UBC Engineering Physics 2019

UBC Engineering Physics License Plate Detection Competition

Final Report

Jackson Gayda 90408758
Tianna Hudak 62188750

THE UNIVERSITY OF BRITISH COLUMBIA

03-10-2019

Contents

1. Introduction

1.1 About ENPH 353

ENPH 353 is a self-directed project course that allows students to practice some of the most recent techniques and developments in software. The first six weeks of class were filled with interesting lectures on topics to help with competition development. These topics included Python, CV and Jupyter Notebook, ROS and Gazebo, and neural networks. Neural network lectures were particularly interesting, as we learned about convolution neural networks (CNN), image augmentation, Q-learning, gradient descent, and backprop among other topics. In addition to lectures in the first six weeks, we completed labs to help complement the lectures and gain skills for competition work. Labs included:

1. Set-up Linux Environment
   - This lab was for students to set up Linux on our systems. While Xubuntu was recommended, we were already running Ubuntu 18.04 and decided to continue with this OS.
   - Installed Gym-Gazebo and setup the environment.
2. Line Detection from Video
   - This lab was helpful to familiarize with Python and openCV.
   - The goal was to program a line detection algorithm from video input. To do this we utilized contours and moments.
3. Line Following with CV and PID, Building Robot with XML
   - In this lab we set up a ROS workspace, created a simulated track for line following, built a differential drive robot with XML, and integrated computer vision based line detection in a ROS node for a PID line follower.
4. Character Recognition with CNN
   - Used a CNN to identify characters on license plates to prepare for competition work. We explored CNN and Keras.
5. Line Following using CV and Reinforcement Learning
   - Trained a reinforcement learning robot to line follow in gym-gazebo.

The remaining six weeks were used to work on the competition code and strategy.

1.2 Competition Overview and Rules

A team consists of two engineering physics students who control one robot on the track. Each team participates in a four minute round (this rule was initially 2 minutes) where they return the characters of the license plates and the locations of the plates on cars parked in the simulated world. For each correct license plate/location pair submitted, teams are awarded $+6$ points for outside cars, or $+8$ points for inside cars.
Teams must also follow the rules of the road. You must maintain at least two wheels inside the lines of the road at all times (-2 points), must not collide with any other vehicles (-5 points), and you must not run over pedestrians (-10 points). You also receive 5 points for making a successful lap around the course. Below is a summary of points:

Action	**Associated Points**
Drive a complete lap of outside track	+5
Return correct license plate characters (outside track)	+6
Return correct license plate characters (inside track)	+8
Return correct license plate characters but incorrect location (outside track)	+3
Return correct license plate characters but incorrect location (inside track)	+4
Run over pedestrian	-10
More than two wheels off the road	-2
Collision with other vehicles	-5

Table 1.1: Competition point summary.

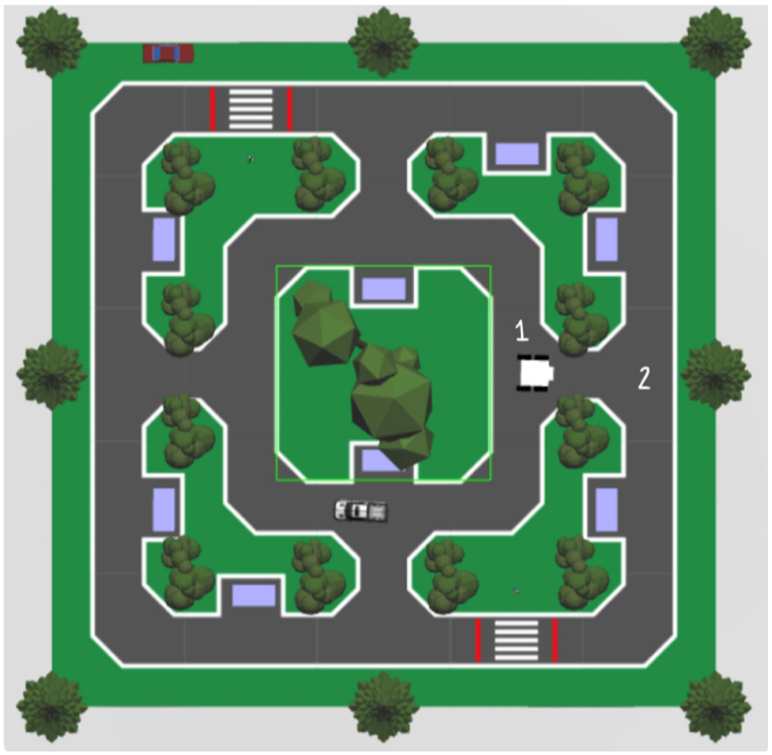


Figure 1.1: Robot course simulation.

2. Data Processing Architecture

2.1 Overview

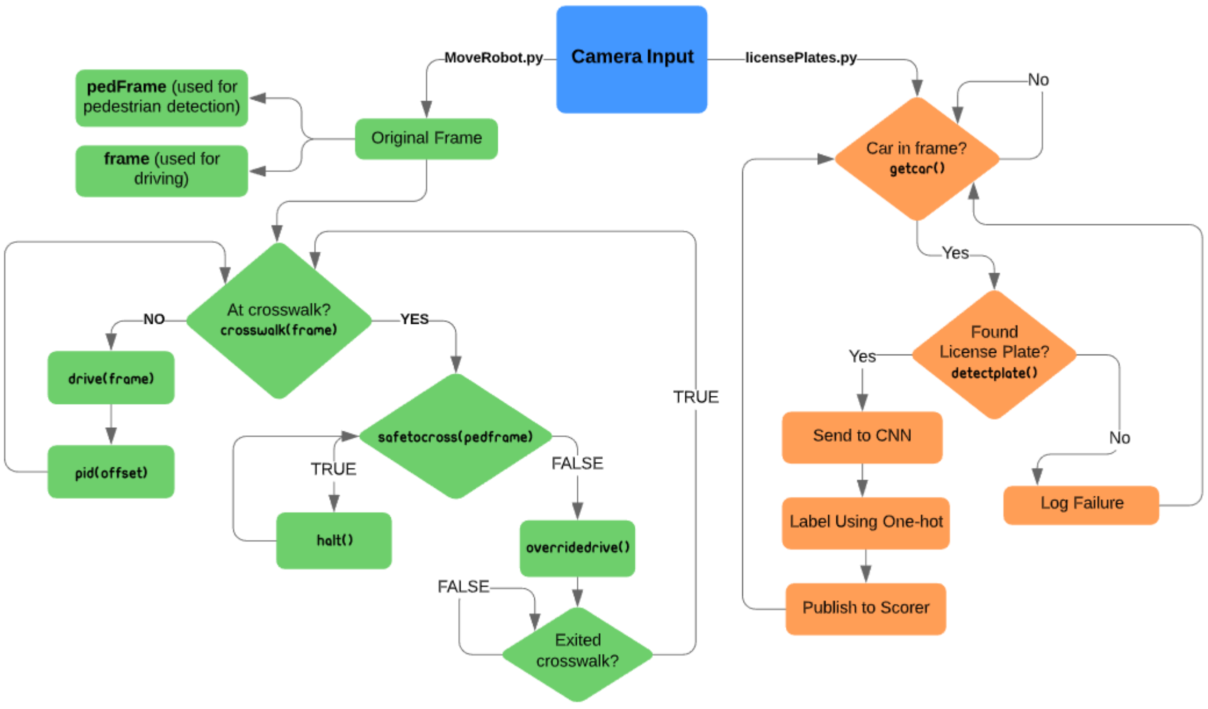


Figure 2.1: Data Flow.

2.2 Functions

1. `def crosswalk(frame)`: uses red masks and contours to determine if a crosswalk is in the frame.
2. `def drive(frame)`: uses white masks and contours to determine an offset from the outside line for the robot to follow.
3. `def safeToCross(pedFrame)`: compares the similarity between two images to determine if a pedestrian is crossing the road.
4. `def PID(offset)`: takes the offset from `drive` and either drives straight or turns slightly according to the offset.
5. `def getCar(cropFrame)`: uses blue masks and contours to determine if a car is in the frame.
6. `def detectPlate(cropFrame)`: uses connected components and cropping to take characters from the license plate to send to the neural network.

3. Implementation

Our project can be split up into two main functions: a driving controller and licence plate detection. We aimed to complete this project in a linear format and therefore began working on our driving first to ensure that our driving was consistent. This way, our license plate detection method would not be limited by our ability to drive smoothly around the track. Our implementation of these two functions are detailed below.

3.1 Driving - moveRobot.py

As our strategy, we chose to focus on the outer ring of the course, and have the inside ring as a stretch goal. Initially, our robot spawned at the inside intersection (position 1), as referenced in figure 1.1, however, we set our robot to spawn for debugging and training at position 2 in figure 1.1. As we were focusing on the outer ring of the course, we decided that we would use the outer road line of the course as our guidance to be able to do a complete lap of the track and reliably stay on the road.

3.1.1 Driving Setup

We adjusted the position of our robot within the simulation while monitoring its camera feed to find the perfect orientation in which it was heading directly straight on the road, noting the pixel at which this occurs. We then cropped out the bottom corner of the robot's frame so that the only object that was in frame was the white line corresponding to the outside of the track. Utilising CV2's contours function and white masks, we were able to find the centroid of this cropped outer line. Using this centroid coordinate, we had the x pixel corresponding to a robot that would be driving perfectly straight. We called this our `refPixel`: the pixel value of the desired outer line centroid coordinate that our robot will reference when comparing it to its current centroid coordinate. We gave our robot a small threshold number of x pixels less than and greater than our `refPixel` to allow for slight misalignment to ensure that the robot did not constantly need to turn (reference Driving Limitations section for details). Our robot was able to safely and reliably navigate the outer course using this method.

To determine if the robot was misaligned, we looked at the current centroid coordinate (`x_coord`) and compared that with the `refPixel`. We then passed `refPixel - x_coord` into our turn function (`pid()`). The passed value was a positive or negative integer value representing the pixel difference between the desired `refPixel` and the current centroid pixel. Our turn function then looked at the sign of this difference, and wrote this difference to the angular-z velocity input for the robot. The sign dictated which way to turn. If the `refPixel - x_coord` value was positive, the robot turned right and if it was negative the robot turned left.

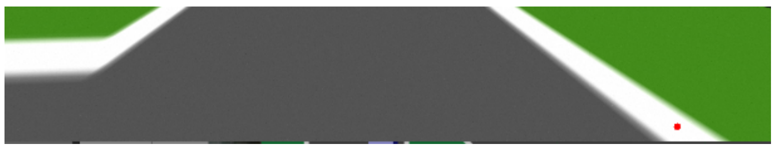


Figure 3.1: Centroid of the outside white road line that was used to stay on the road.

3.1.2 Drive Controller

Our driving method worked perfectly, until we reached a crosswalk. Because a crosswalk is comprised of multiple white "road lines," our robot would get confused as to which line it was supposed to follow. More specifically, it would get confused as to which contour centroid to reference in order to determine if it needed to turn or if it was properly aligned enough to drive forward. We implemented a state machine so that we could shut off the `drive()` function when we were going through a crosswalk. We broke our drive state machine down into four sections:

1. **Regular Drive** - `ON_ROAD`:
   - Objective: Uses our `drive()` function to navigate along the outer edge of the course. Uses `pid()` to adjust its alignment.
   - Exit Condition: Once it sees the red stop line for the crosswalk.
2. **Stop at Crosswalk** - `AT_CW`:
   - Objective: Stops at the crosswalk and waits for the pedestrian to cross.
   - Exit Condition: Once our camera has witnessed the pedestrian fully cross the crosswalk once.
3. **Go through Crosswalk** - `IN_CW`:
   - Objective: Overrides our `drive()` function (assumes that robot is fully aligned with the road) and drives through the crosswalk.
   - Exit Condition: Once it sees the red stop line for the end of the crosswalk.
4. **Exit Crosswalk** - `END_CW`:
   - Objective: Overrides our `drive()` function (assumes that robot is fully aligned with the road) and drives until the robot cannot see the crosswalk anymore.
   - Exit Condition: Once it no longer sees the red stop line for the end of the crosswalk. Returns to the `ON_ROAD` state and resumes the `drive()` function.

3.1.3 Crosswalks and Pedestrian Avoidance

We wanted to be confident that we would never hit a pedestrian, as hitting one would result in a deduction of 10 points to our score. Originally, we stopped at the crosswalk and would wait until there was no detected movement in the frame. However, very rarely we would arrive at the crosswalk at the split second before the pedestrian begins to cross. We would begin to drive and the pedestrian would run into us. Although this would not technically count as getting hit by a pedestrian, the simulation physics were set up in such a way that gave the pedestrian excess momentum, and thus if they ran into the robot it would be knocked off of the course. We needed to ensure that every time the robot arrived at the crosswalk there would be no chance of a pedestrian hitting us. This meant that the robot needed to wait until it witnessed the pedestrian cross the road until it began to drive again.

Initially, we thought of using contours to detect something such as the colour of the pedestrian's pants and determine if the centroid of that area is moving. However, we felt this may not be always reliable and the pedestrian's clothing spawn could also be random so this would not work. We then used a method of frame comparison to detect if the pedestrian was moving or not. Every five frames the robot was at the crosswalk for, we saved that image so that we could compare it to the previous image. We didn't want to compare every frame or every other frame because we wanted to maximize the difference in frames, while also preserving the intent of having our robot being able to adapt quickly to the pedestrian's movements. We used the `compare_ssim()` method included in the library `scikit-image`. Once this was implemented, our robot never hit any pedestrians nor was pushed off track by a pedestrian.

After implementing our drive controller and our pedestrian checking methods, we were confident in our robots ability to not have any points deducted in the competition related to driving. We could then move on to license plate detection.

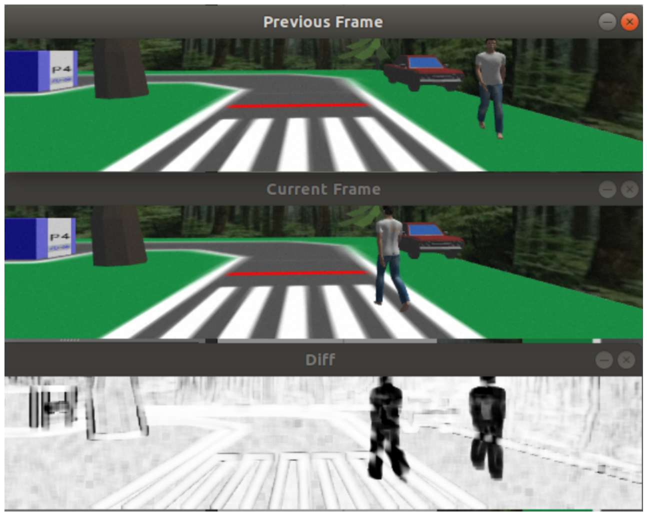


Figure 3.2: Pedestrian detection method. Shows the two frames we compare and the difference between them to determine if the pedestrian is moving.

3.2 License Plate Detection - licensePlates.py

Although license plate detection relied on our driving to be consistent, it was a separate process from the driving and only relied on the camera feed from the robot to be able to complete its task.

3.2.1 Car Detection

The first thing we needed to be able to do was to detect when a car was in frame. We realized, however, that detecting when a car is in frame is not enough to be able to score points confidently. There was a specific frame that we needed to capture in order for our following functions to be able to do complete their tasks reliably. From the `rospy.spin()` call, we received the current frame from the camera and cropped the image so that we were looking at the left half of the frame. This cut down the image processing time as the car would only be visible on the left hand side of the frame.

Each of the parked cars were a very distinct blue colour that did not appear elsewhere in the course. The first function that was called after the subscriber node received an image from the camera publisher was the `getCar()` function. `getCar()` reused our mask function and detected when the robot saw this blue colour. In addition, the area of the blue mask needed to exceed a certain threshold value. This condition successfully prohibited the robot from sending the very first frame (the frame where the edge of the car was just detected) to functions when there wouldn't be any part of a licence plate in the frame. Once this blue threshold was met, we then checked a grey threshold to determine if a license plate was in the frame. This very particular grey-beige colour appeared on the front of the car. These two masked colour threshold checks allowed us to be sure of two things: if the blue threshold condition is met then there is a car, and if the grey threshold condition is met we have detected a licnese plate on the front of the car. If a frame satisfied both of these conditions, it would be passed to the `detectPlate()` function.

To ensure that two frames passing these masking conditions for one car were not both passed, a flag system was used that immediately set a flag when the two masking conditions were met. Once flagged, it prevented the `getCar()` function from passing another frame that met the masking conditions (the very next frame for example) to the `detectPlate()`. This was important, since if we correctly predicted the licence plate characters in the first snapshot but did not in the second snapshot (that may not have contained the full licence plate but still met the masking conditions), our correct guess would be void as the incorrect guess would overwrite the correct one in the score tracker. The flag would be reset once the blue mask area in the frame was 0 (there was no more car visible in frame).

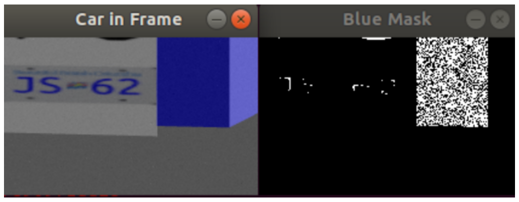


Figure 3.3: Cropped frame used to determine whether or not there is a car in the frame by masking the blue and determining the total blue area.

3.2.2 License Plate Cropping

Our `detectPlate()` function takes in the left half snapshot of the frame containing a clear view of the license plate from our `getCar()` function. Our strategy was to find the connected components of the image. We immediately converted this image into greyscale and then into binary using adaptive Gaussian thresholding. The binary allowed us to have the background of the plate to be all black and the letters themselves would appear in a very sharp and distinct white. We then used a connected component method, which returned the vertices of the box surrounding the connected component (a letter or number in our case). These vertices were used to crop the characters and save them in an array. To ensure that only characters were saved and not other connected components, the cropped images were passed through a filter that checked the horizontal and vertical pixel lengths and compared them to expected values. The next requirement was that four characters were present. If fewer than four were found, the process was aborted as an incomplete license plate reading had no value. If four characters were successfully found, then the letters were sent to the convolution neural network (CNN) to be determined.

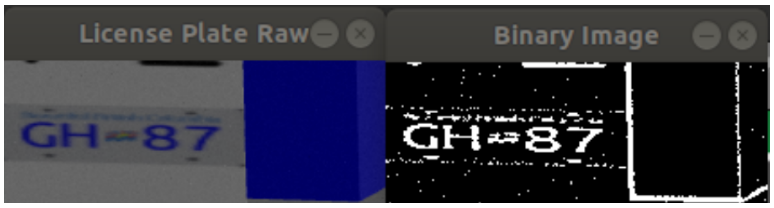


Figure 3.4: Binary license plate used for connected component analysis.

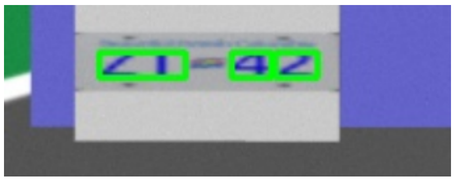


Figure 3.5: Connected component analysis output.

3.2.3 Convolution Neural Network (CNN) Implementation

While we were working on and debugging our `detectPlate()` function, we made sure that we saved the cropped plate letters to a local folder. By the time that we had completed this function, we already had a large amounts of each letter and number stored in a folder to use for training our neural network. We found that when training one model with both the number and letter data sets, it would often get mixed up between certain numbers and letters (most commonly was 2s and Zs, 0s and Os, and 4s depending on the crop). To make scoring points more reliable and to save time training our neural networks, we used two separate models. Because the first two characters on the license plates were always letters, and the last two characters were always numbers, it was a strategic move to have two models to increase reliability. We used the same code that we originally used to train our big model and trained both our character neural network and our number network separately. We had excellent accuracy with both of our models as shown in figure 3.7. We then moved on to integrate our neural network in our code to determine if it maintained its accuracy in the simulated environment.

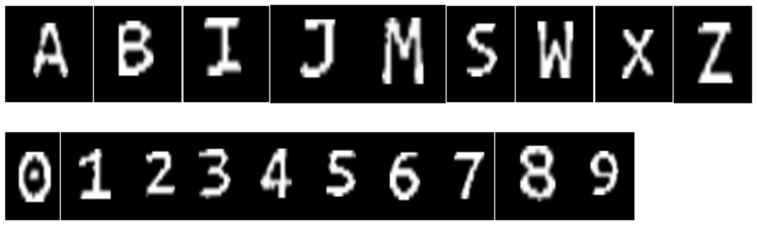


Figure 3.6: Cropped characters sent to the neural network to be analyzed.

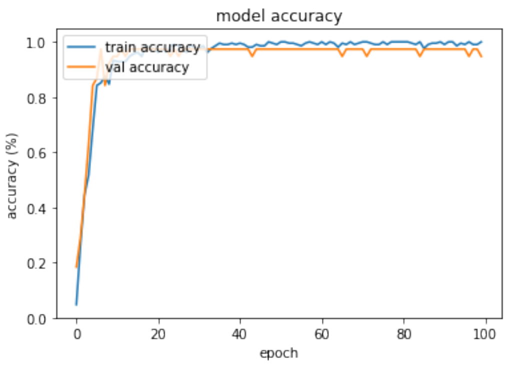


Figure 3.7: Accuracy vs Epochs.

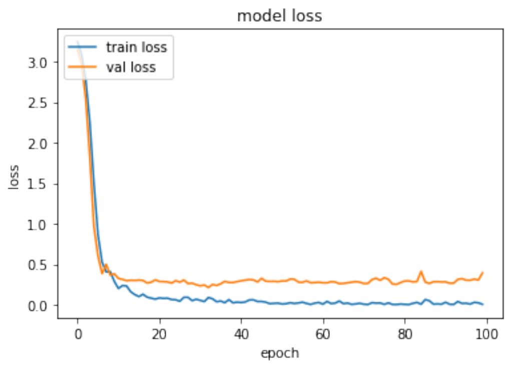


Figure 3.8: Losses vs Epochs.

Parameter	**Metric**
Input Image Size	[22,14,1]
Convolution Layers	2
Max Pooling 2D Layers	2
Flatten Layers	1
Dense Layers	2
Dense Layer Size	10 - Number model 26 - Letter model
Optimizer	"Adam" Optimizer
Loss Option	categorial_crossentropy

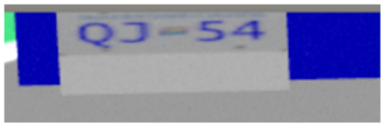
Table 3.1: Convolution Neural Network Parameters.

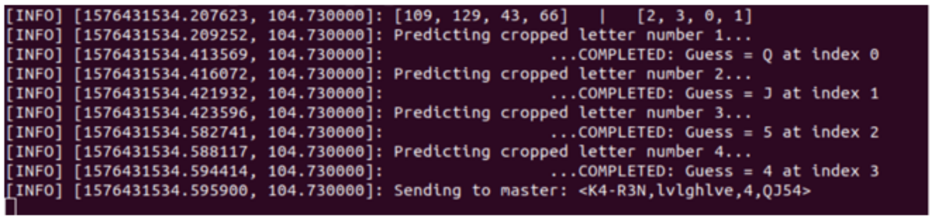
3.2.4 Scoring Points

Lastly, we needed to debug and verify that our model worked with our code as our robot drove the course. In our `detectPlate()` function we sorted all of our cropped letters. Once a detected region passed all of the criteria, we stored its minimum x coordinate (the bottom left hand corner of the crop) and the cropped letter image respectively in two separate arrays (`xindeces` & `croppedLetters`). First, we checked to see that we had actually cropped four characters. If we did not, we would not send the cropped characters to our models. For debuging purposes, we would use `rospy.loginfo()` to note that there was an incomplete licence plate detected, then increment our parking space number, and continue driving.

If there were four characters in the `croppedLetters` array, they then needed to be sorted. Using two `argsort()`

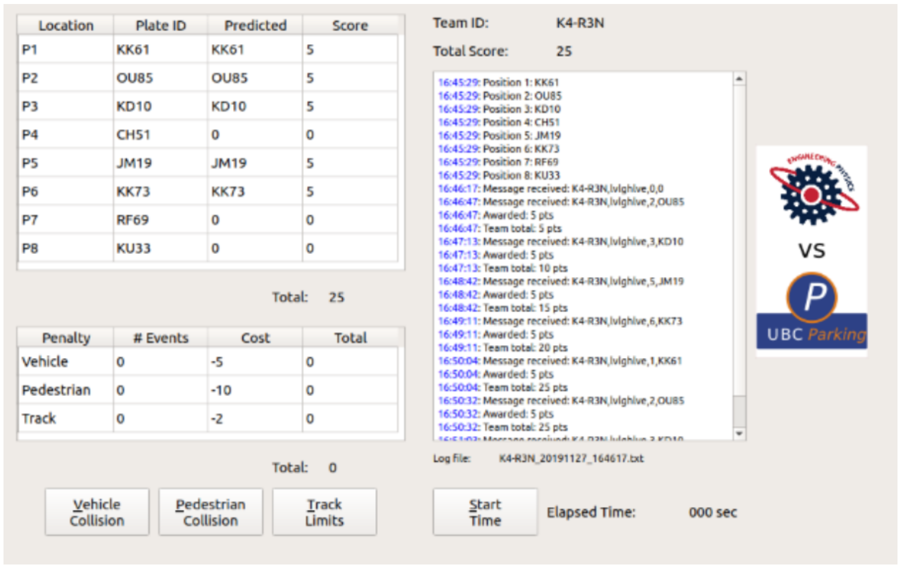
methods, we were able to sort the `xindeces` array such that the order of the original x coordinates was maintained, however the values of the x coordinates were changed to how they ranked in size. For example: using `np.argsort(np.argsort(xindeces)).toList()` on an `xindeces` array that looked like `[200,400,300,100]` would return an array that looked like `[1,3,2,0]`. We then initialized a loop from `n=0` to find the index in the `xindeces` array to determine which index in the `croppedLetters` array corresponded to that chronological character. If the arbitrary 'n' value was either a 0 or a 1, we could determine that the character corresponding to this index was a letter. We could therefore take the *n*-th index from the `croppedLetters` and send it to the letter neural network. If 'n' was either 2 or 3, we would send the *n*-th index from the `croppedLetters` and send it to the number neural network. Each time the model was called to predict from the cropped image, its guess would be appended to the `platePrediction` string. We would then take this `platePrediction` string along with the parking spot of the car and send it to the score node to earn points.





```
[INFO] [1576431534.207623, 104.730000]: [109, 129, 43, 66]   |   [2, 3, 0, 1]
[INFO] [1576431534.209252, 104.730000]: Predicting cropped letter number 1...
[INFO] [1576431534.413569, 104.730000]:                     ...COMPLETED: Guess = Q at index 0
[INFO] [1576431534.416072, 104.730000]: Predicting cropped letter number 2...
[INFO] [1576431534.421932, 104.730000]:                     ...COMPLETED: Guess = J at index 1
[INFO] [1576431534.423596, 104.730000]: Predicting cropped letter number 3...
[INFO] [1576431534.582741, 104.730000]:                     ...COMPLETED: Guess = 5 at index 2
[INFO] [1576431534.588117, 104.730000]: Predicting cropped letter number 4...
[INFO] [1576431534.594414, 104.730000]:                     ...COMPLETED: Guess = 4 at index 3
[INFO] [1576431534.595900, 104.730000]: Sending to master: <K4-R3N,lvlghlve,4,QJ54>
```

Figure 3.9: A correct license plate prediction.



Location	Plate ID	Predicted	Score
P1	KK61	KK61	5
P2	OU85	OU85	5
P3	KD10	KD10	5
P4	CH51	0	0
P5	JM19	JM19	5
P6	KK73	KK73	5
P7	RF69	0	0
P8	KU33	0	0

Total: 25

Penalty	# Events	Cost	Total
Vehicle	0	-5	0
Pedestrian	0	-10	0
Track	0	-2	0

Total: 0

Team ID: K4-R3N

Total Score: 25

```
16:45:29: Position 1: KK61
16:45:29: Position 2: OU85
16:45:29: Position 3: KD10
16:45:29: Position 4: CH51
16:45:29: Position 5: JM19
16:45:29: Position 6: KK73
16:45:29: Position 7: RF69
16:45:29: Position 8: KU33
16:46:17: Message received: K4-R3N,lvlghlve,0,0
16:46:47: Message received: K4-R3N,lvlghlve,2,OU85
16:46:47: Awarded: 5 pts
16:46:47: Team total: 5 pts
16:47:13: Message received: K4-R3N,lvlghlve,3,KD10
16:47:13: Awarded: 5 pts
16:47:13: Team total: 10 pts
16:48:42: Message received: K4-R3N,lvlghlve,5,JM19
16:48:42: Awarded: 5 pts
16:48:42: Team total: 15 pts
16:49:11: Message received: K4-R3N,lvlghlve,6,KK73
16:49:11: Awarded: 5 pts
16:49:11: Team total: 20 pts
16:50:04: Message received: K4-R3N,lvlghlve,1,KK61
16:50:04: Awarded: 5 pts
16:50:04: Team total: 25 pts
16:50:32: Message received: K4-R3N,lvlghlve,2,OU85
16:50:32: Awarded: 5 pts
16:50:32: Team total: 25 pts
```

Log file: K4-R3N_20191127_164617.txt

Elapsed Time: 000 sec

Figure 3.10: Competition results.

4. Challenges

4.1 Addressing Drive Limitations

Due to competition constraints, our robot was limited to performing one movement at a time; our robot could be either turn or drive straight but not both at the same time. This was a result of how commands were allow to be published to the velocity controller. Another limitation that we encountered was that we had no control over the speed at which the robot was able to perform a movement, as both the angular-z velocity input (responsible for turning) and the linear-x velocity input (responsible for driving) had digital-like states. It was either driving at a specified speed, turning at a specified speed, or stationary. This made it frustrating when we needed to make small adjustments to our track alignment, as our robot would often overshoot when trying to readjust. We reused our `halt()` in tandem with our turning. Instead of our robot realizing it needs to turn, turning, and then overshooting the turn while it reenters the drive controller loop, it would turn a small amount, stop, and then reenter the drive controller loop and make necessary adjustments from there. This meant that our robot spent less time realigning itself, was less likely to overshoot, and was more likely to be aligned properly with the cars and hence be able to take clearer images of the licence plates.

4.2 License Plate Detection Challenges

When using connected components to detect license plate characters, there would be more connected components than just characters. In order to combat this, we needed to make a filter for every connected component cropped image to go through before we were sure that it was a letter (must meet height and width upper and lower restrictions). Additionally, there were problems using connected components because of complications with changing the license plate photos to binary. Initially, we used a method that would use a global threshold value, but this led to some photos being too dark to read. A fix to this was adaptive Gaussian thresholds, which determine a threshold value based on every photo separately and then turns the image to binary based on its own threshold value [2].

Bibliography

[1] Rosebrock, A. (2018, December 5). Image Difference with OpenCV and Python. Retrieved from https://www.pyimagesearch.com/2017/06/19/image-difference-with-opencv-and-python/.

[2] Image Thresholding. (n.d.). Retrieved from https://docs.opencv.org/trunk/d7/d4d/tutorial_py_thresholding.html.