ENPH 353 Final Report

Yousif El-Wishahy
Carson Sidhu

Table of Contents:

Background/Goals

The primary objective of the ENPH 353 Competition was to develop an autonomously driving robot, which navigates a simulated world, obeying traffic laws, and returns the license plates and associated parking IDs of various cars it sees along the way. In teams of 2, students were tasked with developing an agent in a Linux environment which could best complete such a task. Each team was granted 4 minutes of simulation time to navigate the simulated track, with points being awarded for correct license plates being read, and points deducted for any traffic law violations. For robot control, teams are provided with a live image feed, captured from the robot in simulation, which can be used both for aiding in navigation, as well as capturing images of license plates. From there, teams must also design, train, and make use of a neural network to read these plates, and publish to a ROS topic the appropriate license plate, and its corresponding parking ID.

Software Architecture

Our general software architecture closely mirrored that which Miti had laid out for us in the competition notes (see Figure 01 below). The primary differences being our naming conventions; ENPH_353_competition acted as a clone of ros_ws, where all of our developed code lies within the src directory in a package named test_controller (GitHub link). The competition repo is also a package in the src directory; notably, it is a submodule of the ENPH_353_competition repository. The key scripts within the test_controller directory were used to drive our robot and detect license plates (see robot_driver.py and license_detector.py). We opted for PID control due to us wanting to avoid over-complication in our steering; we had felt more comfortable and experienced using PID for control, and therefore wanted to stick with what we knew. In terms of neural network training, we opted to do this within Google Collab and import the model into our 'test_controller/models' directory.

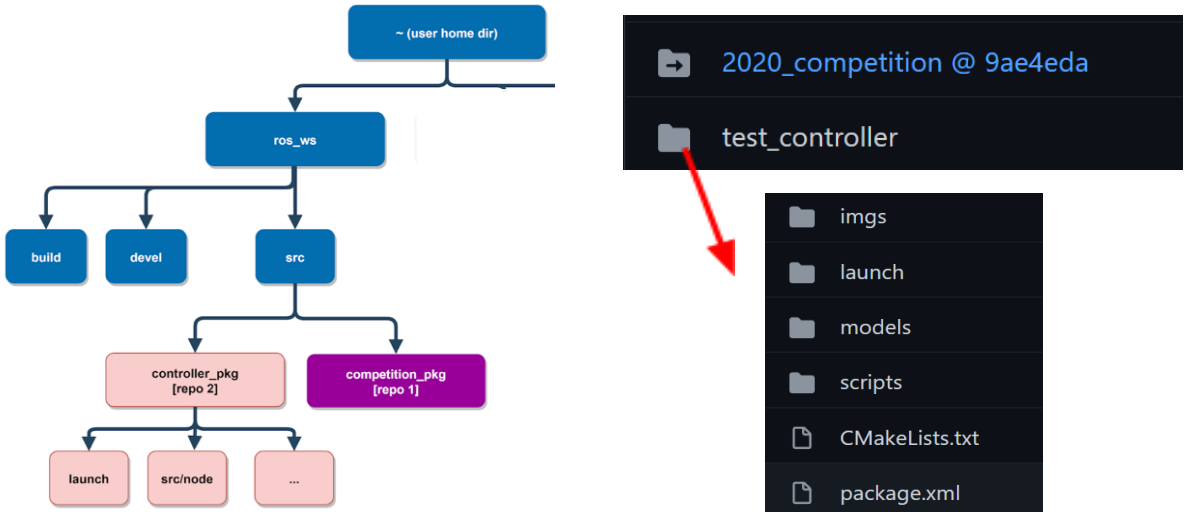


Figure 01: Software Architecture outline provided by Miti in competition notes (left); packages inside the "ENPH_353_competition/src" directory (top right) ; structure inside controller package (bottom right)

Driving

Robot driving ran in a node, named robot_driver.py, that subscribed to the live camera feed and published to the velocity command topic. To traverse the simulated track environment, the robot moved at constant linear speed and had an angular velocity proportional to the PID control's output value. The controller utilized key aspects from Lab 3 (lab 3 code); a calculated PID value was formed using pixel colour detection, where the error is the robot centroid's X displacement from the road's centroid. A similar technique was used for pinpointing red centroids to detect the stop points before crosswalks. These methods formed the basis of our driving control (see "control_loop", "calculate_pid", and "detect_stop" in robot_driver.py). Tuning the PID values was a matter of setting I and D to zero and increasing P until the error started to oscillate. The tuned controller did not use I values as they did not improve the result due to larger turn radii.

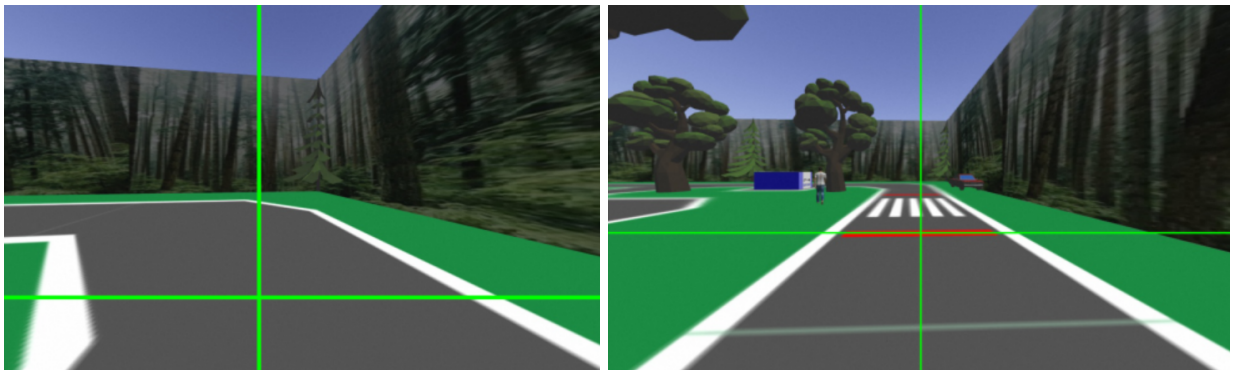


Figure 02: Images of controller's interpretation of road and crosswalk locations, the green lines are debug tools that mark the detection points

Alternative methods for following the road included white contour detection with OpenCV's contour method. The locations of the left and right road borders would indicate the location of the road centroid. However, this technique proved to be more unstable than pure pixel colour detection as the camera feed would occasionally lose sight of the contours when turning or near intersections. Edge cases where the white contour was orthogonal to the camera's vector also resulted in PID instabilities. As a result, we opted to use the pixel colour detection (see Figure 02) for the competition.

License Plate Detection

License plate detection ran in a separate node with a subscriber (see license_detector.py) and consisted of several image processing stages applied to every new frame received from the camera feed. The first step was to detect and crop the parking bin plate, containing the parking ID and license plate. An inverting mask was applied to the image to highlight the specified gray/white colour of the bin plate. This inverted image is then denoised with erosion and dilation, converted to grayscale, and passed to OpenCV's contour function which returns several potential contours. The ideal contour is selected based on area and shape by using OpenCV's area comparator and polygon approximation methods. Finally, the source image is cropped to the dimensions of the largest rectangular contour found.

The next stage after cropping the bin is to crop out the parking ID and license plate separately. To achieve this, a blackhat morphological operation is applied to the image to distinguish the darker text from lighter background (adapted from Adrian Rosebrock's methods) . This returns potential license and ID crops that are then distinguished from each other with HSV thresholding in the blue colour range.

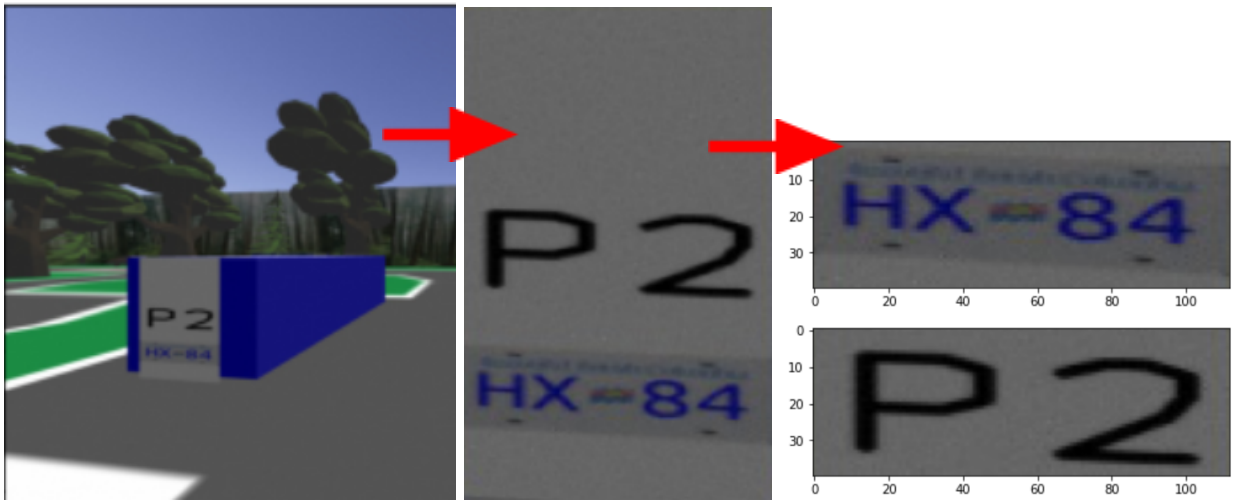


Figure 03: Source image from live feed (left); resulting crop from white/gray contouring (middle); resulting crops from blackhat morphology (right)

After cropping out the license plate and ID, the individual characters need to be cropped as well. For the ID, we found that simply dividing the image in half on the X axis is sufficient. To crop the license plate characters, a sharpening kernel is applied to the image using OpenCV's filter2D, then the image is eroded such that the letters become blocks suitable for contouring. The contours are used as cropping bounds and the resulting four crops would then be passed onto character recognition.

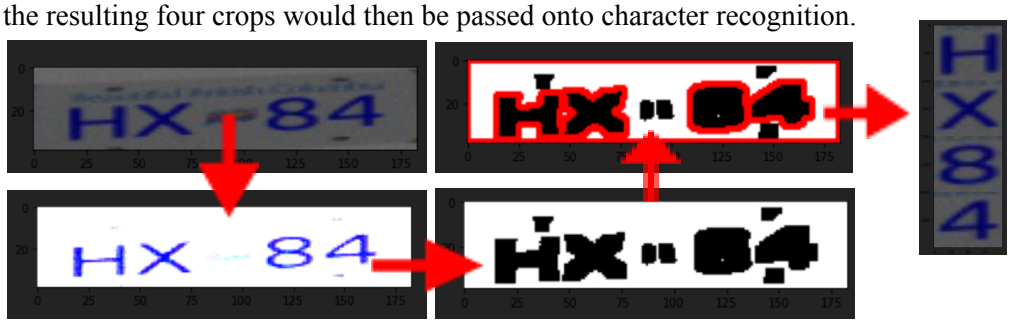


Figure 04: Letter cropping process starting from image of license plate (top left) and resulting in four letter crops (right)

Convolutional Neural Network

For character recognition, our team opted to use a convolutional neural network trained with the TensorFlow Keras API. Initially, we struggled to understand how to go about integrating a neural network with the license_detector.py ros node. After speaking with Miti, this process was clarified. The process was as follows: train a neural network in Google Collab using images augmented to resemble those captured in simulation, export the trained neural network, and then load it into our code. From there, it may be used to make predictions on the characters we passed through it.

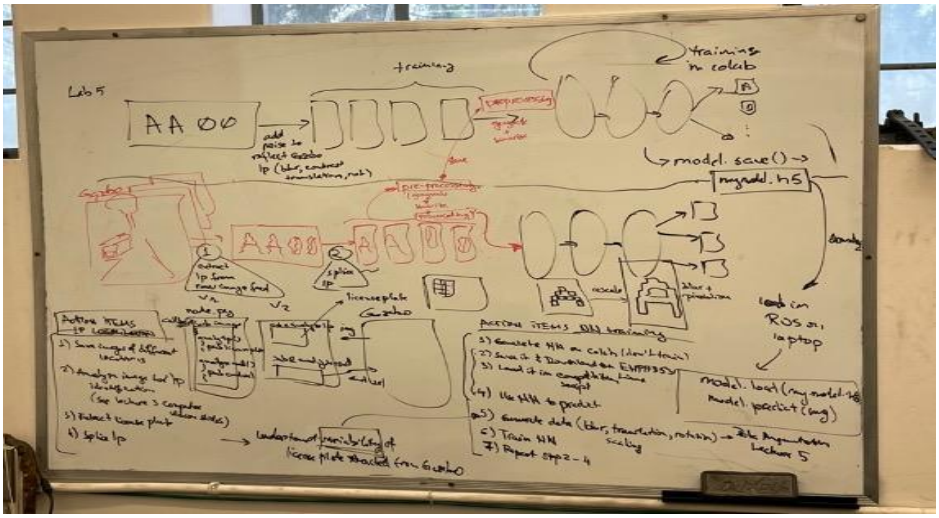


Figure 05: An outline provided by Miti for how to go about training and integrating a neural network with Linux environment

The first step was to train the network in Google Collab, this process closely mirrors the work done in Lab 5. What changed, however, was the images we passed through it and the layers of the CNN. Initially, image augmentation was limited to filtering the images by introducing rescaling artifacts (method adapted from Miti's example). After augmentation, the images were cropped into four equal size segments to isolate the characters. The final step before training and prediction was preprocessing via binary thresholding; this would reduce the effect of lighting conditions on the model predictions. The images were also resized to a standardized image size as the character cropping code could return any size depending on the proximity of the robot to the license plate. Most importantly, this preprocessing stage was applied to both the training data and the ros node data to ensure consistency.

Upon training 10 epochs, with the following architecture shown in Figure 06, we observed the following losses and accuracy, shown in figure 07. See figure 06 (right) for the confusion matrix representing our predictions.

```
[ ]  1 conv_model.summary()

Model: "sequential_7"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_14 (Conv2D)           (None, 296, 148, 32)      896
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 148, 74, 32)       0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 146, 72, 16)       4624
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 73, 36, 16)        0
_________________________________________________________________
flatten_7 (Flatten)          (None, 42048)             0
_________________________________________________________________
dropout_7 (Dropout)          (None, 42048)             0
_________________________________________________________________
dense_14 (Dense)             (None, 128)               5382272
_________________________________________________________________
dense_15 (Dense)             (None, 36)                4644
=================================================================
Total params: 5,392,436
Trainable params: 5,392,436
Non-trainable params: 0
_________________________________________________________________
```

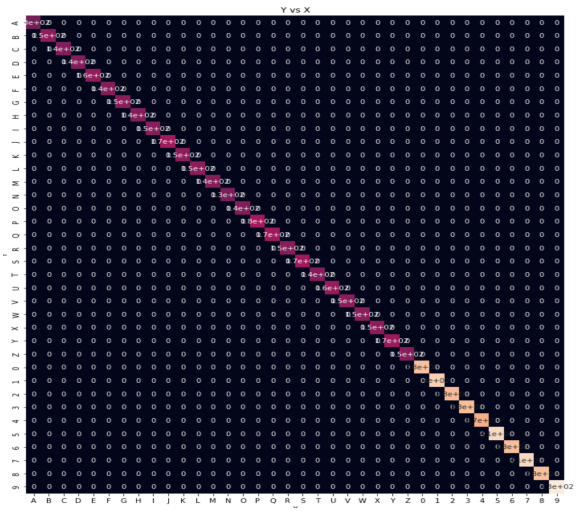


Figure 06: Neural network architecture (left) ; confusion matrix (right)

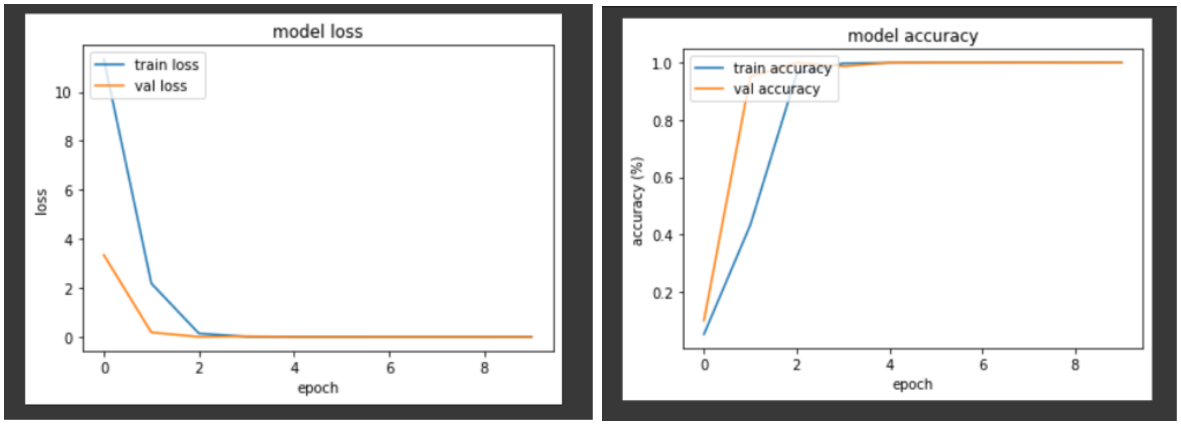


Figure 07: Model loss (left) and model accuracy (right)

After exporting this neural network, and testing it in simulation, we found that it was not able to accurately predict many license plates, often mistaking letters for numbers. After comparing the augmented training data to the simulation data we realized that the training data was sharper and more pixelated than the simulation data. To address these issues, we modified the augmentation stage by first reducing the effects of the rescaling artifacts, then adding gaussian blurring, and finally applying TensorFlow Keras' ImageDataGenerator methods.

Figure 08: Simulation data (left) and augmented training data (right)

In addition, we noticed that the binary thresholding in preprocessing resulted in a noisy image. To address this, preprocessing was modified to use OpenCV's adaptive binary thresholding with a large block size and subtractive constant. This resulted in a less noisy image with more rounded characters. While there were still differences at this stage, we felt that this was inevitable and went ahead with retraining the model. This model utilized the same layer structure as in figure 06 with the changes only being made to the input data augmentation and preprocessing. Due to the variation in data, it required more images (~7000) and 80 epochs to achieve a greater than 90% accuracy. Ultimately, we named this CNN "model_18" and would go on to use it in the competition.

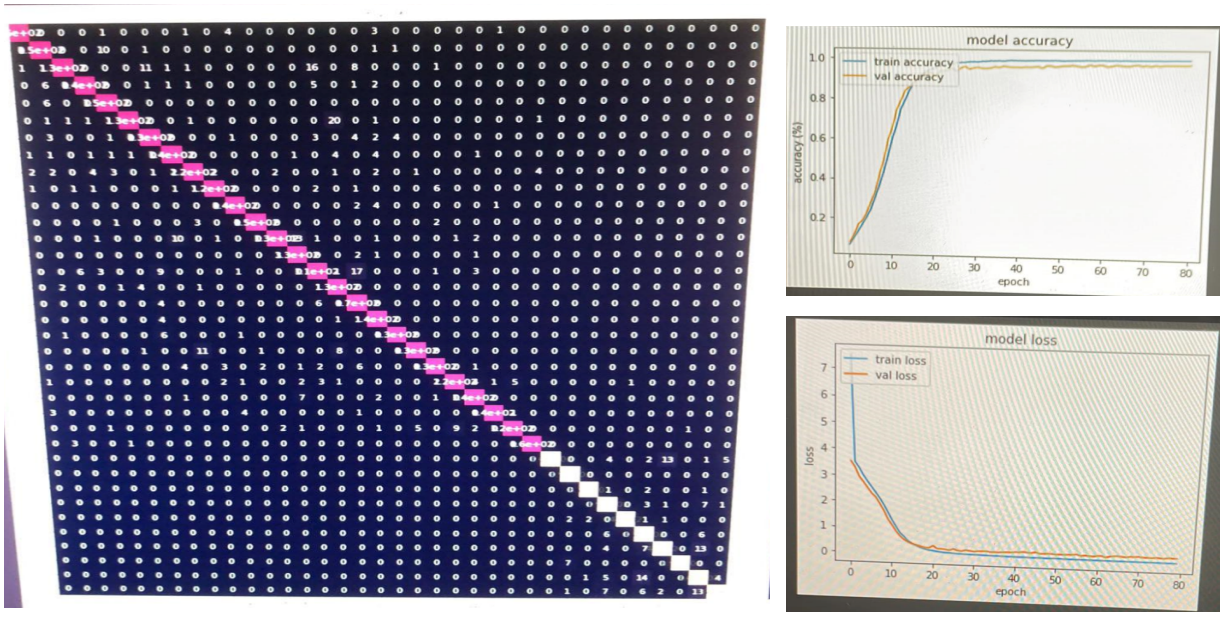


Figure 09: Model 18 confusion matrix (left), loss (bottom right), and accuracy (top right)

Model_18 was still not good enough for plate prediction in the simulation and we thought that this was due to inconsistent cropping. After several more attempts at optimization, it was not until 1 hour prior to competition, where we learned from Miti that we had not taken into account the fact that the license plates fed to the neural network to train, were a different font than those seen in simulation. After scrambling to try and account for this in our neural network, we simply ran out of time to resolve all the other issues. These issues included font size and alignment for plate generation and cropping which arose when we altered the font of our license plate used to train the network. After training the data with 80 epochs, the model's accuracy plateaued at around 0.57. This was mainly due to improper font alignment and cropping which resulted in incomplete or empty letters being passed into training. We did not have enough time to fix these issues and so we opted to use model_18.

Robot Control Method

Unfortunately, we did not have enough time remaining before competition to implement pedestrian, nor vehicle detection. Our temporary fix was to wait a second upon reaching crosswalks, in the hopes that our likelihood of running into a pedestrian was low. We seemed to have underestimated the likelihood of contact being made, as in live runs, we connected with the pedestrian more frequently than we were expecting.

Few edge cases were discovered or accounted for, but the most notable was that upon spawning the vehicle and starting the controller. The lack of contour between the colours of the road and the wall during the initial movement of the robot, could cause the PID to struggle determining which direction to move, and resulted in some unpredictable behaviour. To fix this, we implemented a left turn immediately upon starting the controller, which only lasted for a brief window of cycles through the control loop. Additionally, the robot would turn into intersections briefly before turning out due to the colour detection method encountering more pixels at the intersection turns.

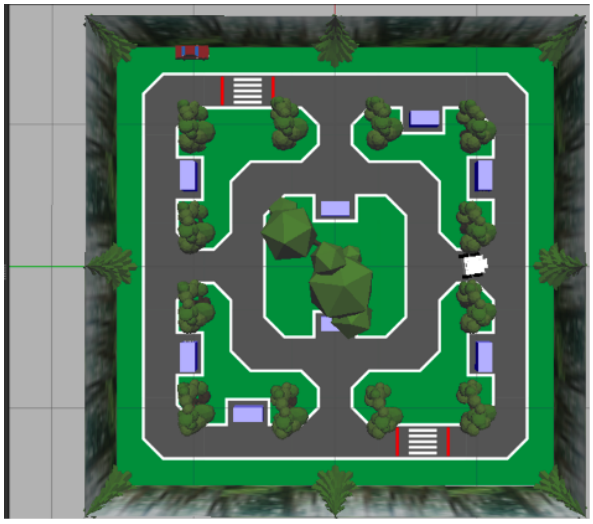


Figure 10: Screenshot of initial movement of robot, slightly angled to the left

Improvements After Competition

After the competition, we felt that we were very close to having a working controller because we knew what we needed to improve. This is especially true after learning about the various implementations of our classmates. Improvements included a more accurate CNN, easier debug tools, increased driving stability, pedestrian detection, and improved license plate detection. All improvements were made in a fork of the competition repo (forked repo link).

Debugging

Previous debug tools included a separate node that received images from the robot_driver.py node and printed them with OpenCV. We realized that this was an unnecessary step. The improved debug tool included the creation of 'debug_item' classes for the pid controller and license detector; these classes encapsulated any images, strings, and other values that indicate the result of the algorithms, their success flags, and the robot's view of the arena. These debug items are constantly published to debug topics after each control loop iteration in license_detector and robot_driver. The end result is that all the debug information can be accessed in 'rqt_image_view' without significantly delaying the process loop. This proved to be very important for improving the controller and license detector as it gave us a clear view of where things were going wrong.

PID Driving Stability

Driving stability was improved by revisiting the white contour methods mentioned in the driving section. While relying on detecting both lines proved to be unreliable, the outer line is present at all times while traversing the outer loop. Therefore, the improved PID controller would measure the error as a displacement from a fixed offset off the white border. This proved to be much more reliable than pixel colour detection or double border detection as both those methods had edge cases whereas the single white border was more consistent.

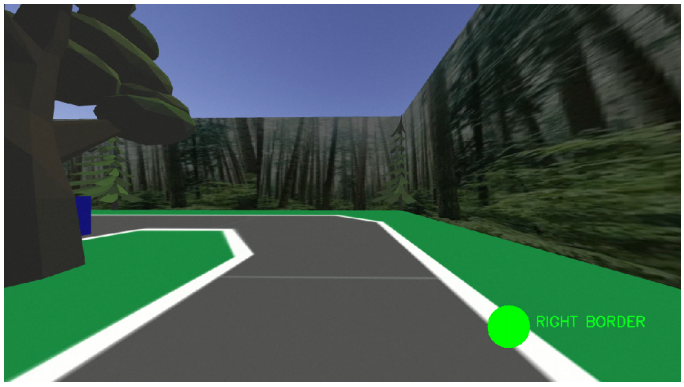


Figure 11: Right border detection debug view for road following, the robot has a hard coded offset of the right border that it follows

Pedestrian Detection

The previous pedestrian detection was nonexistent, the robot would detect the red line before crosswalk and stop moving for a period of one second but continue moving again without validating that the pedestrian crossed. The red detection was not always accurate either because it searched for one BGR pixel value and did not account for red pixels under different light conditions. To improve upon this, the new method utilizes HSV thresholding in the red color range such that only the crosswalk shades of red are detected - and not the red vehicle near the stopwalk. In addition to HSV thresholding, contouring is also applied to the red HSV mask to outline the red crosswalk contours. After the robot stops at the crosswalk, variations in these red contours indicate that an entity, in this case a pedestrian, is moving across the crosswalk. Right after these variations occur, the robot knows that the pedestrian has safely crossed and can proceed to drive forward. This method worked well and resulted in no pedestrian collisions (pedestrian demo).

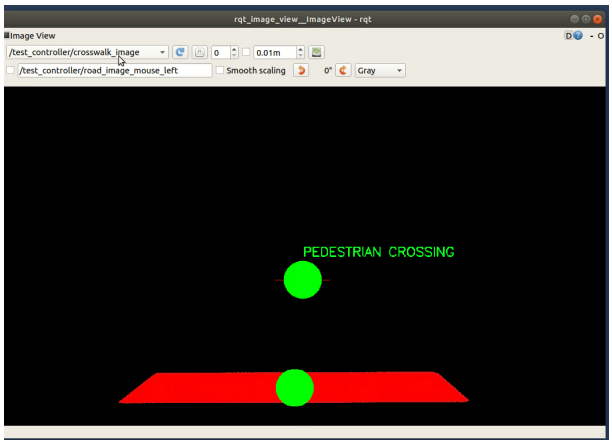


Figure 12: Crosswalk and pedestrian detection debug view

CNN and License Plate Detection

The main improvement for the CNN was utilizing a different font and formatting it correctly to match the competition environment; in other words, training the CNN with data that matches the simulation. All other parameters for the model and data preprocessing remained the same as mentioned above. After this was achieved, the license plate detection code was able to accurately detect the IDs (ID detection demo).

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 98, 98, 32)        320
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 49, 49, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 47, 47, 16)        4624
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 23, 23, 16)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 8464)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 8464)              0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               1083520
_________________________________________________________________
dense_3 (Dense)              (None, 36)                4644
=================================================================
Total params: 1,093,108
Trainable params: 1,093,108
Non-trainable params: 0
_________________________________________________________________
```

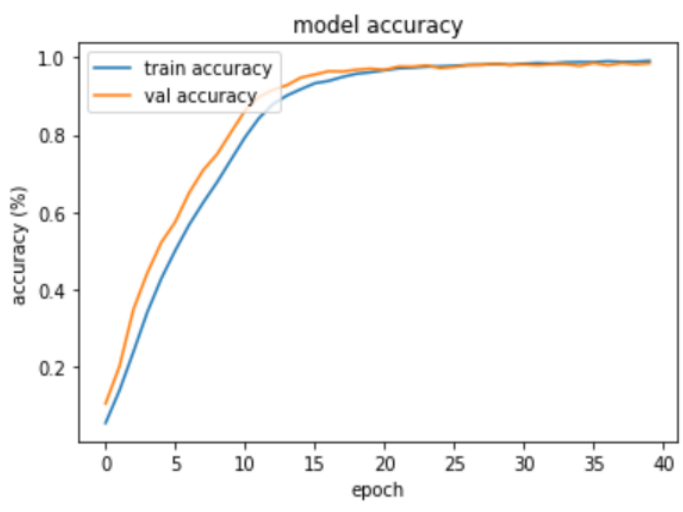
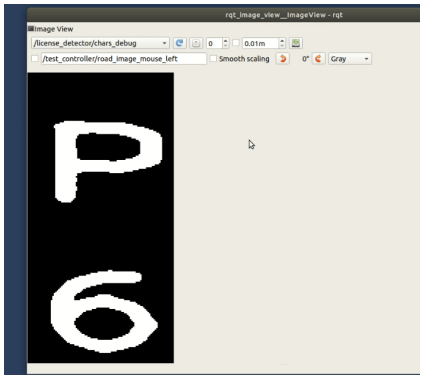


*Figure 13: CNN structure for improved model (left) and model accuracy (right) when trained with ~6000*4 images for 40 epochs*




```
("('R', 0.88980633)('R', 0.9774006)('8', 0.771797
("('P', 0.9729011)('6', 0.47693965)", True)
("('P', 0.99983644)('6', 1.0)", True)
("('P', 0.99503833)('6', 0.99999046)", True)
```

Figure 15: ID detection results with confidence values (right) ; simulation debug view of cropped characters (left)

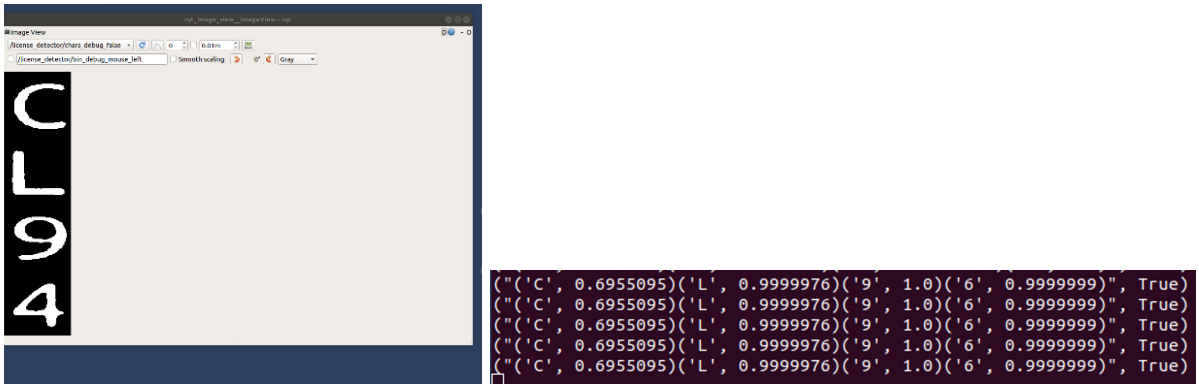


Figure 16: Plate detection results with confidence values (right) ; simulation debug view of cropped characters (left)

As seen from figure 16, the CNN is fairly accurate but still requires improvements on the augmented data fed to it to achieve better results. We suspect that the training data either has too much noise as compared to the simulation data due to differing cropping techniques or over augmentation.

Conclusion

As we came to expect after learning of our last-minute error in training the neural network, we performed very poorly in competition. As was also mentioned earlier, we did not implement pedestrian detection, which led to 3 collisions throughout the course. This was probably a worst-case scenario in terms of how we could have performed, as there were other instances in which we tested where pedestrian detections were uncommon. Furthermore, the first collision spun our robot around and set it on a path to navigate the track on a clockwise path, resulting in zero license plates being seen, so we had no hopes of scoring points apart from +5 for completing a lap.

With more time prior to competition, we felt as though we could have been in a much better place and would have hopefully had time to sort out all the implications which came with a new font being accounted for in our neural network. If we were to have another chance at the competition, we would give more attention to ensuring our driver was in compliance with all traffic rules, and not at risk of losing us as many points as we did. Reflecting on this result, we decided to continue working on the controller after the competition. This included improvements to the PID controller, license detector, and CNN as we had a clear vision of what could be modified to increase performance and accuracy. This resulted in more stabilized road following, pedestrian detection, and an improved character recognition CNN.

To summarize, the final iteration of our robot was able to navigate the simulated arena, follow road rules, and detect pedestrians, license plates and IDs. Vehicle detection was not implemented. Character recognition with the CNN improved drastically from initial results, however, it would still occasionally fail as seen from figure 16.