

---

# Taped: An Imitation Learning and Computer Vision Approach to the ENPH-353 Game

---

**Ruiheng Su**  
Engineering Physics  
The University of British Columbia  
Vancouver, BC, Canada  
ruihengsu@alumni.ubc.ca

**Dora Yang**  
Engineering Physics  
The University of British Columbia  
Vancouver, BC, Canada  
dyang22@student.ubc.ca

## Abstract

We report on the development of Taped: an imitation learning and computer vision approach to the ENPH-353 Game. Taped completed the outer and inner in 45 seconds (the second fastest performance in the 2020 competition), amounting to 43 points total.

## 1 Introduction

Taped is another of the many approaches to the ENPH-353 game (further details here [1]) that utilizes a mix of machine learning and computer vision methods written using the Keras [2] and OpenCV2 [3] libraries in Python. The source code for Taped can be found at:

[https://github.com/sillyPhotons/ENPH\\_353](https://github.com/sillyPhotons/ENPH_353)

We used free GPU runtimes on Google Colaboratory to train our neural networks.

## 2 Methods

### 2.1 Navigation

We were inspired by previous work by Bojarski et al. [4] who reported on an imitation learning approach to predict vehicle steering angles given images taken by three on-board cameras.

In the ENPH-353 game, manual control of the agent's movements used the keyboard. The linear and angular velocity of the moving agent could also be varied using the keyboard. A plugin application translated direction commands received from the keyboard to a combination of linear and angular velocity value that are published to the ROS `cmd_vel` topic.

As a regression problem, our neural network must predict both the linear and angular velocity of agent for any given image example. Further, it proved to be difficult to manually vary linear and angular velocities consistently when taking data. This meant that our network would need to model a function with many steps and jumps.

Instead of attempting to vary the velocities, we used a fixed combination linear and angular velocity and only controlled the robot using direction key presses. This allowed us to restate a regression problem as a classification problem since there is now a finite set of outputs.

Further restricting our direction controls to 5 of 9 possible keys, we trained our neural network to predict one of 5 possible driving commands given an image input from a single on-board camera. The network architecture is shown in figure 1.

To train the network, we wrote a python script to save each frame recorded by a front facing camera on the agent while driving the agent around the competition area manually. Each image name contained the linear velocity and angular velocity of the agent when the image was taken.

Achieving an accurate neural network required many cycles of repeated data acquisition, training, and testing. Our workflow is outlined in figure 2.

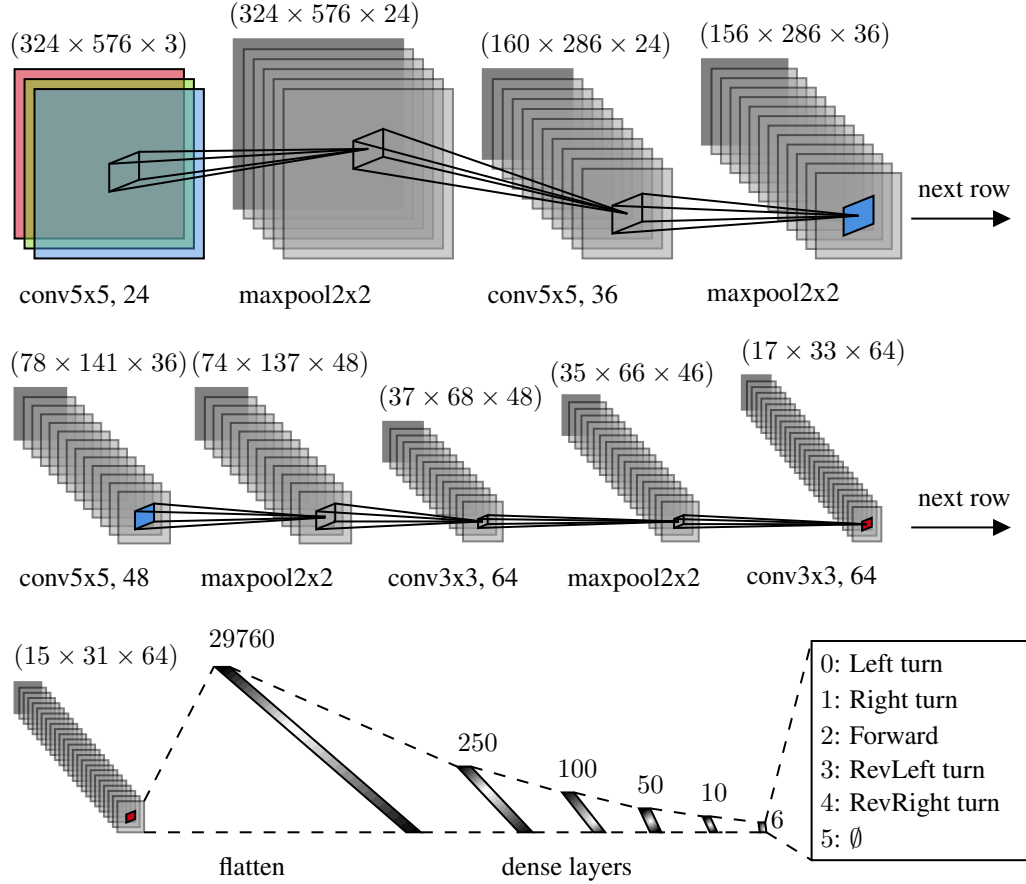


Figure 1: Our imitation learning architecture. Given a image (of dimensions  $324 \times 576$  pixels), the neural network outputs one of 6 possible driving commands.

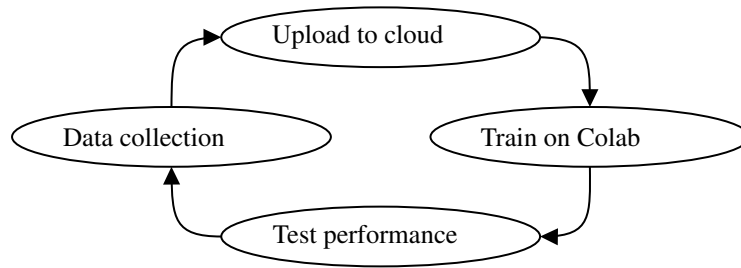


Figure 2: Each cycle began with data collection. The image data is then uploaded to Google drive. We then trained our network on Colab. After saving the model, we moved on to test the performance of our network to navigate the competition area. We had to repeat the cycle and acquire more data until the agent showed no unexpected behaviour.

We reasoned that the best way for agent to navigate the competition area was to first complete the outer ring counterclockwise, then turn into the inner ring at the first intersection after the lower crosswalk. To accomplish this consistently, we trained a neural network for the agent to navigate

the outer ring, and a separate network that took care of steering the agent into the inner ring, and navigating in the inner ring.

The composition of training examples for the outer and inner neural networks are shown in figure 12 in the appendix.

The neural network hyper-parameters for our neural networks are shown in tables 2 and 3 of the appendix.

After perfecting our inner and outer neural networks, we integrated them into a ROS node, which we named "LaneFollower". A flow chart representing our implementation of LaneFollower is shown in figure 15 of the appendix.

## 2.2 Obstacle detection

The ENPH-353 game has two main obstacle detection challenges: two pedestrian NPC that irregularly crosses the upper and lower crosswalks, and a truck NPC that drives clockwise in the inner ring with randomized velocity. The agent must navigate the outer and inner rings without hitting the NPCs.

To ensure that our agent only drives past the crosswalk when the pedestrian is not crossing, we first use a red mask to stop the agent whenever it comes up to a crosswalk. After stopping, we find the masks of a gray filter that has been tuned to the color of the pedestrian's clothing. The algorithm is detailed in figure 3.

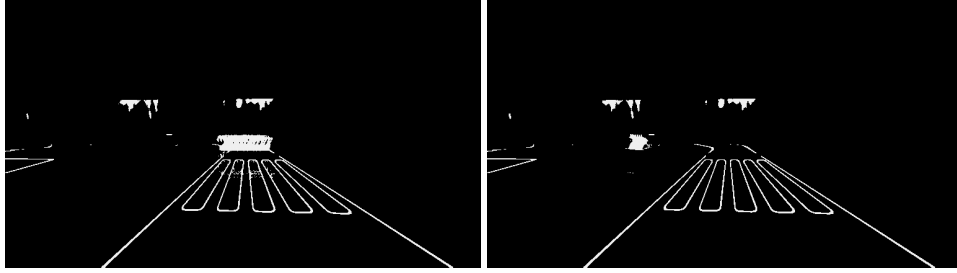


Figure 3: Left: average of 15 gray masks when the NPC is moving. Right: average of 15 gray masks when the NPC is stationary. Mapping each pixel to  $[0, 1]$ , the sum,  $S$ , obtained from adding the value of every pixel together is greater in left image than in the right. Comparing  $S$  to an experimentally determined threshold helped us determine when to steer the agent forward.

To avoid hitting the truck NPC in the inner ring, we tuned a gray filter to match the color of the truck. Comparing the sum of all pixel values in the masks obtained by applying the filter to camera input (much like our algorithm in figure 3) to a threshold value helped us determine whether the truck was within the field of view of our agent.

We configured our agent to always drive faster than the truck. When the truck is detected in the field of view of the agent, we attenuate the linear and angular velocity of our agent by a factor of 4 to ensure that we will never crash into truck.

We integrated the pedestrian and truck detection algorithm into a ROS node called "ObstacleDetector". We detail the logic of ObstacleDetector in figure 16 of the appendix.

## 2.3 Fitting everything together

LaneFollower was responsible to output vehicle steering commands by publishing to `cmd_vel` given camera input. ObstacleDetector takes the predicted driving commands, and depending on whether NPCs are nearby, may alter, or override the predicted steering commands.

General flow of information is shown in figure 4.

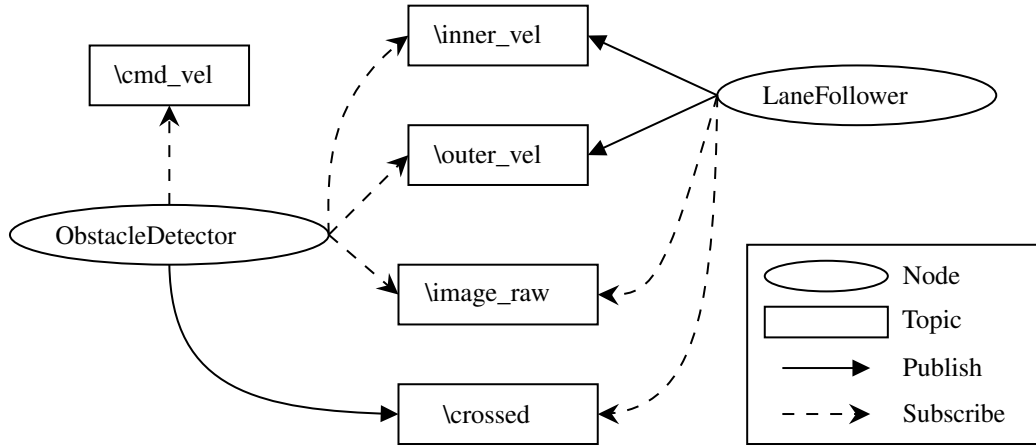


Figure 4: Information flow between nodes and topics as the autonomous agent navigates the competition area.

## 2.4 License plate reading

The ENPH-353 game contains 8 parked cars. Each car is labelled with a parking spot ID number and a license plate. Parking spots and their corresponding license plate must be correctly identified to score points.

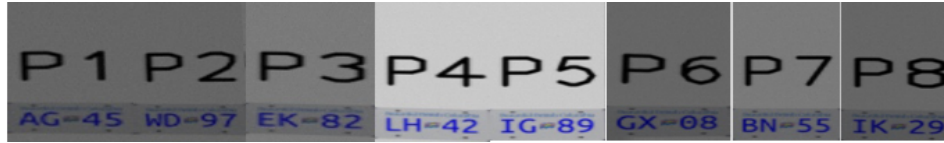


Figure 5: Parked car plates extracted from competition simulation.

From the camera's raw image feed, these license plates were detected and cropped. Once the plate is isolated, a series of convolution neural networks (CNN) were used to identify and classify the characters on the parking plates.

### 2.4.1 License plate detection

The `process_image` node subscribes to the `image_raw` topic of the robots camera. It then uses the `PlateDetector.py` script to extract a license plate from the image suitable for the `PlateReader` CNNs.

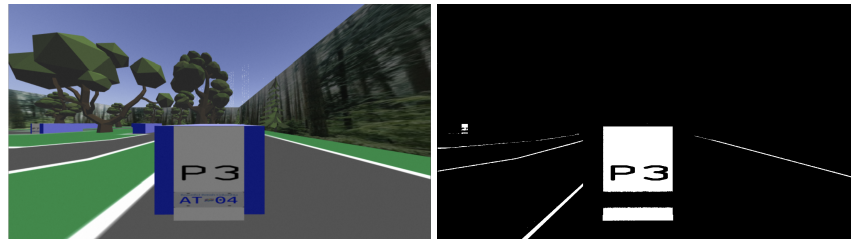


Figure 6: Left: Raw image from robot camera feed. Right: Image after grey mask is applied

A grey mask is applied to the raw image. Pixels that match a target threshold for the license plate are white in the masked image and the rest of the masked image is black. We then use `cv2.contours` to detect contours in the masked image.

If 5 contours are detected, the script processes the contours to determine if there is a valid license plate. It begins by analyzing the largest contour as that is most likely to be the plate and continues analyzing until a valid plate is detected from the image.

Next, `cv2.BoundingRect()` is used to get the coordinates of the contour corners and the image is then cropped along those coordinates to extract the license plate. This image then undergoes some tests to ensure that it is suited for the PlateReader to get an accurate prediction.

The following processes are checked/applied to see if the license plate is valid:

1. License plate meets minimum width, height and area values.
2. It doesn't contain too much blue from the sides of the plate.
3. Perspective transform to de-skew the image if needed.

If the above conditions are met the `process_image` node publishes the valid plate to the `license_plate` topic which the PlateReader node subscribes to.

## 2.4.2 Character segmentation

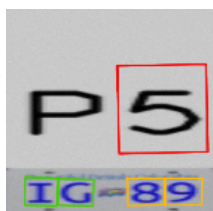


Figure 7: Plate taken in simulation with parking ID bordered in red, plate letters bordered in green and plate numbers bordered in yellow.

The parking spot IDs range from 1 to 8 while the license plate is comprised of 4 characters. The first 2 characters on the left of the plate are randomly generated upper case letters and the two characters on the right of the center are randomly generated digits from 0 to 9.

Three CNNs were built and trained to handle identifying:

1. Parking spot IDs
2. License plate letters
3. License plate numbers

In early iterations of the PlateReader only 2 CNNs were used. One for the parking spot IDs since it is in a different location and has different dimensions than the plate characters. A second CNN was used to classify all the characters on the plate. Splitting the license plate into letters and numbers and training separate CNNs reduces the complexity. Instead of one CNN categorizing between 36 characters, two different CNNs were implemented to categorize between 26 letters and 10 numbers respectively.



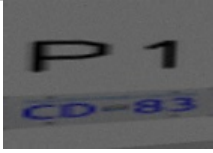
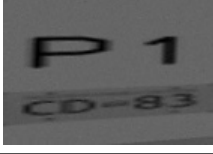
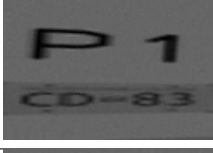
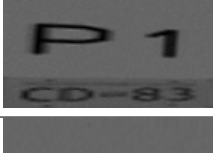
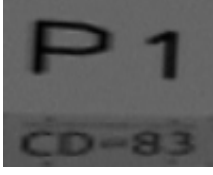
Figure 8: Plate generated using script and character slices segmented from generated plate.

Random license plates were generated using a python script and then cropped into the above slices and stored in 3 data sets. Each image has a corresponding label assigned by a one-hot-encoder.

## 2.4.3 Generating test data set

The plate detector and lane follower were used in the simulation to save images of the license plates. This was repeatedly done to generate sizeable data set for testing. Using OpenCV2 [3]. Table 1 outlines the transformations that were applied to the input image.

Table 1: Sim image processing

Transformation	Image
Image from simulation	
Grey-scale	
Perspective transform to de-skew	
Crop to remove black space at bottom	
Resize to 100 by 100 pixels	

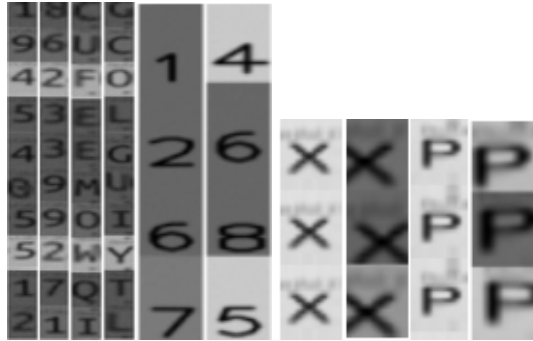


Figure 9: Left: Plate numbers, plate letters and parking ID numbers cropped from transformed test data. Right: Comparison between original generated data and augmented data for training.

Before training the CNNs, ImageDataGenerator from Keras [2] was used to transform the data to be more similar to the test data. Furthermore, applying random perturbations to the training data makes the CNN more robust and increases the generalizability of the model. See table 6 in the appendix for transformation parameter values.

#### 2.4.4 Data normalization

Training, validation and test data sets were all normalized before acting as inputs to the CNN. This technique reduced training time and allowed the models to converge faster. Grey-scale images were used as input data since analyzing only one channel also decreases the amount of information the

CNN needs to process. Grey-scale pixel values ranged from 0 to 255 so all inputs are normalized by a factor of 255.

## 2.4.5 Building CNN models

To build the neural networks a series of different types of layers were used:

**Conv2D** extract features from the image and learn specific characteristics about the outputs. Due to the small size of the input images 2 by 2 convolution kernels were used. The amount of output layers of each Conv2D layer was determined through testing and the combination of 8 to 32 to 128 was found to be most effective.

**MaxPool2D** layers were added following Conv2D layers to reduce image dimensionality when more layers are produced. This results in a lower resolution mapping while still maintaining features.

**BatchNormalization** normalizes layers within the neural network and allow the model to converge faster. From testing it was found that the models were more effective when these layers directly preceded Conv2D layers.

**Dropout** applies data regularization to prevent over-fitting. During each batch of training it randomly drops and ignores 30% of the parameters. This results in multiple independent internal representations being learned by the network.

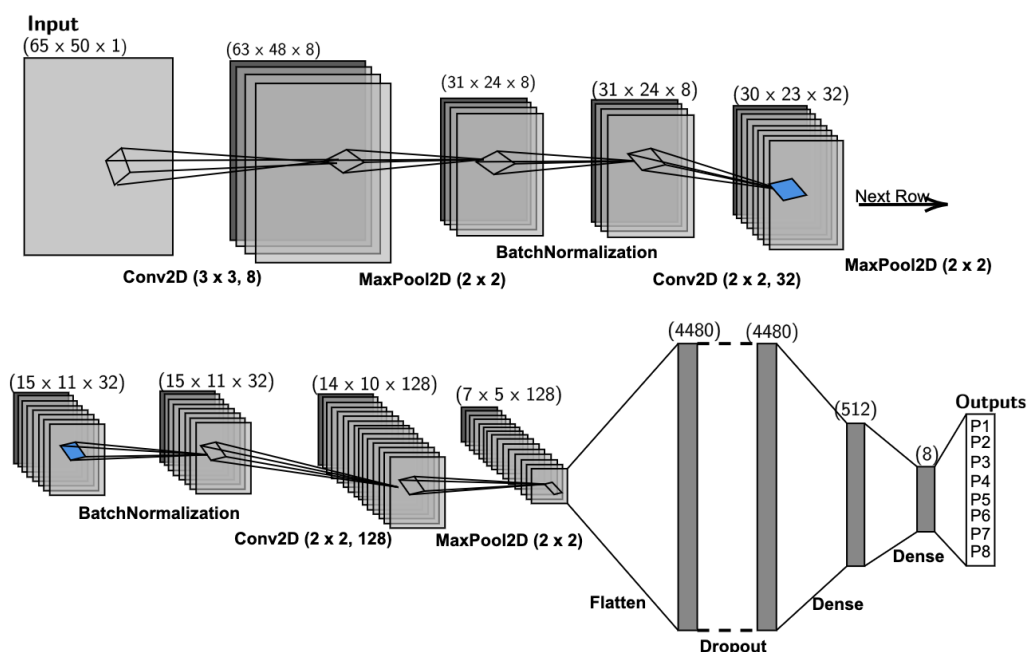


Figure 10: CNN architecture for parking spot ID classification. Given a  $65 \times 50$  pixel image of the parking ID the network returns the number of that spot. The CNNs used to predict plate letters and plate numbers follow the same architecture but differ in input dimension.

## 2.4.6 Using the PlateReader in the simulation

See figure 17 in the appendix for a flowchart outlining the implementation of the PlateReader node. When testing we noticed the following issues with the PlateReader:

1. accuracy greatly improves when input image dimensions are greater than 100 by 100;

2. the very last image input before the robot passes a parked car is the most accurate for that car;
3. parking spots 2 and 4 have the least input data due to a different lighting than the rest;
4. garbage data can pass filters and give an inaccurate prediction and replace a correct one.

We developed a publishing algorithm that considers these factors to publish the most accurate readings for each plate.

Every time the node receives an image message it makes a prediction for the ID and the license plate and stores this pair in a list. After 45s (the ideal time it takes to complete the whole course) the node publishes its predictions. For each parking spot the node publishes only the most frequent plate prediction. If a parking spot has multiple plates with the same number of predictions the node will publish the most recent of those predictions.

### 3 Results

#### 3.1 Navigation

The training history of the outer and inner neural networks used for navigation are shown in figures 13 and 14 of the appendix. The final training and validation accuracy for the outer and inner networks are listed in tables 4, 5 of the appendix.

#### 3.2 License detection

Each model trained for 10 epochs. See table 7 in the appendix for the training history of each model.

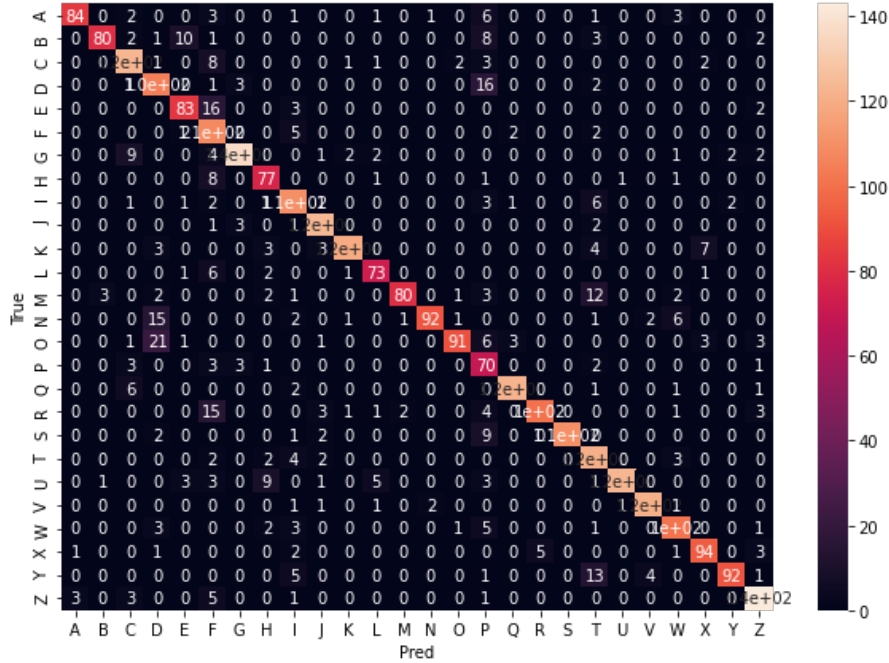


Figure 11: Plate letter confusion matrix

Confusion matrices helped in identifying which characters the models were predicting incorrectly. For example from figure 11 it can be noted that the letter prediction model is sometimes unable to tell the difference between the letters 'D' and 'O'. Confusion matrices for the other two models can be found in the appendix.

Based on the confusion matrix of a model appropriate changes were made to improve the accuracy of the model. For example:



1. adding more data for a specific character if it was frequently misidentified;
2. making variations to training data preprocessing if the model was incorrectly predicting multiple characters;
3. modifying layer order or attributes.

## 4 Conclusion

We demonstrated an approach to the ENPH-353 game using imitation learning and computer vision algorithms. Our navigation algorithms were able to successfully traverse the course in under 50 seconds. The license plate detection was able to correctly identify 6 out of 8 plates. Our approach - Taped - scored a total of 43 out of 57 points.

## References

- [1] Miti Isbasescu. Enph 353 2020t1 ubc parking - competition notes.
- [2] François Chollet et al. Keras. <https://github.com/fchollet/keras>, 2015.
- [3] Itseez. Open source computer vision library. <https://github.com/itseez/opencv>, 2015.
- [4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. 2016.

## 5 Appendix

Table 2: Summary of network parameters used to navigate the outer ring of the competition.

Optimizer	Learning rate	Batch size	Train/Test split	Total examples
RMSprop	$1.0 \times 10^{-4}$	20	0.85	4385

Table 3: Summary of network parameters used to navigate the inner ring of the competition.

Optimizer	Learning rate	Batch size	Train/Test split	Total examples
RMSprop	$1.5 \times 10^{-4}$	50	0.85	5052

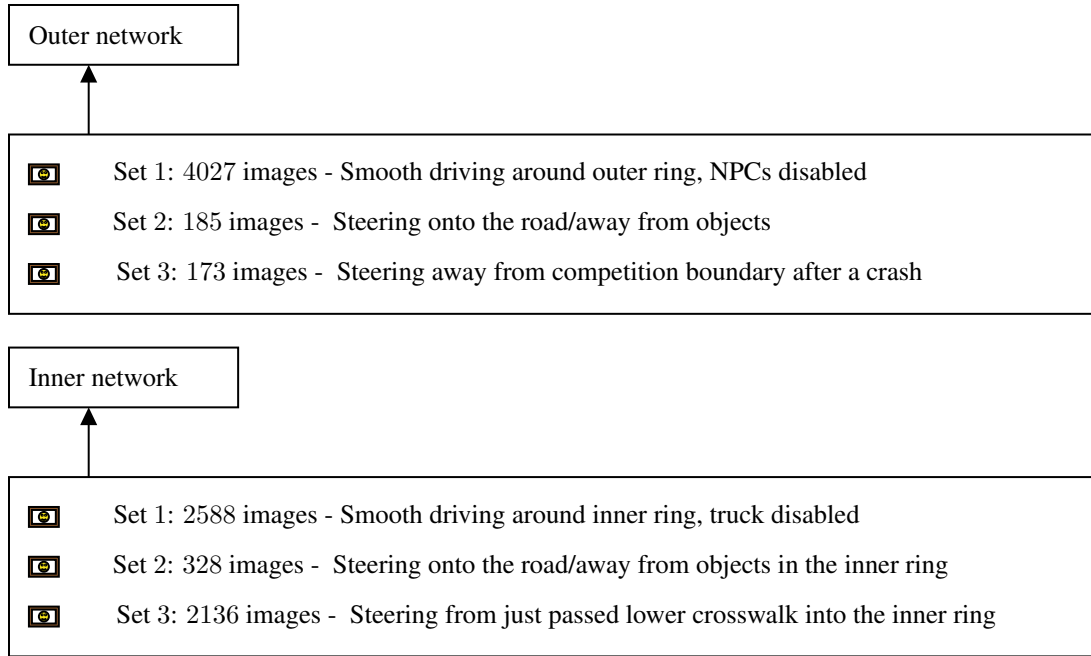


Figure 12: The composition of training examples and the cases they addressed.

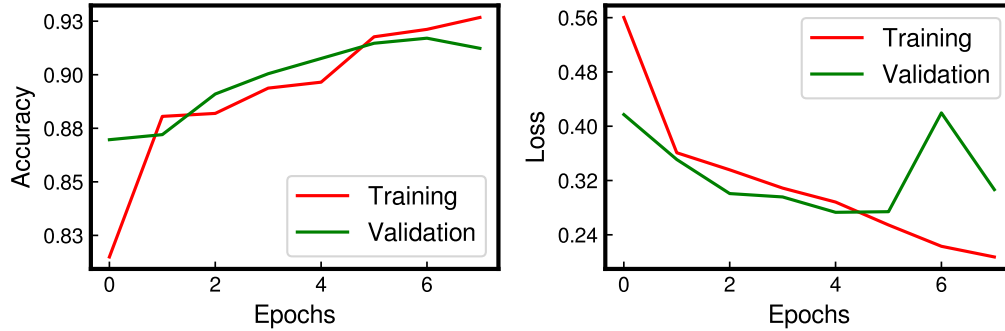


Figure 13: The training history of the the model used to navigate the outer ring in the competition area. We used an early stopping scheme to stop the training at 8 epochs.

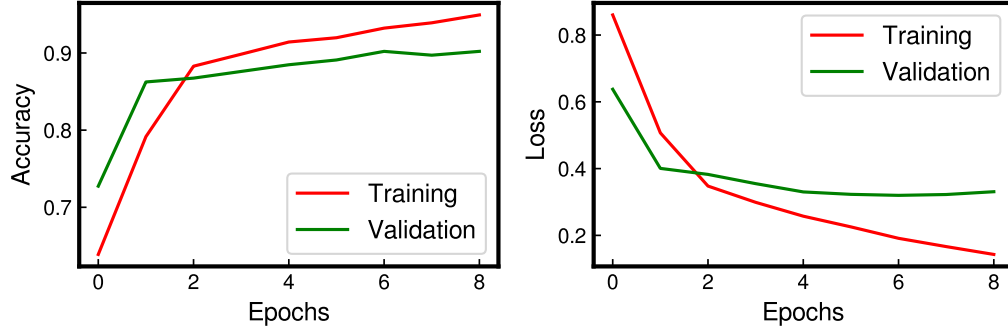


Figure 14: The training history of the model used to navigate inner ring. Training stopped at 9 epochs.

Table 4: Final performance of the outer navigation network on the training and validation sets.

	Loss	Accuracy
Training	0.20	0.93
Validation	0.30	0.91

Table 5: Final performance of the inner navigation network on the training and validation sets.

	Loss	Accuracy
Training	0.14	0.95
Validation	0.33	0.90

Table 6: Image Preprocessing Transformations

Horizontal shift	$\pm 10\%$
Vertical shift	$\pm 10\%$
Rotation	$\pm 3^\circ$
Skew	$\pm 3^\circ$
Brightness	40% – 80% of original brightness
Zoom	1.25X to 1.35X original

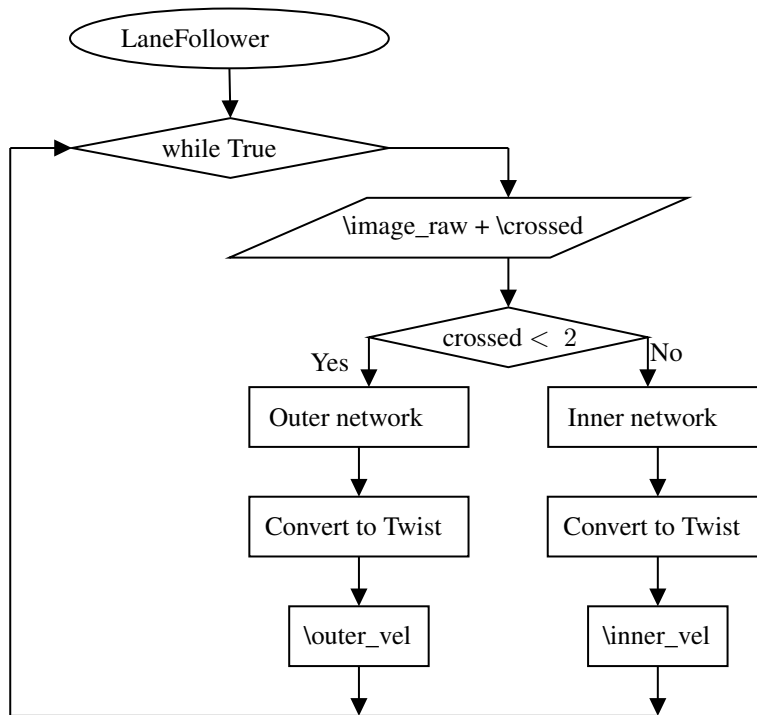


Figure 15: Flow chart for the LaneFollower node. Given camera input and the number of times the agent has crossed crosswalks (from topics called `img_raw` and `crossed`), the image is resized, and fed into either the network used for navigating the outer ring, or the network used for the inner ring. Each network returns a 6-element array of probabilities, which is translated into a `Twist` object. This object represents linear and angular velocity commands that the agent will follow and is published to the `inner_vel` or `outer_vel` topics.

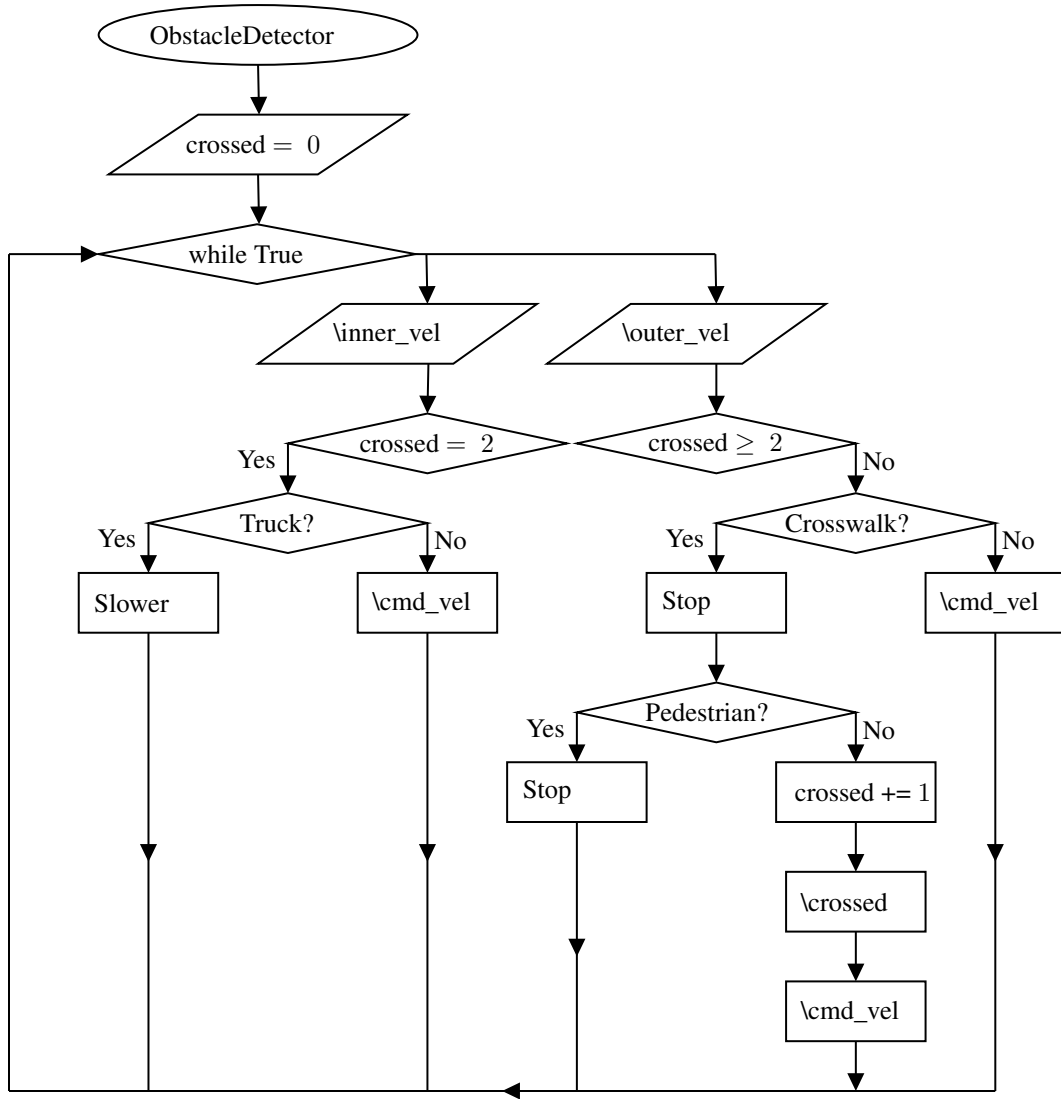


Figure 16: Flow chart for the ObstacleDetector node. Once the agent crossed both crosswalks, the node switches from publishing velocity commands from the `inner_vel` topic to the `outer_outer` topic.

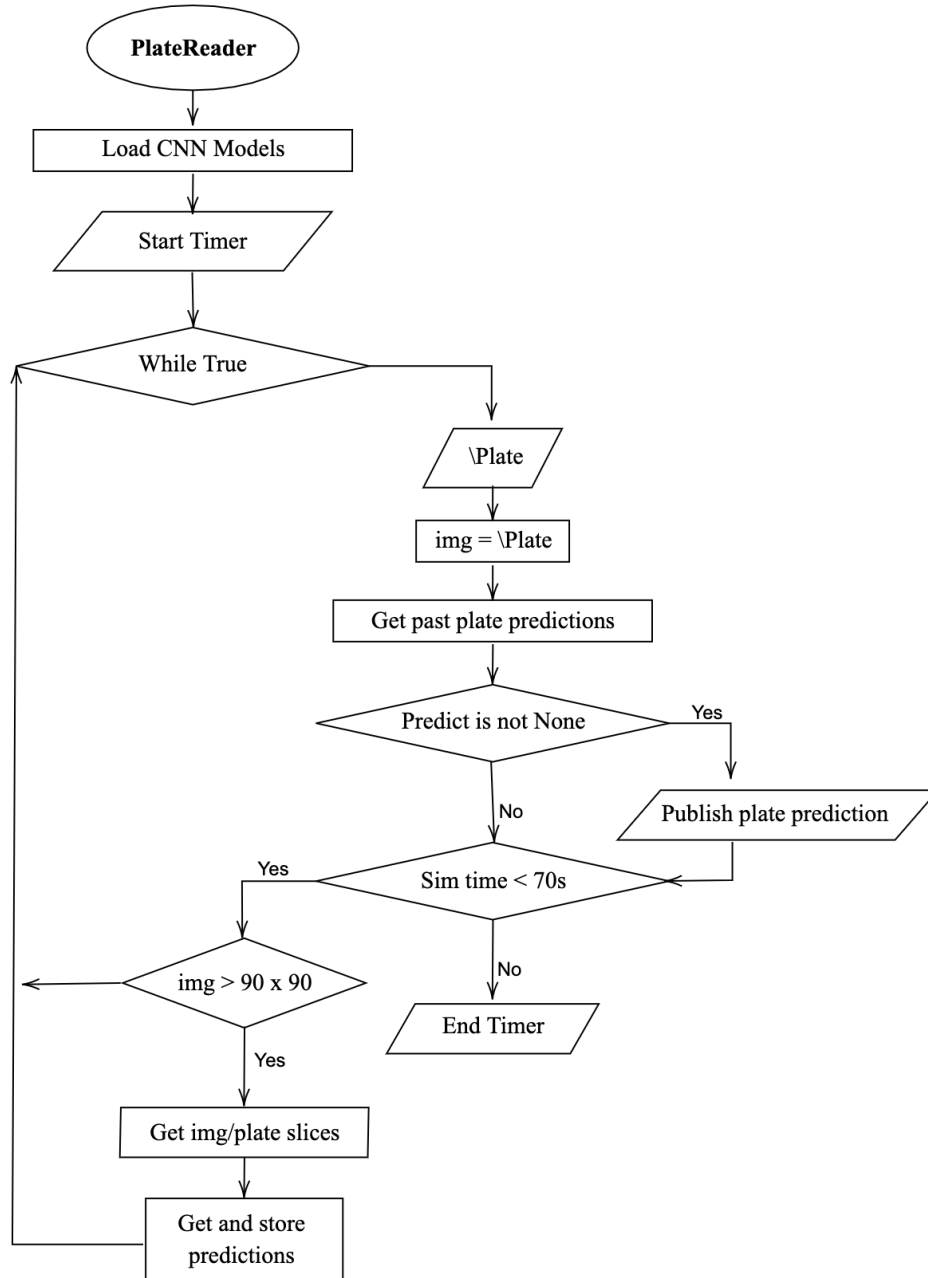
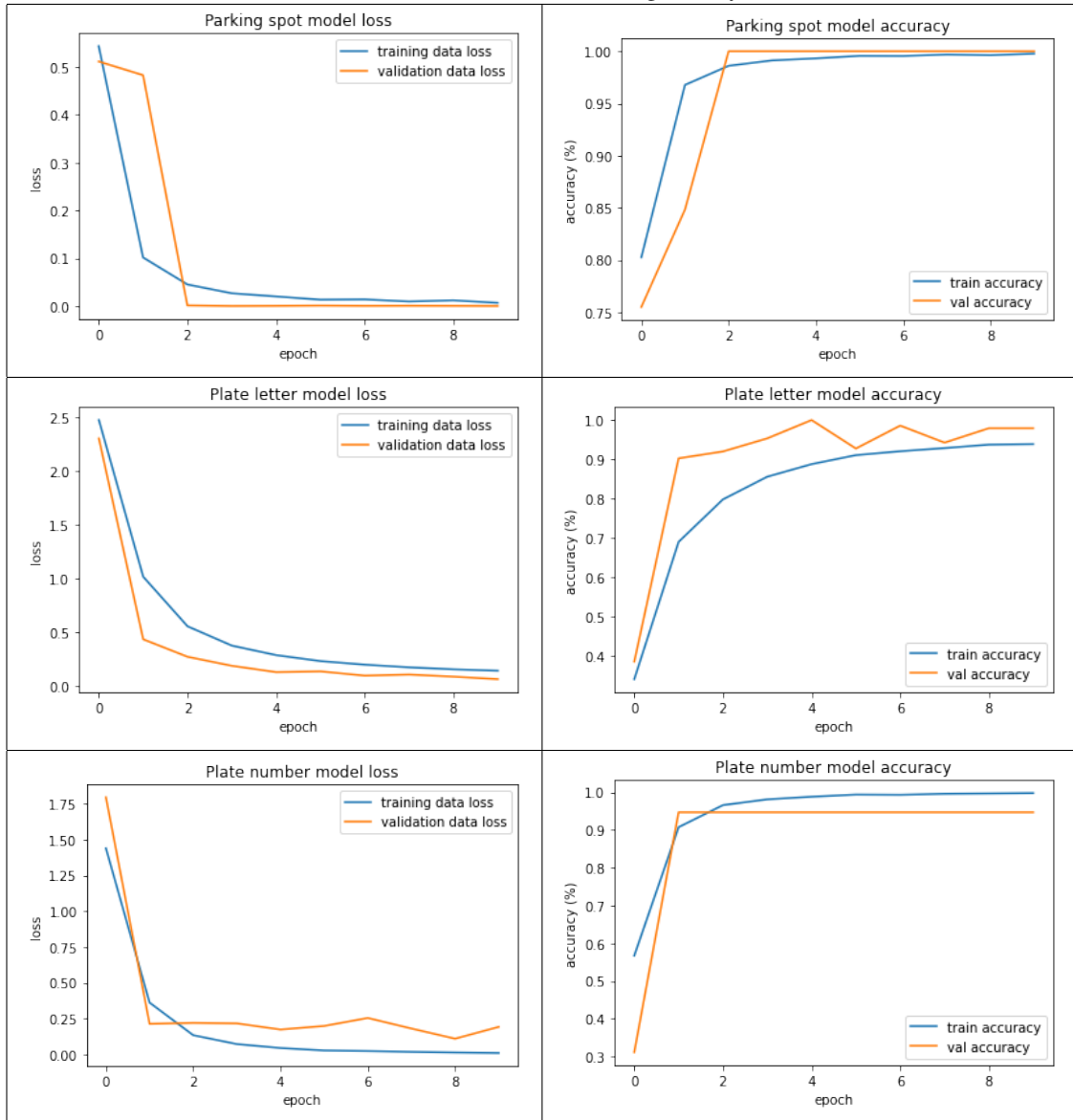


Figure 17: Flow chart for PlateReader node. This node takes input image and makes a prediction for the license plate and parking ID number. When triggered it publishes predicted spots and their associated plates to the score tracker.

Table 7: PlateReader Model Training History



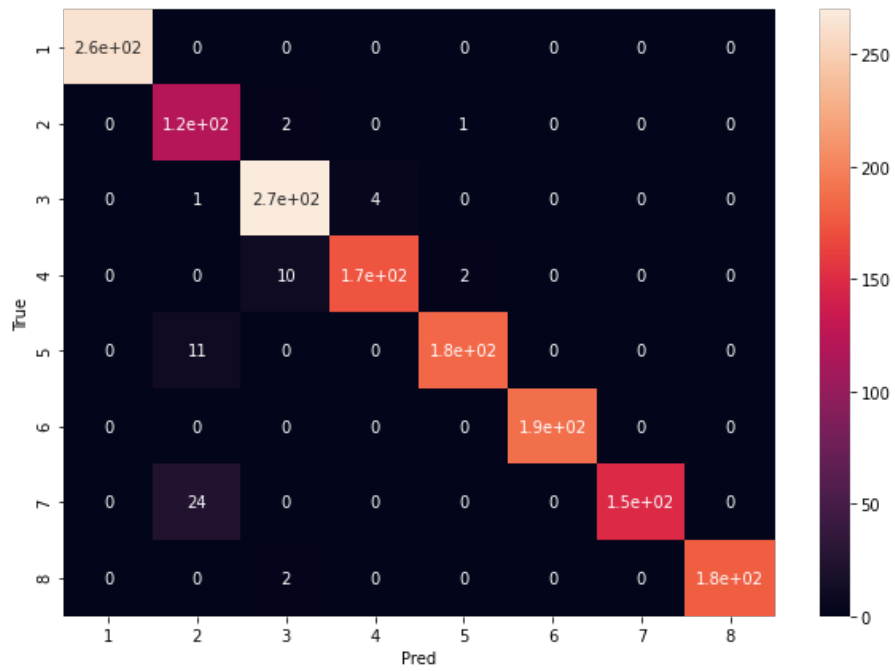


Figure 18: Parking ID confusion matrix.

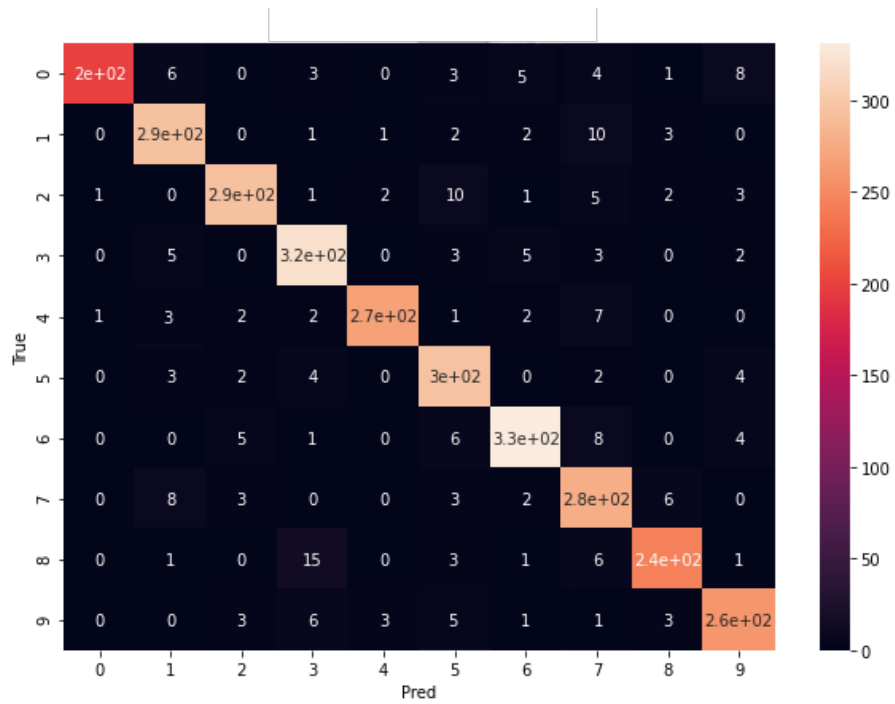


Figure 19: Plate number confusion matrix