Neural Network based ROS Robot for License Plate Detection

ENPH 353, 2022

Ethan Predinchuk, Mack Wilson

Introduction

The ENPH 353 machine learning competition is a competition designed to test the class' understanding and abilities related to machine learning. Over the course of several weeks, we designed and implemented a ROS based robot which utilized both classical techniques and machine learning to complete a series of objectives in a Gazebo simulation environment. The requirements for our competition driver were to navigate between the inner and outer loops, avoid hitting all pedestrians and vehicles on the course, maintain at least two wheel on the road at all times, and analyze the license plate numbers of several cars which were parked around the course. The images being collected as input to the robot controller will come from a camera attached to the front of the robot. Our controller will only be allowed to gather information from this camera as well as the built in simulation clock. Our team's goal going into competition was to complete as much of the course using primarily machine learning based techniques as possible.

Controller Framework

Our robot controller was written in a variety of “nodes” that used the ROS node and topic framework to parallelize some of the different major processes and increase the speed of our program. The three nodes used were related to driving, pedestrian/truck detection, and license plate reading. Each of these nodes was subscribed to the robot’s camera feed, and analyzed the image for specific features which were used to guide the robot in achieving the objectives set out in the rules of the competition. The nodes communicated with one-another through a communications topic - called “/comms”.

The driver node used an imitation learning trained convolutional neural network (CNN) which was given images from the point of view of the robot and returned one of three commands “forward”, “left”, or “right”. The pedestrian handler monitored the robot’s point of view and guided the driver in avoiding collision with the pedestrian and truck. The plate handler constantly analyzed the images for license plates and, when detected, passed them through two separate convolutional neural networks which predicted the license plate number.

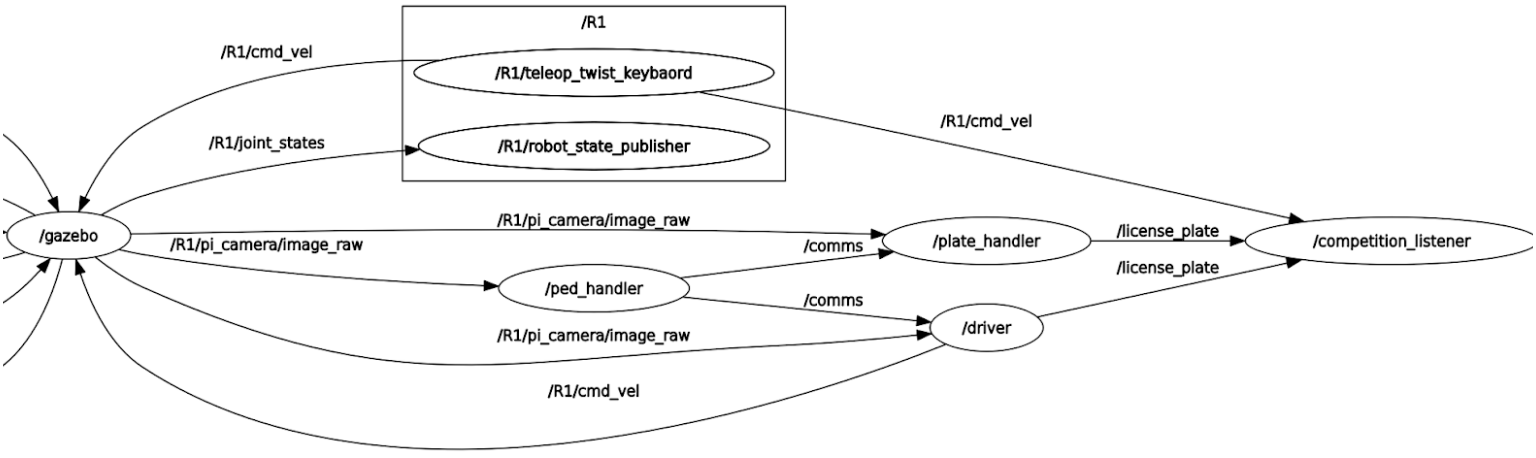

Figure 1: Graph of ROS nodes and topics used by our robot controller.

License Plate Recognition

Plate Detection and Isolation

For license plate reading, we originally attempted to find the outline of the whole white area taken up by the license plate. This meant using an HSV filter that tuned to keep the white of the plate while removing the white of the road. Testing showed this approach would not be viable, it didn't seem possible to remove the road lines without removing parts of the plate itself. Since this was unsuccessful, the next approach was to filter the blue of the car, as it was unique and could therefore provide a useful contour for determining where the plate was supposed to be. However we ran into issues with contour sizing, as the second largest contour on screen was not consistently the blue part of the car on the far side of the license plate. This would have led to a large and difficult algorithm where we would need to look at the largest contour, and then find the next largest contour that also fell within a certain number of pixels of the original. Whether or not this actually would have been complicated, it was decided to abandon this idea in favor of a third attempt. We determined if we filter to the white of the license plates, we can isolate the section above the gray plate number, and seeking for this part of the license plate makes the road lines very thin, as can be seen in Figure 2.

This image was also sliced around the middle of the screen, since we noticed the plate very rarely wasn't within this slice. Cutting the image had two purposes; first, a smaller image was less to process and sped up compute time. Second, the slice further limited the size of the road line contours. Since we were using the largest on screen contour to decide where the top of the plate was, we had to take every measure possible to ensure the largest contour was always the plate. The function "seek_license" in enph353_controller/node/plate_handler.py [1] takes care of this functionality. It also takes the largest contour and approximates it to the minimum number of points it can, which is the corners of the shape. If there are exactly four corners, then we expect that contour to be a plate. If all the prerequisite criteria are met, the approximation of the plate is applied to the raw camera feed and passed along the pipeline.

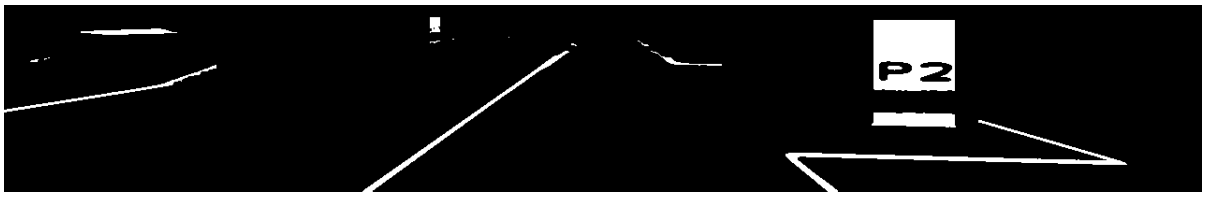

Figure 2: Threshold feed of the robot license plate reader

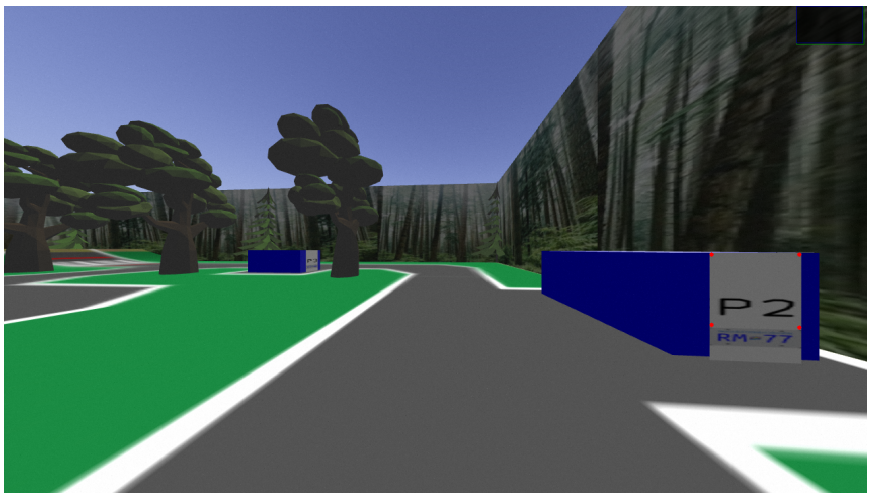

Figure 3: Raw camera feed displaying where the license plate is on screen

Once the plate had been located, the raw image was then processed more. We stored a variable called “biggest_plate_size” that continuously updated if the plate in the current frame was bigger than any previous frame. Whenever this variable was updated we also updated “biggest_plate” which was holding the coordinates of the four corners of the plate and “best_plate_image” which held the image we were looking at when we saw the biggest plate. Once there was a break in the frames where we don’t see a big enough contour, the biggest plate image would be sent to the rest of the pipeline to have the CNN make a prediction on it. The point of this process was to limit the number of poor quality plate images being sent to the CNN as performance was a priority. In retrospect, it probably would have been ideal to store more images because this method only ended up sending around 3-4 images to the CNN, which is very low and resulted in lower guess confidence due to outliers having a large influence.

Once there was a break in the frames of seeing a plate, either due to a camera angle making the plate appear too small, or the plate being off screen, the plate was submitted to the CNN. However, before the plate could actually be submitted, it was first heavily processed. The points that defined corners were sorted in the order of [top left, top right, bottom left, bottom right]. It was important to know this order so the plate could be easily perspective transformed, otherwise the plate may end up upside down or sideways. We perspective transformed the image using the built-in function in openCV, specifically “getPerspectiveTransform” and “warpPerspective”. Our implementation of this functionality was based on code written by Akshay Chavan, see Straightening an image reference 4 for more. The combination of these two functions would transform the points we had seen, plus 150 extra pixels in the y at the bottom. This number was found through experimentation, and made it possible to capture the identifier number and plate numbers at the same time. The results of this transform can be seen in Figure 4.

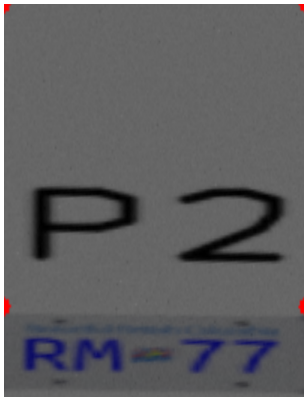

Figure 4: Perspective transform of a plate seen in the simulation

Once the plates were isolated and transformed, giving us a clean upright image, we then began preparing the images for prediction by the CNN. We separated the lower portion of the image, which contained the letters into a separate image and processed the two slightly differently. First, we applied a gaussian blur with a 3x3 kernel to both images and then an HSV filter was applied. For the parking stall number we used a filter which allowed us to mask the black characters. For the plate number itself, we used a filter which specifically isolated that shade of blue. Figure 5 shows the results of this stage of the process. The purpose of this masking was two-fold, first we were able to produce a simpler image for prediction, but also, the masking process allowed us to leverage OpenCV’s contour detection functions to isolate the individual letters. By finding the two largest contours, and taking the rightmost one, we were able to determine the position of the stall number, similarly for the plate number we found the four largest contours. Upon finding our contours we used the minAreaRect() function which gave us the coordinates of the rectangle which enclosed each contour. We then used array slicing to produce five separate images, the first corresponding to the stall number, and the rest the plate number. A single pass of the OpenCV erode function, followed by a single pass of the OpenCV dilate function was then applied, this reduced some issues we were having with black, pixelated noise showing up on our characters. These masked, filtered, and isolated characters were then resized and the CNN was used to make predictions.

Figure 5: The plate after being filtered for the identifier number (left) and the plate numbers (right).

Convolutional Neural Network for Plate Labeling:

We developed our plate isolation framework prior to training our convolutional neural network, this allowed us to use these functions to generate a dataset which was more representative of the plates we will be seeing in the competition. To do this we collected the raw plate images, like the one seen in Figure 4, without any further post-processing, and saved these as files for training. With this we were able to generate approximately 350 images of plates, giving our dataset ~350 stall numbers and over 1200 individual characters for the license plate. A script was used to quickly label these. Given more time - we would have favored a combination of both these plates, as well as a set of generated and enriched plates, which would have allowed for a significantly larger sample size.

We trained two separate networks, the first was used to predict on the stall number alone, and the second was used to predict on all four characters of the plate number. We chose to separate these datasets as we found that training a single network to predict on all 5 characters often led to reasonable accuracy for the plate numbers but lower accuracy on the stall number. This likely arose due to differences in the distortions seen on the stall number, or differing distortions that arise due to the resizing of the larger character to the smaller ones. Considering the fact that an incorrect prediction on the stall number would make all four predictions for the plate numbers incorrect as well, we decided that it was worthwhile to offload this task to a separate network.

Prior to training the network on the plate data, we applied the previously described preprocessing techniques to mask the image and isolate the individual plate characters. These images, along with the manually produced labels, were used to train the network.

Our network architecture was built following the provided framework for a character prediction architecture from lab 5, initially we followed this structure exactly, and found reasonable prediction accuracy. See Figure 16 in appendix for an overview of this architecture. Upon importing the models into our simulation driver, we found that our biggest problems had to do with performance, as the networks ran dramatically slower on the laptop we were going to use for the competition relative to the desktop that was used for training. Considering this issue, we decided thinning our networks was necessary. This was done experimentally, by reducing the sizes of layers, and then testing to see if any effects were seen, eventually reducing the network size, and associated computation time dramatically.

The training parameters were set in order to maximize the validation accuracy while reducing overfitting, when training to predict on the characters we found that often the validation accuracy would reach its maximum after 1-2 epochs of training, and then would oscillate around this maximum, indicating our step sizes were too large in our gradient descent, causing us to step oscillate around our local minimum, to counteract this, we found that decreasing the learning rate to 5×10^{-5} allowed us to train for 20 epochs to approach this minimum without oscillating, as seen in Figure 6.

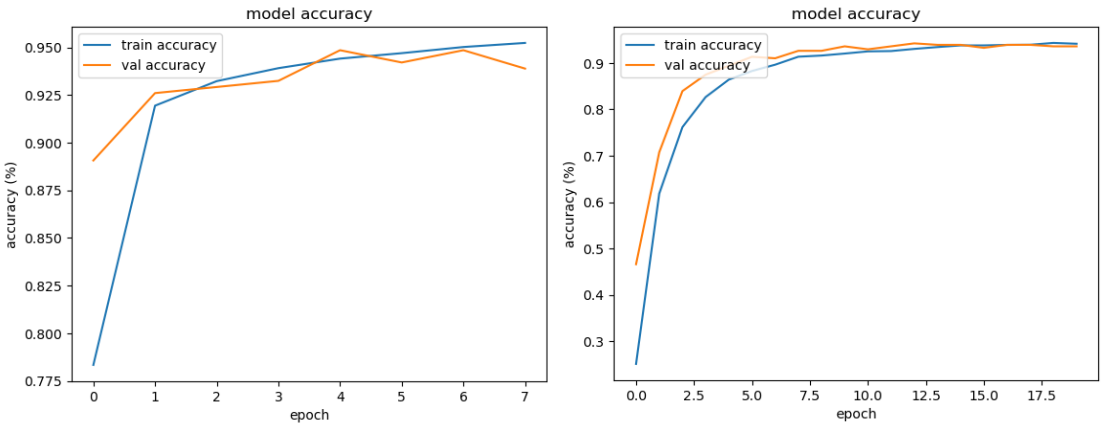

Figure 6: Character prediction model accuracy vs. epoch for different learning rates. $1e^{-4}$ learning rate (left) and $5e^{-5}$ (right).

We used categorical cross entropy for our loss function, which is suitable for the task of categorizing inputs into a set of classes; and ReLu as our activation function, as compute speed was a priority, with softmax activation at the output. As our dataset was generated, we did not have an ideal data spread, which was one of our biggest problems in the competition. We had several characters which were under-represented and therefore would be very unlikely to be guessed accurately. Given more time, enriching our data with generated plates would have been a reasonable way to remedy this. Figure 7 shows a confusion matrix of our network's predictions on our training set, our data distribution used to train the letter network at competition time, 10% of the training set was reserved as a validation set, and so the network's weights were never modified by these images.

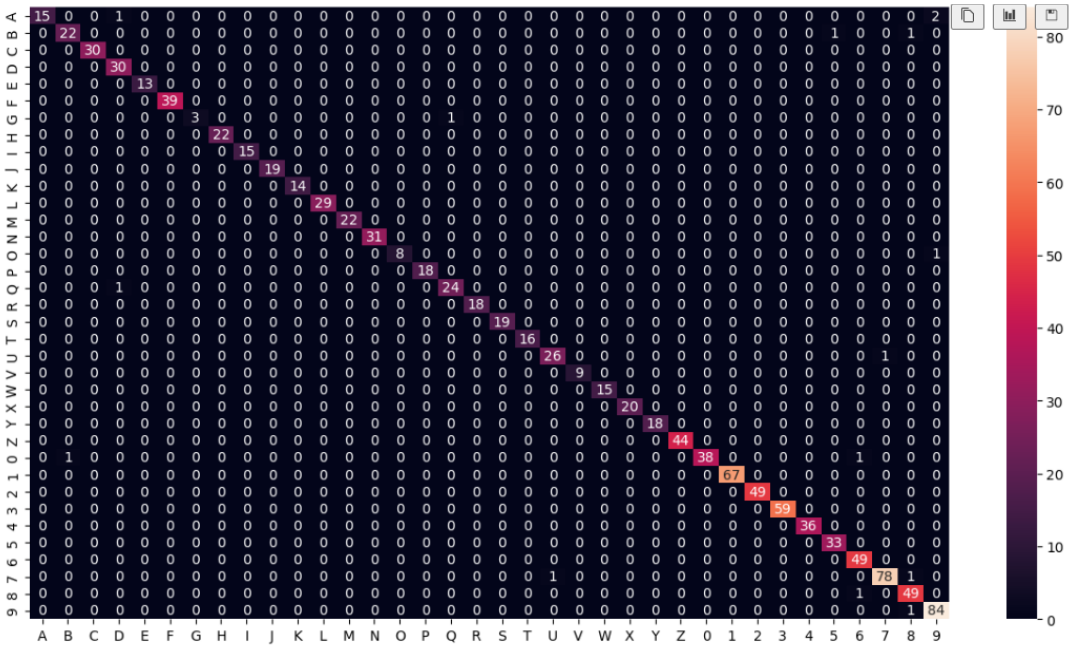

Figure 7: Confusion matrix of plate number network predictions (Real values on Y-axis, predicted on X)

The sparsity of our dataset is clear from this confusion matrix, however in practice, it had very reasonable prediction accuracy on most characters when run in the simulation. Although we were unable to increase this dataset size prior to the competition, we labeled more data afterward and were able to significantly increase the model's reliability.

Prediction Averaging Strategy

As a reminder, we were sending the plate and identifier numbers to their respective CNNs every time there was a break in consecutive frames being seen. However, we did not immediately submit these guesses for scoring. In testing we found our first and last frame were often the most uncertain of all the pictures we took; usually there were 2-4 frame submissions between the first and last that were much clearer. Each character guess was stored as a one-hot array, and all the guesses for a given character were summed together. Then, the index pertaining to the max value in each array was used to convert the arrays back into characters. This ensured that we were using the most frequent guess for each character. These guesses would be summed until no plates were seen for two consecutive seconds. In hindsight this design could have been improved by simply making an array for each stall number, and averaging the guesses in this array, this way we could be sure that every guess for a plate is properly registered, to that plate, and we do not need to wait until we see the next plate to submit.

Robot Controller

Control Strategy

The control of our robot was handled by two nodes, one which handled navigation of the course, henceforth referred to as the driver, and another which handled detection and avoidance of the pedestrian and truck, which will be referred to as the pedestrian handler. The nodes communicated through a ROS topic, where the pedestrian handler would send interrupt commands to indicate to the driver when it should stop, and when it is able to drive.

Our driver node operated as a state machine, with the following states:

1. Outer: In this state the driver is actively driving in the outer loop, repeatedly passing camera inputs to the outer loop driving network to navigate.
2. Pedestrian Stop: This state is entered when the pedestrian handler sends an interrupt to the driver, the driver will stop immediately and wait for further commands from the pedestrian handler. This state is entered at both crosswalks and before entering the inner loop.
3. Pedestrian Drive: In this state, the robot will undergo a predetermined maneuver in order to travel through the crosswalk without colliding with the pedestrian.
4. Inner: This state is entered after passing the second pedestrian. In this state the driver is actively driving a path from the second crosswalk into and around the inner loop.
5. Terminate: The controller is finished, the driver stops and ceases to command the robot.

Driving Convolutional Neural Network

Our driver node was controlled by a convolutional neural network which was trained through imitation learning. We generated two separate driving networks, the first of which was used to drive the outer loop, where there is a bias towards left turns, and the second was used to drive the inner loop, where a sharp left turn is required to enter the loop, and then there is a bias towards right hand turns once inside. These networks were trained on images taken while we were manually driving the robot. We generated the images without the pedestrian or the truck in the simulation, and it generalized well once they were reintroduced, with no noticeable changes in performance.

In order to generate our images to train the network, we created a script which subscribed to the robot's cmd_vel topic, and to the robot's camera, this allowed us to retrieve images from the robot's perspective alongside the current velocity information. See enph353_controller/scripts/capture_cnn_images.py [1] for this script. Saving every image from the camera resulted in a massive bias towards going straight, as the frames in which the car is turning represent a very small subset of the total images. To make the dataset more appropriate we modified the script to take every n-th image or whenever a move command was sent - where n was modified depending on if we wanted to add turning images, or forward images. The image associated with each frame produced by this script was labeled with the exact x-linear velocity, and z-angular velocity so that we could modify the way we interpreted the velocity information down the line. No frames were recorded in which the robot was stopped. An example of an image generated from this script can be seen in figure 8.

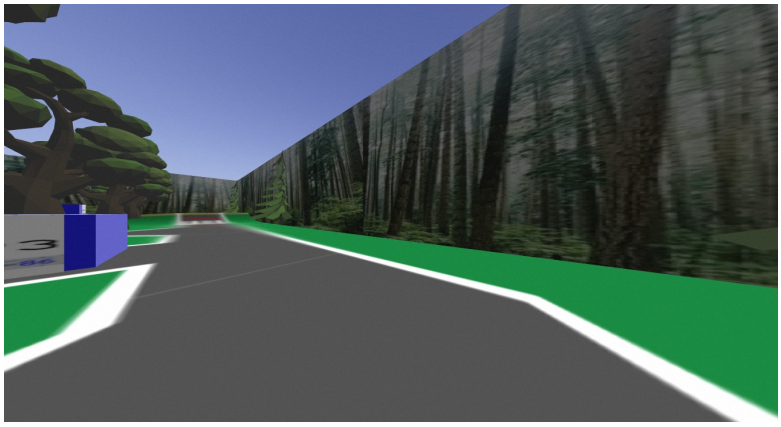

Figure 8: Example of unprocessed image used for training the driver, showing a left turn

The generated images were all saved at full resolution, with the full velocity information, this allowed us to do any data processing that was necessary later down the line, in case we decided to modify our processing pipeline. We experimented with several processing methods, finding that the robot had similar driving performance in color vs. grayscale, or even after downsampling the image. Our final network was trained to work with a 4x downsampled image, in grayscale which was unsliced (the full image contents was kept), this allowed for significantly shorter computation time without dramatic losses in prediction accuracy.

In generating images for the outer loop, our strategy was to develop a diverse image set, by driving the loop in different ways, sometimes driving the loop as perfectly as possible, staying in the center of the road and avoiding turning except for at the corner, other times overturning, and oscillating from side to side. This made our driver very robust, before we added the pedestrian handling mechanisms it would sometimes even correct itself back onto the road after being knocked off by the pedestrian.

Our network architecture was originally based on Bojarski *et al.*'s work [3], which we modified after we successfully got it driving to reduce the overall network size, see Figure 17 in appendix for an overview of the network architecture. This used mean square error for its loss function, and ReLu as the activation function, with softmax activation at the output. The learning rate we used was 1×10^{-4}, interestingly reducing this learning rate further (such as to 5×10^{-5}) lead to the model being biased against right turns, see Figure 13 in appendix. Again, thinking about the process of gradient descent, this could mean that the network was falling into a local minimum, and when the learning rate was low, it was unable to climb back out of this minimum, increasing the learning rate and training on a larger number of epochs (60) resolved this problem.

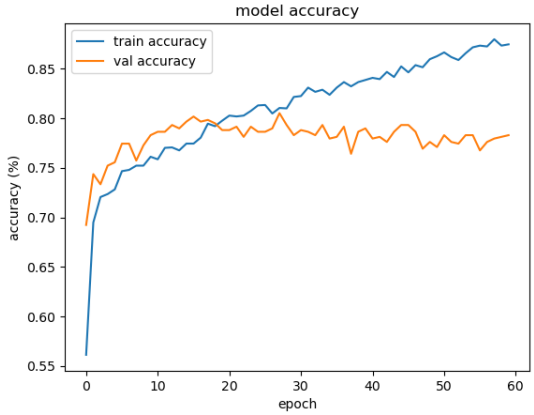

Figure 9: Model accuracy vs epoch for the imitation learning based driver controller.

Although Figure 9 seems to imply that we may have wanted to stop training of our network at epoch 20, we found that stopping the training this early again led to the driver being overly biased away

from making right turns. A confusion matrix showing the outer driver model's prediction accuracy can be seen in Figure 14 in appendix. In general, we found it difficult to eliminate the bias towards going straight, as if we enriched our dataset too much with turning images, the driver would sometimes get stuck oscillating between left and right turns, in general we found that some bias towards going straight helped reduce this problem. In this confusion matrix we can see that for the outer loop we had a dataset that had mostly left turns and straights, with some right turns added to allow the driver to error correct.

The inner loop driver was structured identically to the inner loop one, however it was trained on a different dataset. We took a different ideology into training this driver, as we wanted to train it to automatically take the first left turn after the second pedestrian crosswalk and then stay in the inner loop indefinitely. Rather than trying to build a set of images that would make a very robust driver which had great error correction ability, instead we built a set of images where we repeatedly followed a specific path. Although this in theory would make our driver less generalizable, we believed that this strategy would ensure that it always followed our desired path exactly. In general this strategy was effective, as we would consistently take the left turn into the inner loop the same way, which was important for how we handled the truck. The network architecture and training parameters for this network were the same as for the outer loop network, Figure 17 (appendix) is a confusion matrix with its training predictions, here we can see that there was less of a tendency towards straight turns, as the inner loop involves significantly more maneuvering relative to the outer.

Pedestrian and Truck Detection System

Our pedestrian handler node ran synchronously with the driver, and analyzed the images that our robot was seeing to determine when to stop and when was safe to drive in order to avoid colliding with the pedestrian and the truck.

Similar to the driver, this operated as a state machine, where in each state it would look for a specific feature in the image which are indicative of the areas of interest. To handle the pedestrian, the driver would take a significantly downsampled (reduced in size by 10x) version of the robot's image, convert the image into HSV colorspace, and apply a filter which isolates the red line. By monitoring the size of the largest red contour the handler is able to determine when the robot is approaching the crosswalk, once the red line was detected the handler would wait until it was no longer seen in two consecutive images, indicating entry into the crosswalk, at which point a stop command was issued to the driver.

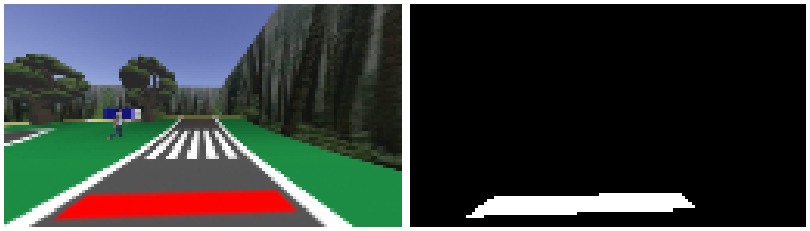

Figure 10: Downsampled and masked images showing the mask the pedestrian handler used to detect the crosswalks.

In order to determine when it was safe to pass through the crosswalk, the handler used a background subtraction mask. This used the OpenCV function createBackgroundSubtractorMOG2, with the detect shadows parameter set to False. We specifically compared the KNN (k-nearest neighbors) and the MOG2 (mixture of Gaussians) background subtractors and found the MOG2 method to be more effective, as the KNN mask tended to be noisier. In order to build this mask, the driver would update the mask for 30 consecutive frames, after which it would begin monitoring the largest contour in the image. The optimal strategy for avoiding a hazard which oscillates from side to side at a random interval is to wait until it has just passed through the center before traversing the hazardous region. The pedestrian handler implements this by repeatedly monitoring any contours that were larger than a specific threshold (to reduce effects of noise) and once this contour passes through the center of the field of view, it signals for the driver to pass through the intersection. Once safely through the crosswalk, the pedestrian handler returns to scanning for the next intersection, which is handled with the same strategy.

Figure 11: Background subtraction mask showing how the pedestrian handler detected the motion of the pedestrian at the crosswalk.

After detecting, and successfully traversing the second intersection, the final task for the pedestrian handler is to safely guide the entry into the inner loop. A similar strategy was used for this task, first the pedestrian handler monitors the right half of the field of view of the robot for a sufficiently large contour with a HSV filter to the blue color of the parked cars. The right half of the field of view is used to avoid falsely detecting the car which is on the left immediately after the crosswalk. Once the contour is sufficiently large, the pedestrian handler sends an interrupt to the driver to stop. It then builds another background subtraction mask and waits for the truck to pass in front of it. The robot reliably traversed the inner loop slower than the truck and so it was safe for the robot to begin moving once the truck passed in front of it. Once the truck was detected in the center of the screen, the pedestrian handler would signal for the driver to continue traversing the inner loop, and its tasks would be complete.

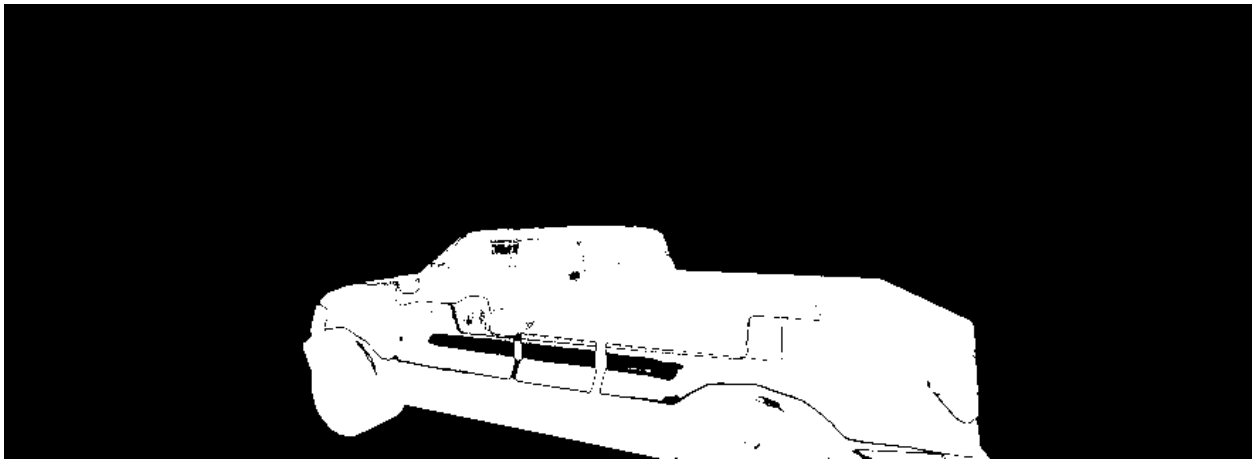

Figure 12: Background subtraction mask showing how the pedestrian handler detected the motion of the truck before entering the inner loop.

Conclusion:

Our final robot controller had three concurrently running nodes, the driver, the plate handler, and the pedestrian handler, each of which was tasked with handling a specific part of the competition. In general each of these nodes was effective at completing its task, the driver could reliably traverse the whole course with the aid of the pedestrian handler to avoid collisions with the pedestrian and the truck, and the plate handler could make predictions about license plate numbers reasonably consistently. We were able to detect most plates with reasonable accuracy, and reliably navigate the whole competition course.

Overall our robot performed reasonably well in the competition. We were able to accurately report all the outer plates and our robot was able to see the 7th plate. However, due to the algorithm we had implemented for managing the plate submissions, we never submitted our predictions for this plate, as we did not detect the next plate. We also had an issue with our termination sequence, which we were unable to resolve prior to the competition beginning. This meant our robot never entered it's termination state, and therefore never stopped the clock before colliding with the truck. Many of our difficulties arose due to issues in transferring our controller script from a home PC with a GPU onto a laptop with integrated graphics. We needed to dramatically alter the driving speed of our robot, among other things in order to ensure it would properly drive, as the increased amount of time required to process an image through our network lead to very irregular driving relative to what the scripts were optimized for. Given sufficient time we believe that we could have optimized the networks, and the controller scripts, however we were unable to make these optimizations prior to the competition date.

At the beginning of the competition we attempted to pursue reinforcement learning. We thought the idea was interesting and had decided on it without much thought into the challenges we might face. It wasn't until after we had lost at least a few days setting up a training environment and sorting out the repos that we realized how difficult the task would be. When it came time to establish the actions the robot could take for the reinforcement learning algorithm, the idea was abandoned due to how limited the time we had to develop the controller was. Other than that, most of the ideas we pursued more or less made it to the final design. Originally we tried to HSV filter the blue of the cars to figure out where the license plates were, but decided the white was easier to deal with.

If we were to do this again, the biggest change would be making things work on the machine we are going to use for competition. We committed a significant amount of time towards optimizing the controller to work on a home desktop but the functionality was , or worked differently on the laptop being used for competition. Had we optimized for the laptop from the start, we could have spent the time we used on that testing and training instead. We also would have interwoven the development of the different systems more closely so each member had a better idea of what was going on in the code. There were times when one of the team members would be waiting on the other to help because they didn't understand how a particular system or algorithm was supposed to work. This led to some times where instead of only one person being unable to work, we both were. This also could have been helped by better commenting and more explicit communication between members after someone had finished something and shared it. However we are happy with the design we ended up with and think the experiences we gained will be extremely beneficial going forward in this field, or anything related to engineering.

Appendix

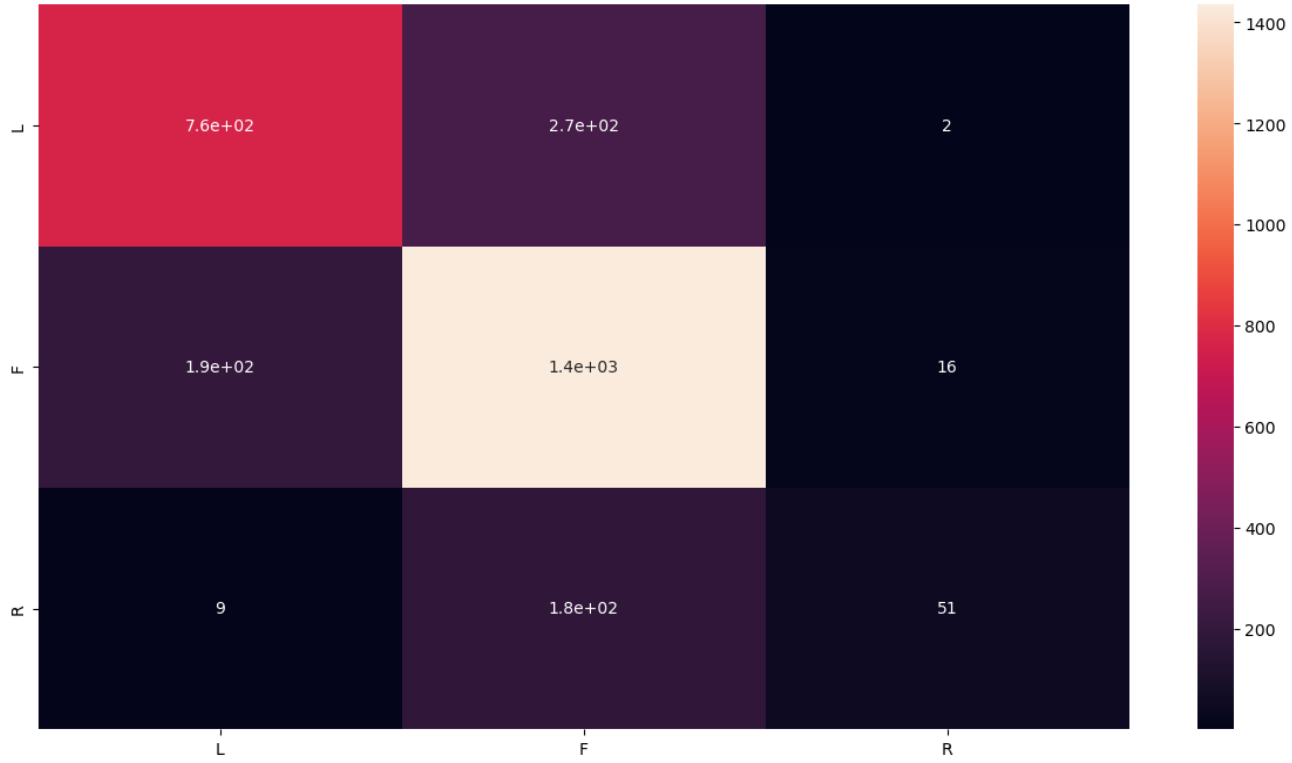

Figure 13: Confusion matrix of driving network when trained with a small learning rate ($1e^{-5}$).

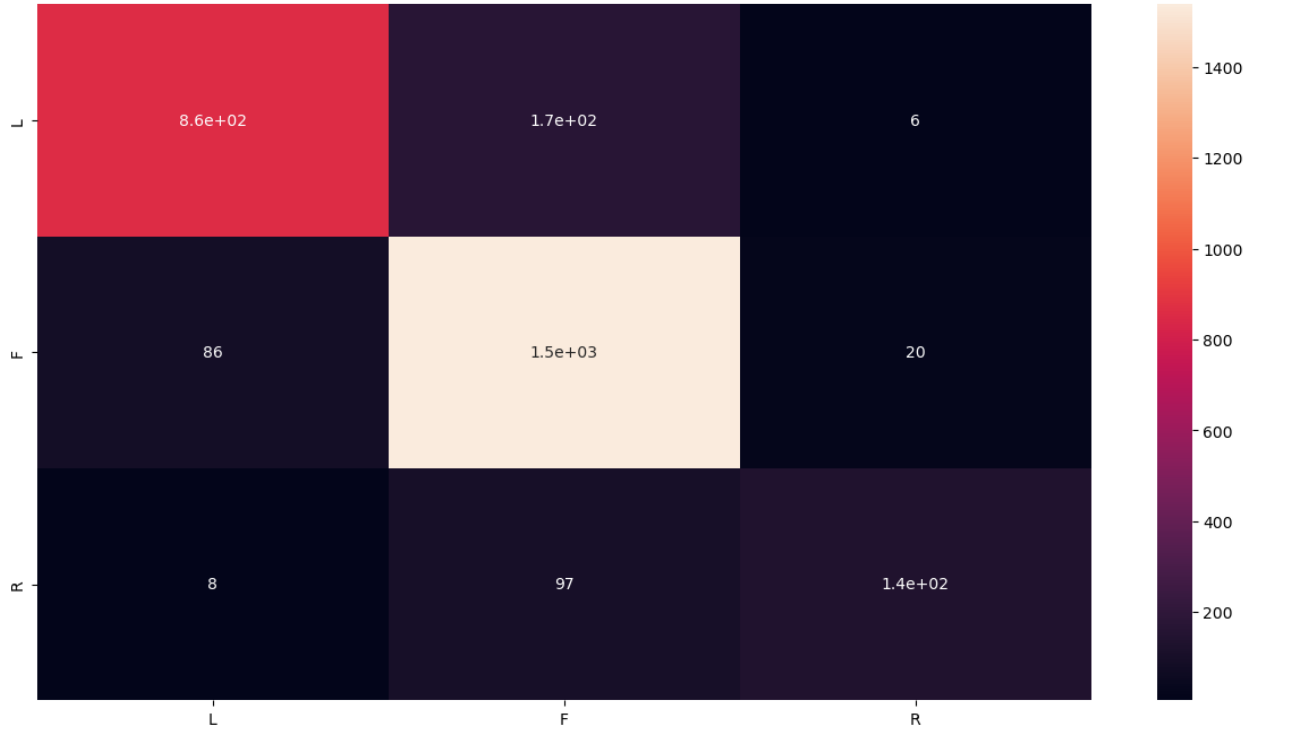

Figure 14: Confusion matrix showing outer loop driver prediction accuracy and dataset distribution.

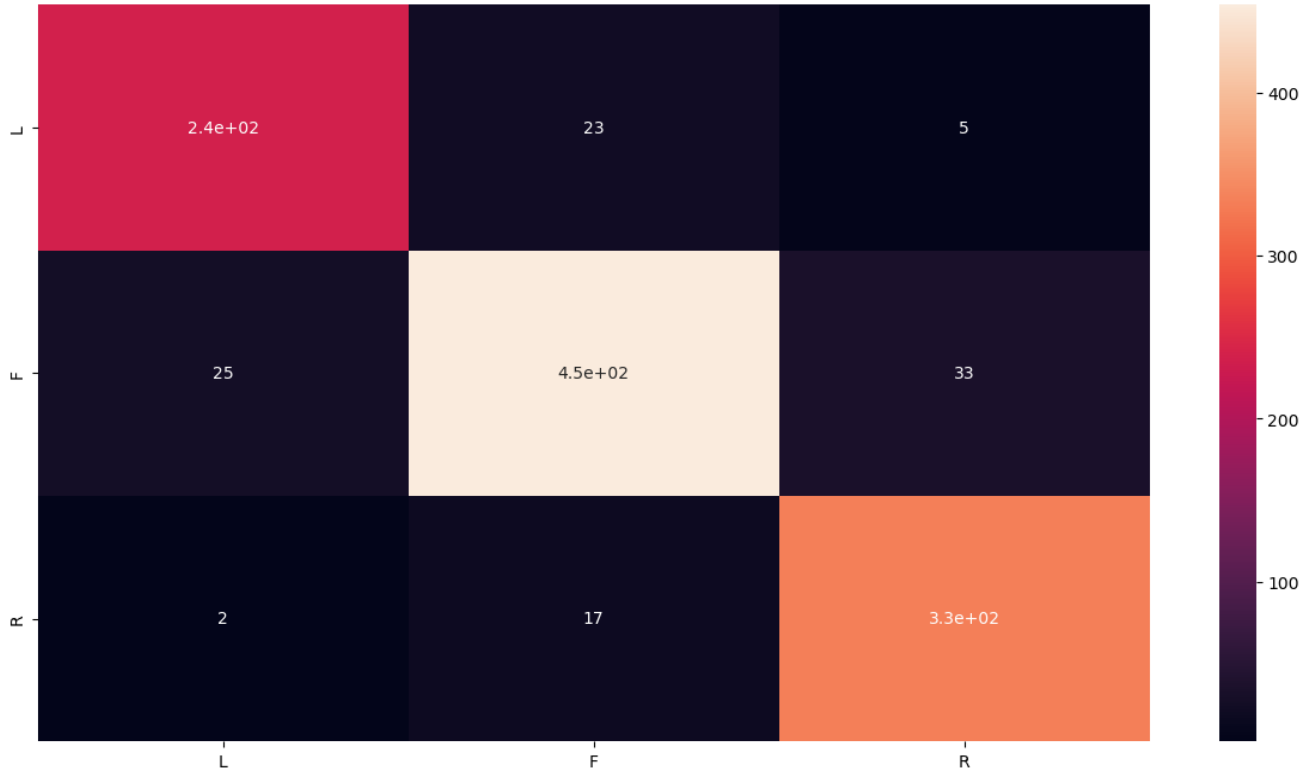

Figure 15: Confusion matrix showing inner loop driver prediction accuracy and dataset distribution.

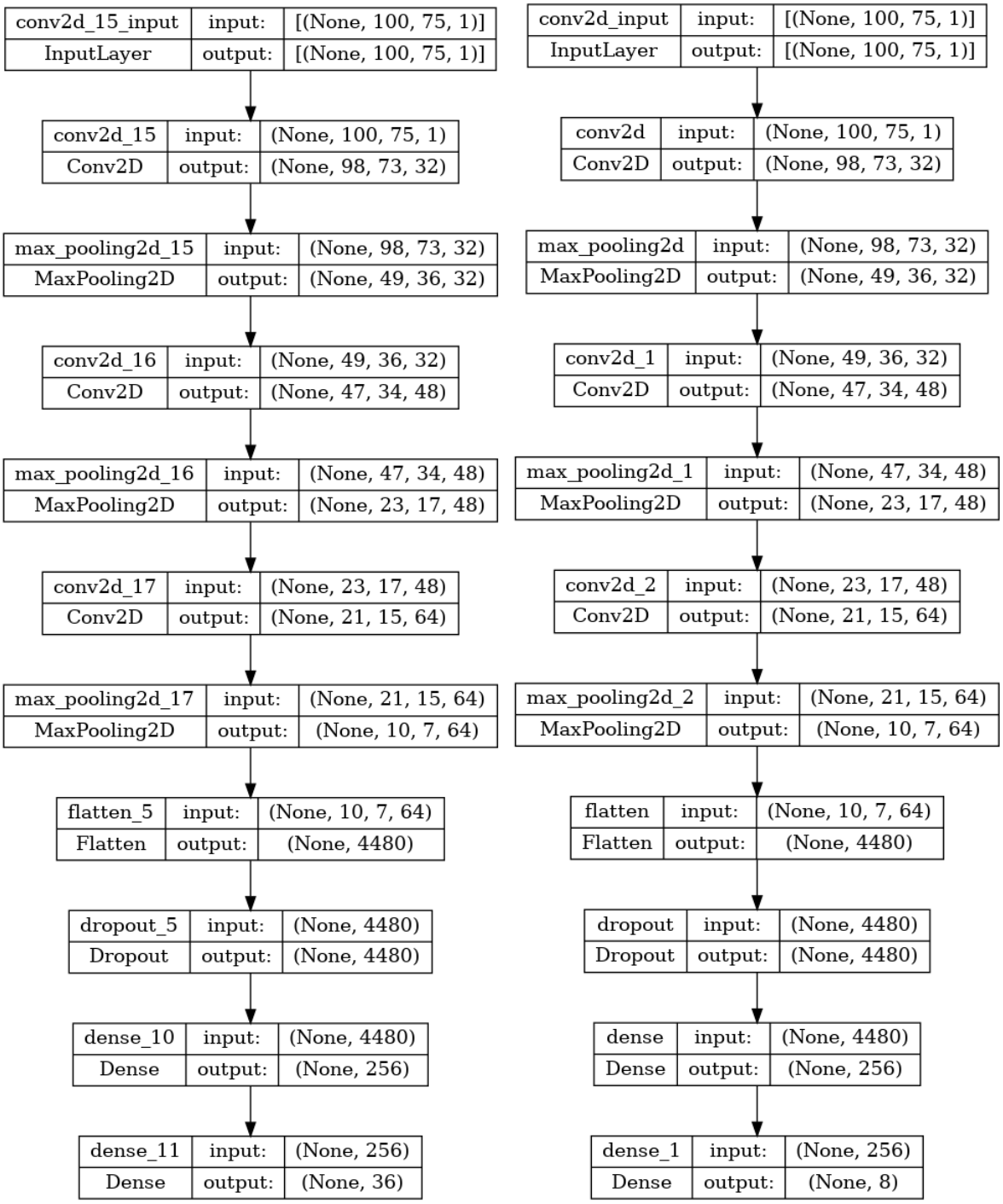

Figure 16: License plate number (left) and stall number (right) convolutional neural network architectures.

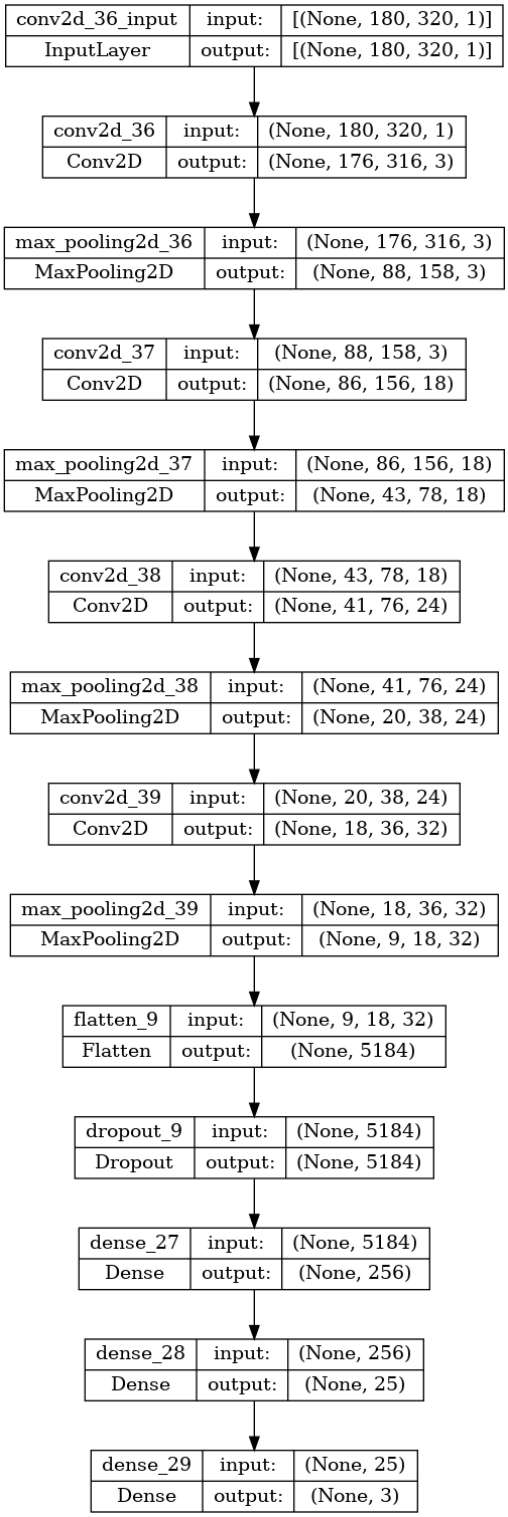

Figure 17: Imitation learning trained driver convolutional neural network architecture.

References/GitHub Repositories

1. Robot controller for competition, and associated accessory scripts: https://github.com/Ethan4416/enph353_controller
2. Convolutional Neural Network training notebooks, saved networks: https://github.com/mackWilson2795/353_cnn_trainer
3. Bojarski, M., Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., & Zieba, K. (2016). *End to End Learning for Self-Driving Cars*. https://arxiv.org/pdf/1604.07316.pdf
4. Akshay Chavan “Straighten an image”: https://arccoder.medium.com/straighten-an-image-of-a-page-using-opencv-313182404b06