TITLE

IMAGE AUGMENTATION TO IMPROVE NEURAL NETWORK PERFORMANCE IN AUTONOMOUS VEHICLES

AUTHOR

Fareed

DATE & LOCATION

26th August 2024
Melbourne, Australia

Abstract

Overview

Autonomous vehicle technology that utilizes Artificial Intelligence (AI) and Machine Learning (ML) is gaining widespread adoption around the world with many commercial and industrial applications with the promise of transforming the way we travel and transport goods. This technology has commercial, industrial and scientific applications globally and utilizes a variety of AI/ML technologies including image processing, computer vision pattern recognition, object detection, sensor fusion, satellite navigation and radio communication.

Artificial neural networks take the centerstage of such advances playing a crucial role in synthesizing inputs from various sources and making real-time decisions about speed, direction and manoeuvring vehicles in diverse conditions such as different types of roads, terrain and traffic conditions, while maintaining safety, comfort, time and efficiency.

Training these neural networks requires a substantial amount of data, which needs to be generated and utilized for training models along with such methods as behavioural cloning for accurate inference when deployed. However, this data tends to have aspects of localization and thus the models trained on the data tend to not perform well when deployed in environments that differ from the training conditions.

Expanding the variety and quantity of training samples collected is one way of increasing the quantity and variety of training samples. However, simply collecting unique and varied images and label data is not always feasible or practical and can be expensive in terms of time, money and effort. Instead, techniques that augment the data are employed to modify existing sample images. These techniques modify images to simulate different

conditions, while preserving the essential information relating to the road and other factors affecting driving behaviour. This enhances the training dataset and improves model performance in a variety of real-world scenarios.

Project Objectives

This project aims to investigate the benefits of using image training data augmentation while training a neural network of an autonomous land vehicle. The goal is to achieve accurate self-driving behaviour with a reduced number of training data image samples. This will decrease time for data collection and also create a more robust model with precise features extracted that can perform well on different road and lighting conditions.

Methodology

This study was conducted on a self-developed fully-functional DIY RC car using a hobby-grade RC car chassis with a motor and servo controller by a microcontroller with a radio and a Raspberry Pi. This setup was controlled by a self-designed remote controller for controlling the data acquisition process. Associated software was developed for the remote controller, the radio microcontroller with the 433MHz receiver, the host communicator between the Raspberry Pi and the receiver, data collector, model trainer, model validator and the autonomous car driving controller. A sample track was created in an open area for data collection and validation of neural network models. The setup allowed for experimentation in a controlled environment with sufficient variance in terms of sunlight, brightness and other real-life outdoor conditions. The source code of this project is available at https://github.com/fareed1983/racer.

Results

It was found that data augmentation improved the trained neural network model's robustness, accuracy and generalization capability. Far superior model fit without overfit or underfit characteristics were observed when some image augmentation techniques were which were validated by superior performance observed in models trained with augmented image data while navigating the track under different conditions compared to models trained without such augmentation. Thus validating the efficacy of data augmentation techniques for enhancement of model performance.

Conclusion

The project successfully demonstrated that augmentation of image training data is beneficial in training of neural networks used to control autonomous vehicles. Accurate self-driving behaviour was achieved with a smaller dataset and thus reduced the time and effort required for data collection. The findings highlight the importance of using image augmentation techniques to train models capable of better performance in real-world scenarios that may differ from conditions at the time of training data collection. Future work could explore enhancement of the architecture of the neural network and other image augmentation techniques.

Keywords

Autonomous Vehicles, Artificial Intelligence, Machine Learning, Computer Vision, Data Augmentation, Convolutional Neural Networks, Sensor Fusion, Image Processing, Raspberry Pi, Remote-Controlled Car, TensorFlow, Keras, Autonomous Navigation, Model Training and Evaluation, AI, ML.

Table of Contents

Abstract.....2

Overview.....2

Project Objectives.....3

Methodology.....3

Results.....4

Conclusion.....4

Keywords.....4

List of Tables.....8

List of Figures.....9

Chapter 1: Introduction to the Project.....11

Overview.....11

Study Background.....11

Research Aims.....12

Research Significance.....13

Study Limitations.....13

Structural Outline.....14

Chapter 2: Review of Literature.....15

Historical Foundations and Early Work.....15

Advancements in Convolutional Neural Networks.....16

Augmentation Techniques for Error Correction and Bias Reduction.....17

Broadening the Scope of Image Augmentation.....18

Simulated Environments for Augmentation Testing.....19

Conclusion and Future Directions.....19

Chapter 3: Research Objectives and Methodology....20

Research Objectives....20

Research Problem....20

Hypothesis Statement....21

Research Design....22

Types of Data Used....24

1. Generated training data (images, throttle and steering)....24

2. Data collected when training the model....24

3. Data collected when self-driving....25

Data Collection Method....26

Steps to Build Self-Driving Apparatus and Collect Data....26

Step 1: Assembling the R/C Model Kit....26

Step 2: Experimentation in Controlling the Car....27

Step 3: Building the Transmitter (TX) and Receiver (RX)....27

Step 4. Components Mounted on R/C Car Chassis....28

Step 5: Building the Test Track....29

Step 6: Collecting training data....30

Step 7: Training neural network model....30

Step 8: Validating model against validation set and measure performance....31

Step 9: Loading the model on car and observe self-drive behaviour....32

Data Collection Apparatus....33

Description of Software Written for the Project (https://github.com/fareed1983/racer).....34

unoTest module....34

rx Module....35

tx Module....37

pi/host Module.. 38

pi/progs/mqttClient Module...40

pi/progs/trainData Module.. 40

training Module..42

pi/progs/pilot1 Module..50

Data Collection Instrument...52

Sample Size...53

Sampling Technique.. 54

Data Analysis Tool...55

Chapter 4: Data Analysis, Results and Interpretation...56

Experiment 1: Model trained without any augmentation...56

Experiment 2: Model trained with flip augmentation..57

Experiment 3: Model trained with brightness augmentation..59

Experiment 4: Model trained with contrast image augmentation.......................................60

Experiment 5: Model trained with image invert augmentation..62

Experiment 6: Model trained with image flip augmentation combined with invert augmentation... 63

Experiment 7: Model trained with image blur augmentation...65

Experiment 8 – Model trained with image flip combined with brightness augmentation. .66

Chapter 5: Findings and Conclusion...69

Comparisons of Trained Models..69

Findings.. 70

Conclusion.. 70

Chapter 6: Recommendations and Limitations of the Study...71

Bibliography.. 73

List of Tables

Number	Description
Table 1	Description of training samples collected and used for model training

List of Figures

Number	Description
Figure 2.1	ALVINN Architecture (Pomerleau, 1988)
Figure 2.2	CNN architecture of the NVIDIA self-driving model. (Bojarski et al. 2016)
Figure 3	Some random samples of collected training images
Figure 3.1	Assembling base Tamiya TT-02 car model kit
Figure 3.2	Experimentation for controlling the assembled car
Figure 3.4.a	Labelled car top profile showing components
Figure 3.4.b	Car transmitter with components
Figure 3.5.a	Building the track in the balcony of my home
Figure 3.5.b	Completed track for self-driving car data collection and validation
Figure 3.6	Capturing training data by driving car on track
Figure 3.7	Block diagram of the data collection apparatus
Figure 3.8	SBC states tracked by RX
Figure 3.9	Sending MQTT commands using MQTT Dash application
Figure 4.1	Loss curve of model trained without any augmentation
Figure 4.2	Loss curve of model trained with flip augmentation
Figure 4.3	Loss curve of model trained with brightness augmentation
Figure 4.4	Loss curve of model trained with contrast augmentation
Figure 4.5	Loss curve of model trained with invert augmentation
Figure 4.6	Loss curve of model trained with flip and invert augmentation
Figure 4.7	Loss curve of model trained with blur augmentation
Figure 4.8	Loss curve of model trained with flip and brightness augmentation
Figure 5.1	Comparison of model fit characteristics

Figure 5.2 Comparison of self-driving performance scores

Chapter 1: Introduction to the Project

Overview

Artificial Intelligence, namely neural networks, and Machine Learning is increasingly used to control autonomous vehicles (AV). Such vehicles have many applications in industrial, commercial, military, logistical and scientific applications. Self driving vehicles can also help people with medical disabilities. Such vehicles use latest advances in technologies like image processing, computer vision, pattern recognition, object detection, sensor fusion, satellite navigation, and radio communication. Inputs from various sources are processed and neural networks are used to infer the speed and to steer the vehicle while ensuring the safety of passengers and goods. Autonomous land vehicles encounter a variety of road, terrain and traffic conditions. For neural networks to be robust, they have to be trained on a significant volume of training images, sensor and control inputs that are meticulously collected.

Study Background

Several car manufacturers are already training autonomous driving models and deploying them to vehicles sold to end-users. Enthusiasts and researchers have developed projects like DonkeyCar (donkeycar.com) that demonstrate the potential of DIY autonomous vehicles to study, experiment and refine self-driving technologies on a smaller scale. This project utilizes a similar approach by developing an end-to-end system encompassing hardware for a scaled-down autonomous vehicle and the remote for control and training, the software for the same, the data collection method, the training of the neural network utilizing images augmented in different ways, validation of the trained model mathematically and by real track testing of the RC vehicle and comparison of different augmentation techniques.

Convolutional neural networks or CNNs enable AVs to interpret and infer responses to different road environments. They process a large number of image data, extracting features and making decisions for navigation of AVs. Such networks are trained by adjusting weights to reduce the loss value of a loss function. Overfitting is a common occurrence in such training which is a condition leading to being trained too well on the training data, including noise and outliers, which leads to poor generalization of new, unseen data used at the time of validation or post-deployment.

Data augmentation techniques involve the generation of additional training samples by transforming the original dataset with operations like rotations, translations, scaling, flipping, varying brightness and contrast, hue, and adding noise. Thus different conditions are simulated without painstakingly collecting additional data gathered under all possible conditions.

Research Aims

A significant challenge remains: the performance of neural networks are found not to perform as expected when deployed in environments that differ from training conditions. This research explores the potential of image augmentation techniques to enhance neural network performance of autonomous vehicles, aiming to achieve self-driving behaviour with fewer training samples.

To explore the effectiveness of image augmentation to train neural networks of an autonomous driving car to be more robust and performant and to compare between the different kinds of such augmentation.

Research Significance

Autonomous vehicles technology is gaining widespread adoption. Newer methods of training neural networks for AVs are constantly evolving. The methods should ensure that networks continue performing under diverse conditions of deployment. This research will contribute to these advances by demonstrating the benefits of image augmentation techniques specific to training of neural networks of autonomous vehicles. The study is of value to both academic research and also has practical applications in the AV industry. As this study is done on a small-scale implementation based on an RC car platform, it can serve as testing grounds for larger-scale applications and as a platform for further experimentation. For the AI/ML hobby community affiliated to enthusiast and competitive AV interests, this research is of particular importance because competitions are held on a track which the AV system under test has not been exposed to before and the operators are provided a limited time to gather input training data thus requiring augmentation to improve robustness in trained models.

Study Limitations

This research sets the scope as deliberately narrow, focusing on the specific impact of image augmentation on neural network performance in AVs. Some of the limitations are:

1. We are only using one CNN based neural network model throughout the scope of this research. Different image augmentation techniques could benefit different neural network designs. Thus our findings will be limited to the specific neural network model under study.
2. We use a scaled-down vehicle platform based on a hobby grade chassis and a custom remote controller. Also our road simulation or track is in a 15×15 feet space created

using vinyl tape. This may not simulate real-world driving conditions when translated into a full-scale land vehicle.

3. There are different types of autonomous vehicles and we are focused only on land vehicles driving on a flat surface and this may not be universally applicable to all AV system types and environments.

Structural Outline

Chapter 1: Introduces the study, outlining the research objectives, questions, significance, and limitations.

Chapter 2: Reviews existing literature on neural networks, data augmentation, and their application in AVs to identify key strategies and research gaps.

Chapter 3: Presents the research design, including the qualitative and quantitative methodologies used in the study. It also shows the experimental setup and data collection, and analysis procedures used.

Chapter 4: Discusses the results and compares the performance of models trained with and without data augmentation and evaluating their robustness in various conditions.

Chapter 5: Concludes the study by summarizing key findings in-line with the hypothesis tested.

Chapter 6: Discusses key recommendations and limitations of the study

Chapter 2: Review of Literature

Image augmentation techniques to improve neural networks for autonomous driving vehicles has been a focus of research in the relevant area since 1988 or earlier. This section synthesizes the literature available to gauge the most common techniques employed that improved robustness in such networks.

Historical Foundations and Early Work

Dean Pomerleau (1988) demonstrated ALVINN (Autonomous Land Vehicle In a Neural Network) consisting of a 3-layer back-propagation network, as depicted in Figure 2.1, designed for the task of road following using images of 30×32 resolution using techniques now called behavioural cloning. He noted that collection on actual road images would be logistically difficult due to a very large number of training samples required. They developed a simulated road generator to augment for the gap in image data availability. Network training was performed using these artificial road "snapshots" with realistic noise levels. There were also a variety of lighting conditions simulated in the training images.

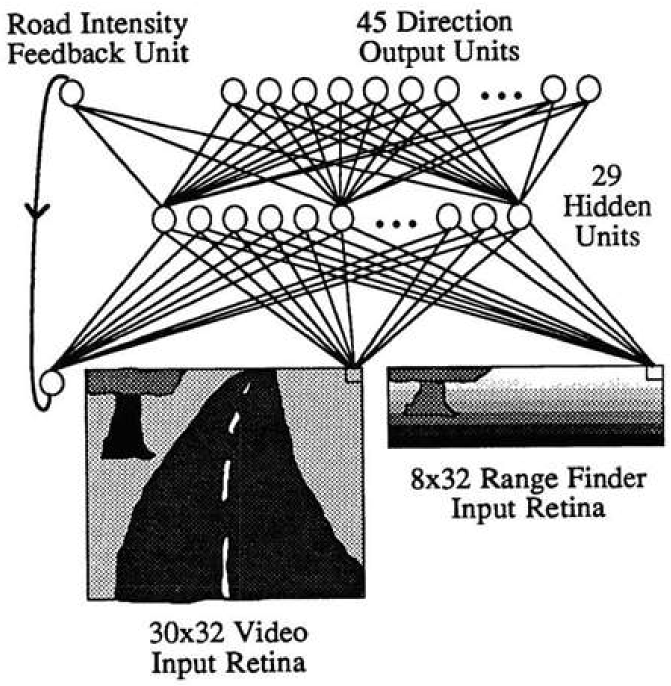


Figure 2.1: ALVINN Architecture (Pomerleau, 1988)

He noted that doing so would provide a more robust activation level for the output road intensity unit a.k.a. feature in modern parlance. He also noted that the network must be shown examples on how to recover once mistakes are made. This early use of data augmentation aimed to enhance the robustness of the network by exposing it to diverse scenarios and recovery situations. (Pomerleau 1988).

Advancements in Convolutional Neural Networks

Bojarski et al. (2016) trained a CNN in their study on the end-to-end learning approach for self-driving cars at NVIDIA avoiding recognition of human-designated features and a collection of "if, then, else" rules. They used raw data from three cameras; left, centre and right; to feed three 66×200 pixel data to train a neural-network. Their CNN architecture depicted in Figure 2.2, consists of a normalization layer, multiple convolutional layers for feature extraction, and three fully connected layers for steering control. They collected data by driving on a wide variety of road in diverse set of lighting and weather conditions. The data was collected in clear, cloudy, foggy, snowy and rainy weather, both day and night.

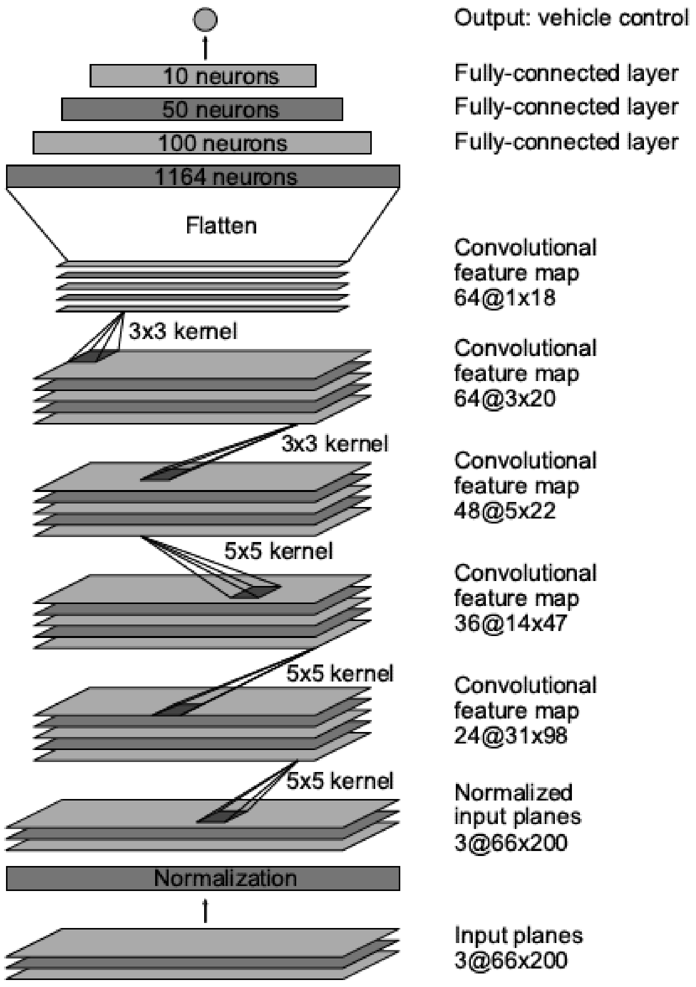


Figure 2.2: CNN architecture of the NVIDIA self-driving model. (Bojarski et al. 2016)

They emphasized the importance of data augmentation in training robust neural networks for autonomous driving. They added artificial shifts and rotations to teach the network how to recover from a poor position or orientation. They noted that images augmented artificially adds undesirable artifacts as they increased the magnitude. (Bojarski et al., 2016).

In further research, Mariusz Bojarski et al. (2017) explored the decision-making process of their self-driving model, PilotNet. They came to the conclusion that PilotNet learns features that “make sense” to a human, while ignoring structures in the camera images that are not relevant to driving without the need for hand-crafted rules. This study underscored the network's ability to discern important visual cues autonomously. (Mariusz Bojarski et al. 2017)

Augmentation Techniques for Error Correction and Bias Reduction

Chen Z. and Huang X. (2017) presented an end-to-end learning approach to obtain proper steering angle to maintain the car in the lane. They did not consider night driving images to simplify the problem. They removed straight driving bias by up-sampling the data of curved roads by five and shuffled the data randomly before training. On data augmentation, they noted that self-driving vehicles tend to drift away and error correction data must be provided during training. (Zhilu Chen and Xinming Huang 2017)

With the understanding that a self-driving car model would benefit from specific augmentation techniques that can be logically applied with steering and throttle data, to avoid overfitting on such models, it would thus make sense to retain characteristics that ‘make sense’ to humans and retain the conceptual entities such as the edges of the road and the drivable track itself.

Broadening the Scope of Image Augmentation

Shorten and Khoshgoftaar (2019) examined various data augmentation techniques and their importance in improving deep learning models especially convolutional neural networks (CNN). Such techniques are very important in expanding the variations in the datasets used for neural network training. They noted that the techniques improved model generalizations and robustness. They surveyed techniques like geometric transformations (including flipping, rotation and cropping), and color space transformations like varying brightness and contrast. They also used kernel filters for sharpening and blurring images. This helped models in varying real-world conditions.

While the focus of the survey done by Shorten and Khoshgoftaar (2019) was not autonomous vehicles, the did mention self-driving cars for transferring training data into night-to-day scale, winter-to-summer, or rainy-to-sunny scale. Thus we argue that the discussed augmentation methods can be applied to the field. AVs require robust neural network models capable of handling diverse driving conditions. Their studies show that by improving the range of training image data points with artificial augmentation, we can reduce overfitting and improve the performance of a model on new data not seen before. (Shorten & Khoshgoftaar, 2019)

Fernández-Llorca et al. (2017) applied random data augmentation techniques, including geometric transformations such as affine and perspective transformations, and changes like noise addition and color changes and noted substantial improvement in road detection performance. These finding would apply to a self-driving model which theoretically learns features that are analogous to road detection in the CNN layers but use the hidden layers to infer steering and throttle outputs. (Fernández-Llorca et al. 2017)

Simulated Environments for Augmentation Testing

Kiaan U. (2021) used a car simulator instead of a real scaled-down self-driving car to test the models trained with different image augmentation techniques. They also used three simulated cameras as required for the original NVIDIA self-driving CNN-based neural network model. They concluded that brightness and flip augmentation together help in training the generalization model of a self-driving car tested on a simulated environment in a 3D virtual track. Note that they did not generate training data on a real track neither did they test the end-result on a physical self-driving car. (Upamanyu & Ramaswamy, 2021).

Conclusion and Future Directions

The studies that were reviewed highlight the critical role of image augmentation in training robust neural networks for autonomous driving. Such techniques as simulated data generation, geometric and color space transformations, and simulated environments have been shown to be effective in improving neural network model performance. Research in the future should integrate these techniques with more specific methods that are justified in the domain of self-driving cars and evaluate the performance in real-world test scenarios.

Chapter 3: Research Objectives and Methodology

Research Objectives

Following are the objectives of this research:

1. To develop a hardware platform for self-driving vehicles utilizing an off-the-shelf single-board computer with a camera and independent microcontrollers for human inputs with a remote controller.
2. To augment image data used for training of neural networks of an autonomous vehicle obtained from a scaled-down model of a real-world autonomous vehicle platform and then gauge the improvement of efficiency of the model given such augmentation.
3. To compare the performance of neural network models trained using different combinations of image augmentation techniques suitable for autonomous driving vehicles using the same initial data-set.
4. To evaluate the robustness of these models in different driving scenarios against a validation data-set obtained from scaled-down autonomous driving vehicle and also try the models for real self-driving of the scaled-down RC car based AV.

Research Problem

Although there is extensive research done on data augmentation techniques to improve neural network performance, much of this research is focused on the general benefits of data augmentation in AI/ML, specific impact and techniques for augmenting image training data specialized for autonomous vehicles remains relatively unexplored. General image augmentation techniques cannot benefit autonomous vehicles where precise steering and throttle control is required and an approach tailored for autonomous vehicles will perform better than ones suited for other neural network domains. Such image augmentation have to be tailored for autonomous driving vehicles. Self driving cars require a lot of training samples under different conditions which is not feasible. Thus, training data has to be augmented to efficiently use training data to train self-driving models. Which augmentation techniques best enhance self-driving car models have not been well documented which this research intends to do.

This research aims to answer the following questions:

1. Do CNN based autonomous driving vehicle neural networks perform better when trained with image augmentation techniques as compared to without them?
2. Which image augmentation techniques cause CNN based neural networks for self driving cars to perform better?
3. How do autonomous vehicles with models trained on data augmentation techniques handle diverse road and lighting conditions?

Hypothesis Statement

Null Hypothesis (H0): Performance of CNN based models for autonomous vehicles trained with image augmentation techniques is not significantly improved.

Alternative Hypothesis (H1): Performance of CNN based models for autonomous vehicles trained with image augmentation techniques is significantly improved.

To test the hypothesis H1, we will compare the fit of models trained on a primary real-world dataset quantitatively by measuring the fit using mean-square error function and also empirically by providing a score to self-driving behaviour on a test track.

Research Design

In this research a quantitive approach is used to evaluate the effects of image augmentation techniques stand-alone or in combination. Models are trained using the behavioural cloning approach and evaluated for their validation loss against a randomly chosen test dataset from the available samples. A complete end-to-end apparatus including a 1/10th scale RC car with receiver and transmitter was developed for this research as depicted in Figure 3.7 which included a test track as shown in Figure 3.5.b. Following is the high-level design of the research:

1. A human drives the car using the transmitter around the test track created in a 15×15 feet area. The throttle and steering inputs are captured and at the same time the camera image frames are tagged with the timestamp of the inputs and both, the inputs and camera frames are saved to disk. Each session creates a new folder with the current timestamp and collects the image samples and the control and sensor data in it.
2. The collected training samples are transferred to an Apple MacBook Air M1 laptop which has a decent GPU and models are trained with various combinations of augmentations applied. The image samples collected are resized to 160×120 and converted into greyscale.
3. A CNN based neural network model is trained using the original resized images and also images on which augmentations as specified in the current experiment are applied. The model is allowed to train for a maximum of 60 epochs while providing a provision of early stoppage

4. We perform a series of experiments by training the model on different types and combinations of augmentation techniques. We train the model without augmentation as a base value and subsequently train models using augmentations like flip, brightness, contrast, blur and inversion and combinations of these. The base neural network model design is kept constant to compare and evaluate the performance of the different models trained.

5. 20% of the image samples collected from the total samples is used for validation of the trained model. The validation set is selected by random sampling in each experiment. At each epoch of training, the validation error is measured and graphed. This allows us to gain the quality of the fit of the model to the sample control data collected. The epoch resulting in the lowest validation error is selected for the training session. A training session may stop early if determined that the validation error is not decreasing.

6. A correlation to models trained with and without different types of augmentation is studied quantitively to deduce if autonomous vehicle models are more robust when augmentation is applied. We do so by comparing the validation loss and model fit parameters that determine overfit or underfit.

7. As the controlled conditions of pre-existing dataset for validation which is a random sample from the overall dataset may not reflect adequately the real-wold conditions, each experimental model trained with different image augmentations or image augmentation combination is then loaded on the self-driving car platform and tested on the track. The observations are video recorded and observed for various aspects related to proper driving performance.

Types of Data Used

1. Generated training data (images, throttle and steering)

The image size is 640×480 and are captured at a frame-rate of 20 frames per second. While the steering and throttle inputs are received at approximately 8-10 samples per second. After the data was collected, sections that had mistakes while driving and sections where the car was not been driven were deleted. However minor mistakes made while driving which were corrected were kept in to mimic such behaviour in the trained models. Some varied random frame image data samples used are represented below.

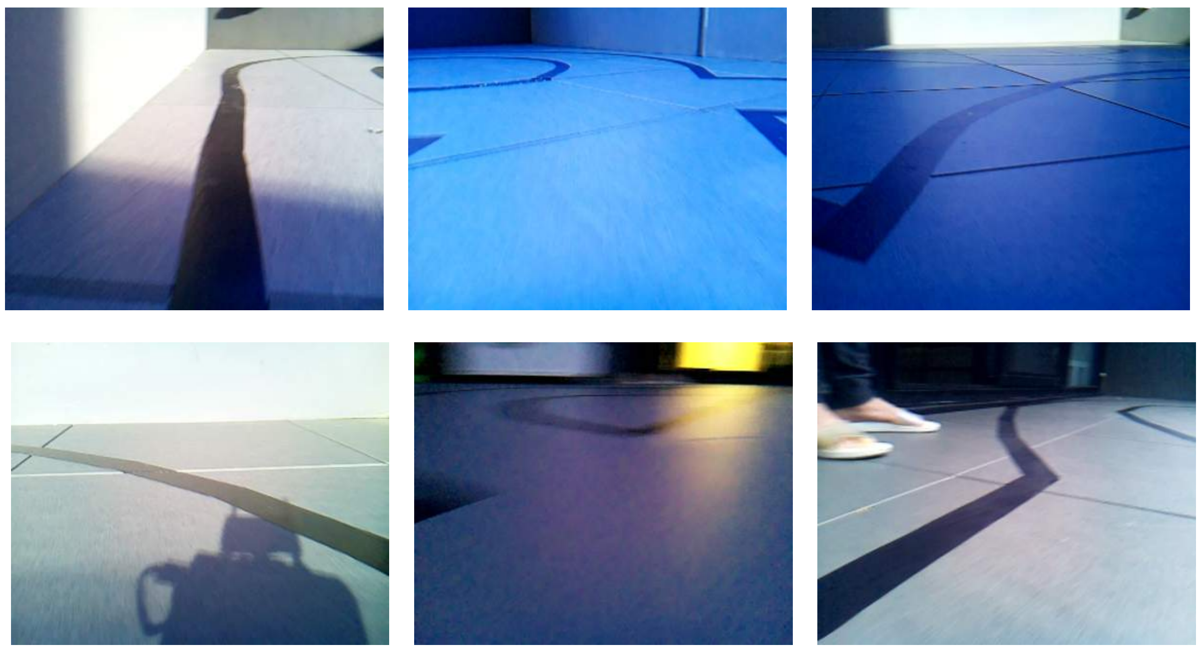


Figure 3: Some random samples of collected training images

2. Data collected when training the model

For each experiment, the learning curve is generated and collected which plots the training and validation loss over each epoch period. After training validation is done again and the validation loss is captured. This is useful to compare performance of models trained with different training data augmentation methods.

Machine learning models are trained in epochs in which a complete pass is made through the entire dataset. We use the Mean Squared Error (MSE) function to calculate training and validation loss. There are various benefits of using MSE to calculate loss values which is defined as the average of the squares of the differences between the predicted values and the actual values. Smaller errors don’t have much of an effect on the value however larger errors would be amplified and thus incorrect predictions are penalized.

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

<u>Training Loss:</u> At each epoch, training loss is calculated using forward pass on the training data and the difference between predicted and the actual values are calculated using mean squared error. Then a backward pass is done where the loss value is used to update weights using back-propagation. This is done in multiple batches for every epoch and the loss values are accumulated for the average epoch training loss.

<u>Validation Loss:</u> At the end of each epoch, the validation data set is computed for predictions and MSE is used to calculate the accumulated average loss. This shows the generalization capabilities of the model on data that was not used for training.

3. Data collected when self-driving

Observations from the video recorded are filled-in as a sheet and kept as reference for the real-world practical performance comparisons of the performance of models trained with different training data augmentation methods. This sheet references the points on the track numbered as per Figure 3.5.b. This is loosely empirical data but a system for proper measurement of on-track performance is future work.

Data Collection Method

To achieve this objectives of data collection, we developed a scaled-down version of a self-driving vehicle platform based on a customised remote-controller car by adding a base microcontroller with sensors and actuators for the electronic speed/throttle controller and a steering servo motor. To control the car, a custom remote controller was developed communicating with the base microcontroller through 433 MHz radio frequency. The microcontroller on the vehicle platform was interfaced with a Raspberry Pi 4 single-board-computer with a camera module where python programs were executed to capture training data. The car was run on a sample track simulating a real road journey that contained many different combination of curved and straight sections. These training data samples were moved to a high-specification GPU accelerated computer where the TensorFlow and Keras frameworks were used to train a Convolutional Neural Network (CNN) based models using different combinations of image augmentation. Subsequently these models were tested on the autonomous driving vehicle platform. For the models that performed reasonably while self-driving, combinations of image augmentation were tried and their evaluation loss functions were compared.

Steps to Build Self-Driving Apparatus and Collect Data

This section outlines the steps in developing the scaled-down self-driving platform in its evolutionary stages.

Step 1: Assembling the R/C Model Kit

The first step involved procuring and building a 1/10 scale Tamiya TT-02 R/C model assembly kit that has a polycarbonate body (Figure 3.1). The kit comes with a brushless motor. A compatible Hobbyking electronic speed controller (ESC) and Savox servo for steering control was also procured. A 5000mAh Ni-Mh rechargeable battery was also procured.

Figure 3.1: Assembling base Tamiya TT-02 RC car kit

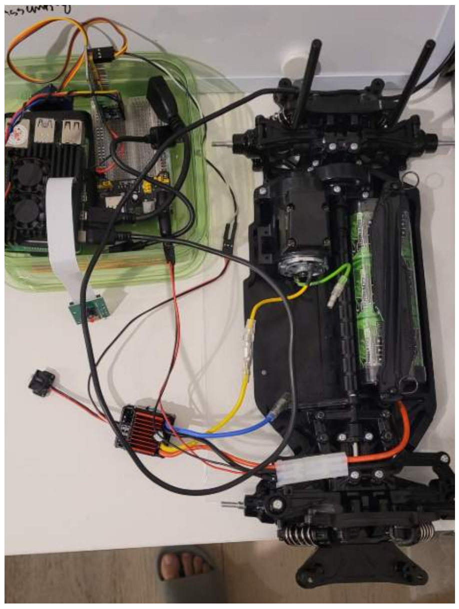


Figure 3.2: Experimentation for controlling the assembled car

Step 2: Experimentation in Controlling the Car

A PWM module was obtained and connected via I2C to the Raspberry Pi 4 (with camera) which was used to control the electronic speed controller and the servo. Figure 3.2 shows an early experimental setup using MQTT to control the apparatus. It was found that control through WiFi would not be as responsive and therefore accurate steering and throttle inputs would not be possible. Bluetooth or ready-made joystick control was not explored due to lack of time and resources and it would make the project have less novelty.

Step 3: Building the Transmitter (TX) and Receiver (RX)

Two Adafruit Feather M0 RFM69 modules were already available to me and it was decided to use these as a low-level 433MHz based radio controller for the RC model. The RX module would communicate to the Raspberry Pi through I2C and UART. The TX module was built on protoboard. Note that most of the software for RX and TX was written by this stage. The RX module was also built on a prototype PCB at the same time.

Step 4. Components Mounted on R/C Car Chassis

The RX, display, ultrasonic sensors, Raspberry Pi, camera, power supply, and SSD were r profile.

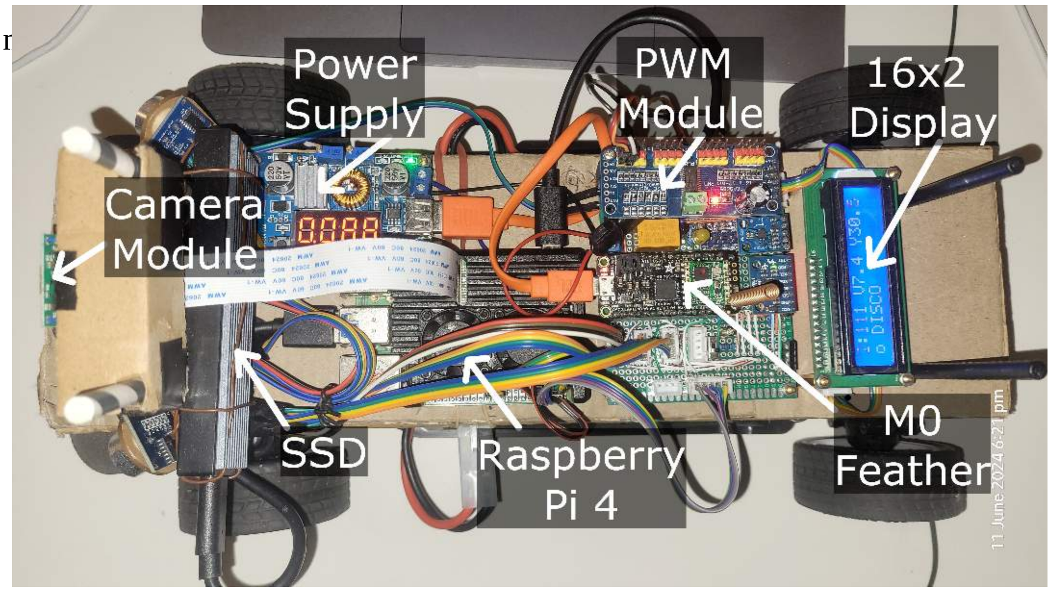


Figure 3.4.a: Labelled car top profile showing components

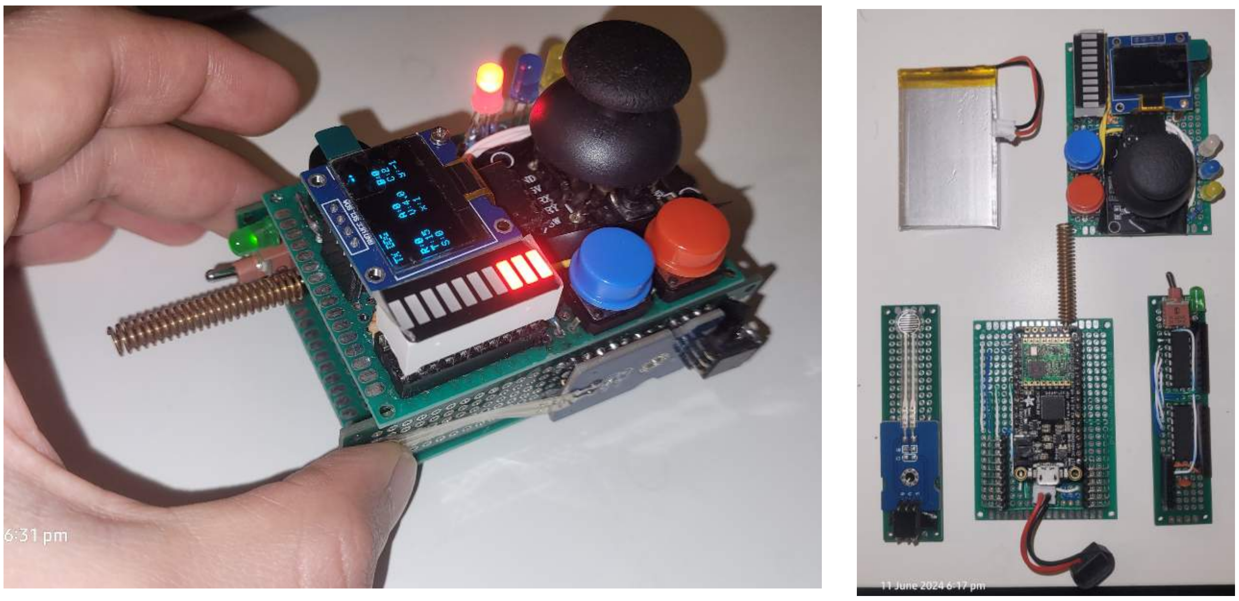


Figure 3.4.b: Car transmitter with components

Step 5: Building the Test Track

A proper track with many turns and straights was made with adhesive vinyl as shown in figure 3.5.a and completed track in Figure 3.5.b.

Figure 3.5.a: Building the track in the balcony of my home

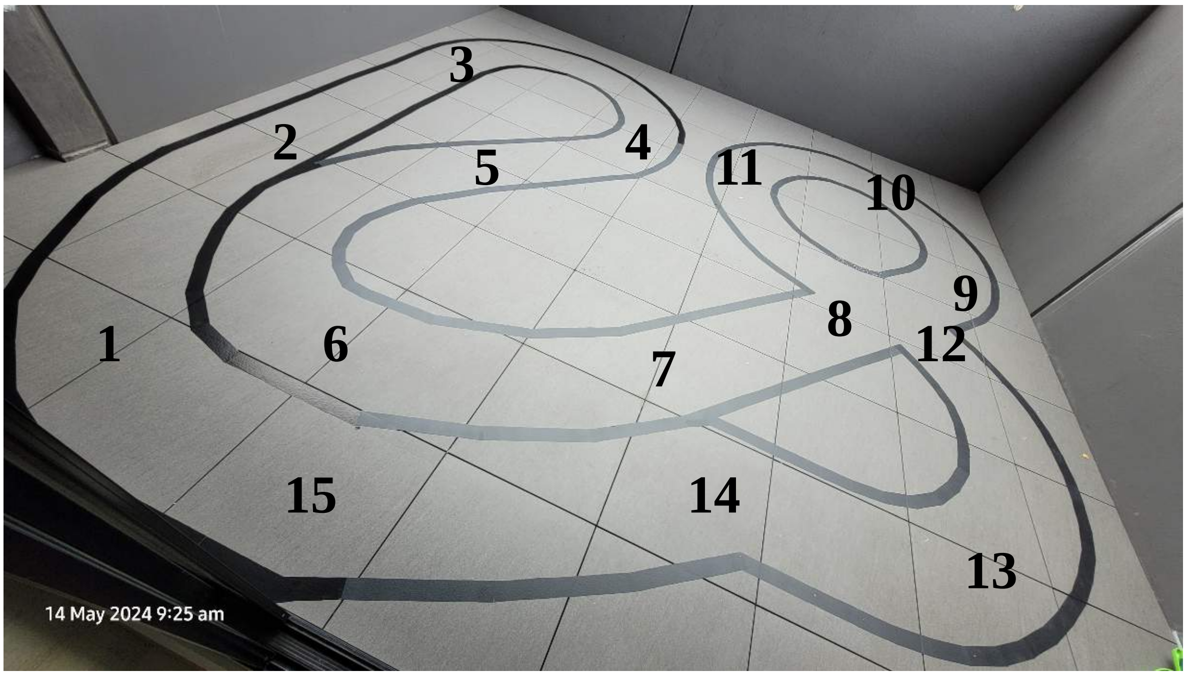


Figure 3.5.b: Completed track for self-driving car data collection and validation. Note the numbering is overlaid on the image for model evaluation purposes.

Step 6: Collecting training data

Training data images and video samples were collected using completed car and track. For this, a host program for communicating between the Raspberry Pi and the M0 Feather was was written and Python program for collecting training data was developed. First controlling the car was learnt and after practice, the car was driven around the track using the TX module as shown in Figure 3.6. Video images were sampled at about 20 FPS and saved into a SSD with steering and throttle data.

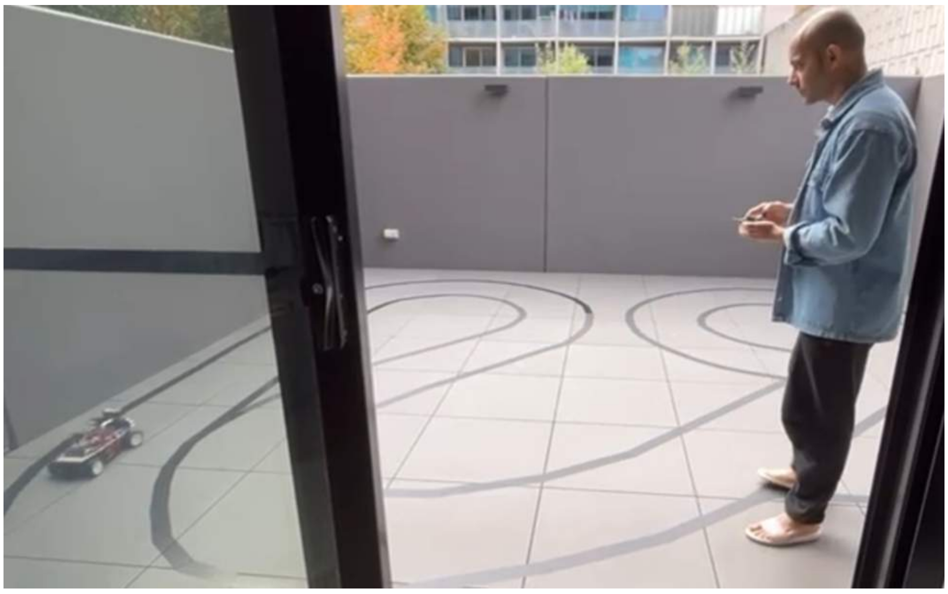


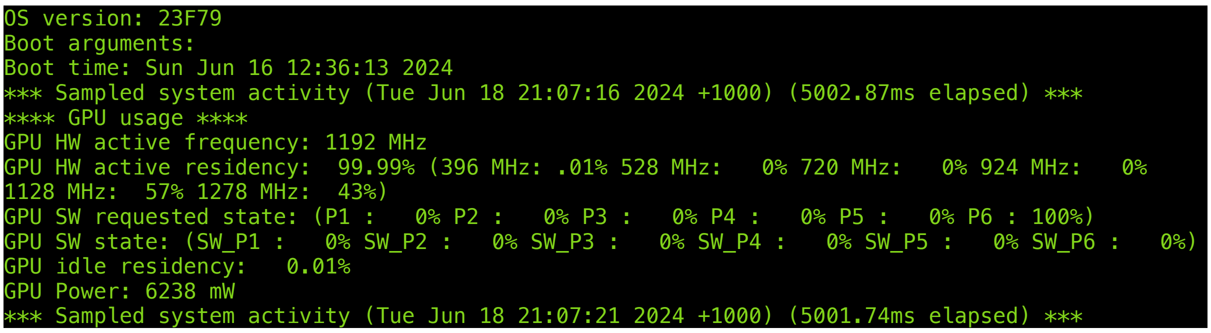
Figure 3.6: Capturing training data by driving car on track

Step 7: Training neural network model

The scp utility is used to copy the timestamped folder generated during each data generation session. The training program is used to utilize the images and the input data-set to train a neural network model. The training is accelerated using the GPU and this results in faster model training.

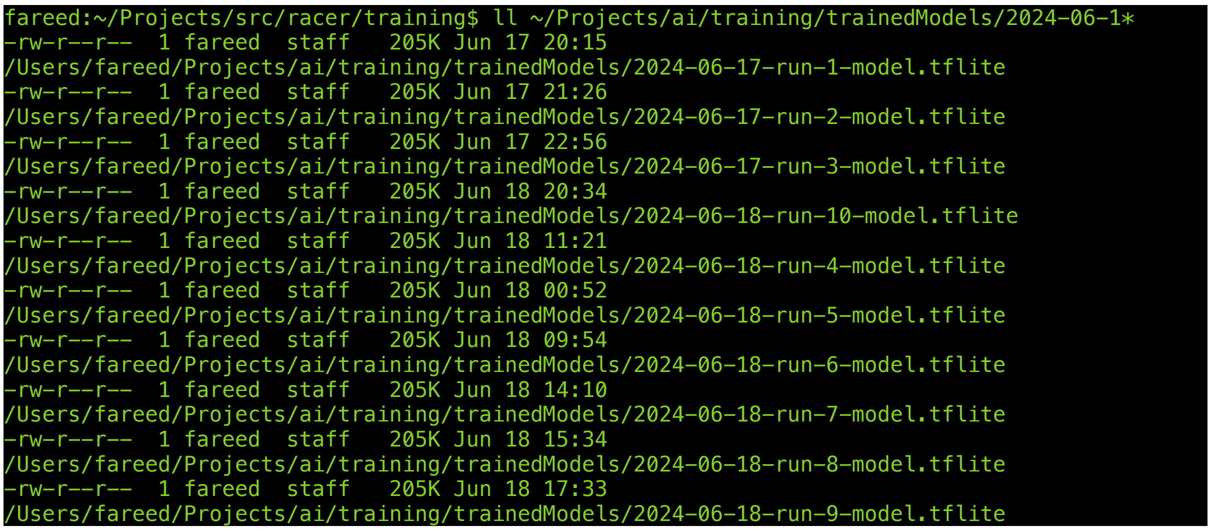
One way to execute the training program is by activating the Python virtual environment, use PIP to install all dependencies and then using the command, ‘python main.py `find ~/Projects/ai/training -type d -name "2024*"`’. Note that this command will train the neural network using all folders starting with 2024 in the given path.

GPU utilization can be checked with the “sudo powermetrics --samplers gpu_power –show-usage” on macOS. Below is the sample output of this command when training is in progress and GPU is utilized. Note that the GPU HW active residency shows 99.99% in the sampling time when a model was being trained.



```
OS version: 23F79
Boot arguments:
Boot time: Sun Jun 16 12:36:13 2024
*** Sampled system activity (Tue Jun 18 21:07:16 2024 +1000) (5002.87ms elapsed) ***
**** GPU usage ****
GPU HW active frequency: 1192 MHz
GPU HW active residency:  99.99% (396 MHz: .01% 528 MHz:   0% 720 MHz:   0% 924 MHz:   0%
1128 MHz:  57% 1278 MHz:  43%)
GPU SW requested state: (P1 :   0% P2 :   0% P3 :   0% P4 :   0% P5 :   0% P6 : 100%)
GPU SW state: (SW_P1 :   0% SW_P2 :   0% SW_P3 :   0% SW_P4 :   0% SW_P5 :   0% SW_P6 :   0%)
GPU idle residency:   0.01%
GPU Power: 6238 mW
*** Sampled system activity (Tue Jun 18 21:07:21 2024 +1000) (5001.74ms elapsed) ***
```

After training is completed, the Keras model is exported to the folder self_drive_model and subsequently converted to a TFLite model named self_drive_model.tflite. These models are then collected in another folder as below:



```
fareed:~/Projects/src/racer/training$ ll ~/Projects/ai/training/trainedModels/2024-06-1*
-rw-r--r--  1 fareed  staff   205K Jun 17 20:15
/Users/fareed/Projects/ai/training/trainedModels/2024-06-17-run-1-model.tflite
-rw-r--r--  1 fareed  staff   205K Jun 17 21:26
/Users/fareed/Projects/ai/training/trainedModels/2024-06-17-run-2-model.tflite
-rw-r--r--  1 fareed  staff   205K Jun 17 22:56
/Users/fareed/Projects/ai/training/trainedModels/2024-06-17-run-3-model.tflite
-rw-r--r--  1 fareed  staff   205K Jun 18 20:34
/Users/fareed/Projects/ai/training/trainedModels/2024-06-18-run-10-model.tflite
-rw-r--r--  1 fareed  staff   205K Jun 18 11:21
/Users/fareed/Projects/ai/training/trainedModels/2024-06-18-run-4-model.tflite
-rw-r--r--  1 fareed  staff   205K Jun 18 00:52
/Users/fareed/Projects/ai/training/trainedModels/2024-06-18-run-5-model.tflite
-rw-r--r--  1 fareed  staff   205K Jun 18 09:54
/Users/fareed/Projects/ai/training/trainedModels/2024-06-18-run-6-model.tflite
-rw-r--r--  1 fareed  staff   205K Jun 18 14:10
/Users/fareed/Projects/ai/training/trainedModels/2024-06-18-run-7-model.tflite
-rw-r--r--  1 fareed  staff   205K Jun 18 15:34
/Users/fareed/Projects/ai/training/trainedModels/2024-06-18-run-8-model.tflite
-rw-r--r--  1 fareed  staff   205K Jun 18 17:33
/Users/fareed/Projects/ai/training/trainedModels/2024-06-18-run-9-model.tflite
```

Step 8: Validating model against validation set and measure performance

Refer to section describing software module “training”.

Step 9: Loading the model on car and observe self-drive behaviour

Refer to section describing software module “pi/progs/pilot1”.

Data Collection Apparatus

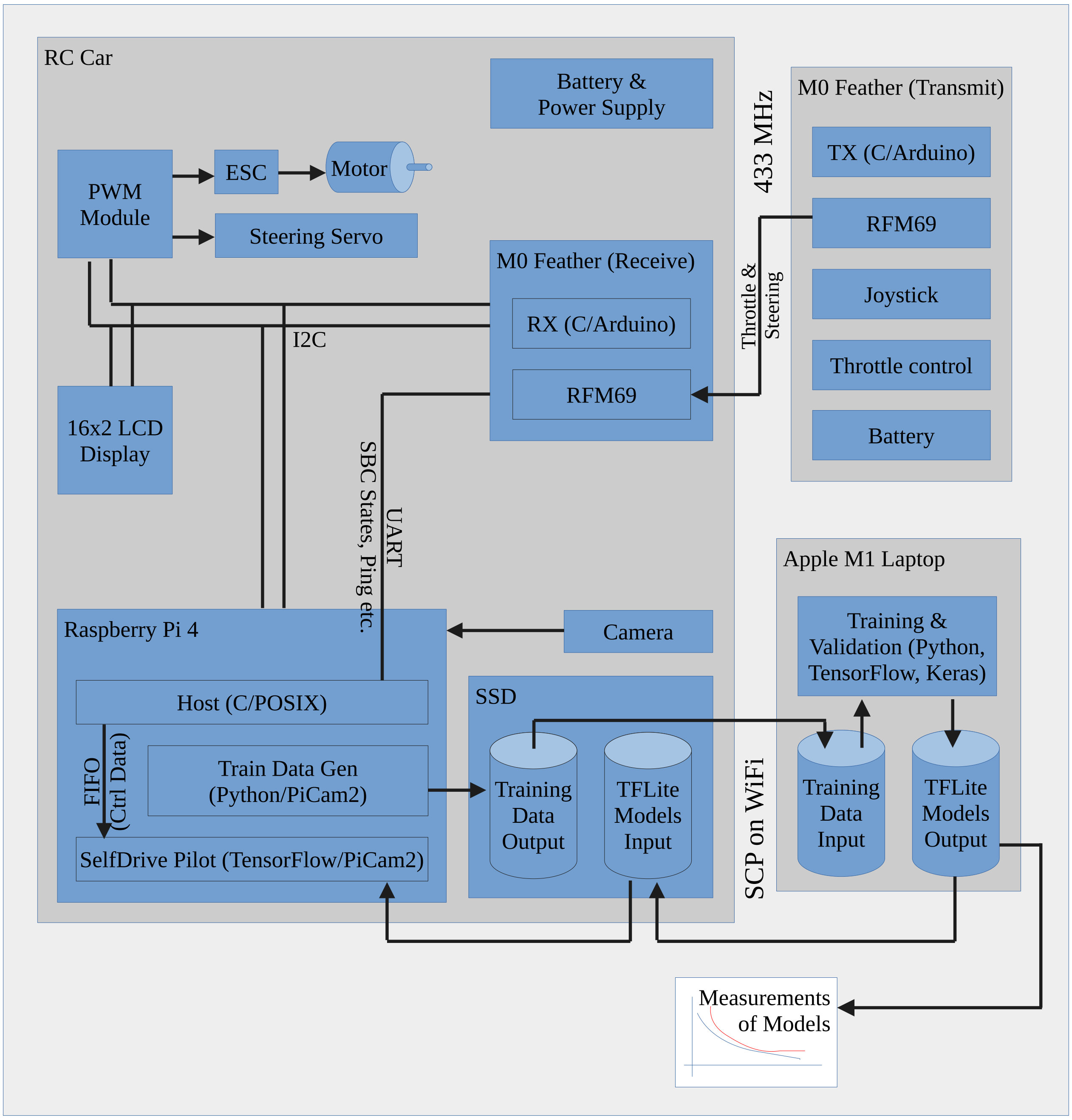


Figure 3.7: Block diagram of the data collection apparatus

Description of Software Written for the Project (https://github.com/fareed1983/racer)

The source code is available on the Github repository https://github.com/fareed1983/racer. The code was written using the VS Code IDE. For software that runs on microcontrollers, the PlatformIO extension for VS Code is used. It consists of the following structure:

unoTest module

Testing OLED, ultrasonic distance, pressure switch & joystick peripherals on an Arduino Uno prior to constructing the RX and TX modules. This was the preliminary test-bed for validating the functionality of sensors and output devices. It is built using PlatformIO extension for VS Code. It uses the Arduino framework with the following libraries:

- ericksimoes/Ultrasonic – for distance measurement using ultrasonic sensors
- adafruit/Adafruit SSD1306 – for 0.96 cm OLED display module
- arduino-libraries/Servo – for controlling the servo and ESC (electronic speed controller) using PWM output of the Arduino

The steering inputs are taken from the joystick module which consists of potentiometers on two axis which are read through the analog inputs of the Arduino. The force sensor is also read through analog inputs and will be used for throttle. The program averages readings of the steering and throttle inputs over time and displays debug information over the OLED module. It was used for hardware validation purposes. Note that the code also includes experiments that were undertaken to ascertain the heading value from the gyroscope module however it was kept out of the scope of the AI input data due to lack of time.

rx Module

The source code of the receiver module written in C based on a Adafruit M0 Feather RFM69. It uses I2C and UARTG to communicate with the SBC (Raspberry Pi or other). Peripheals it uses are: 16x2 LCD on I2C, PCA9685 PWM driver on I2C, relay to control SBC power, 4 HR-SC04 ultrasonic sensors, gyroscope on I2C and magnetometer on I2C. It receives commands from the TX module. It was built using the Arduino framework and uses the following external libraries:

- marcoschwartz/LiquidCrystal_I2C
- adafruit/Adafruit PWM Servo Driver Library – To control PCA9685 PWM module
- adafruit/Adafruit SSD1306 – To control 16x2 diagnostic display
- ericksimoes/Ultrasonic – For 3 ultrasonic distance sensors
- z3t0/Irremote – For IR LEDs for future expansion (not in current scope)
- rfetick/MPU6050_light – Gyroscope for future expansion (not in current scope)
- fastled/FastLED – Neopixel type LED strips for future exansion (not in current scope)
- adafruit/Adafruit HMC5883 Unified – Magnetometer for future expansion (not in current scope)

Note that I have also written a comms module that defines the communication between the RX and TX modules and also between the TX and SBC (Raspberry Pi). The protocol is self-explanatory and uses a CRC check. The RadioHead library is used to ensure reliable and encrypted communication between the RX and TX modules. More information can be found at https://github.com/hallard/RadioHead.

The flow of the main.cpp source code is as follows:

- Initialize the required peripherals and components like Serial communication, PWM driver, I2C devices, MPU6050 sensor, ultrasonic sensors, LCD display, and RF69 radio module.
- Center the servo and ESC
- Debug output all I2C devices present
- Gyroscope and magnetometer calibration for future expansion
- Display initial messages on 16x2 display
- Loop
  - Try to receive command from TX with timeout
  - If command received,
    - If command is TX_RX_CMD_SIMPLE_CTRL or TX_RX_CMD_DIRECTION_CTRL then map the values to the ESC and servo pulses in microseconds
    - Else follow the state chart for the SBC. See Figure 3.8
  - Read distances from ultrasonic sensors and if within a certain range, stop the vehicle to avoid collision
  - Measure battery voltage and display on the LCD
  - Gyroscope calculations – for future expansion
  - Update display with steering, throttle and other inputs received or if disconnected from TX
  - Send ping message to the SBC and receive a pong in response through the serial port.

- If in SBC_ST_PROG_PASSIVE mode, send the sensor inputs to the SBC (Raspberry Pi) which may be used for such purposes as training data generation.
- If in SBC_ST_PROG_MASTER mode, stop any I2C communication (as I3C multi-master did not work) and allow for the SBC to use I2C peripherals.

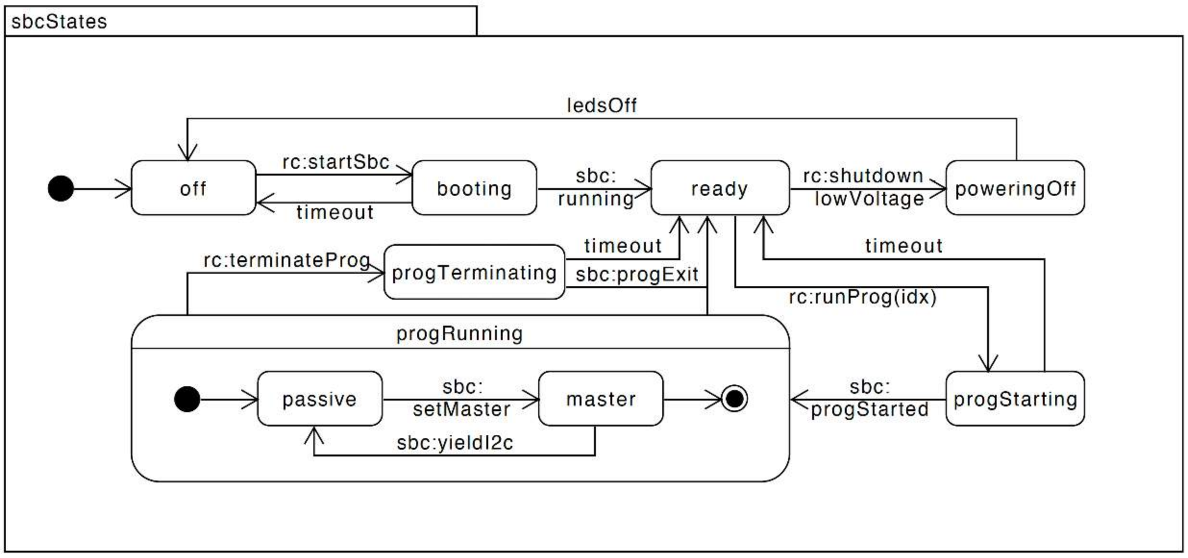


Figure 3.8: SBC states tracked by RX. Note that not all states were implemented due to lack of time

tx Module

The source code of the transmitter module written in C based on a Adafruit M0 Feather RFM69. It uses inputs from a joystick module and a force sensor. It uses a 128x64 I2C OLED LED module, 10 segment LED bar graph, buttons, 2 color LEDs, 1 RGB LED and a buzzer for output. It sends commands to the RX module. It was built using the Arduino framework and uses the following external libraries:

- adafruit/Adafruit SSD1306 – for 0.96 cm OLED display module
- z3t0/IRremote – IR LED command generator for future expansion

See the note on rx regarding communication as the same comms module and RadioHead is used here.

Following is the high-level flow of the code:

- Initialization of components like shift register, input and output pins, joystick, serial and I2C communication, RF69 radio module, OLED display etc.
- Loop
  - Read sensor inputs from force-sensitive resistor and map it to linear scale, joystick x and y axis and buttons.
  - Display the LED outputs and buzzer output through a shift register.
  - Also map the force-sensor value to 0-10 to output on the 10 segment LED bar graph.
  - Map the joystick and force-sensor values to a scale from -100 to 100 and calculate the moving average for smoother transitions of inputs.
  - Display joystick x and y values, steering mapped value, throttle value, button status, battery voltage, connection status on the OLED display.
  - Send control command to the TX module using the RF69 radio module and the RadioHead library.

pi/host Module

On the Raspberry Pi, it is the communicator between the SBC and RX. It sets the state of the SBC on RX, (is envisioned to) host Python scripts (other processes) that run for AI training, control and other purposes. opens a named FIFO to pass data to such processes on one thread and receives and transmits control commands and events to and from RX.

It uses the same comms library I wrote also used in RX and TX for communication between the SBC and RX. It also starts the Raspberry Pi as a I2C slave (address 0x03) to receive sensor and control (applied steering and throttle values) data being generated by the RX module. The data is only received when the SBC is set to be in passive mode. A loaded Python script (other process) is required to set it in master mode if it intends to use the I2C bus as multi-master mode was not found to be working and debugging it's issues was out of scope. The state negotiation commands are passed through the serial interface. Note that minimal functionality of this module was completed for the project due to lack of time. Currently the module can be executed to manually set the SBC to master (-m as command like argument) or passive mode (-p).

The following is a high-level flow of this program:

- Parses options and sets up the FIFO and UART
- Creates two threads, one for the UART reader and another for the I2C (slave) reader
- Sets state of SBC as running (by sending SBC_RX_EVT_RUNNING on UART) in the TX so that it can start receiving sensor and control data on the I2C bus and then sets the SBC as master if argument provided (by sending SBC_RX_CMD_MASTER). See figure 3.8 for state transitions from current states and states pending implementation.
- In the UART reader thread:
    - If ping received, responds with pong
    - If shutdown received, powers-off the SBC
    - If sensor and control data is received from UART, it is ignored (currently)
- In I2C reader thread:
    - If sensor and control data is received, it is parsed and printed

    - Instance of structure of type senProgData_t is created that is assigned the received sensor and control data and a timestamp
    - The structure data is written to the named FIFO

pi/progs/mqttClient Module

A sample program written to test the I2C communication when the SBC (Raspberry Pi) acts as master. It receives commands via MQTT and controls the steering and throttle. The broker has to be run on the Raspberry Pi. It was tested using the "MQTT Dash" application installed on a mobile phone as in Figure 3.9. The ESC and servo centre points were tested using this method which were found to be around 1500 microseconds. We will not go into much detail of this application as it was only intended to be a test.

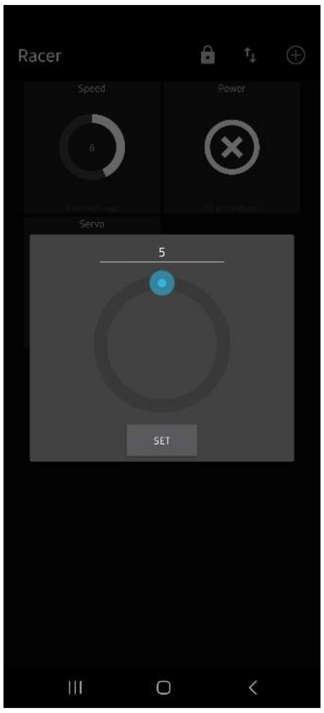


Figure 3.9: Sending MQTT commands using MQTT Dash application

pi/progs/trainData Module

This program written in Python runs when the SBC is in passive mode and is used to capture image frames, resize them and store them as images with a timestamp. It parallelly reads the sensor input from the FIFO produced by pi/host and stores it in a file in the same folder as the images.

Important libraries used are:

- struct – used to unpack binary data that was generated from C structure from pi/host

- cv2 – This is OpenCV which is used for processing images captured from the camera
- picamera2 – This library is used to capture images from the camera attached to the Raspberry Pi

Overall flow:

- The program creates a new folder with the name of the current timestamp
- Starts the read_sensor_data thread and starts capturing images on the main thread
- Sensor thread
  - Opens the FIFO that pi/host produces for consumption
  - Opens binary file for car control and sensor and control data of the running session
  - Reads messages on the FIFO and unpacks them to a python dictionary
  - Sets currTimeStamp variable utilizing a mutex
  - Writes sensor and control data to the binary file
- Main thread
  - Initializes the Raspberry Pi camera using picamera2 and the capture configuration
  - In a loop
    - Captures images in 640x480 and converts it into RGB
    - Saves the image as a JPEG filr with a file name as the currTimeStamp-<n>.jpg – This naming convention allows easy linking of the input sensor and control data to frames later
    - Displays the image

Operation of this program is as follows:

- The TX (433 MHz transmitter) is turned on
- The pi/host is run for SBC to enter passive mode and start receiving sensor and car control data
- This program is run and images, car control and sensor data starts getting written to files – this is the input data for training or training samples
- The program can be terminated with Ctrl+C

training Module

The main.py in this folder is used to train a CNN based neural network model with the training images recorded with pi/progs/trainData and uses the control inputs as labels. It then displays the training loss graph as epochs progress. This program is run on a Apple MacBook Air M1 laptop which can use GPU acceleration for training the neural network faster and more efficiently. The trained model is converted to a TFLite format from Keras which executes more efficiently on a limited resourced Raspberry Pi 4.

The neural network model used is loosely based around the NVIDIA paper "End to End Learning for Self-Driving Cars" (Bojarski et. Al 2016). Figure 2.2 shows the original model and the below is the model adapted used in this project after experimentation with various models:

```
DIM = (160, 120) # (width, height)

model = Sequential([

      Input(shape=(DIM[1], DIM[0], 1)),
      Conv2D(24, (5, 5), strides=(2, 2), activation='relu'),
      Conv2D(24, (5, 5), strides=(2, 2), activation='relu'),
      Conv2D(64, (5, 5), activation='relu'),
      MaxPooling2D(pool_size=(2, 2)),
      Conv2D(64, (3, 3), activation='relu'),
      MaxPooling2D(pool_size=(2, 2)),
      Conv2D(64, (3, 3), activation='relu'),
       Flatten(),
       Dense(100, activation='relu'),
       Dropout(0.4),
       Dense(50, activation='relu'),
       Dropout(0.3),
       Dense(10, activation='relu'),
       Dense(2) # Output for throttle and steering
])
```

Following is an explanation of the neural network model:

- All layers use the ReLU (Rectified Linear Unit) activation function because it is efficient to compute and faster than sigmoid or tanh. ReLU provides sparse activation reducing computation requirement. Also the vanishing gradient problem is mitigated which leads to faster training time because gradients are propagated more effectively. It is said that the activation of ReLU is loosely similar to the activation of neurons in the human brain. It is found to work well with deep learning models.
- Following is the explanation of the layers of the network:
    - Input layer - The experiments were conducted on an input shape of resolution of 160x120 grayscale images.
    - Convolutional Layers1 & 2 – 24 filters with size 5x5 kernel and stride 2. Note that (Bojarski et. Al 2016) in the NVIDIA model had mentioned that the first 3 convolutional layers were strided. Generally it is observed that the first layer extracts very basic features from images like textures and edges. The second layer is thought to extract a bit more complex patterns.
    - Convolutional Layer 3 – This layer is followed by a MaxPooling2D layer which reduces the image The third layer would focus on more complex patterns.
    - Convolutional Layer 4 uses a 3x3 kernel without a stride similar to the NVIDIA model without a stride. However, I have used a MaxPooling layer with a pool size

    of 2x2 to reduce the image size as the starting input shape was larger than the NVIDIA paper. This convolutional layer would be extracting more detailed features.

    - Convolutional Layer 5 similarly uses a 3x3 kernel without a stride. It is possible that this layer continues to refine extracted features.
    - Flatten layer – The 3D feature maps are reduced into a 1D vector that can then be passed to the dense layers which are also called fully connected layers.
    - First dense layer – this layer is fully connected and has 100 neurons. It is intended to learn combinations of high-level extracted features.
    - First dropout layer of 0.4 – Reduce the activations are randomly set to 0 while training thus preventing overfitting to the training data set.
    - Second dense layer – this layer is fully connected and has 50 neurons. It is intended to refine the features learnt in the first dense layer.
    - Second dropout layer of 0.3 – Again, this prevents overfitting.
    - Third dense layer – This has 10 units to narrow down to fewer inferences and the final decision points.
    - Output layer – This is a 2 unit fully connected dense layer to provide the normalized throttle and steering values.
- Further note that the total params produced by this network was 197,050 which is in the same order of magnitude as the parameters used in the NVIDIA model (250 thousand).

Following is the overall flow of the training program:

- The program uses TensorFlow and Keras for AI model training.
- The program takes input directories as input arguments. A list of input directories must be provided and each folder containing the images and sensor data file recorded by pi/progs/trainData.
- The types of augmentations to perform are also taken as input arguments.
- For each input directory, the sensor and control data (labels) file is read and unpacked into throttle and steering input and sensor values. Note that the sensor values are discarded currently as it is beyond the current scope.
- The steering and throttles inputs are normalized from -100 to 100 into 0 to 1.
- For each entry in the sensor and control data added, image files matching the timestamp are searched with names like "source_image_{timestamp_str}_{idx:03d}.jpg"
- If matching files exist, they are appended to the paths list and the control data is appended to the labels list for correspondence by index
- A custom data generator is implemented to provide resized image data on-demand while model training instead of storing all the resized source images and augmented images in RAM which would not be sufficient. Here is where the image augmentation is applied as specified in the command line arguments such that augmented images are also used to feed into the model when training. Currently supported augmentations are: flip, brightness, contrast, blur and invert. The CustomDataGen __getitem__ method when provided an index, returns a batch of resized images and labels on the fly by opening the image in the path at that index, converting it to grayscale and normalizing it to 0-1. Image augmentation is then done as per the parameters when

the program is started. Note that this function returns images in batches given an index. The following code listing shows how the augmentations are applied and augmented images added to the training data set inside the custom data loader class.

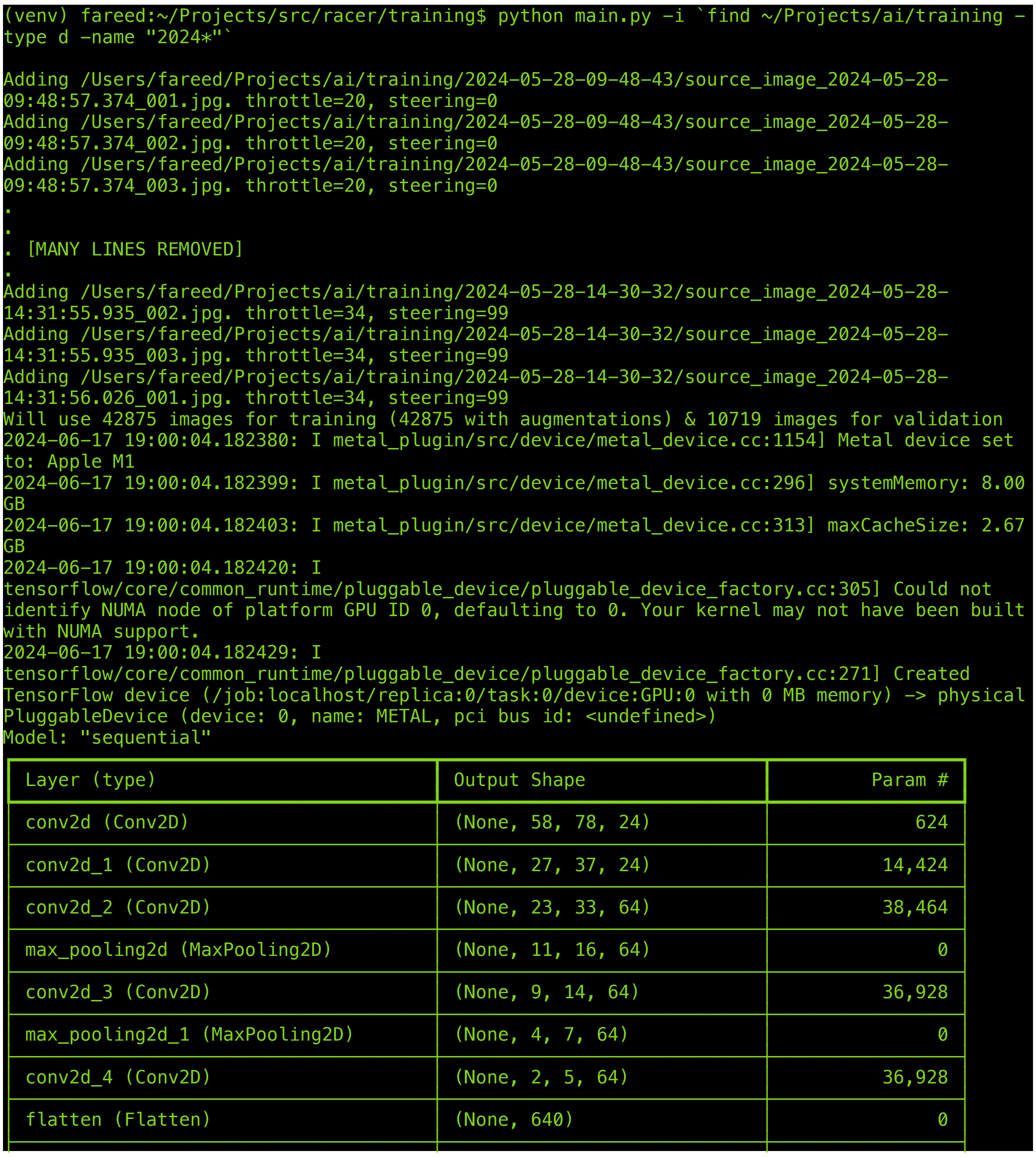
```
def __getitem__(self, index):

    X = []
    y = []
    num_augmentations = len(self.augmentations) + 1

    # The index param in this funciton will assume augmentations are included
    start_index = index * self.batch_size // num_augmentations
    end_index = (index + 1) * self.batch_size // num_augmentations


    batch_indexes = self.indexes[start_index:end_index]
    batch_data = [self.data[k] for k in batch_indexes]
    for path, label in batch_data:
        # Load and preprocess the image
        # print(f"Loading {path}. throttle={label[0]}, steering={label[1]}")
        image = Image.open(path).convert('L') # open jpg and convert to
grayscale
        image = image.resize(self.dim)
        image_array = np.array(image)
        image_array = image_array / 255.0 # Normalize the image

        # Append original image data and label
        X.append(image_array)
        y.append(label)

        for augmentation in self.augmentations:
            aug_image = image.copy()
            if augmentation == 'flip':
                aug_image = aug_image.transpose(Image.FLIP_LEFT_RIGHT);
                aug_label = [label[0], -label[1]]
            elif augmentation == 'brightness':
                factor = random.uniform(0.5, 1.5)
                enhancer = ImageEnhance.Brightness(aug_image)
                aug_image = enhancer.enhance(factor)
                aug_label = label
            elif augmentation == 'contrast':
                factor = random.uniform(0.5, 1.5)
                enhancer = ImageEnhance.Contrast(aug_image)
                aug_label = label
            elif augmentation == 'blur':
                radius = random.uniform(0.25, 1.0)
                aug_image =
                    aug_image.filter(ImageFilter.GaussianBlur(radius))
                aug_label = label
            elif augmentation == 'invert':
                aug_image = ImageOps.invert(aug_image)
                aug_label = label

            aug_image_array = np.array(aug_image)
            aug_image_array = aug_image_array / 255.0
```

```
                    # Append augmented image data and modified label
                    X.append(aug_image_array)
                    y.append(aug_label)

                    X = np.array(X)
                    y = np.array(y)

        # Reshape data for the CNN input
        X = X.reshape(X.shape[0], self.dim[1], self.dim[0], 1)

        return X, y
```

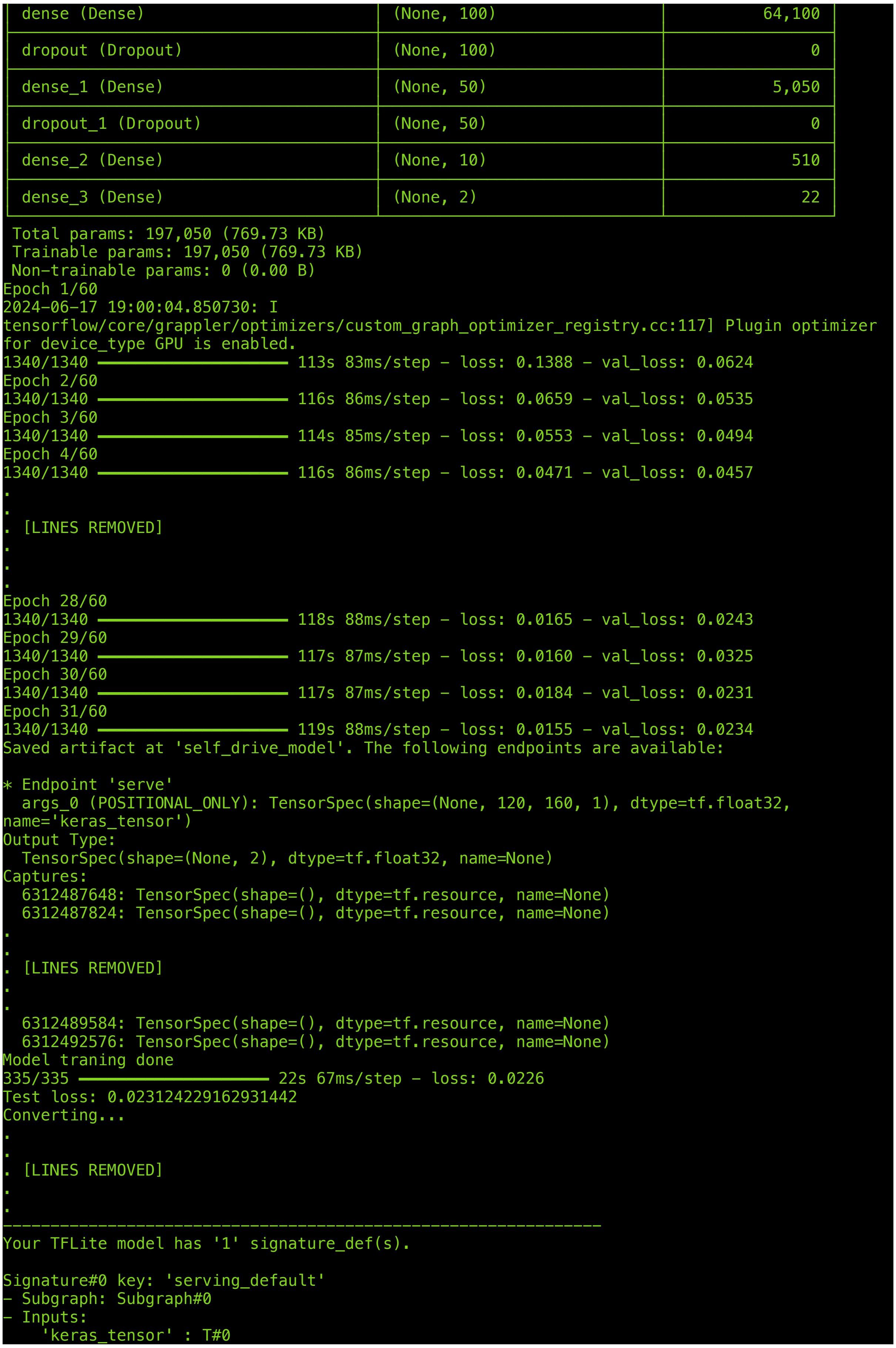
- The input data is split into training data and validation data in an 8:2 proportion.
- The model is created and compiled with the Adam optimizer. Adam is highly efficient by combining Momentum and RMSprop. It uses techniques like momentum to compute gradient descent in the relevant direction faster and is less sensitive to Hyperparameters. An added benefit is that it requires less tuning. It also allows for faster convergence compared to other optimization algorithms. Sparse gradients are handled well
- The model is compiled with a loss function of mean squared error (MSE). MSE is a very commonly used loss function for continuous values. The calculation involves squared difference thus larger errors are emphasized and thus the model is encouraged to be more accurate. It is also easier to implement and provides a single value to grasp the error of the model. It has a single global minimum which helps gradient algorithms optimize and thus provide a more accurate model.
- The model is trained with the model.fit function with provision of early stopping so that training is stopped in case overfitting gets detected. I am using a 60 epochs maximum.
- The model is exported as a Keras model and then converted into a TFLite model
- The test data set is again used to evaluate the final loss

- The loss vs epoch graph is shown. This is the graph that represents the training and validation loss per epoch and is reproduced for each combination of augmentation in the following data interpretation sections.

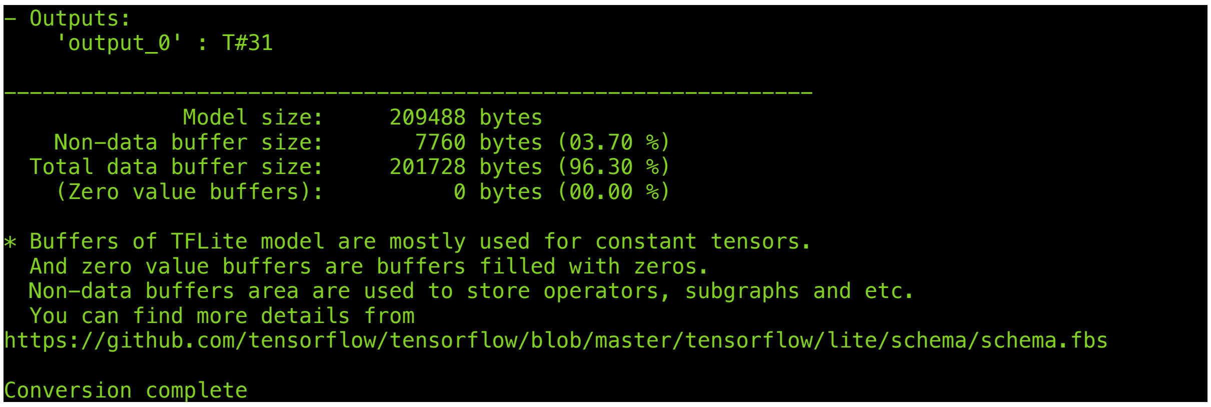
Following is what a sample run of the program looks like. The output is suppressed with dots in the below listing as it would be thousands of lines long. Note the line “Will use 47801 images for training & 11951 images for validation” in the below output.



```
(venv) fareed:~/Projects/src/racer/training$ python main.py -i `find ~/Projects/ai/training -
type d -name "2024*"`

Adding /Users/fareed/Projects/ai/training/2024-05-28-09-48-43/source_image_2024-05-28-
09:48:57.374_001.jpg. throttle=20, steering=0
Adding /Users/fareed/Projects/ai/training/2024-05-28-09-48-43/source_image_2024-05-28-
09:48:57.374_002.jpg. throttle=20, steering=0
Adding /Users/fareed/Projects/ai/training/2024-05-28-09-48-43/source_image_2024-05-28-
09:48:57.374_003.jpg. throttle=20, steering=0
.
.
. [MANY LINES REMOVED]
.
Adding /Users/fareed/Projects/ai/training/2024-05-28-14-30-32/source_image_2024-05-28-
14:31:55.935_002.jpg. throttle=34, steering=99
Adding /Users/fareed/Projects/ai/training/2024-05-28-14-30-32/source_image_2024-05-28-
14:31:55.935_003.jpg. throttle=34, steering=99
Adding /Users/fareed/Projects/ai/training/2024-05-28-14-30-32/source_image_2024-05-28-
14:31:56.026_001.jpg. throttle=34, steering=99
Will use 42875 images for training (42875 with augmentations) & 10719 images for validation
2024-06-17 19:00:04.182380: I metal_plugin/src/device/metal_device.cc:1154] Metal device set
to: Apple M1
2024-06-17 19:00:04.182399: I metal_plugin/src/device/metal_device.cc:296] systemMemory: 8.00
GB
2024-06-17 19:00:04.182403: I metal_plugin/src/device/metal_device.cc:313] maxCacheSize: 2.67
GB
2024-06-17 19:00:04.182420: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:305] Could not
identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not have been built
with NUMA support.
2024-06-17 19:00:04.182429: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:271] Created
TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical
PluggableDevice (device: 0, name: METAL, pci bus id: <undefined>)
Model: "sequential"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ conv2d (Conv2D)                      │ (None, 58, 78, 24)          │             624 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_1 (Conv2D)                    │ (None, 27, 37, 24)          │          14,424 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_2 (Conv2D)                    │ (None, 23, 33, 64)          │          38,464 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d (MaxPooling2D)         │ (None, 11, 16, 64)          │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_3 (Conv2D)                    │ (None, 9, 14, 64)           │          36,928 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ max_pooling2d_1 (MaxPooling2D)       │ (None, 4, 7, 64)            │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ conv2d_4 (Conv2D)                    │ (None, 2, 5, 64)            │          36,928 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ flatten (Flatten)                    │ (None, 640)                 │               0 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
```



```
│ dense (Dense)                     │ (None, 100)              │          64,100 │
├───────────────────────────────────┼──────────────────────────┼─────────────────┤
│ dropout (Dropout)                 │ (None, 100)              │               0 │
├───────────────────────────────────┼──────────────────────────┼─────────────────┤
│ dense_1 (Dense)                   │ (None, 50)               │           5,050 │
├───────────────────────────────────┼──────────────────────────┼─────────────────┤
│ dropout_1 (Dropout)               │ (None, 50)               │               0 │
├───────────────────────────────────┼──────────────────────────┼─────────────────┤
│ dense_2 (Dense)                   │ (None, 10)               │             510 │
├───────────────────────────────────┼──────────────────────────┼─────────────────┤
│ dense_3 (Dense)                   │ (None, 2)                │              22 │
└───────────────────────────────────┴──────────────────────────┴─────────────────┘
 Total params: 197,050 (769.73 KB)
 Trainable params: 197,050 (769.73 KB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/60
2024-06-17 19:00:04.850730: I
tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:117] Plugin optimizer
for device_type GPU is enabled.
1340/1340 ━━━━━━━━━━━━━━━━━━━━ 113s 83ms/step - loss: 0.1388 - val_loss: 0.0624
Epoch 2/60
1340/1340 ━━━━━━━━━━━━━━━━━━━━ 116s 86ms/step - loss: 0.0659 - val_loss: 0.0535
Epoch 3/60
1340/1340 ━━━━━━━━━━━━━━━━━━━━ 114s 85ms/step - loss: 0.0553 - val_loss: 0.0494
Epoch 4/60
1340/1340 ━━━━━━━━━━━━━━━━━━━━ 116s 86ms/step - loss: 0.0471 - val_loss: 0.0457
.
.
. [LINES REMOVED]
.
.
.
Epoch 28/60
1340/1340 ━━━━━━━━━━━━━━━━━━━━ 118s 88ms/step - loss: 0.0165 - val_loss: 0.0243
Epoch 29/60
1340/1340 ━━━━━━━━━━━━━━━━━━━━ 117s 87ms/step - loss: 0.0160 - val_loss: 0.0325
Epoch 30/60
1340/1340 ━━━━━━━━━━━━━━━━━━━━ 117s 87ms/step - loss: 0.0184 - val_loss: 0.0231
Epoch 31/60
1340/1340 ━━━━━━━━━━━━━━━━━━━━ 119s 88ms/step - loss: 0.0155 - val_loss: 0.0234
Saved artifact at 'self_drive_model'. The following endpoints are available:

* Endpoint 'serve'
  args_0 (POSITIONAL_ONLY): TensorSpec(shape=(None, 120, 160, 1), dtype=tf.float32,
name='keras_tensor')
Output Type:
  TensorSpec(shape=(None, 2), dtype=tf.float32, name=None)
Captures:
  6312487648: TensorSpec(shape=(), dtype=tf.resource, name=None)
  6312487824: TensorSpec(shape=(), dtype=tf.resource, name=None)
.
.
. [LINES REMOVED]
.
.
  6312489584: TensorSpec(shape=(), dtype=tf.resource, name=None)
  6312492576: TensorSpec(shape=(), dtype=tf.resource, name=None)
Model traning done
335/335 ━━━━━━━━━━━━━━━━━━━━ 22s 67ms/step - loss: 0.0226
Test loss: 0.023124229162931442
Converting...
.
.
. [LINES REMOVED]
.
.
---------------------------------------------------------------
Your TFLite model has '1' signature_def(s).

Signature#0 key: 'serving_default'
- Subgraph: Subgraph#0
- Inputs:
    'keras_tensor' : T#0
```



```
- Outputs:
    'output_0' : T#31

---------------------------------------------------------------
              Model size:      209488 bytes
    Non-data buffer size:        7760 bytes (03.70 %)
  Total data buffer size:      201728 bytes (96.30 %)
    (Zero value buffers):           0 bytes (00.00 %)

* Buffers of TFLite model are mostly used for constant tensors.
  And zero value buffers are buffers filled with zeros.
  Non-data buffers area are used to store operators, subgraphs and etc.
  You can find more details from
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/schema/schema.fbs

Conversion complete
```

pi/progs/pilot1 Module

This program is written in Python and takes a trained model as input and runs in master mode sending commands to the PWM module. It is the self driver. It can either capture images live from the Raspberry Pi camera or can be provided a folder of images for simulated driving. It shows the current video frame and useful data.

It uses the following libraries:

- TensorFlow – For inferences from the model provided as input to the program
- PiCamera2 – For getting image frames from the Raspberry Pi camera.
- PIL – For frame processing like conversion to grayscale.
- CV2 – Image utilities. Uses for such purposes as image pixel format conversion and displaying the image overlaid with telemetry information.
- Numpy – For numerical methods.
- Adafruit_PWMServoDriver – To control the PWM module as I2C master mode.

The following is the overall flow of the program:

- Initializes the servo driver and centres the ESC and servo

- Parses command line arguments for trained model path to use and optional folder to iterate input images from if simulating rather than self-driving.
- It initializes the TensorFlow Lite model provided in the command line arguments.
- It extracts the shape information from the model dynamically.
- Either:
    - Iterates through folder specified in command line argument and processes them calling the drive_frame function
    - Or captures frames from the Raspberry Pi camera converts them into grayscale and processes them calling the drive_frame function
- drive_frame function
    - Resizes the grayscale image to the dimensions of the loaded model.
    - Normalizes the image pixels of 0-255 to 0-1.
    - Expands dimensions to match the model.
    - Uses the supplied model to get inference from the image frame.
    - Convert the output to throttle and steering values by denormalizing to -100 to 100.
    - Display the current image in a window for debugging and evaluation purposes.
    - Draw steering and throttle values and graphics.
    - Draw time taken to infer in last 10 frames on the window.
    - Use I2C to control 2 channels of the PWM module.

Data Collection Instrument

Quantitive evaluation of models trained on different augmented images and combinations of augmentation types was done on an Apple MacBook Air M1 by calculating the mean square error at each epoch while training the Keras Sequential model in the TensorFlow package. At the end of a model training, the validation set was evaluated again for the trained model using the below code.

```
test_loss = model.evaluate(val_gen)

print(f'Test loss: {test_loss}')
```

When evaluating the model with experiments on the test track depicted in Figure 3.5.b, observations on the behaviour of the self-driving car depicted in Figure 3.4.b. on each numbered segment was noted and observation was used as a clerical tool. The car was run on the track on multiple occasions for each model with videos being created for reference and comparison with accompanying commentary. These videos were meticulously reviewed and an overall comparative driving performance empirical score was given from a scale of 1-5 with 5 being the highest.

As mentioned on the sections earlier, training data was collected by the RC car platform as an instrument as described in Figure 3.7. This includes the Raspberry Pi with a camera attached which stored the image with control and sensor data on the solid-state drive.

Sample Size

Table 1

Description of training samples collected and used for model training. The training primary data was collected with the following characteristics.

Date & Time	Environmental Conditions	Image Samples	Control Samples
2024-05-14 10:09:15	Mid brightness, grey hue	1875	2950
2024-05-14 10:15:32	Mid brightness, grey hue, shiny floor	562	1225
2024-05-19 10:16:09	Mid brightness, grey hue, shiny floor	705	1240
2024-05-19 10:41:25	High brightness, grey hue, reflections	4132	5100
2024-05-19 12:37:04	High brightness, grey hue, glare	1586	1900
2024-05-26 15:50:34	High brightness, blue-grey hue, glare	3600	1326
2024-05-26 15:53:20	High brightness, blue-grey hue, glare	5553	1428
2024-05-27 17:00:29	Mid brightness, grey hue, lights	2735	1530
2024-05-27 17:03:51	Mid brightness, grey hue, lights	1542	816
2024-05-27 17:09:28	Low brightness, purple hue, lights	3090	1836
2024-05-27 17:13:37	V. low brightness, purple hue, lights	2288	1224
2024-05-28 09:48:43	Low brightness, blue hue	5037	1428
2024-05-28 09:51:40	Mid brightness, blue-grey hue	9121	2244
2024-05-28 14:30:32	High brightness, blue hue, sunlight	2443	724
2024-05-28 14:33:04	High brightness, blue-gray hue, sunlight	4469	1112
2024-05-28 14:37:42	High brightness, blue-gray hue, sunlight	5064	1428

Table 1 shows that there is a difference between the control samples and the image samples. This is because (1) image data generated in driving sessions were cleansed for removing less than ideal driving patterns (2) One control sample can cover more than one image sample as at a maximum, the control inputs updated at 10 Hz while the images were captured in more than 20 frames per second (FPS).

For model robustness measurement data, while training 10 models trained with different combination of image augmentation techniques, loss curves were extracted. The training data and validation data split was kept at 80:20. Note that when augmented images are added to the training images, the split will change however the number of validation samples remain the same. The maximum epoch was kept at 60 and most of the training runs completed in about 25-40 epochs with the configured early stoppage callback. The image and labelled data was provided in batches of 32.

Sampling Technique

For selection of the images for training and validation, random sampling was used. This was done with the intention of balancing the training and validation data-set. As shown in Table 1, training data was obtained under different environmental conditions which included varying levels of brightness, hues, glares, sunlight etc. This helps the model generalize better and prevents overfitting to a specific condition or environmental factor thus makes the model perform on a variety of real-world conditions.

Data Analysis Tool

PyPlot was used from Matplotlib for visualizing the loss curve as shown in the below cude.

```
import matplotlib.pyplot as plt

.
.
.
history = model.fit(train_gen, epochs=EPOCHS, validation_data=val_gen,
callbacks=[early_stopping])


# plot traning and valuation loss values

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(loc='upper right')
plt.show()
```

For model performance comparison graphs, LibreOffice Calc spreadsheets were used.

Chapter 4: Data Analysis, Results and Interpretation

Experiment 1: Model trained without any augmentation

Description: No augmentation was used and only the training data and thus the training image count is the lowest amongst all the experements.

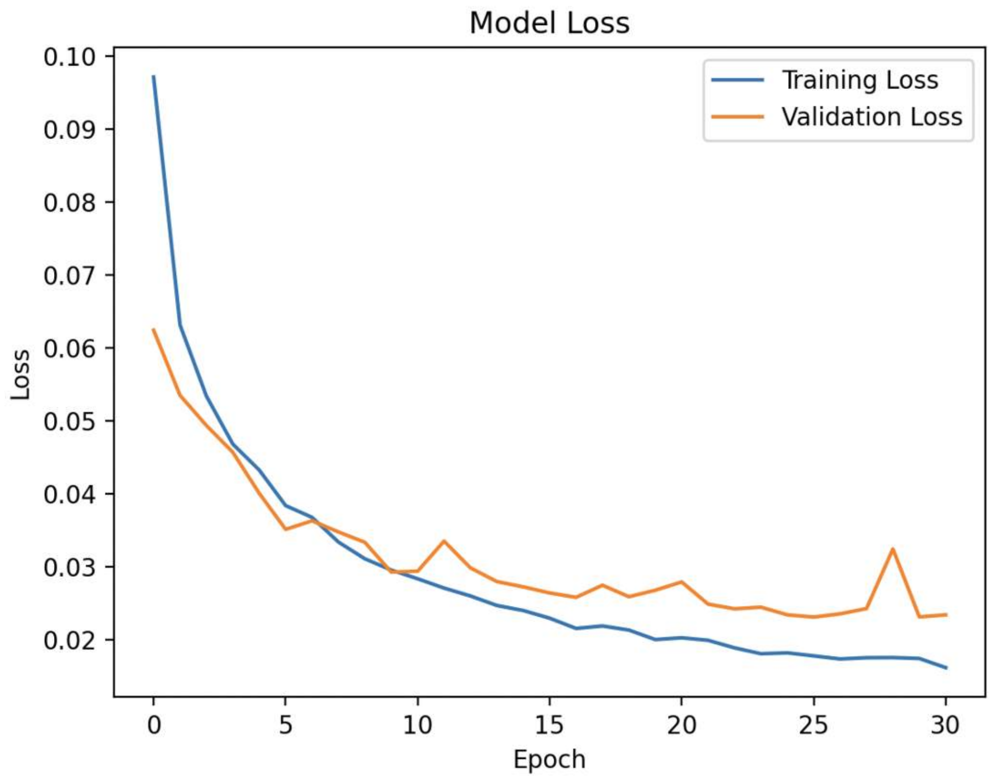


Figure 4.1: Loss curve of model trained without any augmentation

Self-drive observations: With this model, the self driving car pilot program inferring the outputs, turns wide on curve 1 (see Figure 3.5.b) but then recovered and drove through sections 3, 4, 5 and 6 well. On section 7, it oversteered and did not pass intersection 8-9 but rather turns right and hit the wall. When given a hand, it continued to steer right until helped by reseting to 9, continues and finishes the track. There was also a tendency noticed to turn right directly from 7-11. Overall, it was noticed that on the track, self-driving using this model performed fairly but made non-trivial mistakes.

Self-driving performance score: 2/5

Training image count: 42875

Validation image count: 10719

Total batches: 1340

Lowest epoch: 30/60 – 117 seconds - loss: 0.0184 - val_loss: 0.0231

Final epoch: 31/60 – 119 seconds - loss: 0.0155 - val_loss: 0.0234

Model test loss: 0.023124229162931442

Interpretation: As seen in the loss curve in Figure 4.1, there was some tendency to overfit in this model where the training loss is quite lower than the validation loss at the end of training. This explains the inability to generalize onto a real driving scenario however due to an average model test loss, the car is still able to achieve self driving behaviour for most of the track sections.

Experiment 2: Model trained with flip augmentation

Description: To train this model the image was flipped and the label data was modified in a way to multiply the steering inputs by -1.

Self-drive observations: With this model, the self driving car pilot program inferring the outputs, the car drove well from section 1 (see Figure 3.5.b) upto 7 where it did a slight oversteer going wide but recovered at 9 crossing the intersection. It did well up to 13 where it had a tendency to understeer. But it negotiated the track quite well in comparison to no augmentation. It also did fairly well when it was allowed to run in the reverse direction but not as good as the default direction.

Self-driving performance score: 3/5

Training image count: 85750

Validation image count: 10719

Lowest epoch: 23/60 – 149 seconds - loss: 0.0207 - val_loss: 0.0196

Final epoch: 28/60 – 152 seconds - loss: 0.0201 - val_loss: 0.0204

Model test loss: 0.019589142873883247

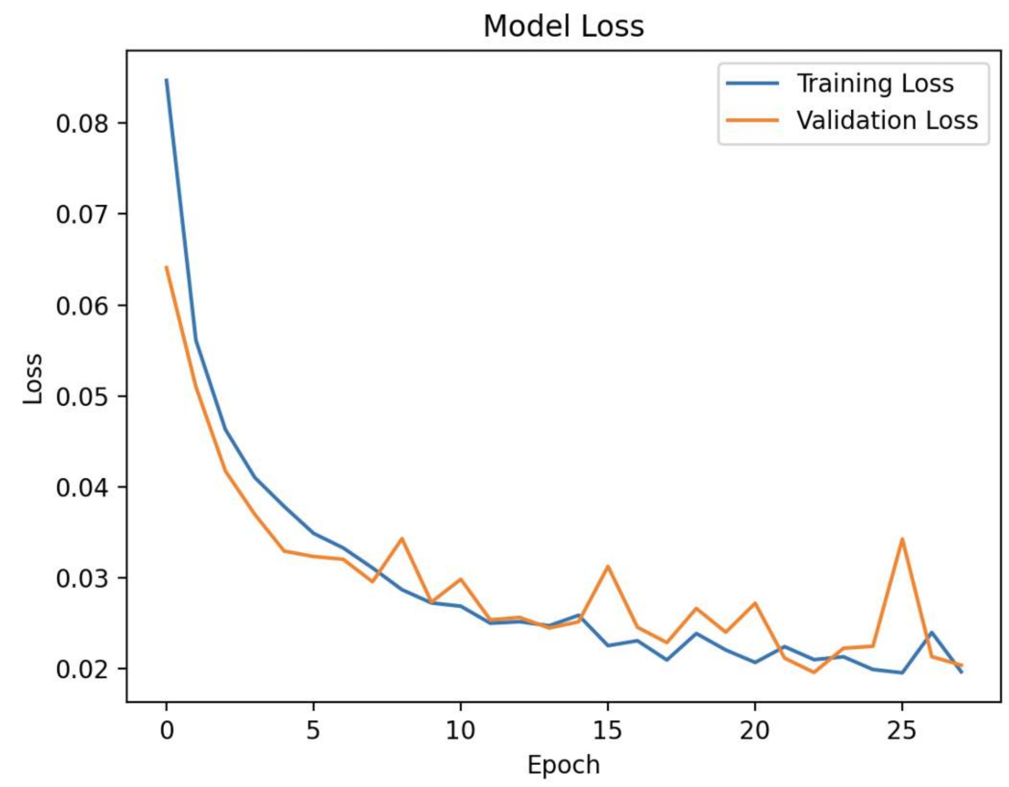


Figure 4.2: Loss curve of model trained with flip augmentation

Interpretation: As observed from Figure 4.2, the addition of flipped images made the model fit well with the training and validation loss very close and quite low test loss (0.0196).

This was reflected in better driving performance with the majority of the track negotiated successfully.

Experiment 3: Model trained with brightness augmentation

Description: To train this model, augmented images with random brightness values of 0.5 to 1.5 of the original image were used in addition to the training data set.

Self-drive observations: The car ran well from track section 1 (see Figure 3.5.b) till section 6 where it turned right onto section 14. When reset again at 7, it crashed at the wall near 8. When reset again, went straight from 11. It did well from 8 to 15 continuing from 1 to 6. It did not do well when reversed. In net, less than half the track could be negotiated well but these were different sections from the flip augmentation model.

Self-driving performance score: 2.5/5

Training image count: 85750

Validation image count: 10719

Lowest epoch: 28/60 – 151 seconds - loss: 0.0164 - val_loss: 0.0192

Final epoch: 33/60 – 152 seconds - loss: 0.0155 - val_loss: 0.0205

Model test loss: 0.01916453242301941

Interpretation: As seen in Figure 4.2, the model shows some overfitting as the training loss is noticeably lower than the training loss with quite low test loss (0.0191). This reflects in the driving performance. However, a point to note is that it negotiates the bottom half of the track well which is a different region from where the flip augmentation performed better.

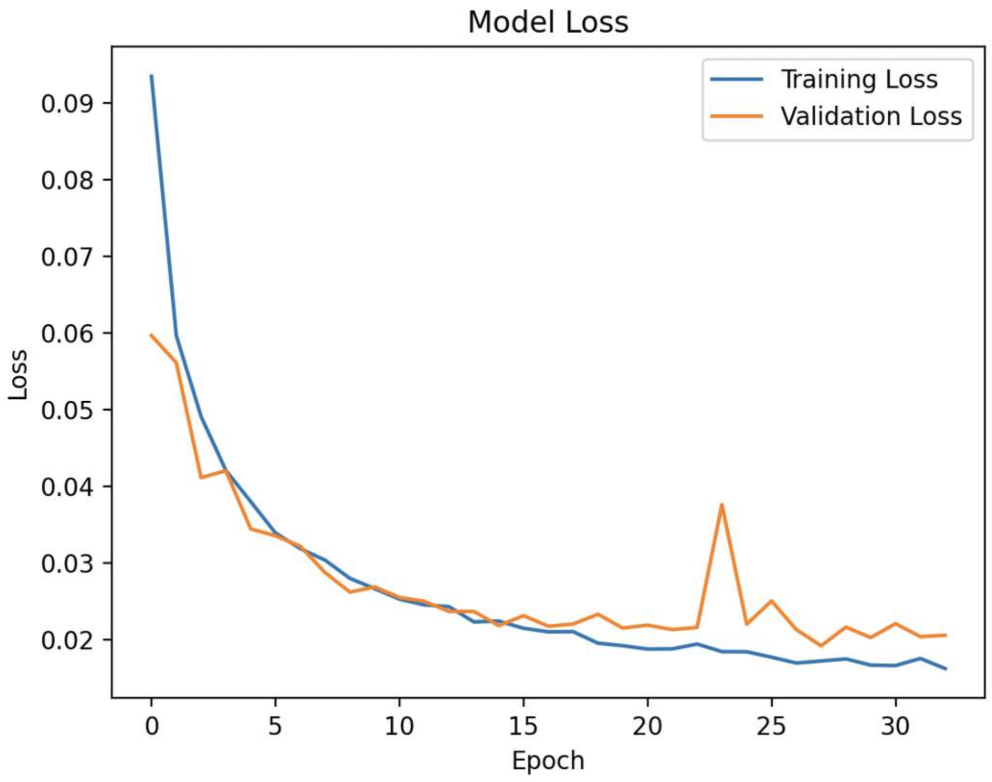


Figure 4.3: Loss curve of model trained with brightness augmentation

Experiment 4: Model trained with contrast image augmentation

<u>Description:</u> While training this model, duplicates of the training image were applied contrast enhancement from 0.5 to 1.5.

<u>Training image count:</u> 85750

<u>Validation image count:</u> 10719

<u>Lowest epoch:</u> 28/60 – 141 seconds - loss: 0.0176 - val_loss: 0.0195

<u>Final epoch:</u> 30/60 – 141 seconds - loss: 0.0176 - val_loss: 0.0236

<u>Model test loss:</u> 0.01941482536494732

<u>Self-driving observations:</u> The car had difficulty in maintaining the straight section 2 (see Figure 3.5.b) and it would go straight between 3 and 4 thus loosing the track. Recovery was also seen to be bad. It had the tendency to miss curve 6. The steering was jittery with too much steering adjustment without consistency in steering while negotiating the curves. There was no significant self-driving behaviour noticed and the experiment was abondoned.

<u>Self-driving performance score:</u> 1/5

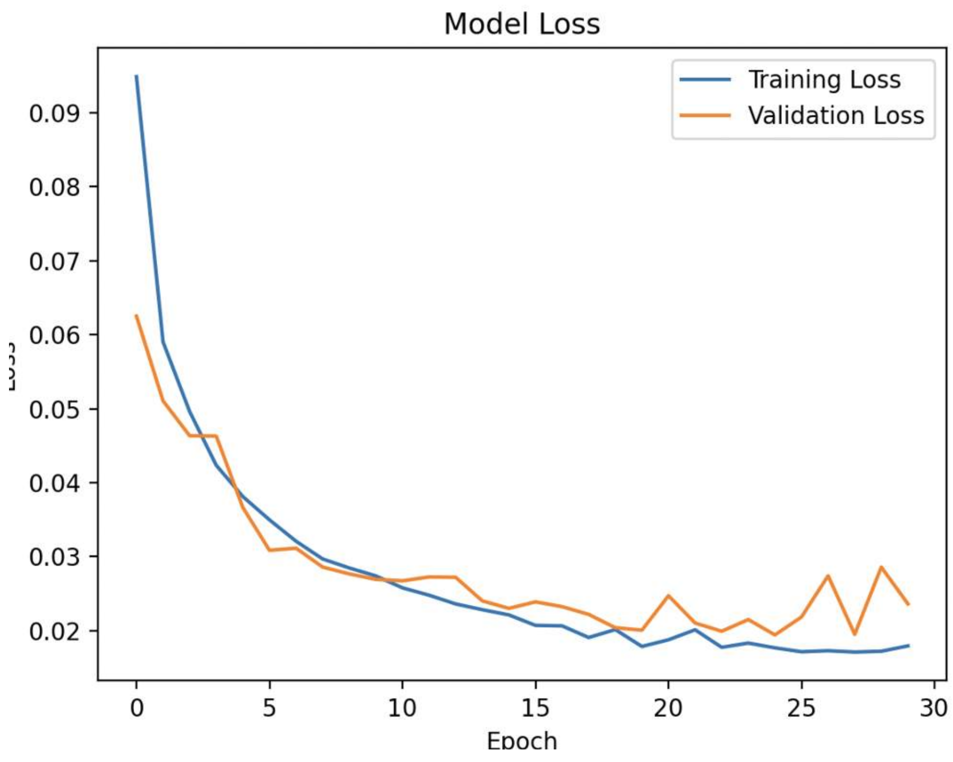


Figure 4.4: Loss curve of model trained with contrast augmentation

<u>Interpretation:</u> From the loss curve in Figure 4.4, it can be seen that there is significant overfitting even though the lowest validation loss was quite low (0.0195). This reflects while self driving as this model provided bad performance without being able to achieve self-driving behaviour for a reasonable length of time. It could be noted that the contrast values may work in a smaller range than the one provided but this was not attempted.

Experiment 5: Model trained with image invert augmentation

Description: When training this model, the image was inverted such that white pixels of the grayscale image would become black and vice-versa. This inverted image was added to the training data along with the original.

Training image count: 85750

Validation image count: 10719

Lowest epoch: 36/60 – 153 seconds - loss: 0.0191 - val_loss: 0.0178

Final epoch: 41/60 – 139 seconds - loss: 0.0211 - val_loss: 0.0199

Model test loss: 0.017778560519218445

Self-drive observations: The car ran well from track section 1 to 8 (see Figure 3.5.b). From 8, it had a tendency to go straight and not proceed on the curve to 9. However, when reset at 8, it negotiated the intersection and continued to the next track run. Thus it negotiated most of the track well however failed at a different section than the other models. It did not exhibit any notable self-driving behaviour when on the track in the reverse direction.

Self-driving performance score: 2.5/5

Interpretation: As seen in Figure 4.5, the model training was jittery but it showed a good fit. However, the loss curve is unusual with many peaks and complete reversal of training per epoch. Self driving behaviour was reasonably good but not consistent. It is arguable that inverse does not allow for proper generalization by the CNN filters even though the model test loss was noticeably lower than other models (0.0178).

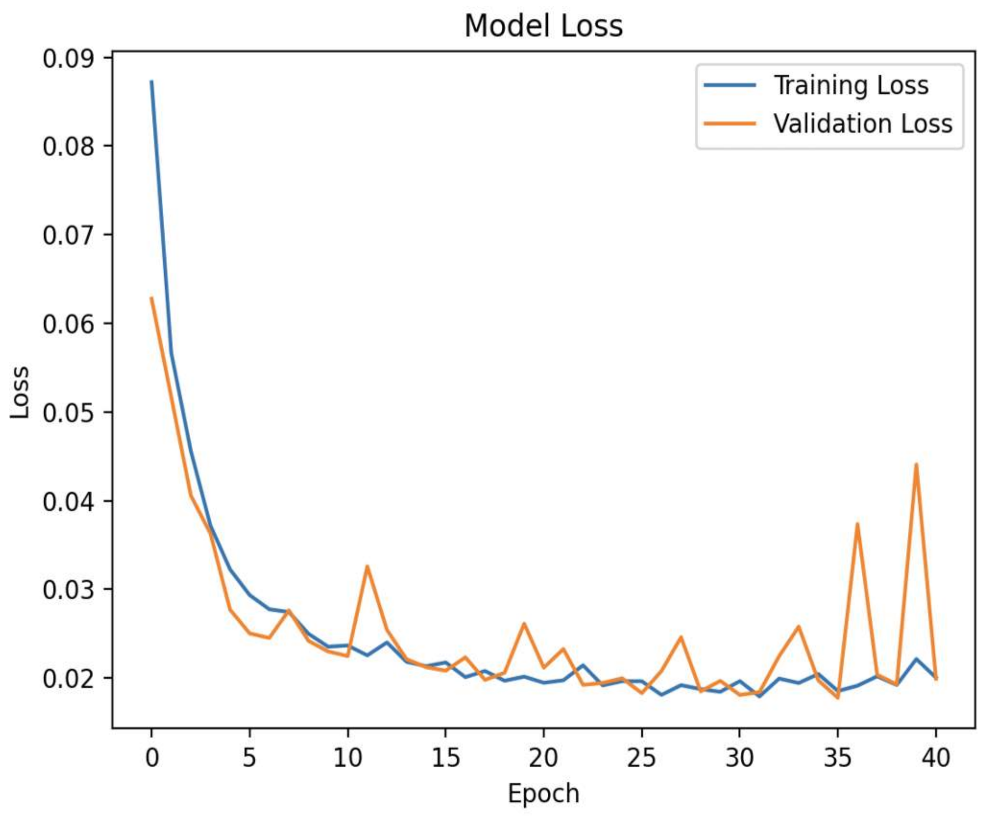


Figure 4.5: Loss curve of model trained with invert augmentation

Experiment 6: Model trained with image flip augmentation combined with invert augmentation

Description: This model was trained with two augmented images with each original image. Image flip and invert were both applied to this model to test for the effects when augmentation techniques are combined when training models for self-driving.

Training image count: 128625

Validation image count: 10719

Lowest epoch: 21/60 - 178 seconds - loss: 0.0249 - val_loss: 0.0194

Final epoch: 26/60 – 180 seconds - loss: 0.0213 - val_loss: 0.0201

Model test loss: 0.019417800000925064

Self-driving observations: The car ran well from track sections 1 to 6 (see Figure 3.5.b) where it had a tendency to continue straight for most of the runs. Continuing straight at section 7 was also a problem however it negotiated 8-13 well covering the intersection well. However, it had further problems at cornering at 14-15. When negotiating the track in the non-default direction, the performance was similar. Overall, the car showed below average self driving performance.

Self-driving performance score: 3/5

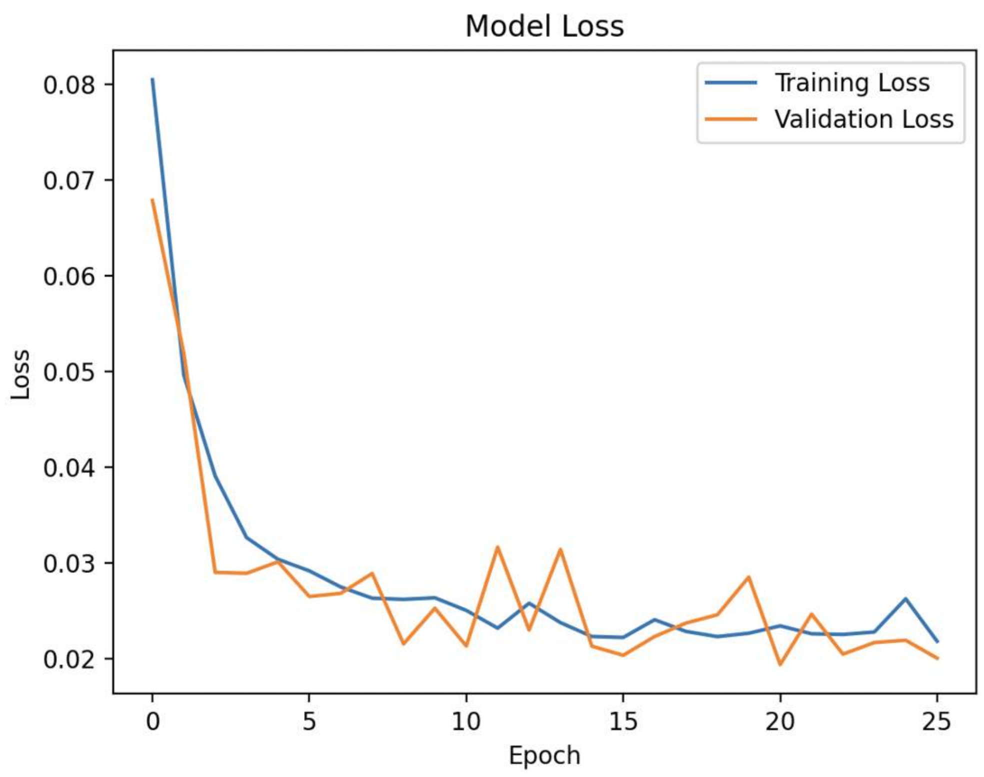


Figure 4.6: Loss curve of model trained with flip and invert augmentation

Interpretation: As seen in Figure 4.6, the validation loss ended up noticeably lower (0.0194) than the training loss. This points to a strong regularization effect where the model may not have captured the features essential for self-driving on a real-world track. Also since the validation data was selected at a random from the same set, different environmental conditions of self-driving could cause improper inference. Also, invert individually also did not lead to much better performance so in combination could have an amplification effect deteriorating effectiveness further. Dropout rates could be reduced in the future for evaluating such models in an attempt to improve the model's performance.

Experiment 7: Model trained with image blur augmentation

Description: This model was trained with images augmented with a gaussian blur filter of 0.25 to 1 radius.

Training image count: 85750

Validation image count: 10719

Lowest epoch: 29/60 – 139 seconds, loss: 0.0191 - val_loss: 0.0214

Final epoch: 34/60 – 148 seconds - loss: 0.0185 - val_loss: 0.0229

Model test loss: 0.02144046127796173

Self-driving observations: The car showed good driving characteristics from section 1 (see Figure 3.5.b) to 7. Further, it turned left from section 7 but continued in the wrong direction, drove on the wrong direction up to 6 where it recovered in the right direction. This was consistent and robust in it's behaviour. It had a strong tenancy to stay on track and recover however not always in the default direction.

Self-driving performance score: 3.5/5

Interpretation: As seen in figure 4.7, there is some overfitting observed and there the model test loss is a little on the higher side when compared to the other models (0.0214). However, the car shows good driving behaviour on-track. It is possible that the blur augmentation works because the filters were trained to ignore the grid pattern created by the tiles on the floor which would be less prominent after the application of the blur.

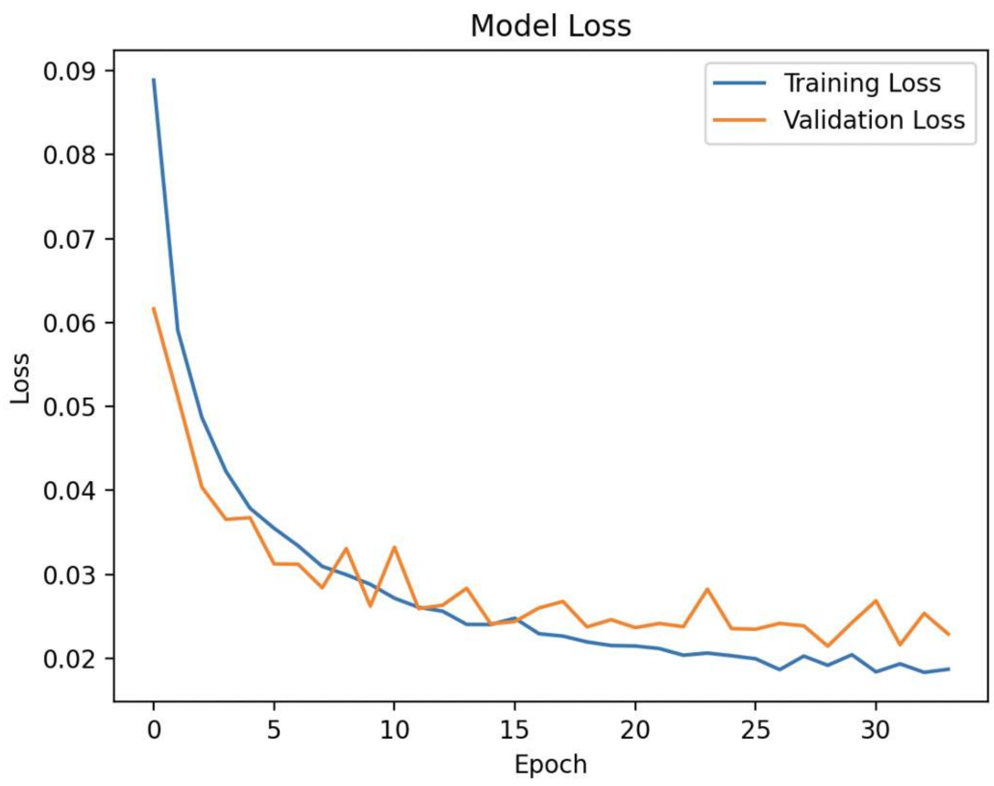


Figure 4.7: Loss curve of model trained with blur augmentation

Experiment 8 – Model trained with image flip combined with brightness augmentation

Description: This model was trained with augmented images that were flipped and the label data was modified in a way to multiply the steering inputs by -1 and also additional images with with random brightness values of 0.5 to 1.5 of the original image in addition to the training data set.

Training image count: 128625

Validation image count: 10719

Lowest epoch: 19/60- 189 seconds - loss: 0.0224 - val_loss: 0.0208

Final epoch: 24/60 – 197 seconds - loss: 0.0216 - val_loss: 0.0212

Model test loss: 0.020775340497493744

Self-driving observations: The car ran well from section 1 to 14 (see Figure 3.5.b) which was observed on multiple runs. The car had good driving and recovery characteristics on all except one section of the track at 14-15 where it had the tendency to go straight failing to negotiate the curve. Overall this model seemed to perform the best on-track compared to the other models. Even when tried on the non-default direction, the performance was better than average.

Self-driving performance score: 4/5

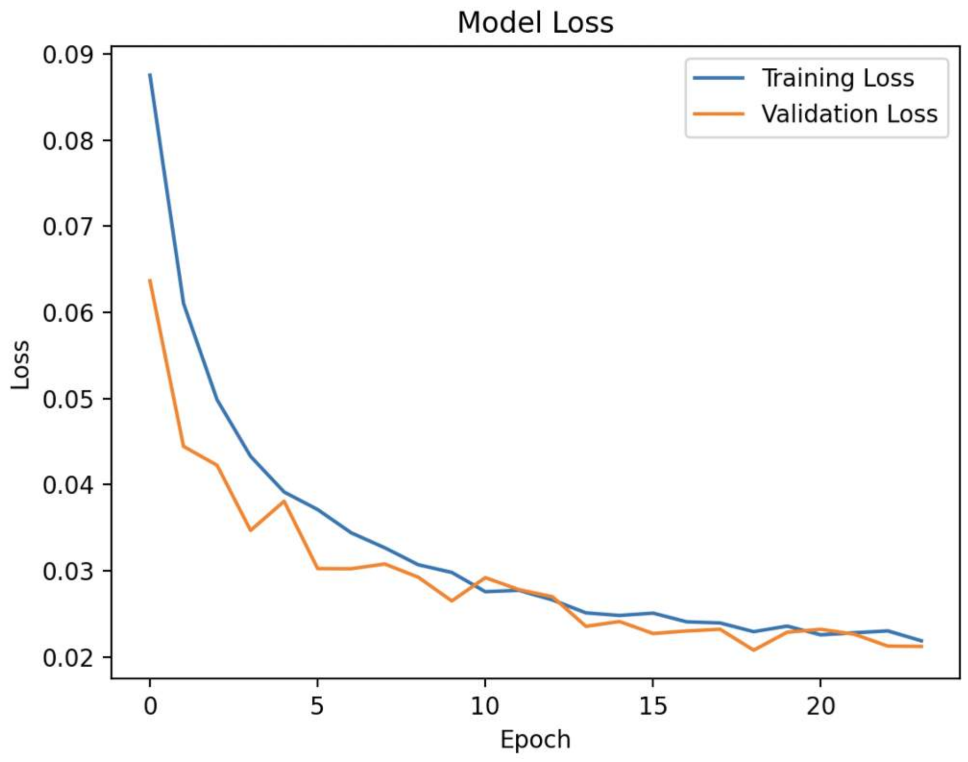


Figure 4.8: Loss curve of model trained with flip and brightness augmentation

Interpretation: A very good fit is observed in the loss curve in Figure 4.8 which is reflected in on-track performance. This model seemed to be having the best real-world performance thus far. Individually, these two augmentations performed well and with the low validation loss score, thus in combination, they seemed to give the model better generalization capabilities.

Chapter 5: Findings and Conclusion

Comparisons of Trained Models

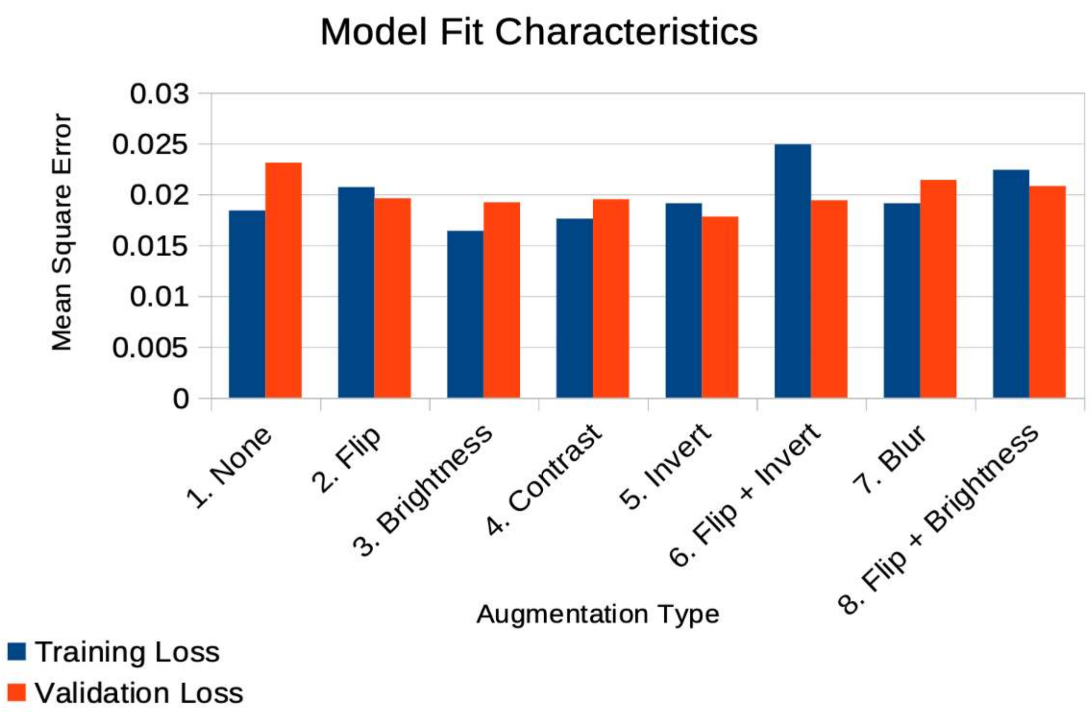


Figure 5.1: Comparison of model fit characteristics

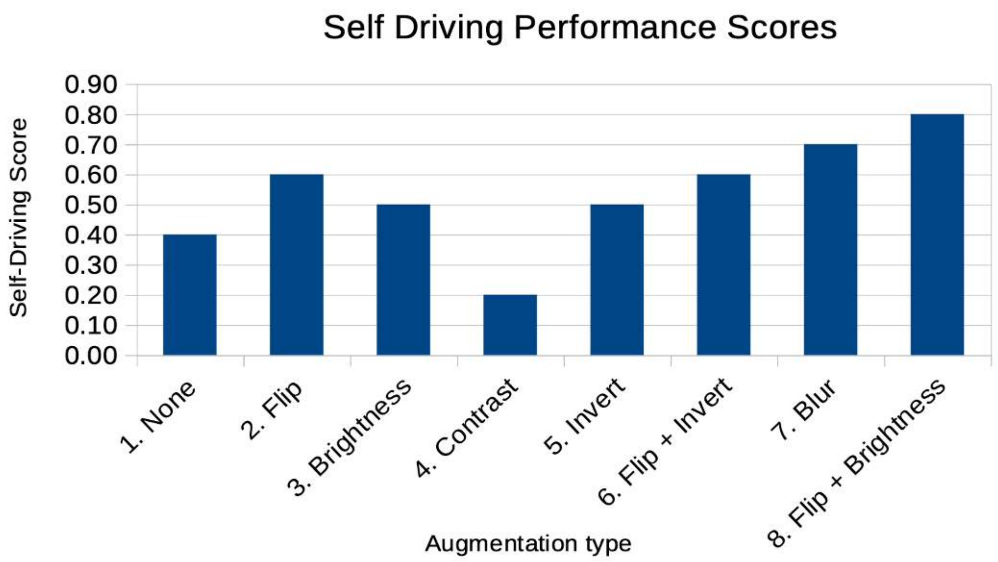


Figure 5.2: Comparison of self-driving performance scores

Findings

1. There was significant performance improvement, quantitatively and empirically, in models trained with image augmentation than models trained without such augmentation for the autonomous driving vehicle. Overfitting in augmented models was lower than the model without augmentation and flip and flip+brightness combined provided better fit.
2. Models trained with blur and a combination of flip and brightness performed better in terms of model fit and real-world driving performance compared to other augmentation techniques. Flip and a combination of flip and invert provided average performance. Contrast augmentation faired the worse than no augmentation.
3. It was observed that when the difference in training loss and validation loss was low, it reflected in real on-track self-driving performance. Overall, image augmentation in particular combinations performed best in real-world scenarios.

Conclusion

The project successfully demonstrated that augmentation of image training data is beneficial in training of neural networks used to control autonomous vehicles. We achieved accurate self-driving behaviour with a smaller dataset and thus reduced the time and effort required for data collection. The findings highlight the importance of using image augmentation techniques to train models capable of better performance in real-world scenarios that may differ from conditions at the time of training data collection. Thus we reject the null hypothesis (H0) and accept the alternative hypothesis H1. Future work could explore enhancement of the architecture of the neural network and other image augmentation techniques.

Chapter 6: Recommendations and Limitations of the Study

Following are the recommendations of the study:

- Image augmentation should be used when training neural network models for more robust models with improved on-track performance.
- Flip, brightness and blur augmentation should be considered out of the many types of augmentations possible.
- Trial of image augmentation techniques on life-sized vehicles must be done only with proper safety risk mitigation on a sample track.
- Application of more image augmentation techniques, for example, ones that mimic sunlight and shadows.
- Improvement of dataset of training samples by more variations and on different test and validation tracks.
- Training can be done on both directions of sample tracks to provide a more generalized model.
- Models with good fit were found to perform better in real-world track conditions and thus overfitting should be avoided in models for autonomous vehicles.
- Utilization of different neural network designs and comparison of performance of different models. Deeper networks may provide better generalization.

Following are the limitations of the study:

- Experiments were performed on a land-based autonomous vehicle only and on a limited sample track on a 15×15 feet space.
- We use a scaled-down vehicle platform based on a hobby grade RC car. This may not simulate real-world driving conditions when translated into a full-scale land vehicle.
- Only one type of CNN based neural network model design was used in the study. The DonkeyCar project offers many different and deeper CNN based models. Performance on such larger and more varied models was not tested.
- Not all image augmentations or their combinations were tested. Also, different value ranges for filters like brightness and contrast were not used.
- The training dataset collected was not significantly large and limited in terms of driving variation.

Bibliography

1. Dean Pomerleau. ALVINN: An autonomous land vehicle in a neural network. In Neural Information Processing Systems (NIPS), 1988. Retrieved from https://proceedings.neurips.cc/paper/1988/file/812b4ba287f5ee0bc9d43bbf5bbe87fb-Paper.pdf

2. Chen, Z.; Huang, X. (2017): End-to-end learning for lane keeping of self-driving cars. IEEE Intelligent Vehicles Symposium, pp. 1856-1860. Retrieved from https://users.wpi.edu/~xhuang/pubs/2017_chen_iv.pdf

3. Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. Journal of Big Data, 6(1), 60. https://doi.org/10.1186/s40537-019-0197-0

4. Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., . . . Zieba, K. (2016). *End to End Learning for Self-Driving Cars*. Nvidia. Retrieved from https://arxiv.org/abs/1604.07316

5. Fernández-Llorca, D., Sotelo, M. A., Muñoz-Bulnes, J., Fernandez, C., & Parra, I. (2017). Deep fully convolutional networks with random data augmentation for enhanced generalization in road detection. In Proceedings of the IEEE Intelligent Transportation Systems Conference (ITSC). DOI: 10.1109/ITSC.2017.8317901. Retrieved from https://www.researchgate.net/publication/323789679_Deep_fully_convolutional_networks_with_random_data_augmentation_for_enhanced_generalization_in_road_detection

6. Mariusz Bojarski, Philip Yeres, Anna Choromanska, Krzysztof Choromanski, Bernhard Firner, Lawrence Jackel, and Urs Muller. Explaining how a deep neural network trained with end-to-end learning steers a car. arXiv preprint arXiv:1704.07911, 2017. Retrieved from https://arxiv.org/pdf/1704.07911

7. Kiaan Upamanyu, Indukala P. Ramaswamy (2021) - *Effects of Image Augmentation on Efficiency of a Convolutional Neural Network of a Self-Driving Car.* Journal of Student Research, Greenwood High International School, Bangalore. Retrieved from https://www.jsr.org/hs/index.php/path/article/view/1486

8. DonkeyCar Platform - https://www.donkeycar.com/ & https://github.com/autorope/donkeycar