5.1 Crosswalk Detection

The white crosswalk was bounded by red lines at the top and bottom making it easy to decide where the robot should stop. We decided to focus on the red line that was first seen by the robot (bottom red line) for crosswalk detection. First the robot was manually driven to the crosswalk where a picture was taken. The picture provided information on the color of the surrounding scenery compared to the bottom red stop line. This image can be seen in Figure 6. A variety of color masks were tested to see which ones made the red stop line stand out the most and which ones removed the most background color pixels. The color mask we chose set pixels that were not red to black and red pixels to white. After masking the image, the number of white pixels in the image were summed and this value was set as a threshold for where the robot should stop. If the number of white pixels is less than that of the threshold, keep driving, if larger, then stop. This function was executed for every camera frame meaning that as the robot approached the crosswalk, the number of red pixels would increase, eventually passing the threshold and stopping the robot.

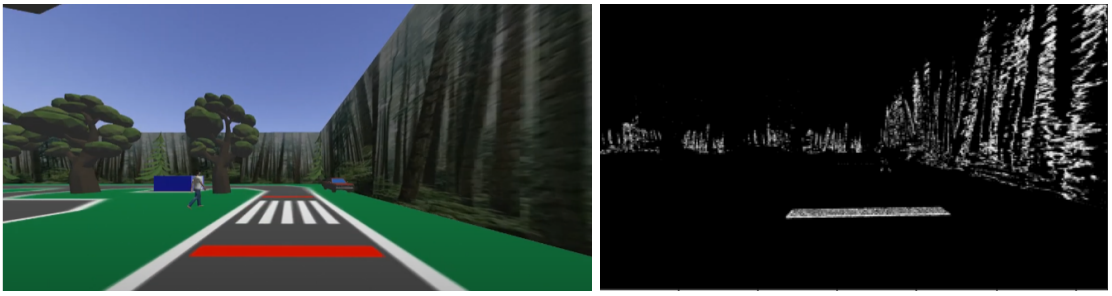


Figure 6: Original and Masked Crosswalk Image

5.2 Pedestrian Detection

Once the robot is stopped due to the red stop line, it begins to check for a pedestrian. Similar to the cross walk, we started by manually driving to the red stop line and taking a picture when the pedestrian was on the side of the road and when in front of the robot. With these images we were able to try different masks focusing on the blue pants of the pedestrian. In addition to color masks, we played around with different image cropping bounds so that the robot would only see the pedestrian when they were on the road. The results of the cropping and color masking can be seen in Figure 7. We again looked for a threshold number of white pixels in the masked image to indicate when the pedestrian was present. The function was repeatedly called at the stop line until the number of white pixels was higher than the

threshold, meaning that the person was on the road. Once the person was sensed the robot was instructed to move forward for 0.4 seconds before continuing with the PID drive.

One concern we had was that the hard coded image crop bounds would not be consistent with where the robot stopped every time, meaning that it might see the pedestrian even when not on the road. After testing and looking at the images after cropping, we found that the robot stopped at a consistent enough location to where the bounds would always crop out the pedestrian when not on the road. A backup plan we thought about in case this method did not work was motion detection. A type of image memory would be implemented so that the current image could be compared with the most recent image. The location of the pixels of the pedestrian would be compared with where the pedestrian was one frame ago.

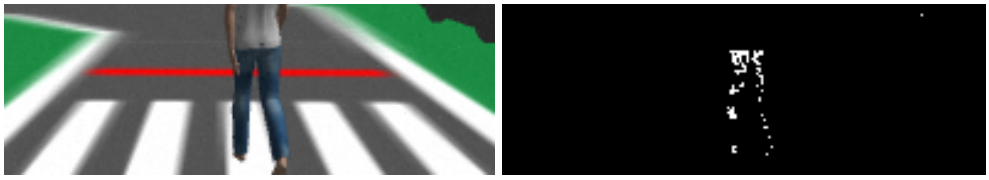


Figure 7: Cropped and Masked Pedestrian Pants Image

5.3 Truck Detection

After the outer loop is finished, the robot enters and stops in the intersection bridging the two loops. The truck detection was implemented the same way as the pedestrian. After collecting images from the intersection location, different cropping bounds were tested after the image was grayscaled. A grayscale mask was used because we noticed that the tires and front bumper of the car were exactly black. The cropping bounds were chosen so that the truck would be sensed when it directly crossed the intersection in front of the robot. The original image and cropped grayscale image can be seen in Figure 8. Again, a black pixel threshold was found to indicate when the truck was in front of the robot. Once the truck was sensed, the robot was instructed to wait one second to let the truck pass and then enter into the loop.

Figure 8: Original and Cropped Grayscale Truck Image

6. License Plate Detection

With navigation of the simulation and obstacle procedures in place, the next key challenge was to detect and process license plates for CNN plate prediction. Our approach to detecting plates can be described in four steps; determining if a frame contains a clear plate, finding plate bounds, plate

segmentation from a frame and extracting individual characters. In this section we will touch on the process of plate detection, bound determination, plate segmenting and character extraction.

Prior to discussing our solution, alternative methods we considered for plate detection included SIFT and homography to determine plate location and existence within a frame, however both algorithms had cons which did show up in our solution. Namely SIFT was comparatively computationally expensive and plates would not always contain consistent key features as plate numbers changed, thus we did not attempt SIFT as it was assumed to be over complicated and hard to tune/debug. Homography was another option but again it seemed over complicated and would require extensive research to implement correctly.

The method used to detect plate existence within frame was relatively simple, relying on a key feature only present when plates existed, that being the existence of black pixels in a frame. We found that only 2 instances in the simulation contained black pixels, the P# on license plates and the truck (windshield/tires) circling the inner loop. Fortunately from the outer loop the truck could not be seen clearly and in the inner loop the car would not be visible due to our specific navigation. With this information and numpy.sum() it was simple and efficient to set a threshold and count the number of black pixels in a frame, with this being greater than the threshold implied plate existence. Furthermore, this method allowed us to tune the clarity and number of plates detected, as larger black pixel thresholds meant only proximate plates would be captured and lower thresholds allowing for more capture instances. This was carefully tuned and we found the value of 10-20 black pixels a good balance for detection.

Next, plate bounds determination from the frame relied on a simple calculation of the center of mass of P# which involved a cv2 binary mask and the numpy average function. With the centroid we knew the location of the plate center, which exists within the plate bounds. Thus if we could move outward from this center until an evident change occurred we could find the left, right, top and bottom bounds of the plate. This evident change or key feature was a characteristic of the parked car, ie the blue car color. Initially we tried a simple mask in grayscale to find this blue color, however due to changing brightness around the course this was not consistent, thus we decided on a color mask in BGR to isolate blue (refer to appendix) and then applied the cv2 medianBlur function to remove any noise, this consistently yielded the blue car color adjacent to the plate left and right bounds as seen in figure 9. Thus the algorithm to get the bounds was to first move left and right from the centroid of the P# feature until blue was reached. Then to get the top and bottom bounds start at (X_bound_right+5, Y_CM) and search up and down until blue is no longer detected. This worked consistently and was a simple color mask approach to getting the plate bounds.

Figure 9: Binary Mask of Blue Car

With the plate bounds, cropping out the plate for character extraction was simple as seen below in Figure 10 Image 3 and as seen by other plates in the appendix.

Figure 10: Black Mask of P#, Blue Mask for Parked Car, Resultant Plate

Lastly with the cropped plate, extracting the characters was relatively simple as we only needed to find manual bounds based on percentages of the image width and height. Then for resizing we utilized the imutils library to closely maintain the aspect ratio and provide a consistent character image for the CNN to process. Please refer to the appendix for visualization of character extraction.

7. Convolutional Neural Network for Character Recognition

The CNN was created using the Keras training framework to recognize the license plate characters on the parked cars while driving around the loops. Note that the CNN layer options, summary and confusion matrix can be found in sections 9.4 and 9.5.

7.1 CNN Structure

We first started by looking at the structure of the license plate CNN designed in lab 5 and seeing if we could modify it so that it could work for this application. First, the file paths were set up so that the training images in google drive could be used in colaboratory. This training data was then read into the colaboratory notebook using cv2.imread and the images were stored in a set. From there, the images were cropped and cut into their four plate characters. These characters were stored in an X_dataset where they would be normalized by dividing by 255. To make the corresponding Y_dataset, a function was created to obtain the correct plate characters based on the plate name. Next, the characters in Y_dataset were matched with an index number from a dictionary with an index corresponding to the letter in the alphabet and numbers from zero to nine. The dictionary contained all letters in alphabetical order and then numbers zero to nine in increasing order. For example the letter D would be mapped to an index of four. The Y_dataset was then converted using one hot encoding. One hot encoding is a process by which the Y_dataset is converted into a form that could be provided to the machine learning algorithm to aid in better prediction. Figure 11 shows the image and label mapping.

After both data sets were in place, we started setting up the model training. A learning rate of 1e-3 and validation split of 20% was used initially to test the model. Different types of layer options for training the model were tested. This is further described in section 7.3. Note that it is very important to reset the model weights before every train. The model was chosen to have the following training parameters: loss = categorical_crossentropy, optimizer = optimizers.Adam, 80 epochs and a batch size of 36. After training,

the model loss and accuracy were graphed based on the validation split. These can be seen in Figure 12. An additional tool used to help see the progress of the model was a confusion matrix. This can be seen in section 9.5 along with the model summary. After the CNN was created it was just a task of getting more plates to increase the accuracy of the model.

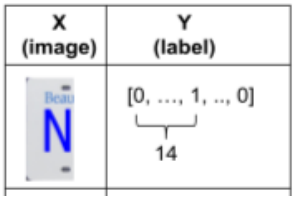


Figure 11: Conversion of Plate Image to Label

7.2 Data Generation

At first we decided to use the plate generator that Miti created and trained the model based on that. We then thought that we would need to add noise to these plates because they were ideal plates and did not look like those of the real simulation. Instead of going through the work of modifying the ideal generated plates with noise or stretching, we decided to save the plate images generated from the actual simulation. This would be a more useful data set to train on because they are of the same format as the plates in the competition. The plates were obtained by saving the six outerloop plates as the robot drove around with every simulation. These plates were then renamed to match their plate characters and imported into a google drive folder which the colaboratory CNN trainer had access to. Examples of these plates can be seen in section 9.3. To save time, instead of changing the image names by hand to match the plate numbers, an algorithm was implemented to assign the plate characters to the image using the plate name when the world was generated. In total 1433 real simulation plates were generated meaning that 5732 character images were used in training the model.

7.3 Training

With the CNN structure in place and the simulation license plates generated, we started training the model and testing different parameters. The first train was around 90% accurate but that was with 400 plates. As the number of plates increased, so did the accuracy. By the last train, the model was 98% accurate which can be seen in the graphs in Figure 12. The confusion matrix along with the model summary of the last train can be seen in section 9.5. We noticed that the model sometimes confused C's and G's so we targeted those characters by adding more plates that contained them.

Two layer options were experimented with while training the model. These two layers can be seen in section 9.4. Layer option two was constructed from the model used in lab 5 and layer option one was derived from option two but with some layers removed. The removed layers were done so by trial and error and exploring online sources. The two options presented were found to work the best out of all ones tested, with the second option working slightly better than the first. Note that while training with these layer options we used a manual validation split so that we could vary the number of plates used for validation versus training. For the model used in the competition, a validation split of zero was chosen so that all plates could contribute.

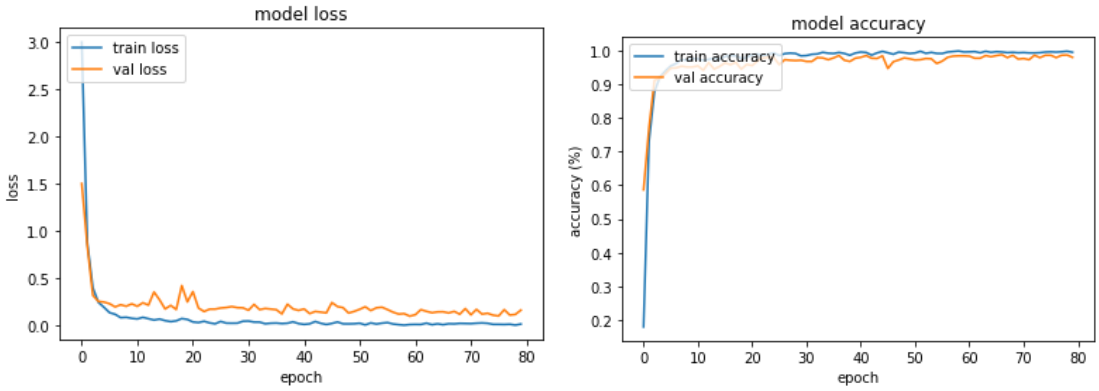


Figure 12: Model Loss and Accuracy of Model Used for Competition

7.4 Model Save/Load and Integration

After training, the CNN model needed to be exported from colaboratory and downloaded into the correct folder where it could be accessed by the controller.py file. Pip, which is a standard package manager for python, was used to install H5py. H5py stores huge amounts of numerical data such as models. Note that version 2.10.0 of H5py was used to save the model to google drive. Once downloaded, the model was moved into the cnn_trainer/Models folder where it could be accessed from the controller file.

Many problems were encountered trying to integrate the model into the python file. The following steps helped solve these problems. First the proper tensorflow libraries were imported which can be seen in section 9.4. Then we created a global instance of the tensorflow session and computational graph. The model was then loaded in as part of a class. The prediction was done using the global session and computational graph. After these steps the model was able to interface with the controller.py file.

After the model was interfacing correctly with the python file, we tested it to see if it detected the plates correctly. At first it was completely wrong and getting values confused that did not look similar. We found the problem to be that we trained the model on images that were read using Image.open() while we used cv2.imread() to read images in controller.py. Once the model was retrained using the cv2 image reader it started predicting correctly.

8. Strategy and Final Decisions

In our final considerations it was decided that a quick outer loop and being able to read the license plates took precedence. Once those were implemented, driving around the inner loop consistently could be considered. A key factor that aided in this decision was that the inner loop carried more risk in terms of losing points by driving off the track or hitting the truck. Our outer loop and plate recognition was finished a couple days before the competition so we decided to attempt the inner loop. The inner loop driving was working, but not nearly as reliable as the outer loop. We toyed with the idea of only completing the outer loop and removing the inner loop code but ended up deciding the inner loop was worth the risk and drove the full course.

9. Appendix

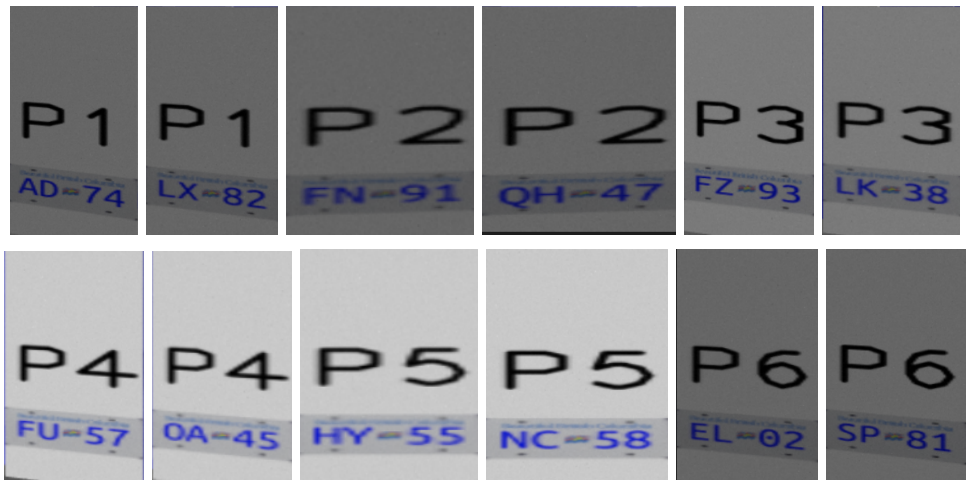
9.1 Links

Github:
https://github.com/moolanp/ENPH353
CNN Trainer Colaboratory:
https://colab.research.google.com/drive/16JR8RNUjoRyi630QqL-DXlOjq8bZyvTq?usp=sharing
All License Plates Detected Video:
https://drive.google.com/file/d/14h1b8z-23ZIXDSR_0CsSZI4E9u9mINZp/view?usp=sharing

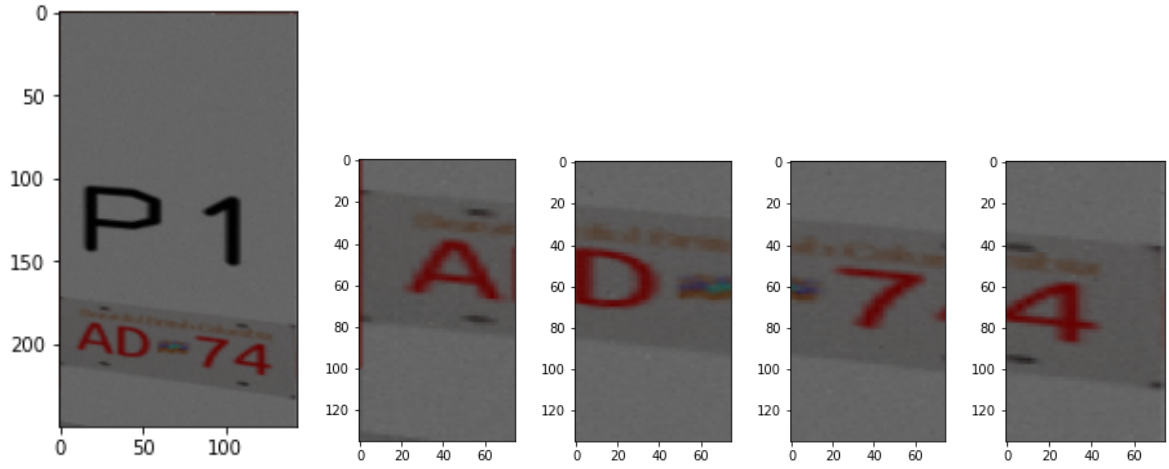
9.2 Color Masks

Crosswalk: cv2.inRange(img, (0,0,255), (0,0,255))
Pedestrian: cv2.inRange(img, (50,0,0), (55,50,50))
Grayscale: cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
Blue Car: maskblue = cv2.inRange(cv2_img, (100, 0, 0), (255, 100, 100))
blue = cv2.medianBlur(maskblue,7)
P# Mask = cv2.inRange(gray_img, 0, 0)

9.3 Plate Examples



Note the different brightness levels due to the position of the parked car and angled view of the plate.



Note the original image has its characters extracted then resized to (75,135) images.

9.4 CNN Libraries & Layer Options

Libraries Imported:
import tensorflow as tf
from tensorflow.keras import models
from tensorflow.python.keras.backend import set_session
from tensorflow.python.keras.models import load_model

Option 1 (Tested):

```
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Conv2D(64, (3, 3), activation='relu'))
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Flatten())
conv_model.add(layers.Dropout(0.5))
conv_model.add(layers.Dense(512, activation='relu'))
conv_model.add(layers.Dense(36, activation='softmax'))
```

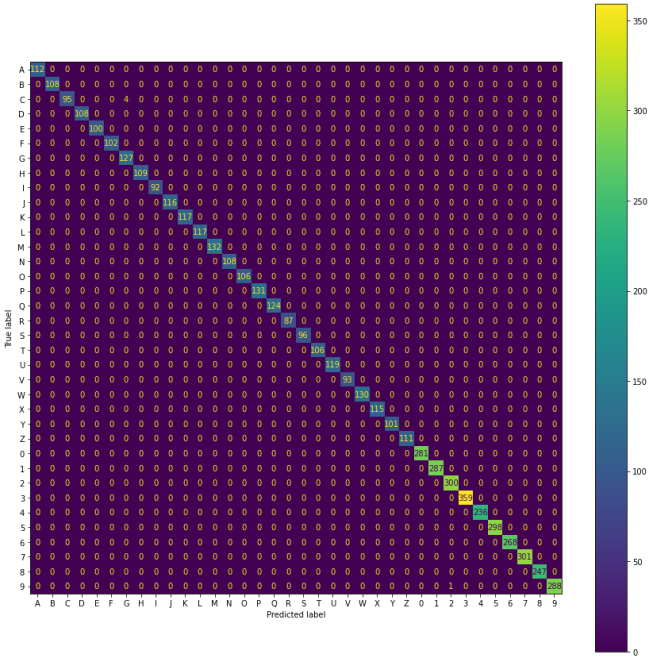
Option 2 (Used):

```
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Conv2D(64, (3, 3), activation='relu'))
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Conv2D(128, (3, 3), activation='relu'))
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Conv2D(256, (3, 3), activation='relu'))
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Flatten())
conv_model.add(layers.Dropout(0.5))
conv_model.add(layers.Dense(512, activation='relu'))
conv_model.add(layers.Dense(36, activation='softmax'))
```

9.5 CNN Model Summary & Confusion Matrix

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 133, 73, 32)	896
max_pooling2d_4 (MaxPooling2	(None, 66, 36, 32)	0
conv2d_5 (Conv2D)	(None, 64, 34, 64)	18496
max_pooling2d_5 (MaxPooling2	(None, 32, 17, 64)	0
conv2d_6 (Conv2D)	(None, 30, 15, 128)	73856
max_pooling2d_6 (MaxPooling2	(None, 15, 7, 128)	0
conv2d_7 (Conv2D)	(None, 13, 5, 256)	295168
max_pooling2d_7 (MaxPooling2	(None, 6, 2, 256)	0
flatten_1 (Flatten)	(None, 3072)	0
dropout_1 (Dropout)	(None, 3072)	0
dense_2 (Dense)	(None, 512)	1573376
dense_3 (Dense)	(None, 36)	18468

Total params: 1,980,260
Trainable params: 1,980,260
Non-trainable params: 0



Model summary and confusion matrix of the most recent trained model which resulted in 98% accuracy.