University of Glasgow | School of Engineering

Design Special Topic 5 Report

Implementation of New AI Algorithms Inspired by Deep Learning

Cameron Bennett *2193356B*
Mikkel Caschetto-Böttcher *2198221C*
Andrew Smith *2198699S*
Neil Wood *2197680W*

May 18, 2020

Abstract

Continuing experimentation work performed by Daryanavard and Porr at the University of Glasgow in deep learning applications, this project involved creating a new plugin-based framework and development environment to aid neural network development. Utilising the Robot Operating System (ROS), it allows for both a physical and a simulated robot to be used to test and display the effect of different neural network configurations. A physical robot was designed and two simulated robots were produced as part of this project. With a focus on closed-loop back propagation systems, neural network experimentation was conducted on the simulated robots and a convolutional neural network was developed and tested. The project is hosted on GitHub.

Acknowledgements

We would like to thank Dr Bernd Porr for organising and supervising the project.

We would also like to thank Sama Daryanavard for her continued support and attention throughout the project, especially during this unusual situation. We wish Sama the best of luck with the rest of her PhD.

Being our final project, we would like to extend a thank you to the James Watt School of Engineering and the University of Glasgow for supporting us throughout our degree programmes.

Contents

List of Figures

1 Introduction

Following on from Daryanavard and Porr's project outlined in "Closed-Loop Deep Learning Generating Forward Models with Back-Propagation" [1], this project aims to develop an environment for testing closed-loop algorithms. Line following was chosen as the testing environment because it can be modelled as a closed-loop system with error signals being controlled by line sensors and a camera feed acting as an input. To allow for neural network comparisons, there are several metrics that can be drawn from monitoring the line following process such as deviation from the line. This makes the line-following domain well-suited for closed-loop network experimentation.

1.1 State-Of-The-Art

The existing system was tested using a line following robot which used a camera to view a line ahead and light sensors to provide a reflex action and provide an error signal for closed-loop learning. The closed-loop, shown in Figure 1.1, uses the reflex action for learning. When the system has learnt how to follow the line, the reflex action should not be necessary and will be bypassed as the robot can predict the required movements.

Daryanavard and Porr's work has opened up many interesting areas of potential further investigation; however, there were a number of problems with the development setup. Because of this, there was a desire for a project to take place which

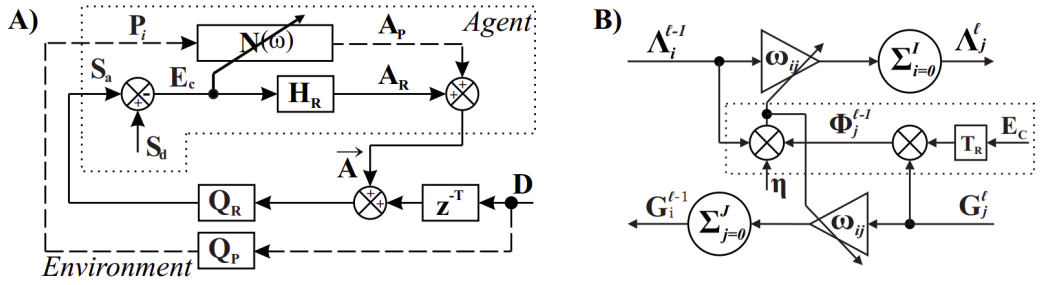


Figure 1.1: *The control loop of the system showing the action of the agent which reduces the reflex action loop (left) and the closed-loop back-propagation operation (right)[1].*

could improve on the existing framework to enable efficient further investigation. The issues with the setup were identified are as follows:

- The existing robot is inflexible: The robot could not be adapted easily to suit changes in experimentation.
- The existing robot is unreliable: The robot had consistency issues related to the line sensor array. Frequent calibration was required and the sensors did not adapt well with changing ambient light conditions.
- Changing parameters in experiments was very time consuming: In order to change parameters between experiments the robot would need to manually have its parameters changed in code and the system recompiled, increasing testing and development time.
- The development system is not flexible: It is currently difficult to add new features to the system or to make changes to the existing system components.

1.2 Project Requirements

As well as tackling the issues set out in Section 1.1, additional requirements were given for the project:

- Development of New Artificial Intelligence (AI) Algorithms: Developments in closed loop error based neural networks should be investigated.
- Graphical User Interface (GUI): A new GUI should be developed to provide convenience to the user to change parameters during experimentation and allow experimental values to be viewed in real-time.
- Investigate integration with ROS: The benefits of system integration with ROS and the use of ROS features should be investigated.
- Gazebo Simulation: A Gazebo (ROS based) simulation environment should be implemented and used for investigation.

1.3 Project Aims

To be successful in achieving the requirements of the project, several activities must be completed. These include updates and improvements to the robot design as well as the development of a framework for deep learning research and testing. The project aims to meet these requirements are summarised below.

1.3.1 Robot Redesign

A new robot should be designed to resolve the issues that had arisen with the old robot while experiments were carried out.

- New light Sensors and sensor setup: The old robot had light sensors that were unreliable and in a fixed position. The new Robot should be set up so that the sensors can be moved. Sensors which provide consistency in all lighting conditions should be adopted for line sensing in the robot. A custom Printed Circuit Board (PCB) should be used to provide rigidity and consistency in placement and lighting.
- Modify to run with ROS: ROS is a middleware used by many robotics applications, integrating the robot's software with ROS will allow the robot to interface more easily with many other robotics platforms.
- GUI Integration: A new GUI is to be developed to allow users of the robot to change parameters during experiments, and to monitor the development system. Integration with the new GUI needs to be achieved.
- Hardware upgrades: As well as an upgrade to the robot chassis to improve flexibility, additional hardware upgrades should be investigated to aid further development of AI algorithms. The simplicity of the design should also be improved, removing the additional microcontroller from the existing design.

These changes to the robot design should make the robot easier to configure, test and debug when developing new neural networks. This will produce a highly customisable and reliable platform on which to test further deep learning systems.

1.3.2 Simulation

Simulations are an important medium for the development of autonomous systems. They allow for experiments to be carried out on the learning system without the need for a physical robot. There are therefore specific requirements regarding the development of simulations for this project:

- Implementation of Gazebo Simulator: A Gazebo simulator should be implemented due to its integration with ROS. This allows the simulator to be easily used with other system components which are updated to use ROS features.
- Integration with the new GUI: Simulation parameters should be easily configurable from the new GUI, and monitoring of simulation parameters should also be incorporated.
- Neural Network Integration: The new simulator should be able to simulate the new AI algorithms to allow for further development and testing.

1.3.3 GUI

There are several features that the GUI requires. These features include changing system parameters and monitoring the performance of the robot, simulator and neural network.

A GUI that facilitates these changes will allow a user to quickly set up experiments and will allow the user to easily view and track the results of experiments all through the GUI.

1.3.4 Neural Network Integration

One of the core requirements for the project is to integrate the closed-loop network proposed in [1] with a flexible testing framework. The model should be able to interact with the environment in the test framework and be able to interface with controls and diagnostics in a GUI. This is intended to help facilitate the research and development of the closed-loop network.

1.3.5 ROS

The existing development framework should be integrated with ROS in order to utilise ROS's features to improve the development and testing of AI algorithms.

1.4 Report Outline

In Chapter 2, this report begins by describing ROS and how it was utilised in the project. The GUI and the configuration features developed for the project are then presented in Chapter 3. The physical robot which was designed is described in Chapter 4, followed by the simulated modelling of this robot in Chapter 5. Following this, the integration of the neural network and modifications are discussed in Chapters 6 and 7 respectively. The project can be found on GitHub [2].

2 The Robot Operating System

ROS [3] is middle-ware designed for use with robotic applications. It provides a communication framework for integrating different robot-related processes. ROS encourages modular design, by providing an abstraction layer that can be used to interface with different components.

2.1 ROS Features

A ROS system is comprised of nodes, each one representing a process. For example, a node might be responsible for controlling motors on a robot or obtaining sensor values.

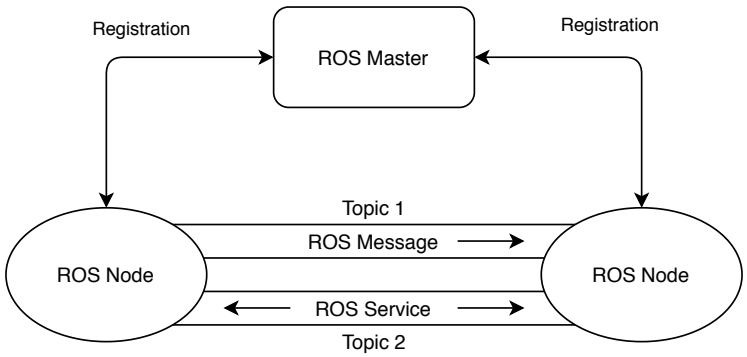


Figure 2.1: *Basic ROS Architecture.*

The ROS communication framework provides dedicated data pipelines known as topics. Topics are used to direct both ROS messages and services. Topic access management is performed with the ROS Master, a single, central control module which is required for any ROS application. Nodes have to register with the ROS Master, which allows the ROS Master to act as a platform for connecting different nodes. In this way, nodes located on different machines can still be connected. It should also be noted that multiple nodes can connect to one topic, allowing the broadcast of a single message to multiple nodes.

Communication on a topic is achieved using ROS messages and services. A node can either publish a ROS message on a topic or subscribe to a message on a topic. The publishing node simply sends out its message and continues with its

other tasks. A node or nodes subscribed to that particular message on that topic, implement callback functions to handle the message received event.

A ROS service acts in a slightly different way. ROS services are a more direct link between nodes, where a node can either act as a service client or server. The client node will send out its service request and then wait for the response from the server node. The server will again implement a callback function to handle the service request and then send its response.

Both ROS messages and services can be constructed from different message types. These can be standard message types or user-defined, custom message types. This allows multiple ROS messages and services to exist on a single topic, as long as they are defined to have different message types.

2.2 Project Implementation

ROS has been used in this project to provide a framework which easily allows development and testing of neural network implementations with a line following robot.

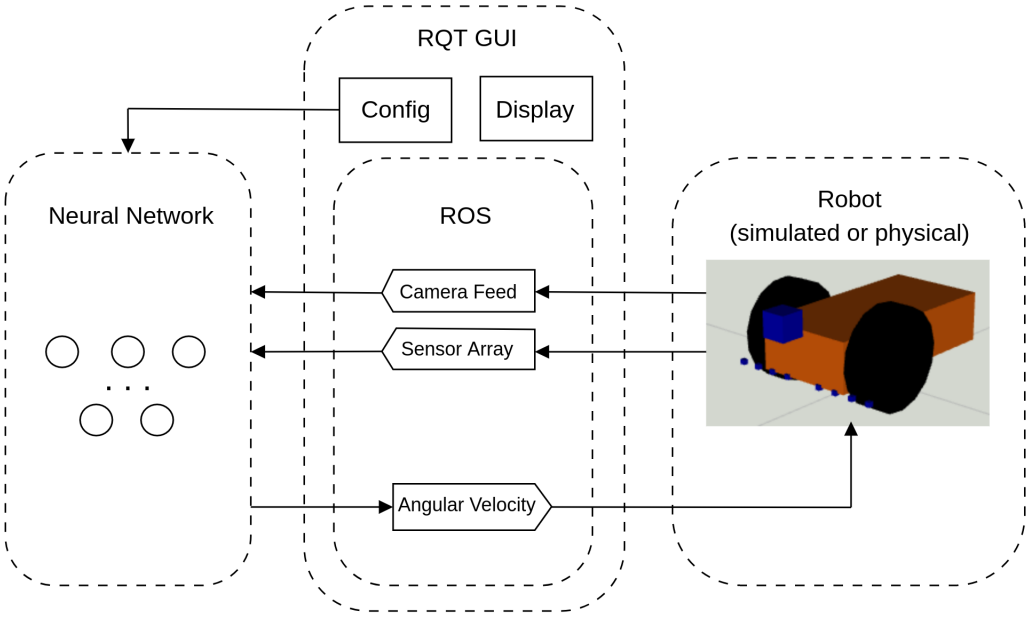


Figure 2.2: *The system diagram showing the system modules and ROS control system.*

2.2.1 Modular Design

The abstraction layer provided by ROS allows a modular design approach. By specifying interface requirements for each component, the framework allows different components to be easily switched in and out, without the need to alter other system components to be compatible. The interface specification is derived from the interaction between modules, as shown in Figure 2.2.

There are three main components supported by the framework: the robot component, the neural network component and the GUI component. The robot component can be either a physical robot or a simulated robot. Both access the same neural network implementation which makes it extremely easy to switch between a simulation and a physical robot for testing a network implementation.

2.2.2 Custom GUI

The framework also provides a GUI that can be used for all components. This provides the user with a consistent environment to control and monitor the robot and neural network, which is highly convenient.

The GUI was developed using the existing ROS based GUI framework, ROS Qt (RQt) [4]. RQt is highly modular in nature, due to its use of a plugin-based system. This means the user can plug in GUI components to tailor the GUI experience to their needs. RQt provides a set of existing plugins but also allows the user to create custom plugins.

Another nice feature RQt provides is the ability to organise the GUI by moving and resizing the plugins. Different GUI configurations can be saved by the user as a perspective. The custom RQt plugins used for this project are detailed in Chapter 3.

3 RQt Plugins

As described in Section 2.2.2, RQt [4] allows the construction of a custom GUI by using plugin components. This chapter describes briefly the plugins used for this project, including custom plugins, made specifically for this application.

3.1 Line Sensor View

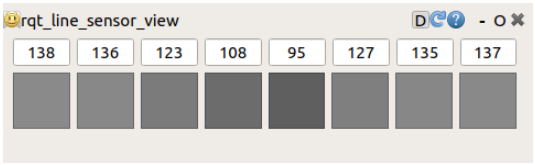


Figure 3.1: *Line Sensor Viewer RQt Plugin.*

This plugin is designed to monitor the state of the robot's line sensor array in real-time. It displays both the raw value of the sensor and a representation of the value on a grey-scale, ranging from black to white. This gives a nice visual representation of how the sensors are responding. The plugin subscribes to the ROS topic of the robot component which publishes the sensor values as a ROS message.

3.2 Line Sensor Control

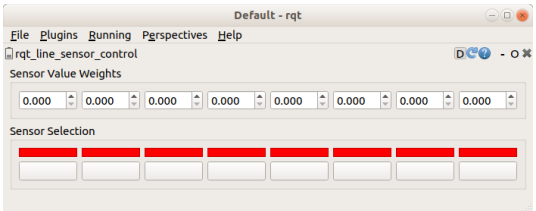


Figure 3.2: *Line Sensor Control RQt Plugin.*

This plugin is used to set line sensor related configuration parameters for the neural network module. A weight for each sensor can be defined which is used as a multiplier to determine how much each sensor contributes to the closed-loop error of the system.

3.3 Neural Network Configuration

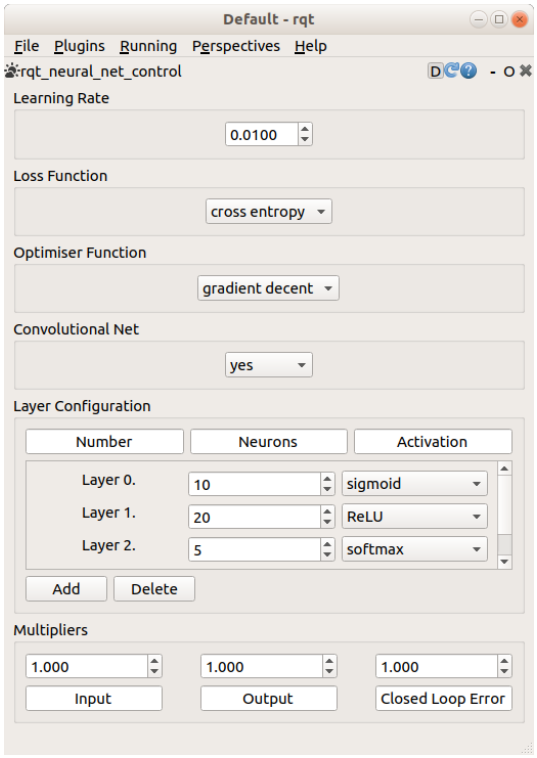


Figure 3.3: *Neural Network Configuration RQt Plugin.*

This plugin is used to set a range of configuration parameters for the neural network, as shown in Figure 3.3. It allows the user to add and remove layers to build up a network architecture, while also selecting an activation function for each layer. The configuration data held in the plugin is retrieved by a ROS service request when the neural network package is launched. The request uses a custom message type defined by the plugin package.

3.4 Neural Network Diagnostics

This plugin is designed to monitor the performance of the neural network in real-time. As shown in Figure 3.4, the motor command output from the neural network is displayed visually as a dial representing a steering-wheel for the robot. The values of the network output node and closed-loop error are also displayed.

3.5 Miscellaneous

As well as the custom plugins designed specifically for the project, some existing plugins are also extremely useful for the application.

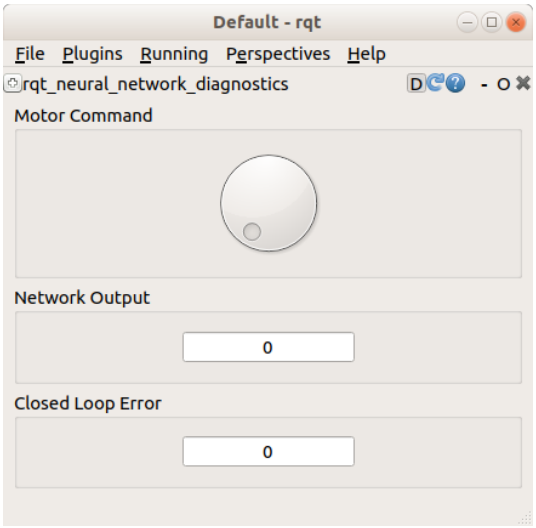


Figure 3.4: *Neural Network Diagnostics View RQt Plugin.*

The "Image View" plugin is used to display image ROS messages. This plugin is primarily used for displaying the camera feed from the robot in real-time. The plugin can also be used for viewing the kernels of the convolutional neural network as shown in Figure 3.5. These weights are updated as the network learns which can help the user to understand what features are being learnt and whether the network is still making updates.

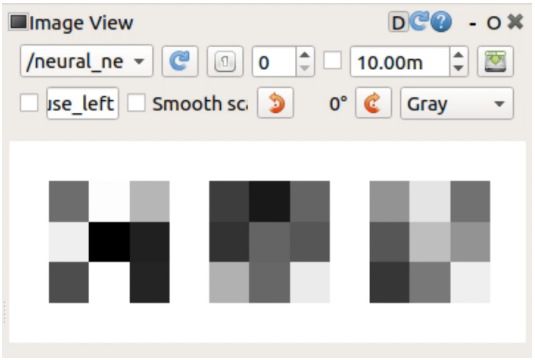


Figure 3.5: *Visualisation of convolutional layer kernels in RQt.*

Another useful plugin is the "Plot" plugin. This provides a real-time plotting capability which can be used for visualising data in the system. For example, it can be used for visualising the closed loop-error produced by the neural network.

Plugins are also available to allow the user to more easily interact with ROS processes. For example, one plugin allows the user to publish on ROS topics without using the terminal, which can be useful for debugging.

4 Physical Robot

4.1 Robot Architecture

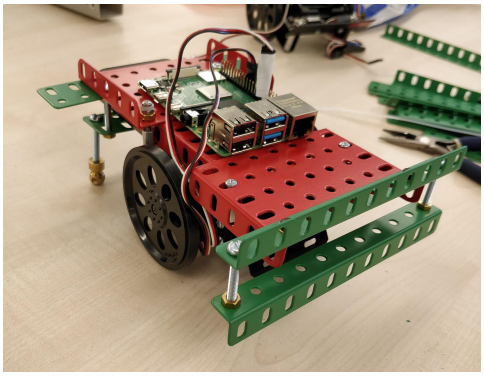
One of the aims of the project was to improve the design and performance of the physical robot used for testing the AI algorithms. A set of specifications was produced to achieve this:

- **Physical Build:** The robot had to have a robust chassis, capable of securely housing and mounting key components such as the Raspberry Pi, battery pack, camera and line sensors. A flexible design was desirable to allow for easy changes to the robot build.
- **Line Sensors:** The sensors had several requirements including high immunity to ambient and environmental interference, an easy interface to communicate with the Raspberry Pi, a large dynamic range and flexibility in the sensing distance from the line.
- **Reduce Components:** The design was to reduce the number of required components to simplify the system. This would mean all system components including motor control and communication with sensors would all be achieved using just a Raspberry Pi.
- **Wireless Access:** The physical robot would incorporate some form of wireless communication interface for real-time control and monitoring.
- **ROS Compatibility:** The robot would have compatibility with the ROS based framework, using the same interface as the simulations to ensure the modularity of the framework.

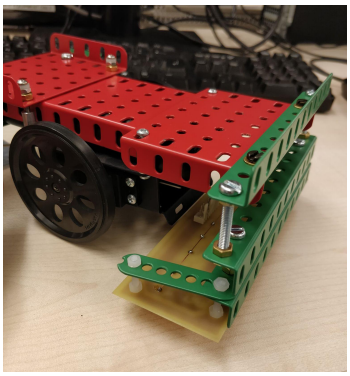
4.2 Robot Chassis

A Sumobot [5] chassis, with continuous rotation servo motors, was used as the base for the physical robot construction. It was considered suitable for the application as it had been used in previous, related projects, and would reduce the time required to produce a working prototype.

As shown in Figure 4.1, Meccano was mounted on top of the Sumobot base. The use of Meccano produced a very robust and secure build, providing mounting for the battery pack underneath, raspberry pi on top and the line sensor PCB on



(a) The robot chassis with the Raspberry Pi.



(b) The robot with line sensor PCB.

Figure 4.1: *The robot with a Meccano chassis and a Raspberry Pi (a) and the line sensor PCB (b).*

the front. A ball bearing was placed at the back to reduce steering friction and provide stability.

The use of Meccano provides flexibility meaning the robot can be dismantled and changed easily. The position and height of the line sensors are easily adjustable. The camera can also be mounted very easily, allowing the camera position (height and angle) to be changed.

4.3 Line Sensors

The line sensor PCB was designed in accordance with the specifications outlined in Section 4.1.

As shown in Figure 4.2, the sensor board uses 8 analogue sensors with an 8-channel Analogue to Digital Converter (ADC). This ADC uses 12 bit resolution and can be read using Serial Peripheral Interface (SPI) making it compatible with the Raspberry Pi.

The sensitivity and dynamic range of the sensors can be set by changing the resistors γR and xR as shown in Figure 4.2, meaning the setup is quite flexible to change if required.

A total of 4 pairs of Infra-Red (IR) sensors are used, each a different distance from the centre point where the line will ideally be. This provides the required flexibility to change the reactive error element of the whole system to use sensor pairings at different distances from the line.

Using IR sensors provides high immunity to ambient and environmental interference. The operating wavelengths, as seen in Figure 4.3, are chosen to be 850 nm. With correctly calibrated sensors, this setup was shown to have extremely high immunity to ambient and environmental interference when tested and worked

well at differentiating between the black line and the surface it was on.

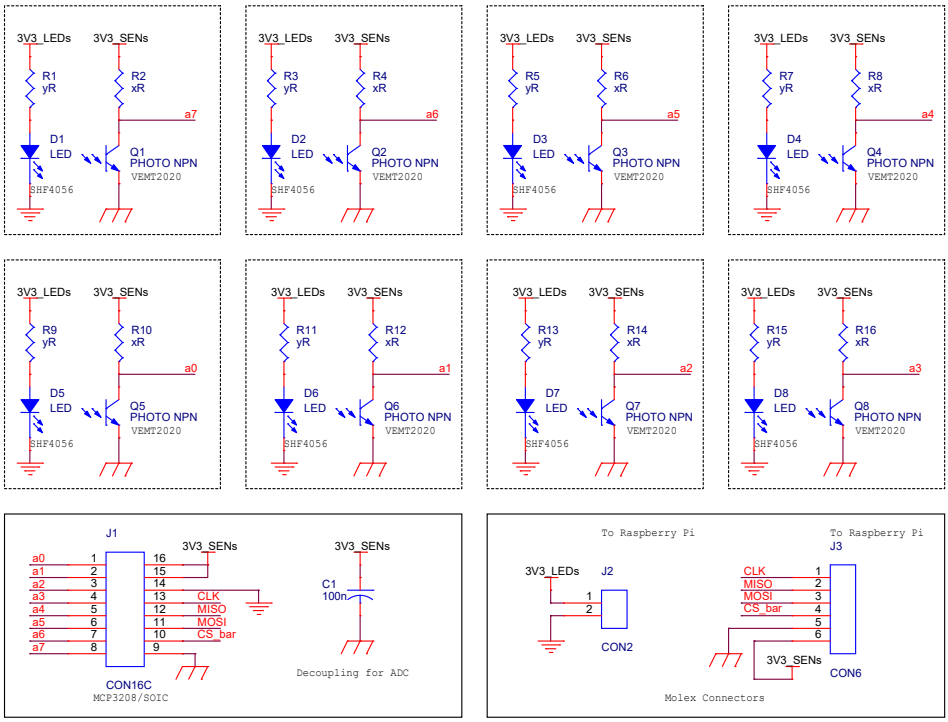
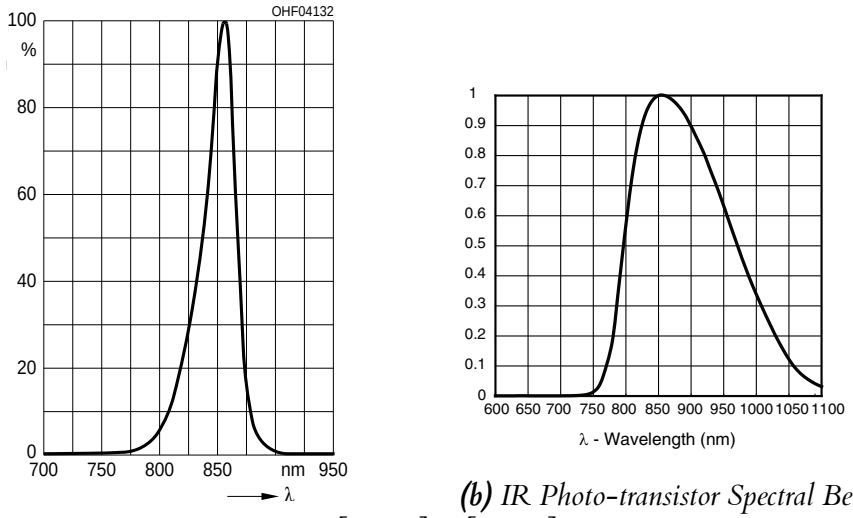


Figure 4.2: *Line Sensor Board Schematic.*



(a) *IR LED Emission Spectrum [6, p. 3].*

(b) *IR Photo-transistor Spectral Behaviour [7, p. 3].*

Figure 4.3: *Spectrum matching of (a) SFH 4056 IR LED and (b) VEMT 2020 Photo-transistor.*

4.4 Raspberry Pi Camera Feed

A program was written using OpenCV [8] to modify the Raspberry Pi camera feed. It was required to make it suitable for input into the neural network by reducing the pixel count and by converting it to grey-scale. The program took the High Definition (HD) camera feed output and converted it to grey-scale before pixelating it to a lower resolution as required. The resultant image feed could be represented as an array of integers which indicate how dark the area in front of the robot is. This meant that it could be directly processed by the neural network. An example can be seen in Figure 4.4, showing the reduced resolution camera feed.

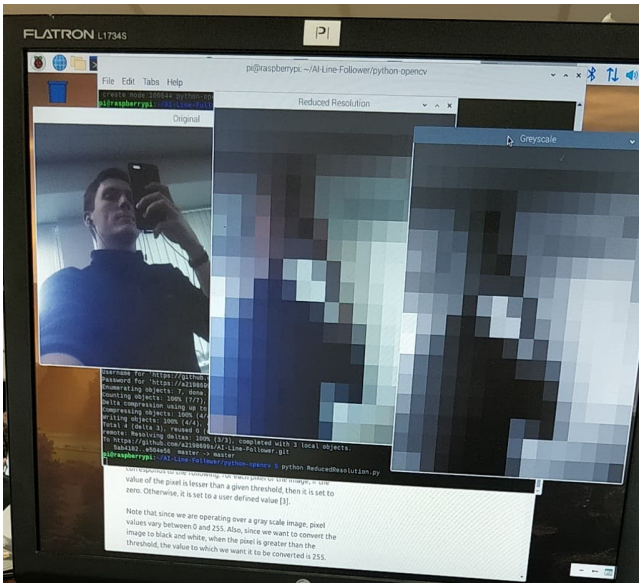


Figure 4.4: *Reduced Resolution Raspberry Pi Camera Output*

4.5 Further Improvements

Unfortunately, the physical aspects of this project could not be fully completed due to the closure of the University. The required remaining development of the physical robot is detailed in this section.

4.5.1 Finalise Construction

One of the requirements was to simplify the design of the robot by removing the need for an additional microcontroller to control the robot's motors and obtain values from the line sensors. Therefore, all components are connected to the Raspberry Pi header, the configuration of which is shown in Figure 4.5. A connector was produced to remove the hassle of attaching individual wires,

however, the exact setup could not be fully documented in time before lab closures.

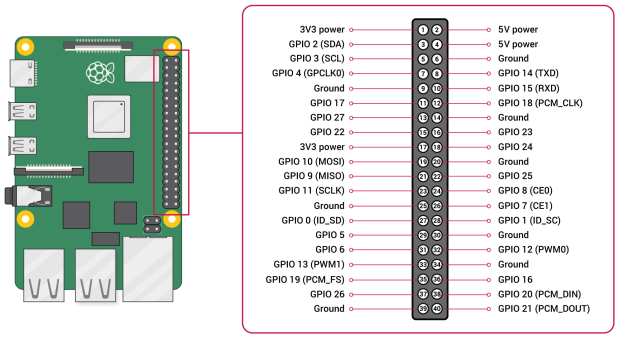


Figure 4.5: *Raspberry Pi GPIO Pinout [9].*

Furthermore, to finalise the construction of the robot, the camera needed to be mounted to the robot chassis. The hardware required for remote communication with the Raspberry Pi would also need to be sourced and mounted.

4.5.2 Remote Communication

Remote communication was a requirement for the physical robot. It was planned to use a WiFi transceiver on the robot to connect with a control computer. The user could then use Secure Shell (SSH) to access the Raspberry Pi. A WiFi connection would also support the use of the ROS framework over multiple machines.

4.5.3 ROS Compatibility

The software for the physical robot was not fully completed and tested. ROS packages to drive the motors and read the line sensor values have been produced, but have not yet been tested. Software for obtaining the camera readings would have to be integrated, and it would have to be ensured the software ROS packages are compatible with the same ROS interface specified and used for the simulated robot modules.

5 Simulated Robot

To test the effect of the neural network on the behaviour of the line-following robot, two ROS-based simulation environments were setup. Simulated testing became the focus of development for the project to allow for remote testing in the line-following environment. This section describes the two simulation models of the robot and their corresponding line-following environments. The Gazebo simulation is described in Section 5.1 and the Enki simulation is described in Section 5.2.

5.1 Gazebo Simulation

The first simulated model of the robot was designed using the Gazebo environment [10]. Gazebo is fully integrated with the ROS and this is how the simulation was controlled and monitored. It allows for accurate 3D models of robots within defined environments. Gazebo components, including robots, environments and sensors, are described using XML.

5.1.1 Gazebo Robot

The Gazebo robot was modified from an example in a Generation Robots tutorial [11]. This provided the construction of the body, wheels and camera which can be seen in Figure 5.1. Several modifications were made to this robot to allow for line following using a neural network. The front-facing camera was modified to have a raised position, pointing downwards to allow for the upcoming path to be visualised. This allows the robot to interpret the upcoming paths and make decisions based on the path which can be seen. The camera position and angle can be further modified for testing. The array of eight line sensors were modelled closely to the behaviour of the physical sensors which produce an integer based on how light or dark the area is under the sensor.

Each of these sensor types were implemented in Gazebo using a standard camera module. For the front-facing camera, the image resolution was set to 9x9 pixels and the image was set to gray-scale to allow for input straight into the network. An example of the camera image produced is shown in Figure 5.2. As for the line sensors, the grey-scale information that needed to be conveyed is best done using a single-pixel which acts as an average over the span of the camera. Example

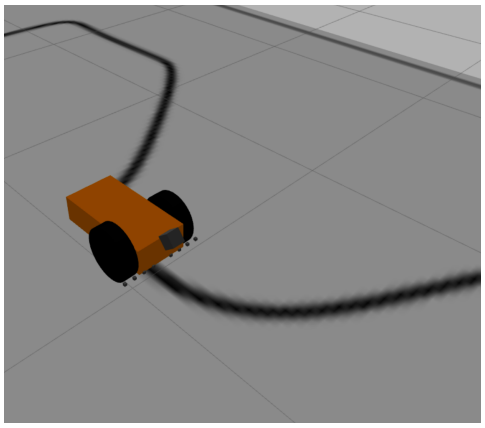


Figure 5.1: The Gazebo robot model, modelling the line sensors, front-facing camera and the wheel actuators of the physical system, in a Gazebo simulation environment.

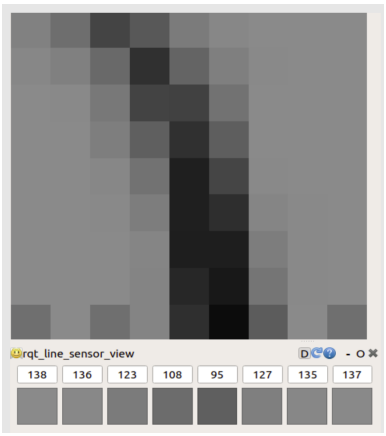


Figure 5.2: Example output of the 9x9 pixel simulation camera visualising the track and the line sensor array showing the line underneath the robot.

sensor readings can also be seen at the bottom of Figure 5.2. The darkness of the pixel represents whether or not a line is present at a given sensor. This value is represented as an integer in the same way as for the physical sensors.

5.1.2 Gazebo Environment

The Gazebo world provides the robot with a testing environment. This is where the track was produced. It was set up to allow for any image to be used as the ground plane. This allowed for different line tracks to be used easily and for the response of the robot to be observed and monitored. An example simulation environment, containing a black line track, is shown in Figure 5.3.

5.1.3 Rviz

Rviz is a visualisation tool which can be used alongside Gazebo simulator. It tracks the information of active ROS topics to provide information about sensors and actuators on a given robot. The Rviz representation of the actuators and sensors is shown in Figure 5.4 where the red lines indicate the direction of the sensors. The array of line sensors at the front of the robot, configured as single-pixel camera sensors, are pointed down towards the ground to provide information about the track position under the robot. The camera is placed above the robot at an angle towards the ground, replicating the setup for the hardware robot and allowing it to clearly see the path ahead.

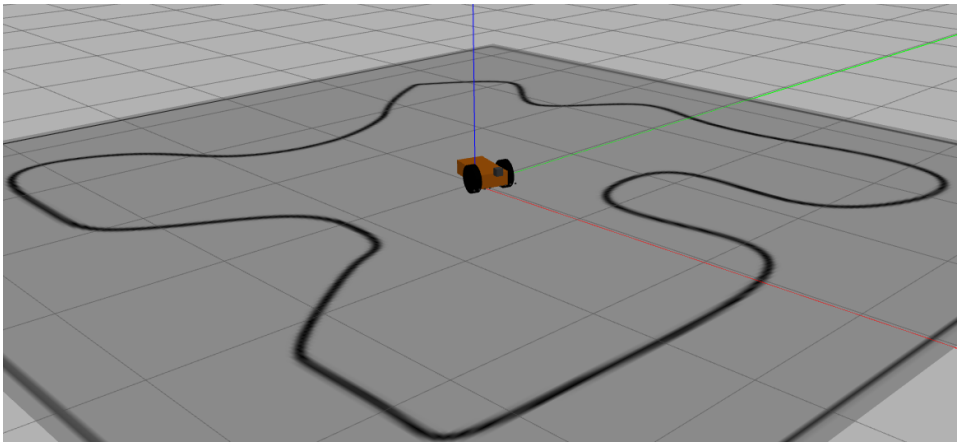


Figure 5.3: *The Gazebo line-following environment showing the robot and an example black line track used for testing.*

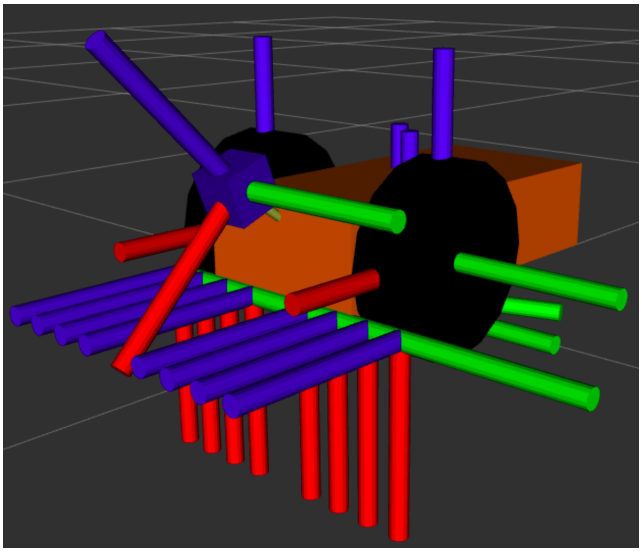


Figure 5.4: *Rviz representation of actuators (wheels) and sensors (line-sensors and camera).*

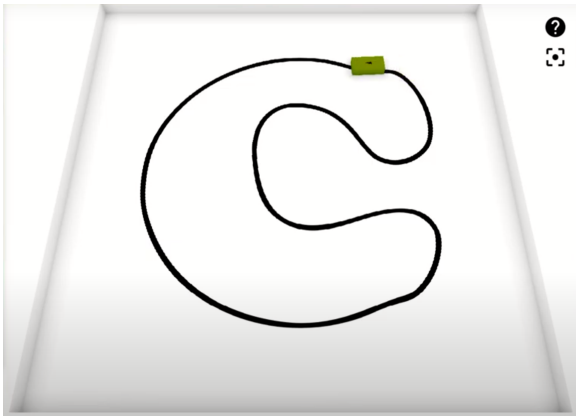


Figure 5.5: The Enki robot model in an Enki sandbox simulation environment.

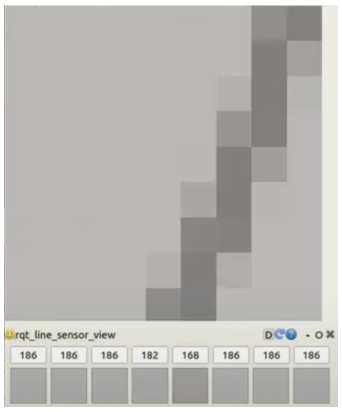


Figure 5.6: Enki sensor visualisation in RQt.

5.2 Enki Simulation

Enki simulator is open-source and provides a light-weight 2D simulation environment [12]. It provides a sandbox environment to allow for the simulated robot to be monitored as it explores. In a similar way to Gazebo, it allows for an environment to be defined as well as a robot with a variety of sensors.

5.2.1 Enki Robot

The Enki robot is visually represented in the simulation by a rectangle. This can be seen in Figure 5.5. The sensors are set up to be the same as in the Gazebo setup. This allows for the same RQt visualisation to be used, see Figure 5.6. These sensor visualisations are taken as the robot is in the position shown in Figure 5.5 as the track moves off to the right in front of the robot.

5.2.2 Enki Environment

The Enki environment can also be configured with a line track image. Examples can be seen in Figures 5.5 and 5.7. The simplicity of the simulator and robot setups means that the simulator can be run more quickly than the Gazebo simulator. This is useful in reducing the run-time of simulated testing.

5.2.3 ROS Integration

The Enki simulator was already adopted by the project was set-up directly to work with the neural network; however, it is not set up to work directly with ROS commands. The simulator QMake build, authored by Daryanavard [13], was modified to accept ROS commands by including our ROS-based framework in a CMake build system. This allowed for the alternative Enki simulation

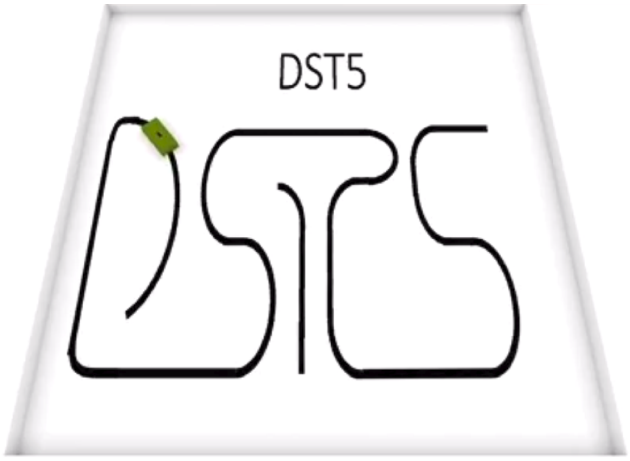


Figure 5.7: *The Enki simulator with a customised line-follow environment.*

environment to be plugged into the framework with minimal changes.

5.3 Simulation Conclusion

Each of the simulation models created plug directly into the ROS-based framework developed for this project. Gazebo provides a powerful and realistic 3D simulation environment. A robot was modelled closely to the behaviour of the physical robot which can be visualised. Running the environment is computationally demanding, recommending use with a high-performance GPU, which meant that obtaining simulation results was difficult. Enki simulator provided a light-weight 2D simulation environment. This was useful as it meant that results could be obtained more quickly. The adoption and integration of each of these simulation environments into the ROS-based framework, which was created in this project, proves the portability of the network. The framework allows any simulated model with the correct configuration to be plugged-in and to interface with the neural network for testing.

6 Neural Network Integration

Artificial Neural Network (ANN)s have become a powerful tool for regression and classification in a wide variety of applications. However, they are most commonly adopted as static networks that have been subjected to a training phase that uses a predefined dataset before being deployed. In contrast, the network architecture proposed by the research team adopts an iterative data collection and training cycle that means the system is trained as it traverses its environment. The closed-loop nature of the proposed network poses many challenges that do not arise in conventional neural networks. This section describes how the ROS-based framework was designed to incorporate closed-loop networks.

6.1 Neural Network Interface

A key theme of the framework so far has been the modular approach that has been adopted and this continues with the integration of neural networks. While the core aim for this project was focused on the incorporation of the closed-loop network, future research will no doubt benefit from the ability to add new deep learning architectures and libraries. With this in mind, a neural network interface was developed as a consistent way to interact with the framework and simulations.

The interface was designed with an abstract *NeuralNetworkInterface* class that contains a standardised set of virtual methods. This allows users to create their own subclasses with customised network construction, forward passes and training which makes it simple to integrate different architectures and libraries with the framework. The UML diagram in Figure 6.1 gives an overview of the functions to be implemented in subclasses and the flexibility of being able to add custom behaviour for each type of architecture.

The interface contains a set of callback functions that are triggered by updates to ROS topics. The neural network's training cycle is carried out in the camera callback function meaning that the weights are updated with each frame of the image stream. In parallel, the sensor values in the interface are also updated by ROS topic callback functions and these can be used by the neural network as ground truth. The closed-loop error is calculated in this callback by comparing the difference between pairs of sensors equidistant from the centre of the line follower.

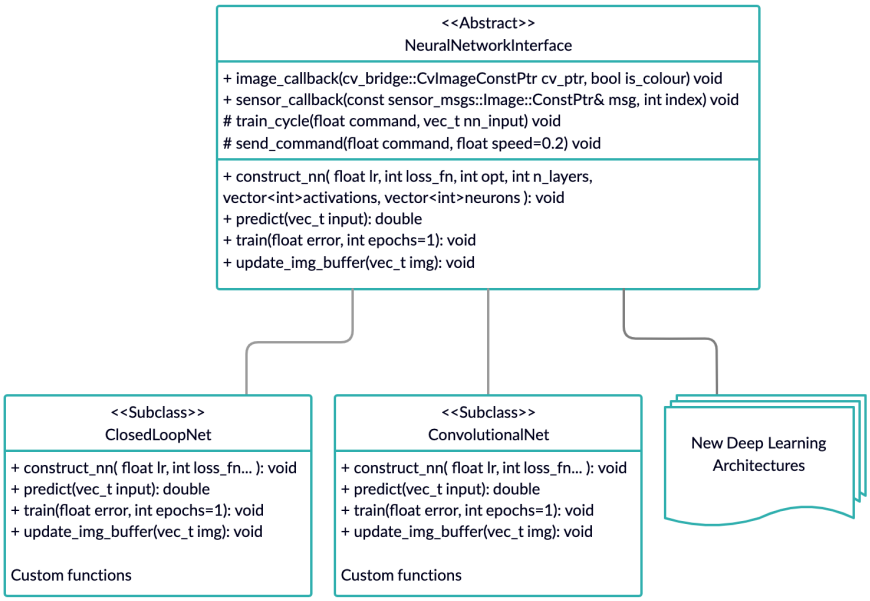


Figure 6.1: *UML diagram of neural network interface.*

6.2 Data Augmentation

In order to maximise the performance of the networks, the input data is conditioned before being used for training. The first step of data processing is to convert the 3 RGB channels to a single monochrome image as this will reduce the dimensionality and a single channel contains all the necessary information for this problem. The following step is to invert the input data. Naturally, pixels are represented by 0 for black and 255 for white. However, in this test environment, the feature of interest is the black line that the robot is to follow so images are inverted in the input to make these lines have positive values. This also results in very sparse activations of the network as the input image is predominantly filled with white pixels which will now have a value of 0. This avoids the unnecessary activation of neurons in background white space that doesn't offer distinctive features and therefore making the network more efficient. This applies well to a simulation environment where background pixels are guaranteed to be consistently white. In a live environment, an averaging stage followed by thresholding could be included to set the background surface to have a value of 0.

Following inversion, the input data is also normalised before being fed into the neural network. As stated before, the input image contains values from 0 to 255 indicating the brightness of each pixel. However, including values that are significantly larger than 1 can make it more difficult for the network to converge to an optimum state due to the large fluctuations in the network caused by the range of these numbers. In general, it is better to maintain input values between

the range of -1 and 1 to optimise the network performance and for images, it is common to use a range of 0 to 1. Hence, the input is normalised between 0 and 1 with 0 representing white and 1 as black after inversion. Using this range of values also helps to prevent explosive multiplication in deep networks that could result in inaccuracies caused by overflow.

6.3 Timing Challenges

In conventional applications of neural networks, the output label is immediately available for comparison after the network has completed a forward pass. However, in a closed-loop system, the action resulting from the forward pass will not be represented in the environment until after a certain time delay. For example, in the line follower application, the steering commands will not be represented by the sensors until the command has had time to have been executed and the robot has moved to the part of the line represented by the predictor camera. Therefore, a time delay must be incorporated to align the input camera image with the correct sensor readings.

6.3.1 Circular Buffer

In order to align images and sensor readings caused by this time delay, a circular image buffer was employed. Every time that the image callback is triggered, rather than feeding an image straight into the neural network, it is put on the image buffer. The read/write index of the buffer is then incremented and the network is trained on the next image which was captured *buffer length* frames beforehand. At the outset, there is a delay before the neural network begins training as it waits for the buffer to fill. During this stage, only forward passes are carried out in the network to get commands with no updates to the network's weights. The commands are also stored in a buffer as a reference to use as a label if the network in the subclass requires this.

In each iteration, a forward pass is performed using the current image frame to attain the command to be carried out at time t_0. This results in a steering command that is sent to the line follower and the input image is stored in the buffer. Following this, sensor values are read and are used to calculate the closed-loop error. An image from time $t_{-buffer\ length}$ is input to the network and a forward pass is completed followed by a backward pass using the closed-loop error.

It is important that the predictor image is correctly aligned with the sensor values and this makes the choice of buffer length very important. There are four factors that impact the choice of buffer length: the speed of the robot; the distance between the camera's field of view and the sensor array; the time it takes for motor controls to be carried out; and the frame rate of the camera. The frame rate of the camera is fixed at 30Hz meaning that each index in the image buffer

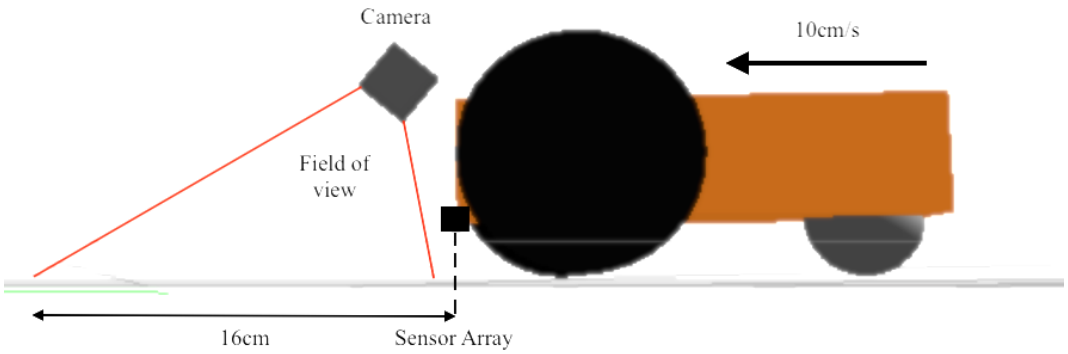


Figure 6.2: *Side view of line follower visualising the camera angle and distance between sensor array and the top of the field of view.*

corresponds to a time delay of a thirtieth of a second. The speed has been set at 10cm/s and the camera's field of view extends 16cm in front of the sensor array. The values for these factors are visualised in the diagram in Figure 6.2.

Adjusting the buffer length will alter which part of the image is most influential to the neural network's output. If the buffer is very long, then the neural network should learn that the top row of pixels are most influential and vice versa for a small buffer. However, in order to make the most of the input image, the time delay should allow for both predictions and time for commands to have an effect so a value in the mid-range is best suited. A buffer length of 14 was chosen after testing which can be approximated to a delay of 0.47s or 4.7cm by the following relationships in Equation 6.1. This length was found to provide the required amount of time for a command to take effect at a speed of 10cm/s.

$$t = \frac{L}{f}, \quad d = v\frac{L}{f} \tag{6.1}$$

where t = the time delay caused by the buffer, d = the corresponding distance travelled in that time from sensor array, v = the velocity of the robot, L = the buffer length and f = the frame rate of the image data.

6.4 Control Flow and Feedback

When the network package is called, a service request is sent to the ROS topic containing the network parameters chosen in RQt. The network is then built using these configuration parameters so that training can begin. Before the network is trained, the images are processed and added to the buffer which must be filled before training. Additionally, the sensors at the edge are checked for high pixel values so that the robot can have a reflex response if it deviates too far from the line. As the network learns, it publishes information to ROS topics to allow visualisation of the network's progress in real-time. This helps the user to debug the network as it learns and visualise it's weights. Examples of these are shown in Chapter 3.

6.5 Summary

The neural network interface described in this section allows future users to easily integrate their custom architectures and libraries with the ROS-based framework. This interface sets a consistent integration format and training regime that allows users to easily switch between network architectures while testing.

7 Neural Network Adaptations

In addition to including the requested closed-loop neural network, another model has been built and integrated with the ROS-based framework. This architecture has been designed to act as a benchmark for any advances made in the development of the closed-loop network. This has also demonstrated the simplicity of using the neural network interface to integrate a new model with the framework. This benchmark model must adopt state of the art design principles to accurately evaluate the significance of any research developments in the closed-loop network. This section will describe the rationale behind the design choices made while building this neural network.

7.1 Convolutional Neural Networks

The first adaptation that was adopted in the benchmark network was the use of convolutional layers. The main motivation behind using these was to improve the speed of convergence and efficiency of the network. One of the desirable feature requests of the research team was to be able to use an entire image rather than a sampled version of the input. This will be helpful for when the closed-loop testing extends to problems where fine details are of vital importance. A dense network would struggle to meet the time deadline from the image frame rate and would have very slow convergence with so many pixels at the input.

7.1.1 Efficiency

Convolutional layers operate by convolving a matrix of values, known as the layer's kernel, over the input data. As the network begins training, these kernels are updated to represent important features in the input that usually provide a good indication of the correct output. For example, in the line follower, a kernel may be adapted to detect a right turn by using a diagonal shape. This is depicted in the example in Figure 7.1. With this example, a strong response will be triggered when the line turns right. This reduces the dimensionality of this feature from 9 pixels to a single input for the next layer in the neural network. This means that rather than updating weights for an entire input of 81 pixels, the network only has to update the 9 weights in the kernel, which makes it more efficient. Of course, the network needs to be able to detect more

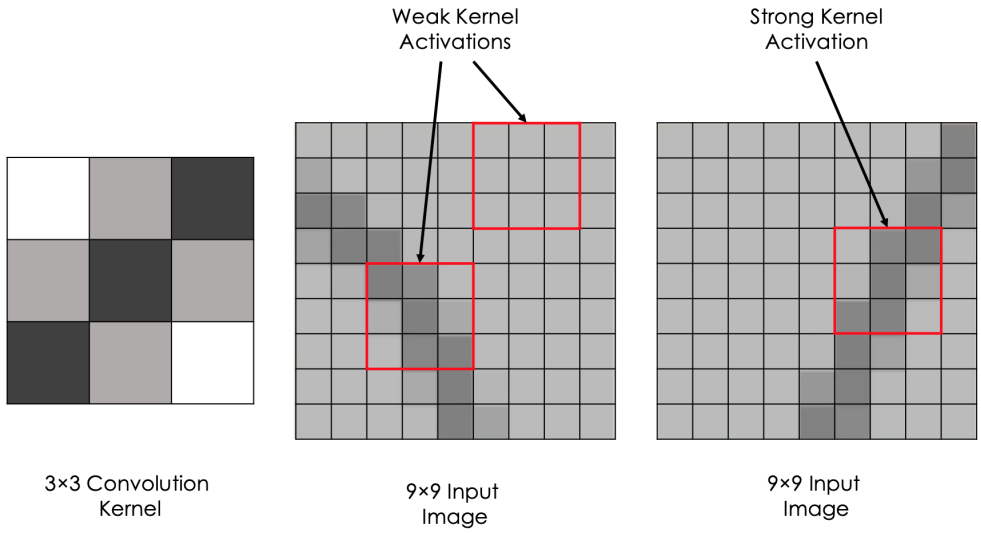


Figure 7.1: *An example of a simple kernel (left) in a convolutional layer to detect a right turn. Two examples of input images from the simulation environment are given to show how this kernel would respond to input images.*

than just right turns in the input image so multiple kernels can be used in the convolutional layer. For this problem, 3 kernels were used with the intention that each of them would become receptive to left turns, straight lines and right turns. Even with the use of 3 kernels, the convolutional layer at the input only has 27 weights to update, less than the 81 inputs that would need to be trained with a fully connected layer. This reduction in trainable parameters makes the network more efficient and helps the training time stay within the deadline of 30 fps.

7.1.2 Accelerated Convergence

Convolutional layers also help the network to converge faster. In addition to having fewer weights to update, the kernels in these layers possess an immediate association between adjacent pixels from the outset of training. In a fully connected layer, each pixel is treated as an independent feature and the network has to learn which pixels are adjacent and correlated. The nature of pixels in an image means that they are always linked to their surrounding neighbours and so using convolutional layers can avoid unnecessary and inefficient training to identify these relationships. As mentioned, the kernels use a collection of pixels to determine a single 'right turn' feature for example, which immediately ties these pixels into a single feature for the following dense layers.

7.1.3 Location-Invariant Feature Extraction

The final motivation for using convolutional layers is that they make the network more robust to alterations in line position or camera angle. By convolving the kernels over the entire input image, they can extract the corresponding feature regardless of where it appears in the predictor camera. For example, a right turn can be extracted at the top of the image or at the bottom. This avoids the network becoming overly dependant on a specific region of the input and helps it to focus more on the features that occur in the image as well as where they are.

7.2 Vanishing Gradients

One of the key issues highlighted at the beginning of this project was to do with limitations on the network's capabilities caused by vanishing gradients. This is a problem that can occur in deep networks where the optimiser is no longer making any updates to some or all of it's neurons. In each epoch of the training cycle, a forward pass is performed in the network which results in a prediction at the output neurons. This prediction is then compared to the label using a loss function and this results in a prediction error to be back-propagated through the network.

The objective of the network's optimiser is to minimise the loss provided by the cost function and so each iteration of the training cycle should aim to update the weights in a way that minimises this metric. To accomplish this, the network must know which direction each weight should be updated in and the magnitude of the update that should be made. Therefore, a Jacobian matrix of partial derivatives is formed containing information on how a small change in each weight, ∂w impacts the result of the loss function, L. Consider a simple network with just one neuron in each layer. These partial derivatives are calculated using the chain rule as follows:

$$\frac{\partial L}{\partial w_i} = \frac{\partial z_i}{\partial w_i} \frac{\partial a_i}{\partial z_i} \frac{\partial L}{\partial a_i} \tag{7.1}$$

where L is the loss function, w is the weight at layer i, z is the input to the activation function and a is the output of the activation function at layer i. These variables are visualised in Figure 7.2.

This diagram is simplified to a single layer but it can be extended by including the same pattern and repeating the chain rule for preliminary layers. The multiplicative behaviour of the gradients caused by the chain rule means that the gradients can be sensitive to very large or very small numbers that will either explode or dissipate the gradients very quickly. The latter is what is referred to as vanishing gradients and is ultimately what prevents the optimiser from getting useful gradient information.

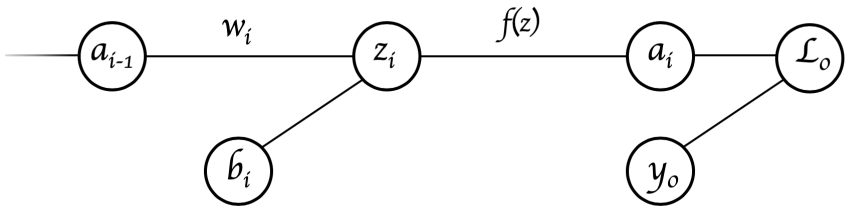


Figure 7.2: *Components in a single layer representing a simplified view of backpropogation parameters using a single layer with a single neuron. b represents the bias in each neuron, y represents the label at the output and f() represents the activation function.*

7.2.1 Activation Functions

Activation functions can be a key problem in the dissipation of weight gradients. In Equation 7.1, one of the partial derivatives is of the activation function with respect to it's input and this is used to multiply the rest of the partial derivatives and inputs from previous layers. Therefore, if an activation function results in a very small gradient, particularly if it is used in multiple layers, then this can result in vanishing gradients. Figure 7.3 shows an example of a commonly used activation function: the Sigmoid. This activation function has the benefit of being within a range of 0 to 1 which can enable it's use for a probabilistic output. It also has a steep gradient at it's centre which forces outputs to polarise quickly with large parameter updates. However, when this activation function becomes saturated, the function's gradient tends to 0. This means that using this function in internal layers can result in vanishing gradients.

As an alternative, the benchmark network has adopted a ReLU activation function that is shown in Figure 7.4. This function consists of no activation below it's threshold and a unit gradient above. This means that every time a neuron is activated, it has a unit gradient and therefore will not cause multiplicative decay when used in the chain rule. This allows it to be used as part of deep networks. The activation value of 0 below the threshold also makes this activation function very efficient as it results in sparse activations. Layers with ReLUs must be carefully initialised however, as if their weights accumulate to a value too low, the updates can become stuck in the flat edge of the ReLU. In order to combat this, a Leaky ReLU option has also been included as an option in the framework. These activations are the same as ReLU but include a very small positive gradient below the threshold to prevent the network updates freezing.

7.2.2 Batch Normalisation

A batch normalisation layer was also introduced as another measure to prevent vanishing gradients. When parameters are updated in each training cycle of

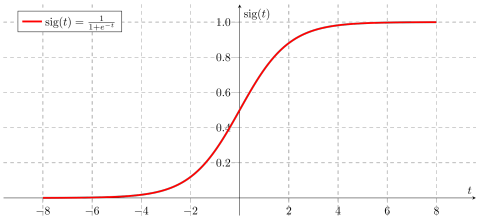


Figure 7.3: Sigmoid activation function.

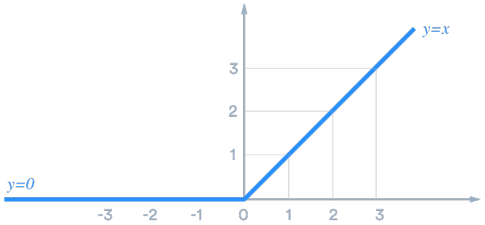


Figure 7.4: ReLU activation function.

the neural network, the optimiser makes updates to weights in all the layers of the network based on a single reading. However, changes to the the weights in the first layers will impact the outcome of subsequent layers. This causes the distribution in each of the following layers to change and can mean the optimiser is battling against itself to reach and optimum convergence. These shifts in distribution caused by parameter updates are often referred to as covariate shifts and can severely hinder the network's performance. Batch normalisation was first proposed in 2015 by S.Ioffe and C.Szegedy [14] as an approach for solving this issue. They proposed the concept of batch normalisation layers which be could be placed after a network layer as a way of normalising the output distribution to have 0 mean and unit variance. This prevents the distributions in the network from having large fluctuations and focuses the distribution around the threshold of the activation function. For example, Figure 7.5 shows how batch normalisation focuses the distribution of activation inputs from the previous layer to the threshold region of the Sigmoid. This focuses the input distribution to the area with the steepest slopes which will help to avoid vanishing gradients.

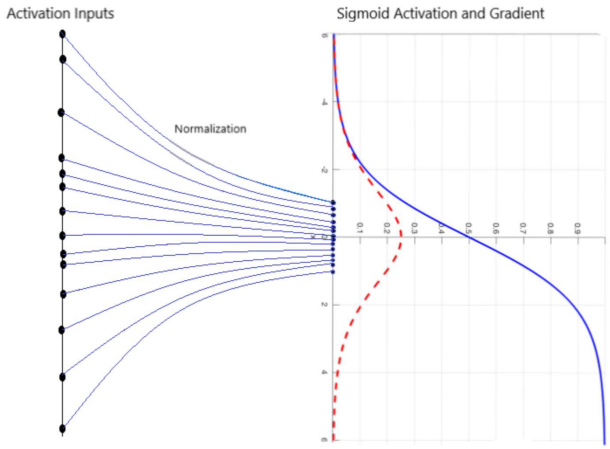


Figure 7.5: A diagram visualising the effect of batch normalisation. The blue line indicates a Sigmoid activation function and the red dashed line is it's corresponding gradient. [15]

7.3 Summary

This section has given an overview of the key design principles adopted while building the benchmark neural network. Convolutional layers were adopted to increase efficiency and help the network towards faster convergence. The rapid dimensionality reduction of these layers results in a small number of trainable weights and their kernels allow the network to learn an instantaneous relationship between between adjacent pixels. ReLU activation functions were also adopted to combat the problem of vanishing gradients. This function provides a constant unit gradient above the threshold while maintaining a crucial non-linear nature. Finally, batch normalisation was adopted to prevent large internal covariate shifts and offer further protection against vanishing gradients. This normalisation focuses the distribution around the critical area at the activation function's threshold using a mean of 0 and unit variance. These adaptations resulted in faster convergence and consistent performance as the line follower traversed it's environment.

8 Project Evaluation

This section will provide an evaluation against the project aims stated during the introduction. Direct mapping to each of the original project aims is discussed in Section 8.1. Project demonstrations, which were published online, are described in Section 8.2. Further work, which could enhance the project further, has been identified and is presented in Section 8.3.

8.1 Evaluation Against Project Aims

In Section 1.2, the core aims of the project were outlined and in Section 1.3 the requirements to complete these aims are presented. The aims which were defined are as follows: build a line-following robot, implement a ROS-based framework, add an intuitive GUI, implement a simulation environment and integrate the closed-loop neural network with the new framework. Each of the aims is discussed below and the extent to which they have been achieved is evaluated.

8.1.1 Physical Robot Implementation

The overall design philosophy for the building of the new robot was to make the robot consistent, modular and re-configurable. These properties lend themselves to consistent and flexible experimentation. This philosophy was accompanied by specific key requirements which included using new light sensors and an upgrade to using a Raspberry Pi 4.

To ensure that the robot would remain as re-configurable as possible it was decided to build the full robot from Meccano and using standard screws and bolts. This allows the dimensions of the robot to be altered or reconfigured and for additional modules to be fitted to the chaises while maintaining the reliability associated with the rugged Meccano material. These design decisions make the new robot far more configurable than the old robot which was glued into place meaning that there was very limited scope for re-configuration.

To ensure that the robot operated as reliably as possible in all configurations, a PCB was designed which incorporated infrared light sensors with a high dynamic range and IR LEDs that aim to provide consistent IR lighting for the light sensors.

The PCB should provide superior performance to the old robot as the sensors have a higher dynamic range with each sensor being paired with a separate IR LED to provide consistent infrared lighting. Initial testing was done on the line sensors and they were able to successfully view and interpret the line track. This contrasts the old robot that would use visible light to track the line which was an issue as it meant that lighting conditions were affected by the lights in the room. This meant that lighting was often not consistent between experiments. This is less of an issue when working with IR lighting, meaning that the new robot should behave more reliably than the old robot.

8.1.2 ROS Framework

Modularity is an important feature of the new robot. ROS has been used so that users can seamlessly transition between using the new GUI with a simulation or with the robot by subscribing and posting to ROS topics. The ROS setup will also allow the robot to interface with other applications built using ROS far more easily than would otherwise be the case. This also means that components of the framework can easily be integrated into other ROS-based systems. This adds flexibility to the old system where all software was custom with non-standard interfaces.

8.1.3 Cutomisable RQt GUI

The ROS framework also allows for utilisation of the RQt GUI framework. It works by subscribing to ROS topics and displaying sensor information, such as the camera feed and line-sensors. With the ROS-based framework, the addition and configuration of plug-ins can be done directly to monitor the environment with no interference to the operation. The GUI was tested in the simulation environments described below.

RQt was also used to configure neural network parameters. The configuration is input into the GUI plugin. When the neural network package is launched, it will request the configuration values from the GUI via ROS service request. This makes it very easy for the user to change neural network parameters quickly, increasing development efficiency.

8.1.4 Robot Simulations

Simulation environments were set up to test the ROS-based framework. Two environments were successfully tested: Gazebo which provides a realistic model of the robot but is computationally demanding and Enki which was integrated with ROS to provide a lightweight 2D simulation environment. Each of these plugged seamlessly into the framework and were incorporated successfully in the GUI components. The two environments were and were used to test different

aspects of the network configurations.

8.1.5 Neural Network Experimentation

Neural network evaluation was based on monitoring the closed-loop error values of the system. It was found that the introduction of convolutional layers, ReLU activations and batch normalisation significantly sped up the reduction of the system closed-loop error.

8.2 Project Demonstrations

To demonstrate the operation and performance of the framework which was created for this project, several demonstration videos were published online. Each of the demonstrations shows one of the simulation environments and a robot being controlled by either a neural network or a line-following algorithm, as stated. The GUI is also shown throughout the demonstrations to show the sensor and value responses in real-time throughout the simulation run.

The first demonstration shows the Gazebo robot model in a Gazebo simulation environment [16]. The robot is shown following the line without any learning. Additionally, the RQt GUI window displays the sensor values as it follows the track.

The second demonstration video shows the light-weight Enki simulation environment [17]. Again, this demonstration does not involve any learning and is using a line-following algorithm which responds to the sensor array readings.

In the third demonstration, the configuration options for the neural network setup are shown [18]. The example uses a closed-loop back-propagation neural network to control the robot.

The fourth and final demonstration shows the convolutional neural network, implemented as part of this project, controlling the robot [19]. At the beginning of the experiment, the robot can clearly be seen learning to follow the line after a few reflex actions.

8.3 Further Work

Further work has been identified to provide ideas for how this project could be taken forward. Three activities have been identified to improve on and test the system which has been developed.

8.3.1 Results Database

To aid the development and testing of different neural network configurations, it would be useful to have a database for results comparison. This addition to the system would allow for easy results comparison with storage of system metrics and simulation playbacks.

ROS already provides tools for data collection and management such as *rosbag* [20]. It supports a number of features such as recording data published on topics, and is also able to play back the recorded data. Adding this with a database framework would be a useful addition to the development framework.

8.3.2 Physical Robot Testing

Further work for the physical robot involves designing and testing the physical robot which was described in Section 4. This activity was halted by the unexpected closure of the university. Further work for this area has been described in detail in Section 4.5. These activities include finalising the construction, remote communication and ROS compatibility.

8.3.3 Further Network Development

Having now established a framework and benchmark network for testing, more improvements can be made and evaluated in the closed loop system. The highlighted additions in the benchmark network would no doubt benefit the closed loop network and these could be implemented and compared to the benchmark. To further improve the speed of convergence, filter could be intialised in the convolutional kernels. For example, a bank of Gabor filters could be used to detect lines from the outset. Another area for future work that was highlighted while testing the closed network was decaying activations. Ocassionally, the closed loop network would stop activating after a length of time and result in errors. It is speculated that this may be caused by underflow errors where the network weights have become multiplicatively low, resulting in values of 0. This was particularly relevant when normalised inputs less than 1 were used. Some incorporation of underflow protection may be an area that the network would benefit from.

Another useful addition to the framework would be to find an evaluation metric that takes into account the sharpness of the corners. When monitoring the error, there were large spikes where the robot turned a corner and the error tend to vary depending on the curve of the line being tracked. This makes it difficult to make an objective comparison between models and mostly relied on visual indication.

8.4 Evaluation Conclusion

The project has been successful in achieving its aims. A modular framework for future neural network experiments has been designed. With this framework supporting simulations, a GUI and a real Robot all communicating through ROS. Finally, the evaluation has uncovered several activities which can take the project forward.

9 Conclusion

Through this project a modular framework has been created which allows for plug-and-play neural network development. The plugin based RQt GUI front-end provides a personalised data visualisation and easy neural network configuration. The ROS framework allows for both the neural network and the robot components to be substituted and swapped out for alternatives. The system allows for both a physical robot and a simulated robot to be controlled.

Two simulator environments were setup and plugged into the framework. Gazebo providing a computationally heavy and realistic model and Enki providing high performance results, each of these simulators were useful for testing the neural network configurations. Each of the simulation models was integrated closely with the ROS framework allowing for RQt visualisations and neural network control.

The requested closed-loop neural network has been integrated as part of the ROS-based framework and appropriate controls and feedback have been included. In addition to integrating this network, a benchmark model has been built that adopts design principles intended on maximising it's performance in this closed-loop environment.

Therefore, this project has been successful in producing a plugin-based framework which supports the development of new AI algorithms. This framework has been tried and tested using simulation environments and several neural network configurations. The success of the project has been demonstrated through the development of a new AI algorithm.

Bibliography

[1] S. Daryanavard and B. Porr, "Closed-loop deep learning: Generating forward models with back-propagation", *arXiv preprint arXiv:2001.02970*, 2020.

[2] *AI Line Following Robot*, https://github.com/a2198699s/AI-Line-Follower, Last accessed: 2020-05-17, 2020.

[3] *ROS Wiki, Documentation*, http://wiki.ros.org/, Last accessed: 2020-05-15, Open Source Robotics Foundation, Nov. 2018.

[4] *ROS Wiki, rqt*, http://wiki.ros.org/rqt, Last accessed: 2020-05-15, Open Source Robotics Foundation, Aug. 2016.

[5] *SumoBot - Mini-Sumo Robotics, Assembly Documentation and Programming*, version 2.1, Parallax inc.

[6] *SFH 4056, High Power Infrared Emitter (850nm)*, version 1.6, OSRAM Opto Semiconductors, Aug. 2016.

[7] *VEMT2000X01, VEMT2020X01*, Vishay Semiconductors, Jan. 2019.

[8] *OpenCV, About*, https://opencv.org/about/, Last accessed: 2020-05-15, OpenCV, 2020.

[9] *GPIO*, https://www.raspberrypi.org/documentation/usage/gpio, Last accessed: 2020-04-30, Raspberry Pi Foundation, 2020.

[10] *Gazebo*, http://gazebosim.org/, Last accessed: 2020-05-15, Open Source Robotics Foundation, 2020.

[11] G. Robots, *Robotic simulation scenarios with Gazebo and ROS*, https://www.generationrobots.com/blog/en/robotic-simulation-scenarios-with-gazebo-and-ros/, Last accessed: 2020-04-24, 2020.

[12] *Enki*, https://github.com/enki-community/enki, Last accessed: 2020-05-15, GitHub, 2020.

[13] S. Daryanavard, *enkiSimulator*, https://github.com/Sama-Darya/enkiSimulator, Last accessed: 2020-04-24, 2020.

[14] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", *CoRR*, vol. abs/1502.03167, 2015. arXiv: 1502.03167. [Online]. Available: http://arxiv.org/abs/1502.03167.

[15] *Batch normalisation*, https://towardsdatascience.com/intuit-and-implement-batch-normalization-c0548033, Last accessed: 2019-03-20, Medium, 2019.

[16] *Line Follower Robot in Gazebo (without learning)*, https://www.youtube.com/watch?v=av-4ONKG6Zo, Last accessed: 2020-05-16, AI Line Following Robot, 2020.

[17] *Line Follower Robot in Enki Simulator (without learning)*, https://www.youtube.com/watch?v=98zSnN27VGs, Last accessed: 2020-05-16, AI Line Following Robot, 2020.

[18] *Line Follower Robot in Enki Simulator with a CLBP Neural Network*, https://www.youtube.com/watch?v=RX5ARqVHBWw, Last accessed: 2020-05-16, AI Line Following Robot, 2020.

[19] *Line Follower in Enki Simulator using a Convolutional Neural Network*, https://www.youtube.com/watch?v=pmnaS254G1A&t=68s, Last accessed: 2020-05-16, AI Line Following Robot, 2020.

[20] *ROS Wiki, rosbag*, http://wiki.ros.org/rosbag, Last accessed: 2020-05-15, Open Source Robotics Foundation, Jun. 2015.

Acronyms

ADC Analogue to Digital Converter
AI Artificial Intelligence
ANN Artificial Neural Network

GUI Graphical User Interface

HD High Definition

IR Infra-Red

PCB Printed Circuit Board

ROS Robot Operating System
RQt ROS Qt

SPI Serial Peripheral Interface
SSH Secure Shell

Individual Contributions

Cameron Bennett *2193356B*

- Design and implementation of the interface for neural network architectures.
- Design and implementation of the benchmark neural network (convolutional neural network).
- Integration with ROS and simulations: OpenCV (C++) image processing for network inputs, network configuration, network monitoring.
- Chapter 6: Neural Network Integration.
- Chapter 7: Neural Network Adaptations.

Mikkel Caschetto-Böttcher *2198221C*

- RQt GUI Plugins
- Enki ROS compatibility
- Line sensor PCB

Andrew Smith *2198699S*

- Gazebo simulation environment setup
- Modeling of physical robot, sensors and actuators in Gazebo simulation environment
- Worked on Raspberry Pi camera stream modification for the network input

Neil Wood *2197680W*

- Open CV video input.
- Enki simulator build and deployment.
- Physical robot parts and meccano.