UNIVERSITY OF CAPE TOWN

Department of Mechanical Engineering

RONDEBOSCH, CAPE TOWN
SOUTH AFRICA

MEC4128S: FINAL REPORT 060

Object Detection for the Duckiebot

Dino Claro

CLRDIN001

Project Brief

Autonomous vehicles (AV) have transitioned from science fiction to a focal point of extensive research endeavours. An indispensable aspect of any AV safety system is proficient object detection, enabling the vehicle to perceive its surroundings. Machine Learning (ML) techniques utilising Convolutional Neural Networks (CNN) have transformed object detection and are now standard in AV systems. However, implementing these object detection models can be challenging due to real-world complexities. Duckiebots are affordable mobile robots with the necessary hardware, providing a simplified environment for investigating object detection models. The Duckietown environment, where duckiebots operate, offers developer tools and a simulation platform. This project aims to explore the Duckietown platform for ML applications by implementing an object detection model. ML techniques will be employed to train an object detection model capable of identifying relevant objects in the Duckietown environment, such as rubber ducks and duckiebots, both in simulation and on the duckiebot.

Supervisor: Dr Arnold Pretorius

Word Count: 7890

Declaration

---

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the `IEEE` convention for citation and referencing. Each significant contribution to, and quotation in, this report/project from the work(s) of other people has been attributed, and has been cited and referenced.
3. This report/project is my own work.
4. I have not allowed, and will not allow anyone to copy my work with the intention of passing it off as his or her own work.

Signature:

`DINO FILIPE LOURENCO CLARO`

Problem Statement

Duckietown is an up-and-coming future city powered by artificial intelligence. This rapid development means that there are many duckies moving around the city. The duckies are developing a self-driving car, and to ensure that the Duckiebot can be safely tested in Duckietown, a Duckie Detector needs to be implemented.

Abstract

Autonomous Vehicle (AV) development is a complicated endeavour, now pursued in both academic and commercial contexts. A simplified AV environment, like Duckietown, alleviates some of the complexities and costs of real-world scenarios, facilitating faster development. Despite the reduced complexity, Duckietown provides a non-trivial learning curve, particularly at the undergraduate level. This report investigates the platform's viability for establishing an AV pipeline in the context of the MEC4128S project structure. The crucial first step in AV development is *environment perception* through object detection. You Only Look Once version 5 (YOLOv5) is implemented on the duckiebot for object detection and as the groundwork for future development. The YOLOv5 model is trained using real and synthetic data relevant to the duckiebot. The inclusion of synthetic data proves highly effective, achieving an mAP-0.5:0.95 of 0.735, higher than that of larger YOLOv5 models trained solely on real-world data. In Duckietown, duckies and other duckiebots present potential hazards to the vehicle. To validate the YOLOv5 model and achieve obstacle avoidance, a simple Braitenberg controller is implemented. The model reliably detects duckies in most Duckietown scenarios. Finally, an extensive viability analysis is presented, proposing three promising directions for future work on the platform: extending the Braitenberg controller, developing an End-to-End Reinforcement Learning agent, and implementing a Depth Estimation model.

Acknowledgements

Mistakes are the portals of discovery.

- James Joyce

For Avô Nito my most formidable supporter,

This project, marking the conclusion of my undergraduate degree, is a culmination of the privilege and support I have received throughout my degree. Firstly, to Tio Luis and the Marketos and Tsironis families, thank you. Your generosity has allowed for this journey.

Thanks to Kyanna, a valiant effort as the head of my nourishment team, my mom for her sacrifices and Sam who endured many fascinating lectures presented by me on the ins and outs of this project.

The degree and, more immediately, this report would not have been a reality without the rallying support of my collegues. In particular, a thanks to Nathan for always being a constant source of assistance and the best validation scenario namer out there.

To Dr Arnold Pretorius, thank you for granting me complete freedom to explore a field with pure interest and minimal prior knowledge. In stumbling along the way, I have learned a great deal in computer vision, machine learning and robotics. Perhaps to a greater value, I have learned the importance of conducting research before generating project briefs. In this, I appreciate Arnold's guidance and resistance to the urge not to burst into laughter when my initial project brief was an ambitious Masters brief.

Finally, an apology to loved ones and friends for the mania that ensued in the final weeks of this project. Thank you for bearing with me, I will reply to messages shortly.

Table of Contents

List of Tables

List of Figures

Abbreviations

AV	Autonomous Vehicle
AVA	Autonomous Vehicle Architecture
CNN	Convolutional Neural Network
FC	Fully Connected (Layer)
FSD	Full-Self-Driving
MLP	Multi-Layer Perceptron
mAP	Mean Average Precision
ML	Machine Learning
NMS	Non-Maximum Suppression
NN	Neural Network
ROS	Robot Operating System
SVM	Support Vector Machine

CHAPTER 1

SETTING SIGHTS

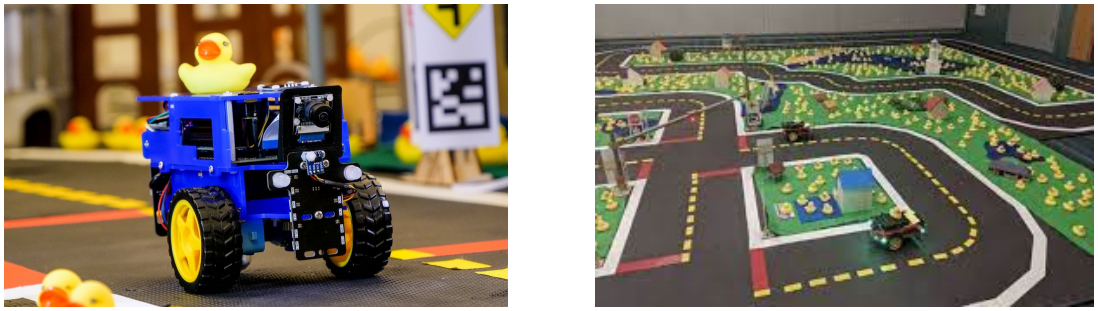
An Introduction

> The world is full of magic things, patiently
> waiting for our senses to grow sharper.
>
> W.B. Yeats

1.1 Motivation

The push for realising Full-Self-Driving (FSD) vehicles is driven by the promise of minimising accidents, reducing carbon emissions and freeing humans from laborious driving situations [1]. Advancements in computer vision through Machine Learning (ML) techniques have abetted the emergence of vision or camera-based Autonomous Vehicles (AVs). These AVs are promising FSD solutions as they no longer require extensive sensor suites. However, current solutions are only level 2 or 3 according to the SAE J3106 [2] taxonomy. This taxonomy ranges from fully driven level 0 vehicles to FSD level 5 vehicles.

Despite the level 2 or 3 classifications, these vehicles are complicated endeavours where development is no longer an academic exercise but a commercial one. Development of an FSD vehicle is a daunting task with many components to consider. Simplified AV environments, such as Duckietown, aim to reduce the complexity and cost associated with AV development. Duckietown, illustrated in Figure 1, consists of differentially steered robots, known as duckiebots, that navigate Duckietown which resembles real-world roads.



(a) Duckiebot (b) Duckietown

Figure 1.1: The Duckietown platform. Image source.

Duckietown provides standardised rules and object sizes, alleviating some of the challenges associated with AV development. However, constructing an AV pipeline, even within the context of Duckietown, remains a non-trivial task, particularly at the undergraduate level. The platform is geared toward ML applications and leverages the Robot Operating System (ROS) middleware and Docker for containerization, which are all unfamiliar to undergraduate students. As there is a desire to develop an AV pipeline for the Duckietown platform at UCT, it is necessary to investigate the viability of duckietown platform within the framework of the MEC4128S project structure. The first step for an AV is to sense its environment and identify relevant objects, This is known as object detection and is identified as a foundational component of the pipeline in this report. Convolutional Neural Networks (CNNs), a ML subset, are currently the most popular choice for object detection and serve as an excellent starting point for exploring the Duckietown platform for its ML suitability.

1.2 Aim & Objectives

This report aims to investigate the viability of the Duckietown platform by implementing a ML object detection model on the duckiebot platform. Various objectives have been identified to realise the aim:

(a) General survey of literature to identify a suitable CNN-based object detection model.

(b) Generate a Duckietown-relevant dataset to train the model.

(c) Train the object detection model on the generated dataset to detect Duckietown-relevant objects.

(d) Implement a Duckietown Docker-compliant ROS image to enable the use of the object detection model on the duckiebot.

(e) Extend the Docker image constructed in objective (d) by implementing a control scheme to allow the duckiebot to avoid duckies.

(f) Conduct a viability analysis of possible future work at the undergraduate level using the implemented object detection algorithm.

1.3 Scope & Limitations

This report is confined to the consideration of ML object detection models, with a specific focus on CNN-based models. Similarly, the future work proposed for the development of the AV pipeline falls under the ML umbrella and is tailored to the MEC4128S course. This direction is motivated by the fact that the Duckietown platform is designed to serve as an educational tool for ML.

There are some notable limitations to this report. The constraints of time and computational resources, particularly the absence of a laptop equipped with a GPU, made it unfeasible to implement a CNN model from scratch. As a result, a pretrained CNN model is customized with the generated Duckietown-relevant dataset, rather than training a model with initial random weights. Secondly, any control implemented on the duckiebot is primarily for the validation of the object detection model and does not constitute a robust controller for the duckiebot.

1.4 Report Outline

The report and the code found at the repository location in Appendix A are structured to serve as a stand-alone reference for future development on the duckiebot at the undergraduate level. In this endeavor, a substantial background on CNNs is provided in Chapter 2, while an overview of Neural Networks (NNs) can be found in Appendix B. The inclusion of this background context is based on the assumption that a student in the Mechanical Engineering department is likely not to be familiar with these fields.

Chapter 3 reviews the literature on object detection models and provides a brief overview of Braitenberg controllers used for control in this report. Chapter 4 delves into the YOLOv5 object detection model, presenting and discussing training results. Chapter 5 outlines the process of implementing the YOLOv5 model and the Braitenberg controller on the duckiebot, along with validation results for the implementation.

Chapter 6 summarises findings from a viability analysis that was conducted to identify the immediate future work on the duckiebot platform. Future work is presented as

three projects that emerged from the analysis and the work completed in this report. The complete viability analysis is provided in Appendix E for interested readers. The remaining appendices include miscellaneous information that arose during the course of this report

CHAPTER 2

FROM NEURONS TO PIXELS

A gentle introduction to Convolution Neural Networks

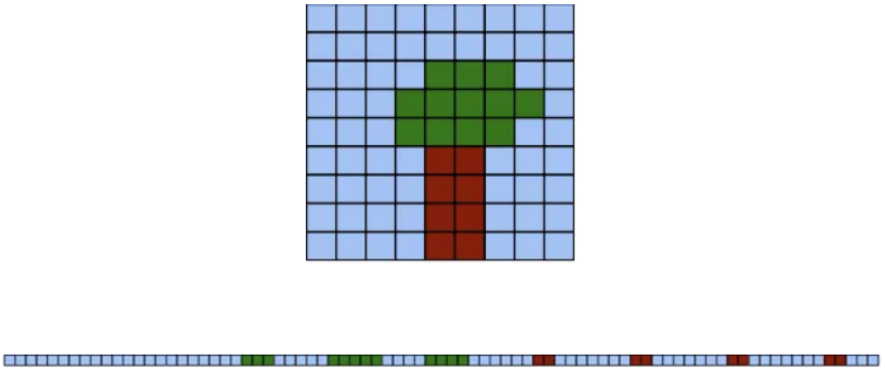
Great ideas need landing gear as well as wings

C.D. Jackson

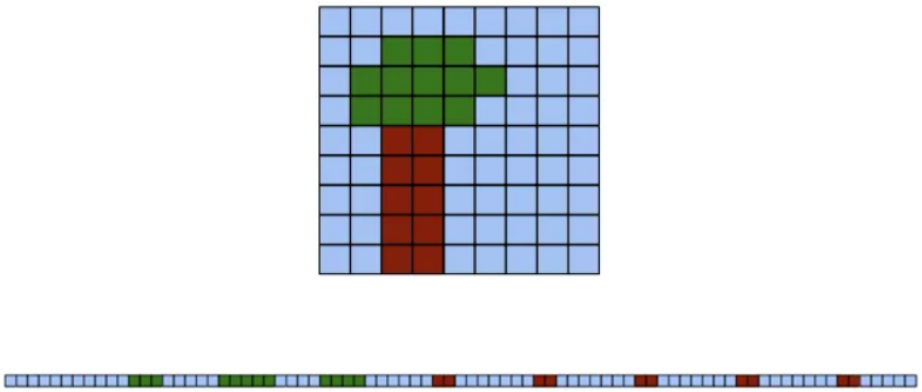
In the world of artificial intelligence, *Convolutional Neural Networks* are the architects of perception enabling machines to see. This chapter provides a brief introduction and motivation of the use of CNNs to the field of computer vision in general. Computer vision is generally considered to be computer science field outside the scope of the Mechanical Engineering curriculum motivating this explanatory chapter. For readers unfamiliar with NNs, Multi-Layer Perceptron (MLP) network, and the backpropagation algorithm refer to Appendix B.

2.1 Convolution Neural Networks

Traditional NNs, such as the MLP, are constrained to accept column vector inputs by design. The network uses these vectors to make predictions based on a set of learned parameters optimised for a particular ground truth dataset through backpropagation. In other words, the network recognises patterns from training in the new data and makes a prediction based on the particular combination of patterns recognised. Consider using an MLP network to detect a tree shown in Figure 2.1. If the RGB image is of size (m,n,3), the input to the MLP would be a flattened vector of the image pixels with a size of m*n*3.



(a) Labels



(b) Predictions

Figure 2.1: Two 9×9 images demonstrating the *curse of dimensionality* and unpraticality of using an MLP for image classification. Notice that the tree's shape is identical in both images and only translated. However, the 1D concatenation vectors show no correlation. An MLP would have to learn both patterns to classify the tree correctly. Image source.

Taking note of the flattened vectors in Figure 2.1, the MLP would have to train for each pixel combination to classify the tree correctly. This is known as the *curse of dimensionality* [3] and occurs when higher dimension information attempts to be extracted from lower orders. If the hidden layer [1] contains k neurons, then the number of parameters increases to (m*n*3)*k, which becomes unpractically large to train or store these parameters. Additionally, images are translation invariant, where features are the same regardless of their position in an image. The locality of an image is another important consideration, where pixels nearby are more strongly correlated than other pixels. These correlations are called features, and combining features results in objects. For example, horizontal lines and vertical lines together represent a square.

[1] A hidden layer in a NN is any layer between the input and output layers. See Appendix B for further information

The CNN, inspired by the organisation of animal visual cortex [4], provides a solution for object detection by considering the 2D structure of images and adaptively learning spatial hierarchies of images [5]. In other words, CNNs consider that low-level features make up high-level features. CCNs are a neural network that uses the convolution operation in place of the general matrix multiplication [6]. A traditional CNN typically consists of three layers: convolution, pooling [2] and head or fully-connected (FC) layers [4] shown in Figure 2.2.

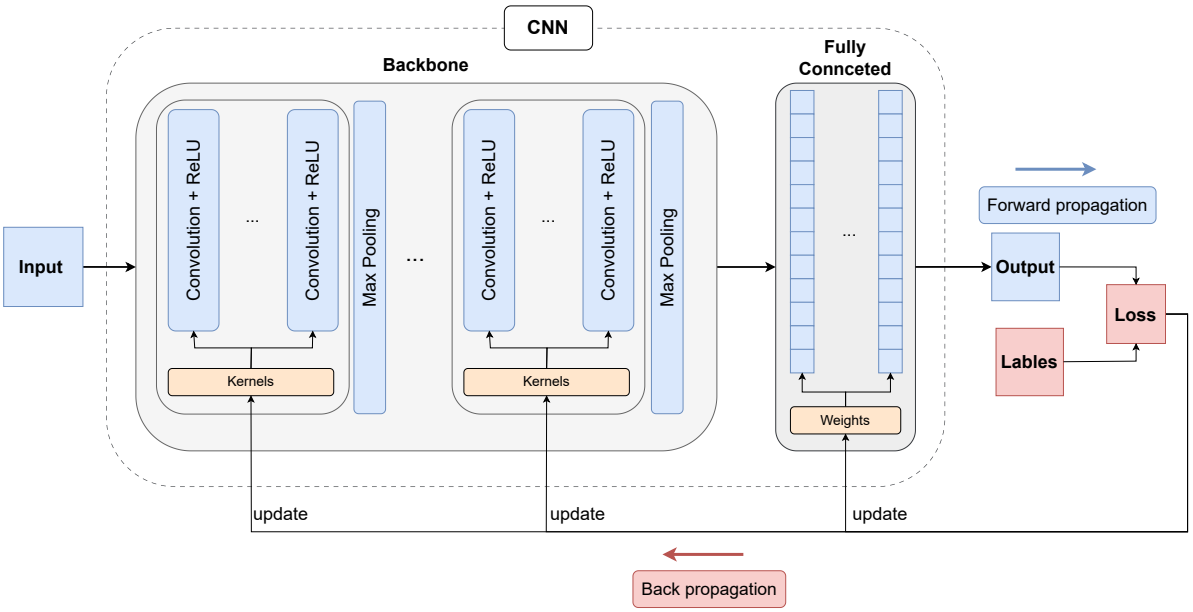


Figure 2.2: Overview of the CNN architecture and training process. A CNN consists of sequential building blocks: convolution, pooling, and FC layers. During training, convolution kernels and FC weights are updated during back propagation with a suitable optimisation algorithm [5].

The convolution layer applies the 2D discrete convolution operation shown by Eq. 2.2 where I and S are the input and output feature maps, respectively, K is the convolution kernel, and (m, n) are the image dimensions. A *feature map* results from performing the convolution operation to an image, and a *kernel* is a small, learnable matrix applied to an image. The convolution operation re-estimates the image at every pixel, assuming a stride of one, as the weighted average of the pixels around it. It can be visualised as a small window (kernel) sliding over an image shown in Figure 2.3.

[2] This report refers to the backbone as the grouping of convolution and pooling layers.

$$S(i,j) = (K * I)(i,j) \tag{2.1}$$

$$S(i,j) = \sum_{i=1}^{m}\sum_{j=1}^{n} I(i-m, j-n)K(m,n) \tag{2.2}$$

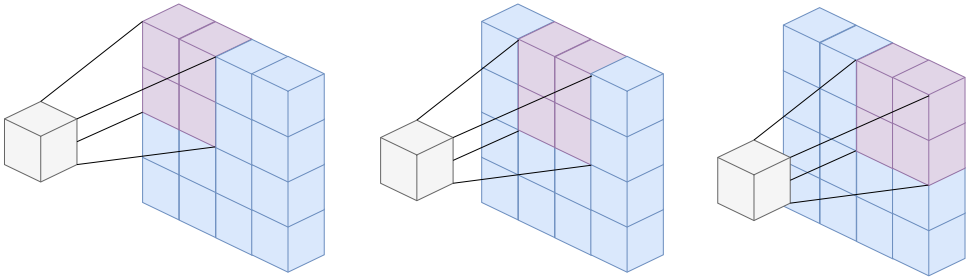


Figure 2.3: The sparse interaction convolution operation for a 2D discretised grid. The kernel slides over the image, and at each position, the convolution is computed by multiplying the overlapping pixels and kernel values and summing these products. The result is a tensor with element values resulting from the weighted average of surrounding pixels.

In computer vision applications, the convolution is used as a 'filter' that enhances certain features. Consider two Sobel convolutions kernels shown by Eq. 2.4a; applying each of these kernels to an image extracts the horizontal and vertical edge gradients. Creating a new image, with the magnitude of the edge gradients, enhances the edges in an image, as shown in Figure 2.4b.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

(a) Sobel kernels

(b) Sobel edge enhancement

Figure 2.4: Sobel kernels and example of using the kernels for edge enhancement. (a) shows the kernels G_x and G_y for extracting the horizontal and vertical edges, respectively. (b) is a comparison of the original image and the result of Sobel edge detection. The Sobel operator uses horizontal and vertical kernels to extract the gradient of edges in an image. Image source

CNNs use three architectural features that ensure a degree of transformation invariance: sparse interactions, parameter sharing and equivariant representations [6]. *Sparse interactions* are achieved using a kernel smaller than the image. The motivation is from the locality characteristic of images where meaningful features can be extracted from a small

set of nearby pixels. As the kernel size, k, decreases, the number of operations decreases according to $O(w^2k^2)$ where w is the image's width. *Parameter sharing* refers to applying the same kernel at every image position with the motivation that a feature extracted by the kernel can be present at any location. Reducing memory requirements as each layer only has the kernel parameters, k^n, as a pose to $m * n$ [3]. Parameter sharing naturally results in *equivariant representation* or translation invariance. Suppose an object moves in the input map. In that case, its representation will move by the same amount in the output map [4] [6].

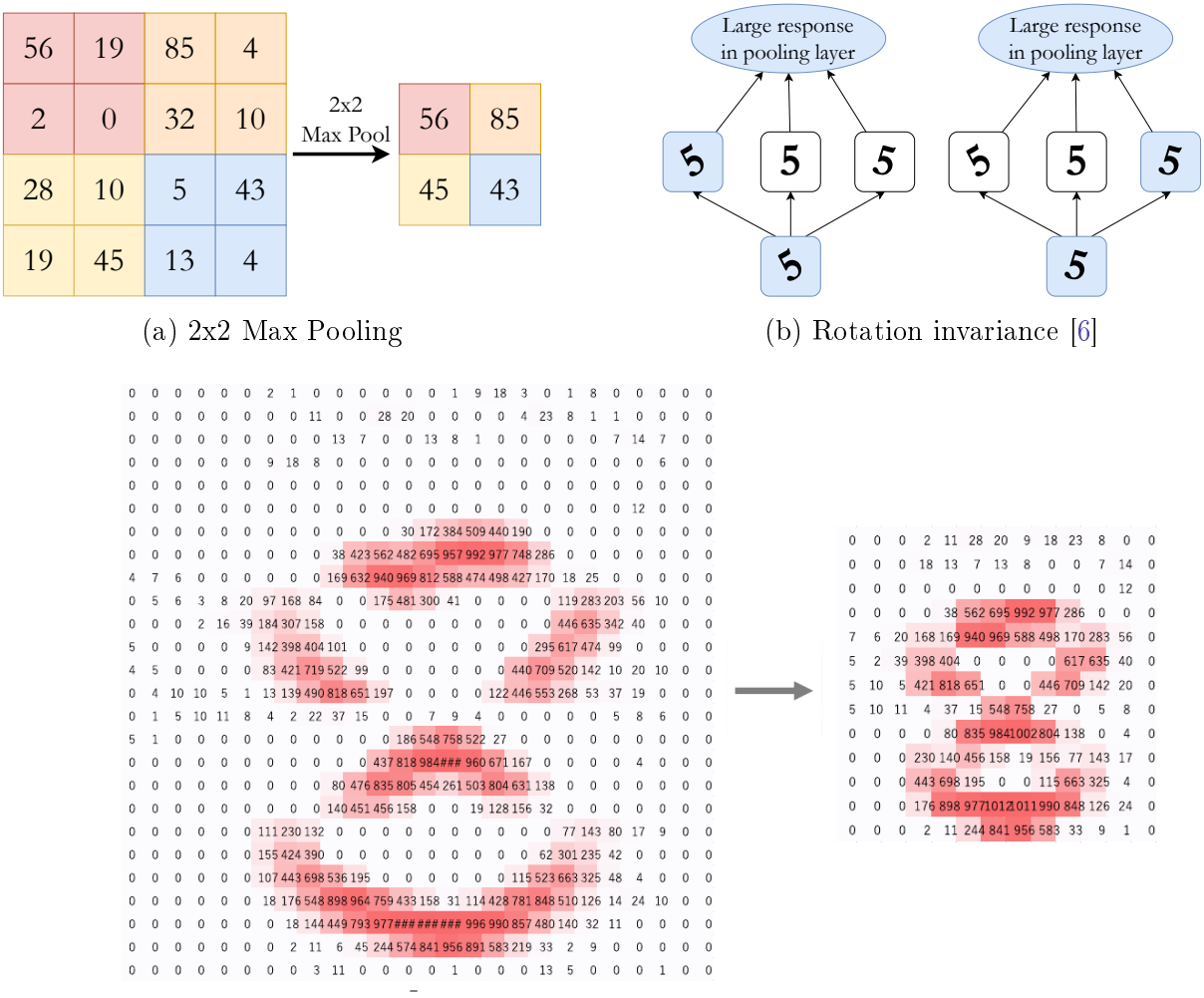
The convolution is a linear operation; hence, the motivation for applying an activation function afterwards is to introduce non-linearity into the model. Non-linearity is crucial as it allows the model to learn complex patterns in data [5]. The pooling layer is a downsampling operation that replaces the feature map with a summary statistic of nearby pixels in the feature map. Pooling is performed for two reasons: dimensionality reduction and invariance to rotations. Eq. 2.3 shows how dimensions are reduced where m represents the dimension of the feature map, assuming a square map, f is the pooling window size, and s is the stride between each pooling operation. Max pooling is the most common pooling operation and returns the maximum value of a window of pixels shown in Figure 2.5 (a) and (c). This operation is used to reduce the dimensionality of the feature map. By placing the pooling layer after the activation layer, the pooling extracts the most essential features shown in Figure 2.5 (c). Additionally, it provides rotation invariance, see Figure 2.5 (b), and noise suppression.

$$m_{out} = \frac{m_{in} - f}{s} + 1 \tag{2.3}$$

[3] k is usually several orders of magnitude smaller than the image dimension.

[4] The convolution is not naturally equivariant to scale and rotation, which are handled by the pooling layer and other mechanisms

[5] Appendix B provides a motivation for non-linear activation functions and explains common activation functions



(a) 2x2 Max Pooling

(b) Rotation invariance [6]

(c) Max Pooling example [5]

Figure 2.5: Max pooling operation. (a) Max pooling with a window size of 2x2 and a stride of 2 extracts the maximum value in each window. (b) shows how pooling is invariant to small transformations. The network has learnt to detect a 5 using three three convolution kernels. Notice that max pooling has a large activation regardless of which kernel is activated [6]. (c) Max pooling example showing the extraction of the most pertinent features [5].

Considering the CNN in Figure 2.2 again, multiple convolutions, activation and pooling layers are stacked together in the backbone to extract features hierarchically. The general idea is that layers close to the input learn low-level features (e.g. lines), and layers deeper in the model learn high-order or more abstract features like a duckie in Duckietown [5]. However, CNNs are still considered black boxes in many cases where a branch of research called feature visualisation attempts to decode feature maps. Figure 2.6 shows the famous adversarial example by Goodfellow et al. of adding a small amount of carefully constructed noise resulting in misclassification [7]. The research on the vulnerability of these networks is crucial for safety-critical fields like AV applications.

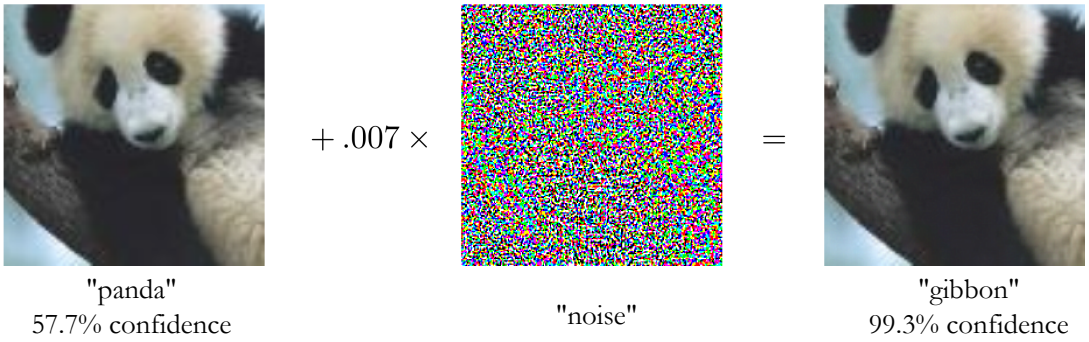


Figure 2.6: Adversarial example by Goodfellow et al. where a small amount of targeted noise results in a misclassification; however, there is no visible change to the image [7]. Suggesting that CNNs learn patterns in data rather than just visual clues.

The final layer is called the head and classifies the high-level features. The CNN in Figure 2.2 uses a fully connected (MLP) network to learn non-linear combinations between the feature map and the final outputs, which are typically probabilities [6]. With the three common layers in the CNN architecture, Table 2.1 summarises the general parameters and hyperparameters for setting up a CNN. The differentiation is that a parameter is a variable optimised during training, and hyperparameters are user-defined.

Table 2.1: Parameters and hyperparameters in a convolution neural network [5]

	Parameters	**Hyperparameters**
Convolution layer	Kernels	Number of layers, kernel size, stride, padding, activation function
Pooling layer	None	Window size, stride, padding
Fully connected layer	Weights	activation functions
Others		model architecture, optimiser, learning rate, loss function, batch size, epochs, dataset split

The structure of a CNN lends itself to *transfer training* where a common backbone is trained on a large dataset to learn feature representation. The process is explained in Figure 2.7 and uses the principle that features learnt on a large enough dataset apply to smaller datasets. Transfer learning is beneficial for specialised applications where it is challenging to generate datasets or compute is limited. Yamashita et al. provide an excellent description of the benefit of transfer training for object detection in the field of radiology [5]. Fixed feature extraction, as per Figure 2.7, is used in this report to use the relatively small generated dataset to customise a pretrained object detection model.

[6] Appendix B provides a further explanation of final layer activation functions for the interested reader

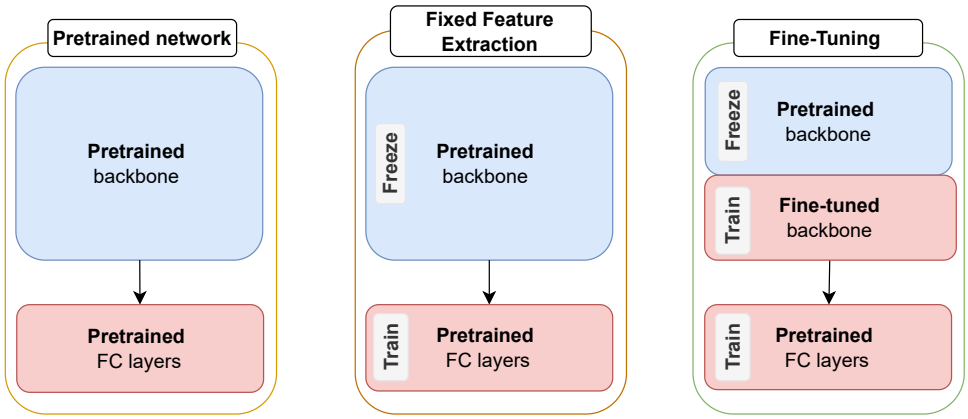


Figure 2.7: *Transfer training* is a strategy to train a network on a small dataset using a CNN that has been trained on a large dataset. Fixed feature extraction is the process of customising the fully connected (FC) layers from the pretrained network while maintaining the backbone that has learnt good feature representation. Alternatively, the fine-tuning method replaces FC layers and a portion of backbone layers. This method can produce better results but requires longer training periods [5]

CHAPTER 3

OBJECT DETECTION AND EMOTIONAL ROBOTS

A Literature Review

> If you wish to make an apple pie from scratch,
> you must first invent the universe
>
> Carl Sagon

A fundamental requirement of an AV is the perception of its environment. Object detection is recognised as a core process in this pursuit of environment perception. The chapter surveys camera or vision-based object detection models relevant to AV applications. The goal is to select an appropriate model for implementation on the Duckiebot. As mentioned in Section 1.3, control of the duckiebot is not the report's primary focus but is used to validate the object detection model. A Braitenberg reaction controller is used in this report, which offers a straightforward means of control while still maintaining real-world relevance. The chapter briefly introduces these controllers and their applications for the reader's understanding.

3.1 Object Detection

Object Detection is a branch of computer vision that is concerned with accurately *locating* and *classifying* all instances of objects in an image [8]. Conventional vision-based techniques like Scale-Invariant Feature Transform (SIFT) [9] and Histogram of Orientated Gradients (HOG) [10] extract user-defined features representative of an object. Edges and corners are the most basic features, and a particular arrangement of these features forms a more complex feature. Each feature needs to be defined and classified by the user. During deployment, the model searches for these predefined features to detect objects. These techniques are well-established and optimised for performance and power.

Walsh et al. provide an overview of simple or data-stricken applications where traditional techniques are still preferred [11]. However, they are limited by the complexity of the environment where defining features naturally reduces the algorithm's robustness and increases the number of defined features to an unfeasible amount [12].

Time-of-flight (TOF) sensors were employed for object detection in dynamic environments, such as AV applications, to overcome the limitations of traditional vision-based object detection. Figure 3.1 shows the system architecture used by the winning team of the 2012 Hyundai AV competition [13]. Notably, the system utilises both cameras and LiDAR sensors. In the architecture proposed, the LiDAR sensors detect moving objects [14] and barriers [15], and cameras extract lane markings [16].

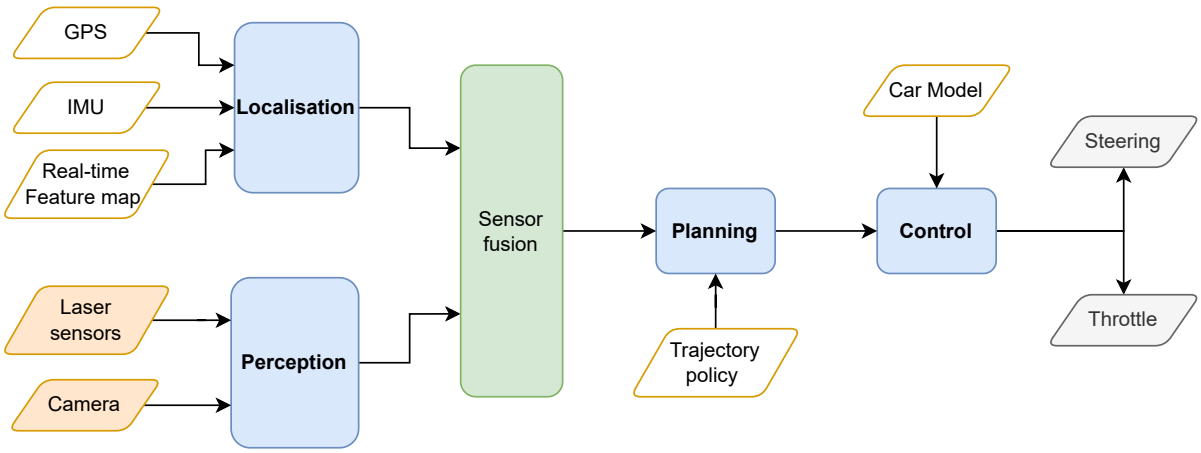


Figure 3.1: 2012 Hyundai AV competition winning software architecture implemented by Jo et al.[13]. The architecture uses a host of sensors with camera and LiDAR sensors used for robotic perception.

While combining TOF sensors and cameras proved effective in early AV development, it failed to produce a marketable AV solution due to the added cost and complexity of using multiple hardware components [17]. The advancements of CNNs in computer vision have allowed a shift to pure vision-based object detection. To the extent that CNNs are the preferred approach for AV object detection, offering cost-effectiveness and greater reliability and robustness [18].

Region-proposal Convolution Neural Network (R-CNN) [19], the first CNN-based object detection algorithm, introduced by Girshick et al. in 2014 uses the conventional *Selective Search* algorithm [1] to propose 2000 regions of interest (RoI). Each RoI is fed into a CNN sequentially to extract a feature vector that is classified with a Support Vector Machine

[1]Selective search generates a diverse set of potential object regions by grouping image segments based on colour, texture, and other cues for subsequent object detection

(SVM) [2]. This is a two-stage process of *RoI proposal* and *classification*. R-CNN showed a 30% mean Average Precision (mAP) increase on the PASCAL VOC benchmark dataset over the traditional techniques but demands extremely high computational and memory costs due to passing each ROI to the CNN sequentially. mAP is a common baseline metric summarising the performance of object detection algorithms and is derived in Section 4.3. Girshick published Fast R-CNN a year later, in 2015, that improved inference speed by using a single CNN to extract a feature vector map for all RoIs in a single pass [20]. Shaoqing et al. added to the R-CNN family with Faster R-CNN in 2016, replacing the selective search algorithm with a Region Proposal Network (RPN) for RoI proposal [21]. The RPN is placed after the feature map generation shown in Figure 3.2. Overall, these improvements reduced the inference time of Faster R-CNN to R-CNN from 40 to 0.2 seconds [21]. Nonetheless, the two-stage algorithm of the R-CNN family, which relies on the sequential RoI proposal and classification, is not optimal for real-time applications [12].

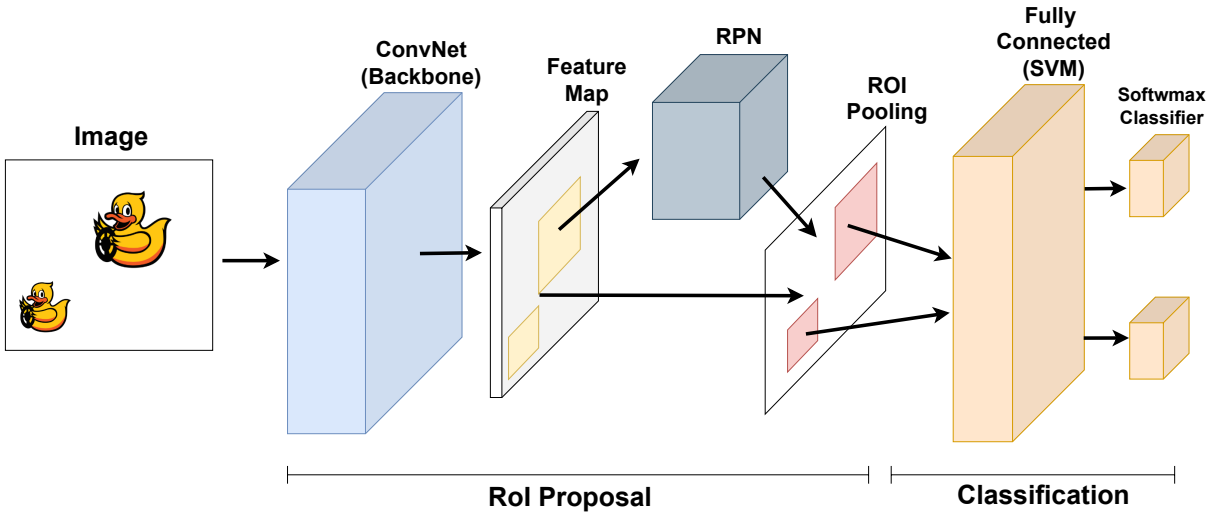


Figure 3.2: Faster R-CNN uses a unified CNN network for region proposal. The RPN is a CNN that takes an image as an input and outputs RoIs, each with an objectness score [21]. The RoI pooling layer uses the feature map and RoI outputs to extract prominent RoIs, which are then fed into a fully connected SVM. The network outputs two vectors per RoI: softmax classification probabilities and per-class bounding box regression.

Single-stage algorithms such as RetinaNet [22], Single Shot Detector (SSD) [23], and You Only Look Once (YOLO) [24] improve inference time by using a single pass through a CNN. Generally, two-stage algorithms have better detection performance while the single-stage algorithms are faster [25]. Of these algorithms, YOLO adopts a relatively simple and efficient architecture explained in Figure 3.3, which has been used extensively for

[2]SVM is a supervised ML algorithm used for classification and regression tasks and is particularly well-suited for binary classification.

real-time applications [26, 27, 12, 28].

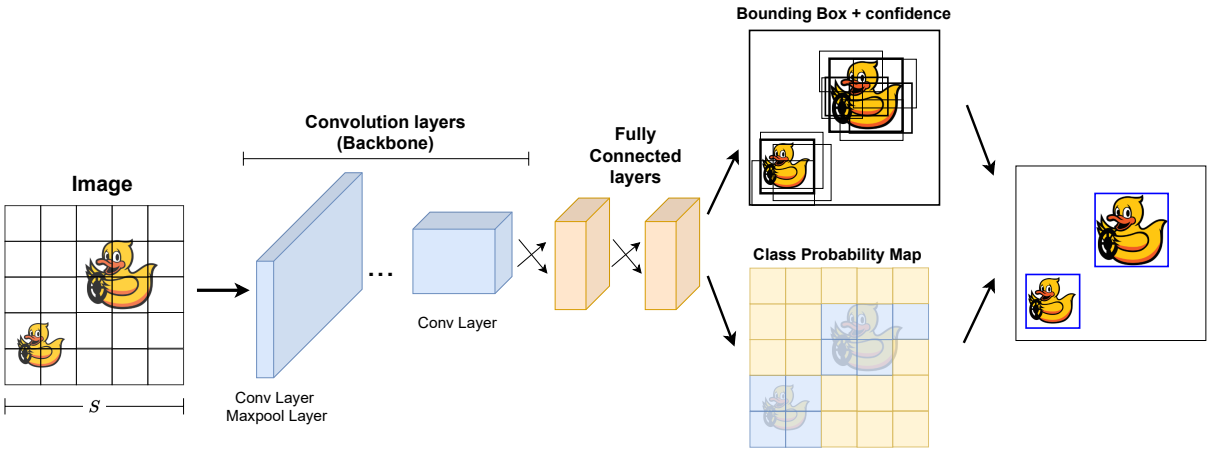


Figure 3.3: YOLO [24] architecture consists of multiple convolutional (24 in total) and pooling layers, followed by two fully connected layers. The image is divided into an $S \times S$ grid, with each grid element responsible for predicting bounding boxes and class probabilities for objects whose centres are located within that specific grid cell [24]. This grid-based structure enhances object localisation and computational efficiency. The final layer produces predictions for class probabilities and bounding box coordinates. These predictions are refined using the Non-Maximum Suppression (NMS) algorithm, which helps eliminate redundant and overlapping detections since multiple grids can predict the same object.

YOLO frames object detection as a regression problem. *Regression* is an ML technique to predict continuous values. YOLO uses regression to predict the bounding box centre location and dimensions, class probabilities and objectness scores [3] as continuous values. According to the authors of YOLO, Redmon et al., unlike the R-CNN family, YOLO considers the entire image encoding implicit information lost by R-CNN methods during region proposal, resulting in YOLO making half the number of background errors. R-CNN methods still produce better mAP values but are at least 2.5 times slower than YOLO [24].

The YOLO architecture shown in Figure 3.3 struggles to generalise to classes in new or unusual aspect ratios since only course features are learnt through the downsampling layers. However, the YOLO algorithm is regularly updated with the latest version at YOLOv8, addressing the limitations and implementing numerous new features. This report makes use of YOLOv5 as it provides scaled-down models optimised for computing efficiency [29] and is compatible with the `Python 3.6`, necessary for the duckiebot [30]. A detailed explanation of the YOLOv5 architecture is provide later in Section 4.1

[3] Objectness is a measure of the probability that a given bounding box contains an object of interest.

YOLOv5 has been used for a wide range of applications. Xu et al. use YOLOv5 for forest fire detection [31], Yadav et al. use images from a drone and YOLOv5 to detect cotton plants at different growth stages [32], Qiu et al. show YOLOv5's real-time viability by detecting foreign objects using a ground radar [28]. In AV applications, object detection models are part of larger state estimation controllers [33]. Tesla's FSD development architecture uses a YOLO-like for object detection. Zhang et al. use YOLOv5 with a binocular fisheye camera for real-time pose estimation of an aerial robot [34].

On the duckiebot, Chang et al. extend YOLOv5 for imitation learning [4] to control the duckiebot [12]. Yuhas and Easwaran implement an Out-of-Distribution detection model that extends a YOLOv7 model to predict Duckietown objects reliably in unseen environments conditions like snowfall [27]. Nguyen et al. implement different optimisers and YOLOv5 architectures with a real-world Duckietown dataset and is used as a comparison for the object detection model trained in this report [30].

This report differs from Nguyen et al. by supplementing the real-world dataset with synthetic, simulated images from the Duckietown-gym simulation environment [35]. Using synthetic data is a common practice in computer vision as it allows for generating large datasets with minimal effort. However, real-world generalisation is not guaranteed [36]. Kalapos et al. showed that a model trained in the Duckietown-gym simulator generalises well to the duckiebot when data augmentation techniques like *domain randomisation* are employed [37]. The report takes inspiration from Kalapos et al. and uses domain randomisation to generate a synthetic dataset. The implementation of this is explained later in Section 4.2

3.2 Robots that can feel

A connection directly from a sensor to an actuator is known as a sensorimotor connection. Sensorimotor controllers, consisting purely of sensorimotor connections, are the most basic form of reaction-based controllers. Consider Figure 3.4 showing E. coli moving through a protein concentration by means of chemotaxis [5] [38]. The E. coli can be abstracted as a single sensorimotor connection.

[4] In imitation learning, instead of the reward function, an expert, usually a human, provides the agent with a set of demonstrations.

[5] Chemotaxis is the movement in a direction corresponding to an increase in the gradient of a chemical

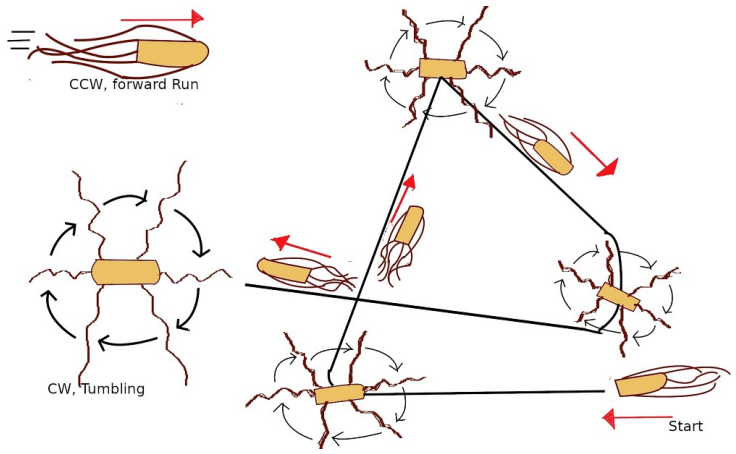
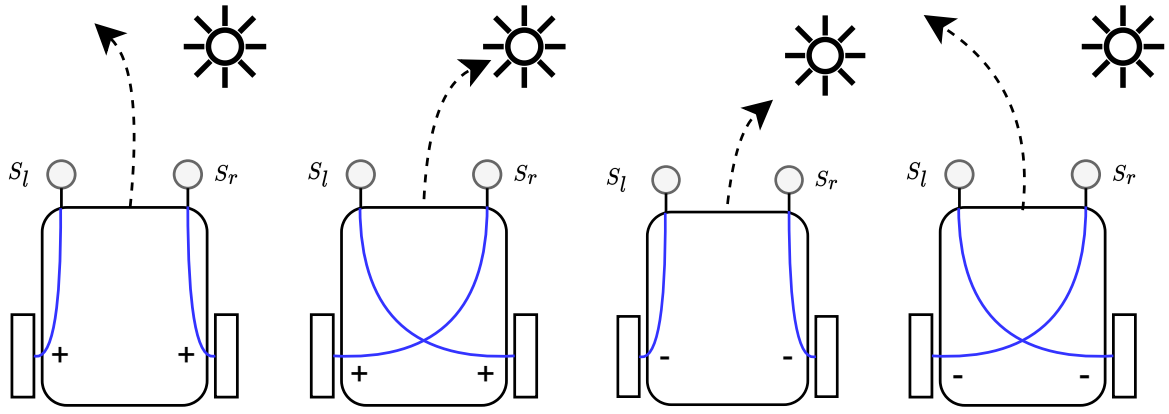


Figure 3.4: E. coli swim in relatively straight lines in areas of high protein concentration when their flagellum, or 'tails', rotate in a counter-clockwise direction and tumble in a random direction when rotating in a clockwise direction [38]. E. coli receptors are directly linked to the flagellum, and in times of high concentration, the E. coli swims straight. In contrast, in low concentrations, the E. coli switch between tumbling and straight motions until reaching an area of high concentration. Image source

Valentino Braitenberg theorised sensorimotor connections for the differential-drive robot consisting of two sensors on either side of the vehicle [39]. This results in four possible configurations that all exhibit different types of motion described in Figure 3.5.



(a) Fear (b) Hate (c) Love (d) Curiosity

Figure 3.5: Differential drive Braitenberg vehicles where S_l and S_r are sensor activation values [39]. In this case, sensor activation increases linearly as the vehicle approaches the light source. The activation can be chosen to increase or decrease motor activation, represented with a + and -, respectively. As vehicle (a) approaches the light source, the right wheel increases its velocity to a greater extent than the left wheel. This gives the motion of the vehicle steering away of being fearful of the light source. Vehicle's (b), (c), and (d) motion can be described using a similar logic.

If the sensor activation is non-linear, exotic motion shown in Figure 3.6 can be achieved. From these seemingly simple vehicles, complex behaviours and non-trivial tasks can be achieved with little to no computational cost. Lilienthal and Duckett use the Love vehicle to locate a gas source in a room [40]. Braitenberg vehicles have been used as computational tools to model the behaviour of animals, such as the echolocation in bats [41] and the co-ordination of swarm robots to search for targets and move in formation [42]. This report uses the fear vehicle to avoid duckies, representing pedestrians in Duckietown.

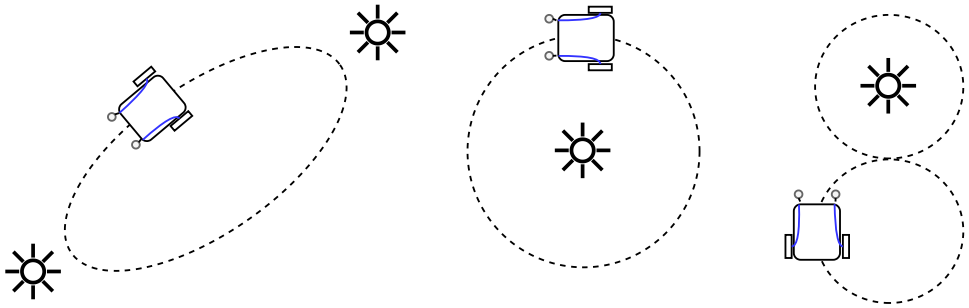


Figure 3.6: Complex motions using the same sensorimotor connections with non-linear sensor activation [39]

CHAPTER 4

THE DUCKIE DETECTOR

YOLOv5 Object Detection Model

> A system is composed of components; a component is something you understand
>
> Howard Aiken

This chapter discusses the *Duckie Detector*, a YOLOv5 object detection model. The chapter first explains the YOLOv5 network. The focus then shifts to the dataset generation and training of the model. Finally, the chapter presents a discussion of the training results.

4.1 YOLOv5 Architecture

YOLOv5 was released with five different sizes shown in Table 4.1 [1]. All the models use the same architecture shown in Figure 4.1, except the number of layers and, consequently, the number of parameters change within the convolution layers. The performance of the model, based on the `COCO 2017 Val` [2] dataset improves as the number of parameters increases, but the inference time increases as the number of FLOPs increases. As computing is limited on the duckiebot and real-time inference is required, the smallest YOLOv5 model was chosen.

[1] Table 4.1 only includes YOLOv5 advertised CPU inference times as the GPU times are performed on the Nvidia Tesla V100 GPU which is a 32GB, 128 core GPU compared to the Nvidia Jetson Nano 2GB. According to [43], the Nano is up to 60 times slower and was deemed unrepresentative.

[2] Microsoft Common Object Detection in Context (COCO) is the benchmark for evaluating the performance of SOTA computer vision models

Table 4.1: YOLOv5 model Sizes

Model Size	Speed CPU (ms)	Parameters (M)	FLOPs@640 (B)	mAP-50
YOLOv5n	45	1.9	4.5	45.7
YOLOv5s	98	7.2	16.5	56.8
YOLOv5m	224	21.2	49.0	64.1
YOLOv5l	430	46.5	109.1	67.3
YOLOv5x	766	86.7	205.7	68.9

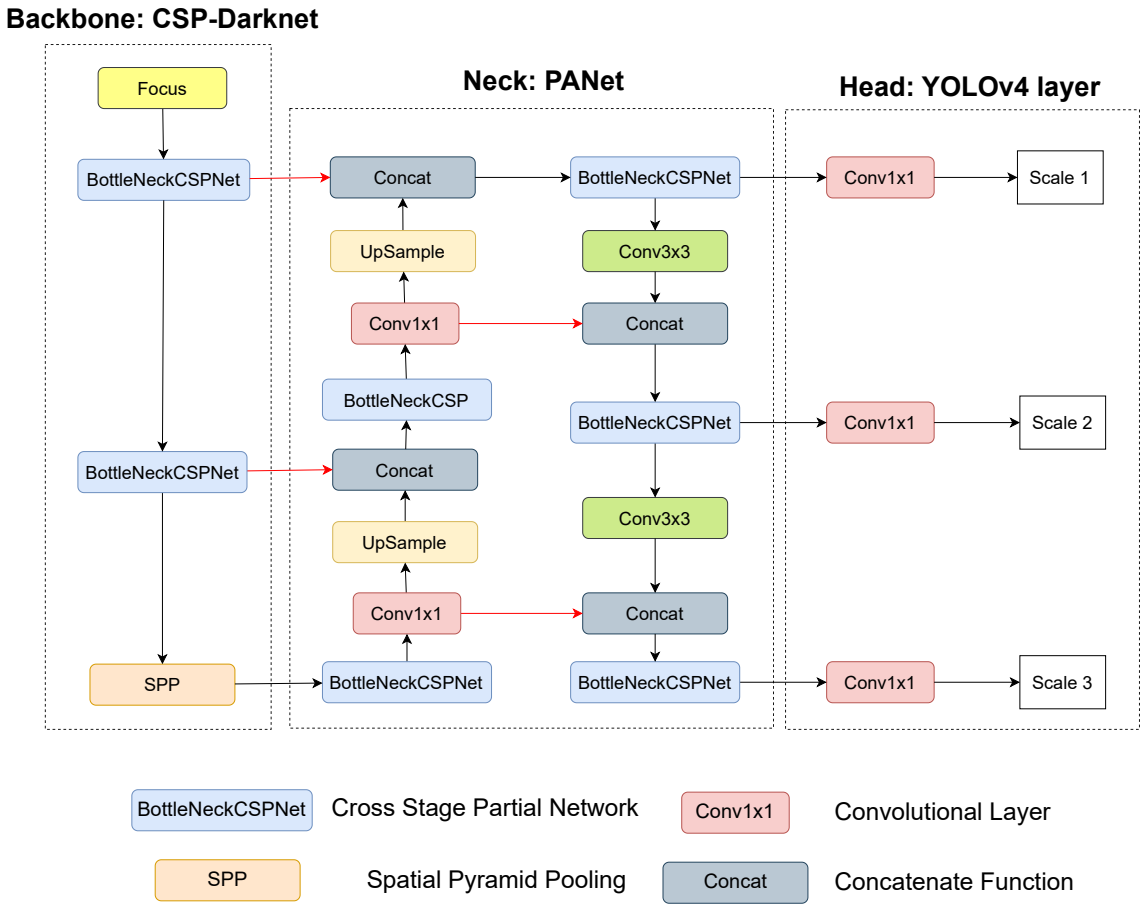


Figure 4.1: YOLOv5 network architecture. The network consists of three parts: CSP-Darknet backbone, PANet neck and YOLOv4 head. The backbone extracts features, the neck fuses various extracted features, and the head outputs detections [31].

Consider the YOLOv5 architecture shown in Figure 4.1, the network is composed of 3 components: CSP-Darknet53 backbone, PANet neck and YOLOv4 head [31]. In the YOLOv5 backbone, the Darknet CNN is used to extract features. Incorporating the Cross Stage Partial Network (CSPNet) ensures features at different granularities are integrated into the feature map at different stages of the network, as shown by the red arrows in Figure 4.1. According to the Wang et al. CSPNet helps overcome the *vanishing gradient problem* by truncating the gradient flow and allowing a deeper backbone [44]. *Vanishing*

gradient is the phenomenon where the gradient of the loss function with respect to the weights becomes very small as the number of layers increases, limiting the depth of the network. The BottleNeckCSP internal mechanisms explained in Figure 4.2 help reduce the number of parameters, increasing the inference speed of the model, which is crucial for real-time applications [45].

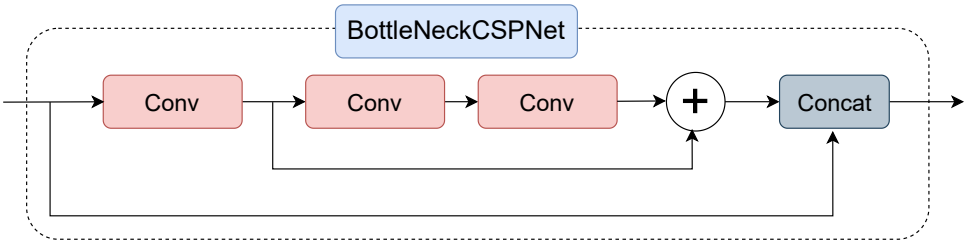


Figure 4.2: BottleNeckCSP internal mechanism [45]. The BottleNeckCSP is known as a residual layer. The input is passed through the convolution layer, and the output is concatenated with the input. The addition of the intermediary feature maps is known as the residual connection. The motivation for feeding the input to the output is that CNN only learns the residual or difference of the two layers.

The Spatial Pyramid Pooling (SPP) and Path Aggregation Network (PANet) are leveraged for feature fusion in the neck. SPP performs max pooling sequentially to the feature map and then concatenates them together, as shown in Figure 4.3. This feature, taking inspiration from the human eye, compresses the feature map generated by the backbone [30]. The reason is twofold: the compression reduces the network's computational complexity, and pooling stages extract the main features from the feature map.

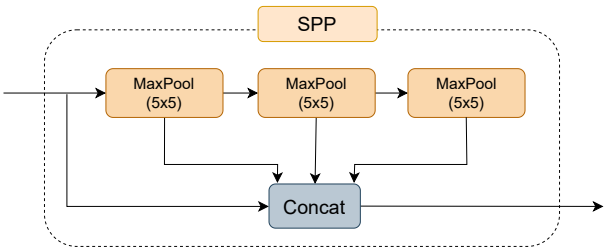


Figure 4.3: Spatial Pyramid Pooling internal mechanism [30]. The purpose of pooling at different scales is to capture various levels of abstraction. Fusing these features allows the model to learn features at different scales.

PANet is a Feature Pyramid Network (FPN) that incorporates the backbone's information at different stages to boost information flow [44]. YOLOv5 uses a bottom-up PANet where the SPP output is integrated with features extracted from lower levels in the backbone shown in Figure 4.1. The authors of YOLOv5 believe this approach improves the

propagation of accurate localisation signals in low-level features, enhancing the model's accuracy [46].

The head comprises three convolution layers that predict the bounding box, scores and object classes. Single-stage networks like YOLO are typically worse than two-stage at detecting objects at different scales, as explained in Chapter 3. YOLOv5 uses features from different stages of the pyramid to generate feature maps at three different scales (18x18, 36x36, 72x72), allowing the model to handle small, medium and large objects [31].

4.2 Data Collection

The dataset consists of two primary sources, namely, the Duckietown simulator, Duckietown-gym [35], and the Duckietown real-world dataset. A real-world Duckietown dataset is publicly available with 1956 images of the Duckietown environment [47]. Table 4.2 shows a breakdown of the various classes labelled in the dataset, and Figure 4.4 shows labelled samples from the dataset. The images are resized from a resolution 640×480 to 416×416 with the OpenCV `resize` function. Using a smaller input matrix reduces the computation cost of inference, and ensuring a square input further improves training efficiency by taking advantage of matrix multiplication optimisations.

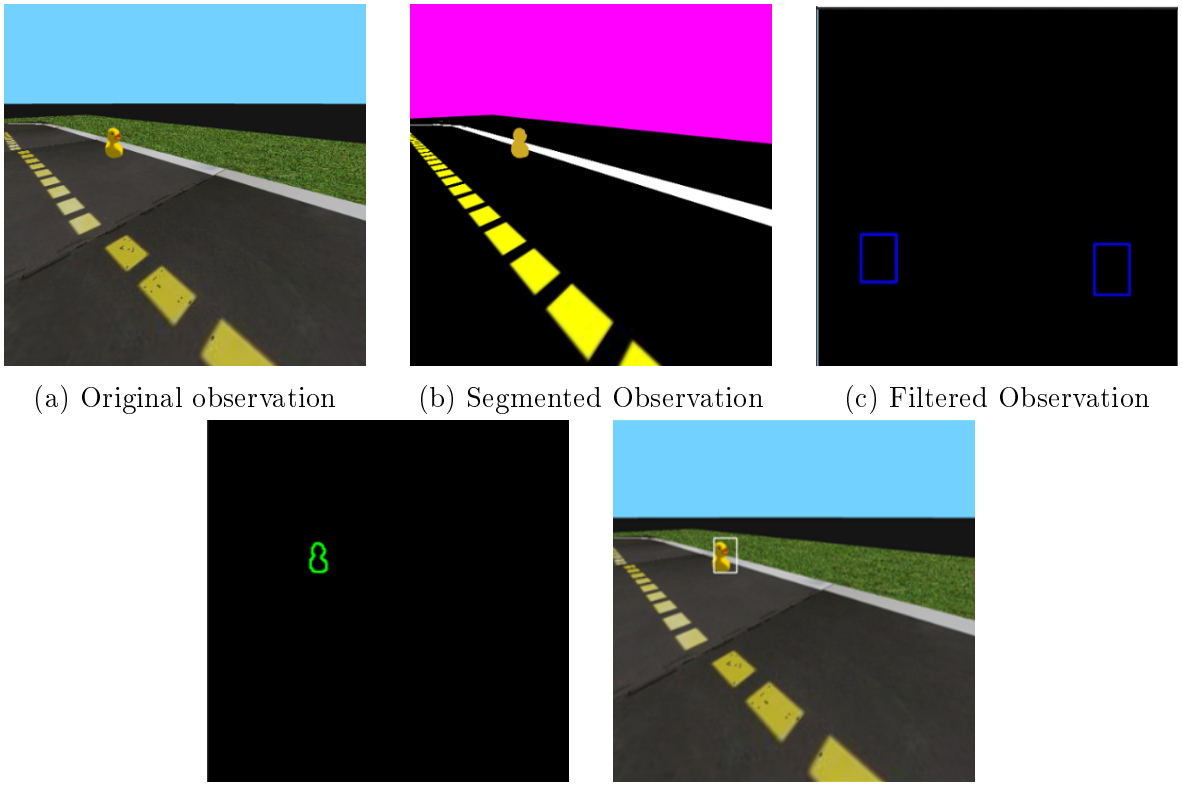
Table 4.2: Real-world dataset class details

Class	ID	Instances
Cones	1	372
Duckies	2	2570
Duckiebot (Old)	3	1419
Duckiebot (New)	3	707

(a) Labelled sample (b) Resized sample

Figure 4.4: Sample image from the duckietown dataset. (a) shows a labelled image from the dataset (b) shows the sample resized for YOLOv5 training

The Duckietown-gym simulator is a valuable resource for generating larger datasets. It is widely accepted that the most influential aspect of a CNN's performance is the size and richness of the dataset. Duckietown-gym provides facilities for segmented simulation [3] where a single pixel value represents each class (or object) in the simulation. In this way, the labelling process is a matter of filtering out specific pixel values, determining the object's contour and drawing a bounding box around the contour outlined in Figure 4.5. The `OpenCV` functions `findContours` and `boundingRect` were used to determine the contour and bounding box shown in Figure 4.5 (d) and (e) respectively. A publicly available Pure Pursuit agent [4] was used for the dataset generation in the Duckietown-gym environment. This agent drives around the simulation environment following the centre of the lane and collects observations. `data_collection.py`, in the `DT_train` directory, contains the code for the labelling process outlined in Figure 4.5.



(a) Original observation (b) Segmented Observation (c) Filtered Observation

(d) Contour of duckie (e) Bounding box

Figure 4.5: Process for labeling a simulated observation

[3] In segmented mode, the simulator publishes two observations: simulator observation with realistic graphics and, by extensions, realistic pixel values and segmented observations. The segmented observation is used for labelling bounding boxes, whereas the simulator observation is used for training

[4] The agent can be found at this repository

Increasing the size of the dataset using simulated observation then becomes trivial. However, *overfitting* [5] and the presence of a *domain gap* [6] are limitations with using a disproportionately large split of simulated images in the dataset. To mitigate this, domain randomisation, accurate camera distortion and differential-drive physics (realistic waddling) are Duckietown-gym features that allow for the 2:1 split of simulated to real images used in this report. *Domain randomisation* is the process of randomising the simulated environment to make it more representative of the real-world environment. Figure 4.6 shows that it is achieved by randomising the lighting, texture, colour and position of objects in the simulated environment. Horváth et al. show that using domain randomisation techniques, a Yolov4 model can be trained using only a single real image and the rest of the simulated images motivating the 2:1 split used here [48].

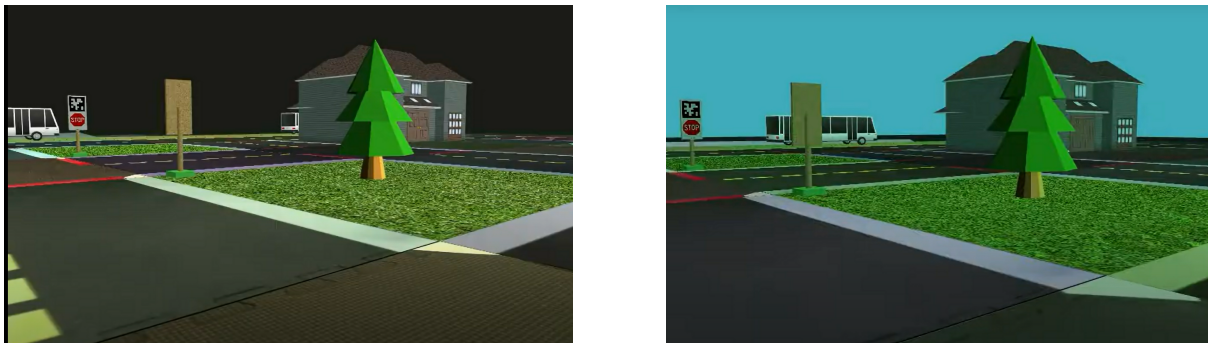


Figure 4.6: Domain randomisation in the Duckietown-gym simulation environment

YOLOv5 utilises the `[x_min y_min width height]` convention for the bounding box, with values normalised relative to the image width and height. This normalisation enhances training efficiency with faster convergence and improved stability of backpropagation gradients. Bounding box labels are stored in a `.txt` file sharing the same name as the image file. The `DT_train` repository contains various wrappers facilitating the generation and formatting of the dataset. The dataset is comprised of a training and validation section. The validation set assesses how well the model generalises to new data, and training is stopped early if the model performs worse as it is overfitting. Table 4.3 shows a dataset summary using an 80:20 split.

[5] Overfitting occurs when a model learns the training dataset too well and cannot generalise to new data.

[6] Domain Gap occurs when the distribution of the training dataset is different from the real-world dataset.

Table 4.3: Summary of training dataset

Type	Origin	Instances
Training	simulation	3200
	real	1564
Validation	simulation	800
	real	392

YOLOv5 provides compatibility for both Gradient Descent (GD) and ADAM (Adaptive Moment Estimation) optimisers. These optimisers are explained in Appendix B. Choosing an optimiser requires in-depth knowledge of the network architecture and stability criteria outside this report's scope. Nguyen et al. analyse GD and ADAM using YOLOv5 on the duckiebot and show GD methods perform best [30]. Masters and Luschi review different batch sizes and found batch sizes between 2 and 32 produce the best results [49]. This report uses the median batch size of 16 as a compromise between training time and optimisation stability. Table 4.4 summarises the hyperparameters used in training the YOLOv5 model.

Table 4.4: Summary of hyperparameters used for training the model

Parameter	Value	Explanation
Model	yolov5n	YOLOv5 architecture
epoch	50	number of complete passes through dataset
img	416	Dimension of input image
weights	default	Initial value of parameters
batch	16	Mini-batch GD batch size
hyps	hyp.scratch-med.yaml	Hyperparamters for medium-augmentation COCO 2018 training

`hyps` in Table 4.4 refers to a yaml file of additional hyperparameters for the loss function, data-augmentation and optimisation, found in the `hyp.scratch-med.yaml` [7] file shown in Appendix C.1. These are the YOLOv5 recommended hyperparameters for customising a YOLOv5 pretrained model. YOLOv5 can be trained from scratch using randomised parameter weights. Alternatively, YOLOv5 provides the capability for transfer learning and is the recommended training method. The default model is trained on the `COCO 128` dataset and has learnt good feature representation, resulting in much shorter training times than training from scratch.

[7] See https://github.com/ultralytics/yolov5/blob/master/data/hyps/hyp.scratch-med.yaml

A crucial aspect of training is defining the loss function, $\mathcal{L}$. In the case of YOLOv5, the loss function comprises three separate losses shown in Eq. 4.1, optimised during training. Table 4.5 describes each term Eq. 4.2, 4.3 and 4.4 derive each term.

$$\mathcal{L} = \mathcal{L}_{box} + \mathcal{L}_{cls} + \mathcal{L}_{obj} \tag{4.1}$$

Table 4.5: YOLOv5 loss function terms

$\mathcal{L}_{box}$	Complete Intersection over Union loss of the predicted bounding box overlap with ground truth bounding box
$\mathcal{L}_{cls}$	Binary Cross Entropy loss of the confidence of predicted class
$\mathcal{L}_{obj}$	L1 loss of the confidence of object's presence

$$\mathcal{L}_{box} = \lambda_{box} \sum_{i=0}^{S^2} \sum_{j=0}^{B} I_{ij}^{obj} [(x_i - x_i^{\Lambda j})^2 + (y_i - y_i^{\Lambda j})^2 + (w_i - w_i^{\Lambda j})^2 + (h_i - h_i^{\Lambda j})^2] \tag{4.2}$$

$$\mathcal{L}_{cls} = \lambda_{cls} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \sum_{c \in \text{classes}} I_{ij}^{obj} [P_i^{\Lambda j} \log(P_i^j)] \tag{4.3}$$

$$\mathcal{L}_{obj} = \lambda_{obj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} I_{ij}^{obj} [C_i - C_i^{\Lambda j}] + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} I_{ij}^{noobj} [C_i - C^{\Lambda j}] \tag{4.4}$$

Recall that YOLO discretises the image into a $S \times S$ grid. Each loss term calculates the loss for the predicted object, j, in grid element i. I_{ij}^{obj} is set to 1 if an object is present in the location and 0 otherwise. The superscript Λj denotes model predictions, and terms without the superscript are ground truth values. x and y are the centre of the predicted object, w and h are the width and height of the bounding box, respectively. P_i is the probability of an individual class, and C_i is the confidence of an object's presence at the location. The loss coefficient, λ_{box}, λ_{cls}, λ_{obj} and λ_{noobj}, measure the importance of the different losses. This report uses the suggested values specified in the `hyp.scratch-med.yaml` file.

YOLOv5 employs data augmentation features to improve the model's generalisation. The motivation is that the model is forced to find new features instead of only relying on the features present in the original dataset. Mosaics are one feature where four images are combined into a single image, shown in Figure 4.7 [50]. This is done by randomly selecting four images from the dataset and creating a 2x2 grid. The bounding boxes are then adjusted accordingly. Mixup [51] averages two images together according to Eq. 4.5

and 4.6 based on a sampling parameter, λ, defined from a beta distribution represented by Eq. 4.7.

$$\hat{x} = \lambda x_1 + (1 - \lambda)x_2 \quad \text{where } x_1, x_2 \text{ are images} \tag{4.5}$$

$$\hat{y} = \lambda y_1 + (1 - \lambda)y_2 \quad \text{where } y_1, y_2 \text{ are one-hot label encodings} \tag{4.6}$$

$$\lambda \sim \text{Beta}(\alpha, \alpha) \quad \text{where } \alpha \in (0, \infty) \tag{4.7}$$

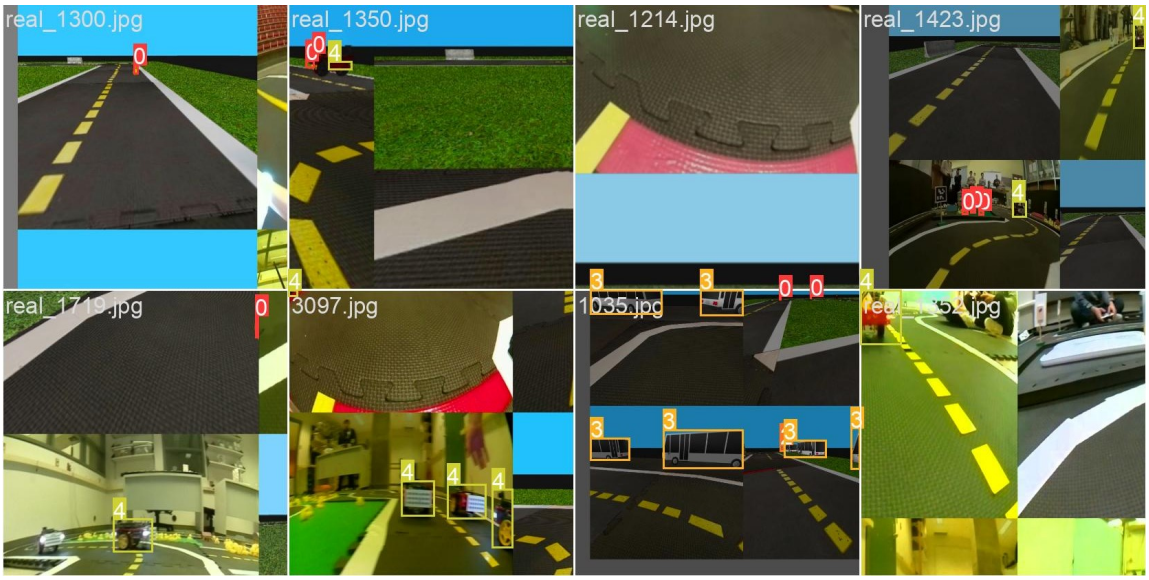


Figure 4.7: Training images showing the use of mosaics and mixup

Mixup extends the training distribution by incorporating prior knowledge that linear interpolations of features should result in linear interpolation of the associated objects [51]. This report uses $\alpha = 0.10$, specified in the `hyp.scratch-med.yaml` file. These augmentations are present in some of the images in Figure 4.7 but are less visible to the eye as the weighting parameter is usually close to 1. Nonetheless, these changes are significant enough to improve the model generalisation and robustness to adversarial cases mentioned in Section 2.1 [51]. In summary, these augmentations prevent overfitting and improve the model's generalisation by exposing the model to different training images while using the same base images.

Google Colab's TPU [8] resources were leveraged for training the model as this allowed more epochs or complete passes through the dataset while keeping the training time reasonable. The Google lab training script details are in the `DT_training.ipynb` script. In summary, the `yolov5n.yaml` file defines the YOLOv5 model, the directory of the dataset and the

[8] Colab is Google's cloud-based computing resource. A TPU is an application-specific integrated circuit (ASIC) designed by Google for ML applications.

number and name of the classes. YOLOv5 uses a simple API for training, where the user can train the model in a single line by specifying the hyperparameters.

```
python3 train.py --cfg ./models/yolov5n.yaml --img 416 --batch
 16 --epochs 50 --data duckietown.yaml --weights yolov5n.pt --
hyp hyp.scratch-med.yaml
```

Listing 4.1: YOLOv5 Training API

4.3 YOLOv5 Training Results

Figure 4.8 and Figure 4.9 show the loss function components defined in the previous section for training and validation. The horizontal axis represents the epoch number, N_e, and the vertical axis relates to the specific loss terms. The plots show that convergence was achieved within the 50 epochs as the loss curves asymptotically tend to a steady value for training and validation. Additionally, the training and validation plots are approximately the same shape, indicating that the model was not overfitting on training data. Note how the value of the loss dramatically decreases during the first few epochs, which is a product of starting with the pertained weights that have good feature representation.

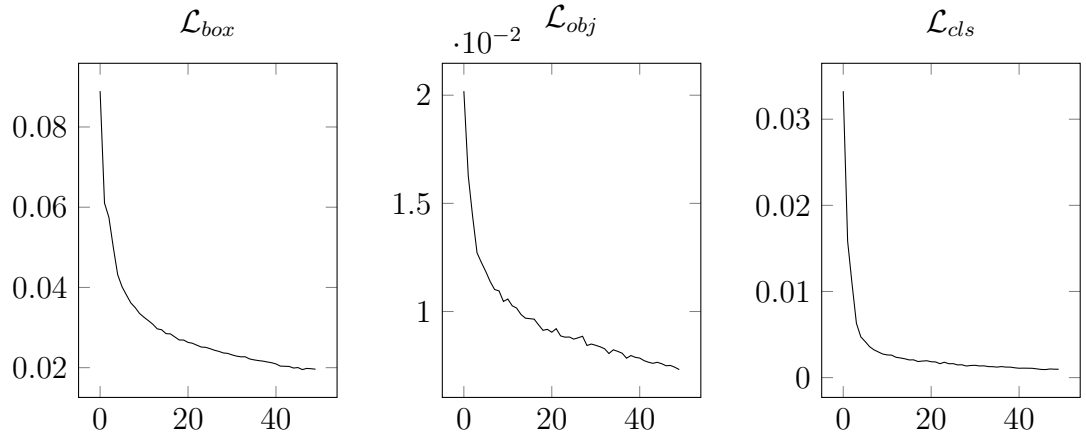


Figure 4.8: Training loss functions for 50 epochs.

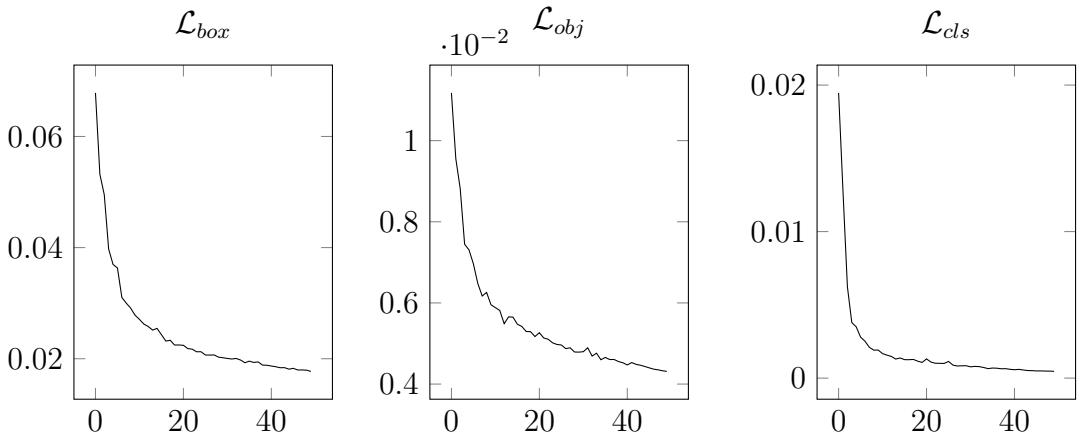


Figure 4.9: Validation loss functions for 50 epochs.

The performance of an object detection model is traditionally accessed using the metrics precision (Pr), recall (Re), mean average precision (mAP) and IoU [9] derived in Eqs. 4.8, 4.9, 4.11 and 4.12. TP, FP and FN are the numbers of True Postivies, False Postivies and False Negatives respectively. N_c is the number of classes. B is the ground truth bounding box area and B^{Λ^j} is the predicted bounding box area.

$$Pr = \frac{TP}{TP + FP} \tag{4.8}$$

$$Re = \frac{TP}{TP + FN} \tag{4.9}$$

$$AP_k = \frac{1}{11} \sum_{R_i} Pr(Re_i)_k \tag{4.10}$$

$$\text{mAP} = \frac{1}{n} \sum_{k=1}^{N_c} AP_k \tag{4.11}$$

$$IoU = \frac{|B^{\Lambda^j} \cap B|}{|B^{\Lambda^j} \cup B|} \tag{4.12}$$

In simpler terms, Pr is the fraction of positive detections that are correct, and Re is the fraction of found positive predictions. Therefore, Pr is concerned with FPs, and Re is concerned with FNs. IoU measures the overlap between the ground-truth and predicted bounding boxes. Eq. 4.10 refers to a single class's average precision (AP). AP is defined as the area under the Precision-Recall (PR) curve and summarises the PR curve to a single scalar. In practice, the PR curve is obtained by interpolating 11 equally spaced recall: $\{0, 0.1, 0.2, \ldots, 1.0\}$. mAP is defined as the average AP across all N_c classes. **mAP-0.5** and **mAP-0.5:0.95** are common metrics used and have been widely adopted to summarise a

[9] A visual derivation is presented in Appendix C.2

model's performance [32]. mAP-0.5 is evaluated at an IoU value of 0.5 or 50%, meaning that all predicted bounding boxes with an IoU, compared to the ground truth bounding, of less than 50% are discarded. mAP-0.5:0.95 is the official COCO Challenge metric for evaluating a model and considers 10 IoU thresholds from 0.5 to 0.95.

Figures 4.10 and 4.11 show the various performance metrics explained above.

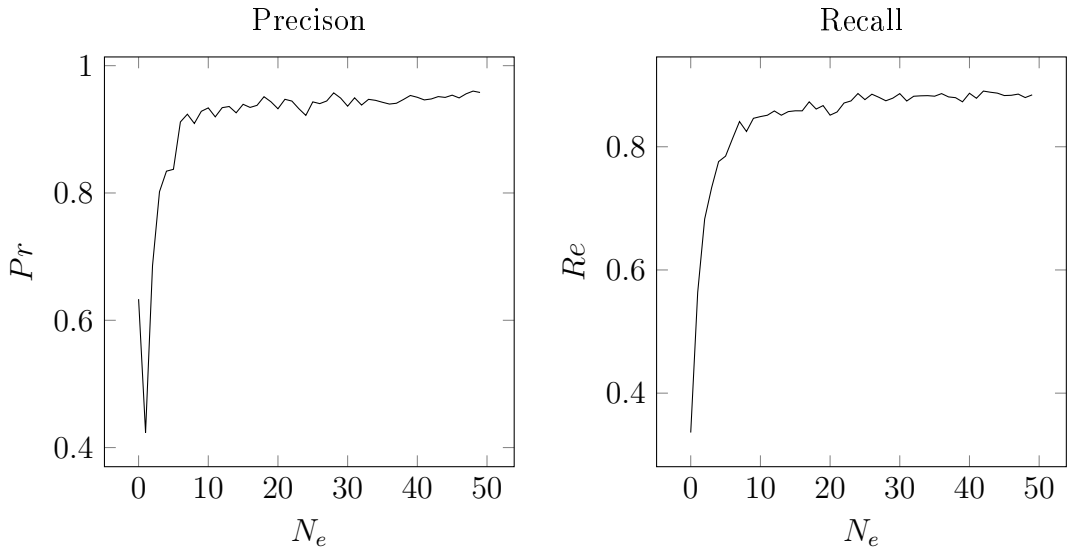


Figure 4.10: Precison, Recall for 50 training epochs where both metrics converge to a horizontal asymptote. Table 4.6 shows the final values from each metric.

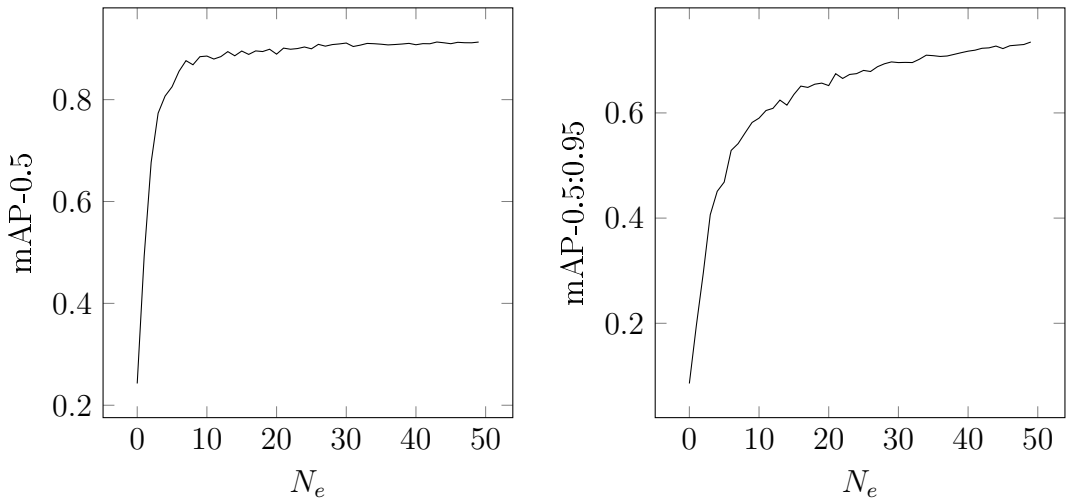


Figure 4.11: mAP-50 and mAP-0.5:0.95 for 50 training epochs. mAP 0.5 appears to have converged, whereas mAP-0.5:0.95 appears on a slight upward trajectory, indicating that it would improve given further training epochs. However, it was decided that additional epochs would only marginally increase this value. Table 4.6 shows the final values for each metric.

The YOLO model trained in this report is compared to other trained models by Nguyen et al. [30]. Table 4.6 compares the results for the various models. Nguyen et al. used the

larger YOLOv5 architectures compared to the Yolov5n model used in this report. Table 4.6 shows that the model trained in this report, despite being smaller and trained on fewer epochs, shows comparable results and is better in some cases. mAP-0.5:0.95, the official COCO metric [52], is the most recognised between mAP-0.5 and mAP-0.5:0.95 metrics, further validating the performance of the trained model in this report. Nguyen et al. only trained the model on the real Duckietown dataset. This shows that including the synthetic, simulated dataset with domain randomisation was an effective means to improve training performance.

Model	**Epochs**	**Parameters**	**Pr**	**Re**	**mAP@0.5**	**mAP@0.5:0.95**
Yolov5n	50	1.9 m	**0.958**	0.884	0.913	**0.735**
Yolov5s [30]	100	7.2 m	0.929	0.939	0.969	0.654
Yolov5l [30]	100	46. 5m	0.95	**0.944**	**0.978**	0.661

Table 4.6: Comparison of the YOLO model trained here (in bold) compared to various other models trained by the authors of [30]. Bold values indicate the best results.

Consider the confusion matrix in Figure 4.12. The value of each element in the matrix indicates the proportion of a ground truth class that was attributed to a given predicted class. In other words, 91% of the total duckies were correctly detected as duckies in the validation set. 9% of ground truth duckie labels were predicted as the background or False Negatives. The confusion matrix is useful in exposing weaknesses or codependence in the model. In this case, the model does not appear to confuse any classes, as all the off-diagonals are zero. The major confusion occurs with the background, which is expected given that cones and duckies are relatively simple shapes that can be easily confused. However, the high precision of the model suggests that this should not be an issue. Notice there is minimal confusion for the truck and bus classes as these objects have defined features that the model can learn.

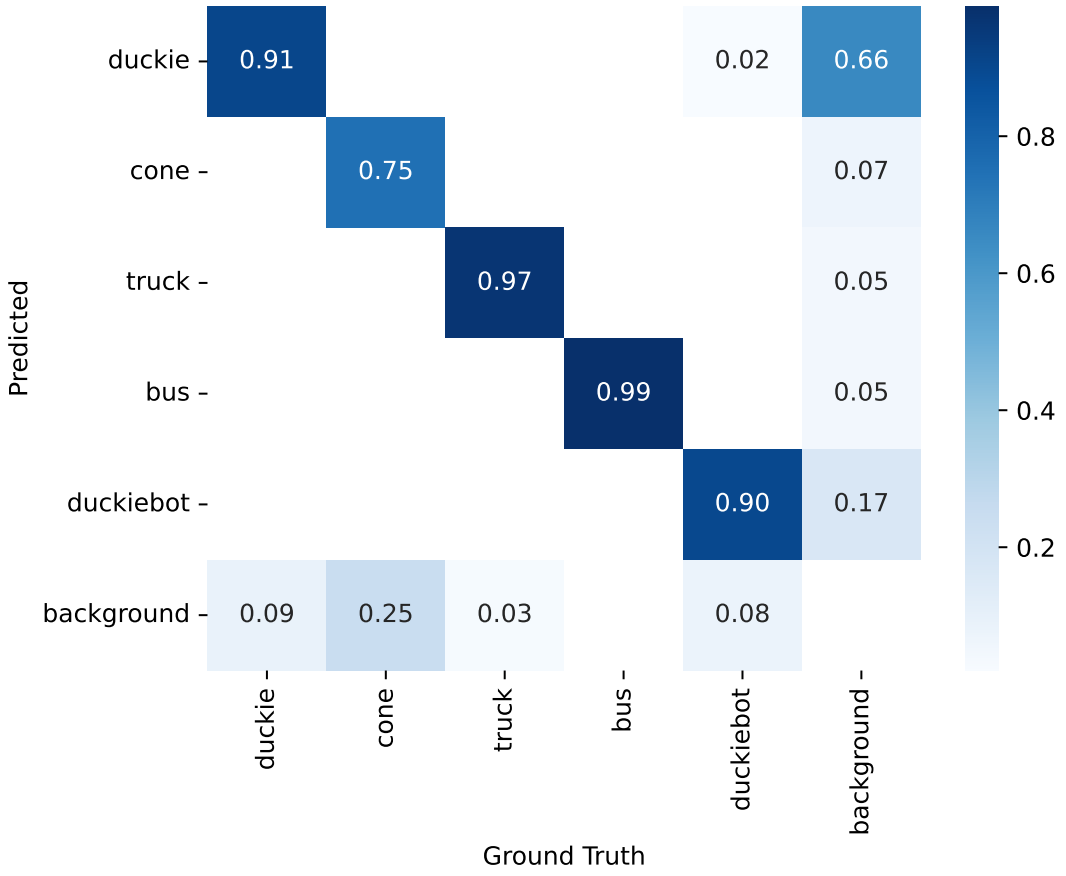


Figure 4.12: Confusion matrix of the YOLOv5 model based on the validation set. 66% of FP detections in the background were predicted as duckies. This may appear significant, but the background FPs are still relatively rare given the high Pr of the model.

4.4 Conclusion

Overall, the results show that including the synthetic, simulated data, with data augmentation techniques like domain randomisation, was an effective means to supplement the training dataset. The model showed competitive results despite having fewer parameters and fewer training epochs. The next chapter shows the implementation of the YOLOv5 model on the duckiebot. Appendix C.3 provides additional training results for the interested reader.

CHAPTER 5

EYES FOR THE DUCKIEBOT

A Duckiebot implementation

> Vision without action is a daydream. Action without vision is a nightmare
>
> Japanese proverb

The implementation of the *YOLOv5 Duckie Detector*, on the duckiebot, is discussed in this chapter. A simple Bratienburg controller is used to test the model's efficacy which is acting as an obstacle avoidance agent. The chapter provides validation results of the YOLOv5 model and the Braitenberg controller, followed by a discussion.

5.1 ROS Agent

The Duckietown platform uses ROS as its middleware and Docker for containerisation. A ROS agent containing multiple ROS packages, each as its own Docker container, is deployed on the duckiebot. ROS and Docker are not part of the undergraduate curriculum and are discussed in Appendix D.1 and D.2 for the interested reader. In summary, ROS provides the software infrastructure for integrating high-level logic with hardware. Docker facilitates the isolation and reproducibility of software components, making it an ideal platform for AV development. In this way, Docker is similar to a virtual machine but is more lightweight and efficient, therefore suited to robotics.

Figure 5.1 shows the simplified ROS computation graph of the agent. The `camera_node` publishes images from the camera to an image topic. The `object_detection_node` houses the YOLOv5 model and Braitenberg controller. This node subscribes to the image topic and uses the image as input to the YOLOv5 model. The detection is then

fed to the Bratienburg controller that computes wheel commands and publishes them to the `wheels_driver_node`. Both the `camera_node` and `wheels_driver_node` are native duckietown nodes and not discussed here.

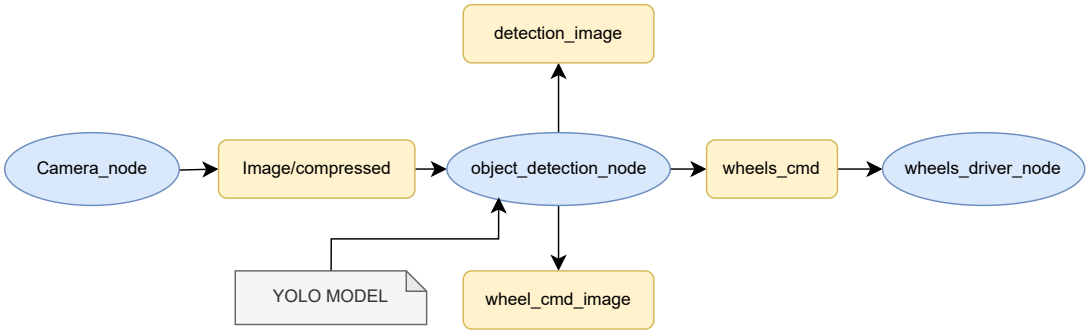


Figure 5.1: ROS computation graph. Ovals indicate nodes, rectangles indicate topics and the document represens data loaded from an external file. In reality, the computation graph contains many more nodes and topics related to drivers, file managers and device health which are omitted for the sake of clarity

5.2 `object_detection_node`

Figure 5.2 shows the computation graph for the `object_detection_node`. The repository location for this node can be found in Appendix A. `image_cb` is a function called each time an image is published to the `image/compressed` topic and invokes the YOLOv5 model. The camera provides images at 30 FPS or every 33 ms. However, the advertised YOLOv5 inference time is 45 ms [1]. As a result, the YOLOv5 model is only called after a specified numb er of frames. This report calls the model every two frames or 15 FPS.

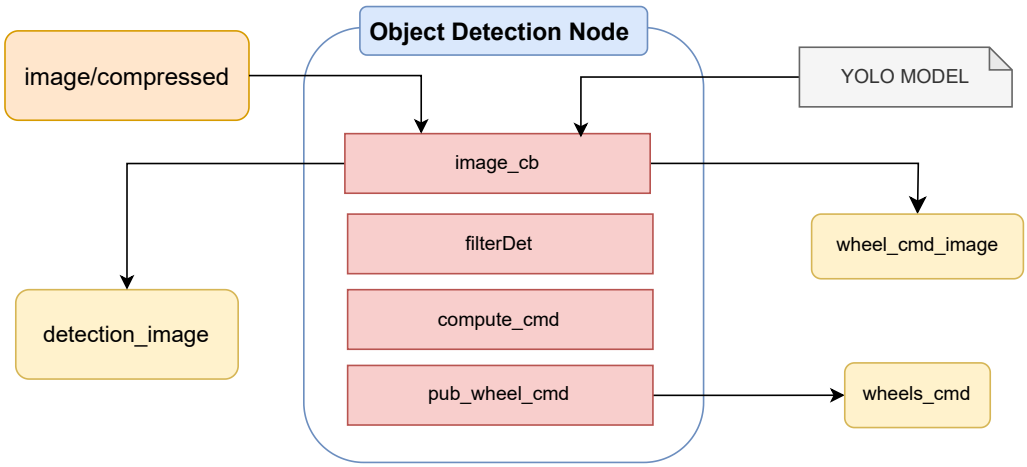


Figure 5.2: object_detection_node computation digram. Rounded rectangles represent topics, and square corner rectangles are functions within the node.

[1] Actual inference times are shown in Section 5.5

Figure 5.3 shows the algorithm for publishing wheel commands.

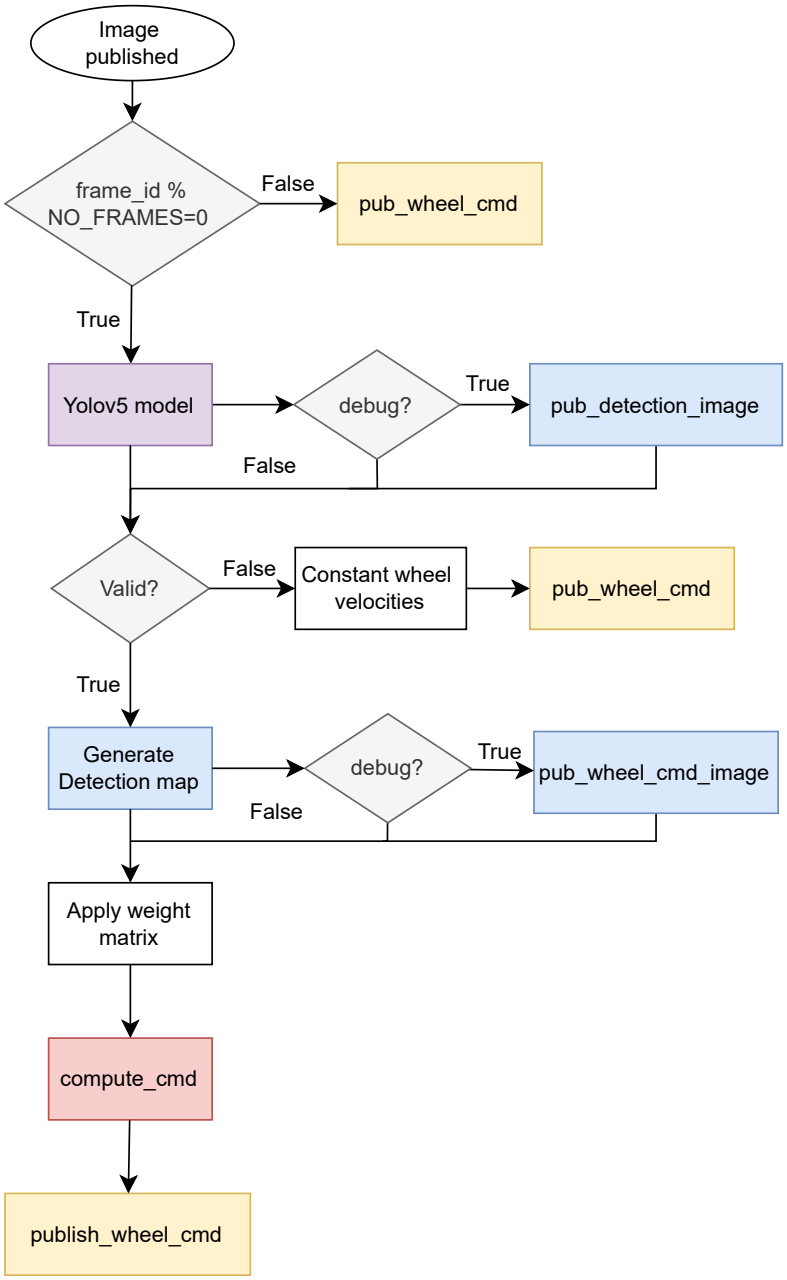


Figure 5.3: Block diagram for computing wheel commands upon an image being published to the image topic. The red `compute_cmd` function is only called for a valid detection. Therefore, for a frame ID that is not a multiple of `NO_FRAMES`, previous wheel commands are published, and for an invalid detection, constant wheel commands are published. Images are only published if the `debug` variable is set to true to reduce computational load.

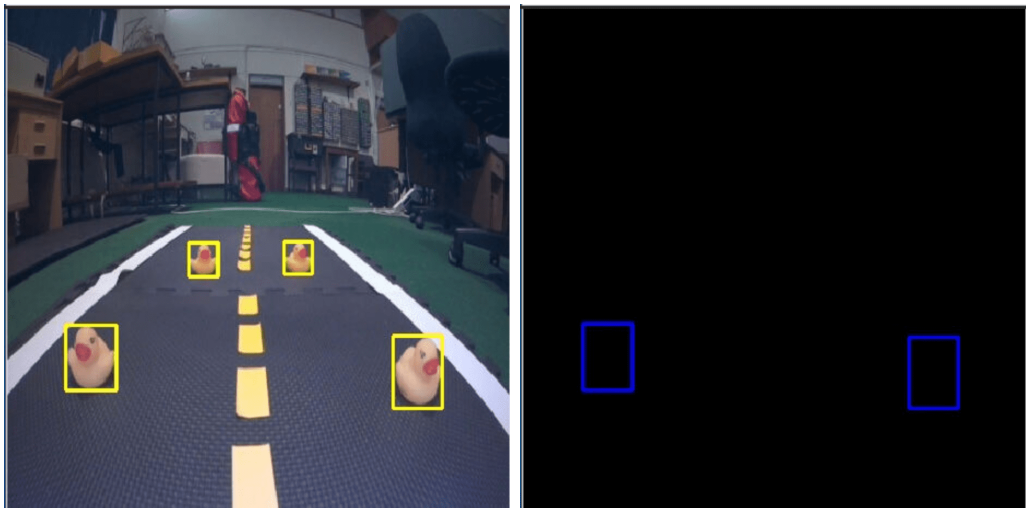
YOLO predictions are filtered by classes, scores and bounding boxes to determine a valid detection by the `filterDet` function. Table 5.1 elaborates on the three filters used to determine a valid detection. `bboxes` in Table 5.1 refers to the minimum bounding box area for a valid detection. This eliminates small detections of duckies relatively further

from the duckiebot. A bounding box area of 1000 roughly translates to a 0.5m radius around the duckiebot.

Table 5.1: Filters applied to YOLO predictions to determine a valid detection.

Filter	Explanation	Value
`classes`	Only user-specified classes result in a valid detection	duckies, duckiebots
`scores`	Minimum confidence for a valid prediction	0.8
`bboxes`	Minimum bounding box area for a valid prediction	1000

Given a valid detection, the `image_cb` function generates a new image **M** that maps the filtered bounding boxes shown in Figure 5.4. In other words, only bounding boxes that result in valid detections are mapped.

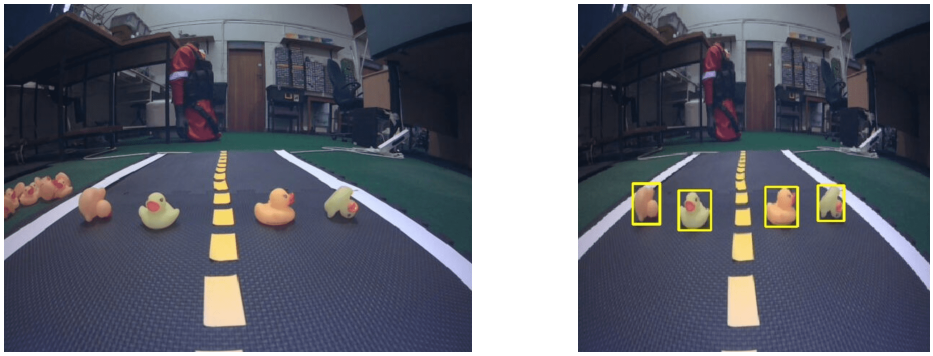


(a) Unfiltered (b) Filtered

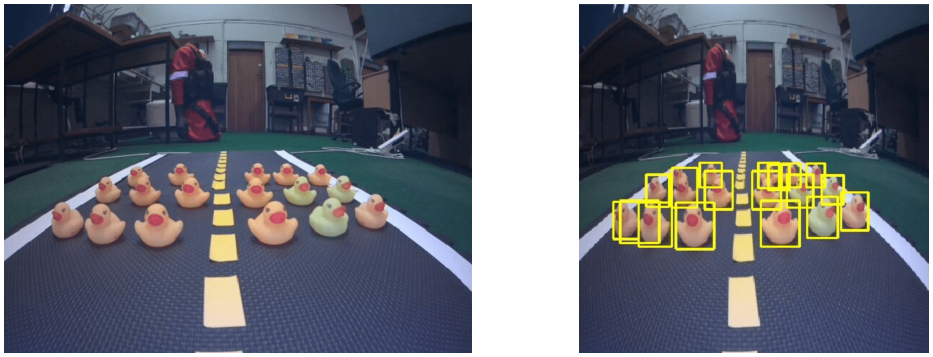
Figure 5.4: Bounding boxes map of valid detections. The duckies further in the scene have been filtered out of the bounding box map.

5.3 YOLOv5 Validation

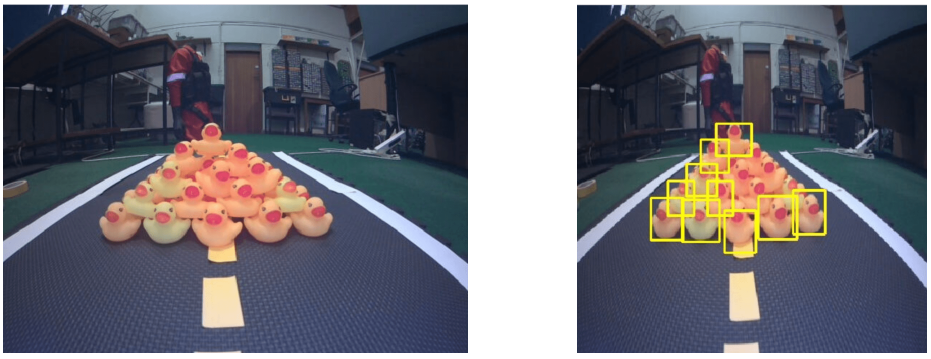
Results for various validation scenarios are shown in the Figures below:



(a) Abbey Road by the Quackles



(b) Duckie Army



(c) Duckie Pyramid

(d) Duckie Impersonators

Figure 5.5: YOLOv5 predictions for various validation scenairos

The YOLO model proves to be accurate for scenarios within the Duckietown environment when duckies are spaced out. Figure 5.5a shows that the model can identify duckies in different orientations. The model struggles with crowded duckies, demonstrated in Figure 5.5c. This is a general weakness of the YOLO algorithm, which relies on a limited number of grids responsible for detection. Another contributing factor is that there were limited training examples of crowded duckies. Finally, it may suggest that the model is learning the outline of the duckie, which is not as distinct in a crowd. However, no rigorous feature visualisation was conducted to validate this claim. This weakness can be deemed acceptable for this application as only a single detection is required for avoidance. Figure 5.5d shows an attempt to confuse the model with unseen objects from the lab and indicates that the model is not overfitting to the Duckietown environment. In theory, the YOLOv5 model should be able to detect duckiebots cones; however, only duckies were available for validation.

5.4 Braitenberg Controller

The `compute_cmd` function implements the Bratienberg controller. Traditionally, Braitenberg vehicles are controlled by having two sensors on either side of the vehicle detecting some quantity. Here, the detection map $\mathbf{M}$ is divided into left and right regions. A weight matrix $\mathbf{W}$ is created for each image region shown Figure 5.6. Performing element-wise multiplication with the $\mathbf{M}$ generated above, an activation for each image region, $\boldsymbol{A_l}$ and $\boldsymbol{A_r}$, is computed as the sum of the elements shown in Eq. 5.1 where the map is quare with a size m.

1	1	1	0	0	0
1	1	1	0	0	0
1	1	1	0	0	0
1	1	1	0	0	0
1	1	1	0	0	0
1	1	1	0	0	0

(a) Weight matrix for left region $\boldsymbol{W_l}$

0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1

(b) Weight matrix for right region $\boldsymbol{W_r}$

Figure 5.6: Weight matrices for the left $\boldsymbol{W_l}$ and right $\boldsymbol{W_r}$ image regions

$$A_s = \sum_{i,j}^{m} (\mathbf{M} \odot \mathbf{W_s})_{i,j} \quad s \in (l, r) \tag{5.1}$$

These activation values are generally large and must be normalised to the PWM signals between -1 and 1. Normalisation is achieved according to Algorithm 1, which uses the history of the maximum and minimum activation values.

Algorithm 1 Activation Normalisation Algorithm

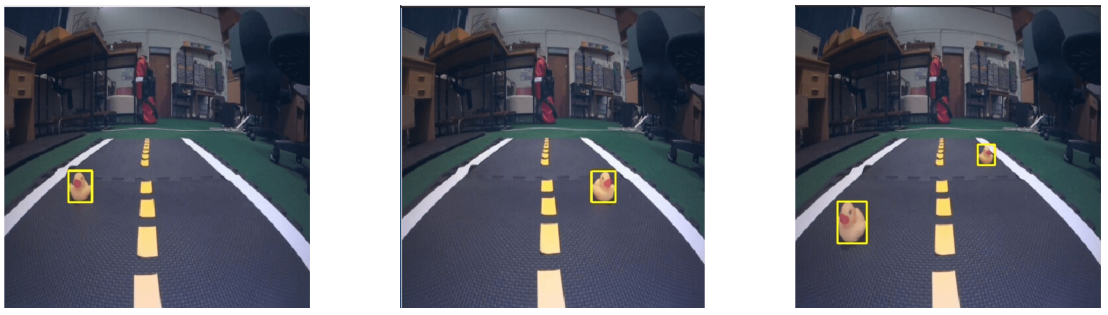
Initialize $l_{max} = -\infty$, $l_{min} = \infty$, $r_{max} = -\infty$, $r_{min} = \infty$
for each valid detection **do**
    $l_{max} = \max(A_l, l_{max})$
    $l_{min} = \max(A_l, l_{max})$
    $r_{max} = \max(A_r, r_{max})$
    $r_{min} = \max(A_r, r_{max})$
    rescale A_l and A_r based on new max and min values
end for

PWM commands are computed using Eq. 5.2 where c is a constant and k is a gain.

$$P_s = c + A_s k \quad s \in (l, r) \tag{5.2}$$

5.5 Braintberg Controller Validation

The Braitenberg controller was validated with three scenarios in Figure 5.7. In each case, the duckiebot travels in a straight line, and duckies are placed in front of the duckiebot with c and k set to 0.3 and 1.

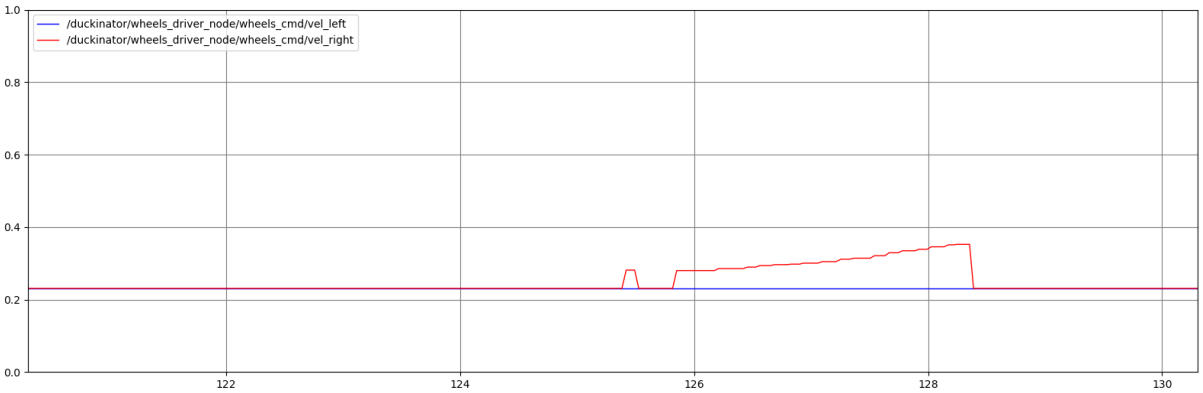


(a) Singel duckie left (b) Single duckie right (c) Double trouble

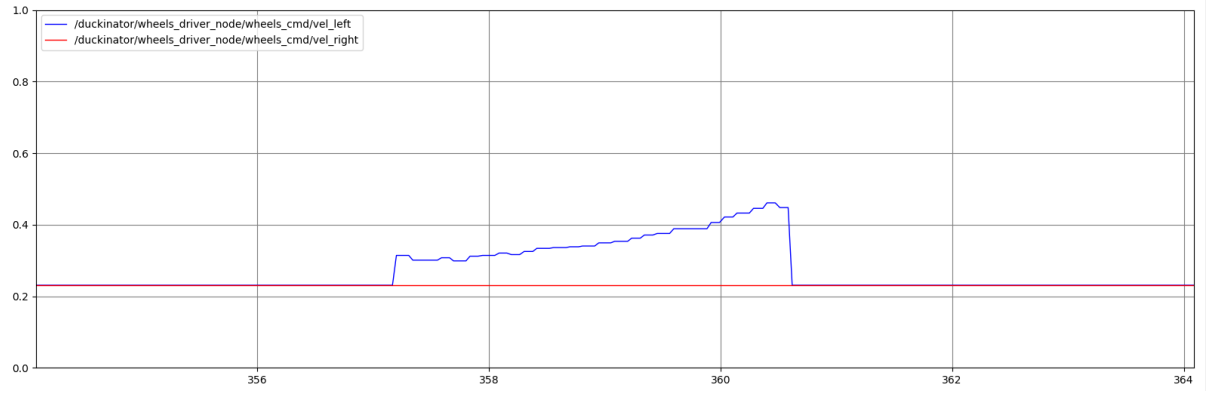
Figure 5.7: Three validation scenarios for the Braitenberg controller showing detection maps from the duckiebot and YOLOv5 model.

Figure 5.8 shows the PWM wheel command published to the `wheel_cmd_node` during validation. The Braitenberg A, or Fear, vehicle is used for the validation and explains

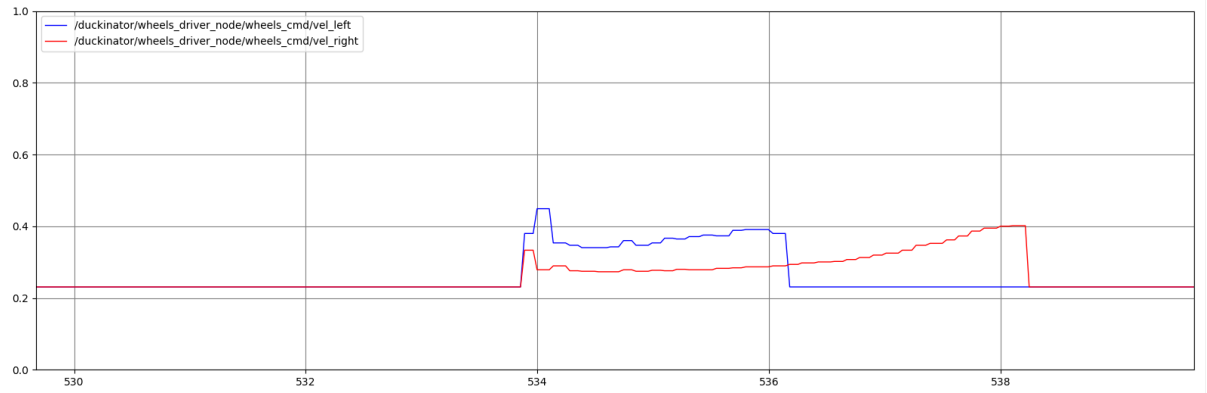
the wheel commands in the Figure 5.8. For all the cases, wheel commands linearly increase as the duckiebot approaches and passes the duckie, consistent with the expected motion for the Fear Braitenberg vehicle.



(a) Single duckie to the right of the duckiebot



(b) Single duckie to the left of the duckiebot



(c) Two duckies on either side of the duckiebot

Figure 5.8: PWM wheel commands for validation scenarios. In this case, the duckiebot was limited to moving forward only. Therefore, PWM commands on the vertical axis are constrained between 0 and 1. The horizontal axis represents time in seconds.

An important validation output is the inference time of the YOLOv5 model and ROS agent. Table 5.2 shows the timing of the YOLOv5 model and Braitenbrug controller during validation. The main limitation of the current implementation relates to the inference time of the model. The ROS agent is implemented using the Duckietown Learning Experience (DT-LX) Docker image template. This image allows for faster development as a simulation environment is integrated, and the developer can seamlessly transfer between the simulation environment and duckiebot. However, this environment runs the ROS master on the user's computer and the duckiebot interface nodes on the duckiebot. As the computer used did not have a dedicated GPU, inference times were impacted. The duckiebot is equipped with a GPU. However, creating a Docker image was non-trivial given the time constraints of this report and is recommended as a future improvemnet in the next chapter.

Table 5.2: Timing data, in seconds, for validation results

	Minimum	**Average**	**Maximum**
YOLOv5 model	0.039	0.0505	0.104
Total	0.049	0.060	0.120

5.6 Conclusion

The implementation results of the YOLOv5 model demonstrate that the model is able to detect duckies for most cases in the Duckietown environment, with the exception being crowded duckies. The inference time of the YOLOv5 model is relatively large for real-time applications. However, this is a flaw of the implementation on the duckiebot rather than the YOLOv5 model.

As mentioned, the Braitenberg controller was implemented to validate the YOLO model and weight matrices and gain terms were only tuned for simple scenarios. The controller fails when the scene is more complex than validation scenario 3 in Figure 5.7. The next chapter presents ways in which the Braitenberg controller can be extended to handle more complex behaviours.

CHAPTER 6

RECOMMENDATIONS AND PROPOSALS

A Discussion

> We choose our next world through what we learn in this one. Learn nothing, and the next world is the same as this one.
>
> Richard Bach, Jonathan Livingston Seagull

This section presents three possible trajectories for future work: extending the Bratienberg controller, Autonomous Vehicle Architectures (AVAs) and Depth estimation. Future work is presented as projects with nominal aims and objectives. This section summarises the key findings and promising projects from an extensive validity analysis in Appendix E. The intention is that this will form foundational material for any future work presented. It is worth noting that the proposed future work emerged from a literature review. While no detailed implementation plan has been developed or tested, they represent promising areas for further exploration.

6.1 Braitenberg Vehicles

The Braitenberg controller implemented in this report is foundational and has many exciting possibilities for future work. The controller fails to navigate environments with more than two duckies reliably. This is partly due to the weight matrices $\boldsymbol{W_{l,r}}$ not being optimally tuned but also due to the shape of the matrices. The simplest shape was used in this report to validate the YOLOv5 model, but other shapes warrant investigation. Consider possible modifications shown in Figures 6.1, 6.2 and 6.3 that could improve the Braitenberg controller.

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
1	1	1	0	0	0
1	1	1	0	0	0
1	1	1	0	0	0

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	1	1	1
0	0	0	1	1	1
0	0	0	1	1	1

Figure 6.1: The first modification to the weight matrices is to only account for the bottom half regions of the image.

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
1	2	3	0	0	0
1	2	3	0	0	0
1	2	3	0	0	0

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	3	2	1
0	0	0	3	2	1
0	0	0	3	2	1

Figure 6.2: A second modification prioritises values closer to the centre.

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	3	0	0	0
0	2	3	0	0	0
1	2	3	0	0	0

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	3	0	0
0	0	0	3	2	0
0	0	0	3	2	1

Figure 6.3: The third modification considers a triangular region in front of the duckiebot.

Figure 6.1 attempts to solve the issue experienced where motor activation is based on detections relatively far from the duckiebot. Figure 6.2 and 6.3 show two further possi-

ble weight matrices that place the importance of objects detected in the middle of the duckiebot to greater importance. By tuning these matrices, achieving more complex behaviour, such as object avoidance in an environment where many duckies are placed in random positions, is possible. Only the Fear vehicle was considered here, but the other three vehicles are capable of other useful behaviour patterns such as target seeking or line following. Another exciting avenue is to use non-linear weight matrices to produce complex trajectories, such as figures of 8s, as mentioned in Section 3.2 and repeated below for convenience.

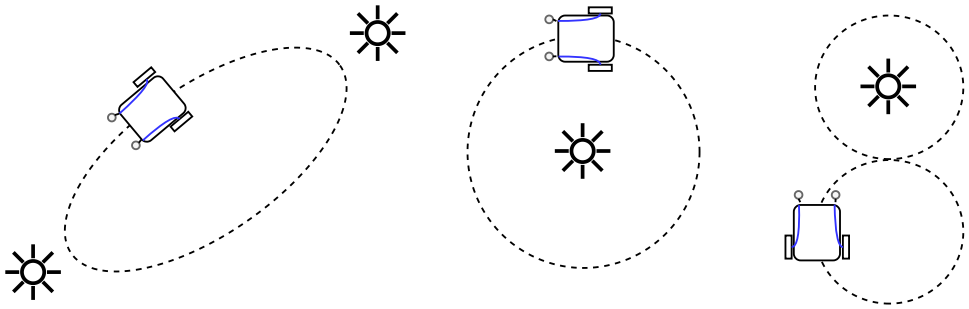


Figure 6.4: Complex motions using the same sensorimotor connections with non-linear sensor activation [39]

Table 6.1: Braitenberg project Aim and Objectives

Aim
To demonstrate complex Braitenberg behaviour by partitioning a YOLOv5 object detection model's prediction map into left and right regions.
Objectives
(a) Implement a stand-alone ROS-compatible Docker project for the YOLOv5 model and improved Braitenberg controller.
(b) Generate a Matlab duckiebot model for tuning the weight matrices. Tuning the duckiebot through trial and error is not recommended, as each change requires the Docker image to be rebuilt, which can take several minutes. A simple Matlab model of the duckiebot would be useful for finding tentative weight matrices that can be fine-tuned.
(c) Demonstrate the ability of the controller to navigate complex environments with many duckies.
(d) Extend the controller by implementing other behaviour patterns such as line following, target seeking or non-linear behaviour.

6.2 Autonomous Vehicle Architectures

AV development has adopted three AVAs: cybernetics (perception, localisation, planning and control), vision-based ML and End-to-End architectures. Appendix E.1 discusses these three architectures in depth. Cybernetics represents the initial phase of AV development and has largely been replaced by data-driven approaches that use vision or camera-based systems. Vision-based architectures have been abetted by the advancement in ML and use CNNs for object detection, mapping environments and planning. These vision-based systems can be seen as extensions of cybernetics that combine perception and localisation. End-to-end architectures are a complete departure from the cybernetics framework and use Reinforcement Learning (RL) techniques to train an agent that streamlines the entire pipeline by using images as input and control commands as outputs. The advantage of End-to-End methods is that the agent does not require prior knowledge of the environment and is trained self-supervised in a simulation environment. Currently, agents cannot completely transfer from simulation to the real world, limiting them to non-safety critical tasks, but are predicted to be solutions to the FSD problem, warranting research efforts. [53].

The Duckietown environment provides an opportunity to investigate RL algorithms as the Duckietown simulation environment already exists. Additionally, Duckietown has standardised rules and dimensions which aid in defining reward functions. Duckietown held an AI Driving Olympic (AIDO) comprising various challenges that require the navigation of Duckietown. Many of the solutions made use of End-to-End architectures and are published. Kalapos et al. implement an RL agent trained purely in the Duckietown-gym that can generalise to the physical duckiebot placing first in AIDO 5 and can be used as a baseline [37]. Table 6.3 presents a project proposal based on the Kalapos et al implementation.

Table 6.2: Reinforcement Learning project Aim and objectives

Aim
Implement an End-to-End agent trained via RL methods in the Duckietown-gym Simulation environment to follow a lane in the simulator environment.
Objectives
(a) Set up the Duckietown-gym simulation environment for manual agent control.
(b) Investigate RL algorithms suitable for the task of lane following.
(c) Define RL reward functions and policy network architecture.
(d) Train RL agent in the Duckietown-gym simulator.

6.3 Depth Estimation

As mentioned in this report, the most basic function of an AV vehicle is to sense its environment. This requires detecting objects in the scene and their spatial configurations or depth in the scene. Depth estimation techniques such as stereovision and laser-based depth maps provide suitable solutions for most applications explained in Appendix E.2.3. However, neither of these methods solves the issue of depth estimation using a single camera. Formally, the problem is called *monocular depth estimation* and falls victim to the *curse of dimensionality* where 3D information is trying to be recovered from a 2D image. Traditional monocular depth techniques attempt to 'recreate' stereovision by considering two frames from an image stream known as Structure-from-Motion (SFM) [54]. These techniques usually make strong assumptions about the scene, thus not robust for AV applications.

CNNs are attractive tools that have been used to improve SFM [55]. More useful for AV applications, supervised and unsupervised CNN models have been used to generate depth maps that can replace LiDAR depth maps [56], [57]. To the best of this report's knowledge, monocular depth estimation has not been implemented on the duckiebot. *Monodepth2* [56] appears to be the most attractive solution and can be readily implemented using the `Tensorflow` python library [1].

[1] See this repository

Table 6.3: Depth Estimation project Aim and objectives

Aim
Implement a monocular depth estimation model, nominally Monodepth2, on the duckiebot to infer the distance of objects in the Duckietown environment.
Objectives
(a) Verify, by means of a literature review, that Monodepth2 is the most appropriate monocular depth estimation model for the duckiebot.
(b) Validate the pre-trained Monodepth2, or equivalent, model in the Duckietown environment.
(c) Propose a method to integrate YOLOv5 and Depth estimation model.
(d) Implement and validate the integration of the two models

CHAPTER 7

CONCLUDING REMARKS

> To succeed, jump as quickly at opportunities as you do at conclusions.
>
> Benjamin Franklin

This report presents an investigation of the Duckietown platform for AV development in the frame of the MEC4128S course. The primary aim of this investigation is to establish the groundwork for future work on the platform. To achieve this aim, the report identified object detection as a foundational component of any AV system. The report limited itself to only ML, specifically CNN-based object detection models, given that the Duckietown platform is purpose-built for ML applications. With the motivation of serving as a valuable resource for undergraduate students, Chapter 2 is dedicated to providing a background on CNNs, an essential concept not covered in the curriculum.

The pursuit of object detection began with the surveillance of literature in Chapter 3 to identify a suitable object detection model. The chapter proposes the YOLOv5 object detection model due to its suitability and widespread implementation for real-time object detection. Chapter 4 presents the YOLOv5 model, dataset generation and training results. The model's training employed a combination of real and synthetic images relevant to Duckietown. The approach of supplementing the dataset with synthetic images produced training results that not only competed with larger YOLOv5 models but surpassed them in some metrics, all while employing a shorter number of training epochs.

Chapter 5 dealt with implementing the YOLOv5 model on the duckiebot. To validate the YOLOv5 model within the Duckietown environment, a simple Braitenberg controller was employed as a rudimentary obstacle avoidance agent. The YOLOv5 model successfully detected duckies in most scenarios within the Duckietown environment. The main shortfall was the inference time, which can be readily improved by creating a stand-alone

ROS-compatible Docker image instead of relying on the `DT-LX` image. However, the overall success of the implementation confirms the viability of the Duckietown platform within the scope of the MEC4128S.

Chapter 6 presents three possible trajectories for future work, namely, extending the Bratienberg controller, End-to-End RL and Depth Estimation. The wealth of opportunities available for future work reinforces the report's success in establishing a groundwork for future work on the platform.

Bibliography

[1] Gourav Bathla, Kishor Bhadane, Rahul Kumar Singh, Rajneesh Kumar, Rajanikanth Aluvalu, Rajalakshmi Krishnamurthi, Adarsh Kumar, R. N Thakur, and Shakila Basheer. Autonomous vehicles and intelligent automation: Applications, challenges, and opportunities, Jun 2022.

[2] Society of Automotive Engineers. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles.* SAE International, 2018.

[3] Eamonn Keogh and Abdullah Mueen. *Curse of Dimensionality*, pages 314–315. Springer US, Boston, MA, 2017.

[4] Yann LeCun, Koray Kavukcuoglu, and Clement Farabet. Convolutional networks and applications in vision. In *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pages 253–256, 2010.

[5] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. *Insights into imaging*, 9:611–629, 2018.

[6] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, Cambridge, MA, USA, 2016. http://www.deeplearningbook.org.

[7] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015.

[8] Jiale Cao, Yanwei Pang, Jin Xie, Fahad Shahbaz Khan, and Ling Shao. From handcrafted to deep features for pedestrian detection: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9):4913–4934, sep 2022.

[9] David G. Lowe. Distinctive image features from scale-invariant keypoints - international journal of computer vision, 2004.

[10] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1, 2005.

[11] Joseph Walsh, Niall O' Mahony, Sean Campbell, Anderson Carvalho, Lenka Krpalkova, Gustavo Velasco-Hernandez, Suman Harapanahalli, and Daniel Riordan. Deep learning vs. traditional computer vision. 04 2019.

[12] Min Young Chang, Suzie Petryk, and Joe Nechleba. Object detection and imitation learning in duckietown - minyoung, 2019.

[13] Kichun Jo, Junsoo Kim, Dongchul Kim, Chulhoon Jang, and Myoungho Sunwoo. Development of autonomous car—part ii: A case study on the implementation of an autonomous driving system based on distributed architecture. *IEEE Transactions on Industrial Electronics*, 62(8):5119–5132, 2015.

[14] Yeonsik Kang, Chiwon Roh, Seung-Beum Suh, and Bongsob Song. A lidar-based decision-making method for road boundary detection using multiple kalman filters. *IEEE Transactions on Industrial Electronics*, 59(11):4360–4368, 2012.

[15] Huijing Zhao, Jie Sha, Yipu Zhao, Junqiang Xi, Jinshi Cui, Hongbin Zha, and Ryosuke Shibasaki. Detection and tracking of moving objects at intersections using a network of laser scanners. *IEEE Transactions on Intelligent Transportation Systems*, 13(2):655–670, 2012.

[16] Yonghui Hu, Wei Zhao, and Long Wang. Vision-based target tracking and collision avoidance for two autonomous robotic fish. *IEEE Transactions on Industrial Electronics*, 56(5):1401–1410, 2009.

[17] Bhaskar Barua, Clarence Gomes, Shubham Baghe, and Jignesh Sisodia. A self-driving car implementation using computer vision for detection and navigation. In *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, pages 271–274, 2019.

[18] Kohei Arai and Supriya Kapoor, editors. *Advances in Computer Vision*. Springer International Publishing, 2020.

[19] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation, 2014.

[20] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[21] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2016.

[22] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2018.

[23] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: Single shot MultiBox detector. In *Computer Vision – ECCV 2016*, pages 21–37. Springer International Publishing, 2016.

[24] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[25] Qijie Zhao, Tao Sheng, Yongtao Wang, Feng Ni, and Ling Cai. Cfenet: An accurate and efficient single-shot object detector for autonomous driving, 2018.

[26] Tesla ai day. Online event, August 2022.

[27] Michael Yuhas and Arvind Easwaran. Demo abstract: Real-time out-of-distribution detection on a mobile robot, 2022.

[28] Zhi Qiu, Zuoxi Zhao, Shaoji Chen, Junyuan Zeng, Yuan Huang, and Borui Xiang. Application of an improved yolov5 algorithm in real-time detection of foreign objects by ground penetrating radar. *Remote Sensing*, 14:1895, 04 2022.

[29] J Terven and D Cordova-Esparza. A comprehensive review of yolo: From yolov1 and beyond. arxiv 2023. *arXiv preprint arXiv:2304.00501*.

[30] Toan-Khoa Nguyen, Lien Vu, Viet Vu, Tien-Dat Hoang, Shu-Hao Liang, and Minh-Quang Tran. Analysis of object detection models on duckietown robot based on yolov5 architectures. 4:17–12, 03 2022.

[31] Renjie Xu, Haifeng Lin, Kangjie Lu, Lin Cao, and Yunfei Liu. A forest fire detection system based on ensemble learning. *Forests*, 12(2), 2021.

[32] Pappu Kumar Yadav, J. Alex Thomasson, Stephen W. Searcy, Robert G. Hardin, Ulisses Braga-Neto, Sorin C. Popescu, Daniel E. Martin, Roberto Rodriguez, Karem Meza, Juan Enciso, Jorge Solórzano Diaz, and Tianyi Wang. Assessing the performance of yolov5 algorithm for detecting volunteer cotton plants in corn fields at three different growth stages. *Artificial Intelligence in Agriculture*, 6:292–303, 2022.

[33] Vikrant Shahane, Hrushikesh Jadhav, Mihir Sansare, and Prathmesh Gunjgur. A self-driving car platform using raspberry pi and arduino. In *2022 6th International Conference On Computing, Communication, Control And Automation (ICCUBEA*, pages 1–6, 2022.

[34] Chi Zhang, Zhong Yang, Luwei Liao, Yulong You, Yaoyu Sui, and Tang Zhu. Rpeod: A real-time pose estimation and object detection system for aerial robot target tracking. *Machines*, 10(3), 2022.

[35] Maxime Chevalier-Boisvert, Florian Golemo, Yanjun Cao, Bhairav Mehta, and Liam Paull. Duckietown environments for openai gym. https://github.com/duckietown/gym-duckietown, 2018.

[36] Greet Baldewijns, Glen Debard, Gert Mertes, Bart Vanrumste, and Tom Croonenborghs. Bridging the gap between real-life data and simulated data by providing a highly realistic fall dataset for evaluating camera-based fall detection algorithms. *Healthcare technology letters*, 3(1):6–11, 2016.

[37] András Kalapos, Csaba Gór, Róbert Moni, and István Harmati. Vision-based reinforcement learning for lane-tracking control. *Acta IMEKO*, 10(3):7–14, 2021.

[38] Clinton H Hansen, Robert G Endres, and Ned S Wingreen. Chemotaxis in escherichia coli: a molecular model for robust precise adaptation. *PLoS computational biology*, 4(1):e1, 2008.

[39] Valentino Braitenberg. Vehicles: Experiments in synthetic psychology. *Philosophical Review*, 95(1):137–139, 1986.

[40] Achim Lilienthal and Tom Duckett. Experimental analysis of gas-sensitive braitenberg vehicles. *Advanced Robotics*, 18:817–834, 12 2004.

[41] Dieter Vanderelst, Marc W. Holderied, and Herbert Peremans. Sensorimotor model of obstacle avoidance in echolocating bats. *PLOS Computational Biology*, 11(10):1–31, 10 2015.

[42] Dongdong Xu, Xingnan Zhang, Zhangqing Zhu, Chunlin Chen, Pei Yang, et al. Behavior-based formation control of swarm robots. *mathematical Problems in Engineering*, 2014, 2014.

[43] Pilsung Kang and Sungmin Lim. A taste of scientific computing on the gpu-accelerated edge device. *IEEE Access*, 8:208337–208347, 2020.

[44] Chien-Yao Wang, Hong-Yuan Mark Liao, I-Hau Yeh, Yueh-Hua Wu, Ping-Yang Chen, and Jun-Wei Hsieh. Cspnet: A new backbone that can enhance learning capability of cnn, 2019.

[45] Zheng-De Zhang, Meng-Lu Tan, Zhi-Cai Lan, Hai-Chun Liu, Ling Pei, and Wen-Xian Yu. Cdnet: a real-time and robust crosswalk detection network on jetson nano based on yolov5. *Neural Computing and Applications*, 34:1–12, 07 2022.

[46] Glenn Jocher, Alex Stoken, Jirka Borovec, NanoCode012, ChristopherSTAN, Liu Changyu, Laughing, tkianai, Adam Hogan, lorenzomammana, yxNONG, AlexWang1900, Laurentiu Diaconu, Marc, wanghaoyang0106, ml5ah, Doug, Francisco Ingham, Frederik, Guilhen, Hatovix, Jake Poznanski, Jiacong Fang, Lijun Yu, changyu98, Mingyu Wang, Naman Gupta, Osama Akhtar, PetrDvoracek, and Prashant Rai. ultralytics/yolov5: v3.1 - Bug Fixes and Performance Improvements, October 2020.

[47] Liam Paull, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, Changhyun Choi, Jeff Dusek, Yajun Fang, Daniel Hoehener, Shih-Yuan Liu, Michael Novitzky, Igor Franzoni Okuyama, Jason Pazis, Guy Rosman, Valerio Varricchio, Hsueh-Cheng Wang, Dmitry Yershov, Hang Zhao, Michael Benjamin, Christopher Carr, Maria Zuber, Sertac Karaman, Emilio Frazzoli, Domitilla Del Vecchio, Daniela Rus, Jonathan How, John Leonard, and Andrea Censi. Duckietown: An open, inexpensive and flexible platform for autonomy education and research. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1497–1504, 2017.

[48] Dániel Horváth, Gábor Erdős, Zoltán Istenes, Tomáš Horváth, and Sándor Földi. Object detection using sim2real domain randomization for robotic applications, 08 2022.

[49] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks, 2018.

[50] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934, 2020.

[51] Hongyi Zhang, Moustapha Cissé, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *CoRR*, abs/1710.09412, 2017.

[52] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[53] Péter Almási, Róbert Moni, and Bálint Gyires-Tóth. Robust reinforcement learning-based autonomous driving agent for simulation and real world. *CoRR*, abs/2009.11212, 2020.

[54] Richard Szeliski. *Structure from motion*, pages 303–334. Springer London, London, 2011.

[55] Jure Zbontar and Yann LeCun. Stereo matching by training a convolutional neural network to compare image patches. *CoRR*, abs/1510.05970, 2015.

[56] Clément Godard, Oisin Mac Aodha, and Gabriel J. Brostow. Digging into self-supervised monocular depth estimation. *CoRR*, abs/1806.01260, 2018.

[57] Yury A. Davydov, Wen-Hui Chen, and Yu-Chen Lin. Monocular supervised metric distance estimation for autonomous driving applications. In *2022 22nd International Conference on Control, Automation and Systems (ICCAS)*, pages 342–347, 2022.

[58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[59] Sridhar Narayan. The generalized sigmoid activation function: Competitive supervised learning. *Information Sciences*, 99(1):69–82, 1997.

[60] Kunihiko Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, 20(3-4):121–136, 1975.

[61] Abien Fred Agarap. Deep learning using rectified linear units (relu), 2019.

[62] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3. Atlanta, GA, 2013.

[63] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[64] Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. Escaping from saddle points — online stochastic gradient for tensor decomposition, 2015.

[65] Fatima Zohra El hlouli, Jamal Riffi, Mohamed Adnane Mahraz, Ali El Yahyaouy, and Hamid Tairi. Credit card fraud detection based on multilayer perceptron and extreme learning machine architectures. In *2020 International Conference on Intelligent Systems and Computer Vision (ISCV)*, pages 1–5, 2020.

[66] Nikolay Stanevski, Dimiter Tsvetkov, and MARGIN CLASSIFIER. Using support vector machine as a binary classifier. 01 2005.

[67] Yunchuan Kong and Tianwei Yu. A deep neural network model using random forest to extract feature representation for gene expression data classification. *Scientific reports*, 8(1):16477, 2018.

[68] Nader Mohamed, Jameela Al-Jaroodi, and Imad Jawhar. Middleware for robotics: A survey. pages 736 – 742, 10 2008.

[69] Molaletsa Namoshe, N S Tlale, C M Kumile, and G. Bright. Open middleware for robotics. In *2008 15th International Conference on Mechatronics and Machine Vision in Practice*, pages 189–194, 2008.

[70] Ayssam Elkady and Tarek Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012, 05 2012.

[71] Anis Koubâa et al. *Robot Operating System (ROS).*, volume 1. Springer, 2017.

[72] Rodney A Brooks. Intelligence without representation. *Artificial intelligence*, 47(1-3):139–159, 1991.

[73] Rolf Pfeifer and Josh Bongard. *How the body shapes the way we think: a new view of intelligence*. MIT press, 2006.

[74] Reinhold Behringer, Sundar Sundareswaran, Brian Gregory, Richard Elsley, Bob Addison, Wayne Guthmiller, Robert Daily, and David Bevly. The darpa grand challenge-development of an autonomous vehicle. In *IEEE Intelligent Vehicles Symposium, 2004*, pages 226–231. IEEE, 2004.

[75] Martin Buehler, Karl Iagnemma, and Sanjiv Singh. *The 2005 DARPA Grand Challenge: The Great Robot Race*. 01 2007.

[76] Yi-Liang Chen, Venkataraman Sundareswaran, Craig Anderson, Alberto Broggi, Paolo Grisleri, Pier Paolo Porta, Paolo Zani, and John Beck. Terramax: Team oshkosh urban robot. volume 56, pages 595–622, 01 2009.

[77] Pietro Cerri, Giacomo Soprani, Paolo Zani, Jaewoong Choi, Junyung Lee, Dongwook Kim, Kyongsu Yi, and Alberto Broggi. Computer vision at the hyundai autonomous challenge. pages 777–783, 10 2011.

[78] Kichun Jo, Minchae Lee, Dongchul Kim, Junsoo Kim, Chulhoon Jang, Euiyun Kim, Sangkwon Kim, Donghwi Lee, Changsup Kim, Seungki Kim, Kunsoo Huh, and Myoungho Sunwoo. Overall reviews of autonomous vehicle a1 - system architecture and algorithms. *IFAC Proceedings Volumes*, 46(10):114–119, 2013. 8th IFAC Symposium on Intelligent Autonomous Vehicles.

[79] Scott Drew Pendleton, Hans Andersen, Xinxin Du, Xiaotong Shen, Malika Meghjani, You Hong Eng, Daniela Rus, and Marcelo H. Ang. Perception, planning, control, and coordination for autonomous vehicles. *Machines*, 5(1), 2017.

[80] Xu Li, Wei Chen, and Chingyao Chan. A reliable multisensor fusion strategy for land vehicle positioning using low-cost sensors. *Proceedings of the Institution of Mechanical Engineers, Part D: Journal of Automobile Engineering*, 228(12):1375–1397, 2014.

[81] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. Designing network design spaces, 2020.

[82] Peide Wang. Research on comparison of lidar and camera in autonomous driving. *Journal of Physics: Conference Series*, 2093(1):012032, nov 2021.

[83] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning, 2016.

[84] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[85] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[86] Maximilian Jaritz, Raoul de Charette, Marin Toromanoff, Etienne Perot, and Fawzi Nashashibi. End-to-end race driving with deep reinforcement learning. *CoRR*, abs/1807.02371, 2018.

[87] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. *CoRR*, abs/1807.00412, 2018.

[88] Joris Dinneweth, Abderrahmane Boubezoul, René Mandiau, and Stéphane Espié. Multi-agent reinforcement learning for autonomous vehicles: A survey. *Autonomous Intelligent Systems*, 2(1):27, 2022.

[89] Eric Marchand, Hideaki Uchiyama, and Fabien Spindler. Pose estimation for augmented reality: A hands-on survey. *IEEE Transactions on Visualization and Computer Graphics*, 22(12):2633–2651, 2016.

[90] Yang Liu, Jie Jiang, Jiahao Sun, Liang Bai, and Qi Wang. A survey of depth estimation based on computer vision. In *2020 IEEE Fifth International Conference on Data Science in Cyberspace (DSC)*, pages 135–141, 2020.

[91] Yaowen Lv, Jinliang Feng, Zhaokun Li, Wei Liu, and Jingtai Cao. A new robust 2d camera calibration method using ransac. *Optik*, 126(24):4910–4915, 2015.

[92] David Marr. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. The MIT Press, 07 2010.

[93] Abbas shakeri, Behzad Moshiri, and Hossein Gharaee Garakani. Pedestrian detection using image fusion and stereo vision in autonomous vehicles. In *2018 9th International Symposium on Telecommunications (IST)*, pages 592–596, 2018.

[94] Kunsoo Huh, Jaehak Park, Junyeon Hwang, and Daegun Hong. A stereo vision-based obstacle detection system in vehicles. *Optics and Lasers in Engineering*, 46:168–178, 02 2008.

[95] Chen Fu, Christoph Mertz, and John M. Dolan. Lidar and monocular camera fusion: On-road depth completion for autonomous driving. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 273–278, 2019.

[96] Carlo Tomasi and Takeo Kanade. Detection and tracking of point. *Int J Comput Vis*, 9(137-154):3, 1991.

[97] Kang Xue, Patricio A. Vela, Yue Liu, and Yongtian Wang. A modified klt multiple objects tracking framework based on global segmentation and adaptive template. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, pages 3561–3564, 2012.

[98] Shiyu Song, Manmohan Chandraker, and Clark Guest. High accuracy monocular sfm and scale correction for autonomous driving. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38:1–1, 10 2015.

[99] Davide Scaramuzza, Friedrich Fraundorfer, Marc Pollefeys, and Roland Siegwart. Absolute scale in structure from motion from a single vehicle mounted camera by exploiting nonholonomic constraints. In *2009 IEEE 12th International Conference on Computer Vision*, pages 1413–1419, 2009.

[100] Ashutosh Saxena, Jamie Schulte, and Andrew Y. Ng. Depth estimation using monocular and stereo cues. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, page 2197–2203, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

[101] Iro Laina, Christian Rupprecht, Vasileios Belagiannis, Federico Tombari, and Nassir Navab. Deeper depth prediction with fully convolutional residual networks, 2016.

[102] Tuan Anh Le, Atilim Gunes Baydin, Robert Zinkov, and Frank Wood. Using synthetic data to train neural networks is model-based reasoning, 2017.

[103] Amir Atapour-Abarghouei and Toby P. Breckon. Real-time monocular depth estimation using synthetic data with domain adaptation via image style transfer. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2800–2810, 2018.

[104] Dmitry Ulyanov, Vadim Lebedev, Andrea Vedaldi, and Victor Lempitsky. Texture networks: Feed-forward synthesis of textures and stylized images, 2016.

[105] Sabir Hossain and Xianke Lin. Efficient stereo depth estimation for pseudo lidar: A self-supervised approach based on multi-input resnet encoder, 2022.

[106] Matteo Poggi, Filippo Aleotti, Fabio Tosi, and Stefano Mattoccia. Towards real-time unsupervised monocular depth estimation on CPU. *CoRR*, abs/1806.11430, 2018.

Appendix A

Repository Location

All source code for this project can be found in two github repositories. The `DT_project_60` repository can be found at the link below and contains the source code for generating the dataset and training the YOLOv5 model:

https://github.com/Dinoclaro/DT_project_60

The `DT_objdet` repository is the Duckietown-compliant Docker image to run the ROS agent containging the YOLOv5 model and Braitenberg controller:

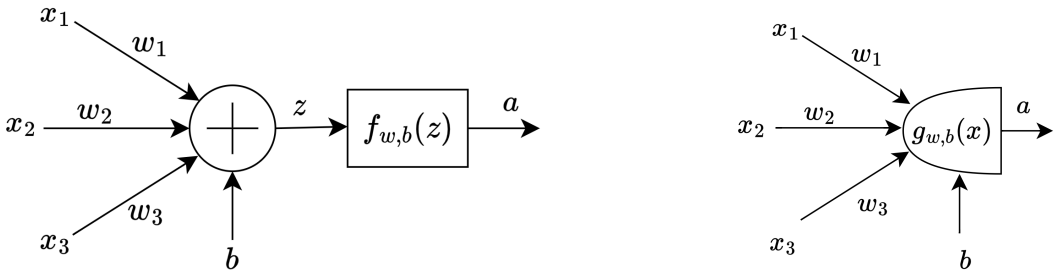
https://github.com/Dinoclaro/DT_objdet

Consult the README.md files in the respective direectories for urther infromation in making use of these files.

Appendix B

Neural Networks

Neural Networks (NNs) are a class of ML algorithms inspired by the human brain's biological neural networks first introduced by Rosenblatt in 1958 [58]. NNs are composed of layers of interconnected artificial neurons Rosenblatt called *Perceptrons*, which attempt to emulate the brain's workings. The neuron is simply a function that maps several inputs to an output depicted in Figure B.1.



(a) Artificial neuron (b) Simplified Artificial neuron

Figure B.1: Artificial Neuron as the building block of a Neural Network. (a) shows the complete neuron with the summing and non-linear function shown separately. (b) is a more concise representation of the neuron combining the summation and activation operations

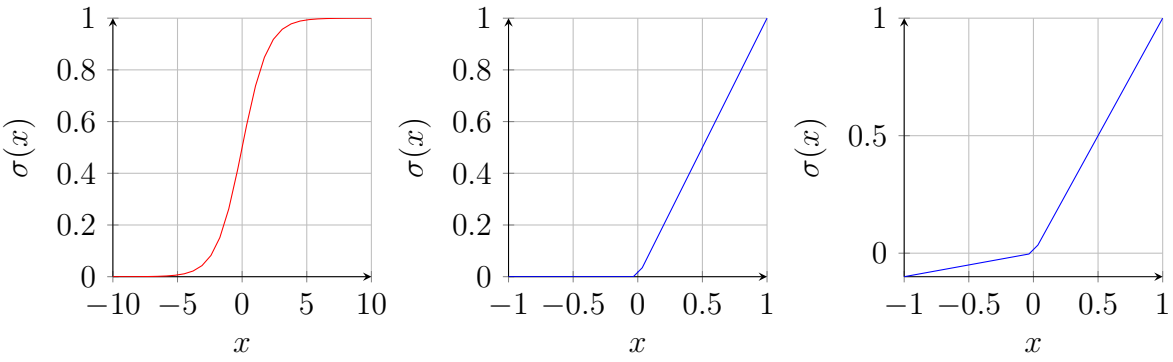
$$z = x_1 w_1 + x_2 w_2 \ldots + x_n w_n + b = \sum_{i=1}^{n} x_i w_i + b \tag{B.1}$$

$$a = f_{w,b}(z) = f_{w,b}\left(\sum_{i=1}^{n} x_i w_i + b\right) \tag{B.2}$$

$$a = f_{w,b}(x) = f_{\boldsymbol{w},b}(\boldsymbol{x} \cdot \boldsymbol{w} + b) \tag{B.3}$$

Input data, $\boldsymbol{x}$, is multiplied by a weight vector, $\boldsymbol{w}$, and added to a bias, b. The bias

term models offsets or baseline values in data. The output, $\boldsymbol{z}$, is then passed through a non-linear, differentiable activation function, $f_{\mathbf{w},\mathbf{b}}$ that determines the output of the neuron. This activation function can be thought of as a gate. Figure B.2 shows common activation functions. Non-linearity is important as it allows the model to learn complex patterns in data. A linear function, resulting in linear combinations of its inputs, limits the model to only learning linear patterns.



(a) Sigmoid (b) ReLU (c) Leaky ReLU

Figure B.2: Common activation functions

The sigmoid function [59] was the first activation function used, but suffers from the *vanishing gradient problem*. The Rectified Linear Unit (ReLU) function [60] is defined as $f(x) = max(0, z)$ and reduces vanishing gradients by allowing activation values larger than one. The piecewise nature of the function ensures that ReLU is non-linear. ReLU can suffer *dying-ReLU* where several inputs only output a value of zero [61]. Leaky ReLU [62] is a modified version of the ReLU function; adding a slight negative slope solves the dying-ReLU problem. ReLU and its modifications are the most common activation functions in modern networks as they are computationally efficient and do not suffer from the vanishing gradient problem. Typically, the last activation layer is not ReLU and depends on the specific application. Table B.1 describes common last-layer activation functions for various applications. Multiclass single-class is the problem of predicting a single class from a list of possible classes, and multiclass multiclass predicts the probability of an object being in each class.

Table B.1: Commonly applied final layer activation functions [5]

Task	**Final Layer Activation Function**
Binary classification	Sigmoid
Multiclass single-class classification	Softmax
Multiclass multiclass classification	Sigmoid
Regression to continuous values	Identity

Figure B.3 shows the most basic NN, the Multi-layer perceptron (MLP) network, where every neuron in one layer is connected to every neuron in the next layer. The layers between the input and output layers are known as hidden layers.

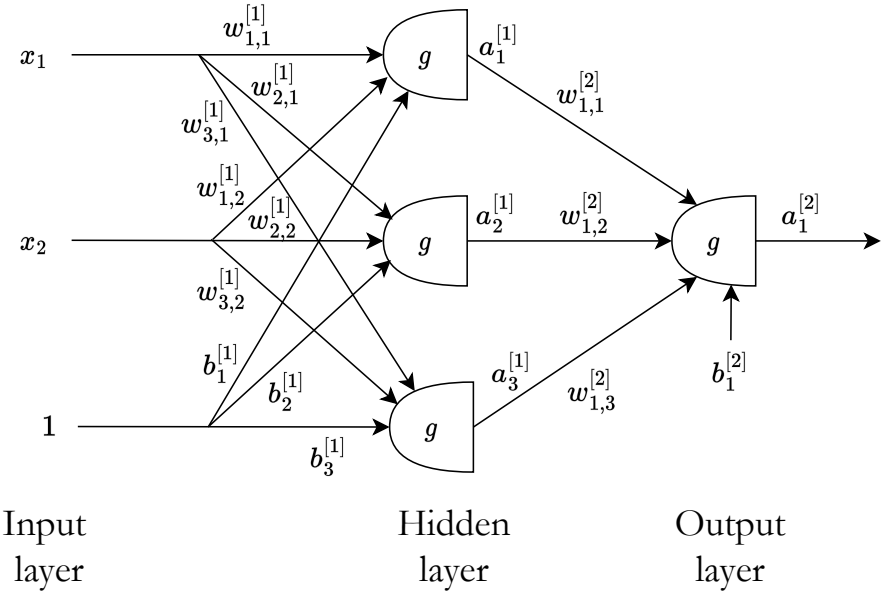


Figure B.3: Multi-layer perceptron Network

Figure B.4 shows a toy example where the model has been trained to predict whether a student will pass or fail based on the number of hours studied, x_1, and the number of hours slept x_2. The output layers provide the probability of passing. In this case, the blue neuron has been trained to learn a particular trend in the data. The model has learned that if only the blue neuron is activated in the first layer, the student will pass, and the green neuron is activated. Here, activation refers to the combination of weights, inputs and biases resulting in a large output. In this way, neurons and their associated weights can be considered as *feature extractors*. Composing many neurons allows for extracting more complicated features, giving rise to the power of NNs in data recognition tasks.

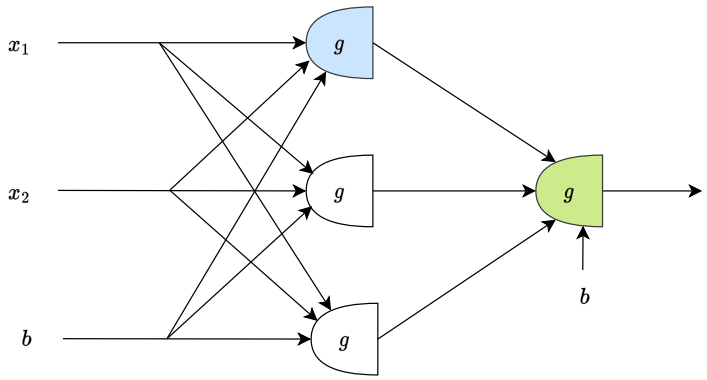


Figure B.4: Multi-layer perceptron Network example

The parameters ($\boldsymbol{w}$ and $\boldsymbol{b}$) of the neurons are adjusted during training. Training is the process of optimising parameters by minimising a loss function given some labelled data through a process known as backpropagation. Backpropagation, introduced by Rumelhart et al. of [63], allows the NN to learn from a dataset. It takes advantage of the differentiability and composability of the network to calculate the loss gradient with respect to each parameter via the chain rule. The algorithm below outlines the backpropagation to update the parameters of the MLP network shown above.

Algorithm 2 Back propagation Algorithm

1: Initialize neural network weights and biases randomly
2: **for** each training iteration **do**
3: Forward Pass:
4: Compute the predicted output $\hat{y}$ by propagating input through the network
5: Compute the loss $\mathcal{L}$ between $\hat{y}$ and the true labels y
6: Backward Pass:
7: Compute the gradient of the loss with respect to the layer $a_k^{[L-1]}$ for all N_L neurons in the layer, L:
8: $\frac{\partial \mathcal{L}}{\partial a_k^{[L-1]}} = \sum_{j=1}^{N_L} \frac{\partial z_j^{[L]}}{\partial a_k^{[L-1]}} \cdot \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \cdot \frac{\partial \mathcal{L}}{\partial a_j^{[L]}}$
9: Compute the gradient of weights and biases for each neuron, j, connecting to k neurons in the next layer:
10: $\frac{\partial \mathcal{L}}{\partial w_{j,k}^{[L-1]}} = \frac{\partial z_j^{[L]}}{\partial w_{j,k}^{[L-1]}} \cdot \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \cdot \frac{\partial \mathcal{L}}{\partial a_j^{[L]}}$
11: $\frac{\partial \mathcal{L}}{\partial b_j^{[L-1]}} = \frac{\partial z_j^{[L]}}{\partial b_j^{[L-1]}} \cdot \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} \cdot \frac{\partial \mathcal{L}}{\partial a_j^{[L]}}$
12: Update output layer weights and biases:
13: $W_{j,k} \leftarrow W_{j,k} + \alpha \cdot \frac{\partial \mathcal{L}}{\partial w_{j,k}}$
14: $b_{j,k} \leftarrow b_{j,k} + \alpha \cdot \frac{\partial \mathcal{L}}{\partial w_{j,k}}$
15: **end for**

In this light, NNs are optimisation problems where a training dataset is used to determine the optimum weights for each parameter in the network. In particular, it is the optimisation of a non-convex loss surface. Figure B.5 shows a 3-dimensional (3D) non-convex surface. In reality, these surfaces are of the same dimension as the number of parameters (1.9 Million for the YOLOv5 model proposed in this report). The difficulty with solving these problems, which can be appreciated in the surface shown, is converging to a local minimum or saddle point [1] instead of the global minimum.

[1] Saddle points are critical points on the loss function surface where the gradient is zero, but the point is neither a local minimum nor a local maximum but instead, a point of inflection.

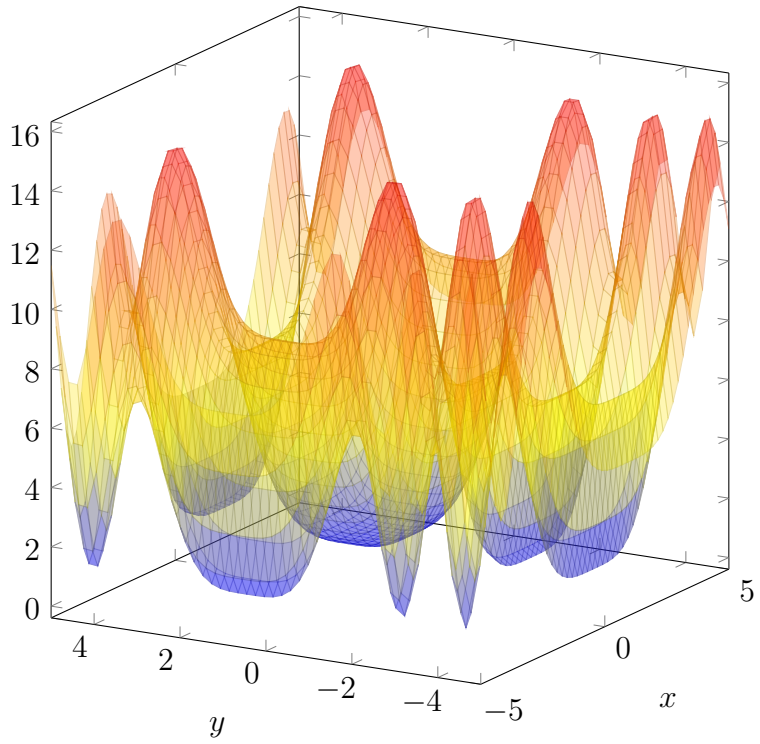


Figure B.5: 3 Dimensional non-convex function

The optimiser used to update the backpropagation algorithm is known as the Gradient Descent (GD) algorithm and adjusts the parameters proportionally to the loss gradient. α is a hyperparameter that determines the step size of the GD algorithm and has a significant effect on the convergence rate. GD examines the error and updates weights over the *entire* dataset. Averaging over the entire dataset results in a potential loss of information and can lead to saddle points [49]. Stochastic GD (SGD) represents the other extreme, where the parameters are updated on a single, *random* sample. This introduces noise into the gradient direction, which is favourable in non-convex optimisation and assists in escaping saddle points [64]. However, this is extremely inefficient and requires many training epochs (complete passes through the dataset) before convergence. Mini-batch GD is a variation that divides the dataset into batches and updates the parameters after each batch. This is the most common approach as it balances the noise of SGD and the efficiency of GD. Other optimisers, such as Adaptive Moment Estimation (ADAM), adaptively change learning rates and store an exponentially decaying average of past gradients, which helps overcome oscillations of noisy gradients. Significant research is conducted to find the most effective optimiser for a given problem [49].

MLPs are a single subset of machine learning algorithms that are particularly useful for solving sequential tabular data and have been used for fraud detection [65]. Other algorithms include Support Vector Machines (SVM) used for binary classifications [66] and Random Forests used in extracting features in gene expression data [67].

Appendix C

YOLOv5 Model

C.1 Hyperparameters

Contents of the `hyp.scratch-med.yaml` file:

```
# YOLOv5 by Ultralytics, AGPL-3.0 license
# Hyperparameters for medium-augmentation

lr0: 0.01  # initial learning rate (SGD=1E-2, Adam=1E-3)
lrf: 0.1  # final OneCycleLR learning rate (lr0 * lrf)
momentum: 0.937  # SGD momentum/Adam beta1
weight_decay: 0.0005  # optimizer weight decay 5e-4
warmup_epochs: 3.0  # warmup epochs (fractions ok)
warmup_momentum: 0.8  # warmup initial momentum
warmup_bias_lr: 0.1  # warmup initial bias lr
box: 0.05  # box loss gain
cls: 0.3  # cls loss gain
cls_pw: 1.0  # cls BCELoss positive_weight
obj: 0.7  # obj loss gain (scale with pixels)
obj_pw: 1.0  # obj BCELoss positive_weight
iou_t: 0.20  # IoU training threshold
anchor_t: 4.0  # anchor-multiple threshold
# anchors: 3  # anchors per output layer (0 to ignore)
fl_gamma: 0.0  # focal loss gamma (efficientDet default gamma=1.5)
hsv_h: 0.015  # image HSV-Hue augmentation (fraction)
hsv_s: 0.7  # image HSV-Saturation augmentation (fraction)
hsv_v: 0.4  # image HSV-Value augmentation (fraction)
degrees: 0.0  # image rotation (+/- deg)
```

```
translate: 0.1  # image translation (+/- fraction)
scale: 0.9  # image scale (+/- gain)
shear: 0.0  # image shear (+/- deg)
perspective: 0.0  # image perspective (+/- fraction), range
0-0.001
flipud: 0.0  # image flip up-down (probability)
fliplr: 0.5  # image flip left-right (probability)
mosaic: 1.0  # image mosaic (probability)
mixup: 0.1  # image mixup (probability)
copy_paste: 0.0  # segment copy-paste (probability)
```

Listing C.1: Hyperparameters

C.2 Intersection over Union

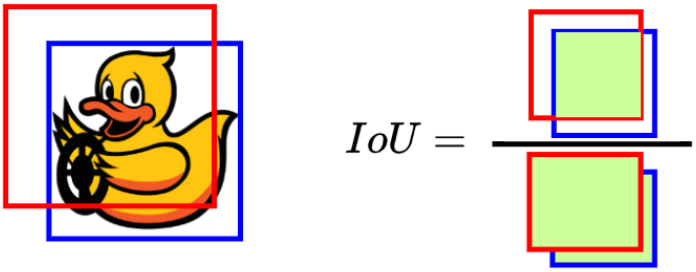


Figure C.1: A visual derivation of the Intersection over Union (IoU) index

C.3 YOLOv5 Training results

C.3.1 Performance metrics

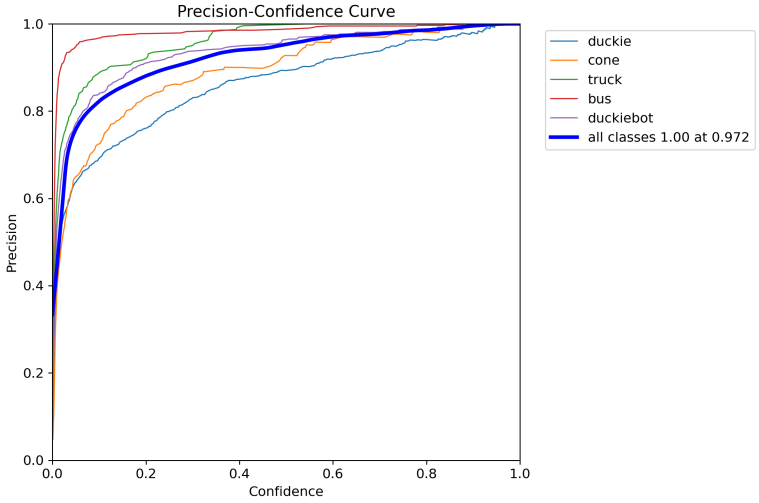


Figure C.2: Precision vs Confidence

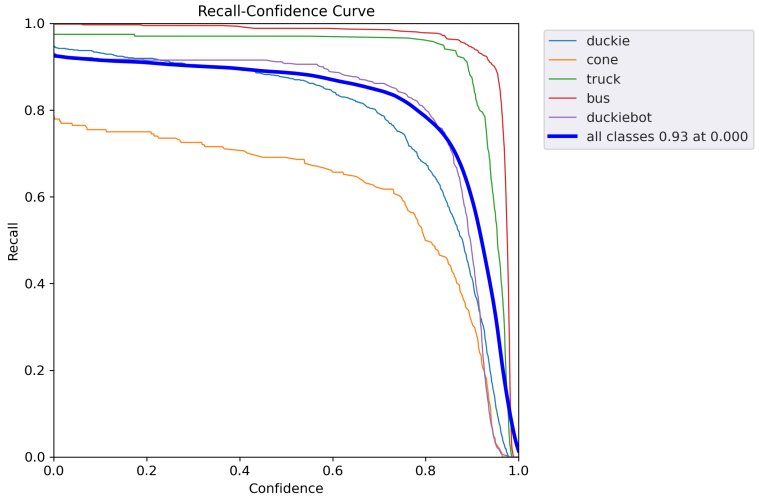


Figure C.3: Recall vs Confidence

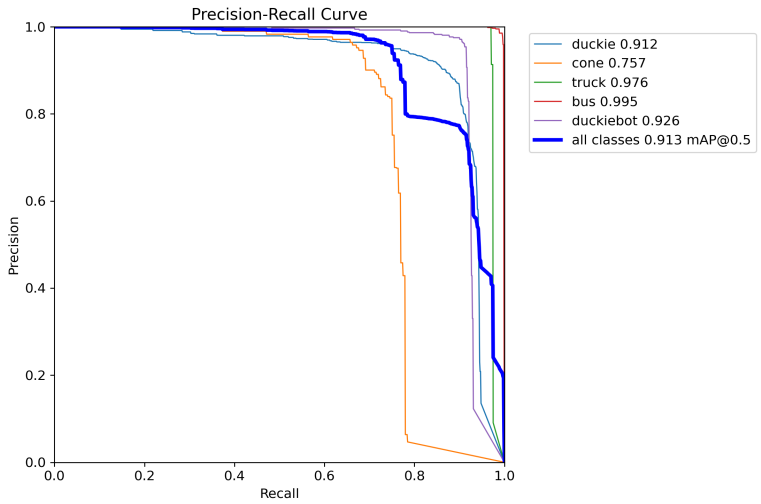


Figure C.4: Precision vs Recall

The F1 Score shown in Eq. C.1 prioritises detecting and classifying correctly instead of the correct bounding box size.

$$F1 = \frac{2 * R * P}{R + P} \tag{C.1}$$

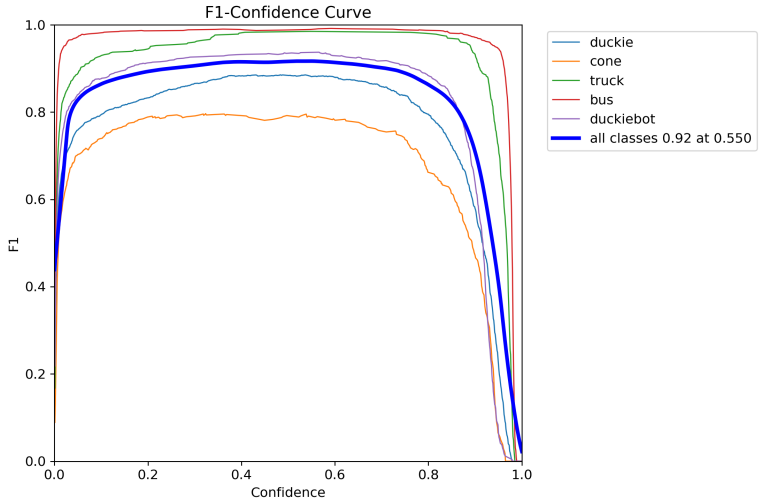


Figure C.5: F1 curve

C.3.2 Label data

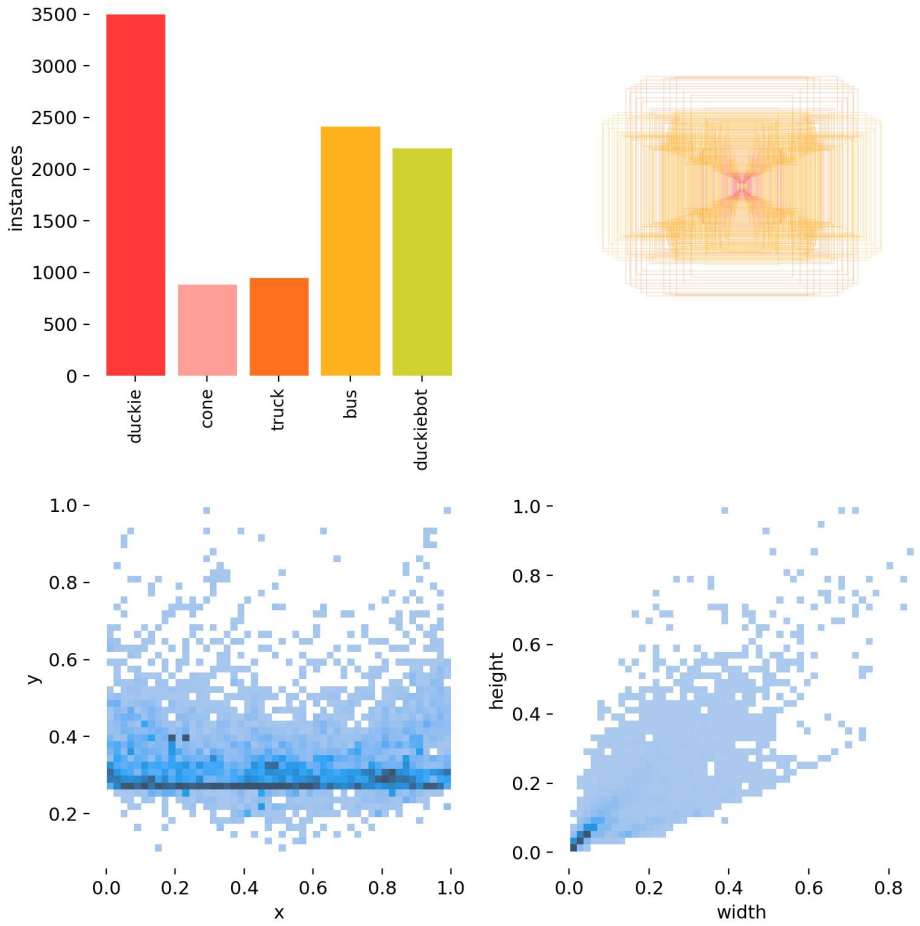


Figure C.6: Statistics on class labels. The top left shows class frequency, and the top right shows all labels in normalised height and width space.

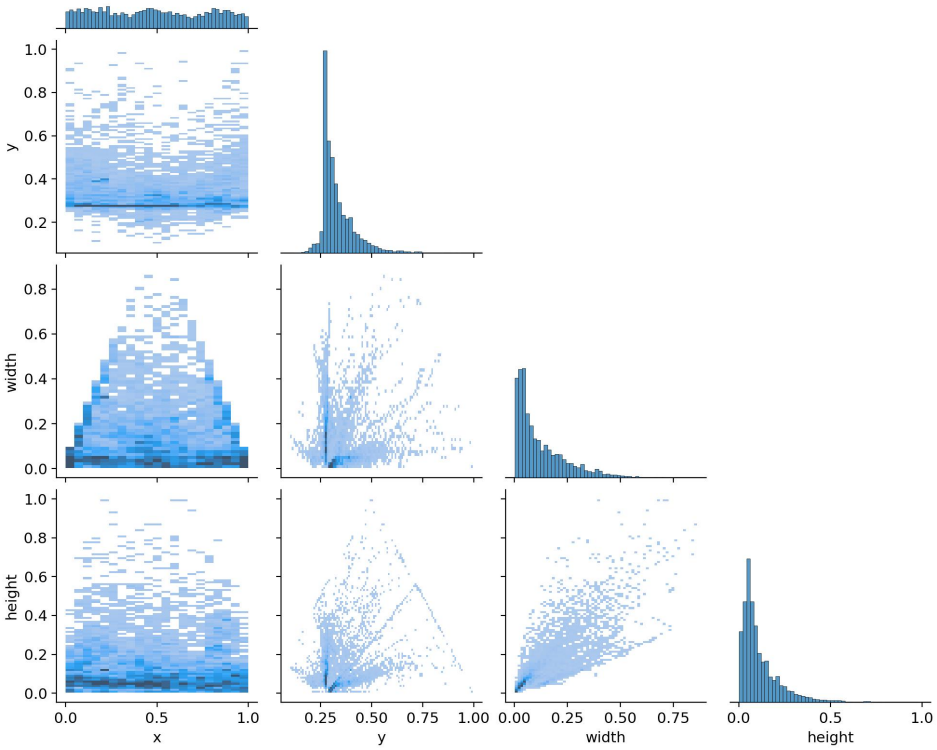


Figure C.7: labels correlogram. Provides insight into how often certain classes appear together in the training dataset. This information is useful for understanding class dependencies and relationships between classes.

Appendix D

ROS and Docker

Duckietown uses Robot Operating System (ROS) as its middleware and Docker for containerisation. As these are specific robotic software packages an explanation is provided for the reader.

D.1 ROS

Consider an AV with various sensors, actuators and communication protocols that need to communicate with a single server. Middleware is a software layer shown in Figure D.1 that attempts to abstract the hardware interactions from the developer [68]. The authors of [69] highlight key requirements for open-source middleware: abstraction of hardware and software, communication framework for data transport, ability to take logs and playback logs, timing analysis and simulation tools. Several middleware exist that all have their use case for different applications and phases of development [70]. Only Robot Operating System (ROS) is discussed here as it is the Duckietown-supported middleware. ROS is considered the primary development framework for robotic applications owing to its open-source, flexible framework [71]. ROS can be thought of as "a computation graph consisting of nodes interacting with one another over edges" [71]. These interactions consist of topics (data structures), messages and services. Considering Figure D.2, which represents the user-abstracted communication protocol, nodes (executables) and topics (hold information). Nodes can be of type subscriber or publisher. Publishers register their topics, and subscribers interested in the data will subscribe to the topic. An example is a camera which publishes a compressed image message to a topic in which an image processing node subscribes to the topic to obtain the compressed image.

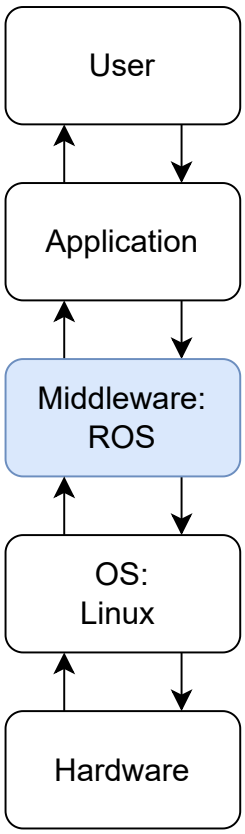


Figure D.1: Typical structure of a software solution showing the position of middleware [70]

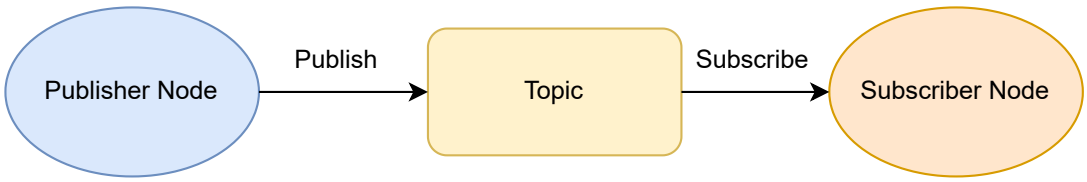


Figure D.2: Abstracted ROS publisher/subscriber communication

Figure D.3 shows a less abstract view where the ROS master manages communication between nodes. In reality, publishers register their topics with the ROS master, and subscribers query the master for the publisher node. Nodes can also provide services to other nodes, which can be thought of as functions that are called by a node and return a response. This centralised form of communication is useful for distributed robotic systems where nodes can exist on different machines [71]. ROS hosts additional tools such as a visualisation tool (Rviz) and a logging tool (Rosbag), and integration of simulator sensors, which are useful for debugging and testing. These tools contribute to the widespread use of ROS.

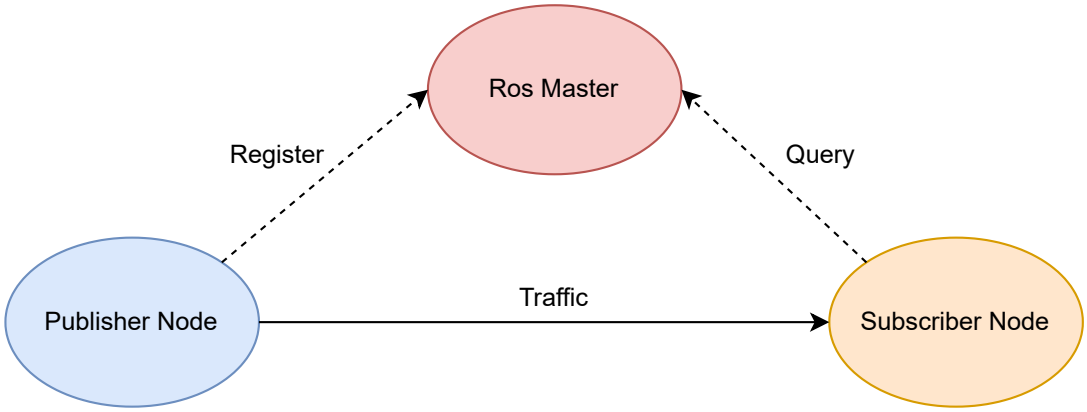


Figure D.3: ROS publisher/subscriber communication

D.2 Docker

Duckietown uses containerisation to ensure reproducibility and ease of deployment. A container is a lightweight, standalone, executable package that includes all the dependencies required to run an application. Figure D.4 shows a comparison of Containers and Virtual Machines (VMs), with the key difference being that containers do not require separate OS, making them more suitable for applications where many independent services need to be deployed, such as robotics platform.

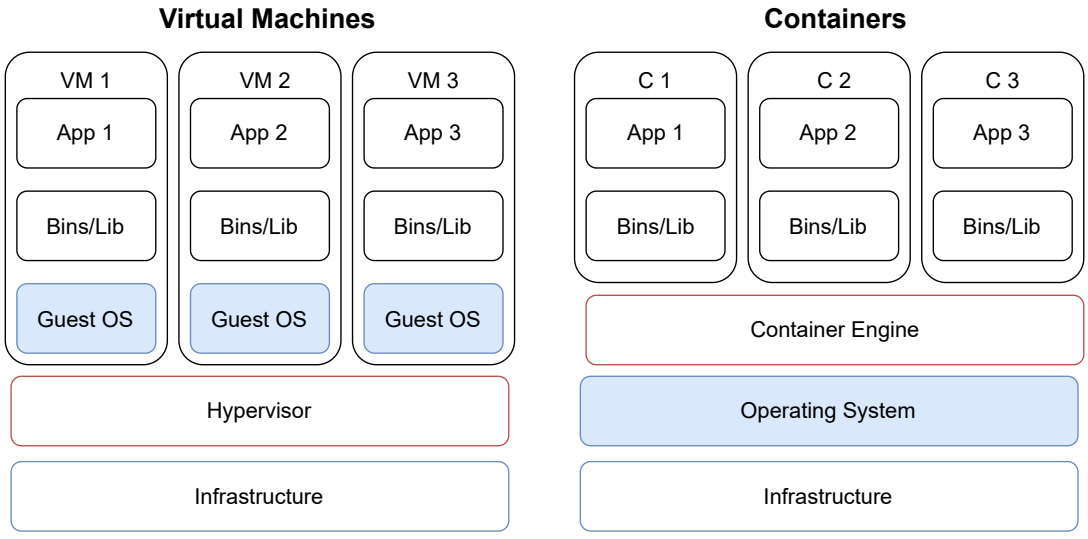


Figure D.4: Differences between Virtual Machines and Containers

Duckietown uses Docker for containerisation. Consider Figure D.5, and images are static, build time constructs and contain all the source code and dependencies required to run an application. Images consist of layers, with each layer depending on the layer below.

Docker containers are run-time constructs that are dynamic instances of an image. Each container has a thin, writable layer that stores all the changes to the container throughout its runtime and allows any number of containers to share access to the same image.

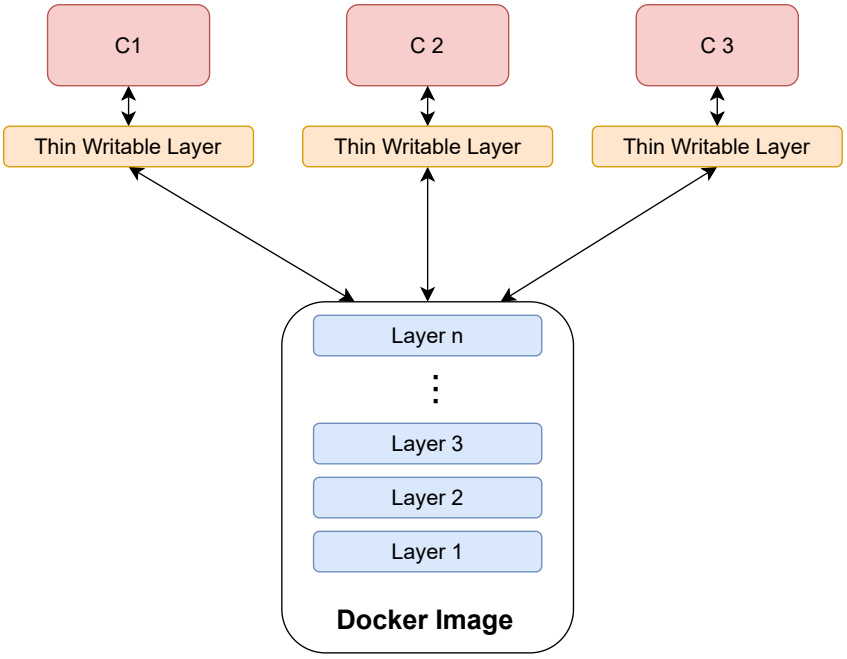


Figure D.5: Docker images, layers and containers

The docker structure is most easily understood using an example. Figure D.6 shows the Duckietown Docker image that all duckietown projects, including this report, adhere to. The image uses the Ubuntu OS base layer, followed by several layers of tools and dependencies indicated by yellow blocks. The blue blocks represent duckiebot-specific layers containing various drivers. Note that each layer represents a higher level of functionality. Duckietown provides ROS-compatible Duckietown Docker images where additional ROS nodes live in the `duckietown/dt-core` layer.

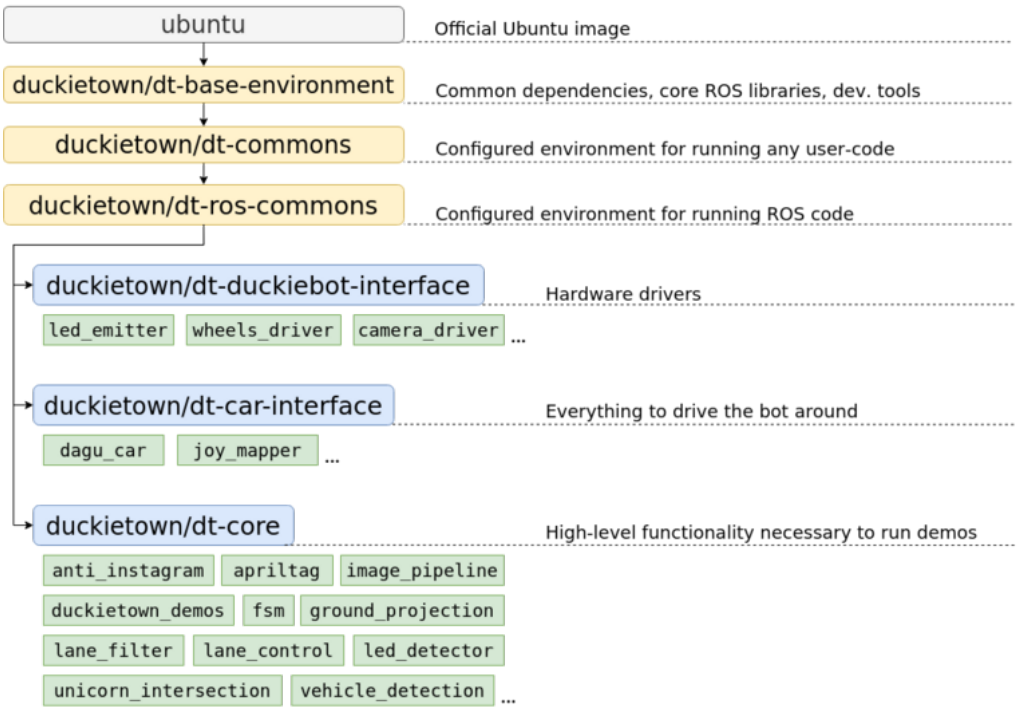


Figure D.6: The Duckietown Docker image hierachy [47]

Appendix E

Validity Analysis

This chapter comprises the research material from the validity analysis presented in Chapter 6. The goal was to identify future work on the duckiebot, primarily in the context of the MEC4128S UCT course. The chapter first delves into Autonomous Vehicle Architectures (AVAs), presenting an overview of the history of AVAs and current architectures. This is presented to motivate possible future work in producing an AVA using the duckiebot platform. The chapter then explores an alternative trajectory for future work by means of monocular depth estimation. This is the task of extracting information from a single camera. The chapter focuses on ML monocular depth estimation models.

E.1 Autonomous Vehicle Architectures

There exists no standardised robotic architecture. Fundamentally, robotics is an implementation-orientated field where architectures are developed for particular applications and target hardware. Hardware and software capabilities are ever-changing, and architectures are constantly evolving. In this way, robotics lack the specificity of other fields, such as mathematics and theoretical physics. However, three schools of thought have emerged: Cybernetics, Artificial Intelligence [1] and Deep Learning with many successful applications utilising a mix of the three approaches [72]. An interesting digression to make is drawing inspiration from nature. These bio-solutions, driven by survival forces, highlight the importance of trade-offs. Taking a frog of the (*Rana pipiens*) genus for example, there exists a direct sensory connection from the eye to the muscles of the tongue [73]. In this case, the philosophy of quantity over quality is favoured whereby having the direct

[1] AI in this case refers to the philosophy that emerged in the 1950s, which focused on symbolic representations and planning, emphasising the use of logic rules and symbolic reasoning. The goal was to replicate human-level intelligence in a machine [72]. Deep Learning refers to the modern approach of AI using data-driven ML algorithms.

connection, the frog can act faster with the trade-off that each strike of the tongue is not necessarily a fly. This is particularly relevant in AV applications where the trade-off between speed and accuracy of software architectures is of concern.

The initial development of AVs was driven by Autonomous Vehicle Competitions (AVCs). The first DARPA Grand Challenge in 2004 was created to initiate the development of the first fully AV [74]. DARPA, a research organisation of the United States Department of Defence, is representative of the specialised applications reserved for AVs in their infancy phase of development. The 2004 Grand Challenge was a 240km off-road course. None of the AVs completed the course, with the furthest completing only 11.78km. No winner was declared, resulting in a second challenge scheduled for 2005 [75]. All but one of the finalists improved on the 11.78km, including five vehicles completing the course.

Of these vehicles, only the TerraMax [76] used a camera as the primary sensor. Hyundai Autonomous Challenge in 2010 [77] required the explicit use of vision to detect obstacles. The 2012 version of the challenge [78] took a further step in using computer vision, requiring artificial vision to detect visible features such as road signs and pedestrians. Figure E.1 shows the system architecture used by the winning team of the 2012 Hyundai AV competition [13]. This architecture adheres to the widely accepted perception, localisation, planning, and control framework rooted in the principles of Cybernetics [79].

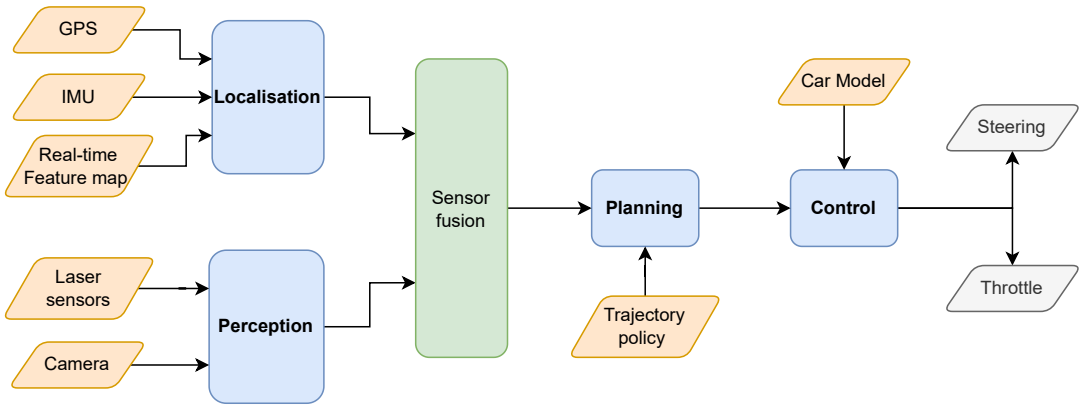


Figure E.1: 2012 Hyundai AV competition winning software architecture implemented by [13].

Notably, the system utilises both cameras and LiDAR sensors. In the architecture proposed, the LiDAR sensors are used for detecting moving objects [14] and barrier detection [15], and cameras are used for extracting lane markings [16]. These sensors are fused with GPS and IMU data using an interacting multiple model (IMM)-filter-based information fusion [80]. Fused observations are used downstream for planning. Since the map of the course was known for this application, the planning is composed of behaviour reasoning and local motion planning modules. Behaviour reasoning executes a rule-based

decision process based on a Finite-State Machine (FSM) and can be seen as an extension of the Braitenberg reaction-based behaviour. Local motion planning comprises two path planning algorithms: a road map-based (nominal situation) and a free-form path (unstructured environments). A selected algorithm is used with the behaviour reasoning to generate a feasible path iteratively. The architecture described is representative of the approach to AVs in these AVCs [78].

The AV industry has shifted its focus to data-driven approaches abetted by emerging ML techniques. Figure E.2 shows a simplified system architecture recently shown in Tesla's 2022 AI Day and used to demonstrate the shift in architecture [26]. Consider the Vision block. The Regular Network Structure (RegNet) is a CNN backbone and extracts features from the camera feeds [81]. The head is composed of several task-specific heads. The car detection head, for example, makes use of a one-stage YOLO-like head [46]. Object Detection, Traffic Lights and lane prediction are other tasks that use the features extracted in the previous layers of the network. These various tasks create the vector and feature maps. This marks a fundamental difference between the architectures shown in Figure E.1 and Figure E.2. Note that the Tesla architecture does not make use of any LiDAR sensors. The expense of LiDAR renders them unfeasible for mass production [17]. This architecture is representative of research efforts in using only vision for AV applications in the pursuit of Full-Self Driving (FSD) [2] [82]. As such, the development of CNN-based object detection algorithms has been a key enabler for the shift in architecture.

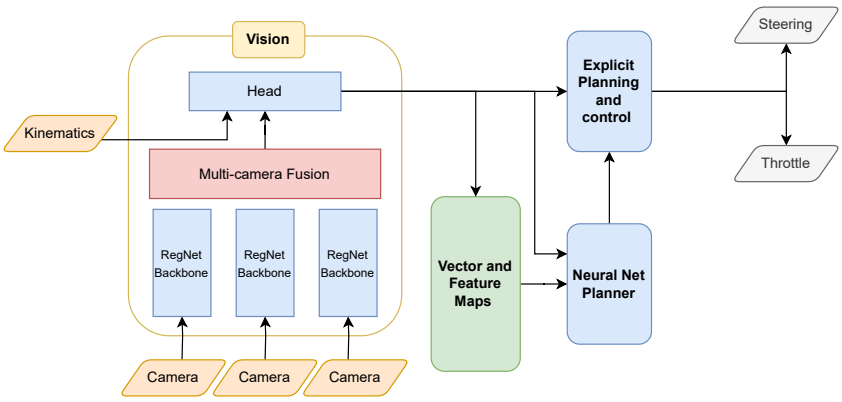


Figure E.2: Simplified Tesla Full-Self Driving software architecture [26]

Note that the Neural Net planner serves the purpose of future planning, where traditional path planning search algorithms generate control outputs similar to the Hyundai AVC architecture. Therefore, the Tesla-like approach can be seen as a modified version of the cybernetics framework that combines perception and localisation.

[2] FSD in this context coincides with level 5 of the SAE J3106 Taxonomy and is defined as the ability to drive in any condition without human intervention [2].

On the other hand, End-to-end architectures seek to use ML techniques to streamline the entire architecture [83]. Figure E.3 represents a Reinforcement Learning (RL) End-to-End architecture implemented on the duckiebot and trained purely in the Duckietown-gym simulation environment [53].

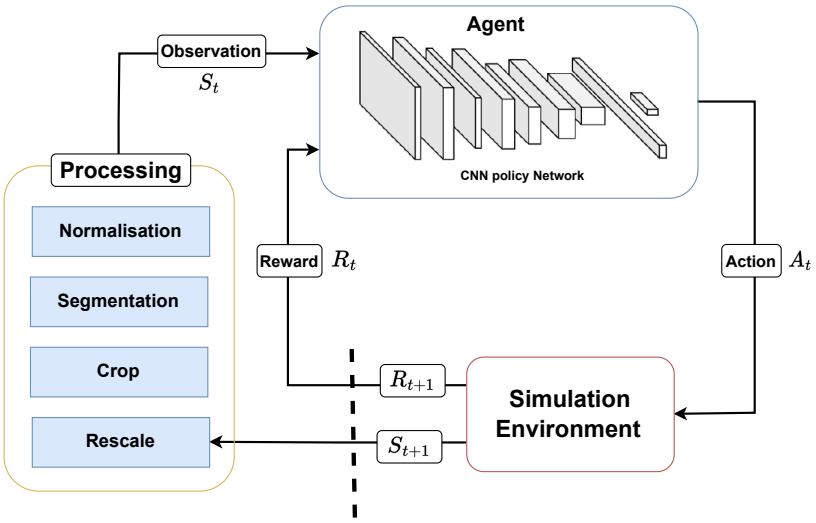


Figure E.3: End-to-End RL architecture proposed by Almasi, Moni and Gyies-Tóth [53]

This architecture is a complete departure from the cybernetics framework. A single neural network takes in raw sensor data and outputs control commands. RL is an ML subfield focused on solving Markov Decision Problems(MDPs) where an agent learns by seeking to maximise a reward [84]. Consider Deep Mind's AlphaZero, achieving superhuman ability in the board game Go, previously thought impossible by a computer [85]. The advantage of this architecture is that it does not require any prior knowledge of the environment, and agents are trained self-supervised in a simulation environment [53]. [86] used an End-to-End framework to train an agent to navigate a simulated racetrack. Kendal et al. propose a method to train a lane-following agent in a simulation environment in under 30 minutes that can follow a lane using a real-world vehicle [87]. Currently, limitations exist with developing optimal reward functions [87] and the ability to transfer the trained agent to the real world, which limits RL techniques for any safety critical task such as AV applications[53]. Despite this, the field is receiving immense attention and is marked as the future architecture for AVs [88].

E.2 Depth Estimation

Depth estimation is generating increased interest due to the necessity for real-time depth maps in other fields, such as virtual reality [89] and robot vision systems. Depth estimation is formalised as recovering higher-dimensional information from a low-dimensional [90]. There are two types of depth estimation techniques: active and passive. Active techniques require using a wave source such as a laser, infrared light or radar. Passive techniques use environment ambient light. Active techniques are more accurate but require additional hardware other than cameras. As such, these active techniques are not considered in this section. Most passive solutions use two or more cameras to estimate depth through stereo vision. However, generating depth estimation using a monocular system is attractive in the AV field as it can replace the need for LiDAR [56]. The section derives the pinhole camera model to explain stereovision and demonstrate the limitation of extracting depth from a single image. After that, a survey of standard monocular depth estimation techniques is presented.

E.2.1 Pin-hole Camera

The pinhole camera can be simplified to the Figure E.4. A light-sensitive sensor is placed in a black box with a small opening, allowing limited rays of light to enter. The idea is that each ray is incident to a specific pixel on the sensor and can be used to reconstruct an image of the world frame.

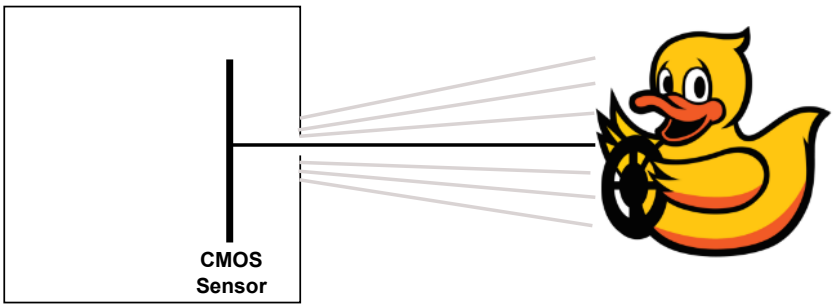


Figure E.4: Simplified Pinhole camera

The pinhole camera is modelled using the projective perspective shown in Figure E.5. Figure E.6 moves the sensor image in front of the camera centre to avoid inversion and assist with visualisation. Figure E.7 then shows a 2D representation of the model.

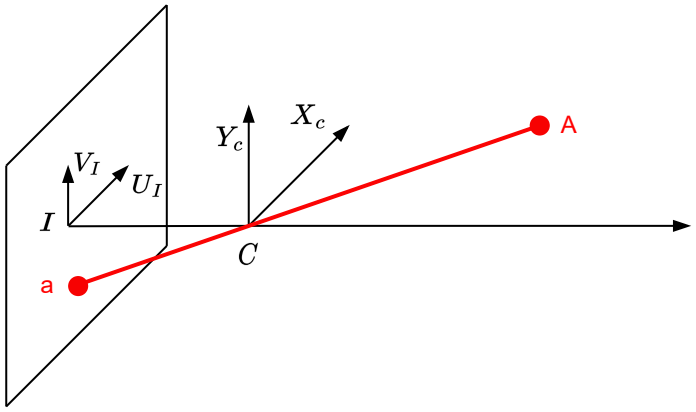


Figure E.5: Projective persepctive 3D model of the Pinhole camera.

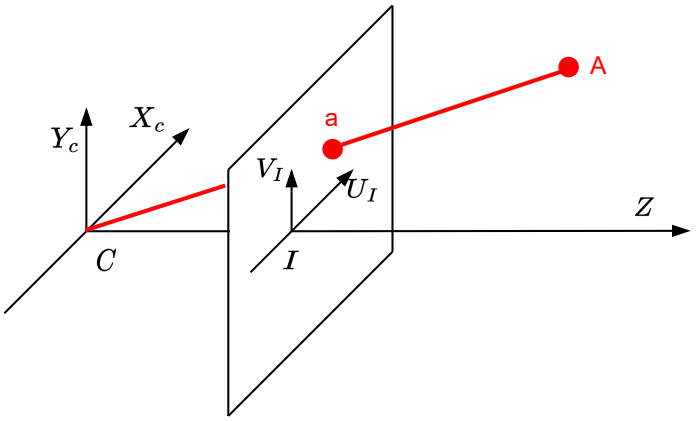


Figure E.6: Moves the image plane in front of the camera centre to avoid inversion in the derivation.

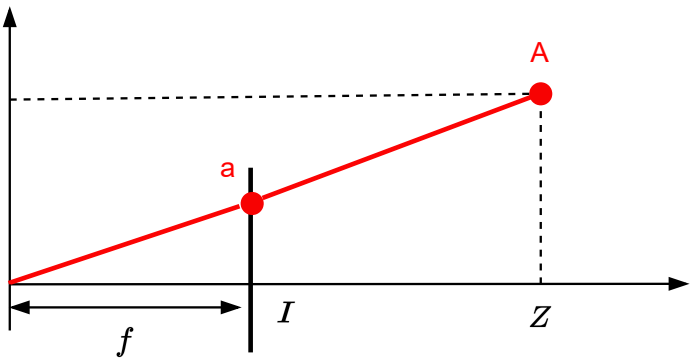


Figure E.7: Planar representation of the projective perspective.

Therefore, a point in the world frame can be mapped to a point on the image sensor. The pinhole camera model is simply a projective transformation between the world and

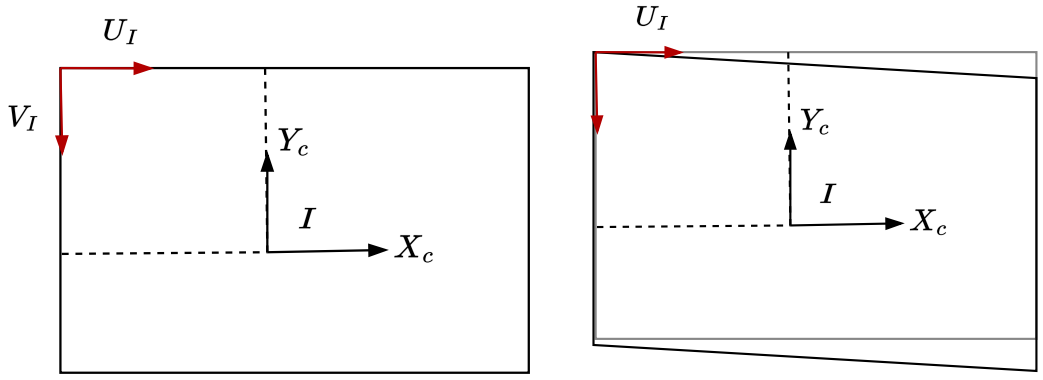
image frames. The model is defined by below where f_x and f_y is the focal length in the x and y direction, respectively:

$$a \approx \frac{fY_c}{Z} \tag{E.1}$$

$$a \approx \mathbf{P}A \tag{E.2}$$

$$P = \begin{bmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{E.3}$$

Figure E.8a and E.8b show considerations that must be applied to the pinhole model. The image plane origin is not necessarily coincident with the camera centre and is represented by U_I and V_I. Additionally, the image plane is not necessarily perpendicular to the optical axis and is represented by s.



(a) Non-coincident camera centre (b) skew

Figure E.8: Pinhole Camera model considerations.

The pinhole model extends to Eq. E.4 where U_I and V_I is the x and y coordinate of the principal point, respectively, and is known as the *intrinsic parameters*.

$$P = \begin{bmatrix} f_x & s & U_I \\ 0 & f_y & V_I \\ 0 & 0 & 1 \end{bmatrix} \tag{E.4}$$

Note that this equation only has 5 degrees of freedom (DoF) as the projective perspective is only defined up to scale. Consider Figure for a visual explanation of scale ambiguity responsible for the loss of a DoF. This is the fundamental limitation of the camera that

makes extracting depth from a projective perspective impossible. In essence, this is another example of the *curse of dimensionality* referred to in Section 2.1.

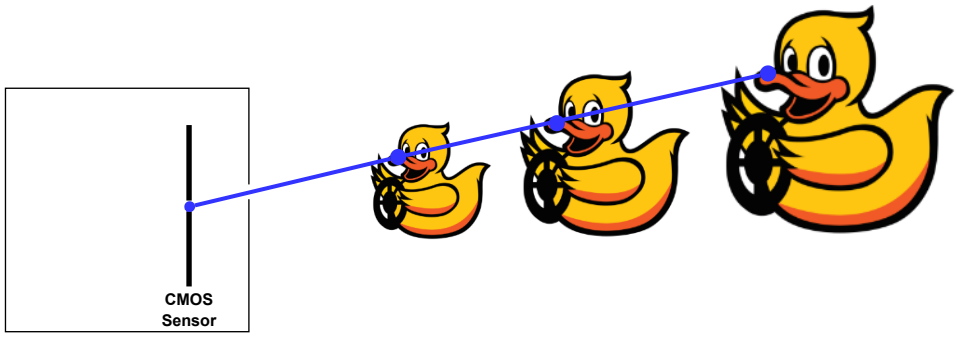


Figure E.9: Pinhole camera scale ambiguity showing the degenerate cases where a line through the camera centre projects to a point on the image plane and a plane through the focal point projects to a line. This makes it impossible to decode the distance of an object using a projective perspective.

A further extension of the model is made to account for the fact that the camera model may not be aligned with the world frame. This is done by introducing a rotation matrix, $\mathbf{R}$ and translation vector, $\mathbf{t}$. The *extrinsic parameters* of the pinhole camera model describe the camera's pose (position and orientation) in the world coordinate system and are represented as:

$$\mathbf{X}_C = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{R}_W \tag{E.5}$$

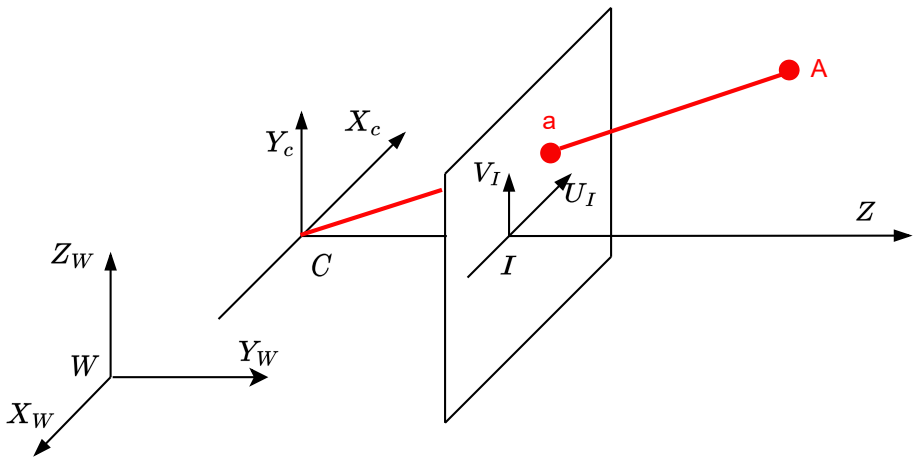


Figure E.10: Projective persepctive represented in the world frame, W.

Note that the transformation matrix is written in homogenous coordinates, which results in the transformation matrix being invariant to scaling. Converting the intrinsic parameter matrix, $\mathbf{P}$, in homogenous coordinates, the complete pinhole camera is defined below. It contains 11 DoF and is obtained through well-established calibration algorithms such as RANSAC [91].

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} f & s & U_I & 0 \\ 0 & f & V_I & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{E.6}$$

E.2.2 Stereovision

Marr is widely accepted to have transported vision problems into computer problems, providing a theoretical computational framework for recovering depth estimation inspired by human binocular stereopsis or stereovision [3] [92]. Stereovision, in the context of computer vision, arose from this framework and can be defined as the process of extracting depth estimation from two or more images taken from different viewpoints shown in Figure E.11 and derived in Eq. E.7

$$D = \frac{fB}{\delta} \tag{E.7}$$

Stereovision is well-defined in literature and can be implemented relatively easily using `OpenCV` and `Matlab`. In the AV field, many successful object detection systems using stereovision have been implemented [93], [94]. Stereovision, in this form, is not considered here as it requires the use of at least two cameras. However, it is presented because many monocular techniques attempt to mimic or use principles of stereovision.

E.2.3 Monocular Depth Estimation

Monocular depth estimation is the problem of extracting depth information from a single image. This problem is inherently underdefined due to the scale ambiguity of an image[4] but not physically impossible. Consider human's ability to estimate depths in novel environments. For this discussion, monocular solutions can be categorised into

[3] Stereopsis is the perception of depth obtained by visual information from two eyes.
[4] write something here

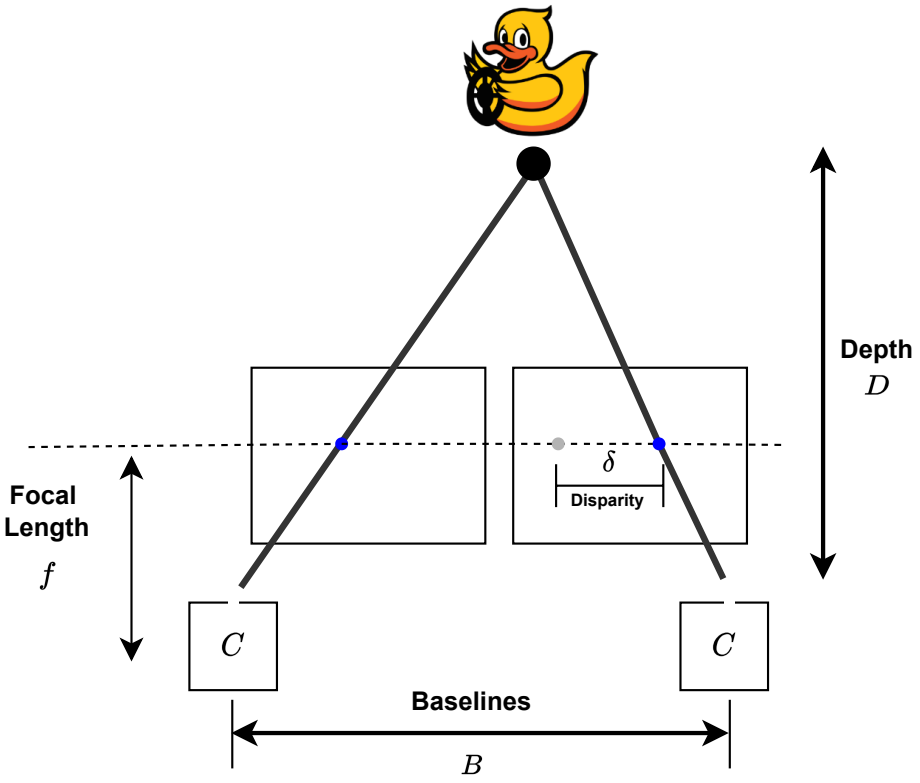


Figure E.11: Stereovision is the process of extracting depth estimation using two cameras to triangulate an object's location in the world frame.

three main methods: depth-map-based, hardware-based and motion-based [57]. Hardware approaches rely on the fusion of cameras with additional sensors. This was a popular approach in the AV field since vehicles usually hosed a variety of sensors [5]. The authors of [95] propose a neural network architecture to fuse RGB image features with sparse LiDAR features for an AV application. Depth estimation has been implemented on the duckiebot, fusing a YOLO model using the main camera and an additional stereo camera [12].

Motion-based approaches rely on a sequence of images. Structure-from-Motion (SFM) is a successful traditional technique, leveraging the motion of a camera for stereo matching [54]. Stereo matching is the problem of calculating the disparity, δ, for each pixel given two images taken at different relative positions. δ is often obtained using feature tracking techniques such as Kanade-Lucas-Tomasi (KLT) [96] or its modifications [97]. Estimating the distance travelled between two images and the disparity can be used to estimate depth via triangulation similarly to Eq. E.7. Monocular SFM are appealing due to the low cost and calibration requirements, but the lack of a fixed stereo baseline leads to inevitable drift [98]. The issue of drift is usually mitigated by using prior knowledge such as non-

[5] These approaches are favoured less for marketable AVs due to the cost of the additional sensors

homolomonic constraints of wheeled robots [99], or a known height such as camera height [98]. The requirement of motion, by definition, is a limitation of the SFM framework in the AV field. Consider when another vehicle moves at the same speed; the car will be considered stationary [57].

As the name suggests, depth-map approaches infer distance from a single image by generating a depth-map. Classical approaches exploit geometric priors such as corners and edges that are expected for a particular object. Additionally, scene information is usually obtained from making strong assumptions of the scene [100]. Further development of these approaches is inspired by analogy to human depth perception from monocular cues: texture variations and gradients, defocus and colour [101]. These approaches are appealing as they do not require additional hardware or motion. However, they are not robust without their various assumptions and, therefore, fail to generalise to novel environments [101].

Recent depth-map approaches use supervised or self-supervised CNN techniques, which exploit a colour image to generate ground-truth depth maps. These supervised techniques are highly dependent on the richness of the dataset, which is non-trivial for real-world applications and often relies on LiDAR for generating these maps [57]. Various exciting approaches have been proposed that use weak or sparse datasets. [55] proposes training a CNN on pairs of small image patches where true disparity is known. These small patches can be readily obtained from existing datasets. Images are first passed through feature detection to generate pairs of small image patches. A CNN then outputs the similarity of them. Modern graphically rendered environments or gaming environments provide another means of obtaining data [102]. The main issue with these approaches is domain transfer from simulation to real-world environments. Various image style transfer approaches have been proposed to deal with this issue [103]. These techniques train a CNN to transfer styles (such as a learnt texture) from one image to another by, most commonly, manipulating image pixels with some noise [104].

Without ground truth depth, self-supervised techniques have been trained using image construction [105]. During training, secondary supervisory signals are used via *stereo pairs* or *monocular video*. Stereo pairs are two images that have been taken side by side. [106] demonstrate a real-time self-supervised model trained by minimising the reconstruction error compared to the second stereo pair. Monocular video is less constrained since the images need not be side by side. These models are required to generate depth maps as well as estimate ego-motion [6] between frames. [56] demonstrate that these models outperform models trained with stereo pairs on the KITTI dataset but with added complexity. [56] provides a SOTA network shown in Figure E.12 that can be trained on steropairs or

[6] Egomotion is defined as any displacement of the observer in the world view.

monovideo. Self-supervised have shown to be highly capable of predicting depth but tend to have poor generality even to different camera models since training relies on intrinsic parameters [57]. [57] propose a CNN network that uses predictions from a YOLOv5 model that the authors claim provides better results than Monodepth2. However, it appears that the YOLOv5 model is constrained to be trained on the KITTI dataset, and it would be non-trivial to generalise to another dataset, such as the Duckietown dataset

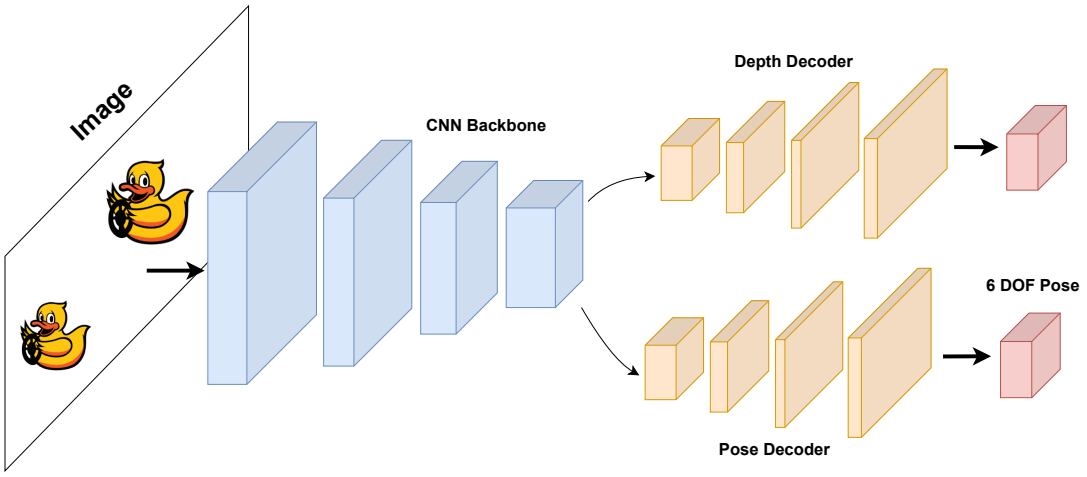


Figure E.12: Monodepth2 [56] uses a CNN backbone to produce a feature Map. The feature map is then fed into a depth and pose decoder network. The depth network is a standard convolution U-Net network. The pose network predicts the pose between frames.

In summary, the CNN-based techniques provide valid solutions for depth estimation and, to the best of this report's knowledge, have not been implemented on the duckiebot; despite this, it appears to be a promising route for future work on the duckiotbot.

Appendix F

Management Portfolio

This section outlines the project's planning management process, including using a Work Breakdown Structure (WBS) and two Gantt charts. The WBS was compiled at the end of the second week once the scope and milestone analysis were completed. The first Gantt chart aligns with the WBS, and the second Gantt chart was created at the project's end to reflect and allow for a comparison of the actual project progress.

The project was divided into seven stages, as shown in the WBS. The project initiation phase included scope and objective analysis, followed by a milestone analysis to identify the significant components of the project. These components included the literature review, object detection model, Duckiebot implementation of the object detection model, and the addition of the depth estimation model. Initially, the project scope included a depth estimation controller. However, it was discovered that the proposed method would not function, leading to a change to using the Braitenberg controller. More details about this change can be found in Section F.2.

The literature review was divided into sections, with the first part involving a broad literature review to identify valuable resources. The second part, titled "Targeted research", analyzed the resources found in the first part. Following the literature review was the object detection model, which was divided into two parts: training and implementing the model on the Duckiebot. The depth estimation model was intended to be added to the Duckiebot implementation as a controller to enable the Duckiebot's movement. The final stage involved report writing, which was planned to begin after implementing the object detection model. This was motivated by the idea that writing the literature review and methodology would be easier immediately after implementation.

Following these documents are two sample agendas and minutes from the two meetings held with the supervisor. The complete set of agendas and minutes can be found in the additional submissions management portfolio.

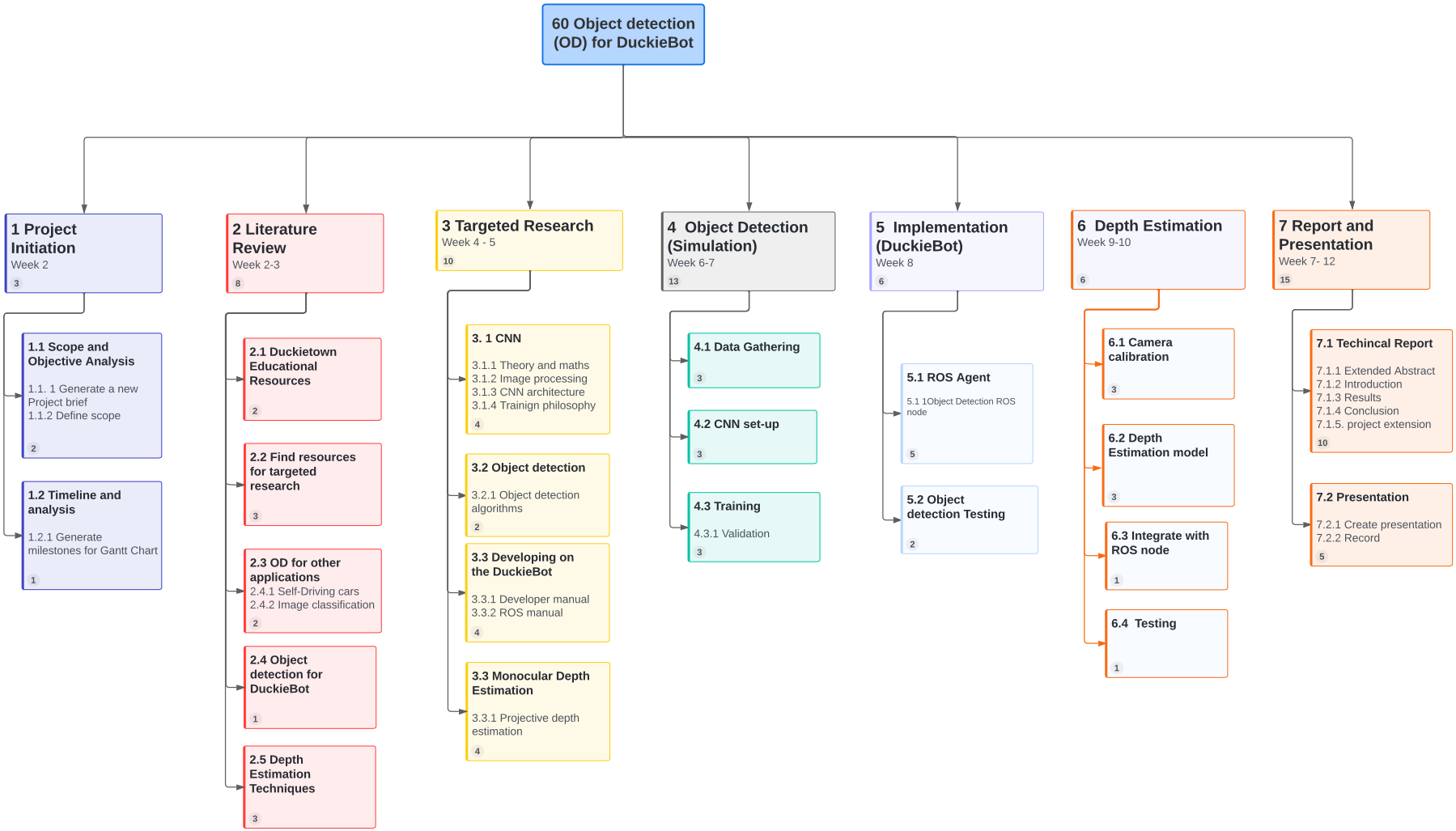


Figure F.1: Work Breakdown Structure

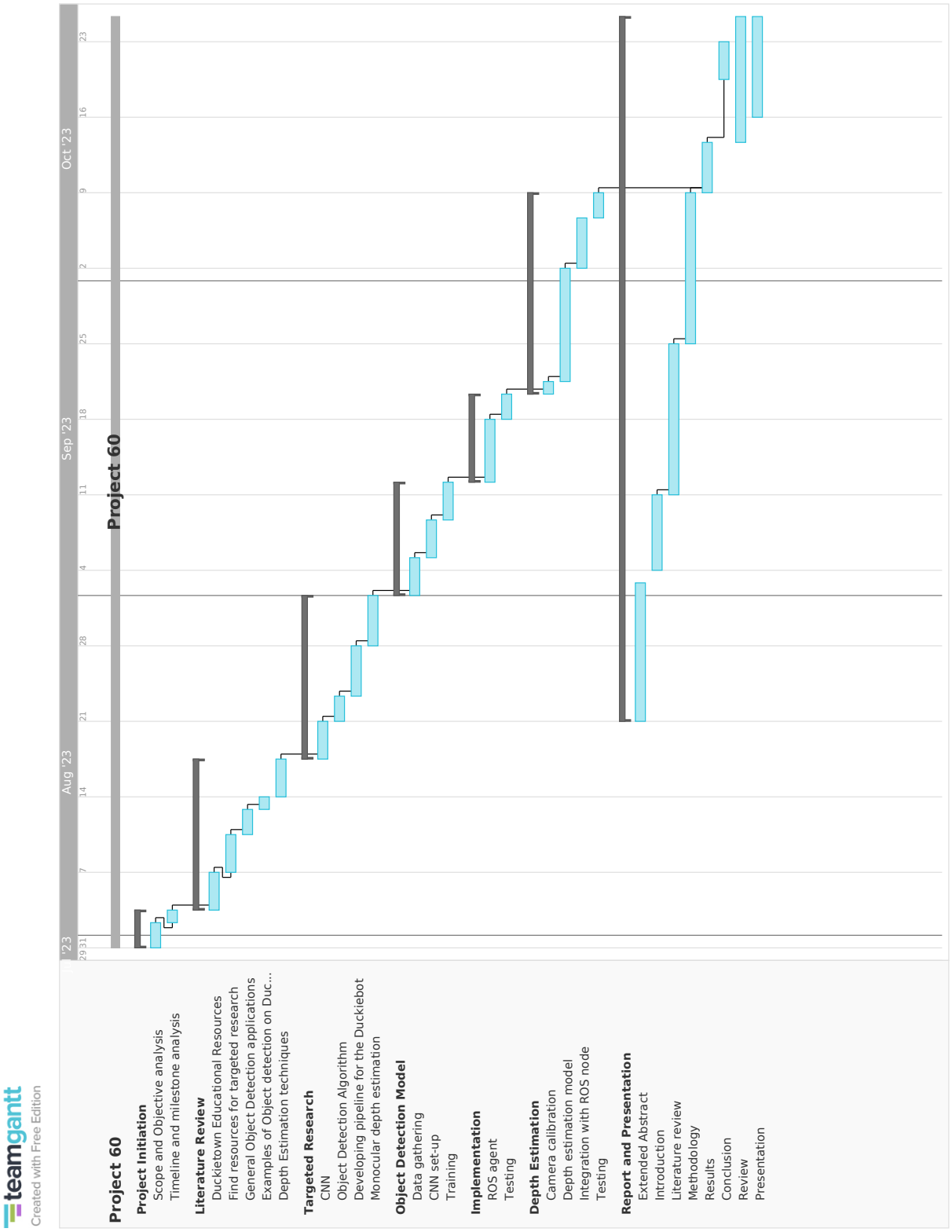


Figure F.2: Initial Gantt Chart

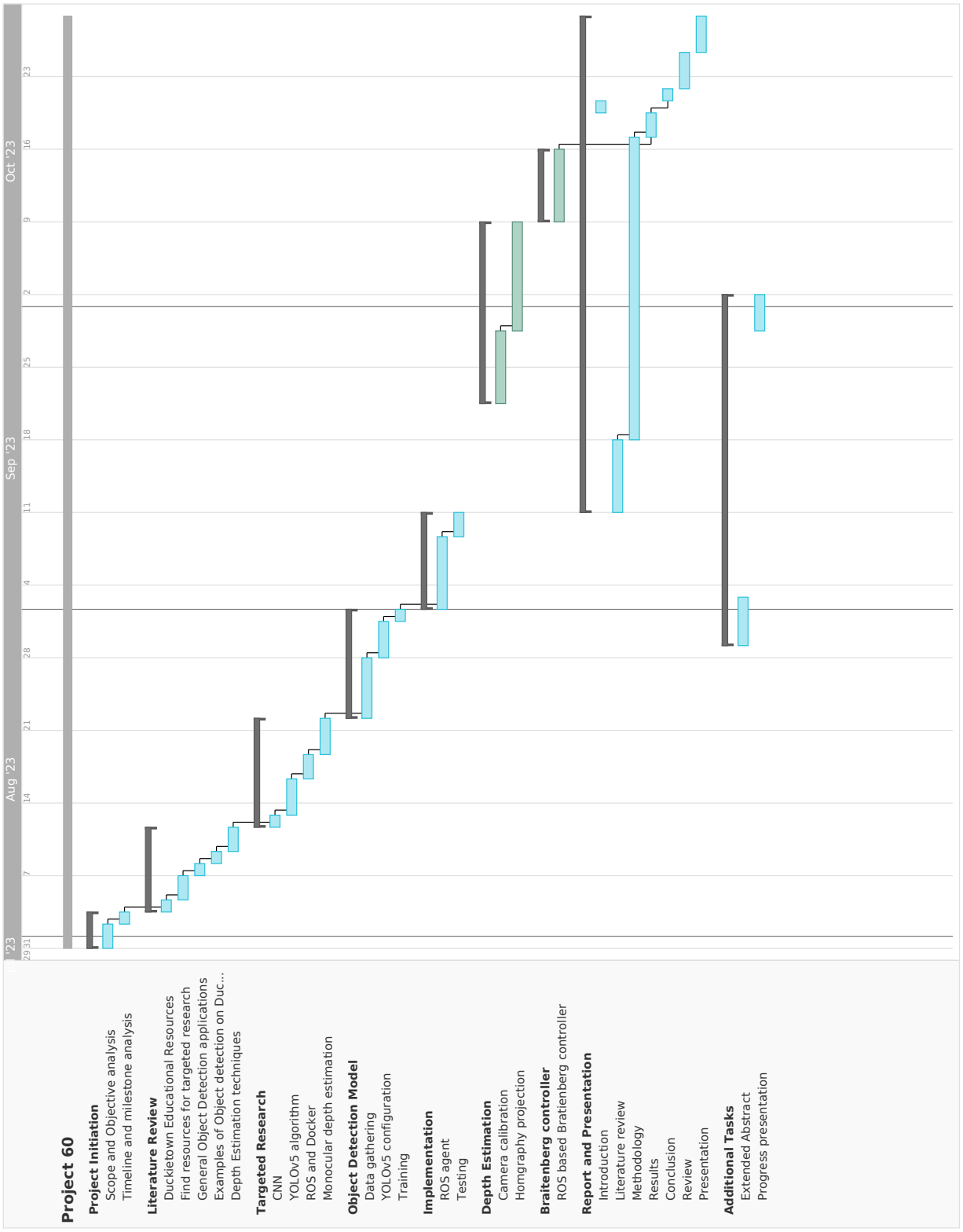


Figure F.3: Revisted Gantt Chart. The tasks in red represent the major changes in the two Gantt charts.

F.1 Agenda and Minutes

In the meeting on September 18, a PowerPoint was presented, and 4 sample slides were shown. The additional submissions management portfolio includes the complete PowerPoint and other PowerPoint.

F.1.1 18 September

Time: 10:00 am Location: Room 215.2

Agenda

1. Review of action items [2 minutes]
2. Update on progress [3 minutes]
    a. Report
    b. ROS Agent
3. PowerPoint Presentation [15 minutes]:
    a. Dataset generation
    b. Training details
    c. Training results

Minutes:

1. Review of action items [2 minutes]
    a. Dino contacted Zak and received various extrinsic matrices that can be used to back-calculate a plausible homography.
    b. The homography issue has not been resolved even after using the homography modified with Zak's extrinsic parameters. Various alternative depth estimation implementations exist; however, they are fairly complex, given the limited time left in the project.
2. Update on progress [2 minutes]
    a. Report: An initial outline of the Final Report was presented, including the subsections for the introduction, literature review, and methodology.
    b. ROS Agent: A new YOLO model was trained to detect duckiebots.

3. PowerPoint Presentation [15 minutes]

    a. Dataset generation: The generation of the dataset can be divided into two categories—real and simulated images.

        i. Real images: Duckietown provides a dataset of real images from the duckiebot. These images must be resized from 640x480 to 416x416, and labels must be reformatted according to the YOLO format.

        ii. Simulated images: The simulated images are obtained from the Duckietown-gym. These images are segmented, meaning a single pixel value represents each object. The pixel values are iteratively filtered, and bounding boxes are drawn for each contour in the filtered images. The images were resized, and labels were generated in the same manner as the real images.

    b. Training details

        i. 10 epochs and a batch size of 32

        ii. Explanation of the SGD optimizer and the importance of batch size

    c. Training results

        i. Brief explanation of precision, recall, IOU, and mAP.

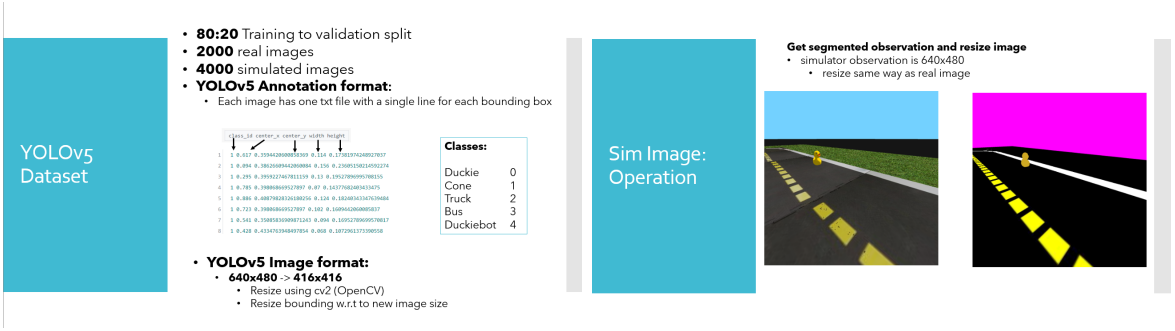
        ii. Data showing the various loss functions over the epochs.

        iii. Training and validation labels, as well as predictions, were shown.

Action Items:

1. Complete the first draft of the literature review by September 25, 2023 [Dino].

2. Interpret the remaining YOLO results by September 25, 2023 [Dino].



(a) Drunk duckies crossing the street.

(b) Duckie army.

Figure F.4: Sample slides from the PowerPoint presentation on September 18.

F.1.2 9 October

Time: 12:30 pm Location: Room 215.2

Agenda

1. Calibration procedure and result [5 minutes]
    a. Calibration procedure
    b. Validation
2. Progress on using the homography [10 minutes]
    a. Depth estimation algorithm for using the homography.
    b. Roadblocks
3. Discuss report structure [5 minutes]
    a. Table of contents and aims.

Minutes:

1. Calibration procedure and result [5 minutes]

a. Extrinsic calibration is carried out using the chessboard pattern placed on the ground. The chessboard corners are then mapped to a world coordinate frame that sits at the midpoint of the duckiebot's axle. The homography is then estimated using the duckiebot image of the chessboard and the world map.

b. Validation was shown by computing the inverse homography, applying it to the chessboard corner points from the image, and overlaying this on the world map.

2. Progress on using the homography [15 minutes]

    a. The homography is applied to the corners of detected bounding boxes to project the bounding boxes onto the ground plane. After this, distance would be inferred.

        i. Arnold explained that it is not possible because the bounding box of the duckiebot does not purely exist. As such, it is impossible to map bounding box points onto the ground map. This provides a possible explanation for the homography giving negative values as points on perpendicular planes are being projected.

    b. Possible alternatives to test the object detection model on the duckiebot.

        i. Braitenberg controller: This controller divides the image into a left and half region. Different behaviours can be obtained depending on how the left and right regions are connected to the wheels. By connecting the left half to the left wheel and the right half to the right wheel, the "fear" Braitenberg vehicle is achieved, where the vehicle will always steer away. If more objects are detected on the left, the left motor increases its velocity and steers away from an object.

            1. Arnold noted that this would be acceptable to test the YOLO model.

        ii. The left and half regions can further be broken down into regions which carry more weight and increase the motor speeds to a higher degree. An example was provided where regions closer to the duckiebot carry more weight. Testing different regions could be a possible output of the project.

            1. Arnold accepted this as a possible outcome.

Action Items:

1. Implement the Braitenberg controller and perform preliminary tests by October 12, 2023.

F.2 Reflection

The project's first half went relatively according to plan, with implementing the YOLOv5 model slightly ahead of schedule. As a result, the focus was shifted entirely to writing and putting depth estimation on hold for a week and a half. Returning to the depth estimation controller, camera calibration took much longer due to ongoing issues with the Duckietown calibration tool. Eventually, calibration was achieved using openCV tools. The next setback was the homography not working as expected, causing a delay in the depth estimation controller. During a meeting on October 9, it was discovered that the homography would not work in this case, as it is impossible to project bounding box points in a vertical plane onto the ground plane. A new direction had to be taken, and the Braitenberg controller was implemented. This was a relatively quick process, and the controller was implemented and tested within a week. However, this severely impacted the report writing, which had to be completed in less time than planned.

In hindsight, the most obvious shortcoming in the project planning was specifying a scope without rigorous investigation to determine its feasibility. This is a significant lesson that I have learned and will be sure to include in future project plans.