UNIVERSITY OF LEICESTER

SCHOOL OF ENGINEERING

Evaluating the performance of end to end deep learning models in a custom car driving simulator

Module code:	**EG3005/3008**
Module name:	Individual Project/Engineering Management

06/05/2021

Author(s):	**Morgan Colling**
Student ID(s):	**189046533**
Degree:	**Meng General Engineering**
Tutor/Project supervisor:	**Dr Avinash Bhangaonkar**

Summary

This report documents the creation of a custom car driving simulator in the Godot game engine and evaluates the performance of convolutional neural networks (CNNs) within the simulator. The CNN models were trained using an image of the environment in front of the car and the corresponding steering angle; images and angles used were gathered from the simulator. The model performance was evaluated by calculating an autonomy value which represented the average time a model could control the simulated car without needing to be corrected by the driver. This report found that trained CNN models can perform tasks such as lane detection, road tracking and obstacle detection using a small dataset. It was also found that increasing the dataset size resulted in an increased performance on those tasks. This was evidenced by the increased performance of Model V2 compared to Model V1 (0.0854 interferences per second compared to 0.1135, respectively).

Contents

1 Introduction

1.1 Deep Learning

Deep learning (DL) can be defined as neural networks (NNs) with a large number of parameters that include layers in one of the four fundamental network architectures: unsupervised pretrained networks, convolutional neural networks (CNNs), recurrent neural networks and recursive neural networks (Patterson and Gibson, 2017). DL allows computational models to learn representations of data with multiple levels of abstraction; developments within the field of deep CNNs have caused breakthroughs in image, video and speech and audio processing (Lecun, Bengio and Hinton, 2015).

1.2 Convolutional Neural Networks

1.2.1 CNN Architecture

CNNs are one of the most popular deep learning neural networks. They consist of multiple layers; the layers being a combination of convolutional, fully-connected, non-linearity and pooling layers. CNNs perform best in applications that use image data (Albawi, Mohammed and Al-Zawi, 2017). To ensure that CNNs can produce good results, it is important to use a large, high quality training data set (Simard, Steinkraus, and Platt, 2003). In this project, the training data was collected from a custom-made car driving simulation (made for the purpose of this study) so that the quality can be controlled.

1.2.2 Convolutional Layers

Convolutional layers use parameters that are designed around the use of learnable kernels. The layer convolves each kernel across the input to the layer, the scalar product for each value in the kernel is calculated. From this, the network will learn which kernels activate when a feature is seen at a specific spatial position of the input (O'Shea et al, 2015). A visual representation of the layer input and kernel are shown in Figure 1. For 3D inputs such as a coloured image, the kernel convolves across each 2D colour channel matrix.

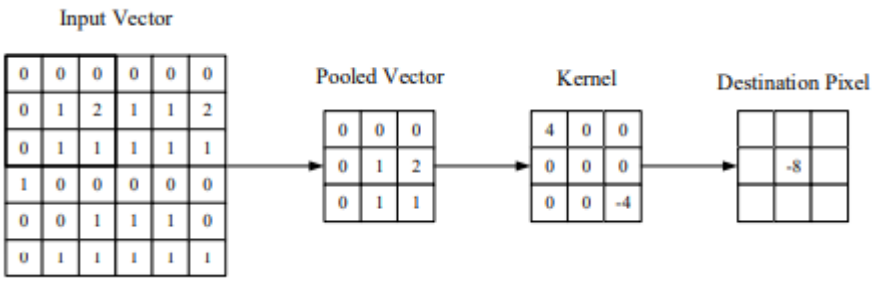


Figure 1 – Visual representation of the input vector, pooled layer, kernel and destination pixel (O'Shea et al, 2015).

Convolutional layers use activation functions to map the calculated features for input into subsequent layers. The activation function used is normally nonlinear, an example would be the Rectified linear unit function known as ReLU (Lin and Shen, 2018).

1.2.3 Activation Functions

Activation functions are used in neural networks to control network or layer outputs within neural networks. They do this by computing the weighted sum or the input and biases to a node. The value calculated using the activation function decides of a neuron will be fired or not (Nwankpa et al, 2018). The activation function used for the nodes in this project is the exponential linear unit (ELU) function. ELUs speed up learning in deep neural networks and can lead to higher classification accuracies. The increased speed is a result of having negative values which allows the mean unit activations to become closer to zero. As the mean shifts towards zero, the normal gradient becomes closer to the unit natural gradient which speeds up learning due to a reduction in the bias shift effect (Clevert, Unterthiner, and Hochreiter, 2015).

The equation for the ELU activation function is given in equation 1.

$$f(x) = \begin{cases} x & if\ x > 0 \\ \alpha(\exp(x) - 1) & if\ x \leq 0 \end{cases} \quad (1)$$

Where α is the ELU hyperparameter that controls the saturation value of the function for negative net inputs and x is the sum of the inputs and biases from the node

Saturation means the derivative is small and therefore the propagation to the next layer in the network is decreased (Clevert, Unterthiner, and Hochreiter, 2015).

1.3 Training Models

To train neural network models the weights and biases need to be optimised to allow the model to give the desired output. Optimization is done using a loss function, a loss function quantifies how bad the performance of the model is (Zafar et al., 2018). The loss function used to train the models used in this project is a mean squared error function (MSE). The equation for the loss function used is given by equation 2.

$$loss = \frac{1}{n} \sum_{i=1}^{n} (y_{i,true} - y_{i,pred}) \quad (2)$$

Where loss is the measure of error of the model with its current weightings and biases; n is the number of data points, $y_{i,true}$ is the correct value the model should output when shown the data point i and $y_{i,pred}$ is the value predicted by the model when shown the data point i.

1.4 NVIDIA

This project was inspired by the End to End learning for Self-Driving Cars paper by NVIDIA. In the paper they were able to train a convolutional neural network (CNN) using real world images from three front facing cameras to output steering commands. The system they created automatically learns internal representations of the necessary processing steps (for example detecting useful road features) only using the human steering angle as the training signal (Borjarski et al., 2016). The following project aims to apply the techniques used within the NVIDIA paper to recreate a version of their CNN model and use it to drive a car within a custom-designed car driving simulator.

1.5 Car Driving Simulators

The use of simulation systems in the development and validation of self-driving car technologies are essential. The most common method of generating a simulator is to use a combination of computer graphics, physics-based modelling, and robot motion planning techniques together to create an artificial environment where moving vehicles can be animated as well as rendered. Some example simulators would be NVIDIA's Drive Constellation and Intel's CARLA (Li et al., 2019). The current game engines available to code a custom driving simulator would be Unity, Unreal Engine and Godot. The game engine used for this project is the open-source game engine Godot. This is due to its 3D vehicle simulation capabilities as well as the cost-effectiveness obtained by using free to use software.

1.5.1 Godot

Godot is a free and open source 2D and 3D game engine released under MIT licensing. The website to Godot and its documentation can be found in Appendix A. Godot provides the 3D space required to create the car driving simulator to test self-driving car models and collect data to train them. It simplifies the development of the simulator due to the extensive game engine code already written. The node structure style of the game engine and reusability of scenes also allow for faster development.

1.5.2 Justification of the Custom Simulator

The use of a custom-designed car driving simulator within this project was to allow for better testing of the reproducibility of findings found within the literature surrounding end to end deep learning models. For example, the custom simulator could include obstructions that are purposely created to test the model's ability to detect road features, (the detection of road features was one of the findings of the Nvidia paper on end to end learning for self-driving cars). The use of a custom simulator also allows for the creation of a range of tracks that vary in difficulty, this would allow for better model evaluation as the modes can tested in more challenging environments.

2 Method

This project contains three main sections of work. Section 1 includes the development of a car driving simulator in the game engine Godot; section 2 involves creating the application used to collect data, train models and test models within the simulation and section 3 uses the two prior sections to evaluate the performance of self-driving car AI in the custom car driving simulator. Sections 2 and 3 were coded in the programming language Python.

2.1 Creating a Car Driving Simulator in Godot

Before creating the drivable car, the environment and working on other sections of the simulator, a new project in Godot was made, no templates were used for the project. All the code for the simulator is included in the Appendix A as well as the documentation for Godot.

2.1.1 Importing the Drivable Car Model into Godot

This section details the process of creating the drivable car within the simulator so that the car can be driven by both a human user and a trained model depending on whether the user is collecting data to train a self-driving car model or is trying to test a model that they have already trained.

Godot has a built in 3D vehicle class called VehicleBody. This class was used to simulate different features of a car. The textures and vehicle model that were used in the simulation were sourced from the internet and were both free to use. The model can be seen in Figure 2.

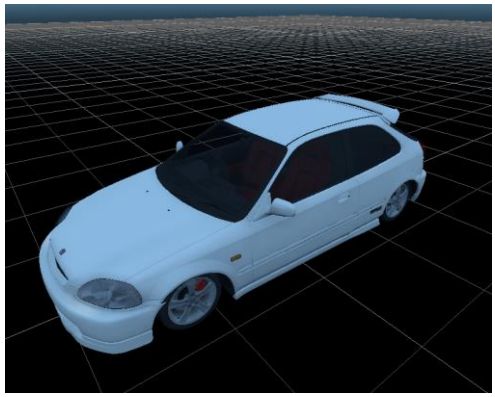


Figure 2 - Car Model used in Simulator

The model shown in Figure 2 was chosen due to the realism and detail in the texturing and modelling to make it more comparable to the real world. The model mesh also has a low polygon count and is therefore optimised to be used in game engines as it requires less processing power to render within the simulator.

The VehicleBody class used on the car also inherits properties from the RigidBody class, therefore the car required a collision shape. The collision shape used for the car was cylindrical and not a detailed mesh as simulating accurate collisions is not part of the scope of this project. The cylindrical shape also reduces the processing power required to calculate object collisions. The collision shape on the car inherits from the CollisionShape class.

2.1.2 Simulating the Car Mechanics

In order to simulate different mechanics of the car, the parameters on the VehicleBody and VehicleWheel classes assigned to the car were modified. The VehicleBody class requires the use of the VehicleWheel class to simulate the car wheels, the final parameters chosen for the VehicleWheels are shown in Figure 4 and the user controlled variables of the VehicleBody are shown in Figure 3.

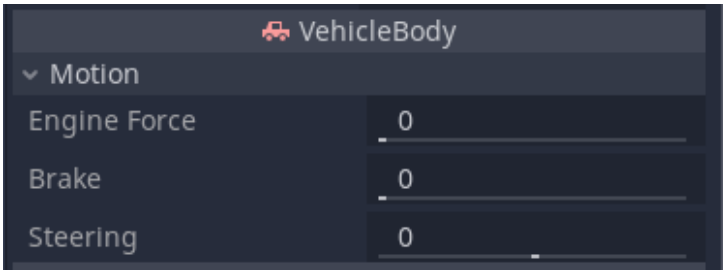


Figure 3 – Image showing the main parameters that control the VehicleBody

In Figure 3 the engine force variable is used to accelerate the car; the break variable is used to control the brake force applied to the car and the steering variable is used to control the steering angle of the wheels. These values are controlled by user input via either a controller or keyboard. The controller provides finer control of all the shown variables due to the continuous nature of the output values of the controller. Keyboards are only able to give discrete inputs to the program and therefore less precise control.

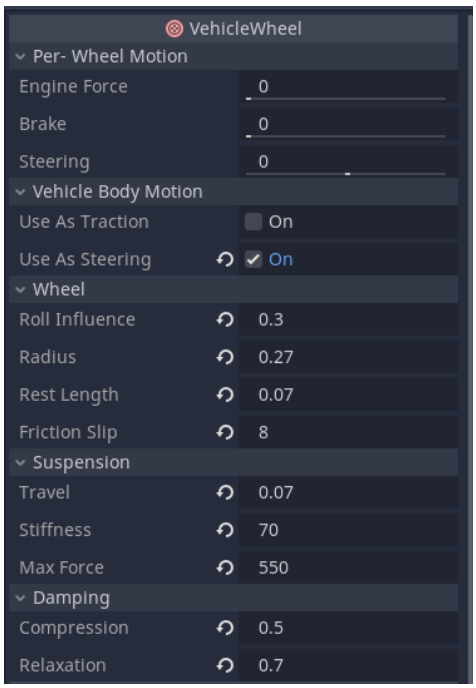


Figure 4 –Main parameters that control the VehicleWheel

The first three variables are equivalent to the VehicleBody variables and are set by the VehicleBody. The variables in the Vehicle Body Motion section are used to decide if the wheel controls the steering or is used to drive the car. Wheels marked as use as traction drive the car and wheels

marked with use as steering steer it. Only driven wheels have an engine force value and only steering wheels have a steering value. Each wheel on the car has its own VehicleWheel parameters.

Other relevant VehicleWheel parameters include the roll influence which refers to how easy the car will roll over; a higher value is more likely to roll. The friction slip determines how much grip each wheel has – a higher value means the wheel will slide less. The variables within the suspension section control the how far the suspension on each wheel can travel; how stiff the suspension on each wheel is and the max force the suspension on each wheel can provide. The components in the damping section control the damping on the suspension during compression and extension.

The car used in the simulation was also given headlights which can be switched on and off by the user to gather a more diverse data set.

The script attached to the car allows the user to control the engine force, brake force and steering angle of the car, it also allows user to toggle the car headlights and self-driving mode.

2.1.3 Creating the Simulator Tracks

The following sections (2.1.3 to 2.1.7) detail the creation of each of the tracks within the simulator. Each track was designed to allow a more diverse data set to be obtained from the simulator as well as provide challenging environments for the AI to be tested in. All the models acquired for the objects within the tracks and all textures are free to use. One of the designed tracks is shown in Figure 5.

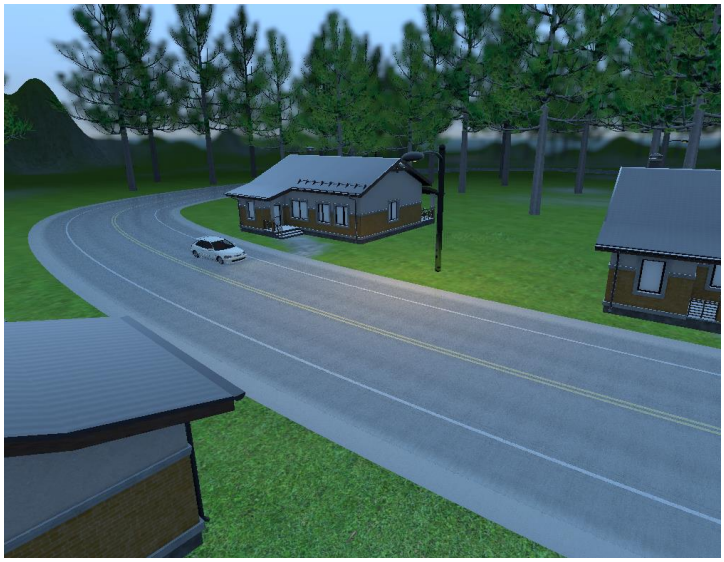


Figure 5 –Grass themed track created for the simulator.

In Figure 5, the ground texture used was the grass texture, the track also included objects such as trees, cottages and streetlights. A table detailing the features of each track is shown in Table 1.

Table 1 – Table summarising the information about each track

	FEATURES					
TRACKS	Ground Texture	Number of Unique Objects	Number of Turns	Inclines	Multiple Light Sources	Road Obstruction
GRASS TRACK	Grass	3	8	No	No	Yes
GRASS2 TRACK	Grass	4	7	No	Yes	Yes
GRASS3 TRACK	Grass	1	7	Yes	No	Yes
SNOW TRACK	Snow	3	11	No	No	No
SNOW2 TRACK	Snow	1	7	No	No	Yes
SNOW3 TRACK	Snow	1	8	Yes	No	Yes
DESERT TRACK	Sand	1	9	No	No	Yes
DESERT2 TRACK	Sand	1	13	No	No	Yes
DESERT3 TRACK	Sand	0	11	Yes	No	Yes

For the road obstruction section of the table, an obstruction can be an object such as a car or overlapping terrain covering the track.

2.1.4 Importing the Objects for each Track

All objects created within the simulator inherit properties from the RigidBody class and have collision shapes like the simulated car. All objects aside from other vehicle objects used in the simulator use the variant of the RigidBody called StaticBody, this variant is used as the objects in the simulator are designed to not move from their original positions, the StaticBody prevents any possible movement. The collision shape added to the StaticBody prevents the user or AI controlled car from passing through objects, this is because the collision shape on the car was set so that it cannot enter the collision shape of another object. An example of an object with its collision shape is shown in Figure 6.

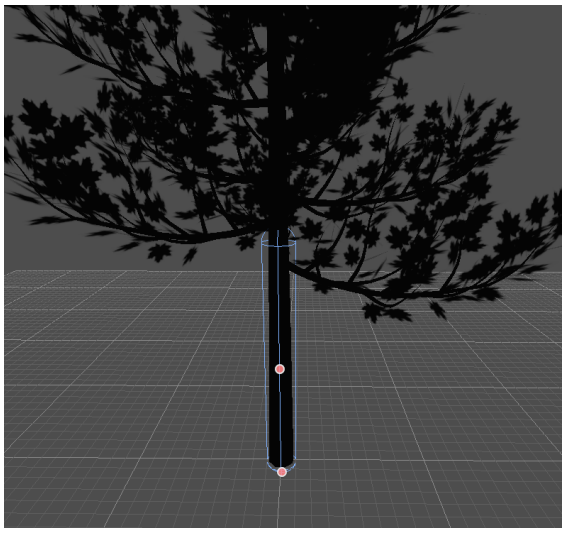


Figure 6 – A tree object with its blue collision shape

Figure 6 shows a tree object used in two of the designed tracks. Its collision shape is shown by the blue rounded cylinder surrounding the trunk of the tree, the simplified shape was chosen to reduce the processing power required to run the simulation. The collision shape does not cover the whole tree as the car would only ever collide with the tree trunk during a collision.

2.1.5 Setting up the Surrounding Environment

To create a more a realistic simulation, two world environments were created in the simulator, this was done using the WorldEnviornment class. The first environment created was a day environment that featured a bright sky and light scene, the second environment featured a dark sky and has a dim lighting. These two environments were chosen to simulate driving during the day as well as during the night. Both environments have properties such as depth of field (DOF) far blur and fog enabled, this is used to blur objects within the simulation at distances far away from the camera, this is used to make to the simulation more comparable to the real world.

Day Environment (a)

Night Environment (b)

Figure 7 – Images showing the day environment and night environment

Functionality to change the environment whilst using the simulator was added so that the user could collect data from both environments without reloading the track from the main menu. Figure 7 shows the Snow Track in the day environment and night environment.

2.1.6 Simulating Terrain

To create different terrain heights across the tracks and apply the ground texture, the hterrain addon for Godot was used. This addon allows the creation of terrain maps which are used to create features such as hills within the tracks, it also adds a collision shape to the ground which prevents the car from falling through the floor. An example of the features created using the hterrain addon is shown in Figure 8.

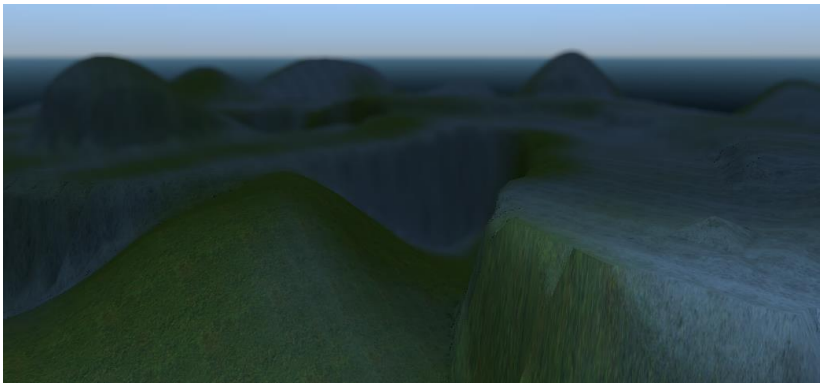


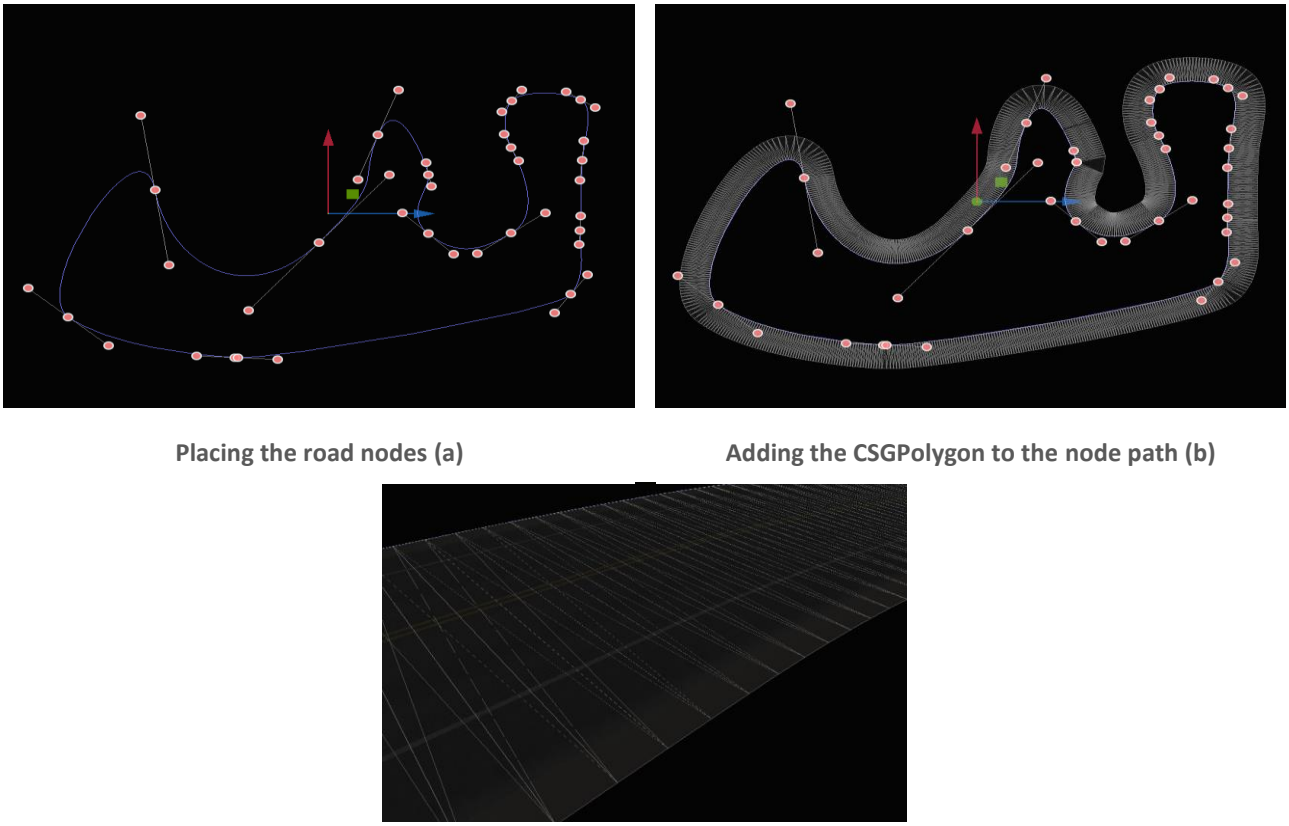
Figure 8 - Mountain and hill features created using the hterrain addon

2.1.7 Simulating the Roads

The roads in the simulator were made using the Path and CSGPolygon classes. Firstly, a closed path was created using the Path class that mapped the interior edges of the road. The initial closed path only connected the series of nodes forming the path with straight lines, curves were added by adding extra nodes onto each original node to define the Bézier curve for the path. (The Bézier curve is the method used by Godot's Path class to draw curves).

The nodes used in creating the outline of the track were moved in the x, y and z directions to create different shaped road layout as well as inclined roads on different tracks. After the closed paths had been created, the shape of the roads was defined using the CSGPolygon class. The CSGPolygon class uses arrays to define the shape of 3D objects and was used to create a flat cuboid to simulate the road.

After the closed path and shape of the road has been defined, the texture for the road was added, and a script to control the appearance of its textures. The script attached controlled properties such as how metallic the road surface appeared and how reflective the surface was. The process of creating the track is summarised in Figure 9.



Placing the road nodes (a)

Adding the CSGPolygon to the node path (b)

Adding the textures to the CSGPolygon (c)

Figure 9 – Images showing the process of creating the roads for the Grass Track. Placement of the nodes is shown in the top left image, the addition of the CSGPolygon is shown in the top right image and the addition of the textures is shown in the bottom image

The images shown in Figure 9 show the process for creating the roads for the Grass Track, the same process was the same for all roads created.

2.1.8 Camera Position

To gather data from the simulation, a camera node was added to the front of the car. The camera inherits from the Camera class and is used to output a stream of images to the screen whilst using the simulator. A second camera was also added to the simulator and was coded to follow the car as it drives round the track. Both cameras can be used for collecting data from the simulation, but the camera positioned on the front of the car was used for this project. This is because the second camera would not be viable in the real world as it is not attached to the car.

Main Camera Position (a)

Secondary Camera Position (b)

Figure 10 – Images showing the two possible camera positions for the user to choose from

The images in Figure 10 show the views from both cameras within the simulator, the image on the left shows the view from the camera mounted to the car and the image of the right shows the view from the camera that follows the car. The camera system was designed so that the user of the simulation can toggle between cameras either whilst collecting data or testing the AI.

2.1.9 Sending Data from the Simulation

To send the steering angle data from the simulated car to the external Python program collecting the data (the self-driving car simulator application), the User Datagram Protocol (UDP) was used. UDP allows programs to send data to other programs using IP addresses and ports. A script in Godot was written using Godot's inbuilt UDP functionality to do this. As both the simulator and the application were ran on the same computer, the IP chosen to send the data on was the local IP address 127.0.0.1. Data was programmed to be sent every 0.1 seconds from the simulator using Godot's Timer class. The script showing the Godot code for sending the data is shown in Figure 11.

```
1  extends Node
2
3  var socket
4  var paused = true
5  signal pause
6  signal unpause
7  onready var steeringAngle = PoolByteArray(str(Autoload.steeringAngle).to_utf8())
8
9  func _init():
10     socket = PacketPeerUDP.new()
11     socket.set_dest_address("127.0.0.1", 5005)
12
13 func _process(delta):
14     steeringAngle = PoolByteArray(str(Autoload.steeringAngle).to_utf8())
15     is_paused()
16
17
18 func is_paused():
19     if Input.is_action_just_pressed("pause_data"):
20         if paused == true:
21             paused = false
22             emit_signal("unpause")
23         else:
24             paused = true
25             emit_signal("pause")
26
27
28 func _on_Timer_timeout():
29     if paused == false:
30         socket.put_packet(steeringAngle)
31     else:
32         socket.put_packet(PoolByteArray(str("paused").to_utf8()))
33         pass
```

Figure 11 – Godot code used to send the steering angle data from the simulator

The script in Figure 11 was attached to a default node within each tracks Godot scene. In lines 3 to 7 of the script the script variables and signals were defined; the socket variable was used to store information about the UDP method of transmission and the variable steeringAngle was used to store the value of the steering angle of the car before it was sent. The script gets the steering angle value from the global variable script called Autoload, which is updated by the script controlling the car. Global variables are used to store information that is used across multiple scripts.

In lines 9 to 11, the function called _init (which runs as the scene is loading) sets the socket variable to the socket type PacketPeerUDP which is then used to send raw UDP packets of data. The destination address of the socket was set to the local IP address and the port to 5005. The port was used to separate the steering angle data from other data sent to the same IP address.

The function _process in lines 13 to 15, runs every frame within the simulator. It was used to update the steering angle variable in the current script with the value from the global variable script, as well as call the is_paused function. The is_paused function is used to check if the user has pressed the keyboard or controller key that corresponds to toggling the sending of the steering angle data. The

emit_signal functions on lines 22 and 25 were used to indicate to other scripts to update the user interface (UI).

The final function in this script was the on_Timer_timeout function (lines 28 to 33 in the code), this function was connected to a Timer node also within each track's Godot scene. The Timer node was set to emit a signal every 0.1 seconds which when received by the script triggered the calling of the on_Timer_timeout function. The calling of the function tells the script to check if the user wants to send data and then either sends the steering angle of the car or a string indicating that angle data is not being sent.

2.1.10 Car Driving Simulator User Interface (UI)

The main menu for the car driving simulator was made using the Godot UI tools. The layout was created using 9 buttons, 4 text labels and 2 texture rectangles. The menu scene shown in Figure 12, is the default scene for the car driving simulator and will load whenever the executable file is ran. The 9 buttons are coded to load the track corresponding with the track name written on them.

Figure 12 – The main menu UI for the car driving simulator

The controls for the simulator are shown to the right side of the image in Figure 12 and explain to the user how to control the car, toggle the environment settings as well as send data to the application. The user is also able to return to this main menu from within a track scene to select a different track.

A UI to display once the track scene is loaded was also made for the simulator which displays the engine force, brake force and steering angle of the car. The within track UI also displays whether the user is sending data to the application.

2.1.11 Exporting the Simulator

The completed simulator was exported from the Godot game engine editor to a single executable file that could be ran from the application.

2.2 Self-Driving Car Simulator Application

The following sections detail how the application that communicates with the simulator was created. The application can collect data from the simulator, train models with the gathered data and test the trained models within the simulator. The application was coded in Python with a link to the final code provided in the Appendix C. All modules used in this project are also listed in Table 2 in Appendix B.

2.2.1 Data Collection

To collect data from the simulator, the application was coded to receive the steering angle sent by UDP to the local IP and then capture the corresponding frame from the camera within the simulator. Receiving the steering angle was done by listening to the data sent to the local IP address on port 5005. The corresponding frame was captured by coding the application to screenshot the simulator window that contained the view from the camera whenever data was received. Relevant parts of the data collection script are shown in Figure 13.

```
71  # receives steering angle and image data from simulator
72  def data_collection(window, folder_for_original_images, folder_for_steering_angles, file_name_for_steering_angles,
73                      folder_for_small_images, list_of_angles_paths, list_of_frames_paths, remove_excess):
74      # Server
75      s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
76      s.bind(("127.0.0.1", 5005))
77
78      # Steering Angles
79      steering_angles = []
80      # Screen Capture
81      frames = []
82      sct = mss()
83      # collecting data
84      while True:
85          bounding_box = window_size_and_pos()
86          data, address = s.recvfrom(1024)
87          data = data.decode("utf-8")
88          if data != "paused":
89              steering_angles.append(float(data))
90              print(data)
91              screenshot_img = sct.grab(bounding_box)
92              img = Image.frombytes("RGB", screenshot_img.size, screenshot_img.bgra, "raw", "BGRX")
93              open_cv_image = np.array(img)
94              # Convert RGB to BGR
95              open_cv_image = open_cv_image[:, :, ::-1]
96              frames.append(open_cv_image)
97          if break_collecting_data is True:
98              break
```

Figure 13 – Python code used to receive steering angle values and screenshot the simulator

In line 75 of the code in Figure 13, the code specifies that the socket is to listen for UDP messages on a named port, the IP and port to listen on is given on line 76. In line 79 a list was defined to store the steering angles received from the simulator and on line 81 another list was defined to store the screenshots of the simulator. Screenshots were taken by the MSS module; it is initialised on line 82.

In the while loop starting on line 84 and ending on 98, the function gets the position and dimensions of the simulator window using the window_size_and_pos function shown on line 85, it then listens for UDP messages on the specified IP and port using the recvfrom function (on line 86), the argument to the recvfrom function specifies the maximum size of the message that can be received on each iteration of the function. The program waits on the recvfrom function line until it receives a message from the simulator (the limitation of this method is that the user can only stop collecting data if messages are still being received from the simulator).

Once the application receives the message, it decodes it using the utf-8 decoding method (on line 87). If the data is received is a steering angle, it is appended to the list of steering angles (on line 89), and a screenshot is taken of the simulator using the MSS function stored in the variable sct (on line 91). The frame is then converted into the format of an OpenCV image (on lines 93 and 95) and appended to the list of frames (on line 96).

The while loop is coded to continue until the break_collecting_data variable is set to True. The variable is set through the graphical user interface (GUI) explained in section 2.2.5. After the loop is broken, the data collection script saves all the data to the paths specified by the user through the GUI. The application data saved includes one folder of original frames, one folder of resized and cropped frames, a text file containing the steering angles, and two text files containing the paths to the images and steering angles.

The data collection script was also given the functionality to remove excess data from the data collected from the simulator. This was added to allow the user to balance the distribution of steering angles included in the collected data sets, as the data collected from the simulator contained a high percentage of steering angles with a value of zero. This function works by finding the positions of zero value steering angles in the steering angles list and using the index to remove them in both the frames list and steering angles list. Zero value angles and frames are removed from the data set until they make up a maximum of 15% of the total data in the data set.

2.2.2 Implementing the Nvidia Model

The deep learning neural network used to predict the steering angle of the car a CNN. This project used the same CNN architecture as the End to end learning for self driving paper by Nvidia.

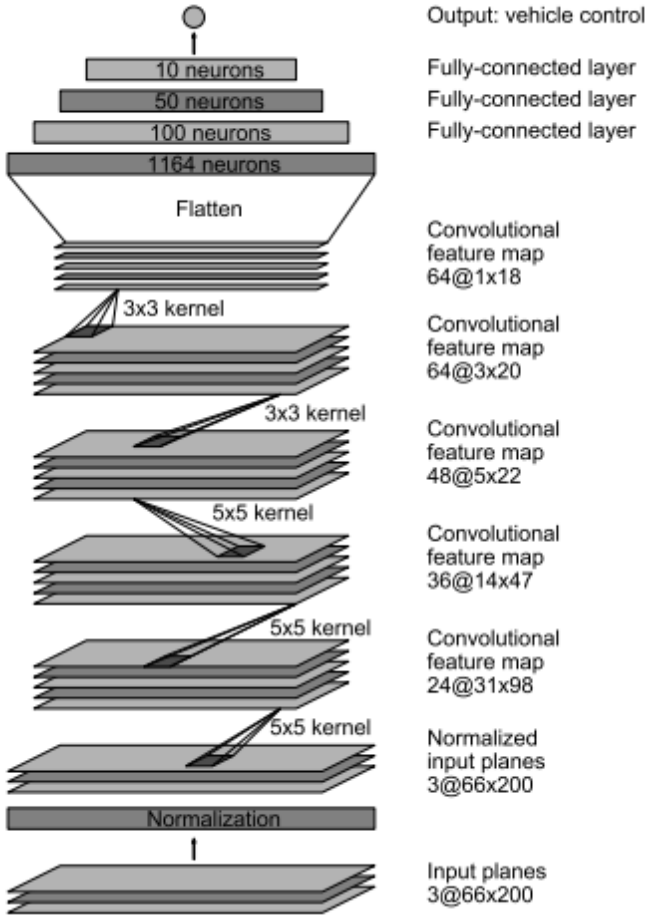


Figure 14 – The CNN Architecture (Borjarski et al., 2016)

The diagram in Figure 14 is from the Nvidia paper - it shows the architecture of the NN. The input images to the network are 3-channel RGB (Red Green Blue) images with dimensions of 66 by 200. After the image has been normalized, the neural network uses five convolutional layers to perform feature extraction. The first three layers use a 5x5 kernel size and the last two convolutional layers use a 3x3 kernel size. The filter size used on each layer is also shown in Figure 2. In layer order 5 the filter sizes are 24, 36, 48, 64 and 64. After the convolutional layers, a flatten layer is used, this is used to format the data into a one-dimensional array so that it can be inputted into the three fully-connected layers which output the steering angle.

As this is a regression machine learning problem (due to the continuous value output), a mean squared error function is used as the final activation function for the network. The equation is given in equation 2 in section 1.6 of this report.

The Nvidia CNN model was coded in Python using the imported Tensorflow module and is shown in Figure 15.

```
595  # creates CNN model architecture and returns compiled model.
596  def model_maker_nvidia():
597      # Model
598      model = Sequential()
599      # Add Layers Convolutional
600      model.add(Convolution2D(24, (5, 5), (2, 2), input_shape=(66, 200, 3), activation='elu'))
601      model.add(Convolution2D(36, (5, 5), (2, 2), activation='elu'))
602      model.add(Convolution2D(48, (5, 5), (2, 2), activation='elu'))
603      model.add(Convolution2D(64, (3, 3), (1, 1), activation='elu'))
604      model.add(Convolution2D(64, (3, 3), (1, 1), activation='elu'))
605      # Add Layer Flatten
606      model.add(Flatten())
607      # Add Layer Dense
608      model.add(Dense(100, activation='elu'))
609      model.add(Dense(50, activation='elu'))
610      model.add(Dense(10, activation='elu'))
611      model.add(Dense(1))
612      # Compile Model
613      model.compile(Adam(lr=0.0001), loss='mse')
614
615      return model
```

Figure 15 – Coding the Nvidia model in Python using the Tensorflow and Keras modules

In Figure 15 shows the code used to construct the model within Python. The activation functions for each layer were set using the activation argument of the add function. For the convolutional and dense layers shown in lines 600-604 and 608-610, the activation function used was ELU. The loss function for the NN was set in line 607 using the loss argument of the compile function. The loss function used to train the model was a MSE function.

The code in Figure 15 does not include a normalization layer like the Nvidia model, as the images are normalised before they are inputted into the model. The Dense layers in the network in Figure 15 are the fully-connected layers from Figure 14. The optimiser used for the model is the Adam optimiser, it is used to change the weights in the neural network using the output of the loss function. The speed at which the weights are changed by the optimiser is set using the learning rate, the learning rate used in this model was 0.0001.

The image normalization required for the network was done during the pre-processing of the frames (explained in section 2.2.6); the array that represents the image is divided by 255 so that each RGB pixel is represented by a float between 0 and 1 instead of an integer between 0 and 255. This allows for faster convergence during the training of the CNN. The steering angles used for training are also normalized before training, this was done by multiplying each original steering angle value by 2, so that they fall within the -1 to 1 range.

2.2.3 Training Models

The code implemented to train the model was designed to import the collected simulator data; train the model using the parameters set by the user, record data about the training and then save the trained model.

As the CNN does not include any pretrained weights all node weights in the NN start the training process as randomised values. An image augmentation function was added so that more data could be generated from the collected data set. This was done so that the network had more data to train on as networks that are not pretrained require more data to optimise the weights. The image augmentation function takes one training image and changes the image by either adding one or a combination of the following effects, translation, zoom, brightness, or flip; the probability the augmentation function adds an effect is 0.5 for each effect.

The translation effect moves the centre of the image in the x or y direction by -10% to 10% of the original dimensions of the image (200x66). The zoom effect scales the image up in the range of a 0 to 20% increase, this effect also crops the image as the parts of the scaled image outside the original dimensions are removed. The brightness effect multiplies the default image brightness by a value between 0.2 and 1.2. The flip effect reflects either side of the image (relative to the central y-axis) to the other side of the image and multiplies the corresponding steering angle by -1, this means a left curving road becomes a right curving road and vice versa.

Models are trained using an iterative method that uses a batch generation and augmentation function to generate augmented training and non-augmented validation batches from the larger training and validation data sets. The batch generator function also calls the pre-processing function to pre-process both the validation and training sets. The image augmentation function is only used on the training set as the validation set needs to represent the images the model will see when being tested. The code in Figure 17 is used to split the data and train the model.

```
564     print("Performing Test Train Split")
565     # perform test train split
566     x_train, x_valid, y_train, y_valid = train_test_split(frames, steering_angles_float_array, test_size=0.2,
567                                                           shuffle=1)
568     print("Training Model")
569     call = callbacks.ProgbarLogger()
570     window.write_event_value('-THREAD PROGRESS-', str(call))
571     history = model.fit(batch_gen(x_train, y_train, images_per_batch_train, 1), steps_per_epoch=epoch_steps,
572                         epochs=num_epochs, validation_data=batch_gen(x_valid, y_valid, images_per_batch_validation, 0),
573                         validation_steps=val_steps, callbacks=[call], verbose=2)
```

Figure 17 – Code used to split data and train model

In line 566, the data is split into a training set which uses 80% of the original data and a validation set which uses the remaining 20% of the data. The x variables in line 566 are arrays containing resized and cropped frames with dimensions 200x66 pixels and the y variables are arrays containing the corresponding steering angles. The argument “shuffle” is used to tell the function to shuffle the order of the arrays before splitting them into sets. The function train_test_split is imported from scikit-learn module.

The model is trained in line 571 and the progress of the training process is displayed using lines 569 and 570. TensorFlow's fit function is used with the compiled model (stored in the variable "model"), this function calls the batch generation function as an argument, which will generate a new training and validation batch from the split data for each step of training. The number of batches per epoch is specified by using the steps_epoch_argument; the number of iterations is specified by the epochs argument; the number validation batches generated for each epoch is specified by the validation_steps argument. The callbacks and verbose arguments are used to tell the function to return data about the training during training. A summary of the training process is shown in Figure 18.

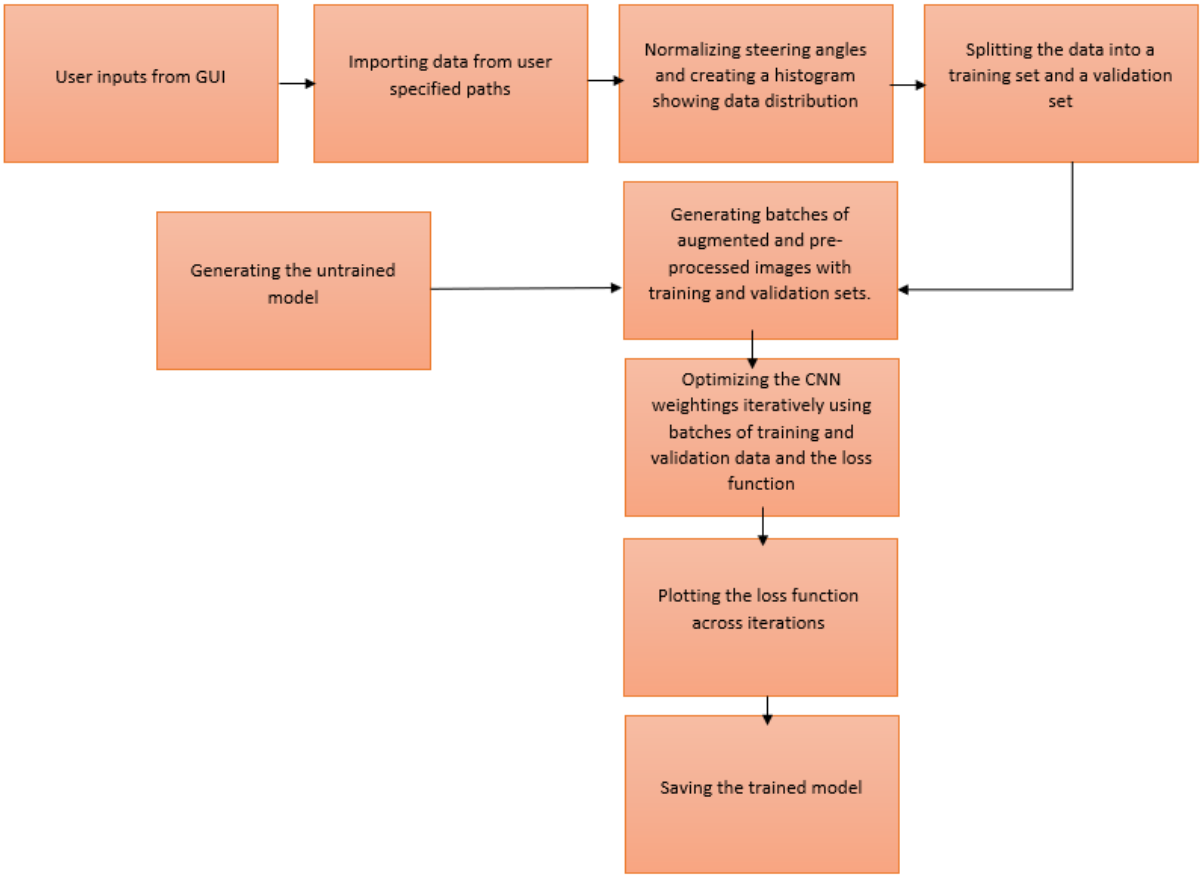


Figure 18 – Summary of the training process

Figure 18 shows the process of training end to end self-driving car models using the simulator application.

2.2.4 Testing Models

The functionality implemented to test the trained models using the application was created using the code shown in Figure 19. The code loads the trained model inputted from the GUI (line 517), it then finds the window containing the simulator (line 523) and screenshots images every frame (line 524). This image is then converted to OpenCV format before being cropped, resized, pre-processed, and normalized using the same method used in image training (lines 526-532). The trained model is then shown the image and outputs a value between -1 and 1. This value is then divided by two to get a value between -0.5 and 0.5 radians for the angle, the angle is then rounded to 3 decimal places (line 533).

```
514  # gets image data from the simulator and sends back predicted steering angle from model used
515  def test_model_simulator(path_to_model, window):
516      # Import trained model
517      trained_model = keras.models.load_model(path_to_model)
518      # Server
519      s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
520      # Screen Capture
521      sct = mss()
522      while True:
523          bounding_box = window_size_and_pos()
524          screenshot_img = sct.grab(bounding_box)
525          input_image = Image.frombytes("RGB", screenshot_img.size, screenshot_img.bgra, "raw", "BGRX")
526          open_cv_image = np.array(input_image)
527          open_cv_image = open_cv_image[:, :, ::-1]
528          h, w, c = np.shape(open_cv_image)
529          frame_cropped = open_cv_image[int(h / 2) + 50:h - 145, 20:w - 20].copy()
530          frame_resize = cv2.resize(frame_cropped, (200, 66), cv2.INTER_AREA)
531          frame_processed = frame_preprocessing(frame_resize)
532          test_image_array = np.array([frame_processed])
533          prediction = round((float(trained_model.predict(test_image_array)) / 2), 3)
534          print(str(prediction))
535          window.write_event_value('-Angle-', prediction)
536          prediction = str(prediction).encode('utf8')
537          s.sendto(prediction, ("127.0.0.1", 1234))
538          if break_testing is True:
539              window.write_event_value('-Angle-', "Stopping Testing")
540              break
```

Figure 19 – Code used to test trained CNNs

After a value has been predicted, it is encoded and sent to the simulator using UDP on the local IP address on the port 1234 (line 536-537). The method for sending data to the simulator is the same as the method used for sending the data to the application except the simulator listens on the port instead of the application.

2.2.5 Image Pre-Processing

The original images collected during data collection as well as the stream of images the model is shown during testing are all cropped, resized, and then pre-processed using computer vision techniques implemented using the OpenCV module for Python.

All images are firstly cropped so that the sky and front of the car are removed from the image, this was done as both areas do not contain any patterns that should impact the steering angle of the car. After the images were cropped, they are then reduced in size to have dimensions of 200x66 pixels, this was done as the CNN requires this image size for the input layer, but also because smaller images take up less memory and can improve the speed at which the NN trains. The resizing of image was done using the OpenCV's resize function.

Images were pre-processed at different times depending on whether the images were used for training or used for testing. Training images that are part of the training set and not the validation set are first augmented before being pre-processed, all other images are not augmented.

The pre-processing of the images was done using the frame_preprocessing function which uses OpenCV functions to pre-process the image. The first pre-processing done to the image was to change the image from a RGB image to a composite analogue signals (YUV) image using OpenCV's cvtColor function. This was done to reduce the memory the images used as YUV images have a smaller file size than equivalent RGB images. Smaller file sized images load faster than larger images and therefore can speed up the importing of data into the model during training.

After the images was changed to the YUV format, a Gaussian blur was applied. The Gaussian blur reduces the noise and detail in the image by applying a low pass filter, this allows the CNN to better features in the data as feature detection can be sensitive to noise. The final section of the frame_preprocessing function was to normalize the images by dividing the array of numbers representing the images by 255 to get a value between 0 and 1. The frame_preprocessing function is shown in Figure 20.

```
46  # preprocesses a frame and returns it
47  def frame_preprocessing(frame):
48      frame_yuv = cv2.cvtColor(frame, cv2.COLOR_RGB2YUV)
49      frame_blur = cv2.GaussianBlur(frame_yuv, (3, 3), 0)
50      frame_normalized = frame_blur / 255
51      return frame_normalized
```

Figure 20 – The frame_preprocessing function

The image is converted from RGB to YUV in line 48; the Gaussian blur is applied in line 49 and then the image is normalised in line 50. The kernel size used for the Gaussian blur was 5x5 shown in the second argument input to OpenCV's Gaussian blur function.

2.2.6 Graphical User Interface

The GUI for the application was made using the PySimpleGUI module for Python. The main menu GUI was created to allow the user to select a function and to input the required arguments. The main menu created is shown in Figure 21.

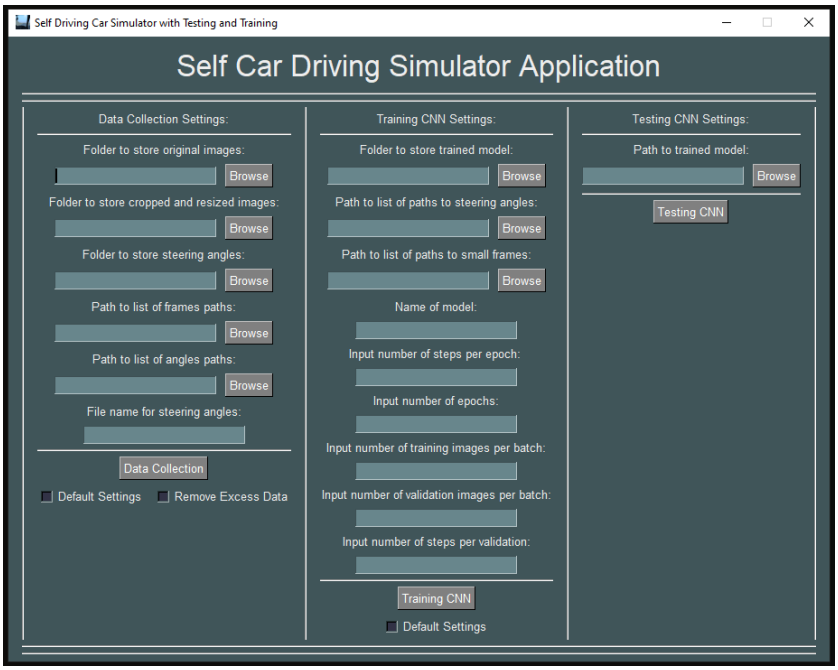


Figure 21 – The main menu for the simulator application

For the data collection and testing sections of the main menu, it was made so that the user could choose to use default settings. The default settings for collecting data and training models save all the created files in the user's home directory (location where the application is installed when using the installer). For the data collection section, the user can also choose to remove excess data. Using the default settings or choosing to remove excess data can be done by checking their corresponding checkboxes.

Each section also contains a button that closes the main menu and opens another window which is used to control the function that the user has selected. If the user has not inputted the required arguments for their chosen function, an error message will display prompting the user for valid arguments. When valid arguments are inputted by the user and either the collecting data or testing functionality is selected, the application will run the simulator executable file using the subprocess module alongside loading the relevant GUI for the function.

To allow the running of the GUI and the selected function at the same time, the threading module was used. The threading module allows multiple functions to run at the same time.

2.2.7 Exporting the Application

The application was exported using the Pyinstaller module, this module compiles the Python code to a single executable file. The Pyinstaller module was used by calling it from the command line of the command prompt. To specify that the Python code should be compiled to a single executable, the argument "—onefile" was used.

2.2.8 Creating an Installer for Application Distribution

The installer for the application was created using the Inno Setup Compiler program. The installer installs folders within the user's program folder as well as the self-driving car simulator application. The installed folders are used to store simulator data, trained models and the car driving simulator executable file.

2.3 Evaluating AI Performance

2.3.1 Creating Models

To evaluate the performance of the models within the simulator, image and steering angle data was first collected from the simulator and then used to train models. Data was collected from the Grass1, Snow1, Desert1, Snow2 and Desert2 tracks (5 out of the 9 simulator tracks). Data was not collected from all the simulator tracks as some were used for testing. Data was also only collected using the day environment simulator setting with headlights turned off. During data collection, the application received steering angle data from the simulator every 0.1 seconds and captured an image when this data was received. The camera on the simulated car therefore captured frames at a frame rate of 10 frames per second (FPS). The images were stored in folders and the steering angles were stored in text files on the computer running the application.

The data collected from the simulator was then balanced by removing excess zero values until at maximum 15% of the data was of the car driving on a straight road. Images were then cropped to remove features irrelevant to the predicting of the steering angle and resized so that they could be inputted into the model. The data for each model was then split into a training and validation set with 80% of the collected data forming the training set and the remaining 20% forming the validation set. Training and validation batches were then generated from the larger sets. During batch generation, the training batches were also augmented before being pre-processed whist the validation batches were only pre-processed. The images explained in this section are shown in Figure 23.

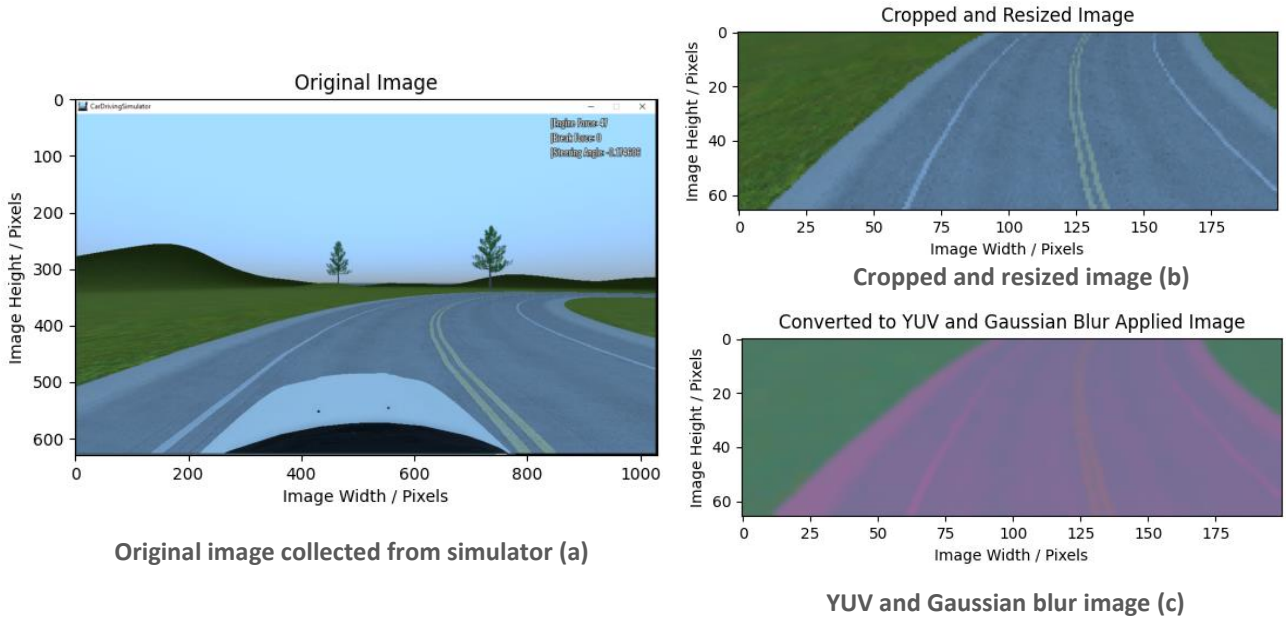


Figure 23 – Examples of original, cropped and resized and pre-processed images

Models were then trained from the generated batches using varied training settings dependent on the amount of data in the data set for that model. The loss calculated from the MSE error function for both the training and validation batches of each epoch was recorded to analyse the training of the models.

2.3.2 Testing Models

The models were tested by allowing them to control the steering angle of the car within the simulator, this was done by inputting a stream of images from the simulator in real time into the model and sending the output of the model to the simulator. The images shown to the model were pre-processed using the same methods used on the validation set during the training of the model.

To evaluate the performance of each model, the autonomy and average time between interferences was assessed. The autonomy of a model was assessed by calculating the rate of interferences by the user. Interfering with the autonomy of the model was only done when the car steered off the road or crossed into the lane of oncoming traffic If the models instantly corrected the road position of the car, the car was not interfered with. The autonomy of each model was calculated for all the tracks within the simulator. The equation used to calculate the autonomy value is shown in equation 3.

$$Autonomy = \frac{Number\ of\ Interferences}{Total\ Driving\ Time} \quad \text{(interferences per second)} \qquad (3)$$

In equation 3, the number of interferences is the number of interferences by the user over the course of 2 laps of the track in opposite direction. The total driving time is the time taken to complete the 2 laps and was recorded in seconds.

Each model was tested four times on each track. For the first test, the engine force of the car was set to a constant value of 30, this engine force value simulated the car being driven at a steady speed. For the second test, the engine force was set to 60, this simulated the car being driven a faster speed. The autonomy value for both speeds on each track is the result of completing one lap in each direction on the given track, the timer was paused as the car was manually turned around between opposite direction laps. An overall autonomy value was calculated for each model, this value was an average of all autonomy values calculated for the model.

Models were only tested on tracks that had not been used for data collection. The tracks used for testing were Grass2, Grass3, Snow3 and Desert3. The test tracks could be qualitatively described as more difficult tracks due to the presence of features such as multiple light sources and inclines.

As the previous part of the testing only tested the model performance using the simulated day environment, an additional test was done to measure the performance of the models in the simulated night environment. The same trained models that had only been shown training data containing images from the day environment were tested on the 5 tracks used for data collection but with the night environment and headlights toggled. The performance was measured using the same method described for the standard testing.

3 Results

3.1 Data Collection

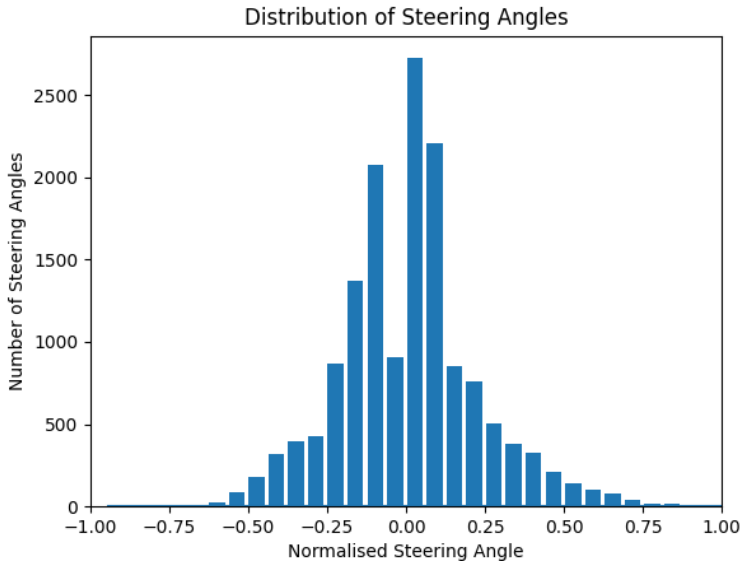


Figure 24 – Histogram showing the distribution of the data collected from the simulator used to train Model V3

The histogram in Figure 24 shows the distribution of normalised steering angle data used to train Model V3, the data is from the Grass1, Snow1 and Desert1 tracks. The total amount of steering angles and therefore images collected was 15055. This provides 12044 training pairs and 3011 validation pairs.

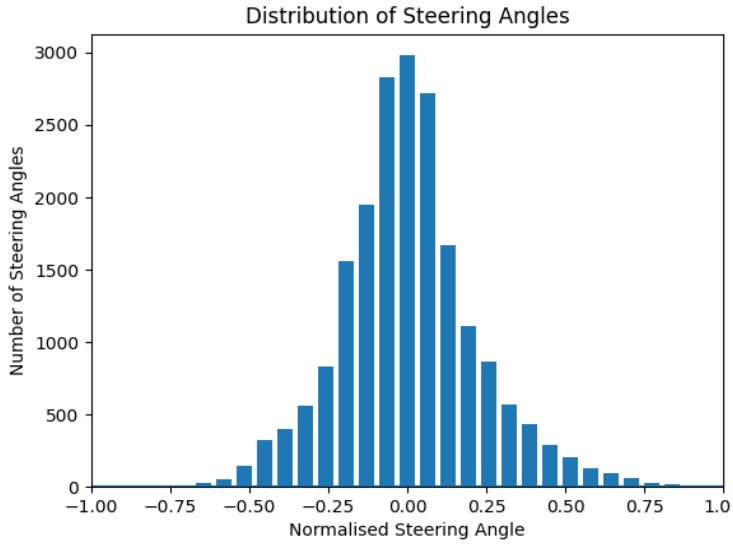


Figure 25 - Histogram showing the distribution of the data collected from the simulator used to train Model V4

The histogram in Figure 25 shows the distribution of normalised steering angle data used to train Model V4, the data is from the Grass1, Snow1, Desert1, Snow2 and Desert2 tracks. The total amount of steering angles and therefore images collected was 19893. This provides 15914 training pairs and 3979 validation pairs.

3.2 Training

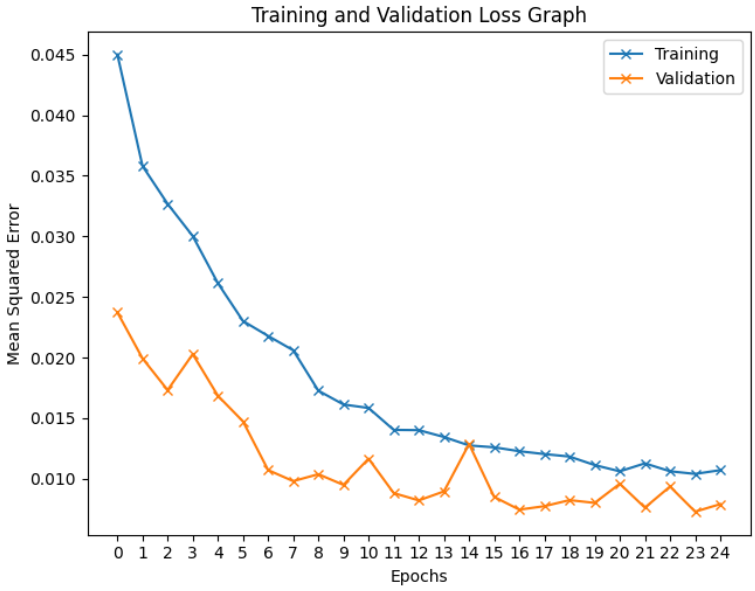


Figure 26 – Loss graph for the training of Model V1

Model V1 was trained for 25 epochs with 200 training batches generated per epoch, each batch contained 64 images. Each epoch was validated on 100 validation batches containing 32 images per batch. This means for each epoch, the model was trained on 12800 augmented images and validated on 3200 non augmented images. The final validation MSE for the model was 0.0079. The training time was 2959 seconds or 49 minutes and 19 seconds.

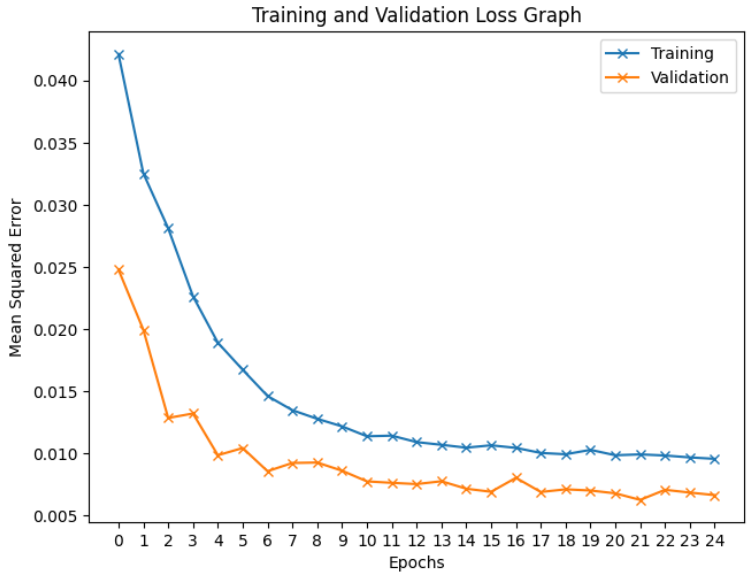


Figure 27 – Loss graphs for the training of Model V2

Model V2 was trained for 25 epochs with 250 training batches generated per epoch, each batch contained 64 images. Each epoch was validated on 125 validation batches containing 32 images per batch. This means for each epoch, the model was trained on 16000 augmented images and validated on 4000 non augmented images. The final validation MSE for the model was 0.0066. The training time was 3700 seconds or 61 minutes and 40 seconds.

3.3 Testing

3.3.1 Day Environment

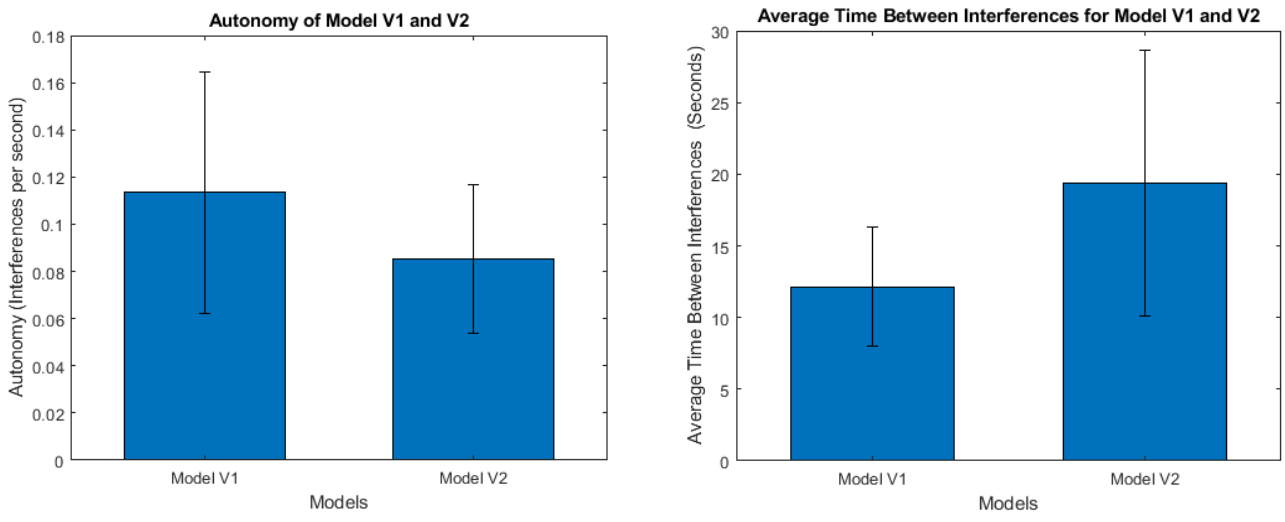
Table 2 – Table showing average time between interferences for Model V1 and Model V2 across the testing tracks

		Average Time between Interferences (Seconds)			
Model	Tracks	Engine Force of 30	Engine Force of 60	Overall per Track	Overall per Model
Model V1	Grass2	25.3	7.2	16.3	12.1
	Grass3	15.6	7.7	11.7	
	Desert3	21.1	9.1	15.1	
	Snow3	6.7	4.4	5.5	
Model V2	Grass2	31.4	8.3	19.9	19.4
	Grass3	22.0	9.2	15.6	
	Desert3	57.3	10.2	33.8	
	Snow3	10.8	5.9	8.3	

Table 3 – Table showing the autonomy value for Model V1 and Model V2 across the testing tracks

		Autonomy Value (Interferences per second)			
Model	Tracks	Engine Force of 30	Engine Force of 60	Overall per Track	Overall per Model
Model V1	Grass2	0.0395	0.1386	0.0890	0.1135
	Grass3	0.0640	0.1294	0.0967	
	Desert3	0.0473	0.1102	0.0788	
	Snow3	0.1502	0.2284	0.1893	
Model V2	Grass2	0.0318	0.1200	0.0759	0.0854
	Grass3	0.0455	0.1084	0.0769	
	Desert3	0.0174	0.0982	0.0578	
	Snow3	0.0928	0.1688	0.1308	

Tables 2 and 3 show the average time between interferences and the autonomy value for models V1 and V2 on each track at each engine force value. It also shows the overall average per track and the average for each model across all the tracks tested on. The raw data used to calculate the values in the tables can be found in Appendix D.



Autonomy (a) **Average time between interferences (b)**

Figure 28 – Bar chart showing the average time between interferences for Model V1 and V2

The error bars in Figure 28 are ± 1 standard deviation of the mean. The data used is taken from Table 2 and Table 3. The standard deviations are 0.044, 4.18 and 0.027, 9.26 for Models V1 and V2 respectively (autonomy stated first and then average between interferences).

3.3.2 Night Environment

The models were unable to autonomously drive the simulated car when the night environment was active with the headlights on. No values were calculated as the model constantly required interference by the user across all tracks tested on.

4 Discussion

4.1 Performance Evaluation

4.1.1 Lane Detection and Road Following

This project found that CNNs can perform lane detection and road following tasks even when these problems are not broken down into individual tasks. This is evidenced by the autonomy values of 0.1135 and 0.0854 calculated for both models V1 and V2, respectively. The project also found that both lane detection and road following tasks do not require a large amount of training data to complete, this was due to Models V1 and V2 only using 15000 and 20000 training images, respectively.

The limitation to these findings is that the training and testing images used the same road type and size, meaning that the CNN models were not trained or tested on a variety of roads, therefore making both lane detection and road following easier for the CNN. This could explain the need for only a small amount of training data.

To improve this project and test the performance of the CNN models further, a variety of road textures and sizes could be used to make the task more challenging. An extension to this would be to also add damaged roads with obscure lane marking further increasing the complexity of the tasks.

4.1.2 Obstruction Detection

Another finding of this project was that the trained CNN models were also able to learn to detect and avoid road obstructions (such as parked cars) using only the steering angle of the simulated car and an image showing the position of the obstruction in front of the car. Results from testing on unseen tracks such as Grass2 (which included a parked car obstructing the path of the simulated car) demonstrated these findings when the model predicted steering angles that resulted in the car moving to avoid the object and then returning to a normal road position.

The limitations that apply to the tasks of the lane detection and road following can also apply to task of obstruction detection. The lack of variety in the obstructions used made this task easier than it would be in the real world. To improve the complexity of the task, the variety of obstructions could be increased, for example, potholes could be added to the roads within the simulator.

4.1.3 Self-Correcting Models

The modes trained in this project were also found to be able to recover from bad road positions and correct themselves. For example, the model would sometimes steer the car so that it partially crossed into the lane of oncoming traffic, when the model did this, it would often steer back in the opposite direction to correct its road position.

An explanation for why this occurred (despite the lack of training data showing the model how to correct its road position) is the use of the image augmentation function. The function can translate

and change the zoom of the image, which simulates car being in different road positions acting as data for self-correction.

A second explanation could be that the CNN detects that the car is not in the road position it expects for the shape of the road in the image, so it predicts a steering angle that returns the car to the expected position.

4.1.4 Importance of Data

As mentioned in the introduction to this report, CNNs require a large amount of high-quality data to be able to produce good results (Simard, Steinkraus, and Platt, 2003). This was supported by the findings of this report and evidenced by the lower autonomy value of Model V2 when compared to Model V1 (0.0854 compared to 0.1135). As Model V2 was trained on a dataset of 20000 images compared to 15000 images for Model V1 (a 33% increase in data set size), this was expected.

The quality of the data used to train Model V2 was also higher when compared to Model V1. This was because Model V2 was trained on 2 extra tracks than Model V1. The extra tracks contained more variety for the model to train on (e.g., more turns for the model to navigate as well as more training data relating to the navigation of obstructions). This also factored into the increased performance of Model V2 in comparison to Model V1.

To improve the performance of models without using a larger dataset, pretrained layers could be added to the Nvidia model architecture. An example of pretrained layers that could be added is Resnet50. This would reduce the need for a large dataset as the pretrained layers are already trained in feature detection, meaning the weightings in the NN would only need to be optimised to detect the relevant features to the task needed for self-driving.

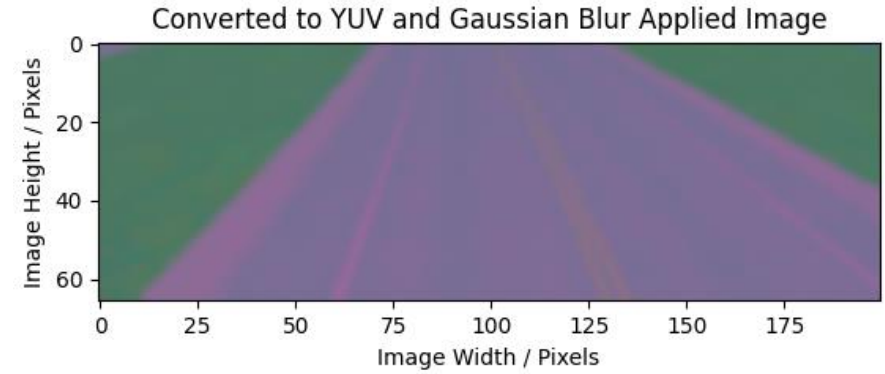
The distribution of the data could also be shown to be a factor affecting model performance, this is evidenced by the histograms shown in Figures 24 and 25. The distribution of steering angles for Model V2 shown in Figure 25 shows a smoother, normally distributed shaped histogram compared to the histogram for Model V1 which is less smooth and has a bias to positive steering angles (right sided). The right-sided bias of the data for Model V1 however could be a smaller/non-existent factor due to the inclusion of the augmentation function, which flips 50% of images and swaps the direction of the steering angles. The bias of the steering angles would still have an effect during the calculation of the validation loss of the model as the validation data set is not augmented.

4.1.5 Training

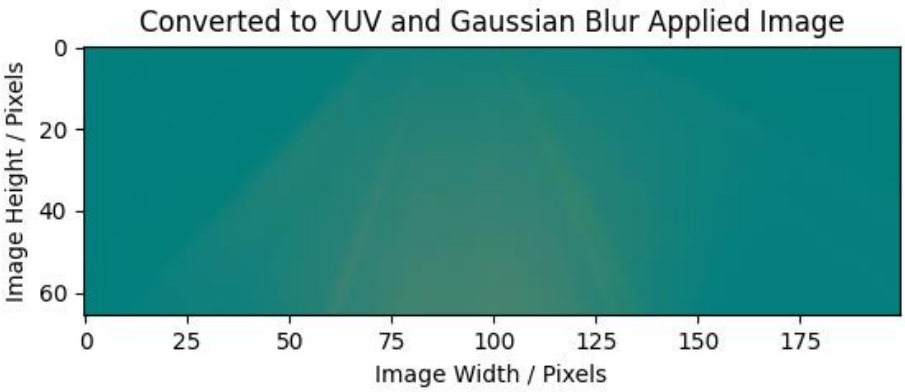
From the final validation loss value (MSE) of the models, the difference in performance of the models could have been predicted. Model V1 has a MSE of 0.0076 and therefore the predicted value has an error of ± 0.087 from the actual value recorded in set in comparison to Model V2 which has a lower MSE of 0.0066 and an error of ± 0.081, (the error is the difference between the normalized predicted steering angle and normalised actual steering angle). Due to the lower validation MSE of Model V2, its increased performance in testing was expected. To further test the correlation more models should be trained and tested.

4.1.6 Driving in Different Conditions

It was found in this project that the CNNs were unable to complete the tasks previously mentioned when the testing environment was changed from that of the training data (tested in night conditions even though training had only been done in day conditions). This can be explained by analysing the differences that the environment conditions make to the images shown to the CNN. Figure 29 is used to demonstrate this.



Day Environment pre-processed imaged (a)



Night Environment pre-processed imaged (b)

Figure 29 – Images showing the effects of pre-processing on images from day and night environments

As shown in Figure 29, the day and night environments appear different when pre-processed, for example, the day environment shows a clearly defined road when compared to the night environment which shows a road with unclear outlines. The effect of this is the inability for the trained CNN models to recognise the patterns found in the day environments within the night environment images.

To improve the ability of the model to drive in a variety of conditions, the model could either be trained on a dataset that also contained images in a variety of environments, or the pre-processing of the images could be changed so that the images from different environments appeared more similar. This could be implemented by experimenting with different computer vision techniques.

4.1.7 Track Differences

Analysing the data gathered from testing the models on the tracks, the track that caused the biggest challenge for the models was Snow3. This was expected due to the features included in the track, in particular the inclusion of steeper inclines relative to the other test tracks. Both models were able to navigate sections of road on small inclines as demonstrated by the good performance in tracks Grass3 and Desert3 which both included two small inclines. Larger inclines caused the models to struggle with road following and lane detection tasks, this was most likely due to the difference in orientation of the roads between the pre-processed image when compared to a flat road. To improve model performance on inclines, data with inclined roads should be added to the training data. None of the tracks trained on for either model included inclines.

4.2 Nvidia Paper

4.2.1 Comparison

The 2016 paper by Borjarski et al., found that CNNs are able to perform tasks such as lane detection and road following without the need to breakdown these tasks into their own independent problems. Their CNN model was able to accomplish this without the need for a substantial amount of training data. This report also found those two findings, this was expected due to the use of the same CNN architecture in both the paper and this report.

The Nvidia CNN model was also able to perform these tasks in different conditions, for example the CNN was still able to steer the car in rainy conditions. These findings were not replicated in this report, but this is most likely due to the lack of variety of conditions in the training data set.

A final finding from the paper was that the CNN was able to learn important road features from the steering angle alone, this report also found this to be case.

5 Conclusion

5.1 Findings

In conclusion this report found that CNNs are able to perform tasks such as lane detection, road tracking as well as obstacle detection using a small dataset but that increasing the size of the dataset resulted in an increased performance on those tasks. This was evidenced by the increased performance of Model V2 compared to Model V1 (0.0854 interferences per second compared to 0.1135, respectively).

The report also found that models could self-correct road position without the need for training data by using image augmentation functions.

Unlike the Nvidia paper, this report did not find that CNNs could also perform the tasks of lane detection, road tracking, obstacle detection and self-correction in different environments. However, this was explained by the lack of training data in those different environments and the image pre-processing methods used within this project.

The performance of models within the simulator can be predicted from their validation loss (MSE) value. Models with a lower MSE were found to perform better. (As only two models were used this conclusion should be tested further with the training and testing of more models).

Increasing the quality of data by including data from a larger selection of tracks can improve model performance, evidenced by Model V2's increased performance when compared to Model V1. (Model V2 was trained on 5 tracks compared to 3 tracks for Model V1).

The distribution of the data could affect model performance. Data that has a bias to one direction of steering can however be augmented to remove the bias and reduce the potential effect on performance.

5.2 Critical Review

The key work packages in this project were all completed in the time designated. This project successfully evaluated the performance of end to end deep learning models in a custom made simulator. The car driving simulator section of this project resulted in a custom car driving simulator designed specifically for gathering data and testing models, it also resulted in the completion of 9 tracks for the simulator. All 9 tracks have a day and night environment setting to provide an even larger variety of test data. The simulator was able to send steering angle data to the created Python program.

The application section of this project resulted in the creation of software that could collect data from the simulator, train CNN models and then test those models within the simulator. Two models were created using the application, the limited number of models trained was due to the long training times on the computer used for this project. Total training time increases with the amount of data used, so future models would have taken even longer to train than the models in this project. Despite the lack of models, the evaluation of models was still complete.

References

Albawi, S., Mohammed, T.A. and Al-Zawi, S., (2017), August. Understanding of a convolutional neural network. In 2017 International Conference on Engineering and Technology (ICET) (pp. 1-6). IEEE.

Bojarski, Mariusz & Testa, Davide & Dworakowski, Daniel & Firner, Bernhard & Flepp, Beat & Goyal, Prasoon & Jackel, Larry & Monfort, Mathew & Muller, Urs & Zhang, Jiakai & Zhang, Xin & Zhao, Jake & Zieba, Karol. (2016). End to End Learning for Self-Driving Cars.

Clevert, D.A., Unterthiner, T. and Hochreiter, S., (2015). Fast and accurate deep network learning by exponential linear units (elus). arXiv preprint arXiv:1511.07289.

LeCun, Y., Bengio, Y. & Hinton, G. (2015). Deep learning. Nature 521, 436–444

Lin, G. and Shen, W., (2018). Research on convolutional neural network based on improved Relu piecewise activation function. *Procedia computer science*, *131*, pp.977-984.

Li, W., Pan, C., Zhang, R., Ren, J., Ma, Y., Fang, J., Yan, F., Geng, Q., Huang, X., Gong, H., Xu, W., Wang, G., Manocha, D. and Yang, R., (2019). AADS: Augmented autonomous driving simulation using data-driven algorithms. Science Robotics, 4(28), p.eaaw0863.

Nwankpa, C., Ijomah, W., Gachagan, A. and Marshall, S., (2018). Activation functions: Comparison of trends in practice and research for deep learning. arXiv preprint arXiv:1811.03378.

O'Shea, K. and Nash, R., (2015). An introduction to convolutional neural networks. arXiv preprint arXiv:1511.08458.

Patterson, J. and Gibson, A., (2017). Deep Learning: A Practitioner's Approach. 1st ed. O'Reilly Media, Inc.

Simard, P.Y., Steinkraus, D. and Platt, J.C., (2003), August. Best practices for convolutional neural networks applied to visual document analysis. In Icdar (Vol. 3, No. 2003).

Zafar, I., Tzanidou, G., Burton, R., Patel, N. and Araujo, L., (2018). *Hands-on convolutional neural networks with TensorFlow: Solve computer vision problems with modeling in TensorFlow and Python*. Packt Publishing Ltd.

Appendix

A - Godot Code

Link to the Godot website and documentation, respectively:

- https://godotengine.org/
- https://docs.godotengine.org/en/stable/

Link to the Github repository containing the Godot scripts

- https://github.com/MorganAaronColling/SelfDrivingCarApplication/tree/main/Godot%20Scripts

B - Imported Modules

This project used a range of Python modules to speed up the production of the application, the modules used are summarised in Table 2.

Table 4 – Summary of the relevant Python modules used in this project and their usage

MODULES	IMPORT NAME	DESCRIPTION	PROJECT USAGE
OPENCV	cv2	Used to solve computer vision problems	Pre-processing images before model training
GLOB	glob	Unix style pathname pattern expansion	Finding path locations when saving simulator data
SOCKET	socket	Low-level networking interface	Sending/Receiving data from the simulator
MSS	mss	For taking screenshots in Python	Taking screenshots of the simulator
PILLOW	PIL	Imaging library	Converting MSS image to OpenCV form
NUMPY	numpy	For scientific computing	Formatting and processing numerical and image data
MATPLOTLIB	matplotlib	For visualising data in Python	Visualising data distribution and training effectiveness
OS	os	For interfacing with the computer operating system	Finding and managing paths
RANDOM	random	For generating pseudo-random numbers	Randomising image augmentation and batch generation
IMGAUG	imgaug	For image augmentation	Augmenting images
PYGETWINDOW	pygetwindow	Obtains GUI information on application's windows	Finding the simulator window to screenshot
THREADING	threading	Constructs higher-level threading interfaces	Running the GUI and other functionality simultaneously
PYSIMPLEGUI	PySimpleGUI	Simplifies other Python GUI modules	Creating GUIs for the application
TENSORFLOW	tensorflow	End-to-end open-source machine learning platform	Creating and training the AI/model
SCIKIT-LEARN	sklearn	Machine learning in Python	Splitting the data into a validation set and a training set
SUBPROCESS	subprocess	Running exe files	Running the simulator exe file within the application

Table 2 shows the name the of all modules used in this project; their import name shown in the Python scripts; a description of each module and how the module was used in this project.

C - Python Code

Link to the Github repository containing all the Python code for the simulator:

- https://github.com/MorganAaronColling/SelfDrivingCarApplication/tree/main/PythonCode

D - Extra Results

Table 5 – Testing Data

		Tracks							
		Grass2		Grass3		Snow3		Desert3	
Model	Engine Force	Interferences	Lap Time	Interferences	Lap Time	Interferences	Lap Time	Interferences	Lap Time
Model V1	30	6	152	8	125	35	233	8	169
Model V1	60	14	101	11	85	37	162	13	118
Model V2	30	5	157	5	110	22	237	3	172
Model V2	60	12	100	9	83	26	154	11	112

The raw data collected during the testing of the models shown in Table 5 was used to calculate the values in Tables 2 and 3 in the results section of the project.

PROJECT MANAGEMENT SECTION

6 Project Planning and Resourcing Review

6.1 Time and resources comparison

Table 6.1 - Estimated vs Actual time

	WP1.1		WP1.2		WP1.3	
	Estimate	**Actual**	**Estimate**	**Actual**	**Estimate**	**Actual**
Time (Hours)	2	2	5	4	20	25

Table 6.2 - Estimated vs Actual resources

	WP2.1		WP3.1		WP3.2	
	Estimate	**Actual**	**Estimate**	**Actual**	**Estimate**	**Actual**
Time (Hours)	2	2	0.5	0.5	0.5	3

Table 6.3 - Estimated vs Actual resources

	WP3.3		WP4.1		WP4.2	
	Estimate	**Actual**	**Estimate**	**Actual**	**Estimate**	**Actual**
Time (Hours)	0.5	0.5	2	2	0.5	0.5

Table 6.4 - Estimated vs Actual resources

	WP4.3		WP5.1		WP5.2	
	Estimate	**Actual**	**Estimate**	**Actual**	**Estimate**	**Actual**
Time (Hours)	5	5	2	2	2	2

Table 6.5 - Estimated vs Actual resources

	WP6.1		WP6.2		WP6.3	
	Estimate	**Actual**	**Estimate**	**Actual**	**Estimate**	**Actual**
Time (Hours)	5	10	4	4	3	3

Table 6.6 - Estimated vs Actual resources

	WP6.4		WP7.1		WP7.2	
	Estimate	**Actual**	**Estimate**	**Actual**	**Estimate**	**Actual**
Time (Hours)	6	6	10	10	0.5	0.5

WP1.2 took less time as model were easy to find, WP1.3 took less as the code was repeatable using Godot's scene structure, WP3.2 took longer due to the troubleshooting of code errors, WP6.1 took longer due to the length of the methodology section.

6.2 Project Costing

6.2.1 Assumptions of Costs

- Trainee Engineer, spine point 16, £22417 pa,
- Supervisor, spine point 24, £44045 pa,
- NI of 14% contribution
- Pensions of 8% contribution
- Fixed overheads and utilities of £1000
- Costs over the course of a full working year

6.2.2 Table of Costs

Table 7 – Table of Costs

Description	Cost (£ per annum)	Running Total Cost (£)
Trainee Engineer (TE)	£22,417.00	£22,417.00
Supervisor (S)	£44,045.00	£66,462.00
TE National Insurance	£3,138.38	£69,600.38
S National Insurance	£6,166.30	£75,766.68
TE Pension	£1,793.36	£77,560.04
S Pension	£3,523.60	£81,083.64
Fixed Overheads	£1,000.00	£82,083.64
Total Cost	null	£82,083.64

The total cost calculated for this project was £82083.64. All other resources (not included in the table) for this project were free to use.

7 Risk Response Evaluation

7.1 Risk Register

Table 7.1: Comparative Risk register

RISK		ACTUAL IMPACT	ACTUAL RESPONSE
R1	Free to use models and textures are unavailable for certain simulated features	N/A	N/A
R2	Underestimation of Neural Network training time	Limited the number of models that could be trained.	Trained only two models to be tested
R3	Unable to input the output from the Neural Network to the Car driving simulator using Python	N/A	N/A
R4	Code Troubleshooting	Increased coding time and therefore delayed the collecting of data for the models	Use of online forums to help troubleshoot code so that troubleshooting did not cause further delays.

7.2 Critical reflection

The risk management process of this project was overall a success. As shown in Table 7.1, only 1 of the original risks manifested as well as one unpredicted risk. The risk 2 in the table, was delt with efficiently and although reduced the output of models for this project, did not affect the time taken as was originally expected. Risk 4 was an unpredicted risk and did cause a delay to the project, the reoccurring nature of this problem for different sections of code made it more troublesome in the management of this project. The use of online forums to solve the troubleshooting issues was effective at fixing the issues, although still time consuming.

For future projects that involve coding, time should be dedicated to the design, testing and bug fixing of the code. This would reduce the risk of delays due to unexpected problems with the functionality of the code.