



ENPH 353 - Final Report

PREPARED BY

Ben Brown

Davis Johnson

DEC 12, 2020

1. Project Architecture

1.1 General Strategy

The general strategy for our robot was to use human informed heuristics and classical computer vision techniques for robot movement and license plate detection, and to use a convolutional neural network for character recognition.

The robot will complete one lap around the outside lane while detecting and recording license plates and stopping at crosswalks. Then, upon completing a full lap, the robot will merge into the inner lane. After entering the inner lane, the robot will avoid the truck and record the inner license plates. The robot will then circle in the inner lane until a predetermined 140 seconds of simulation time have elapsed.

Using this strategy, our objective was to score full points, which we accomplished.

1.2 Controller Structure

Our robot runs on two threads: the ***controller.py*** and ***image_processing.py***.

The ***controller.py*** thread handles driving and all communication to the `'/license_plate'` and `'/R1/cmd_vel'` rostopics.

The ***image_processing.py*** thread independently detects licence plates and sends predictions to ***controller.py***

1.3 Version Control

For version control, we operated using two main github repos: one for tasks related to training the neural network to recognize characters, and one for tasks related to the robots movement.

CNN Repo: https://github.com/johnsonadavis/text_identifier_cnn

Overview of Scripts:

- [training_generator.py](#) - A script for parsing images for characters to be used in training the neural network.
- [data_augmenter.py](#) - A script for producing additional augmented data from the initial data set by changing the brightness, noise, blur, brightness, scale, translation and skew of images.
- [text_identifier_training.py](#) - The script used to train the neural network using the dataset created by the other scripts.

Robot Control Repo: https://github.com/BenYKB/autonomous_driver_enph_353

Overview of Scripts:

- [controller.py](#) - The main script responsible for all robot operations. Implements a state machine for the robot, PID control and truck and crosswalk detection. Also communicates with the score tracker when *image_processing.py* sends license plate data via ROS.
- [image_processing.py](#) - The script responsible for interpreting images from the robot's camera feed, identifying and segmenting license plates and determining the neural network's output. It then verifies that license plate data is valid before publishing the data via ROS.

2. Robot Control

2.1 Driving

The robot detects the road using a HSV color mask. The difference between the centroid of the road and the center of the image is used to calculate the robot's position error.

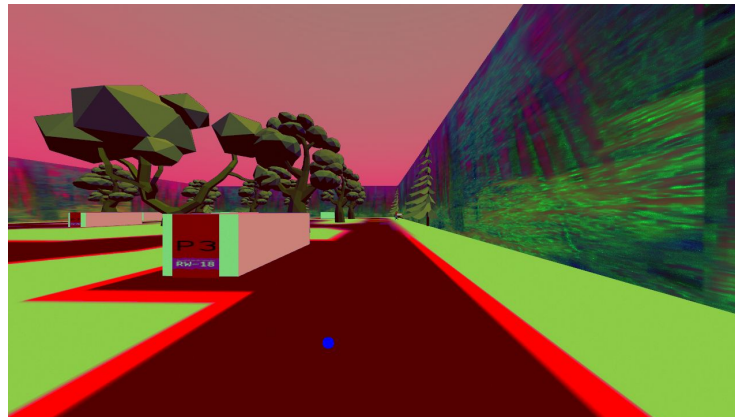


Figure 2.1.1 shows the frame from the robot's camera in HSV color space. The blue circle indicates the centroid of the road detected by the color mask. Notice that the lane markers are very distinct.

Using the road mask alone produces reasonable results; however, there is a risk of the robot merging into the inner loop. Thus, the robot also attempts to follow the white line on its right in order to keep its lane. Similar to the road, HSV masking is used to detect white lines and the centroid of the mask is used to generate error.

These two error estimates are combined using a weighted average with the road mask weighted higher. A proportional feedback loop then modulates the angular velocity of the robot. The linear velocity is held constant.

HSV color masking was essential to the driving and was used for the rest of the driving architecture. To make this process easier, we assembled [a table](#) of BGR-HSV-RBG conversions for various masks recovered from screenshots.

2.2 Pedestrian Detection

The robot operates as a state machine to allow for more complicated behaviors. The driving algorithm described above is the operation of the robot in the **FOLLOWING** state. Pedestrian detection requires two additional states: **CROSSWALK_WATCH** and **CROSSING**.

In the **FOLLOWING** state the robot will continuously scan for a red crosswalk indicator using HSV masking. When the red crosswalk indicator is detected near the bottom of a captured frame. Robot stops all motion and enters the **CROSSWALK_WATCH** state.

In **CROSSWALK_WATCH**, a narrow region near the centroid of white in the image is scanned for a threshold of purple (see Fig 2.2.1). Purple was chosen because the jeans on the pedestrian have a distinctive purple color. Once the pedestrian has been detected crossing the crosswalk, the robot transitions into the **CROSSING** state after a short delay. This is the safest time to do so as the pedestrian's crossing countdown is at its max.

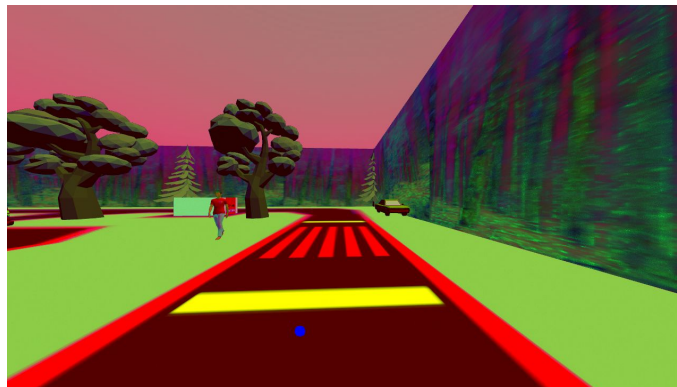


Fig 2.2.1 shows a crosswalk in HSV color space. Notice that the crosswalk indicator and the jeans are easily distinguishable in HSV.

The **CROSSING** state acts as **FOLLOWING** but does not scan for the crosswalk indicator. The robot will thus not mistakenly stop for the crosswalk a second time. This continues for 10 in-simulation seconds before returning the **FOLLOWING** state.

2.3 Merging

Merging requires two additional behaviours. **TO_INNER** performs a merge to the inner loop. **INNER_DRIVE** drives the robot once inside the inner loop.

TO_INNER is activated once the robot has detected all 6 outer license plates. In this state, the robot aggressively follows left hand side white road lines. Thus, from any starting position on the outer ring, the robot will merge into the inner road.

INNER_DRIVE is activated after **TO_INNER** upon detection of a car on the right hand side of the robot. During normal driving cars are always passed on the left, so this provides a clear indication that the robot must now be in the inner loop. Once driving in the inner ring the robot follows the right hand side white road lines.

Both these new states have difficulties when the road line they are following bends near parking stalls (see Fig. 2.3.1). Therefore, the robot uses HSV masking to detect when it is headed directly for a car and to change course if this is the case.

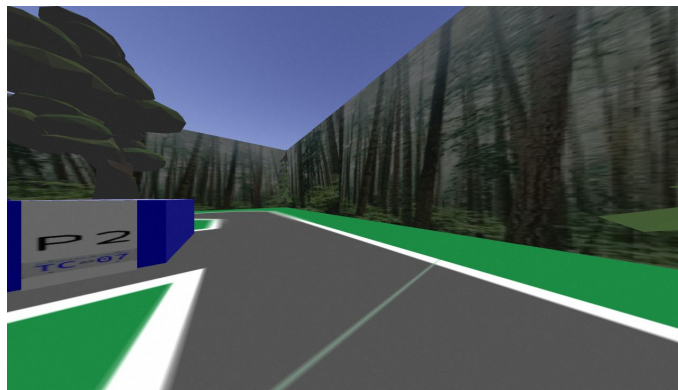


Figure 2.3.1 shows a situation in which the white road line on the left indicates a sharp left turn which the robot should not follow.

2.4 Truck Detection

The truck in the inner loop presents an additional challenge to detect and avoid.

Once the robot has decided to enter the inner ring, it will keep track of how many pixels in a frame are taken up by the truck's distinctive black wheels. After a certain threshold value, the robot will stop until the truck becomes smaller in the frame.

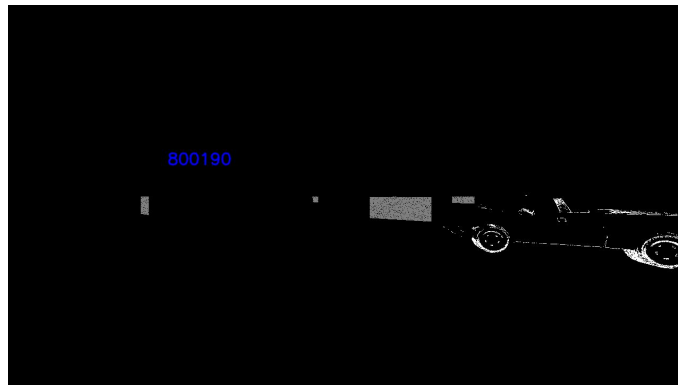


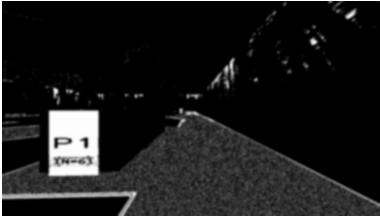


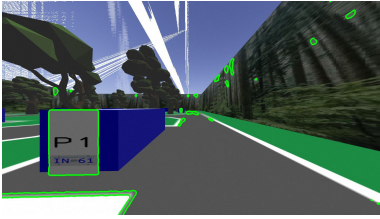
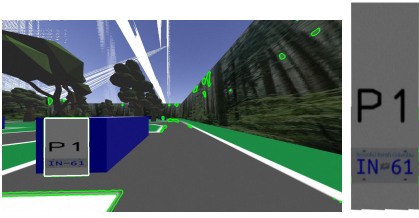
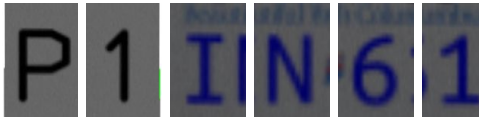
Figure 2.4.1 shows an overlay of HSV masks for the black of the truck's tires (in white) and the purple of the cars (in grey). The number in blue indicates the sum of the pixel values contained in the truck's wheel mask.

3. License Reading

3.1 License Plate Detection and Segmentation

Plate detection and segmentation was the most computer vision intensive process. The process pipeline is described below.

	<p>HSV A frame is converted to HSV color space.</p>
	<p>Mask Two HSV masks are used to identify the license plates backing on the cars. Multiple masks are required since some plates are darker than others.</p>

	<p>Filtering</p> <p>A combination of bilateral and gaussian filters are applied to smooth the image. This removes noise to help with edge detection.</p>
	<p>Thresholding</p> <p>To eliminate the haziness caused by the filtering, the image is thresholded to block unwanted noise. This allows for better edge detection.</p>
	<p>Edge Detection</p> <p>The canny edge detection algorithm is applied to identify contours.</p>
	<p>Contour Selection</p> <p>The contour with 4 sides and the largest area is identified.</p>
	<p>De-Skew</p> <p>Once the orientation of the contour is identified, it is de-skewed using a perspective transform. The image on the right is the de-skewed output from this frame. This is required since the wide-angle lens of the camera distorts the plate (see fig. 3.3.1).</p>
	<p>Slicing</p> <p>All contours are deskewed to identical dimensions from which letters are sliced (from predetermined locations) and sent to the CNN.</p>

3.2 Letter Detection

After each image has been resized to 100x150, each colour image is sent to a convolutional neural network (see part 4) which determines which character is present in the data. The image processor checks that the identified data conforms to the license plate grammar, then passes it to the controller via a ROS topic.

```
p = re.compile("^[P][0-9][A-Z][A-Z][0-9][0-9]$")
```

Figure 3.2.1 shows the Regex for testing license prediction grammar

3.3 Confidence

The image processing thread passes on license plate predictions and confidence values to the main controller continuously through a ros topic.

Confidence is based on the area of the quadrilateral from which the prediction originated.

The system initially had issues with plates near the edge of the frame (with high lens distortion) being misidentified. To fix this, the confidence values of plates near the edge of a frame are reduced.

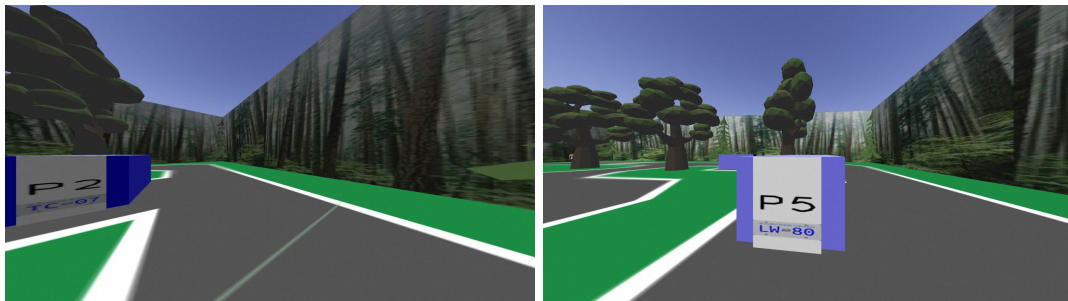


Fig 3.3.1 demonstrates the distortion of images near the edge of the frame in comparison to a plate near the center. Notice also that this warping makes the plate appear bigger which motivates the discounting these predictions.

The controller continually saves the highest confidence prediction and forwards new predictions to the **score_tracker.py** module if these predictions are of higher confidence.

4. Neural Network Development

4.1 Neural Network Architecture

For identifying text on license plates, we chose to implement a convolutional neural network or CNN. A CNN is appropriate for this task as we expect to identify text with

translational invariance - characters should be detected regardless of where they appear in the image.

We chose our network based on the Cats and Dogs example seen in class; our CNN has 4 convolution layers, 4 pooling layers, and finally flatten, dropout and dense layers. Note ReLU (Rectified Linear Unit) activation was selected for convolution layers, and softmax activation was used for the last dense layer.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 148, 98, 32)	896
max_pooling2d (MaxPooling2D)	(None, 74, 49, 32)	0
conv2d_1 (Conv2D)	(None, 72, 47, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 36, 23, 64)	0
conv2d_2 (Conv2D)	(None, 34, 21, 128)	73856
max_pooling2d_2 (MaxPooling2D)	(None, 17, 10, 128)	0
conv2d_3 (Conv2D)	(None, 15, 8, 128)	147584
max_pooling2d_3 (MaxPooling2D)	(None, 7, 4, 128)	0
flatten (Flatten)	(None, 3584)	0
dropout (Dropout)	(None, 3584)	0
dense (Dense)	(None, 512)	1835520
dense_1 (Dense)	(None, 36)	18468
Total params: 2,094,820		
Trainable params: 2,094,820		
Non-trainable params: 0		

Figure 4.1.1 shows the summary for our convolutional neural network as output by keras

4.2 Training Data Generation

We collected a manual data set by moving the robot around the simulated world and taking screenshots of license plates using ***rqt_image_view***. We repeated this process until we had multiple captures of each character that could appear on the license plates. Two such captures are shown below:

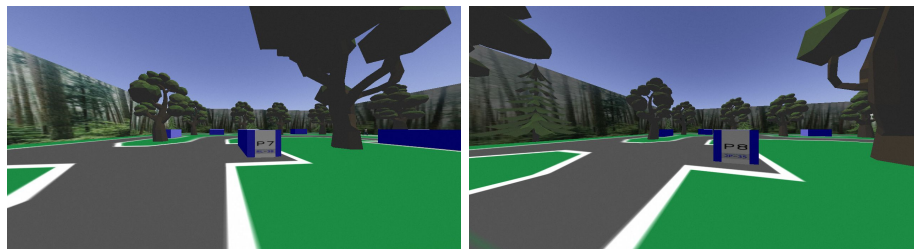


Figure 4.2.1 shows two example screenshots taken from the robot camera to be used for training data

Then, we ran **training_generator.py** - a script that performs the identical license plate detection and segmentation process as the robot (see 3.1), and saves the captured characters with the camera distortion removed. Some examples are shown below:



Figure 4.2.2 shows example characters produced by training_generator.py from manual captures

Given we were creating this data set manually, we did not have a large enough data set to expect accurate results after training our network. Thus, we turned to data augmentation.

In our script **data_augmenter.py**, we took advantage of the **ImageDataGenerator** library from **keras**, in order to increase the size of our data set. Our script took our input images and performed modifications on them including: horizontal and vertical translation, rotation, skew, noise, blur and brightness adjustments. Our script increased the size of our dataset from an initial size of 714 images up to 13632 augmented images.

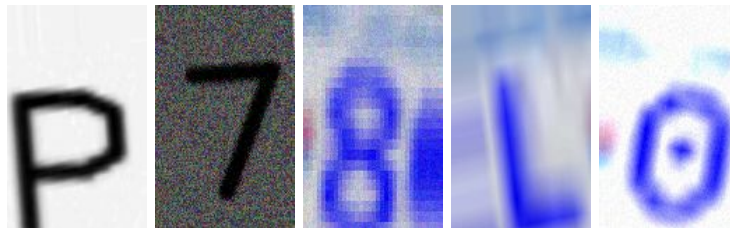


Figure 4.2.3 shows images produced by performing data augmentation on the initial dataset. Observe changes in brightness, noise, blur, brightness, scale, translation and skew.

4.3 Training and Validation

To train our model, we trained for 20 epochs with a batch size of 16, a validation split of 20% and categorical cross-entropy loss using **text_identfier_training.py**.

After the model finished training, our script produced the following graphs to evaluate the performance of the model:

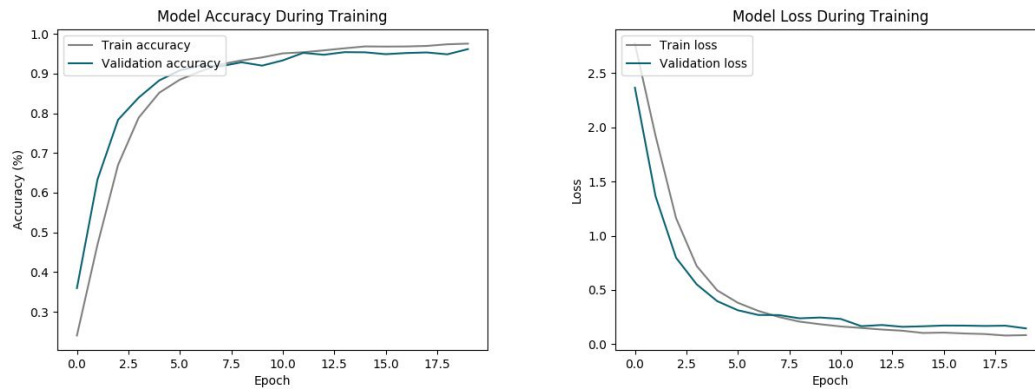


Figure 4.3.1 shows the accuracy and loss of the model as it is being trained. Observe that both the training accuracy and validation accuracy approach 95%, with the validation accuracy being slightly lower. Also, note that both the training loss and validation loss approach 0.25, with the validation loss being slightly higher - a sign of optimal fitting.

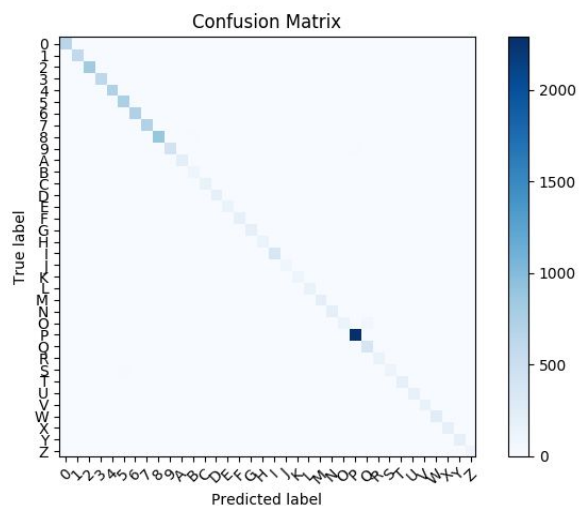


Figure 4.3.3 shows the confusion matrix of the trained model. The confusion matrix shows the relationship between the true letter classification and that predicted by the model. Since all visible points are on the diagonal, we see that the model is mostly identifying characters correctly. The differences in shading along the diagonal are due to different concentrations of letters being present in the data. Note that numbers and the letter P appear significantly more than other letters.