Combat Robotics @ Cornell

Project Team

Final Report

Firmware Subteam
Lance
Manny

Spring 2023
Cornell University

Contents

1 Introduction

1.1 Subteam Purpose

The purpose of the subteam is to investigate the potential of implementing autonomous features in our robots. To enable these features, the robots need to obtain real-time information necessary for their decision-making. To achieve this, we plan to add sensors to the robot. In this semester, we explore the installation of a rgb camera on two of our robots, Manny and Lance. This camera will allow them to perform autonomous tasks, including sign recognition and road tracking. Establishing robots' ability to perform tasks based on real-time sensor data is a solid foundation for future autonomous tasks.

1.2 Subteam Members

Richard Jin	CS '24	Subteam Lead
Sebastian Rivera	ECE '24	Vice Subteam Lead
Ethan Zhang	CS '26	Member
Grace Lim	CS '26	Member
Katie Huntley	CS '25	Member
Shao Stassen	CS '26	Member
Shawn Chen	CS '24	Trainer

2 Spring 2023 Progress

2.1 Logistics

Timeline

Week 1 and 2 (Jan Fab)	Informed newly incoming firmware members about the circuit and sign recognition code that we have done in the last semester
Week 3 and 4	Worked on improving the accuracy of the sign recognition model, attempted to develop a deep learning model for road tracking, and also explored about the integration between the sign recognition and road tracking programs
Week 5, 6, and 7	Continued to work on those tasks from week 3-4 and attempted to integrate the feature of turning upon recognizing a sign into the road tracking program
Week 8	Observed that Manny was poorly designed mechanically and physically unable to move consistently on smooth ground; tried to address the problem in various ways, but all attempts failed
Week 9 and 10	Reviewed and debugged both the sign recognition program and the road tracking program
Week 11, 12, and 13	Transferred the system from Manny to Lance, as Lance was better designed and able to move consistently
Week 14, 15, and 16	Started to work on code documentation, debugged the corner detection and sign recognition code, collected new data for training another road tracking model
Week 17	Fixed the problematic circuit, trained another road tracking model, and successfully integrated sign recognition and road tracking together; Lance can now follow the road and turn upon recognizing a sign

2.2 Progress Overview

During this semester, the firmware subteam focused on implementing sign recognition and road tracking features. To accomplish these, we chose to employ deep learning techniques for robots' decision making.

For sign recognition, we developed a deep learning model that could predict the direction of the arrow on a sign based on real-time images captured by the rgb camera. To train the model, we collected data by running Facial_Recog/Step1-Data-Collection/DataCollectionMain.py on the raspberry pi, pressing the 'share' button of the connected joystick, and then placing each sign in front of the installed rgb camera one at a time. We then trained the model locally by running Facial_Recog/Step2-Training/train.py on our laptops and later deployed it on the robots' raspberry pi. To demonstrate the sign recognition feature, we ran Facial_Recog/Step2-Training/FacialRecogVisualDemo.py on the raspberry pi, which displayed the predicted label based on the real-time images captured by the rgb camera.

For road tracking, we developed a deep learning model that could predict the steering angle based on real-time images captured by the rgb camera. The steering angle would determine how far left or right the robot needed to turn to stay on the road at that moment. To train the model, we collected data by running

RoadTracking/Step1-Data-Collection/DataCollectionMain.py on the raspberry pi, pressing the 'share' button of the connected joystick, and then remotely driving the robot on the road using the joystick. We then trained the model locally by running RoadTracking/Step2-Training/train.py on our laptops and later deployed it to the robots' raspberry pi. To demonstrate the road tracking feature, we ran RoadTracking/Step2-Training/try_out.py on the raspberry pi, which enabled the robot to follow the road and turn left or right upon recognizing the corresponding sign.

3 Designing and Implementing the Circuit and Code

3.1 Circuit

3.1.1 Problems We Faced With Manny's Circuit

The problems we faced with the circuits mainly had to do with Manny because many of the electronics (motor controller module, motors, and pretty much everything) on Manny were specced incorrectly. This is probably due to the fact that Manny was the first robot the team made, so there were many mistakes made by the Meches.

Our main issue was with the motor controller (l298n) as we began to burn through them (it's max current rating was too low). We went through at least 4 of these before we decided to stop using them. Also we had issues with Manny's driving capabilities. There were issues with the axles slipping which meant that the motors sometimes were turning without the actual wheels turning. The wheels on Manny where at different heights which made the driving extremely inconsistent, as Manny couldn't really turn in one direction and also couldn't drive straight.

This ultimately led us to using Lance instead so that we could have a more consistent robot that can drive well.

3.1.2 Lance's Circuit

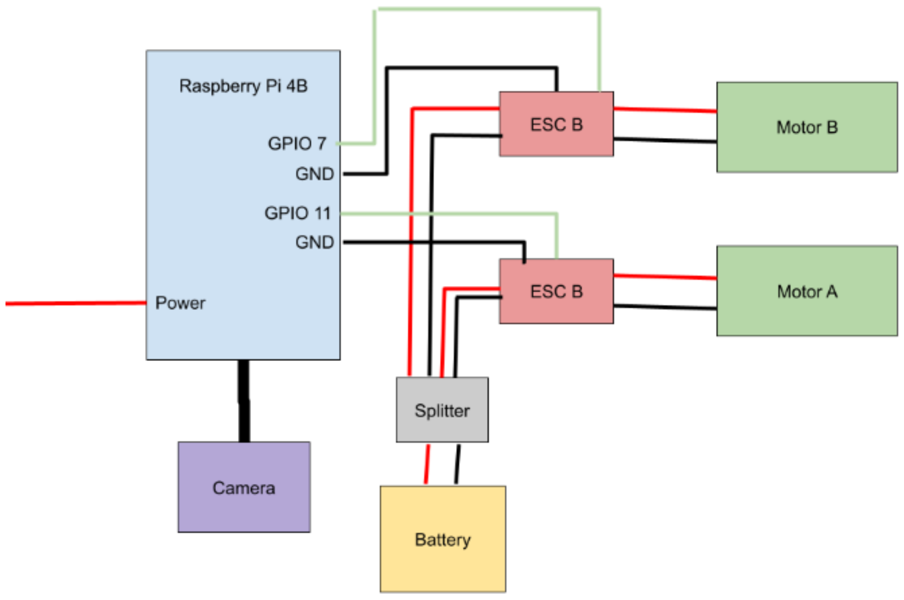


Figure 1: Lance's Circuit

The circuit on Lance was much more simple than the circuit on Manny, since all we needed to do was replicate the transmitter using the Raspberry Pi. Since the transmitter had a single wire as a control signal we checked to see what voltages were being outputted for specific functions and mapped that to duty cycles on the Raspberry Pi pins to replicate the transmitter. This worked perfectly and simplified the circuit greatly as the only thing left on the circuit was the RPI, two ESCS and the camera.

4 Sign Recognition

4.1 Image Processing

4.1.1 change_brightness.py

change_brightness.py uses all the images in the existing image folders we collected under one lighting condition and generates six additional images for each image with all different lighting conditions. This file will help the machine learning model more accurately recognize the "SebastianLeft" and "SebastianRight" signs with different lighting conditions. The goal of this is to manufacture more data sets for the machine learning model to be able to recognize different light conditions and use more data to boost the confidence of the model.

This Python code extracts all the file paths in the designated image folder and puts them into a list.

```
sright_train_img_paths = list(paths.list_images(SRIGHT_TRAIN_IMG_PATH))
sleft_train_img_paths = list(paths.list_images(SLEFT_TRAIN_IMG_PATH))
```

We iterate through each file path and use PIL library's module "ImageEnhance" to change the brightness of a designated image to the parameter value. The parameter value 1 indicates the original brightness of the image, while 0 indicates the completely dark image. We initially chose brightnesses 0.5 to 2 to just see how bright and dark the images would get. Each original image generates 6 new images with varying brightnesses, which we use to train a more accurate model.

```
filter = ImageEnhance.Brightness(image)
new_image_05 = filter.enhance(0.5)
```

After a filter is applied to the image, we save the image into a new image file path we generated. This will go under the new image folder for all the same types of images.

```
new_image_05 = new_image_05.save(f"{new_image_05_path}/SebastianRight_"
+ str(i) + ".png")
```

We originally intended to generate six new different versions of the same image using the original brightness: 0.5, 0.75, 1.25, 1.5, 1.75, and 2. But we found that the brightness of 2 had a really high exposure, which made it extremely difficult to differentiate details in the image, so we decided to not use that brightness. The model was unable to accurately predict images with a brightness of 2.0.

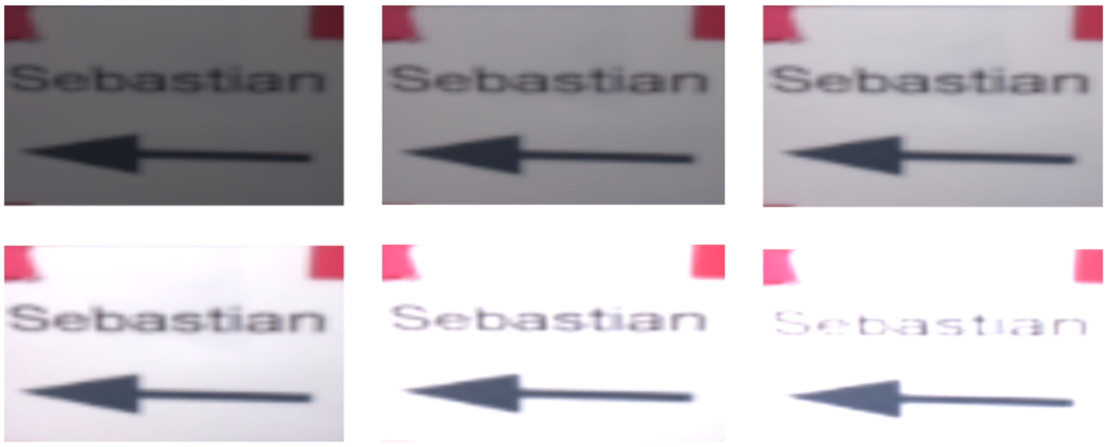


Figure 2: The different brightness used to train the facial recognition model

4.1.2 corner_detection.py

We use corner_detection.py for image processing. It takes in a captured image, searches for the red pixel on the four corners of the target sign, and returns the coordinates of the sample point of corners. If not all four corners are detected, it raises an exception.

The majority of this file is the function "get_corners", which takes in one parameter "img", the camera captured-image. Inside the function, we first find the height and width of the given image (to help determine the coordinates of corners later): height, width = img.shape[0], img.shape[1]

We decided to use red corners, because we found red has the most contrast with the rest of the image and the background environment. Because the picture pixels could be represented by the RGB value, the code used to search for the red corner used a threshold that would resemble the red on the image the most. By trial and error, We picked the threshold for the red element to be between 100-220. While restricting the blue and green elements of colorspace to be 50 less than the red element. The range is intended to be large because we assume that the camera frame would have no other similar red besides the four corners. The range must be able to detect red at different brightnesses, which could have different RGB values.

```
Y,X = np.where((r_query_low <= img[:,:,r]) &
          (img[:,:,r] <= r_query_high) &
          (img[:,:,g] < img[:,:,r]-50) &
          (img[:,:,b] < img[:,:,r]-50))
```

The above code retrieves the coordinates of all the points that have the RGB range fit our requirement we set and X, Y coordinates saves these coordinates into a two-dimensional array.

We find the max and min of X, and Y values in all the coordinates (most extreme corners), and use the below code to generate the average of the min and max values of X and Y separately.

```
x_thresh = (max_x - min_x)/2
y_thresh = (max_y - min_y)/2
```

Because we can't get a visual of the spread of the red pixels, the purpose of the threshold helps us determine the number of corners we actually find. The x and y threshold helps to differentiate corners. For example, multiple red pixels may fall within the same corner. If either no red pixels are found, or the pixels clearly contain less than 3 points, the function will throw the exception. If we don't have any of the problems, we will generate the coordinates of the corners using the code below.

for p in y_points: // if x or y differ substantially from every other point's x and y Diff=True for p2 in fps: if (abs(p2[0] - p[0]) < x_thresh and abs(p2[1] - p[1]) < y_thresh): Diff=False break if Diff: print(p) print(type(p)) fps.append(p)

The nested for loop adds the coordinates of corners to “fps,” the “four points,” which is the final coordinate list used to crop the image to only have the signs. This code takes each red pixel coordinate we have and adds the coordinate if there are no points in the fps that have a similar coordinate would reassemble the same corners.

Based on the nature of how the red pixels are detected, If all four points are found, they will be sorted in a Z manner

1———-2

| |

3———-4

If only three points are found in the below situations, let X be the missing point then we use basic geometry calculation to find the coordinate of X.

X———-2 1———-X 1———-2 1———-2

| | | | | | | |

| | | | | | | |

3———-4 3———-4 X———-4 3———-X

Finally, either we found and calculated all corners of the image, which we will return fps for cropping the image; or we couldn’t find enough corners, and raise exceptions.

4.1.3 image_distance.py

headshot_ratio takes in the x and y coordinates for all 4 corners of the image, as well as the image (the headshot including the background) height and width. The function returns the ratio between the area of the headshot image and the area of the image.

4.1.4 image_warp.py

image_warp is the main Python file for handling the warp and crop function in the image processing stage. Warp takes a camera-captured image, finds the coordinates of the four corners of the intended rectangular sign, and estimates a projective transformation to warp the image, this means that the corners of the image are stretched to fit the size and shape of the desired rectangle. Finally, it returns the warped image along with the projection points. The crop takes in the already-wrapped image and projection points and returns a new image whose corners are the corner of the sign or face.

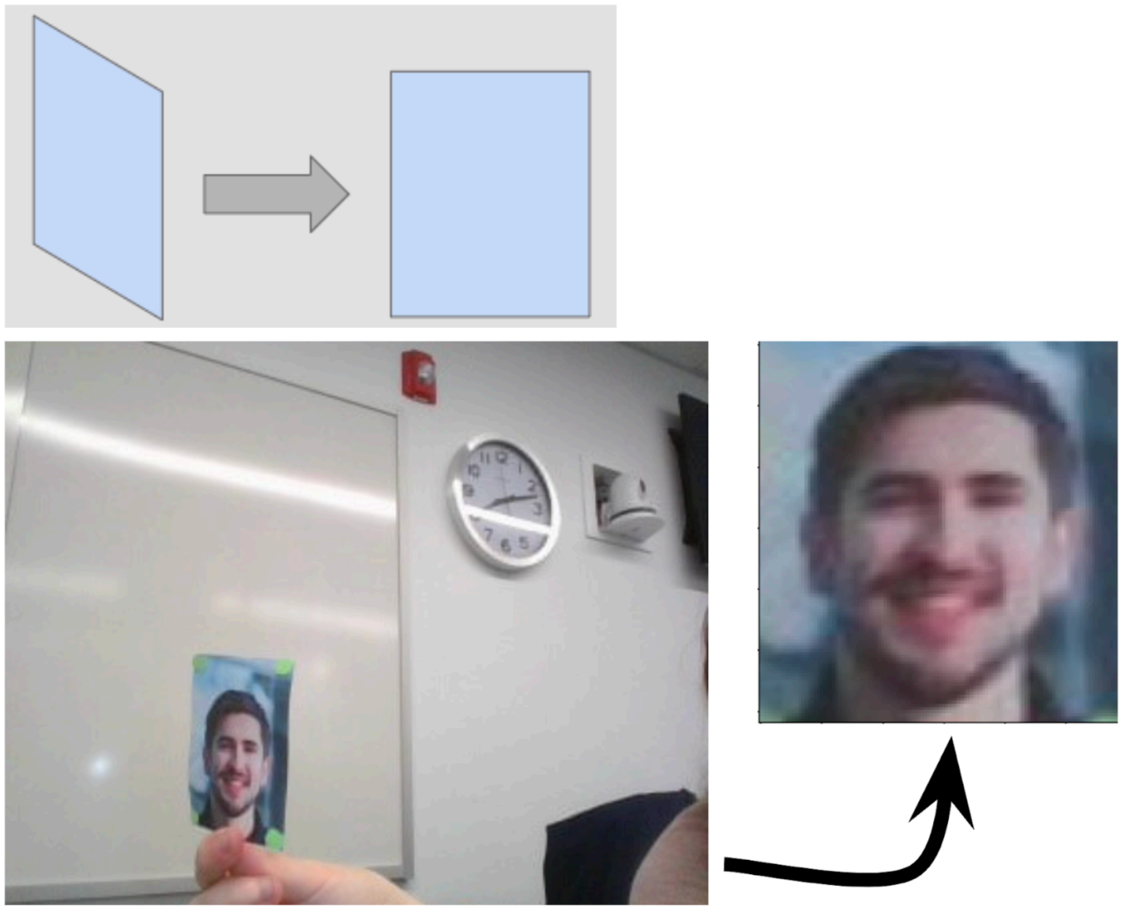


Figure 3: An example of how an image is warped and cropped

Image warp and crop have a critical role in ensuring the image we feed our training set and live data are standardized. We are creating a set of constraints for all the images, and only the images which satisfy our requirements can be interpreted. The warp function uses the red corners of the printed picture in the image and projects it into a rectangle. This creates consistency and therefore helps us make accurate predictions. The crop function deletes all the excess image space beside the intended object. This improves the time complexity of our algorithm because we no longer need to process background pixels.

```
points_of_interest = np.array(corner_detection.get_corners(sign))
```

The projection variable is then defined as an array of four points that correspond to the corners of the output image after warping. These points are set to be positioned in the center of the input image, with a slightly wider height than width, defined by the ratio of height to width. The transform.estimate_transform() estimates the projective transformation between the detected corners and the projection points. The transform.warp() warps the image using the estimated transformation we created from above. Then it returned the warped image and the projection points.

```
ratio = 1.2
projection = np.array([[(width/2) - (2.5*height/12), height/4],
                       [(width/2) + (2.5*height/12), height/4],
                       [(width/2) - (2.5*height/12), 3*height/4],
                       [(width/2) + (2.5*height/12), 3*height/4]]).astype(int)

tform = transform.estimate_transform('projective', points_of_interest, projection)
```

```
tf_img_warp = transform.warp(sign, tform.inverse, mode = 'symmetric')
```

The crop function takes the warped image and projection points as input and crops the image based on the positions of the projection points, which are now the corners of the rectangular signs. Finally, it returns the corresponding cropped section of the warped image.

The intended use of this module is to call the warp function first, and using the return value we get to call the crop function. The resulting image would be fully processed – ready to hand off to the CNN model.

4.2 JoyStick+MotorModule

4.2.1 JoyStickModule.py

Using the pygame library, JoyStickModule.py stores the inputs from a PS4 bluetooth controller in a dictionary, updating the input values in real time. The file connects to the controller and declares a global dictionary, buttons, which represents the current state of the buttons on the controller. The keys of the dictionary are referring to the buttons on the controller, such as "x", "o", "t" and "s". The values of the dictionary are integers that represent if the button is pressed (1) or not pressed (0). The default value is 0 because the buttons are not currently being pressed.

There is a separate list, axiss, which stores the values of six axes on the controller: the X and Y axis for the left and right controllers and the triggers. The default values of the indexes in the list are 0.0. The values range from -1 to 1. axiss[0] represents the X-axis of the left analog stick. axiss[1] represents the Y-axis of the left analog stick. axiss[2] represents the "throttle" axis, which is typically a slider or dial on the controller. axiss[3] represents the X-axis of the right analog stick. axiss[4] represents the Y-axis of the right analog stick. axiss[5] represents the "rudder" axis, which is typically a twisting motion on the controller.

The file also defines a getJS() method, which handles the controller "events," or any instance of new inputs on the controller. When the joysticks are moved, the module updates the corresponding entries in the axis list, then stores them in buttons. If a button is pressed or released, the module iterates through the button entries in the dictionary to see which button was pressed or released, then updates the value in the dictionary.

The file will return the dictionary of all the buttons with the default parameter; if we pass in the name of a specific button or axis, then the corresponding value for that button or axis from the dictionary is returned.

4.2.2 MotorModule.py

Using the built-in Raspberry Pi General-Purpose Input-Output (RPi.GPIO) library, MotorModule establishes a connection with the Raspberry Pi motors at set pin numbers, storing the two motors in one class, Motor. Each Motor object has two methods: move() and stop().

Lance has differential drive, which is a drive system that uses independent wheels on opposite sides of the robot. This system works by varying the relative speeds and directions of the wheels to achieve different movements. If both wheels are moving at the same speed, the robot moves backward or forwards. If the wheels are rotating at different speeds, the robot turns.

To initialize the Motor object, we pass in two pin numbers, In1 and In2 that correspond to the two motor pins on the Pi. GPIO.setmode(GPIO.BOARD) first sets the numbering scheme for the module to use the physical pin numbers on the Pi, then GPIO.setup(In1, GPIO.OUT) and GPIO.setup(In2, GPIO.OUT) establishes those two pins as "out," or output pins.

The program instantiates mySpeed to 0 and defines two pulse-width modulation (PWM) objects (this allows us to power them using duty cycles), pwmright and pwmleft, for the right left motors respectively: self.pwmright = GPIO.PWM(In1, 50) and self.pwmright = GPIO.PWM(In2, 50), where each pin receives a signal of 50 Hz, the standard recommended value for DC motors. self.pwmright.start(0) starts the PWM output at a “duty cycle” of 0, where the motors receive 0 percent of the 50 Hz input and are turned off. The ESCs on Lance are programmed to not move at around 90mV, and a duty cycle of 7.25 percent outputs 90mV.

The move() method takes three parameters: speed, (ranges from -100 to 100) which determines if Lance will move forward or backward and how fast, turn, the turn angle where -1 is left and 1 is right, and t, the duration of the movement command in seconds.

The module determines the leftSpeed and rightSpeed based on those parameters. If speed is 0, then it sets both to 0. Otherwise, leftSpeed = speed+(turn/2)+7.25 and rightSpeed = speed-(turn/2)+7.25.(turn/2) makes the turning input less “sensitive”. 7.25 is the default value for “neutral”, we were able to get this value from trial and error while using the multimeter to measure the output voltage from the Raspberry Pi, so we either add or subtract from that value.

Once the speeds for each motor have been calculated, self.pwmleft.ChangeDutyCycle(abs(leftSpeed)) and self.pwmright.ChangeDutyCycle(abs(rightSpeed)) change the duty cycles for each motor to move appropriately.

Finally, the stop() method takes one parameter, t, the time to sleep after stopping in seconds, and sets the duty cycle for each motor to be 7.25 and mySpeed to 0.

4.3 Step1-Data-Collection

4.3.1 DataCollected

DataCollected contains all the folders of the SebastianLeft and SebastianRight signs, with brightnesses ranging from 0.5 to 1.75.

4.3.2 DataCollectionMain.py

This file is used to collect data for facial recognition. At the top of the file, we imported several modules. We imported the WebcamModule, DataCollectionModule, JoyStickModule and MotorModule. These modules control the webcam, data collection, joystick and motors. A max throttle value is set to a constant value of 0.55. We picked this number after extensive trial and error. A motor object is initialized with values 11 and 8, which correspond to the left and right motors on Lance. 11 and 8 are the pins values.

Inside the infinite loop, the program will read the joystick. In addition to this, the program will calculate the steering angle, throttle and backwards speed based on the input joystick values. The RPi will start capturing images from the camera when we press the share button. The saveData function from DataCollectionModule.py is called. Once the program is killed, the RPi will stop capturing images and save the log file. The saveLog function from DataCollectionModule.py is called.

4.3.3 DataCollectionModule.py

This module saves images and a log file. The images are saved in a folder.

In saveData, the inputs are the image and the associated steering angle. For facial recognition purposes, we don’t use steering angle. We only use steering angle for road tracking. In the try statement, the program will crop and warp the image. The except statement will only run if either cropping or warping fails, and the function will end when this happens. If the program continues to run (i.e. cropping and warping are successful), the cropped and warped image will be multiplied by 255, which will convert the image to an 8-bit unsigned integer (uint8) data type. Then the image is resized to a size of 480x240 pixels. Lastly, the file is saved with a filename that includes the img_index and the image label. img_index refers to the ith photo in the folder. For example, in SebastianRight_81, 81 is the img_index.

In saveLog, saves the log file to the Raspberry Pi.

4.3.4 WebcamModule.py

This code captures video frames from a camera using OpenCV's VideoCapture object, resizes the frames to a specified size, and displays them in a window.

The getImg() function is called in a loop to repeatedly capture frames and resize them to the specified size. If the display argument is True, the resulting image is displayed in a window using cv2.resize. The function returns the resized image. The loop continues indefinitely until the user kills the program or the window is closed. In each iteration of the loop, getImg(True) is called to capture a frame and display it in the window.

4.4 Step2-Training

4.4.1 config.py

config.py stores all the user file paths and variables that are used all throughout the Step2-Training.

Changing file paths and editing the config.py file every meeting took up a lot of time. We needed to find a way to make the config.py file easier to read so that everyone can use train.py locally and not take up a lot of time changing file path names.

config.py was made shorter and more efficient to change depending on the person coding on their local laptops by generalizing file paths. This means that there is less to comment and uncomment in the file.

4.4.2 model.py

model.py uses PyTorch Library to implement a Convolutional Neural Network (CNN) for the sign recognition part of our project. Because we were not familiar with how to create a convolutional neural network, we referred to this Youtube video (`https://www.youtube.com/watch?v=LgFNRIFxuUo&ab_channel=SungKim`), which explains in more detail why the arguments in the function are the values they are.

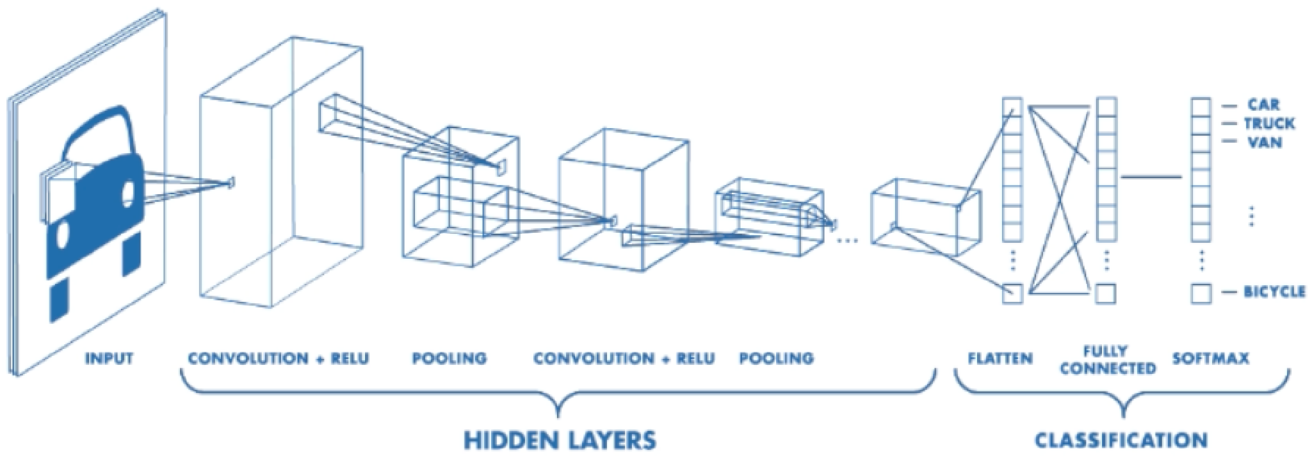


Figure 4: CNN Model

```
self.conv1 = torch.nn.Conv2d(3, 10, kernel_size=5)
```

The above code generates the first convolutional layer. It calls torch.nn.Conv2d, a Pytorch constructor that creates a convolutional layer that takes in a tensor with 3 input channels (RGB), and outputs a tensor with 10 channels, with a kernel size of 5. We chose these values through trial and error.

The kernel size in this case is referring to the dimension of the filter that's applied to our input image. This size would define the filter's height and width, in turn determining the receptive field through some linear algebra operation.

```
self.conv2 = torch.nn.Conv2d(10, 20, kernel_size=5)
```

The second convolutional layer takes in 10 input channels and outputs 20 channels, with a kernel size of 5. We chose 10 as the input channel size because it matches up with the output size of the previous layer and 20 as the output channel size for no particular reason. Here, we used the same kernel size, because we want to keep the filter size constant.

```
self.max_pool = torch.nn.MaxPool2d(2)
```

maxPool2d is a function that generates a max pooling layer. It downsampled the 2D input data. The max pooling layers have a kernel size of 2, which slides a 2x2 window onto the input image and takes the maximum value within each window. This operation downscales the number of interest pixels by a factor of kernel size = 2. This is an important layer because it helps to make the model more efficient.

```
self.flatten_conv = torch.nn.Linear(74420, 2)
```

Finally, the linear layer takes in the max_pooling output after the second convolutional layer, which has a size of 74420. The terminal helps us determine the dimensions of the tensor size. This layer is a flattened layer, meaning that the dimensionality of the output data is reduced. This outputs 2 options (SebastianLeft and SebastianRight). This represents the final layer of the convolutional neural network whose output is our desired result.

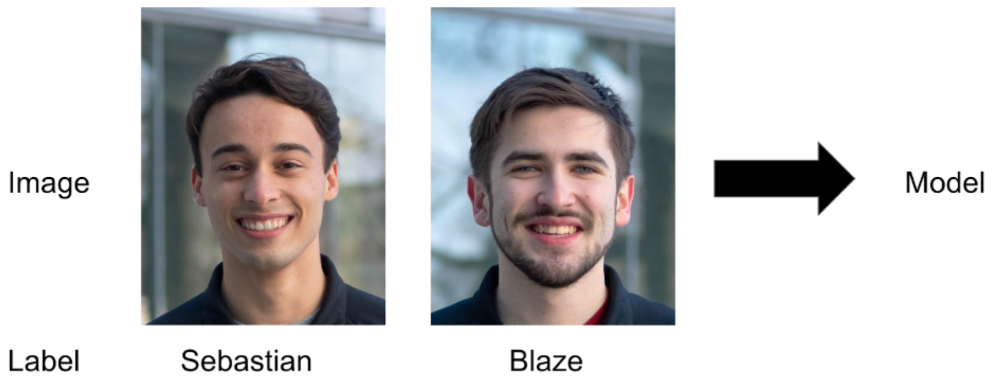


Figure 5: Data Fed Into the Model

The output will have two weights, each one representing one of the options, and the goal of the CNN is to take the input RGB channel and estimate these weights. The weights, ranging from 0 to 1, represent the probability of the input image of one of the two options. When we are predicting in real-time, the robot will use the weights to determine what it thinks the image is. This will then modify the motor's speeds to achieve different turning.

The forward method defines the forward pass through the network. This function specifically handles a neural network that performs image classification, which is a computer vision task that involves assigning labels to an image based on what the image contains. The function in a nutshell is a micro process in how a model is generated. The input x is the batch of images the network needs to process. in_size is the number of images in the batch. The input is first passed through the first convolutional layer, which applies the first set of filters to the image as described above. It extracts the features of the image and outputs a feature map, which is a mapping of the features that were found from the input image. It then gets max pooled. Then it is passed through a ReLU activation. The output is then passed through the second convolutional layer, max pooled again, and passed through another ReLU activation. The output is then flattened and passed through the linear layer with a softmax activation, and the log softmax output is returned.

A linear layer is a type of neural network layer that performs a linear transformation on the input data. It applies a set of learnable weights and biases to the input, producing an output that is a linear combination of the input features.

A softmax activation is a type of activation function that is commonly used in the output layer of a neural network for multi-class classification problems. It maps the output of the linear layer to a probability distribution over the possible output classes.

We found that increasing the number of labels decreased our model accuracy. We attempted to resolve this by adding another layer to our model, but we struggled to understand the difference between a linear layer and a convolutional layer.

We had been working on making an accurate model for the SebastianLeft and SebastianRight signs and now wanted to add a Sebastian label to the model. We thought that adding a new convolutional layer would help the accuracy of the now 3 labeled model since the model can now learn more complex patterns in the input images.

We assumed that changing self.flatten_conv = torch.nn.Linear(74420, 2) to self.flatten_conv1 = torch.nn.Linear(74420, 10) and self.flatten_conv2 = torch.nn.Linear(10, 2) would add a new convolutional layer into the machine learning model.

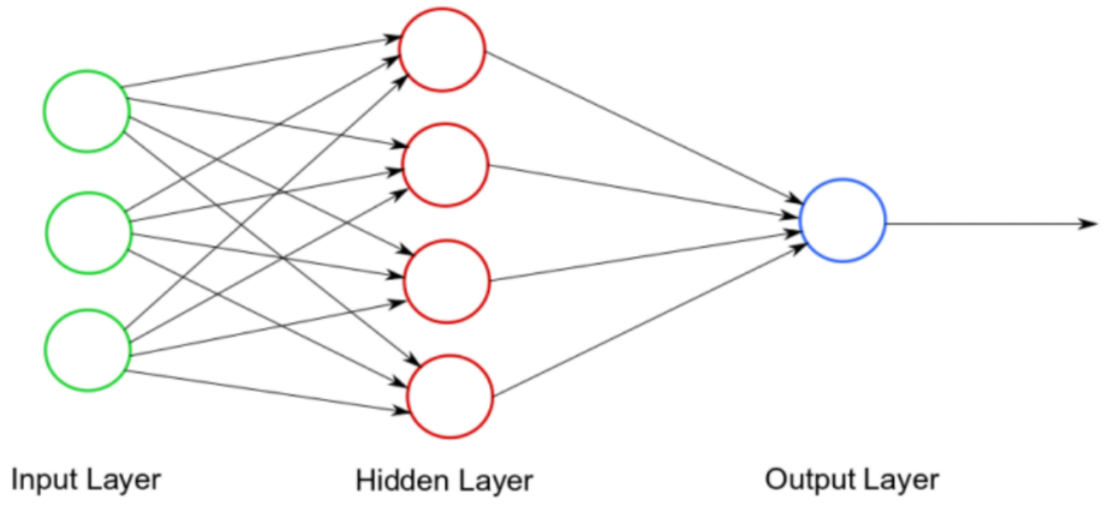


Figure 6: CNN Model Layers

This backfired on us. Even though the 3 labeled models could accurately predict the 3 different signs/faces, the model would take more than 45 seconds to make a single prediction when we tested it live. In addition to this, the program would shut down, most likely due to the large amounts of data the model is working with. Adding one more layer exponentially increases the number of processes the model has to work with.

We tried to push the model onto GitHub, but since the model was such a large file now (500+ MB), it went over the 100 MB GitHub limit, which meant that we had to use GitLargeFileStorage. We found that we had to download GitLargeFileStorage for both our local laptops and on the Raspberry Pi. We had an issue with commits and had to undo all our commits before being able to push the 500+ MB file onto GitHub.

4.4.3 Dataset.py

Dataset is a way for cnn-style machine learning to check the training progress at every epoch of the training. Dataset is a map-style dataset that PyTorch uses to represent a map from indices to data samples. The file has a constructor and two functions.

The constructor takes a data set, which is an image in this case, and the label corresponding to that image, which would either be SebastianLeft or SebastianRight. It also takes in a set of image transformations that would be applied to the images.

The first function "len" returns the number of images in the dataset. The second function "get_item" fetches the data sample from the indicated index. It will then transform the image, which will consist of image crop/warp, and color change. This will make the image become suitable for the training.

4.4.4 FacialRecogVisualDemo.py

This file is essentially the exact same as DataCollectionMain.py, except this file includes the predicting sign in real-time feature.

4.4.5 predict.py

We define a that will take in an image as an input, and output the associated name of the person on the image. Next, we define transformations, and there are three kinds of transformations that are applied to the image.

1. transforms.ToPILImage() converts a numpy array to a PIL (Python Imaging Library) image. The input has dimensions (C, H, W), where C is the number of channels. In our case it is 3 for RGB values, H is the height of the image and W is the width of the image.
2. transforms.Resize((config.INPUT_IMAGE_HEIGHT,config.INPUT_IMAGE_WIDTH)) resizes the PIL image to the specified height and width dimensions. The specified height and width dimensions are 256 pixels, and this can be found in the config.py file.
3. transforms.ToTensor() converts the PIL image to a PyTorch tensor. A tensor is a multi-dimensional array that is used to encode the input and output data of deep learning models. The PyTorch tensor will be used as an input for the machine learning model.

NOTE: This is the exact same code as in the train.py file. The transformations are defined in the same way.

img = transforms(img) is the input image, and this image is transformed with the specifications from above. The CNN model, which we get from the config.py file, is loaded using the torch.load function. The model is also set to evaluation mode using the eval() method. evaluation mode refers to the state of the network during inference or testing. In the evaluation mode, the model is used to make predictions on new data and no updates are made to the model's weights or biases.

Before passing the image to the model for prediction, the image is preprocessed by expanding its dimensions to add a batch dimension using np.expand_dims and converting it to a PyTorch tensor using torch.from_numpy.

After the image is finished with the preprocessing, it is then passed through the CNN model using cnn(img), and this is set to the output variable. output will have predicted probabilities in an array that are associated with the respective labels. The first number corresponds to SebastianLeft and the second number corresponds to SebastianRight. A higher number means that the model has a higher confidence that the given input image is of the associated person.

The torch.argmax function is used to find the label with the highest probability, and this is set to the predicted label of the image.

Outside of the function, we use predict.py to locally test how accurate the model is by going through all the photos for all kinds of brightnesses and seeing the accuracy when testing live at the end of the program. We iterate through each image string path in the image string list, and go through every photo in each image string path. cv2.imread(e), reads the image. numCorrectSebastianLeft or numCorrectSebastianRight will increment by 1 if the model accurately predicts the correct label for the current image.

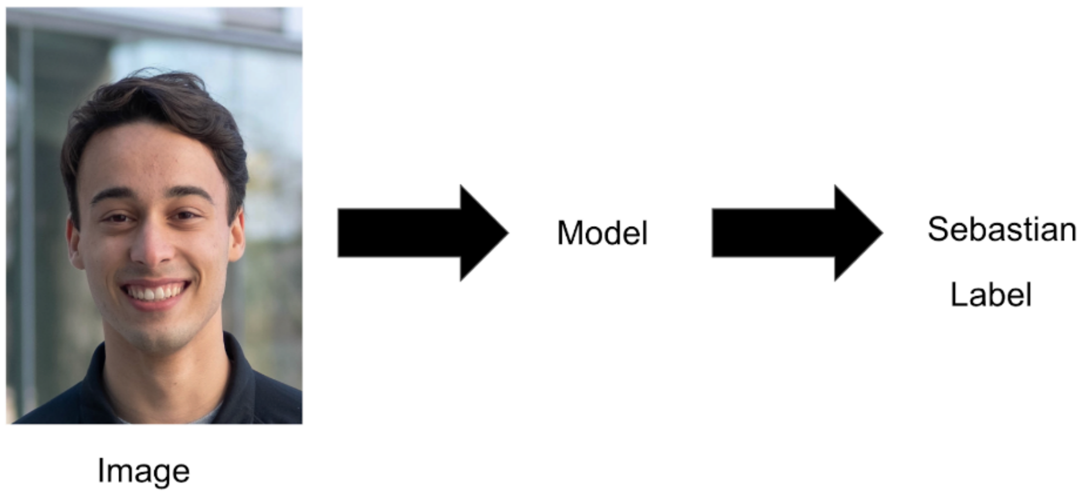


Figure 7: Process of the Model

4.4.6 train.py

train.py is used to train our facial recognition model based on a set of labels and images. We have trained our model to be able to detect the difference between the Sebastian left and right signs.

One problem we had when training models was that it was inconsistent. Training two models with the exact same set of images, batch size and epoch would result in slightly different models. We resolved this problem with torch.manual_seed(7). This ensures that when we run train.py the parameter weights remain the same. By setting the initialized weights to a constant, the model trained on a particular set of images, batch size and epoch will be the same as another model trained with the same settings at a different time.

Next we define transformations, and there are three kinds of transformations that are applied to the image.

1. transforms.ToPILImage()

2. transforms.Resize((config.INPUT_IMAGE_HEIGHT,config.INPUT_IMAGE_WIDTH))
3. transforms.ToTensor(). See predict.py for more information about this.

name2num = ”SebastianLeft”: 0, ”SebastianRight”: 1 is a dictionary which contains the labels for the Sebastian left and right signs. image_path_list = config.LST_TRAIN_IMG_DIR puts all training images of all brightnesses from both the Sebastian left and right folders into a single list. This list is taken from the config file.

The purpose of the nested for loop is to load in data, and accomplish that by accessing the individual file paths from the list of file paths, and then accessing individual photos inside of the folders we got from the file paths.

The outer for loop goes through every image file path in the image path list array. The inner for loop goes through the first 101 images in the image file path (for example, SebastianLeft0.75). train_img loads an image from the specified file path. Then the image is appended to the train_image_lst array, and this is done for all the 101 images in the specified image file path. path_segmented = re.split(r”[_/]”, e) splits the current image file path using the “_” and “.” characters into indexes in an array.

Example: ”/Users/richmjin/Desktop/facial _recog/lib/dataset/input _img/1 _Blaze.jpg” –> [... img/1, Blaze, jpg]

train_label_lst.append(name2num[path_segmented[-2]]) gets the second to last index from path_segmented, which is what we identify as the sign (SebastianLeft or SebastianRight). Then the sign is added to the list of labels. We will be using train_label_lst in the second double for loop.

train_DS creates a training data set and takes in three inputs: train_image_lst, which is an array containing all images in all the image file paths. train_label_lst, which is an array containing all the labels of the images. transforms, which is what defined the transformations earlier in the code.

train_loader loads the training data set train_DS in multiple batches during the training of the deep learning model.

shuffle=True shuffles the training data around at the beginning of each epoch. By shuffling the data at the beginning of each epoch, we ensure that the model sees the data in a random order each time and therefore has to learn the underlying patterns in the data instead of the order of the data. This can help improve the model’s ability to generalize to new data and prevent overfitting.

batch_size=config.BATCH_SIZE specifies the batch size for each iteration of the training process. The specific batch size value is set in the config.py file. When training, batch size is set to usually 10.

pin_memory=config.PIN_MEMORY is set to false in the config.py file. This makes a cache for all the data that gets passed into the data loader, so the program should run faster. Since the value is set to false, we are not using this.

num_workers=0 sets the number of worker processes that are used to load the data. This is set to 0, which means that the data will be loaded in the main process.

A worker process is a separate process that can perform tasks independently of the main process. In the context of PyTorch’s DataLoader, the worker processes are used to load the data in parallel with the main process. By setting the num_workers argument to a value greater than 0, the DataLoader will use multiple worker processes to load the data in parallel, which can speed up the data loading process. However, if num_workers is set to 0, the data loading will be performed in the main process, which can be slower but may be necessary in some cases where parallelism is not desirable or when there are compatibility issues with multiprocessing.

Next, cnn = model.CNN() creates an instance of the convolutional neural network (CNN) model. See model.py for more information about this. This is needed when training the model. cnn.train() starts the training process in train.py.

opt = torch.optim.Adam(params=cnn.parameters(), lr=config.INIT_LR) initializes an Adam optimizer object using the Adam optimization algorithms. The first parameter of the function refers to the parameters of the convolutional neural network. The Lr in the second parameter is the learning rate. This is set to a value of 0.001 in the config file. 0.001 is a traditional default value of the learning rate.

loss_fn = torch.nn.NLLLoss() initializes a negative log-likelihood loss object, which is used for classification, (in this case we are trying to train the model to be able to tell apart the SebastianLeft and SebastianRight signs.

opt = torch.optim.Adam(params=cnn.parameters(), lr=config.INIT_LR) and loss_fn = torch.nn.NLLLoss() work to minimize the loss function. A lower loss function means that the CNN model is able to better distinguish between different labels.

In the second double for loop, the outer loop runs the inner loop by the number of epochs, which we get from the config.py file. The inner loop goes through all the images and labels in train_loader config.py. The double for loop will use the model to make predictions of the provided image, and then compute the loss.

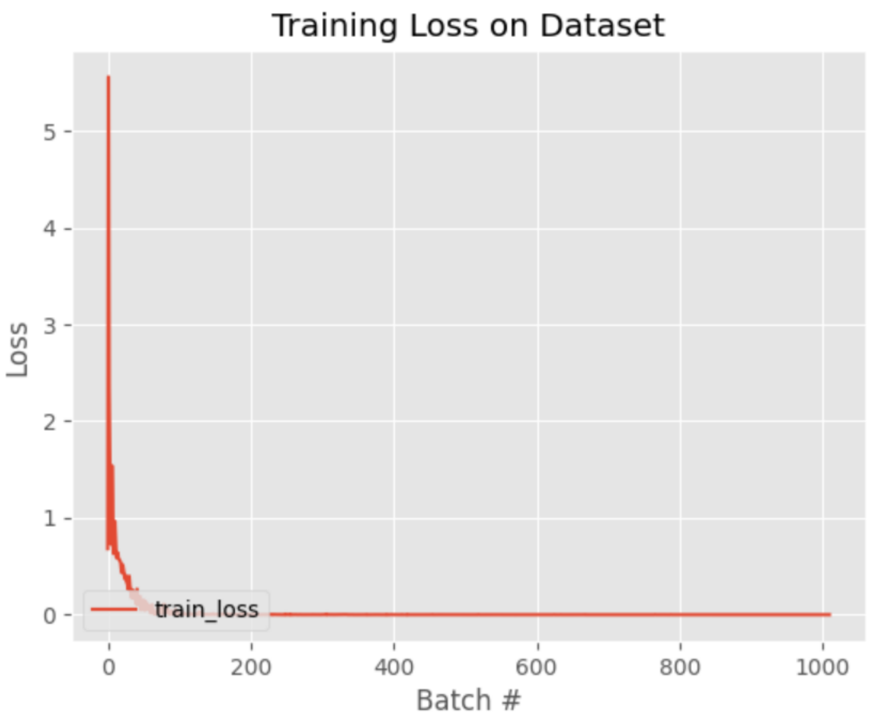
In this code snippet, the imgs variable refers to a batch of input images. The labels variable refers to the corresponding ground truth labels for each of the input images in the batch.

opt.zero_grad(), loss.backward(), and opt.step() update the CNN weights to minimize the loss function.

The loss is appended to the train_losses array. Then after every iteration of the outer for loop, the epoch and loss is printed. opt.zero_grad() cleans up the past gradient of those parameters associated with opt. loss.backward() calculates the new gradient of those parameters. opt.step() makes changes to the values of those parameters associated with opt in the neural network on the purpose of minimizing the loss.

After the outer for loop, the total time that it takes for the model to be trained is printed out, and then the training loss is plotted onto a graph, with the batch size on the x axis and loss on the y axis. As the batch size increases, the loss decreases significantly.

Finally, this graph is saved in PLOT_PATH in the config.py file. Lastly, the newly created model is saved in MODEL_PATH, which is also in the config.py file.



4.4.7 WebcamModuleLabeled.py

The purpose of this file is to get an image from the Raspberry Pi Camera using the OpenCV package. This is a more complex version of WebcamModule.py.

cap = cv2.VideoCapture(0) creates a VideoCapture object. This object is used to read the video frames from the Raspberry Pi Camera. The argument, 0, indicates that the default camera (the Raspberry Pi Camera) should be used.

cap.set(cv2.CAP_PROP_FPS,10) sets the number of frames per second (FPS) to 10. We chose this through trial and error

The following variables are declared and assigned values:

1. counter = 0 : This is used to count how many frames have passed. This will be explained in more detail below.
2. output = "None" : This is the label that will be shown on the screen. This is set to "None" for now.

The function getImg() is called in a loop an infinite number of times (until the program ends) to capture the frames from the Raspberry Pi Camera and resizes them to 480 x 240 pixels. The function will perform the image processing operations (cropping and warping) every 10 frames using the counter variable.

Inside the try statement, the image is warped and then cropped. The resulting image is then multiplied by 255. What multiplying the image by 255 does is scale the pixel values from a normalized range of [0,1] to [0, 255]. We are re-scaling the pixel values to the appropriate range for further image processing.

The image is then converted to an 8-bit unsigned integer format using astype(np.uint8). Then using img = cv2.resize(img,(size[0],size[1])), the cropped image is resized to 480 x 240 pixels. We chose this size because it was the largest size possible without making the camera lagging.

The output variable is set to predict.func(img), which uses a trained neural network to predict a label for the input image. If an error occurs during this step, such as the image failing to crop or warp, output is set to the string ”None”. The predicted label is then overlaid on the image in red font using the cv2.putText function.

Counter is reset to 0 so that the number of frames that have passed can be counted to 10 again. The resulting image is displayed in a window using cv2.putText. The loop continues until then or the window is closed or when the user ends the program.

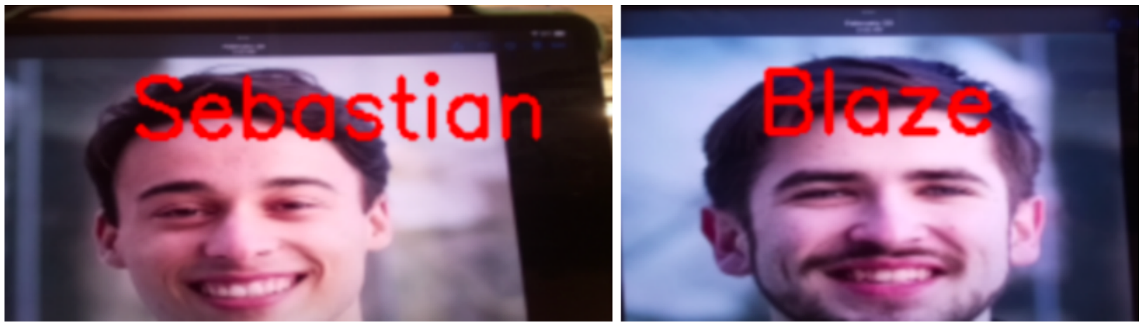


Figure 8: Working Facial Recognition Model Output

5 ROAD TRACKING

5.1 Road Tracking Problems

5.1.1 Manny’s physical restrictions

In the middle of the semester, because of Manny’s physical restrictions, firmware decided to pivot to using Lance. We found that Manny was hard to drive and unreliable. For example, Manny would drift left or right even when we didn’t touch the joystick. We struggled to drive Manny along a road (straight or curved) and gather quality data for our road tracking model. Even when we were able to train a road tracking model, we found it difficult to thoroughly test the model’s accuracy. When Manny steered off course we needed to consider two causes of the error: 1) the model is incorrectly predicting the steering angle or 2) the model is correctly predicting the steering angle, but Manny cannot drive accordingly.

5.1.2 Model is built using tensorflow

The sign recognition model uses pytorch while the road tracking model uses tensorflow. We started working on sign recognition before working on road tracking, so we initially struggled to understand the output of the tensorflow model. For example, the facial recognition model outputs confidence weights corresponding to each one of our labels (“SebastianRight”, “SebastianLeft”). By contrast, the road tracking model outputs a singular steering value. Even if we trained the road tracking model on binary values (0s and 1s), the model would predict intermediate values (ex. 0.73 or 0.27). In some cases, the model predicted steering angles greater than 1 or less than 0. We viewed the training data as buckets, and we wondered how the model could generate values outside of these buckets. After some research we learned that sign recognition is a classification task while road tracking is a regression task. Classification tasks work to classify several distinct values, while regression tasks work to determine continuous values. The difference in output between the sign recognition and road tracking models is a result of the type of task that each model is performing rather than the differences in libraries (pytorch vs. tensorflow).

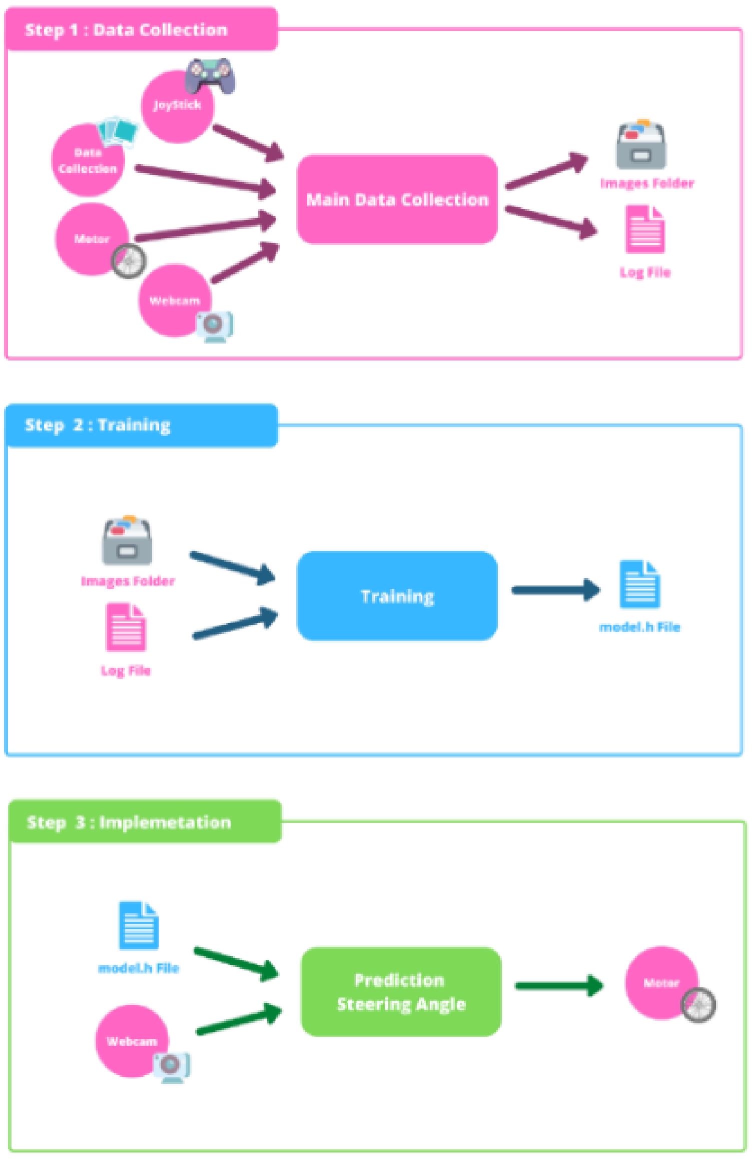


Figure 9: Whole Road Tracking Data Process

5.1.3 GitHub issues

Collaborating on GitHub has been difficult at times. We have encountered various errors such as rebase errors when members have tried to push to or pull from GitHub. These errors can be cumbersome to resolve and the time could be better spent coding or debugging the project. We believe some of the errors come from the way that we have been removing files from our repository. Going forward, we will use the "git rm" command rather than simply deleting them from local file systems.

5.1.4 GitHub organization

As the project became more and more complex, we created more and more branches to our GitHub repository. Over time, the main branch lost its purpose as the central source of code. Some branches became the new home for the most up-to-date code and that code was never transferred to the main repository. This could be problematic in that some code revision in older branches could have been lost.

5.1.5 Debugging code

We obtained a good portion of our road tracking starter code from a guide on CV Zone. When our code didn't work as expected we found it hard to debug code that we didn't necessarily write. Although our source seemed reliable, we needed to determine whether the error resulted from code we added or the original. As necessary we conducted research to understand functions and libraries used in the source code.

5.2 Step1-Data-Collection

5.2.1 DataCollected

DataCollected contains folders of images we collected through DataCollectionMain.py. Additionally, DataCollected contains csv files where each line corresponds to a file path and its associated steering angle. File paths are written in terms of the raspberry pi's file system. Lastly, DataCollected contains a readme file which outlines the contents of each image folder and when the data was collected.

5.2.2 DataCollectionMain.py

DataCollectionMain is used for collecting road data from the camera, getting the steering angle, then storing those together using the DataCollectionModule in the DataCollected directory for future training.

It first establishes the maxThrottle, the maximum speed at which Manny will move, and a Motor object to store the two Raspberry Pi motors.

This module establishes a connection to the controller to determine many important inputs: the steering angle inputs from the left analog stick, whether Manny will move forwards or backwards from the "O" and "X" buttons respectively, and when to turn on the camera and start collecting data from the Pi camera using the share button.

Once the share button is pressed again, the module will stop collecting data and save all of the data in an image folder and a log file, so that all the images and the steering angles are stored.

5.2.3 DataCollectionModule.py

DataCollectionModule takes images and steering and stores them together in the DataCollected directory. It defines two methods: saveData() and saveLog().

First, it creates a new folder in DataCollected based on the number of pre-existing folders, and appends that to the default file name. For instance, if the last folder was IMG6, the module would create IMG7 to store new data.

saveData() takes the image passed in from DataCollectionMain and the corresponding steering angle and adds them to two lists, imgList and steeringList. It also takes the timestamp to generate unique file names for the images, which are then stored in the DataCollected directory.

saveLog() saves the images and steering angles from imgList and steeringList in a csv file corresponding to the image folder with the same number (ex. log_13.csv and the IMG13 directory). It defines a dictionary, rawData, that has keys 'Image' and 'Total Images' which map to imgList and steeringList respectively. That dictionary is turned into a Pandas DataFrame, which functions like a table. It is then converted into a csv file and saved in DataCollected.

5.2.4 Other Python Files

JoystickModule.py, MotorModule.py, and WebcamModule.py are the same files as the Sign Recognition part of the project. This repetition makes the project become a little larger storage-wise, and creates some potential synchronization problems between the copy of the modules in the different parts of the project.

It, however, would make the task of using these modules for moving the robot in road-tracking much easier, since we would be referring to files that are in the same folder. The collision issues will rarely occur as there is no reason to change these modules unless we change the motor system, webcam, or the joystick for our robot.

5.3 Step2-Training

5.4 train.py

Train.py trains a new road tracking model. The name of the new model is specified in line 56: model.save('name.h5'). The training data for the model is determined by the path declared in line 12 and the file utils.py. Data is housed in folders and csv files titled "IMG" and log_.csv respectively (all within the DataCollected directory). Line 29 in utils.py specifies which folders/csv files to use in training. For example, for x in [0, 2, 3]: will use data in files log_0.csv, log_2.csv, and log_3.csv.

Another notable function in utils.py is preProcess(img) which recolors and resizes training images for faster processing. For example, we convert images from a RGB color space to YUV (where Y=luma, U=blue projection and V=red projection). YUV is fairly typical in image processing because it has a smaller bandwidth compared to RGB. Train.py uses these processed images to train the model.

Batch and epoch size are specified on lines 50 and 51 of train.py respectively. Batch size is the number of samples processed before the model is updated, while epoch size is the number of complete passes through the training dataset. In line with common machine learning practices, we typically set our batch size to 100 and epoch size to 10.

After the model is finished training, the program will display a graph of the loss over epochs. Loss generally quantifies how much the model is changing with each epoch. A lower loss means that the model is stabilizing. Note that a stabilized model isn't necessarily an accurate model. The graph should look something like a negative sloping exponential function, and a good model should converge towards a loss of 0.1 or less.

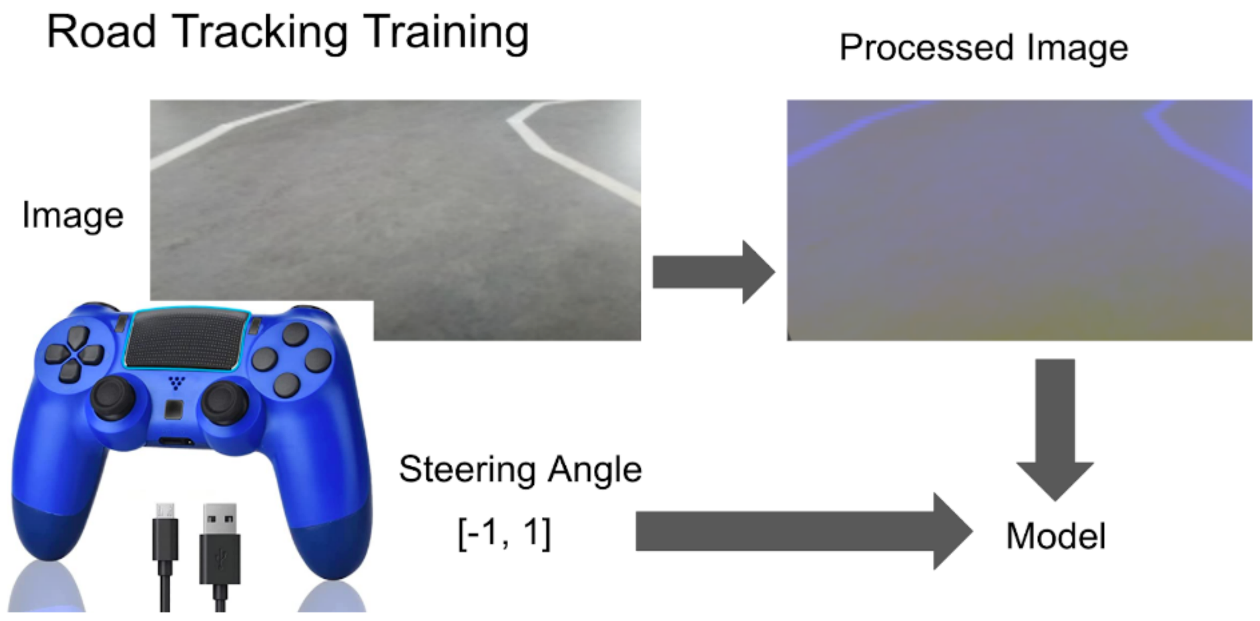


Figure 10: Input into the Road Tracking Model

5.5 rt_predict.py

Predict.py allows us to locally test the accuracy of a road tracking model. The file is split into 2 functions: singular_test and multi_test. singular_test accepts a model and image path as input and returns the predicted steering angle. For example, for a right turn, we should expect the model to return a positive value (ideally between 0.5 and 1). Similarly, for a left turn, we should expect the model to return a negative value. multi_test allows us to predict the steering angle of a series of images. The exact images are specified by the parameters, IMG_dir (the image directory), start_ind (the stat index), and end_ind (the end index). For example, the parameters, "IMG0", 0, and 50 correspond to the first 50 images in the directory "IMG0." Multi_test returns statistics about the predicted steering angles including the minimum, maximum, average, and standard deviation. This data serves as a sanity check before we consider testing the model on a real road

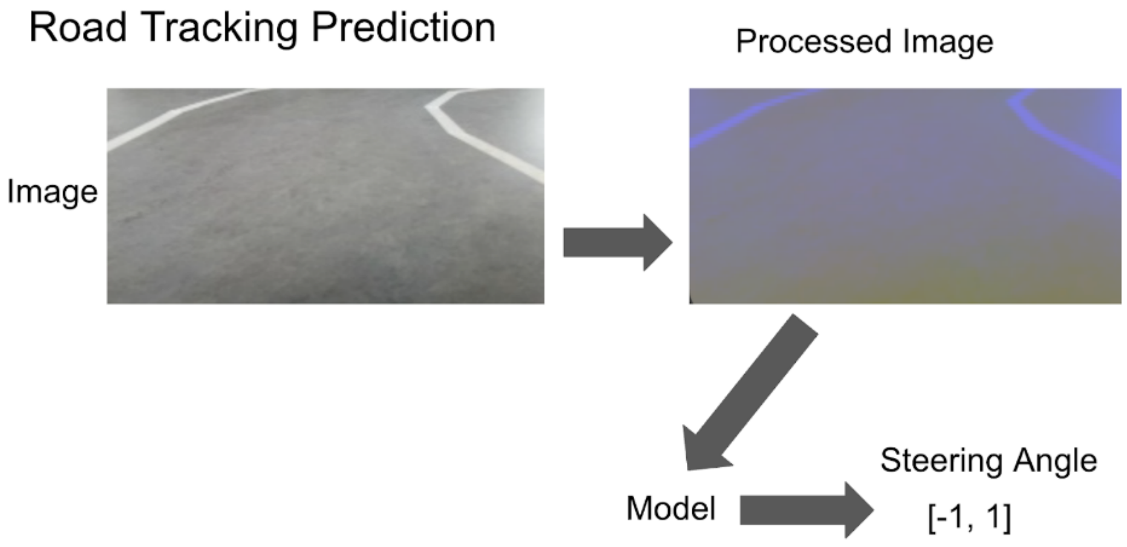


Figure 11: Output of the Road Tracking Model

5.6 try_out.py

This file combines both sign recognition and road tracking into one main file. A while loop continuously runs and receives frames from the Raspberry Pi Camera.

The following variables are defined before the main functions of the file: steeringSen is the steering sensitivity, it could be any valid float, but be aware that the steering angle caps out at 1. maxThrottle is the speed, this ranges from 0 to 100. motor initializes a motor with 8 and 11 for the pin numbers (you can use any GPIO pins, these are just the ones that we chose). This will be used to control the robot's movements. road_tracking_frames keep track of the number of frames the camera is processing. This will be used later on to ensure that steering angle prediction runs once every 30 frames, and sign recognition runs once every 30 frames. We don't want facial recognition and road tracking running every single frame because then the program would run at a much slower speed and make it laggy.

facial_recog_steering and road_tracking_steering hold the predictions of the sign recognition and road tracking models. For road tracking, the predictions run once every 30 frames using road_tracking_frames. When this variable is a multiple of 30, the variable will be reset to 0. Every time road_tracking_frames becomes 0, the program will call on the deep learning model for road tracking and stores the model predicted steering angle into road_tracking_steering. Outside of the if statement, road_tracking_frames is incremented by 1 to keep track that a single frame has passed. When road_tracking_steering is within [0,15], each of the motors will move at a speed corresponding to the steering angle road_tracking_steering.

road_tracking_steering is capped to be within -1 and 1, with negative value indicating left turn, positive value indicating right turn, and zero indicating go straight.

Every time road_tracking_frames becomes 25, if the four sign corners of the real-time image are detected, the program will preprocess the image with warping, cropping, type converting, and resizing. After this sequence of operations, the preprocessed image will be fed into the sign recognition deep learning model, which outputs a sign label that is stored in facial_recog_prediction. Then, the program will check if the sign is close enough based on the proportion between the space within the quadrilateral specified by the four detected corners and the whole region of the real-time image. facial_recog_steering will be updated if and only if the sign is deemed as close enough. Immediately after that, if facial_recog_steering has been updated, then we move the motors at the corresponding facial_recog_steering with a duration of 0.8 second. If not all of the four sign corners of the real-time image are detected, the program will not move the motors at road_tracking_frames 25.

5.7 utils.py

utils.py is a library in Step-2 training of Road Tracking. It contains all the functions necessary to train a convolutional neural network model for this task. All the functions will be used in the file train.py.

Road tracking is a linear regression task that predicts a continuous numerical value for the steering angle. On the other hand, sign recognition is a classification task that interprets the signs as discrete values. This part of the project is based on a CV zone self-driving car course (see appendix). There are 8 steps to fully accomplish this task.

The essential library for this file is cv2 and TensorFlow. cv2 handles tasks related to image and video processing, feature detection, and object recognition. The TensorFlow library takes care of the linear regression machine-learning process of the project. This powerful tool is great for general purposes and building effective neural networks. Numpy and Pandas are used to do the data processing needed for proper training and testing of the model. Not many major changes are made to the data, but using these packages creates an easy way for us to create arrays of data for the models to sort through.

5.7.1 Step 1

Step 1 of model building is to initialize data. There are two functions in this step.

The first function, getName, takes the parameter of the file path of the image file. The function will parse the file path, and return the last two directories of the image file, removing the local directory info. Then importDataInfo takes the file path of the steering datalog.

The number of subdirectories in the specified path is calculated by dividing the length of the list returned by os.listdir(path) by 2. This is because the log data is assumed to be organized into subdirectories, where each subdirectory contains the image files and corresponding driving log data for one recording session.

This data is stored in a Pandas dataframe, which is similar to a spreadsheet or a nested array. The for-loop in this function iterates over the range (however much data you would like to train on) it defines, and it reads through these subdirectories within the range (The files are named in numerical order to define the range). This is a hard-coded value that could be modified to read data from different subdirectories.

For each iteration, the for loop takes the one image folder, which is created through a complete run-through of the data collection process, and the corresponding steering log. We call the getName function to properly format the image files and put both parts of the data into a dataframe called dataNew. It's then appended to the global variable data, to generate a big training set. Finally, the function will return this big dataframe. Usually, the purpose of dividing the image folders is to distinguish between the direction the robot is training on.

5.7.2 Step 2

Step 2 is to visualize and balance data. After the data is cleaned and properly formatted, we graph and scale the data appropriately. The purpose of this step is to check the data we collected are complete, and help make the training to be more efficient.

balanceData aims to balance the distribution of samples in a dataset of driving images by removing some of the excess samples. We often take a lot of straight-road data, which could diminish the right/left turn data. By removing some of the straight data, we can balance the distribution of samples in a dataset, which can improve the performance of machine learning models by reducing the bias in the straight road category.

It takes in a pandas DataFrame called data, which is 2 columns of data, with image and steering angle. The function creates a histogram with 31 bins to visualize the distribution of the steering angles in the data. It then looks through all the samples in each bin, and shuffles them. It then creates a balanced dataset with 50 randomly selected samples per bin called binDataList, and adds the rest data indices into removeindexList. After the function looks through all the bins, it will remove all data entries in the removeindexList, those data would be deleted from the original pandas DataFrame and fed into Step 3. The procedure is called balancing data. The balanced data is then plotted out for comparison purposes.

5.7.3 Step 3

Step 3 is preparing for processing: The main function of this step is to convert the data which is stored in Pandas Dataframe into a Numpy object. The purpose of such conversion is memory efficiency. When we are making complex mathematical calculations over larger scale data such as to create a model using this data, the Numpy object will perform better than the Pandas Dataframe.

loadData takes in the file path of the image, and the steering data, and returns the two numpy Objects, one named imgPath, and the other one as steering.

```
indexed_data = data.iloc[i]
imagesPath.append(os.path.join(path, indexed_data[0]))
steering.append(float(indexed_data[1]))
```

The function uses a for loop to add the data they collected either the path of the image and the steering value into their separate Numpy object.

Once all the data are added to the Numpy object, then they are turned into an Array object, which makes them more convenient to use. The two arrays will be returned from the function and be used in the upcoming steps.

5.7.4 Step 4

Step 4 is present in the train.py file, not utils.py, and makes use of the sklearn module, which provides methods that aid in training neural network models with NumPy. Specifically, we use the train_test_split method to take our road images and corresponding steering data and split those into different data sets: two for training, xTrain and yTrain, and two for testing the accuracy of our model, xVal and yVal, where x represents road images and y represents the steering angles. The images in either a training or validation folder are randomly decided by the method.

- imagesPath: The list containing all the file paths for the images
- steerings: The list containing all the corresponding steering angles for the images
- test_size: Specifies what percent of the data to be used for testing; 0.2 means 20% of the data is used for validation, and 80% for training

- random_state: Sets a seed value for the random number generator that is responsible for randomly sorting the images into one of the four folders. In our case, we use 10, which we got from the CVZone tutorial we followed.

```
xTrain, xVal, yTrain, yVal =
train_test_split(imagesPath, steerings, test_size=0.2,random_state=10)
```

5.7.5 Step 5

augmentImage takes imgPath and steering and performs 1 of the 4 image manipulations to generate different types of images. Nothing is done to the steering angle. The function returns the Numpy array representation of the image and its corresponding steering angle.

Specifically, the function read the image using imgPath, and turns this into a Numpy array. The reason for the conversion is similar to step 3. A Numpy array has a special property that makes it use a lot less memory than others and can be processed at a faster speed. On top of that, python also has convenient methods that would make manipulate the image in Numpy array format.

The image goes through a 4 if statement which decides its transformation, at each transformation, there is one-half of the chance the image would perform the transformation. The transformation in step 5 includes tilt, zoom, changing brightness, and mirroring the image.

5.7.6 Step 6

preProcess takes in the Numpy array representation of the image, and processes the image before it is used to create a model. The main purpose of this image processing is to make the array suitable for neural networks to be trained on and make the training process faster and/or more efficient.

```
cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
```

The function first changes the image's colorspace for RGB(red green blue) into YUV(brightness blue red) using the open cv library. The conversion is common for image processing and it is better for image compression, color correction, and video encoding/decoding.

```
cv2.GaussianBlur(img, (3, 3), 0)
```

This line applies a Gaussian blur to the image using a 3x3 kernel. Gaussian blur smooths and reduces noise in an image or makes the image less sensitive to small details. Because we don't need to look for the finer detail, and only care for the speed of this process. Hence why we decide to use this function. The 0 parameter specifies the standard deviation of the Gaussian kernel, which in this case is set to 0, meaning that OpenCV will automatically calculate the value based on the kernel size.

```
img = cv2.resize(img, (200, 66))
```

This resizes the dimension of the image into 200 px by 66 px, as suggested by the CVZone video.

```
img = img/255
```

Finally, the image's numpy array value is reduced to range from 0 to 1 by dividing by 255. This is also suggested by CVZone video. The processed image is returned.

5.7.7 Step 7

createModel is similar to cnn created by facial recognition. Refer back to the facial recognition model documentation for consultants.

This model is specifically for regression tasks, as the road-tracking process image captures, and predicts a continuous value based on that input. The model is implemented using the Keras API with the TensorFlow backend.

```
model.add(Convolution2D(24, (5, 5), (2, 2),
             input_shape=(66, 200, 3), activation='elu'))
   model.add(Convolution2D(36, (5, 5), (2, 2), activation='elu'))
   model.add(Convolution2D(48, (5, 5), (2, 2), activation='elu'))
   model.add(Convolution2D(64, (3, 3), activation='elu'))
   model.add(Convolution2D(64, (3, 3), activation='elu'))


   model.add(Flatten())
   model.add(Dense(100, activation='elu'))
   model.add(Dense(50, activation='elu'))
   model.add(Dense(10, activation='elu'))
   model.add(Dense(1))
```

The model architecture consists of a series of convolutional layers, followed by fully connected (dense) layers. The convolutional layers are used to extract spatial features from the input images, while the dense layers are used to transform these features into a final output value. All layers use the 'elu' activation function, which allows us to have nonlinear output since the input data is not linear. Except for the last layer, which is a linear layer.

Finally, the model is compiled with the Adam optimizer and a learning rate of 0.0001. The loss function used is a mean squared error (MSE).

The model is then returned, this is a common way to create a model in self-driving tasks, and the majority of coding design is learned from CVZone. Small modifications are made to suit our task.

5.7.8 Step 8

The last function of this file is dataGen, it takes all the image paths, steering values corresponding to the images, batchSize - number of images in each batch, and trainFlag - boolean expression.

The function will train on this batch based on the epoch size, which is not defined here.

For each batch, the function selects a random image from the imagesPath list using and assigns it to the variable index. The function calls augmentImage to apply data augmentation to the selected image and its corresponding steering angle using index if trainFlag is true. Otherwise, it reads the image using mpimg.imread and assigns the steering angle from steeringList to the variable steering.

Either way, after the image is obtained, the function applies a preprocessing step to the image using preProcess function, and add the preprocessed image and the corresponding steering angle to two separate lists: imgBatch and steeringBatch.

Finally, the function uses yield to return a tuple containing the preprocessed image batch and its corresponding steering angle batch. This function is a generator function, which allows it to generate batches of data on the fly instead of loading all the data into memory at once.

6 Appendix

6.1 Code

Github Repository

6.2 Installation Guide

Requirements.txt

6.3 References / Citations

CV Zone Course: Self-Driving Car using Raspberry Pi

6.4 Demo of Features

Linkedin Demo Video