ENPH 353 FINAL REPORT

Maxwell Yang, 26426296
Richard Echegaray, 73815326

INTRODUCTION

The goal of ENPH 353 was to become knowledgeable in an industry-relevant topic, in this case Machine Learning, and apply this knowledge in the term-long project designed to assess our abilities in this particular field. The course was designed as a competition, and our team's goal from the start was to be competitive - and place in the top 3 teams.

SOFTWARE ARCHITECTURE

Our code runs by calling drive.py - this python script navigates the robot throughout the course and publishes velocities to the ROS node in order to control the car. We use computer vision to navigate and make decisions regarding how to drive, and when we spot a license plate, drive.py sends this frame to a script called license_plate_processor.py, which processes the image into the relevant characters and uses a CNN to predict what the license plate reads, and then we publish that data to another ROS node to be evaluated on accuracy.

DATA GENERATION

For data generation, we based our training set off the two iterations of the plate generator script provided by Miti Isbasescu, one being from Lab 5 and the other being from the skeleton code provided for the competition. The former was used to generate license plates and the latter was used to generate parking spot numbers. We altered the plate generator scripts to closely mimic the resulting characters that would be cropped after image processing. For each iteration of the training set, we generated approximately 2000 plates, resulting in approximately 2000 (for parking spot) or 8000 (for license plates) characters each time. The following operations were performed to alter the original plate generator script using the OpenCV library:

- Gaussian Blur: (cv2.GaussianBlur)
  - Operation blurred image to match quality of cropped image after image processing
  - Used with varying sizes of kernels depending on magnitude of blur desired
- Rotation: (cv2.getRotationMatrix, cv2.warpAffine)
  - Operation rotated image to augment image for CNN training

  - Rotated 10 degrees

- Convert to Grayscale: (cv2.cvtColor(image, cv2.COLOR_BGR2GRAY))
  - Operation converted to grayscale to match colouring of cropped image after image processing
- Resize: (cv2.resize)
  - Operation resized image to standardize size of images inputted into the CNN
  - Resized images to (64, 64)
- Perspective Transform: (cv2.getPerspectiveTransform, cv2.warpPerspective)
  - Operation transformed image to a normal perspective after rotating image, used to augment image for CNN training

Figure 1: Sample Training Parking Spot Label with Slight Blurring

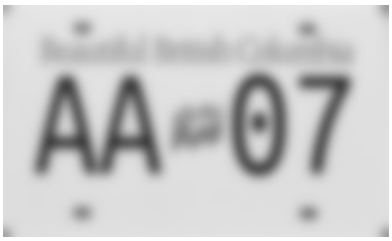

Figure 2: Sample License Plate Training Label with Blurring and No Rotation

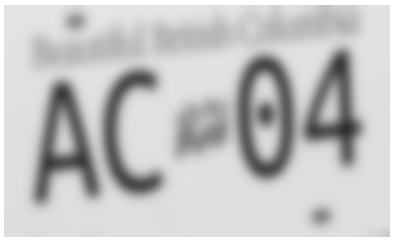

Figure 3: Sample License Plate Training Label with Blurring and Rotation

Each iteration of the training set used different combinations of these operations in varying degrees. The primary training sets, which reaped the best results, involved applying Gaussian Blur with a kernel of (5,5) 0-3 times, rotating by 0 or 10 degrees, converting to grayscale, resizing to (64,64) and had no perspective transform.

After generating the initial training set, we had to split our characters. To do this, we, first, randomized our labels so our training set wouldn't be ordered alphabetically. Then, for each label, we got its corresponding image. For each image, we found contours within a specific range of RGB values, found the bounding box coordinates of each character, and cropped the contours, sorting from left to right. In the case where the aspect ratio of a contour was below a certain threshold, we would split the contour directly in half, which would split the contour into two separate letters. This would occur when two letters' contours merged into one and the method, provided by Anthony Ho, was very accurate in its splitting. Identical to Lab 5, we

would label these characters by taking the corresponding position in the filename string and labelling accordingly.

After splitting the characters, we used Keras' ImageDataGenerator function to augment the images with the following operations included:

- rotation_range: 20, shear_range: 0.15, height_shift_range: 0.15, width_shift_range: 0.15, zoom_range: 0.15, fill_mode: "nearest"

Each of the operations performed were a slight modification in the realm of possibility for the cropped image to result in, therefore this would improve the accuracy of our CNN as it was trained under more flexible conditions. We attempted to alter the operations such that their effects were either increased or decreased, but these parameters worked the best in terms of accuracy of the CNN.

After image augmentation, we took the generated dataset and performed a validation split of 0.2 for our training set and testing set and use these sets to train our CNN.

NEURAL NETWORK ARCHITECTURE

For the convolutional neural network architecture, we decided to have three primary CNNs:

- License Plate Letters CNN: Output Size 26, Used to read letters on license plate
- License Plate Numbers CNN: Output Size 10, Used to read numbers on license plate
- Parking Spot Numbers CNN: Output Size 8, Used to read numbers on parking spots

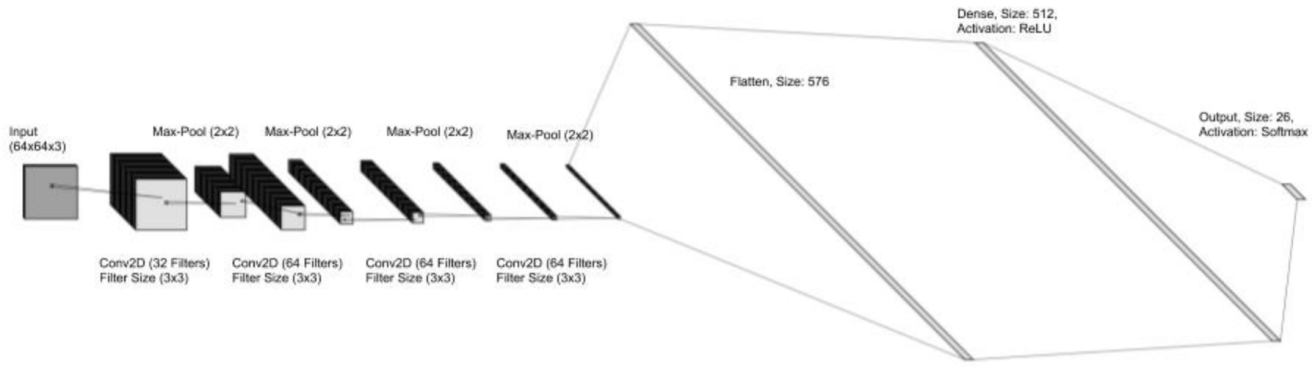

Figure 4: CNN Architecture for License Plate Letters CNN (other architectures are identical except for output size listed above)

We chose to have three separate CNNs as we surmised that this method optimized consistency in each CNN's learning and the outputs of each CNN were more restrictive, therefore increasing the overall accuracy of each of the separate CNNs. Furthermore, when we utilized one general CNN for all characters, we found that similarly shaped letters and numbers would often be confused for each other. By removing any opportunity for this to occur, our CNN became more consistent in its predictions.

In addition, we utilized four backup CNNs, generated using a slight variation of the original training set, to avoid the pitfalls of our primary CNNs. The backup CNNs were designed to be a failsafe for when the primary CNN would predict incorrectly. Though the errors the primary CNN made were necessarily common occurences, the backup CNNs were implemented to ensure consistency of the model. This allowed for more flexibility in the image capturing process, allowing for scenarios with worse angles and poor lighting of the letters. Essentially, if one of the primary CNNs predicted a value of the ones listed below, the backup CNNs would predict their own value and the system would take the backup CNN's value. The four backup CNNs were as follows (with training architecture identical to the corresponding CNNs described above):

- License Plate Numbers Backup CNN 1
  - Usage: Accounted for a predict value of the number 4
  - Purpose: When 8 predicted as 4 in primary CNN
  - Training Set: No rotation and blurring of numbers
- License Plate Numbers Backup CNN 2
  - Usage: Accounted for a predict value of the number 0
  - Purpose: When 8 predicted as 0 in primary CNN
  - Training Set: Rotated 10 degrees and blurring of numbers
- License Plate Letters Backup CNN 1
  - Usage: Accounted for predict values of the letters A, F, V, O
  - Purpose: When B predicts as A, R predicts as F, U predicts as V, D predicts as O in primary CNN
  - Training Set: Biased to B's and R's, Rotated 10 degrees and blurring of letters
- License Plate Letters Backup CNN 2

- Usage: Accounted for predict values of the letters M, G
- Purpose: When W predicts as M, C predicts as G
- Training Set: Rotated 10 degrees and blurring of letters

NAVIGATION

The car always spawned facing the same way, in the same spot, as shown below. Because of this, our approach was quite simple.

Figure 5: Birds-Eye View of Course

The navigation code was written in the form of a state machine, with the following states:

- State 0: Initialization
- State 1: Driving in the inner loop
- State 2: Transitioning to outer loop
- State 3: Driving in the outer loop
- State 4: Watching for pedestrians
- State 5: Cross the crosswalk

Our strategy was to go for the bigger points first - that is, the inner ring. Our initialization method was essentially us turning our car around and beginning to line follow on the inside ring. Since there were only two plates on the inside, we only read two plates, and then transitioned into the outer loop. We went with this approach so that if we missed out on some points early on, we could easily reset without having traversed most of the course. Once we were on the outer ring, our car continued to navigate autonomously, stopping only at crosswalks and watching to see when the coast was clear of pedestrians so that it could continue to drive.

More in-depth description of the states:

- State 0: Initialization:
  In this state, our car rotates counter-clockwise for 2.4 seconds in the ROS environment, which corresponds to 180 degrees. After this, we wait to see the truck pass using computer vision. Once we have seen the truck pass, we give it a 6 second start so that we do not run into it, and begin to line follow in a clockwise direction in the inner loop.

- State 1: Driving in the inner loop:
  In this state, we begin by line-following until we see a license plate. Once we see a license plate (which we know is the bottom inner plate, since we are travelling in a clockwise direction), we snap a picture of the frame and send it over to our CNN codebase. We pause for 5 seconds to allow the truck to once again get a head start, and then continue to drive until we see the second license plate, this time at the top. At this point, we once again snap a picture of the frame, send it off, and then transition into state 2.

- State 2: Transitioning to outer loop:
  Here, we drive forward for a split second, and then rotate our car 120 degrees counter-clockwise, and drive forward, to exit out of inner loop. Because we angled our car at 120 degrees, once we hit the outer edge of the road, we know that we are in the outer loop, facing the counter-clockwise direction, and can simply use line following methods to remain on course for the remainder of the given time, thus transitioning to state 3.

- State 3: Driving in the outer loop:
  Here, we line follow on the outside loop, in a counter-clockwise direction, only stopping when interrupted by identifying a crosswalk, which sends us to state 4.

- State 4: Watching for pedestrians:
  Once we have identified a crosswalk, we wait and watch, to ensure that the pedestrian crossing is safely on the grass, and only when the pedestrian is off the road, we transition to state 5.

- State 5: Cross the crosswalk:
  In this state, we know that the pedestrian is off the crosswalk, so we line-follow for 4 seconds - and afterwards we revert back to state 3, driving in the outer loop.

This sequence of events can continue to run for as long as the user wants, by initially traversing the inner loop once, and then traversing the outer loop an indefinite amount of times, stopping only at crosswalks to ensure we can cross safely.

Frequently used static methods:

- *image_converter(cv_image):*
  This method is run every single iteration, it takes the image that our camera returns, and returns a picture about half of the size. We only work with the returned frame, and it allows code to run a lot faster when outputting what our car sees.

- *get_position(frame, last_pos):*
  This method passes in the frame we are looking at, and returns the position of the centroid of the grey road. We use this to line follow, and we also have a global variable assigned to last_pos, so that we can keep track of how we were previously turning.

- *check_crosswalk(frame):*
  This method is run only when we are in the state corresponding to driving on the outer ring. It serves as an interrupt, that when returns true, forces a change of state. That is, when we notice a red crosswalk at the bottom of the screen, we switch to state 4: watching for pedestrians.

- *check_blue_car(frame, inner_ring):*
  This is the method we use to identify if we have spotted a valid license plate. When on the inner ring, we pass the boolean value True as a parameter for inner_ring, otherwise we pass false. This allows us to either check for license plates on the right if we are in the inner loop, or on the left if we are in the outer loop. If this method returns true, it means that the current frame we are on is a valid frame, and thus we know to send it to our cnn codebase for processing.

Other frequently used methods:

- *watch_people(self, frame, top_crosswalk):* This method is run constantly when we are in state 4: watching for pedestrians. It returns false unless the pedestrian is detected to be off of the road. That is, if we see the green grass pixels change to a white or blue colour (outfit of the pedestrian), we know it is safe to move forward, and we return True, sending us to state 5: cross the crosswalk.
- *stop(self):* This method publishes a velocity of 0, so that the car does not move. It is useful for when we need to let the truck drive by.
- *speed_controller(self, position):* This method takes in the position, and publishes a velocity that will orient us to the middle of the road. That is, if our position is to the right, we will turn left, and vice versa. If we are in the middle, we will continue to drive forward.

PLATE PROCESSING

For plate identification, we would capture an image based off the contour lines and pass it into the license plate processor class. Depending on whether we were in the inner ring or outer ring, which we knew due to our navigation strategy, we would crop the image in half to only show the relevant portion. This would filter out any unnecessary contours which could tamper with the image processing.

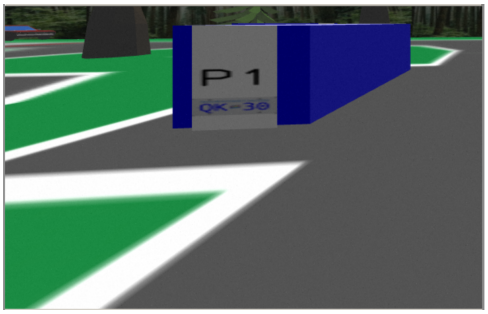

Figure 6: Outer ring frame

To look for ranges of colours, we converted the image into an HSV image and used the cv2.inRange function to identify the areas of the image within the range of colours specified.

To filter out contour sizes, we utilized list comprehensions and cv2.contourArea to filter out contours outside of the range desired.

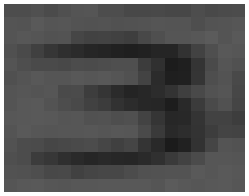

Figure 7: Cropped Number 3 after Image Processing

For this cropped image, we looked for a specific range of white contours of certain sizes as the license plates contained a larger top white contour above the license plate and a smaller bottom white contour below the license plate. After, we took the top and left coordinates of the bottom contour and the top and right coordinates of the top contour

and cropped the image using these coordinates. We allowed for a small leeway in the crop to account for minor errors. Initially, we looked for blue contours instead, but this process did not hold the same consistency as looking for white contours did, as often it would take in neighbouring blue contours, resulting in a bad crop.

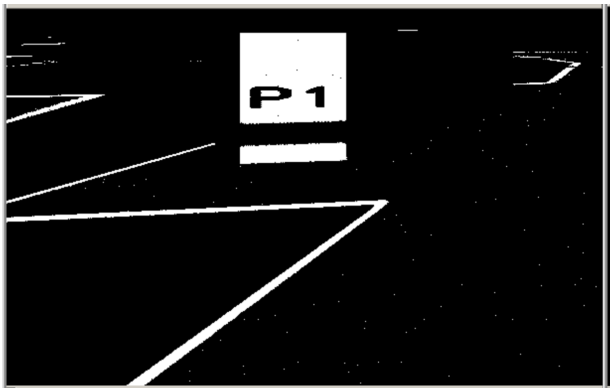

Figure 8: Mask of white contours

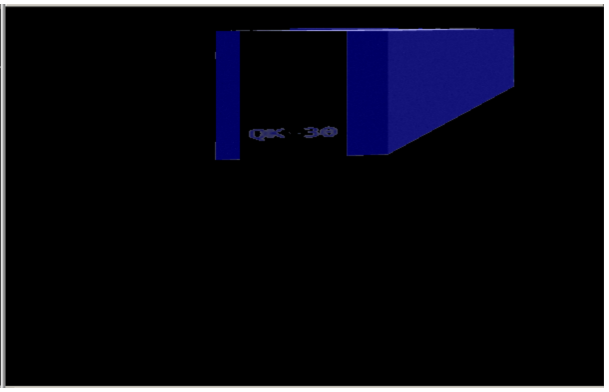

Figure 9: Original method of finding blue masks

For the cropped license plate, we looked for a specific range of black contours of certain sizes to identity the parking spot numbers and a specific range of blue contours of certain sizes to identify the license plate characters. After, we proceeded to sort these contours in terms of their x coordinate from left to right and cropped them based on their bounding box, using cv2.boundingRect. We appended these characters to an array and the CNN would predict its value, after resizing the characters to (64,64), which was the standard we set.

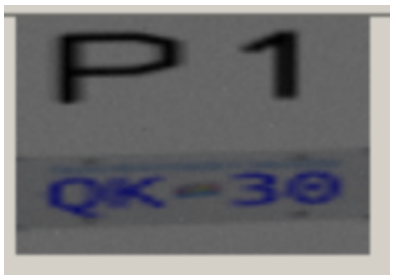

Figure 10: Cropped Plate

Figure 11: Black Mask for the Parking Spot Numbers

Figure 12: Blue Mask for the License Plate Characters

Depending on the current index of the array, the character provided would be predicted in one of the three primary CNNs. These values were appended into a string and published by the ROS node to be evaluated.

CONCLUSION

Our overall result in the competition was 3rd place out 16 teams. We were able to correctly identify every license plate, with an error on one license plate on the initial first loop, and maneuver safely around the course through both the inner and outer ring, without harming a pedestrian or hitting a vehicle. This resulted in us being rewarded full points! The overall takeaways from the project were:

- Familiarity working with the Linux environment
- Basic understanding of ROS and Gazebo
- Knowledge on how to implement an efficient state machine with OpenCV
- Knowledge on how to work with OpenCV
- Knowledge on how to use computer vision to navigate a course
- Knowledge on how to use computer vision for object detection
- Basic understanding of the structure of a CNN and how to train a CNN using Keras
- Knowledge on data generation with OpenCV
- Improved skills on running unit tests and continuous integration
- Improved version control skills by having to fix several merge conflicts
- Improved teamwork and communication skills by collaborating closely with a partner