ENPH353

ENGINEERING PHYSICS PROJECT I: MCQUEEN AUTONOMOUS VEHICLE

Name Jed Yeo
Callum Hepworth

Contents

1. Introduction

The aim of this report is to summarize the development of our autonomous license plate reading robot for Engineering Physics Project I (ENPH 353). Our task was to create a control system for a virtual robot that could navigate a simulated competition environment in Gazebo, identify alphanumeric characters in the world, and finally detect and avoid certain obstacles in simulation.

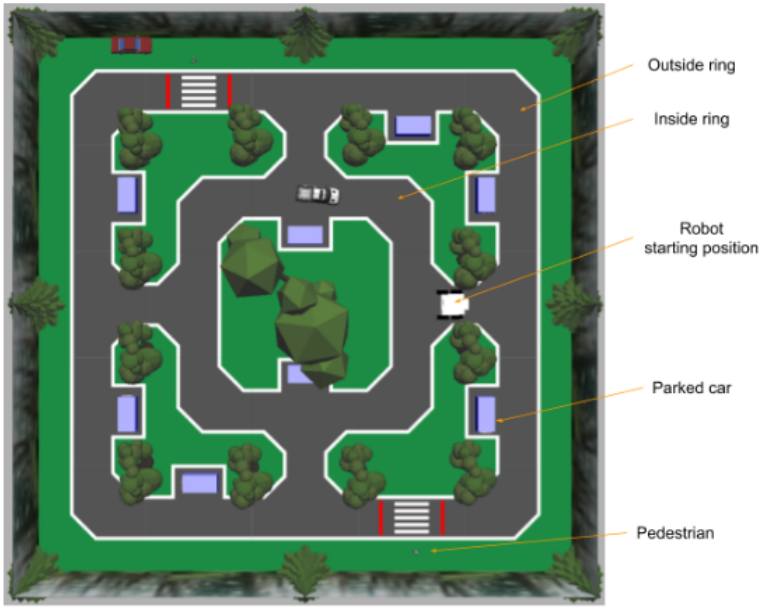

Figure 1.1: Diagram of the virtual competition environment

For the competition, our control system was to navigate an autonomous agent around the simulated world in Figure 1.1. Obstacles such as pedestrians and a patrolling car were to be avoided at risk of severe point penalization. The main goal was to read alphanumeric license plates located on the backs of parked cars, and report the characters to a score tracking system.

In this report, we will first go over our goals with regards to the competition rules, and explain how these influenced our design choices. We will also cover our implementation of these design choices, and finally address our robot performance and potential improvements to be made to the agent for future use.

2. System Design

With the competition rules in mind, we needed to establish expectations with regards to robot performance, as well as a general idea of how to approach the problem. We will also briefly go over our system architecture and repository layout.

We agreed that our main goal would be to simply complete the competition and collect all license plate data without being penalized for colliding with environment obstacles. Thus robot speed was not a major concern; it was more important to us that our robot was staying within the competition rules than potentially risk violating them.

With regards to our agent's navigation system, we felt that we had a strong grasp on classical computer vision control. Thus we opted for this approach to navigate the robot as opposed to a more modern approach of using Q-Learning or similar methods.

For character recognition, we opted to use a convolutional neural network (CNN). To recognize the license plates in simulation, we also used the scale-invariant feature transform (SIFT) for feature detection of the license plates and parking identifiers. Network architecture and implementation of SIFT will be addressed in detail in their respective sections.

Finally, robot communication was implemented using a publisher/subscriber model with the associated `rospy` library.

2.1 System Architecture

In designing our project layout we decided to prioritize compactness over modularity, so as to improve ease of collaboration as the scope of individual components increased. This design decision is reflected in our architecture, seen in Figure A.1.

We decided early on in the development of our project that we would not be using a CNN for use in navigating the competition surface, and thus no separate environment was required for the training of such a model. This allowed us to include our `cnn_trainer` inside our source directory, increasing the compactness of our layout. As expected, the `mcqueen_controller` directory contains all files relevant to the control of our robot. The contents in each directory will be discussed further in their respective sections.

2.2 Robot Navigation

Without a reliable way for the robot to navigate the competition surface, any attempt to record license plate data would surely be doomed. It follows that a concrete robot navigation structure was necessary to ensure the reliability of any plate reading techniques that were to be implemented.

2.2.1 General Movement

We prioritized completing the competition more than winning it, so the speed that came with the machine learning approach was traded for the deterministic control and safety of the classical

computer vision approach. Additionally, since the license plates were on the backs of the parked cars, we needed to go around the loop in a counter-clockwise fashion.

Figure 2.1: Outer border of the competition environment

Similar to the line following lab, the premise of our line following movement is that our robot follows the outer border of the track. We also had to consider that the robot needed to stay on the gray tarmac in order to avoid incurring a -2 point penalty. We needed to devise a control algorithm that would guide the robot along the white border but not be on top of it.

First, we partitioned a vertical slice of the right side of camera feed on the robot. The slice contains the right border of the track in its view. Next, we applied a binary threshold onto this slice. The processed image will be white where the right border is, and black everywhere else. Then, from the `scipy` package, we calculated the center of mass of the white border. Effectively, this gave us a reference point in order to steer and turn the robot. If the center of mass shifted too far to the left (i.e. the robot veered too far to the right), we would simply correct this movement in the callback function by turning the robot slightly left. Similarly if the center of mass shifted too far to the right, then we would correct this by turning the robot slightly right. If the center of mass fell in between some acceptable middle range, we would simply go straight. This range was empirically determined by means of numerous trial runs and rigorous testing in which the robot travelled at speeds we deemed safe.

Here is the control algorithm; the implementation can be viewed in the GitHub repository in `src/mcqueen_controller/src/controller.py`.

1. Receive camera feed from subscriber and partition a vertical slice of the camera feed on the right side, such that the track border is in view of the slice.

2. Apply a binary threshold to the slice, effectively blacking out everything except the white border.

3. Calculate the coordinates center of mass of the processed slice, then use the centroid coordinates as a reference point.

4. Adjust the robot movement to the left if the centroid coordinates fall too far to the left of some range or to the right if the centroid coordinates fall too far to the right.

5. If the coordinates fall inside this range, the robot is currently within acceptable road boundaries and thus we can travel straight ahead.

Observe that this control algorithm also accounts for when the robot must turn at the corners of the track. For inner ring navigation (which we were not able to implement due to time constraints), this control algorithm could have been easily modified to compute the center of mass of the left track border once we had completed a lap of the outer loop. A visualization of the robot following the center of mass of the line can be viewed in Figure A.2.

2.2.2 Pedestrian Detection

Initial testing of using homography to detect features in the pedestrian model proved fruitless. It didn't seem that there were many identifiable features for the pedestrian. We instead opted to use color masking from the camera to first detect whether we were at a crosswalk, then to see if the pedestrian model was in frame.

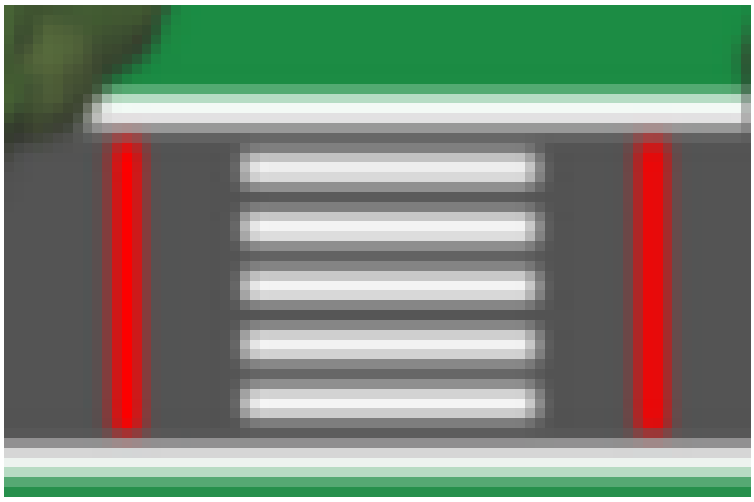

Figure 2.2: Crosswalk with red stripe

First, every time we invoke the callback function from the subscriber to the camera feed, we check if the robot is at a crosswalk. To do this, we first get the frame from the camera feed, then partition a small rectangular bar at the bottom of the frame. Note that we need to convert this bar into the HSV color space. Next we apply a color mask that filters the red in that rectangle. Effectively, this works the same way as a threshold, except we are filtering by red instead of brightness. Then we compute the average amount of red in the bar. This works because the bar will always have the same number of pixels in it, and we can say with confidence that the agent is at a crosswalk if the level of red in the bar is above a certain threshold. A full camera view of what the red masking looks like can be found in Figure A.3.

1. Each time the callback is invoked, get the frame from the camera feed and partition off the small rectangular slice near the bottom of the screen.
2. Convert the slice into the HSV color space, then mask the red pixels and compute the total amount of red pixels in the slice.
3. Divide the sum by the rectangular dimensions to get the average amount of red in the slice.
4. If the average amount of red is above an empirically determined value, then we are at a crosswalk.

Detecting the pedestrian was a similar case with different parameters, and additional states. Once we determined that the robot was at a crosswalk, we needed control conditions depending on the state of the pedestrian to determine the robot's next move. First we set the state of the robot to a waiting state, where it does not move and waits for the pedestrian to come into view. Note that the state is preserved each time as the callback is invoked. We can detect when a pedestrian enters the frame by means of masking the camera frame to filter out blue pixels. We know the pedestrian is in frame when the number of blue pixels is greater than a certain amount. We wait for the pedestrian to leave the frame and then we update the robot's state to leaving the crosswalk,

making sure not to check for red as above, as the robot has to leave the crosswalk and pass over another red bar. A visualization of what the controller processes when masking the pedestrian legs can be seen in Figure A.4. A full implementation can be viewed in the GitHub repository at `src/mcqueen_controller/src/controller.py`.

1. We are at a crosswalk, so set the robot's state to be waiting for a pedestrian to enter the crosswalk frame
2. Mask the frame for blue pixels. We know the pedestrian has entered the frame once the number of blue pixels exceeds some empirically determined amount.
3. Update the robot's state to 'seen pedestrian' and have it idle till the pedestrian is out of frame (when the number of blue pixels falls below the determined amount).
4. Once this condition is reached the robot's state is set to 'leaving' and is allowed to move forward without checking for a crosswalk.
5. After the robot turns we know that we have passed the crosswalk and thus the robot is allowed to look for crosswalk red pixels again.

As an aside, we would like to address why our robot seemingly stalled while passing the second crosswalk during the competition. The robot waited for the pedestrian at the second crosswalk as expected. Once the pedestrian left the frame, the robot proceeded forward without checking for a crosswalk. However, before it left the crosswalk, the controller left the 'leaving' state, so our algorithm permitted it to look for a crosswalk again and thus saw the red bar by the crosswalk's exit. It then got stuck in the 'waiting' state forever, looking for blue pixels that would never come into frame. It should be noted that this issue only occurred one other time out of hundreds of trial runs; in all other trial runs our robot bypassed the crosswalk safely and correctly. We assumed that the timing was sufficient, which was unfortunately not the case during competition.

2.2.3 Truck Detection

At the outset of the project we had earmarked using homography for detection of the truck in the inner loop of the competition circuit. Since colliding with this vehicle incurred a 5 point penalty, having a robust system in place to avoid doing so would be paramount to success in the competition. This system would have involved homographic detection of components of the truck visible at different viewing angles, potentially the rear bumper, front grill, and rear wheel. However, as mentioned previously several extrinsic factors made development of this task infeasible, and so we unfortunately failed to develop this aspect of the robot.

2.3 License Plate Recognition

The core of our project is based in our license plate recognition and detection. We opted to use three different CNNs: one each for letter and number detection in the license plate, and one for parking identifier detection. Although we trained each CNN on different data-sets, their underlying architectures were nearly identical. This design choice was permitted by the similarity in the quality and content of the images on which the models would be trained. Additionally, training separate models allowed us to narrow each of their input spaces, making them more focused in order to reduce frequency of false positives, e.g. mistaking 8's for B's. Since the first two characters of a license plate are always letters, and the last two are always numbers, we can simply use one model for the first two characters and another for the last two.

2.3.1 Data Generation

The first stage in training our network was to generate our training data. To do so we modified a license plate generation script from the competition package. This script allowed us to generate images containing random license plates and plate identifiers of a format similar to those found in the competition environment.

We had three models to train, so it was important to make the plate generation customizable. As a result we modified our script so that the invoker had a choice as to what number parking identifier, or specific plate characters, were to be generated. There were also options to leave these parameters as randomly generated. A full implementation can be viewed at `src/cnn_trainer/src/plate_generator`.

2.3.2 Convolutional Neural Network Architecture

We used an iterative approach when designing our model architecture. We initialized our design parameters, which included our training data set size, training/validation ratio, number of epochs, and many more, to known quantities, and then adjusted them based on feedback gleaned from our model accuracy/loss plots and the performance of our model on our test set.

Based on the strong performance of the model we created in Lab 5, we opted to set our model parameters to mirror those found therein. With some small adjustments to the layers and input size, it hit acceptable levels of accuracy (80%) just after 5 epochs at a learning rate of 1×10^{-4}.

Some issues we ran into early on were that our model was predicting that '8's were 'B's. This inconsistency is what motivated our decision to segment the plate model into two dedicated models, one exclusively for character training and number training respectively. This decision largely remedied this issue in future renditions of our CNN's. Furthermore, although our model was predicting with extremely high accuracy in our validation set, when using the model to predict from a separate test set (in-simulation screenshots) it was not faring well. We addressed this by applying augmentation to our training and validation data-sets.

2.3.3 Data Augmentation

It is generally accepted that training off of images taken exclusively from one's test set is bad practice. As a result, we opted to train our CNN's off of augmented versions of the ideal plate and position images generated through our aforementioned plate generator script.

To do so we leveraged the `tensorflow` package tool called `ImageDataGenerator`, which bundles simple image transformations such as image shearing, rotation, translation, zoom range, and brightness adjustment. It also allowed us to define preprocessing functions to generate Gaussian blur in the images or dilate and erode features.

This data generator created a stream of augmented images that streamlined the input of our training and validation sets into our model through use of its fitting function. Our augmentation parameters can be found in Table A.1.

2.3.4 Model Performance

With the above model architecture, we defined our steps per epoch to be the size of our dataset divided by our batch size. We selected a batch size of 32.

Our model trained relatively quickly at a learning rate of 1×10^{-4}, along with the above hyperparameters. To avoid over-fitting we made sure to augment the data heavily and provide a large dataset (around 5000 items).

Initially our model design was more complex, similar to the cats and dogs classification model. However, this complexity proved to be detrimental to its performance, with it barely reaching 50% training accuracy after 10-20 epochs across all models. Consequently, we decided instead to go for an architecture similar to the one in lab 5 for all three models.

With adjusted parameters and a reduction in the number of layers, we found that we were reaching high levels of both training and validation accuracy at around 5-15 epochs. Additionally, the model history also implied that our model was neither under nor over-fit. Graphs displaying our final model fitting can be found in the appendix; Figures A.5, A.6 correspond to the letter model, A.7, A.8 to the parking identifier model, and A.9, A.10 to the number model.

2.3.5 Plate Homography

Having a CNN to detect characters would be of little use to us if our robot couldn't send it characters to detect. As such, an integral part of our robot design was our homography detection algorithm, which allowed us to glean images of the license plate and car number from a passing car.

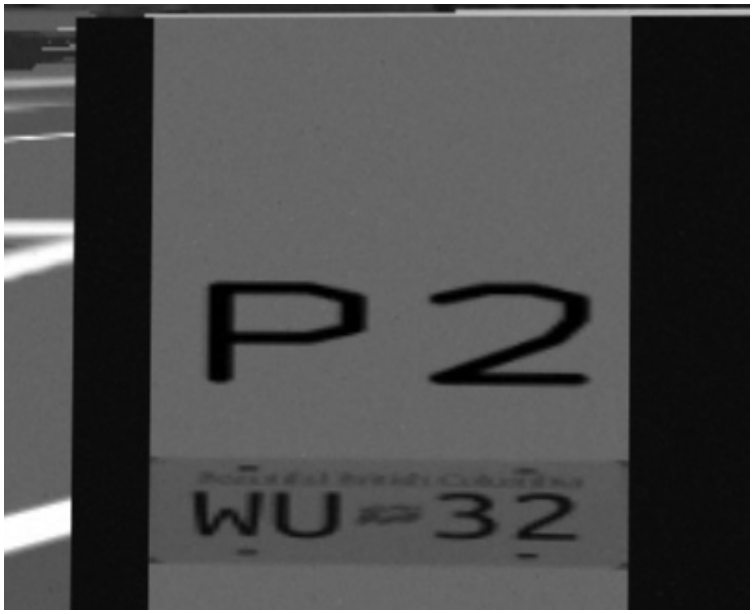

Figure 2.3: Example of car match following perspective transform.

Since our homography methods were unique functionally from the rest of the robot, we decided to spin them off into their own package to improve modularity. Within this package was the `Homography` class, which was initialized on the execution of our controller with our `PlateReader` class which contained the methods required to publish to the license plate topic, as well as our CNN models for car number identification and license plate character recognition. After initialization, we subscribed to the `image_raw` topic, and on callback our homography implementation worked as follows:

1. Call the `run_homography` method from inside callback with callback image. Grayscale and crop said image to minimum allowable dimensions to improve execution speed. This was permitted by us recognizing that a car would only appear on the left side of the frame on the exterior loop.

2. Retrieve car reference image based on previous match from CNN. This property is initially set to the first plate we expected to see in the competition, P2.

3. Run homography algorithm on these two images, generating key-points and homography matrix.

4. If the number of key-point matches is above a threshold, perform a perspective transform on the source image with respect to the homography matrix, and return this image.

5. Use this image to generate two sub-images containing the plate number and license plate, respectively. Since we found the perspective transform of the match to be fairly consistent, this slicing occurred with respect to predefined coordinates on the cropped images. Further segment the license plate into individual characters, and apply inverse thresholding and either dilation or erosion depending on car location.

6. Generate model prediction on the car number and license plate. If the prediction confidence was above a certain threshold, update the reference image to the next expected plate, and publish guess to the `license_plate` topic.

While this implementation had the potential of working well, it had a few key flaws that severely impeded our competition performance. Firstly, this system relies on consistently guessing the car number correctly. If the car number guess is incorrect, then the next reference image for use in the homography will be different than expected, and too few key-points will be registered to confirm the next match. This issue actually occurred during the competition, and a plate was missed as a result. The source of this issue directly relates to our second and most significant flaw: the homography image we obtained was far too inconsistent to warrant a static splicing method.

In our homographic images we routinely had plate characters out of frame, cut in twain, or at odd angles, all of which severely limited our model's ability to make consistent guesses of plate characters and car numbers. Essentially, we were leaving the quality of segmentation up to chance. A properly segmented image can be seen in Figure A.11, while a poorly segmented one in Figure A.12. This issue could have been remediated through use of a contouring system, which we began developing but failed to finish in time for the competition. We will discuss this system in the following section.

2.3.6 Letter Contouring

Confronted with the issue of having inconsistent image characters being produced by our initial slicing method, it was clear we needed a way to produce reliably segmented license plate characters for use in our CNN. The solution we arrived at would have been a procedure of image contouring and boxing to isolate and segment characters from the plate image.

This implementation would leverage OpenCV's `findContours` method, which finds connected bands of pixels with consistent brightness, similar to what would exist in characters framed by a license plate. Using the contours produced by this method we could then segment the image into rectangles containing the contours, which we could then pass to our CNN model to retrieve the plate characters.

Implementing this functionality would have greatly improved our competition result. Our model had relatively strong performance when guessing images of consistent formatting and framing so it stands to reason that supplying a consistent set of images to the CNN, instead of a highly variable and often indiscernible image set, would have removed the inconsistency we experienced in our model prediction. Unfortunately, as was mentioned before, we failed to implement this functionality before competition day.

3. Conclusion

3.1 Challenges

A few technical challenges beset our group over the course of the competition. While these challenges were seemingly unavoidable, better time management would have lessened their impact on our project timeline. One of these issues was a subscriber lag when running costly algorithms in the callback of our subscriber nodes. Specifically, when we attempted to perform operations on the image we obtained from the `image_raw` topic in our homography package, it appeared as if the `image_raw` callback in our robot movement class stalled until said operation was complete. This created a logic-breaking delay whenever homography was used, which was necessary for completing the competition. After multiple days of debugging, we ultimately fixed the issue through adjusting the subscriber node parameters by setting the queue size to 1 and greatly increasing the buffer size.

3.2 Future Improvements

In working on this project we gained valuable insight into multiple technical domains, including computer vision, robot control, and machine learning. Combining this new conceptual understanding with techniques learned through labs and lectures, we were able to develop a robot that could navigate a competition surface and retrieve license plate data from passing vehicles.

While our robot is functional, there are still many areas where it can be improved. For example, improving the reliability of our image segmentation through use of a contouring algorithm such as the one mentioned previously would make the data stream sent to our CNN much more consistent, increasing the confidence of our model output.

Over the course of the project we also gained a renewed appreciation for effective time management, and how it can mitigate the impact of technical issues on project timelines. We intend on placing much more emphasis on setting clear expectations and deadlines in our projects going forward.

In summary, this project allowed us to gain practical experience leveraging intimidating concepts like computer vision and machine learning to complete a project over a short time horizon. The lessons we learned and the skills we gained in developing our agent will serve us well in our future endeavours.

A. Appendix

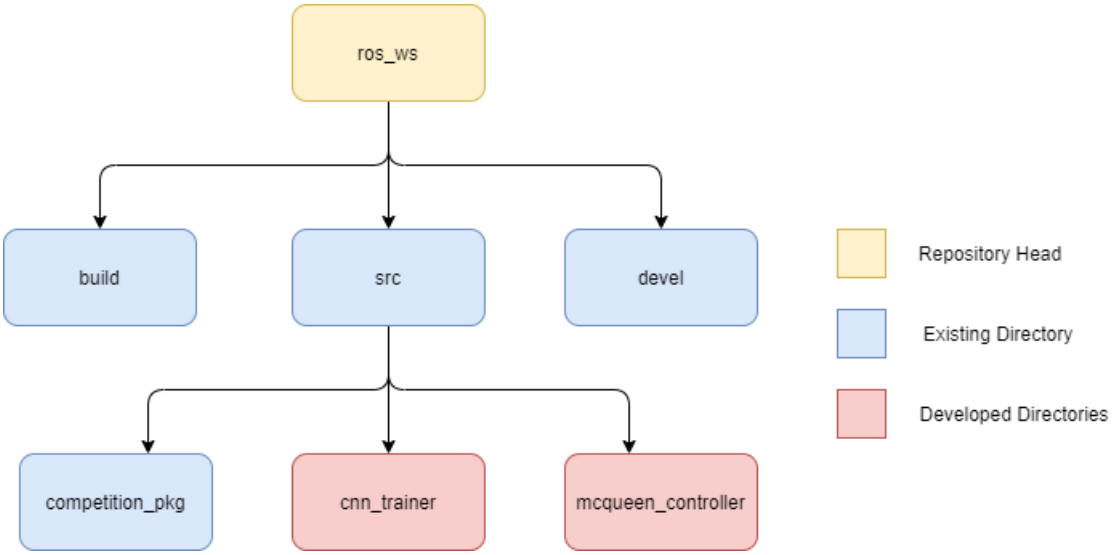

Figure A.1: High level repository architecture.

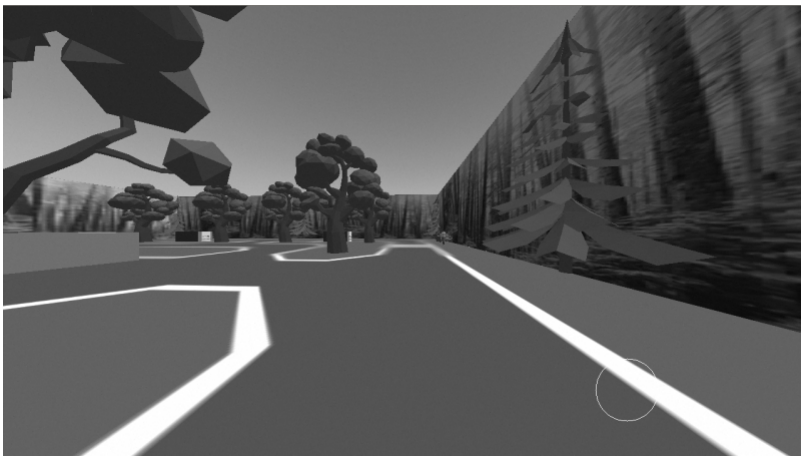

Figure A.2: Visualization of robot following the center of mass of the white track border (circle in left corner of screen represents the center of mass)

Figure A.3: Red masked camera feed from the robot (note that only a small portion of this feed is used to calculate the average red)

Figure A.4: Portion of the camera feed showing pedestrian legs blue masked

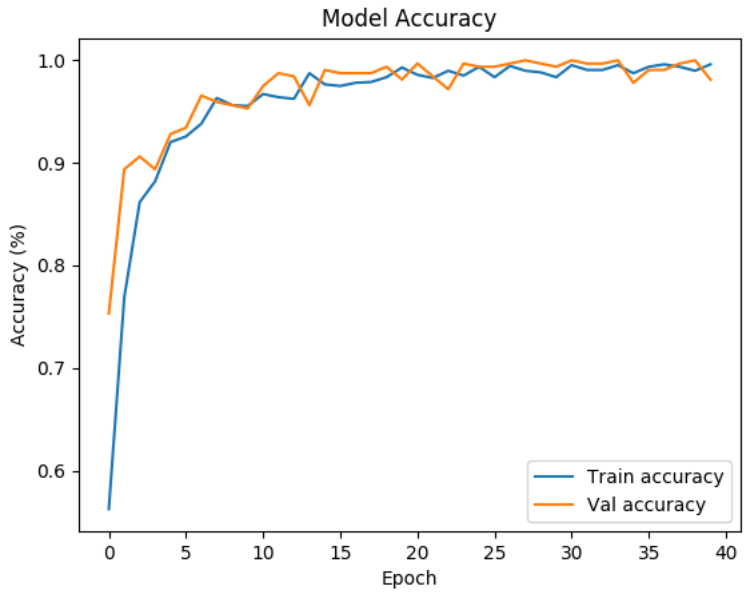

Figure A.5: Final Letter Model Accuracy

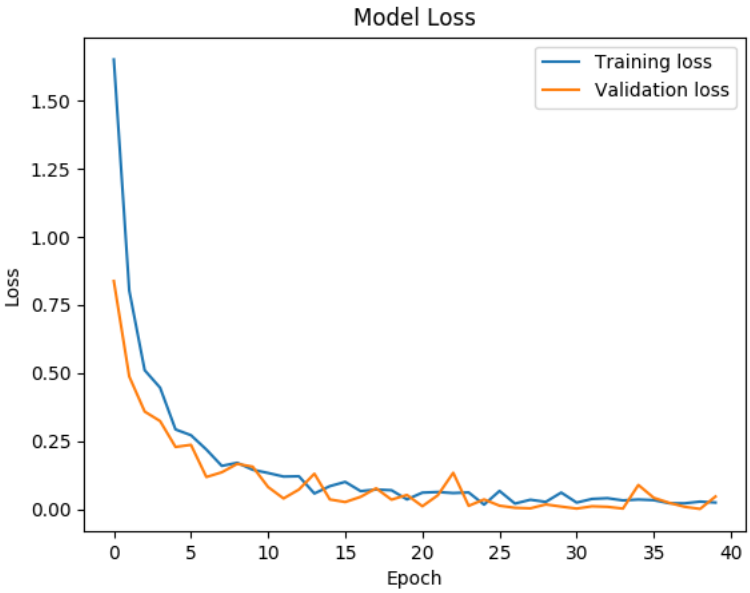

Figure A.6: Final Letter Model Loss

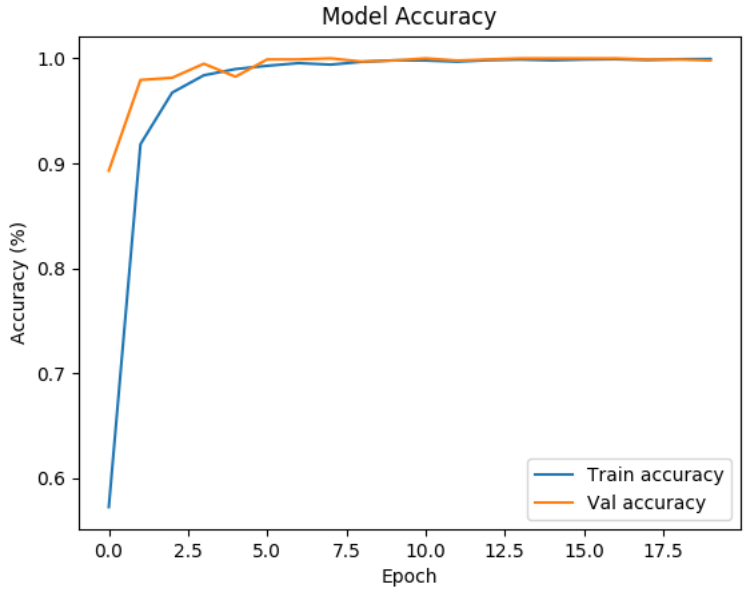

Figure A.7: Final Parking ID Model Accuracy

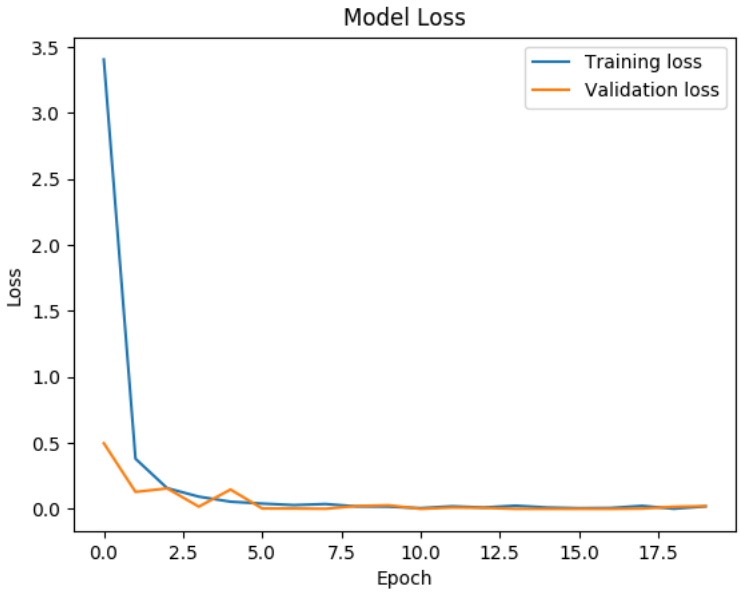

Figure A.8: Final Parking ID Model Loss

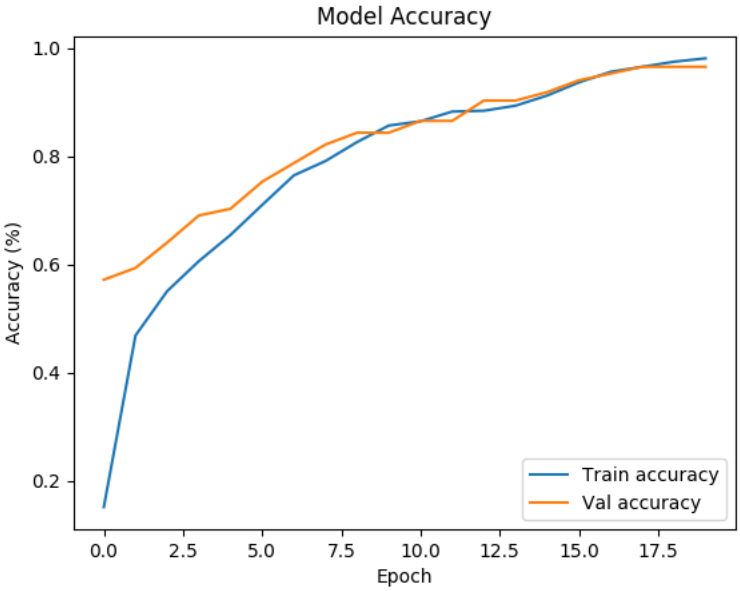

Figure A.9: Final Number Model Accuracy

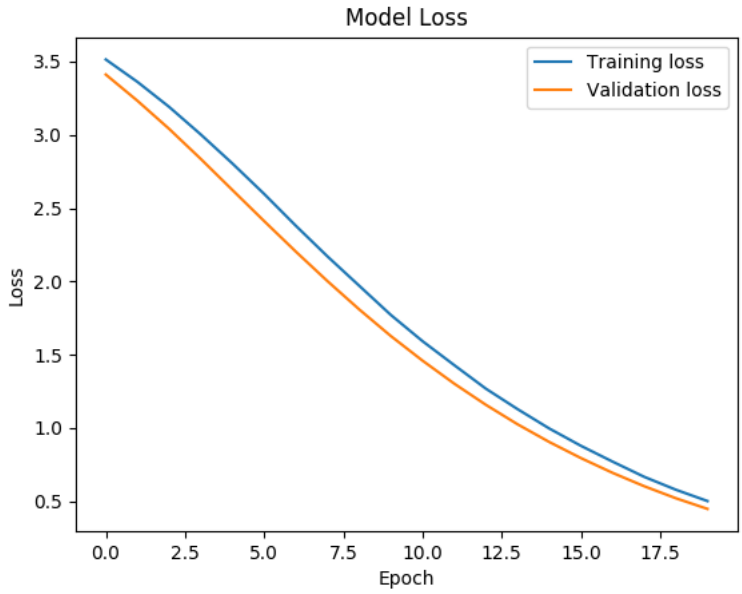

Figure A.10: Final Number Model Loss

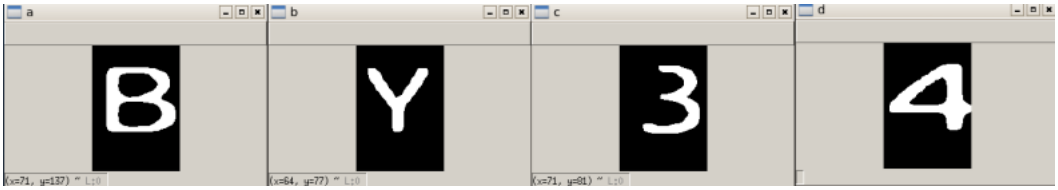

Figure A.11: Properly segmented letters (BY34)

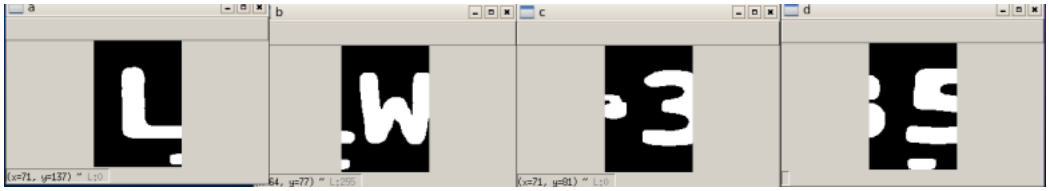

Figure A.12: Poorly segmented letters (LW35)

Table A.1: Augmentation Parameters

Augmentation Parameter	Value	Unit
Shear Range	3	-
Rotation Range	2	degrees
Zoom Range	0.75-2.5	-
Horizontal Shift Range	-32 - 32	px
Vertical Shift Range	-32 - 32	px
Brightness Range	0.8 - 1.1	-
Gaussian Blur Intensity	0 - 10	-