Machine Learning Fizz Detective Robot
UBC, ENPH 353

Ian Hartley
Cyrus Young

Team 5
December 6, 2023

Table of Contents

Introduction:

In this project, we designed a robot (with the appearance of a four-wheel car) to drive around a course and identify clues (in the form of signs adjacent to the road). The robot has a camera (within the simulation) and is thus provided with a live image feed, which we used to identify the signs using a neural network. There are 8 signs that have clues on them that are in the form "{Descriptor} {Thing}" such as "VICTIM ROBOTS". A neural network then identifies the words and puts the words in a chatGPT API which makes a story based on the clues.

On the road, there are two pedestrians (a generic human and Baby Yoda) as well as a truck. Points are awarded for obtaining clues, and reaching the tunnel without the use of teleportation. Points will be deducted for driving off the road (i.e. having three or more wheels off the road simultaneously), crashing into pedestrians or the truck, or teleporting the car to any place on the map. A birds eye view of the setup is shown below:

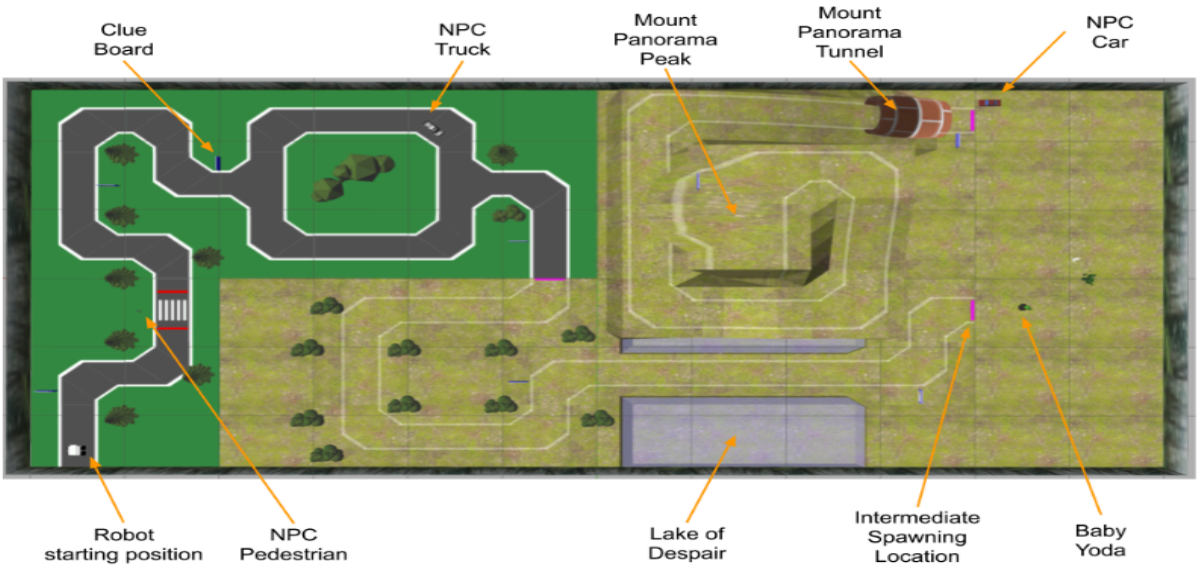


Figure 1: Birds eye view of the course

General Approach and Contribution Split:

Our strategy for the competition was to maximize the number of points as possible. This means we were not utilizing teleportation and planned to drive the entire course. Generally speaking: use PID for driving around the course, a filter for identifying the presence of plates, and a CNN for predicting the contents of the plate.

We started the project by working together and brainstorming on the general strategy we would take. We further worked together on the PID for a small amount before separating tasks.

Ian primarily focused on the driving module, while Cyrus concentrated more on plate recognition. We both worked on the neural net with Ian also further emphasizing the training. However even though we did separate tasks, during this we were in constant contact and giving/receiving feedback on the progress with frequent progress reports (often on a daily basis) and worked together frequently closer to the competition. In the last days before the competition we mostly worked together on improving the neural net.

Overarching Software Architecture:

We started by creating two repositories, one of them was called "controller_pkg" and had all of our main programs (such as our driving module as well as our plate recognition module). The other repository was called "ros_ws_dev" which we created at the beginning but we did not use.

The controller_pkg repository consisted of three main programs; Camera.py and Partitioner.py, which consisted of our plate recognition module, and timetrials.py which was used for our driving recognition module. Camera.py subscribed to the /R1/pi_camera/image_raw and in turn, license plates are identified (further expanded upon in the sections below) and saved in the folder '/home/fizzer/ros_ws/src/controller_pkg/src/scripts' as a png file. Partitioner.py manipulates the aforementioned images, the process which is detailed below, and predicts the content of the images. The image predictions are later published to score_tracker.py. The program timetrials.py was used for the driving recognition module.

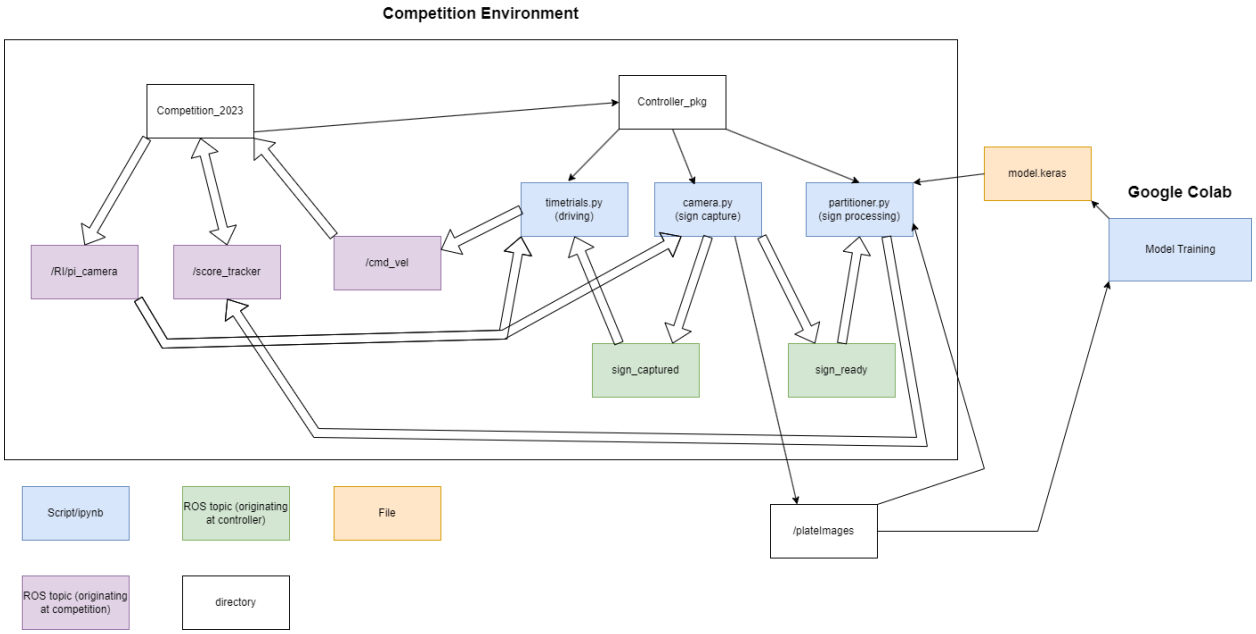


Figure 2: Overview of software architecture for the project.

Plate Recognition Module:

As the agent drives around the course, it has a camera feed that first involves applying a nuanced filter that analyzes each of the three channels for each pixel (red, blue, and green). The first step was to apply a mask to the camera.

The mask works as follows; every pixel in which the value of the blue channel is at least 1.8 times as large as both the red and green channels now has all three channels become zero (i.e. they become fully white pixels) while the remaining pixels take on the values [255, 255, 255] for each channel, corresponding to fully black pixels. The filtered image is then converted to grayscale and using OpenCV, contours are found due to the color gradient.

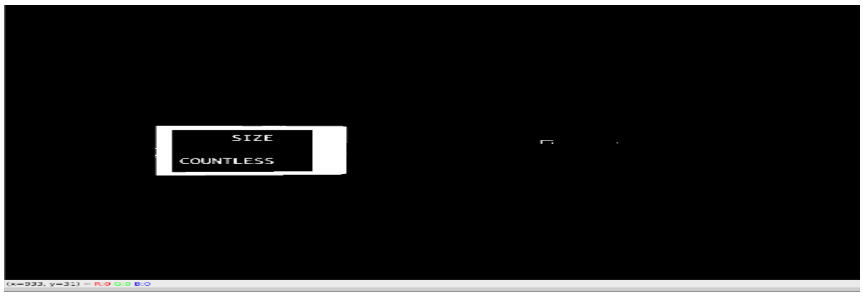


Figure 3: Image of sign after mask was applied to camera

Next, the program measures the length of the detected contours and finds the area of all quadrilaterals in the filtered image by finding all portions of the image in which four contours are connected. The maximum area is recorded and is the area of the face of the sign (which includes the plate as well as the blue background of the sign). The next step was to obtain a picture of just the plate and not the entire face of the sign. We did this by isolating the largest quadrilateral and then applying a perspective transform on the portion of the image which does not satisfy the condition that the value of the blue channel is at least 1.8 times as large as the red and green channels. We then took a picture of this portion (on the original camera) to be used for our character detection program.

Using a kernel and contours, the original image is partitioned so that there is one sub-image for each individual character in the original image. By naming our signs the exact name of the words contained on it, and by partitioning the sign we can automatically extract an image of each individual character and have the name of the file be the character that this contains. For training those images in our image recognition neural network, a HSV filter is applied to each of the images and each image is scaled to a size of 42 pixels (width) by 60 pixels (height). An example of such an image is shown below:

Figure 4: Picture of individual character after HSV filter is applied

For training our model, we used a Convolutional neural network and utilized both the Keras training framework as well as tensorflow to our advantage. As we ran simulations, we took pictures of the signs (as described in the second paragraph of this section), and saved them into our local drive. After doing many simulations, we then partitioned the signs and applied a HSV filter to them (as described in the third paragraph above).

In our final program, we modified our code so that the neural network only read the bottom half of the sign. We did this to decrease the chances of the neural network reading the descriptor wrong and thus getting the clue wrong.

Driving Recognition Module:

Driving Strategy:

Our robot used PID control to navigate for the entirety of the course. Our approach to PID was extremely simple, which allowed us to generalize our algorithm to the entire course, with only minor modifications depending on the section.

The basic approach is as follows:

1. Begin by manually creating a mask, which highlights only the lines of the road.
2. Then, find a location on the screen sufficiently high so that both lines can be seen by the camera at all times.
3. Scan this row of pixels from the outside of the screen inwards on each side, until a set number of white pixels appear in a row, and call this the location of the line
4. Take the average of the two lines to find the center of the road
5. Create a steering value based on the difference between the center of the screen and the center of the road

The only variations in this strategy were for the roundabout, the offroad section, and the tunnel. In the roundabout, we only searched for the left line of the road, which allowed us to easily enter

and exit the roundabout in the correct direction. For the offroad section, there were no road lines, so we used the cliff face as our reference, and did not use PID. In the tunnel, there were also no road lines. In this case we used PID to center us within the walls of the tunnel.

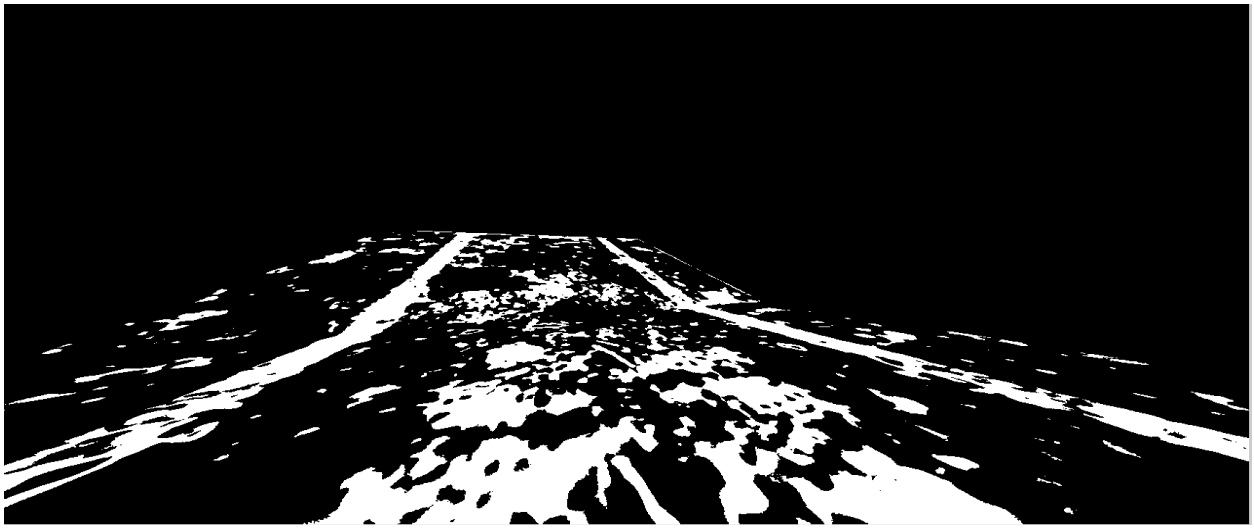


Figure 5: Sample image of filtered road being fed into PID algorithm (grass mountain section).

While our PID algorithm was quite simple, it meant that we needed quite different driving states, depending on our terrain, and potential obstacles nearby. In the end, we used 10 different driving states, each with different triggers to successfully navigate the course. Transition triggers included landmarks on the course, signs, differences in brightness, and detection of obstacles (See Appendix 4 for overview).

Humanoid and Vehicle Detection:

To detect the pedestrian, we used an algorithm that detected the change in pixels as the pedestrian crossed the road. Specifically, the camera attached to our robot started by detecting the red marking before the crosswalk. It then took an image of the scene in front of it, and periodically checked the difference between the current image and the image it took at the time of stopping. A high difference meant that the pedestrian was in a different location on the screen and had crossed the road. In retrospect, this algorithm would have been much more robust if only one side of the crosswalk had been analyzed, since our competition algorithm sometimes ran into trouble if the pedestrian was in the middle of the road when the first image was taken.

Vehicle detection worked similarly, except the current image was compared to the image before it. In this case, a low difference meant the truck was not in frame, and it was safe for our robot to proceed.

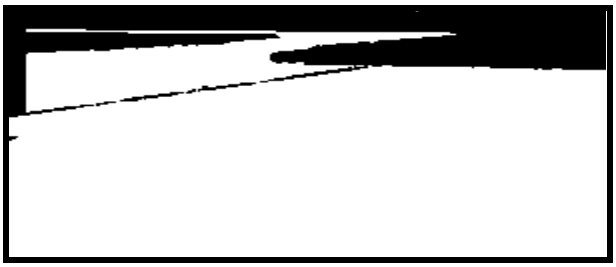


Figure 6: Filtered image of clear intersection

Avoiding Baby Yoda:

We did not have an explicit algorithm to detect Baby Yoda. However, we did have a routine to effectively avoid it. The first step was to detect the purple line labeled “Intermediate Spawning Location”. We did this by measuring the pixel values of a row in our camera, as the pixel values for the purple line were (R: 254 G: 1 B: 254), indicating that the purple line was close the the maximum values for the red and blue channels and close to the minimum value for the green channel. As soon as the robot saw the purple line, it rotated counterclockwise until it detected the side of the hill (which was on the robot's left). The robot then immediately sped straight toward the tunnel until it reached the purple line right before the tunnel.

Because the robot rotated while it was behind the purple line labeled “Intermediate Spawning Location”, it was not in the way of Baby Yoda and thus never hit it on our final version of the code.

Neural Network:

For our neural network, we used a relu as our activation function. We had a learning rate of 5*10^(-6) and a validation split of 0.25. We ran 100 epochs on the data and had a final value accuracy of 0.9978 and a final value loss of 0.0446. The model loss and model accuracy are shown below:

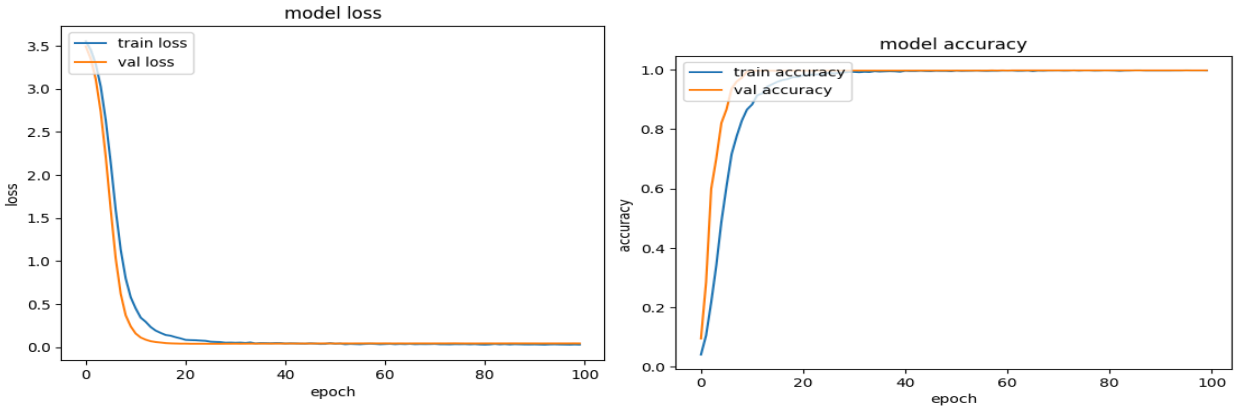


Figure 7: Model loss (left) and Model accuracy (right) for our neural network

Since the problem of recognizing characters of a single font is relatively simple, we found that most neural network architectures were quite effective. Through trial and error, we found that the only parameter that made an appreciable difference in the network's accuracy was convolution kernel size, with peak performance around a kernel size of 5-6.

Because many different network architectures gave us similar accuracy, we instead focused on making our network small. This allowed us to retrain our network very quickly (1-2 minutes), and perform our guesses on the fly in the competition, without interfering with our ability to drive.

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 56, 38, 32)	832
max_pooling2d_6 (MaxPooling2D)	(None, 28, 19, 32)	0
conv2d_7 (Conv2D)	(None, 24, 15, 32)	25632
max_pooling2d_7 (MaxPooling2D)	(None, 12, 7, 32)	0
flatten_3 (Flatten)	(None, 2688)	0
dropout_3 (Dropout)	(None, 2688)	0
dense_6 (Dense)	(None, 512)	1376768
dense_7 (Dense)	(None, 36)	18468

Total params: 1421700 (5.42 MB)
Trainable params: 1421700 (5.42 MB)
Non-trainable params: 0 (0.00 Byte)

Figure 8: Keras summary of final neural network.

Data collection and training

Access to the world generation files made generating artificial signs to train the network on extremely convenient, since it allowed us to generate visually identical signs to those in the simulation environment, or modify the signs generated as desired. Our initial testing with the neural network involved training only on these artificially generated signs, and augmented versions of these signs (e.g. blur). This allowed us to more closely match the conditions of the simulation, as pictured below.

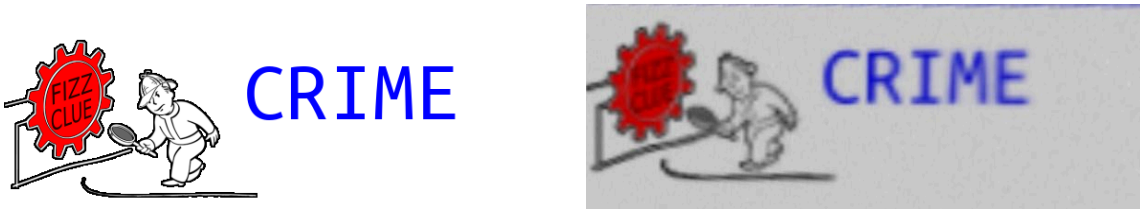


Figure 9: Side by side comparison of an artificially generated sign and one captured in simulation.

This first iteration of our network had an accuracy of approximately 97% per character when tested on real simulation data. While we were happy with this performance, with eight clues to guess per round, it meant that we typically guessed one to two signs incorrectly.

Having seen these results, we decided the best way to improve our results would be to augment our artificial data with real data from the simulation. Conveniently, we were tackling this issue at the same time as we were beginning to tune our driving for consistency. While tuning our PID, we also created a library of approximately 150 unique sign images, matched with labels created when the sign was generated.

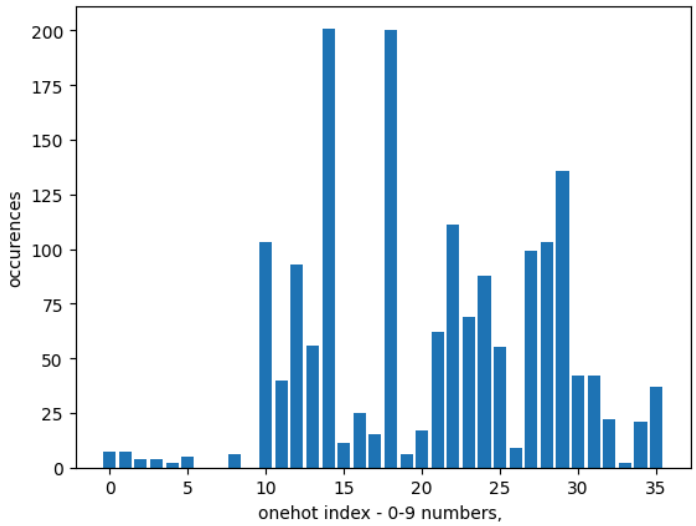


Figure 10: Distribution of characters in images taken using our plate recognition algorithm.

Since the distribution of characters in the simulation was uneven, we then augmented this set with the artificially generated signs used in our first neural network to even things out. After the set was generated, we used the sklearn python library to shuffle the data, and randomly select 25% of the characters to be used as validation data when training the network.

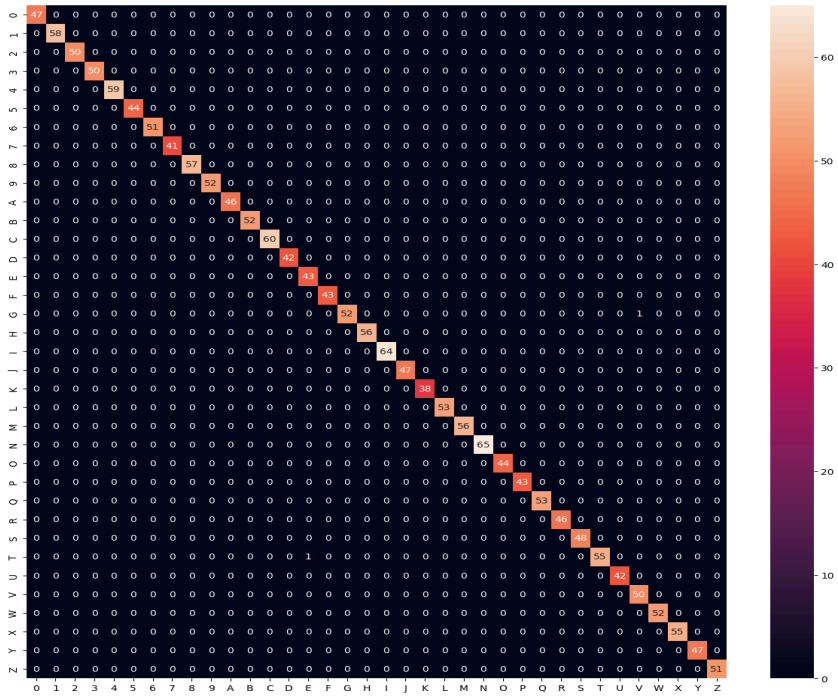


Figure 11: Confusion matrix for final iteration of neural network, trained on combo of competition images and artificially generated images to balance character distribution. Note: character distribution here is not completely equal, as the validation set was selected randomly from the set of images.

After retraining, our accuracy increased to around 99.8% on our validation set. While the new network did outperform the previous version, 99.8% is higher than our in-simulation accuracy. This is due to our new validation set including artificial images, which the network could guess correctly 100% of the time. In simulation runs, our network would guess all signs correctly around 50% of the time. Usually, incorrect guesses were caused by guessing “8” instead of “B”.

Reflecting on this result, there are a variety of options we could have pursued to make our network even more robust. First, we could have modified the in-simulation sign generation code to even out the distribution of characters in our collected data. This would remove the need for artificial data entirely, and could have made our network more accurate. Second, we could have run the network on a series of images of the same sign, to remove any chance of a single bad image causing us to guess characters incorrectly. Finally, we could have looked into the confidence of the network when making guesses - particularly for B’s and 8’s, to see if we could manually correct problematic guesses.

Performance:

Competition: Overall our robot performed well. Originally, there was a slight error in our code as the threshold for our pedestrian detection algorithm was no longer working. This meant that the robot became stuck at the pedestrian crosswalk. Thankfully, Miti allowed us to change the code. We immediately lowered the threshold and ran the simulation one more time. The second time, the robot detected the pedestrian, stopped for around 1 second and immediately drove after the pedestrian was off the road. The robot also successfully detected the truck and entered the roundabout without problems. The robot finished the entire course without collison or the use of teleportation, which allowed our group to get the bonus for reaching the tunnel. The only flaw in our run was that our neural network predicted the second sign wrong. Our neural network occasionally has trouble detecting ‘B’s and sometimes mistakes them for ‘8’s. For our run, the second sign had the word “RABBITS” on it, so our neural network likely mistook one of the ‘B’s in the word for an ‘8’. The neural network detected all other signs correctly giving us a total of 51 points out of a maximum of 57.

Conclusion:

In conclusion, we are happy with our final result. However, if we were to indulge in the hypothetical for a minute: able to redo this entire project, there would be some things we would change. First, we would try to standardize a real time factor starting at the very latest four to five days before the actual competition started. On the day of our competition, our robot would sometimes suddenly behave sporadically and drive off the road or have completely wrong predictions of the words. We later found that the main problem was the real-time factor. Initially it was 0.8, and we changed it to 0.7. It worked only slightly better than 0.8 but was still quite prone to errors (such as having two consecutive runs where it read “TWILIGHT” as “T77XT”). A mere 30 minutes before the run, we changed the real time factor to 0.75 and we had two perfect runs right before the competition run. As mentioned in the paragraph above, the only mistake we made in the competition had to do with the neural network.

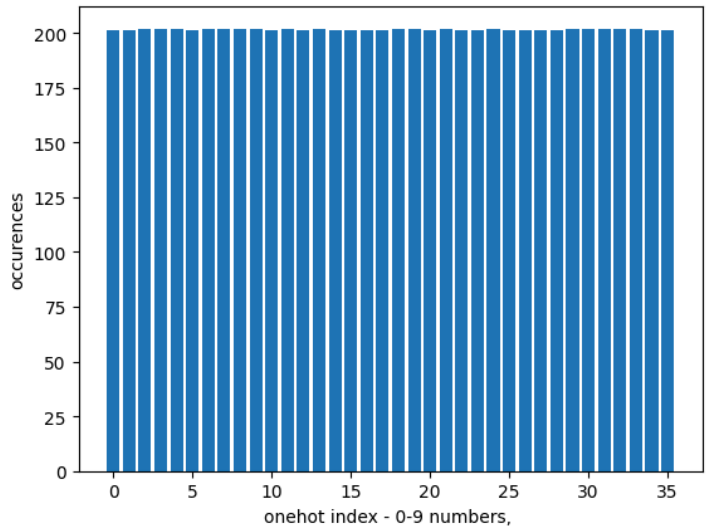
If we had an extra week to work on the robot, we would have added a more robust algorithm for sign detection. The algorithm that we had only took one image when it was close to the sign, which was risky because, although it did not happen often, it would sometimes miss the sign entirely. We would have altered the original sign detection algorithm to take as many pictures of a sign that it could possibly take (while the sign is within a certain size range), and then choose the highest quality image (which is the image with the most pixels).

In the non-hypothetical world though it was an intense learning experience and looking forward to building upon concepts we learned in future projects.

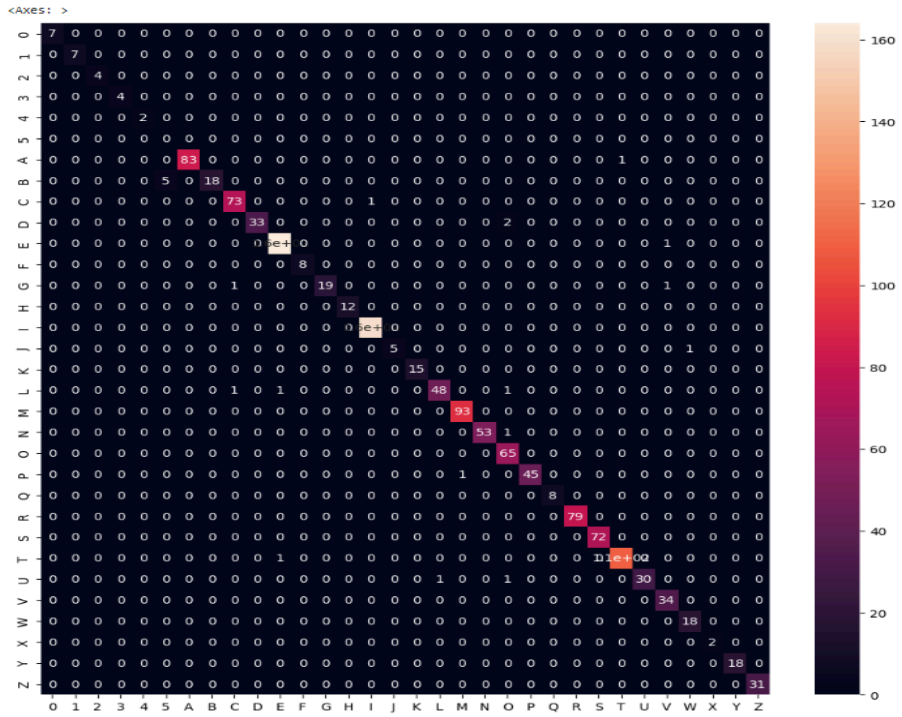
References:

1. Miti and TA office hours
2. ChatGPT (discussions in personal logs)
3. Reference OpenCV
4. ROS tutorials

Appendix:



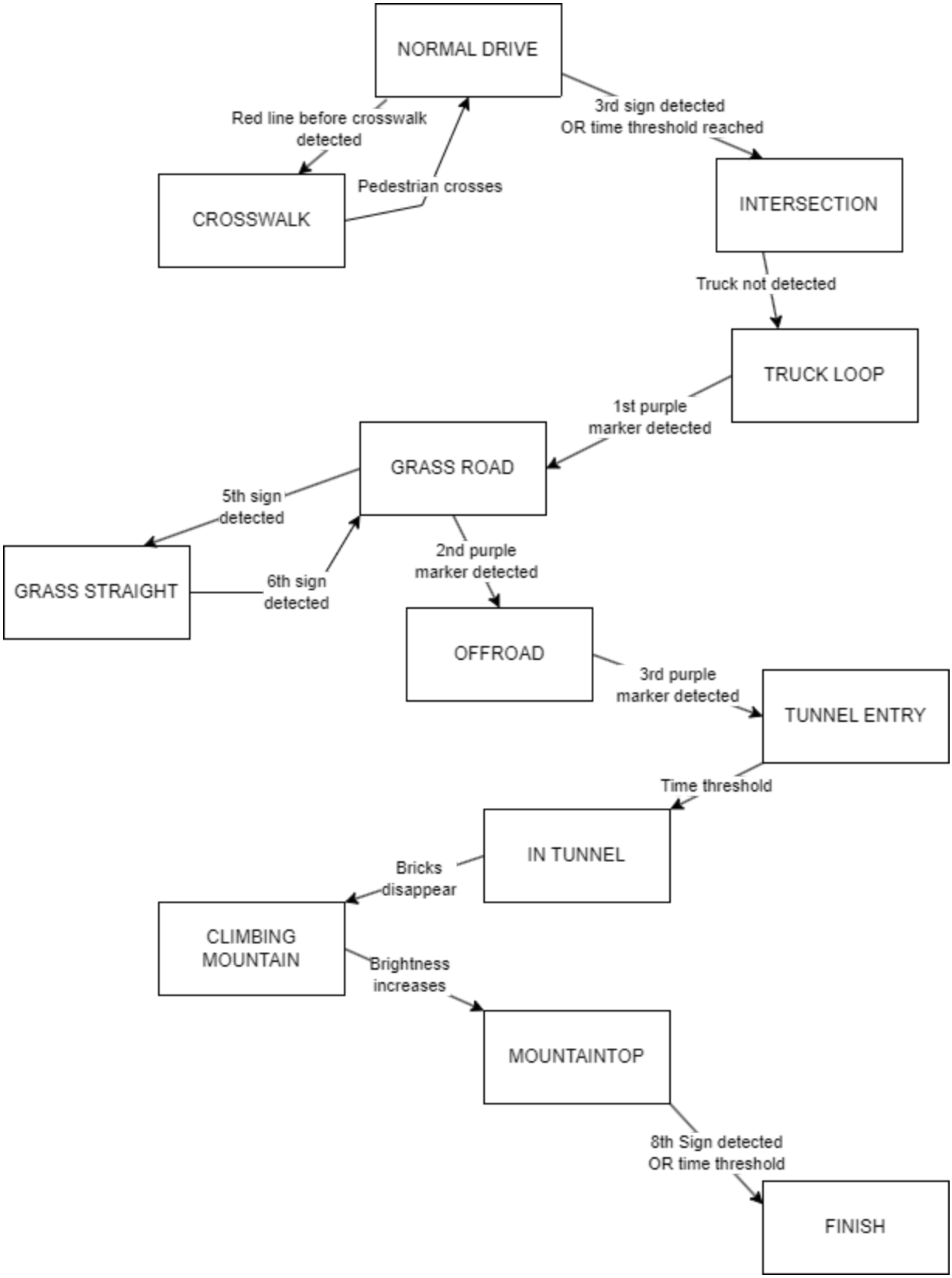
Appendix 1: Character distribution used for final network training.



Appendix 2: Confusion matrix for first iteration of neural network - trained on only 2 copies of each character, generated artificially. Confusion matrix generated from data collected in simulation.

```
conv_model = models.Sequential()
conv_model.add(layers.Conv2D(32, (6, 6), activation='relu',
                             input_shape=(60, 42,1)))
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Conv2D(32, (5, 5), activation='relu'))
conv_model.add(layers.MaxPooling2D((2, 2)))
conv_model.add(layers.Flatten())
conv_model.add(layers.Dropout(0.6))
conv_model.add(layers.Dense(512, activation='relu'))
conv_model.add(layers.Dense(36, activation='softmax'))
conv_model.compile(loss='categorical_crossentropy',
                   optimizer=optimizers.RMSprop(learning_rate=LEARNING_RATE),
                   metrics=['acc'])
```

Appendix 3: Code used to generate final iteration of neural network.



Appendix 4: Driving state machine overview