Christian Malherbe 87908679
Alexi Garbuz 97063432

ENPH 353 Final Report

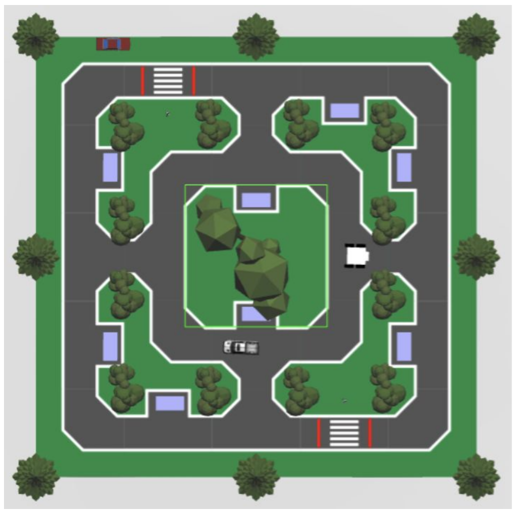

Table of Contents

Overview

The following report illustrates the methods we used to create an autonomous robot capable of driving around a track, obeying traffic laws, and reading license plates. Our strategy utilizes classic computer vision techniques as well as aspects of machine learning in the form of a convolutional neural network. To give a high level overview of our system, our robot uses PID line following for driving, computer vision for detecting pedestrians and extracting licence plates, and a convolutional neural network for determining the characters of each license plate. Throughout the entire process our main algorithm subscribes from and publishes to several ROS topics while our robot performs in a Gazebo simulation.

To score full points on the competition, our robot was required to drive around the outer ring and inner ring, log all the cars and not lose any points from collisions. To balance the risk that came from losing points due to collisions in the inner ring with scoring as many points as possible, and to increase the speed of our run, our strategy was to only traverse one lap of the outer ring, logging all 6 cars there, and then to end our run.

Software Architecture

We used three repositories to structure this project: *ENPH353/2020T1_competition, enph353-controller and ENPH353-License-Plate-CNN*, all of which were located in the ~/ros_ws/src/ directory.

The first repository, *ENPH353/2020T1_competition*, is the pre-made competition repository that contains the robot configuration files as well as the main world to be run in Gazebo. We did not edit this repository at all.

The second repository, *enph353-controller*, is where we stored the bulk of our logic and control algorithms. This repository has four files in its ./src/ directory: *driver.py, main_control.py, plate_detector.py* and *ImageCapturer.py*. The driver file contains our code for PID line following, and pedestrian detection. Next, the plate detector file contains our methods for detecting and extracting license plates from the simulation. Also, the image capturer file contains code to capture and save images from simulation which we used to help write our code, but is not called on in competition. Finally, the main control file is a ros node where all of our robot's main logic happens. This is where the robot creates subscribers and publishers, and where we decide what actions to take based on input from the simulation camera. The main control file communicates and utilizes all of the other files in this repository to accomplish the robot's functions such as driving, pedestrian detection, finding license plates, etc. This repository also contains the trained finalized convolutional neural network model.

Lastly, the *ENPH353-License-Plate-CNN* repository is where we store the data, data generation scripts, and training scripts that all contribute to our convolutional neural network model. We have several folders dedicated to holding training data, augmented training data, simulation data, and processed simulation data. We also have a few folders that were used to experiment with CNN's involving only letter prediction or only number prediction. Finally, a copy of the model gets saved in this repository when we run the *model_train.py* script.

Robot Control and Movement

Driving

We decided early on in the development process that we were going to use a PID line following approach to control our robots movement around the course. We created an initial line following design that we used in time trials, which we eventually modified and further developed into our final design. Our run starts off with a hard-coded turn to orient the robot onto the track, and then the PID algorithm begins.

The first approach we used for the PID involved tracking the inner side of the rightmost white line marking the right side of the lane. We used the distance from the center of the camera frame (which signified the center of the robot) to the right white line as our measure of how close or how far we were from the center of the lane. We did this by having a 'target' value of how far the center of the frame should be from the right white line, and adjusting the angular and linear velocity depending on if the value we were reading was greater than or less than this target value.

On top of this basic PID structure, we also developed a method of remembering what the last velocity command was (straight, left, right) and using that when determining the new cmd velocities to use. For example, if the robot's last command was "straight" and the current command is "straight", keep going at the same speed. On the other hand, if the last command was "straight" and the current command is "left", rather than sending a zero linear velocity and a high angular velocity, we send a small linear and angular velocity. This was implemented to help correct the robot's choppy movement, where it would almost tip over if sent a command to abruptly change the velocity.

This first approach at the PID worked reliably and had no problems going around corners, although it was a little bit slow and lacked sophistication of proportional velocity changes. However, once we started trying to combine licence plate detection with our PID, we began noticing strange driving behaviours, where the robot would go off course unexpectedly. We still don't understand actually why this occurred, but we believe it could be due to the fact that all the image transformations, convolutions, and calculations used in the process of looking for license plates were taking a non negligible amount of real world time to complete, and this meant that we were skipping on frames being received from the image feed (because we couldn't keep up with them). Whatever the cause of the issue was, we knew it had to be solved, and since there were better ways to implement our PID algorithm, we decided to rewrite it.

The logic of how our PID was being called was kept the same, but instead of using the last command and pre-set velocity values, we decided to switch over to using a more classical PID lane following algorithm with a proportional and derivative gain constant. We used the same target value as before, judging our error off of how far the center of our robot was from the right white line against the target value. We used this to determine our angular velocity, and then subtracted a proportional amount off of the linear velocity as well. After tuning the parameters we were able to produce a fairly smooth PID line following algorithm, which had a feature where the robot could even move backwards to overcorrect the error. The PID worked very reliably for the straight sections of the outer ring; however, it had difficulties when it came to the corners, as we will further discuss below.

We initially attempted to implement a PID that could be used around the entire outer ring, but with the algorithm we were using, we had a lot of difficulty in adjusting the parameters to work perfectly for the straight sections, as well as the corners. To deal with this, we decided to instead tune the traditional closed loop PID to guide our robot on the straight sections, and to write a hard coded corner handling function to transition our robot from straight section to straight section.

As our robot moved around the track, it would monitor whether there were grey pixels in the bottom left corner (indicating the road turning off to the left), and also if there were green pixels directly ahead (indicating that the road was ending). If both these conditions were met, we would enter the turning block of code. The robot would then turn at a fixed angular velocity until it saw the turn was ending (indicated by being able to see the edge of the track on the left again), in the process, aligning itself with the next section of the track. Next, it would drive straight to move it onto the following portion of the track, and lastly, we would exit the block of code and begin to use the PID again.

Chained together, all this software allowed our robot to reliably start by turning itself onto the track, PID control it's way along straight stretches of the road and take turns. This meant it could navigate around the full outer ring to make it back to where it started, seeing six of the cars, and earning points for completing one lap.

Pedestrian Detection

Our pedestrian detection strategy can be broken up into two parts: crosswalk detection and pedestrian crossing. Our robot would detect if it was at a crosswalk, and then do the appropriate actions required to detect when it was safe to go through the crosswalk.

For detecting whether the robot was approaching the crosswalk or not, we used the red rectangle that comes before the crosswalk as our signal. Our first approach towards detecting this red line was to simply look at the red channel of the pixels for a certain height range in the frame, and if this value crossed a certain threshold it meant we were at the correct spot to stop the robot, and if not the robot would continue to PID line follow. This method for the most part worked pretty well; however, there was the rare occasion when the robot would not detect the red line and drive through the crosswalk. Since this could possibly lead to a large deduction in points, we decided to improve our method for detecting the crosswalk.

Our new strategy for detecting the start of the crosswalk still involved using the red-line as a marker, but this time utilized an HSV mask. For each frame received, we would convert it into HSV space, apply a mask to pick out the red of the crosswalk as well as apply some dilations and other transformations and then contour the shapes picked out by the mask. If there was no red detected, we know right away we are not at a crosswalk. If there were contours, we would take the one with the largest area and apply an openCV function to find its bounding rectangle. We would then use the y-value of this rectangle (corresponding to the left upper corner of the rectangle) to determine if the red line was above a certain threshold height in the frame. If it was, this indicated that we were at the crosswalk, and we would stop the robot in preparation for the next part of the algorithm.

Once the robot had stopped at the crosswalk, it was time for pedestrian detection. Our main method of detecting when the pedestrian had crossed was by using openCV background subtraction. The high level overview of how we accomplished this is as follows: stop the robot at the crosswalk and wait for one second to ensure we get a good background image. Then send that image (cropped to the area of focus which is the middle of the crosswalk) to the global subtractor object which is of the type: `cv2.bgsegm.createBackgroundSubtractorMOG().`
This will allow the subtractor object to use this image as a 'background' template. After obtaining the background template, we can loop through each new camera frame we receive and use the subtractor to apply a mask to the image where white pixels represent changes from the template image. If there were enough white pixels in the image, we know a pedestrian must have moved across the frame, and so we knew it was safe to cross. Since the image is cropped to the center of the crosswalk, we knew to wait 3 simulation seconds before continuing our PID through the crosswalk (this way we would never hit the pedestrian mid-cross). Also note that once we had determined the pedestrian had crossed, we had to re-initialize the subtractor object so that it would not use all of the previous images as templates for the next crosswalk, thus making this method for pedestrian detection re-usable for each crosswalk.

The following image illustrates the process of using background subtraction by showing the pedestrian filled with white pixels in contrast with the original image. Here, the white pixels are representative of parts of the image that are different from the background image.

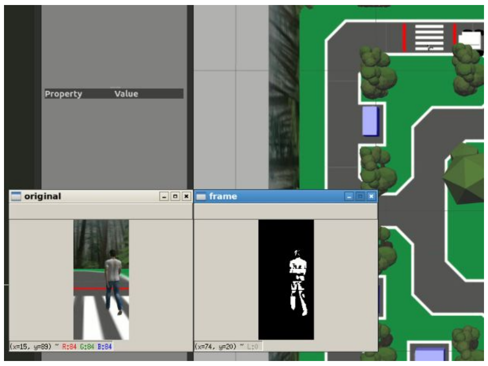

Figure 1: Original frame contrasted with frame after background subtraction mask is applied

Licence Plate Detection and Reading

Parked Car Detection

A key step in registering license plates from parked cars for our agent was being able to recover the license plate from the image feed in order to pass them into the neural network we developed. We used a two step process to accomplish this.

First, we determined whether there was a parked car nearby and visible in the robot's image feed. If there was indeed a car in the feed, we would then proceed to crop the plate out from that image. When we first started the project, we suspected that it might be necessary to drive extra close to the car to be able to see the plate, but as we developed our agent further, we discovered that it was enough to just drive by the car and capture an image of the plate while simply using our PID to keep our robot straight.

As far as determining whether there was indeed a car in the image feed, we used a classical computer vision technique. Our first attempt at this problem was to convert the image feed to HSV space, then apply a mask to pick out the blue components of the image. We then counted the number of blue pixels in the frame, and essentially judged whether there was or was not a car in the image based on whether there were enough of these blue pixels.

We encountered two problems with this technique. Firstly, the way we counted blue pixels in the code seemed to be a very compute heavy operation. Including the method of counting blue pixels had a detrimental impact on our robot's ability to use the PID to track the road. This was solved by switching to a better optimized method using NumPy arrays instead, which are much more efficient when it comes to many array operations. Still, we realized that counting the blue pixels alone was not sufficient because of the fisheye lens of the robot's camera.

As our agent came around turns, it was possible for it to have two parked cars in it's vision. Since the left car was in the corner of the robot's vision, it would appear much larger, and the blue pixel count would pass the set threshold. In order to get around this, we created an additional requirement that there needed to be enough grey pixels in the frame that were the same shade as the back of the car in the frame. Thus, if we had enough blue pixels and grey pixels together, we knew that there was a car in view for us to begin trying to extract a plate from.

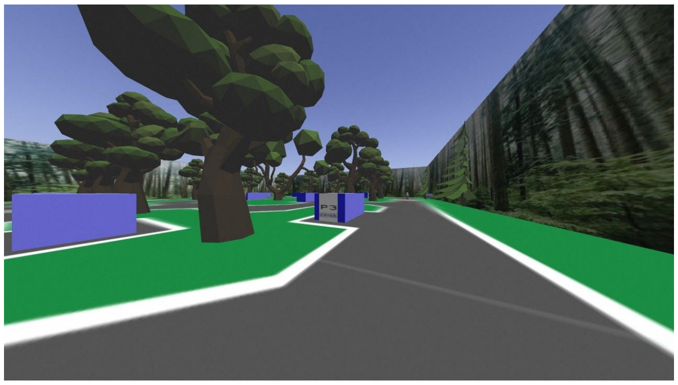
Figure 2: Image feed seen by the robot as it takes a turn

Licence Plate Extraction

Using the procedure described above, we could determine with perfect accuracy whether there was a parked car close to our agent. From there, we needed to extract an image of the license plate to pass on to our neural network. Our first attempt to do all this involved using SIFT. We tried to match a template image of a license plate to the license plate in the image.

We pursued this strategy briefly, but were very unsuccessful with it. One of the main issues was likely that SIFT was identifying key points from our template such as the unique characteristics of the letters in that plate, but the letters in our image feed were not necessarily the same, so there were not nearly enough key points to match. Additionally, it came to our attention that SIFT is a very compute heavy process, so we abandoned that approach and opted to instead use classical computer vision techniques to recover the plate from the image.

To begin, we converted the image to HSV space and applied a mask to highlight grey pixels that were the same shade as the back of the car and of the license plate. Finding the exact values to use in this mask was a difficult task, but it was very useful to use a Colaboratory notebook, and tweak the values by hand while constantly showing the images to the screen.

We developed a mask that highlighted the entire back of the car, but it was impossible to trigger the back alone, our mask also picked up shades of grey where the white lines met the road, grey from the pedestrian and truck, and grey from the exterior background. We used several rounds of image erosion to eliminate this noise in the image, and the result was a new binary image with only pixels on the back of the car being triggered.

Figure 3: Masked image of the back of the car after erosion

We found the centroid of this shape to get the x and y coordinates of a pixel somewhere on the back of the car. We then looked back at our original image, and applied a blue mask to reveal only the blue from the car. Using the centroid we had previously located, we identified the left edge of the back of the car by finding the first pixel to the left of the centroid that had been triggered by the blue mask. We found the right edge using the same strategy, and located the top and bottom of the back of the car by traversing these edges to their ends at the top and bottom. This allowed us to crop out the entire back of the car.

Once we had identified the back of the car, finding the plate turned out to be a much larger challenge than we anticipated. The main hurdle came from the fact that when we filtered in HSV space, varying lighting at different sections of the track caused there to be an overlapping range of values that the plate pixels could occupy, and that the non-plate back of the car pixels could occupy. This meant that by using a static HSV mask to identify the plates, we would either miss the plate or we would pick up on both the plate and the non desired pixels from the back of the car.

To reconcile this issue, we used a dynamic mask based on the car we were investigating. Regardless of what shade of grey the plate and back of the car were, it was always true that the plate was a darker grey (lower hsv value) then the rest of the car, so for each image we identified the average HSV value of the pixels near a point at the upper middle section of the image, which we took to represent the HSV values for the "non-plate section of the back of the car". Due to how reliable our method was for extracting the back of the car, it seemed that this upper mid point procedure always yielded an average that was indeed representative of the "non-plate section of the back of the car" and nothing else.

Given the HSV values of the "non-plate section of the back of the car" pixels, we then applied a mask to pick out all the grey pixels that were darker than what we found the back of the car to be. We combined this with a blue mask which picked out the blue letters on the plate, and used several rounds or dilation, erosion and re-dilation to find a band of white in the binary image representing where the mask is in the image.

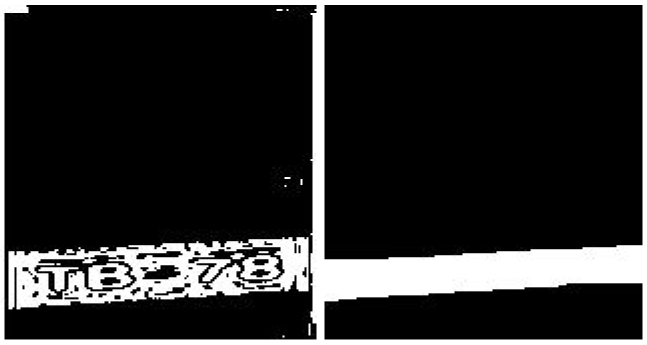

Figure 4: Masked image of the back of the car (left), same image after transformations (right)

Similarly to the strategy described above for isolating the back of the car, we traversed this image to find the four corners of this parallelogram. We then applied a perspective transform to turn that parallelogram into a perfect rectangle. The result was an image of slightly varying dimensions, but that was always a very clear shot of the license plate.

Before sending the images into our convolution neural network, there was some preprocessing we had to do. First, we resized the plates to be of size 256 x 60 pixels. Then, we cropped each plate into individual letters using pre-set pixel values. Each letter was then turned into binary and stored in an appropriate array. Converting the letters to binary took out any issues that would have come from different lighting or shades of colours when training and predicting using our convolutional neural network. The dataset that stored the letters was then normalized and reshaped to match the dimensions of our training data. Lastly, the dataset was sent into a predict function, which we will talk more about in the *Convolutional Neural Network for Reading Plates* section below.

Convolutional Neural Network for Reading Plates

We used a convolutional neural network to predict characters. We will first discuss the data generation, training and model used for our CNN, and then we will explain how we incorporated the model into our main algorithm for detecting and reading license plates.

For training data generation, we used the licence plate generator that was provided to us in lab 05. With this, we generated a large amount of licence plates which we stored in our *ENPH353-License-Plate-CNN* repository in the *data* folder. We then zoomed out on the plates followed by resizing to 256 x 60 to make the plates look less sharp and more like the data we were getting from simulation. Then we cropped the plates into their individual letters and performed a number of augmentations on the letters including shifting, blurring, zooming, and changing to binary. As previously mentioned, turning the data from RGB to binary helped prevent any difficulties to do with lighting, different shades, different colours in the background, etc. These augmented letters were saved to the *data-2.0* folder along with appropriate image naming in order to keep the information we needed for labelling which involved the character itself.

The next step was creating a validation set. We accomplished this by saving licence plates captured from our robot in simulation and manually labelling the plates (stored in *simulation-data*). These labelled images could then be sent into our script to pre-process them before their use in our model. The *simulation_data_generation.py* script would resize the plate images, crop out the letters, turn the letters to binary, and then save them to *simulation-2.0*.

After we created the necessary training and validation data, we could move into the actual script for creating the model. This was done in *model_train.py* within the same repository. The structure of our training script is very similar to that of lab 05. However, rather than processing full plates it only has to process individual letters. The script starts off by one-hot encoding the training and validation data according to character. Before sending the data into the model we normalize it. The actual model itself is a sequential model with two convolution layers and two max pooling layers, followed by a dropout layer, a dense layer of 128 nodes, and finally a dense layer of 36 nodes.

For testing the performance of our convolutional neural network, after fitting the model to our data we graphed the training and validation losses as functions of number of epochs as well as the training and validation accuracies as functions of number of epochs. We also plotted confusion matrices to see the specific weaknesses in our model. The graphs helped us visualize if our model was overfitting and the overall performance of the model. The confusion matrix showed us which letters/numbers the model was most often confusing for one another.

We tried many different combinations of layers for our model, as well as played around with how we augmented our training data; however, we were getting quite poor accuracy from our model. We decided to try creating separate neural networks for letters vs. numbers, but even then the accuracy was still low. After having tested several different models and tried with many different data sets, we decided to switch up our approach for training the model. We found that the licence plate data we were getting from simulation was actually quite good and consistent, and thus we made the decision to collect more data from simulation and use that to train our model. Even after a relatively small amount of plates had been collected (in the 200 range), we tested our model after having trained it with purely simulation data, and the results were exponentially better than before. Our accuracy was over 90%, our losses were minimal and our confusion matrix almost resembled a perfect diagonal. After some more tuning of layer parameters we settled on a model that presented results as in the images below:

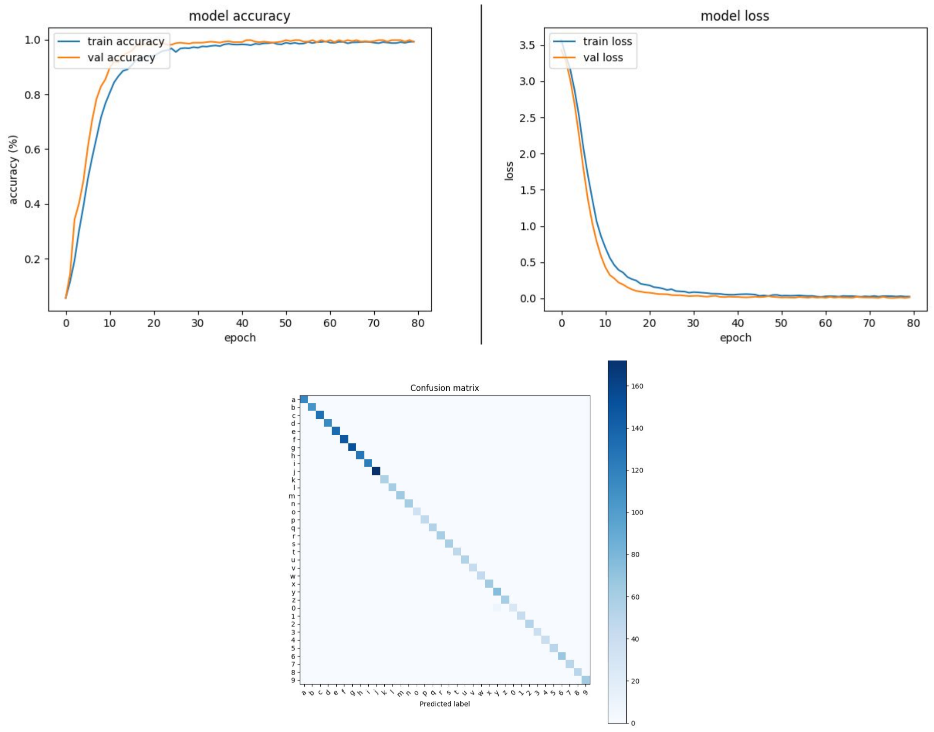

Figure 5: Model Accuracy and Model Loss Graphs, and Confusion Matrix of final model

Switching over to our main control algorithm, once our model was called to predict the characters of a certain licence plate, it would return an array of size 36 for each character with values representing the probability of the character belonging to each class. Although the accuracy of the CNN was very high when used in simulation, there was the rare occurrence where it would mix up a letter and number. To avoid this situation, we added a step after the model makes its predictions to separate the prediction array depending on whether the character should be a letter or number. The first two characters of the licence plates are always letters, and the last two numbers, so we could use that knowledge to splice the prediction array and find the highest prediction according to whether it is a number or letter. We then gather and publish these predicted values to the license plate topic.

In addition to publishing the license plate number, our robot was required to publish it to the correct stall ID number as well. The ID of each car was posted above the license plate so we considered using character detection to read that as well, but since our driving control involved always navigating the course by traversing the outer ring in a counterclockwise circle, our robot saw each parking ID in the same order every time it ran the course. As such, we used simple variables and counters to keep track of which cars we already published, and using that information, we knew which stall to publish each plate as soon as the plate was available.