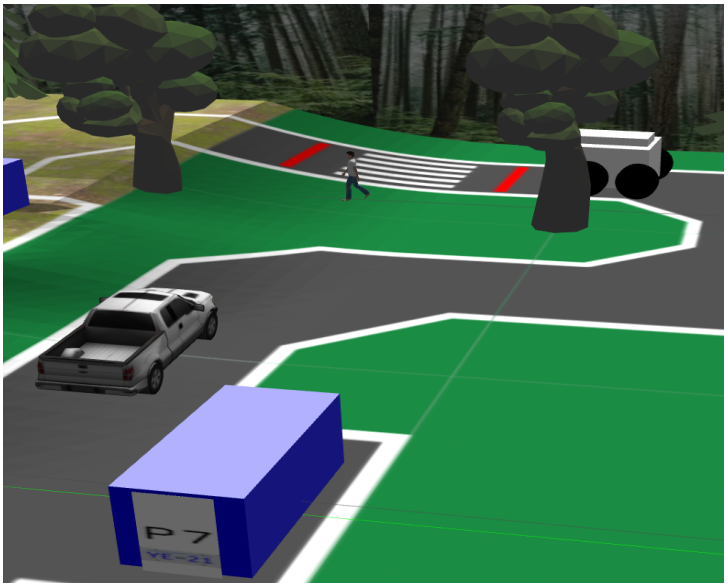
Design Report: Self-Driving Car Simulation

Chris Yoon and Julian Lapenna



Outline

Project Overview

The purpose of this project was to explore and apply control systems involving data classification techniques and machine learning. We performed these tasks by developing autonomous control software for a driving robot in a simulated environment that follows traffic rules and collects license plates and associated parking ID data.

This paper will outline the approaches and strategies our team used from development up to our final model and the analysis techniques we used throughout our process. Our high level strategy was to use fully artificial intelligence based control software and learn as much as we could about them. To achieve this we prioritized capturing large amounts of clean data for training precise and accurate models.

Our approach for the competition was to consistently score maximum points on the outer ring and then transition to the inner ring using the algorithms already developed. Our final controller succeeded in doing this with a total of five convolution neural networks (CNNs). Each neural network model will be explained more in detail in their respective sections along with example training data, prediction outputs and error-analysis.

Strategy

Our main strategy was to drive using imitation learning, slow down while reading license plates to capture clear data, and enter the inner ring after a set amount of crosswalks were reached. For reading license plates, we used three CNNs: one for reading the parking ID, one for the license plate letters and one for the license plate numbers.

Upon approaching a car, we first read the parking ID, then the license plate. The predicted license plate went into a set under the corresponding ID label. Since we took multiple reads of each plate, the set corresponding to each ID label would have multiple license plates, usually with some being incorrect.

To determine which license plate to publish we took the highest frequency license plate in each ID set, and in the case of a tie we took the license plate with the higher cumulative prediction probability output from our model. Since publishing was slow and took time, we waited until we were stopped at a crosswalk to calculate the license plate for a given ID and publish it. Lastly, for the inside since there was no place to stop, we waited until we'd seen five instances of each ID and then stopped, published and ended the timer.

Character Recognition Modules

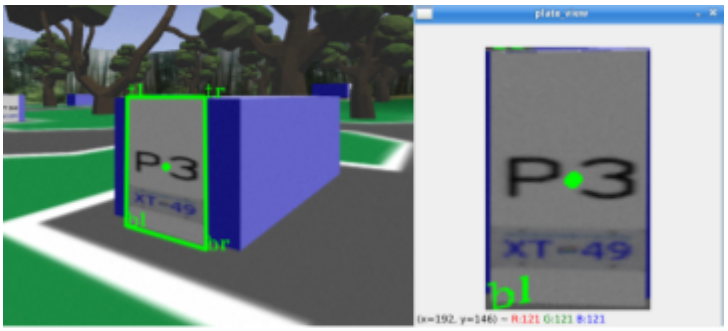
Our license plate reader system was a combination of three CNNs. One was trained for recognizing parking spot IDs (ID model), one for license plate letters (alpha model) and one for license plate numbers (number model). The approaches for gathering data and their architectures were quite similar so we will focus on the general process and address parts that were unique to a specific one individually. At a high level, we explored the environment using different OpenCV processing techniques such as HSV filtering, blurring, and binary thresholding to efficiently gather data and process it during the competition.

Pre-Processing

For all three character recognition CNNs, we first identified when a car's license plate was nearby and in our field of view. This was done by blurring the input image for consistency, then creating a bandpass filter in HSV color space to capture gray values.

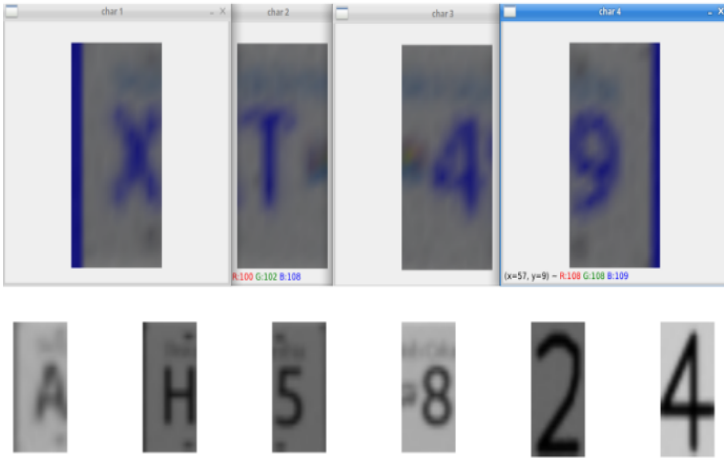
Preprocessed image highlighting license plates

The image on the left shows that the front of the car is clearly distinguishable from the surrounding environment. The next task was to isolate the front of the car. By selecting the largest contour in view we could apply OpenCV's contour approximation and save the vertices of the quadrilateral. In the case where the largest contour was something else, it wouldn't have four vertices to transform and we could discard that frame.



Car front contour approximation and perspective transform

The vertices were then used to extract a clear picture of the license plate from the original input image. Using homography, we transformed the image to isolate the license plate. Next, assuming we got a clear perspective transform of the front of a car, the parking ID and license plate characters would always be in the same positions. Leveraging this we cropped at specific points to get images of the individual characters we tried to read.



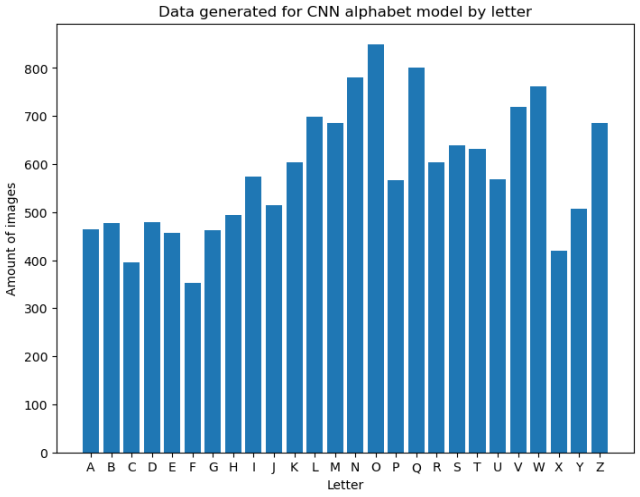
The only other preprocessing performed was to reduce the dimensionality of our input character images. The main goal was to remove as much information as possible without obscuring the character itself. Therefore, we applied a black and white filter and reduced the size to 1% of the original. This then left us with images that were small enough to be processed quickly and with enough information to clearly contain a distinct identifiable character. This final preprocessing also helped with edge cases where the image was blurry because as you can see in the first image, the letter 'A' was blurry but at a smaller scale most of the blur was removed.

Data Generation

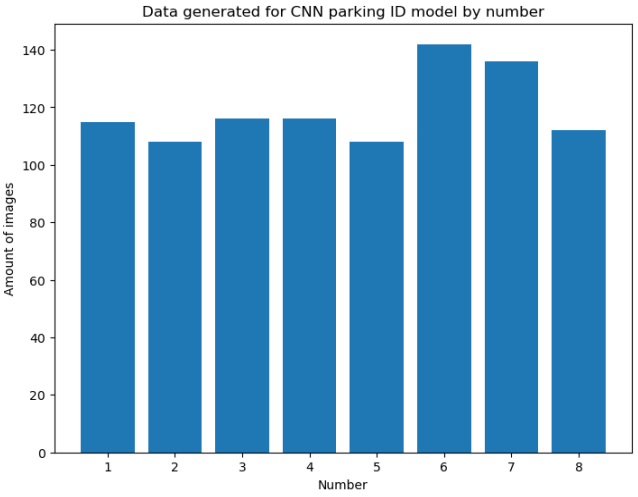
Once our preprocessing routine was set up, capturing and labeling data was straightforward. We edited the license plate generator script to load all license plates as the character we specified. Starting with 'A' and '0' we systematically went through the alphabet and all 10 digits, saving hundreds of images for each character. We captured a variety of blurred and sharp images, dark and light images, and images

from the left and right sides. Additionally, we manually checked each data point captured to verify that we weren't feeding our models bad data. With each callback from the ROS topic `R1/cmd_vel`, we saved the images using OpenCV's `imwrite()` method and labeled them according to the selected character in the generator script.

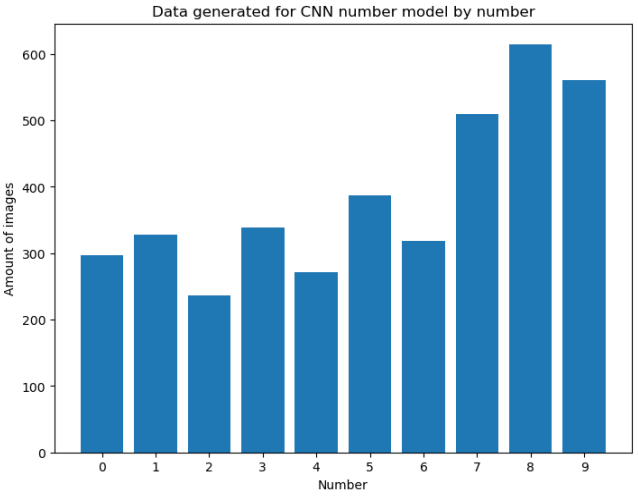
The following three graphs show how much data was fed into each CNN by label.



Total letters fed to alphabet model: 15193



Total numbers fed to ID model: 953



Total numbers fed to number model: 3860

We tried to keep the data roughly equally spread between each label in all our CNN training sets, only adding extra data where it had difficulties.

In the alpha model, we aimed for approximately 450 images of each character, with extras for 'Q', 'O', 'W', and 'V' as these were the letters our model had the most trouble with. In the ID model, there was very little variance in the data we would be reading, so we aimed for 100 images of each number. Lastly in the license plate number model, we aimed for nearly 350 images of each number, and it tended to get confused by '8's and '9's so more images of each were added.
A trend we noticed was that as the output range grew, in general more input training data was required. This seemed to make sense as the model would need more information to distinguish between each output class. Despite the pipeline we established for generating data and processing it all, we found the data generation stage —specifically for the reading models— to be the slowest part of the process.

NN Architecture and Parameters

The architecture of the models we used were the following:

Model: License Plate Alpha Reader		
Layer (type)	Output Shape	Param #
Conv2D	(None, 27, 13, 32)	320
MaxPooling2D	(None, 13, 6, 32)	0
Conv2D	(None, 11, 4, 64)	18496
MaxPooling2D	(None, 5, 2, 64)	0
Flatten	(None, 640)	0
Dense	(None, 512)	328192
Dense	(None, 26)	13338
Total params: 360,346 Trainable params: 360,346 Non-trainable params: 0		

Model:Parking Spot ID Reader		
Layer (type)	Output Shape	Param #
Conv2D	(None, 28, 13, 32)	320
MaxPooling2D	(None, 14, 6, 32)	0
Conv2D	(None, 12, 4, 64)	18496
MaxPooling2D	(None, 6, 2, 64)	0
Flatten	(None, 762)	0
Dense	(None, 512)	393728
Dense	(None, 8)	4104
Total params: 416,648 Trainable params: 416,648 Non-trainable params: 0		

Model: License Plate Number Reader		
Layer (type)	Output Shape	Param #
Conv2D	(None, 27, 13, 32)	320
MaxPooling2D	(None, 13, 6, 32)	0
Conv2D	(None, 11, 4, 64)	18496
MaxPooling2D	(None, 5, 2, 64)	0
Flatten	(None, 640)	0
Dropout	(None, 640)	0
Dense	(None, 512)	328192
Flatten	(None, 512)	0
Dense	(None, 256)	131328
Dense	(None, 10)	2570
Total params: 480,906 Trainable params: 480,906 Non-trainable params: 0		

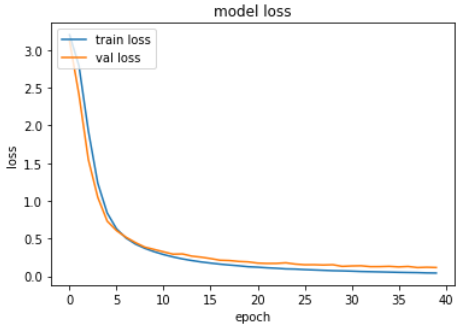
We didn't have a specific architecture in mind when creating our models. We started with a few layers and then added more layers and compared the validation and training loss graphs. We found that a few layers worked well for the alpha and ID readers, but more layers worked better for the number reader.

The only other training parameter we varied was the number of epochs. After training, we looked at the validation graphs and increased the number of epochs until the accuracy began to plateau to prevent overfitting. Lastly, for our later models we kept the same parameters and simply trained with more data but didn't change layers or compare the various graphs because we didn't want to risk maxing out our computing power on Google Colab and losing our environment.

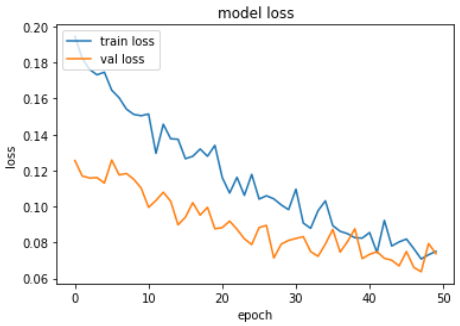
Testing and Analysis

To evaluate each model, we compared the validation and training loss graphs, looked at the confusion matrices, and briefly looked at the top losses. The results for the alpha and ID readers were quite similar, this likely due to them having the same architecture.

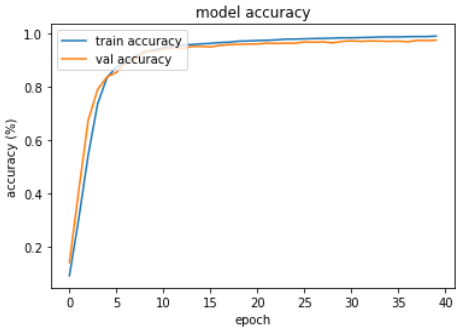
The number of epochs for the license plate reader and parking ID models both seemed to plateau after about 40 epochs while the license plate number model needed 50 epochs.



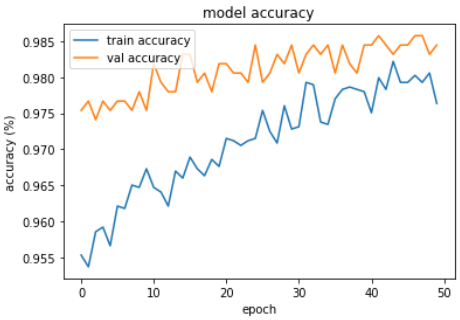
Model loss graph for alpha and ID models



Model loss graph for number model



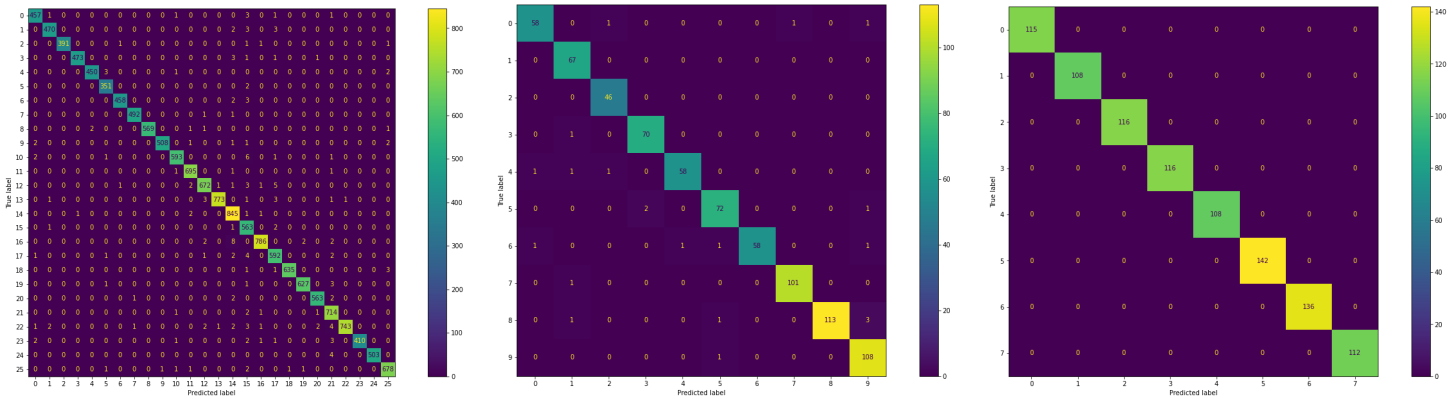
Model accuracy graph for alpha and ID models



Model accuracy graph for number model

The model loss and accuracy for the license alpha and ID reader are as expected. The number reader, while looking choppy, has initial values that are much closer to the convergence values. Therefore only the scale of the graph is zoomed in, the performance is roughly on par with the other two models.

Finally the confusion matrix was useful in evaluating what inputs our models had the most difficulties with. We used it to identify which data we needed more of when coming back to retrain our models.



The confusion matrices from (left to right) our alpha, number and ID models used in competition.

Driver Module

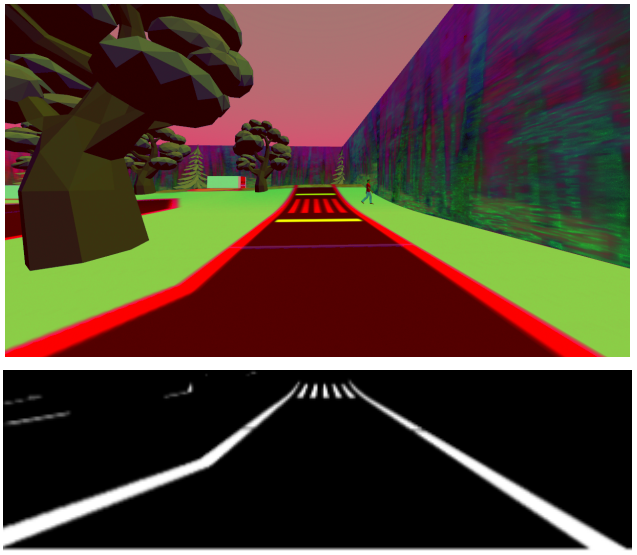
Our driver module, responsible for controlling the robot's movements, was composed of two main components: driving controller and object (pedestrian and vehicle) detection. The driving controller was created with two CNNs (one for the outside track and one for the inside), with the input as images from the robot's front camera and the output being the velocities of the robot. Simultaneously as the robot drove, we performed checks for any pedestrians and parked vehicles, which would adjust the driving of the robot accordingly.

Driving controller

This subsection will cover all details regarding the CNN used for the driving controller. We spent the majority of our time working on the driver for the outside loop and then replicated our process but with different data for the inside loop, so for conciseness and clarity we will look at the process for the outer loop CNN.

Pre-Processing

The main purpose for pre-processing the raw image from the robot's camera was to reduce the dimensionality of input to the CNN, while keeping only the necessary information from an image. Considering this, we first decided to apply an HSV band filter to the white lines on the sides of the road - this also allowed our data to make no distinction between the regular and grass road portions, making the data more uniform. Upon testing, we realized that the image sizes were too large, which slowed the model's training and predictions. Therefore, we cropped the upper half of the images and compressed it by a factor of four. The resulting image still contained all of the necessary data, since the road lines were only present on the bottom half of the robot's field of view.

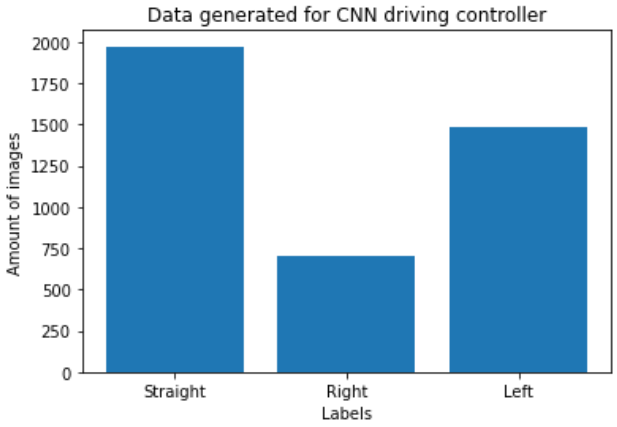


Raw image data in HSV format (top) vs. preprocessed image data (bottom).

Data Generation/Collection

Collecting data was done by driving around the track with the robot, and saving and labeling the images throughout the process. Since we expected to collect a large amount, we tried to streamline our process as possible. Considering this, we created a script that as we drove, would automatically preprocess and label the images accordingly. The labels of each image included the velocities of the robot when the image was received as those should be the predicted output velocities of the CNN. Specifically, we held the following format: <x linear velocity>_<z angular velocity>_<frame number>.png. The frame numbers are unique values so that images with the same velocities were not overwritten. To decrease the output space of our CNN, we discretized our velocity values to only have three possible [x,z] combinations - driving straight [x_set,0], turning right [0,- z_set], and turning left [0,z_set], where x_set = 0.4 and z_set = 0.8 are the setpoint discretized velocities values.

Additionally, our script was made so that we can determine when to start and stop data collection. Using the fact that changing 'y' velocities did not affect the robot's movements but were readable through ROS, increasing the 'y' velocity would start the data collection, and decreasing it would stop. This allowed us to efficiently collect specific types of image data in specific durations while only launching the competition surface once. For instance, we used this method to collect data on the robot's recovery from driving off the road. We would drive the robot off the road, then start recording while driving back onto the road, and repeat the process multiple times without having to run, stop and rerun the script each time.



The overall goal was to collect data such that our robot can be robust in all possible situations. Therefore, we collected large quantities of data for the robot turning left, right and driving straight to maintain performance in all directions. We didn't need as much data of the robot turning right since the outside track only has left turns. Additionally, we collected data with the robot driving within the road, correcting itself when starting to drift off the road, and starting completely off the road. This was done at every region of the track, to maintain its performance in all locations.

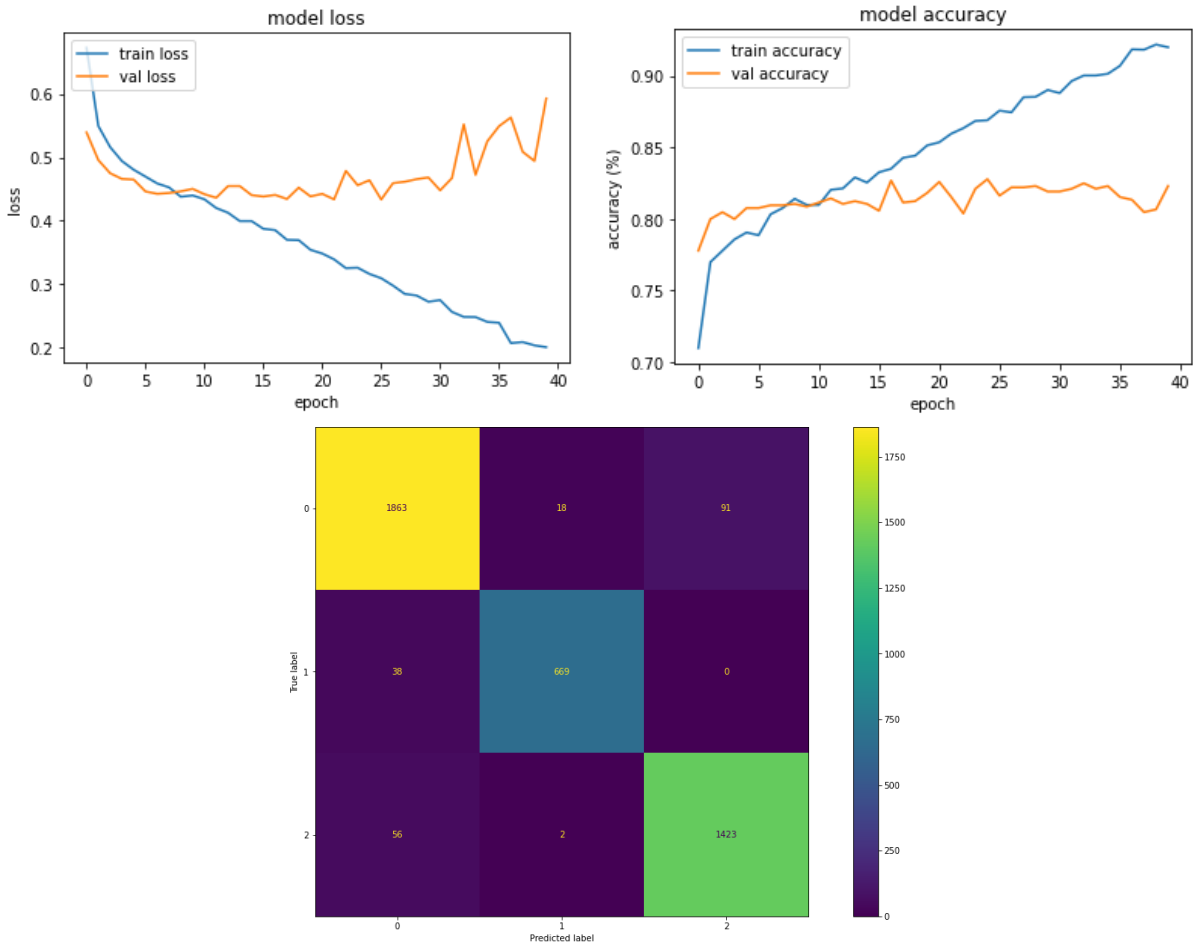
NN Architecture and Parameters

Model:Driver controller		
Layer (type)	Output Shape	Param #
Conv2D	(None, 88, 318, 32)	320
MaxPooling2D	(None, 44, 159, 32)	0
Conv2D	(None, 42, 157, 64)	18496
MaxPooling2D	(None, 21, 78, 64)	0
Conv2D	(None, 19, 76, 128)	73856
MaxPooling2D	(None, 9, 38, 128)	0
Conv2D	(None, 7, 35, 128)	147584
MaxPooling2D	(None, 3, 18, 128)	0
Flatten	(None, 6912)	0
Dense	(None, 512)	3539456
Dense	(None, 5)	1285
Total params: 3,912,325 Trainable params: 3,912,325 Non-trainable params: 0		

Akin to the license plate models, we didn't have a specific architecture in mind, but decided to start with only three layers and planned to make adjustments if deemed necessary. Luckily this model's performance was sufficient and the only deficiencies were solved with a more robust dataset. A summary of the driver model is outlined below.

Testing and Analysis

Likewise to the license plate models, the models were analyzed by plotting the model's losses and accuracies and the confusion matrix, for both the training and validation dataset. In the driver case though, a bad prediction is acceptable because the model would need to make several poor predictions in a row to drive off the road so we also considered its performance on the competition surface in our testing. As seen from the images below, the model's validation accuracy seems to plateau at 80%, while the training accuracy increases. The model seems clearly overfit but when testing it performed adequately so we used it in our driving controller.



Summary of model characteristics. Plot of the losses and accuracy of training and validation data (top) and the confusion matrix of the entire dataset (bottom)

Pedestrian and Vehicle Detection

To detect and safely drive past the crosswalk, we implemented a set of sequences. First, while driving, we checked if the robot was close to the red line. This was implemented by filtering only for the color red, and we deemed it close if the largest contour was above a threshold area, since the area is larger when closer. Once at the crosswalk, the robot stops and only drives across when the pedestrian has moved off the road. This is done by comparing two subsequent images: If the mean squared error between the two images are similar, then the pedestrian is at the same spot and is not crossing. Otherwise, the pedestrian is considered moving across.

A frequently encountered problem was the projected license plate images would become blurry when seen at a far distance from parked vehicles, increasing the risk of a misprediction. To minimize this, we only predicted the license plates when the robot was considered close enough to the parked vehicles. Since all the parked vehicles are blue, we checked the image's blue area to determine the distance from the license plates. To further minimize the risk, the robot also slows down in this region so it can gather more data.

Summary

To summarize, our main strategy was driving and reading using imitation learning, completing the outer loop, then proceeding to the inner loop. We read each license plate we passed multiple times, saving both the predicted license plate and the prediction values. Once we stopped at a crosswalk we

computed the most commonly predicted license plate for every given parking ID and published the result. On the inner ring, after capturing five frames of each parking ID, we computed, published and stopped the timer.

Our biggest challenge was also our biggest accomplishment: the pipeline for gathering adequate amounts of data. It was the biggest challenge by far because we needed *lots* of data to properly train our models for robust performance. However, it was also our biggest accomplishment because we attribute our success to the well-trained and robust models. Gathering the data took the longest of any task in the project despite the amount of time we spent optimizing it. It was also very tiring because we manually checked through each data point to remove bad data before training. Once we had all our data though, this allowed us to test different model structures, test training on different datasets and ultimately create the effective models we used in competition.

Dead Ends

One of the methods we attempted but didn't include in our final build was a driving machine learning model that instead of outputting one of three states, would output an angle to determine how much we should turn at each instance. Instead of simply going straight, turning left or turning right, the model would continuously move forward and turn increasing amounts as required. We planned on gathering data by driving with a joystick instead of a keyboard since this would allow for smooth steering. The desired advantages of this was that it would drive faster, more efficiently, and be a more competitive robot. However, gathering data for this method was not worth the trouble since our alternative of driving using a keyboard did fine, and we had difficulties adapting a joystick controller to give the proper inputs to ROS and reading those values.

Further Development

Some alternative approaches we would have tested with more time were a SIFT detection model for the parking space IDs since they're distinct objects, training our inner circle driver by filtering for the gray road instead of the white lines, and attempting a single reader CNN for license plates and parking IDs.

The parking IDs would have been a perfect opportunity to learn more about SIFT because they are distinct objects that need to be recognized. However, we didn't spend much time working with SIFT because object detection seems weaker when dealing with blur and we already had the neural net infrastructure set up for the other models.

Training with images that highlighted the road instead of the lines would have also been a good idea to test since the inner ring didn't have "grass-road" terrain. Additionally, at a first glance it seems quite effective since the white lines were ambiguous due to there being multiple lanes and intersections that were all within view. Unfortunately, we only considered this option much later and had already built the infrastructure for gathering data that highlighted the white road lines.

Lastly, a larger single reader CNN would have been very interesting (and fun!) to train. We thought that training two separate models would reduce the output space and thus the error, but hadn't considered only looking at the outputs we were expecting for a given input (i.e. when looking at the second half of a license plate, only evaluate data that points to numbers). Building one that also read the parking IDs seemed out of reach though since it would need to interpret two unique fonts and we were already maxing out the computing power available on Google Colab.