Chair of Integrated Systems
Department of Electrical and Computer Engineering
Technical University of Munich

Real-Time Object Detection for Autonomous Driving: An Empirical Study in a Small-Scale Urban Environment

Bachelor thesis

Author: Krittin Chaowakarn

Advisor: Michael Meidinger, M.Sc.
Supervisor: Prof. Dr. sc. techn. Andreas Herkersdorf

Submission date: September 9, 2025

Abstract

The Duckietown platform is designed to study and research autonomous driving systems in a small-scale urban environment. Our lab's previous work has designed a visual-based algorithm for object detection, relying on patterns based on Duckies' hue, saturation, and lightness to tackle the inefficiency of the preceding time-of-flight sensor-based approach, detecting an obstacle without recognizing an object. However, the visual-based algorithm is limited to only Duckie detection, meaning that the algorithm does not work with other objects such as Duckiebots and stop signs. Hence, this thesis focuses on a real-time object detection system and its GPU utilization, and therefore highlights two important improvements at the same time: 1. enabling more targets for object detection for the Duckietown platform, and 2. avoiding CPU utilization by migrating an object detection model to a GPU. This thesis has also collected lab data for the final evaluation.

Contents

List of Figures

List of Tables

1 Introduction

Motivation In recent years, the concept of autonomous driving systems (ADS) has become more feasible, such as Tesla's Autopilot; Cruise, backed by General Motors; and by BMW in partnership with Mobileye. Comparably, tech companies: Waymo (Alphabet), Project Titan (Apple), and Zoox (Amazon), have also conducted advanced research to push the capabilities of self-driving cars [3]. However, due to the high cost of full-scale autonomous driving vehicles [46], this emerging research field leads to research in autonomous driving in a small-scale environment, which also helps ADS research become more approachable, with compact on-board hardware. This thesis adopts the Duckietown platform [56] to perform real-time object detection in a small-scale urban environment because of its rich support from the non-profit Duckietown foundation with its large global community.

In our project, the previous computer perception system implemented on Duckiebots relies on a Time-of-Flight (ToF) laser-ranging sensor, which provides discrete and sparse spatial information, leading to lower computational cost, making it suitable for small devices. However, despite its computational cost-effectiveness, its imprecise object localization due to the sensor characteristics has adverse effects, especially in real-world applications such as collision avoidance. Previous work [43] attempts to recognize a Duckie with a visual-based algorithm relying purely on mathematical logic, yet the approach does not generalize to other objects. As a matter of fact, this Bachelor's thesis focuses on utilizing a camera sensor to increase reliability through deep-learning-based object detection.

Related work Recent studies in deep-learning-based object detection using a camera can be categorized into two types: one-stage and two-stage systems. One-stage detectors, such as YOLO [60] and DETR-based [8] models, predict bounding boxes and classification in a single pass, while two-stage systems, most prominently Faster R-CNN [65], first generate region proposals from an intermediate feature map before refining the final results. Undeniably, one-stage systems generally perform faster than two-stage systems with a trade-off of less accurate results.

Goal of this thesis This project focuses on the model deployment on an NVIDIA Jetson Nano, a small computer with a built-in Central Processing Unit (CPU) and Graphics Processing Unit (GPU) with limited computational resources. Hence, this thesis mainly explores the possibility of model deployment by balancing model size

and performance, utilizing the GPU to alleviate CPU usage, and implementing fundamental decision-making to ensure the model's integrity.

Steps required to reach this goal To reach the aforementioned goal, this project starts by becoming familiar with Duckietown's basic setup based on the Robot Operating System (ROS). Some tools, including the Graphical User Interface (GUI) of the Duckietown system and Docker, are essential for the ease of system deployment and the management of software requirements, respectively. The next step is to find candidate object detection models predicated on specific aspects: the simplicity to train, test, and deploy; and a trade-off between speed and performance. Another step is to assemble a dataset customized to our lab, which is a significant process, for it is a part where the selected model will be truly evaluated in our environment. The final step is to create logic for fundamental decision-making, such as stopping when a Duckie is in the perceptual field.

2 Background Knowledge

The background knowledge is divided into five sections: Duckietown Platform, ROS Framework, Deep Learning, Convolutional Neural Networks, and Real-Time Object Detection Constraints, to serve as a fundamental understanding before delving into the thesis.

2.1 Duckietown Platform

Duckietown is an open-source, educational, and research platform designed for autonomous driving simulation in a scaled-down manner. It consists of miniature self-driving cars, called Duckiebots, that navigate a toy-like environment with roads, traffic lights, signs, and Duckies, as pedestrians, see Fig. 2.1. According to the Duckietown website [18], the project was initiated in 2016 as a class project at the Massachusetts Institute of Technology (MIT) during the course *Autonomous Vehicles: The Duckietown Class*, led by Prof. Emilio Frazzoli and colleagues, and is currently maintained by the non-profit Duckietown Foundation with a global open-source community.

(a) (b)

Figure 2.1: Pictures of (a) a Duckiebot with a Duckie on top [18] and (b) a Duckietown environment [18].

The main contribution lies in designing and deploying a system on a Duckiebot. Computation is handled by the NVIDIA Jetson Nano 4GB, a compact and powerful AI computer that integrates both a CPU and a GPU, making it perfect for a small-scale embodied intelligent system. One important key in executing the decision process is robot perception, relying on two components: a Time-of-Flight

(ToF) sensor and a camera. A ToF sensor provides precise depth measurement without RGB information; in contrast, a camera gives RGB features yet lacks depth estimation.

2.2 Robot Operating System Framework

As previously mentioned, ROS is the principal programming framework and library for operating a robot. The ROS version used in this project is Noetic Ninjemys, which was released on May 23rd, 2020, with its end of life (EOL) on May 31th, 2025, as stated in the official ROS Noetic announcement [68]. Although the software has reached EOL, this project continues with the same version to ensure consistency.

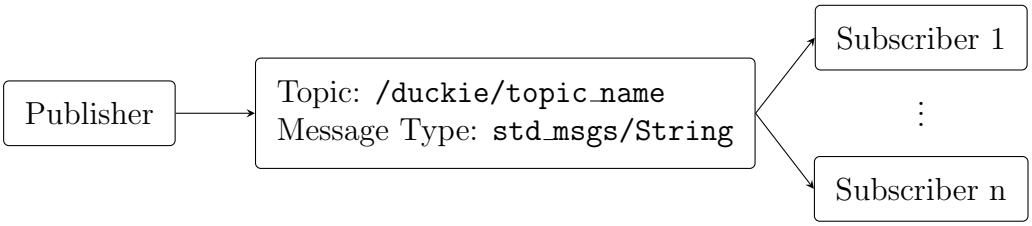


Figure 2.2: Illustration of simple publisher and subscriber nodes in ROS.

As seen in Fig. 2.2, a publisher node sends a message through a channel, called topic, with a name of `/duckiebot/topic_name` and message type of `std_msgs/String` for unidirectional data streaming between ROS nodes – in this case, they are subscriber 1, ..., n.

2.3 Deep Learning

As mentioned in [86], deep learning is a subset of machine learning that uses artificial neural networks to analyze data and make predictions, and was created to imitate human intelligence by applying multiple fundamental computational units called neurons forming a neural network similar to a human brain. It predicts output through the forward propagation process, and updates its parameters through back propagation based on its loss function – it can be perceived as a teacher correcting a student's misunderstanding. This chapter is structured into six subsections: model architecture; weight initialization; training configuration; forward propagation; backward propagation; and regularization and training techniques.

2.3.1 Model Architecture

In deep learning, there are various architectures designed to solve different tasks. Nevertheless, the most fundamental architecture is the multilayer perceptron (MLP),

consisting of three parts: input layer, hidden layers, and output layer – each layer is formed by multiple computational units called neurons [86].

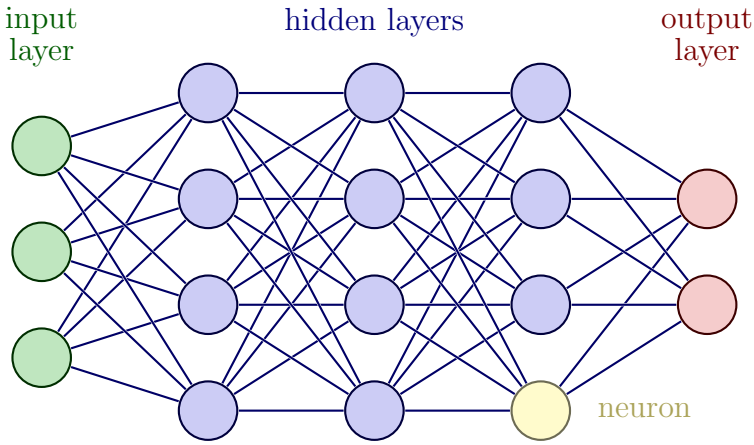


Figure 2.3: Illustration of a simplified MLP consisting of input layer, hidden layers, and output layer; the computational unit (in yellow) is called neuron. Adapted from [78].

As visualized in Fig. 2.3, each layer in a neural network is connected through weights and biases, which can be seen as a linear projection from one layer to the next. In deep learning, there are two types of variables: parameters and hyperparameters. Parameters (such as weights and biases) are learned and updated through training. On the contrary, hyperparameters (like learning rate, batch size, or number of layers) are defined before training begins. The weight and bias are continuously updated until a model's performance stabilizes [86].

2.3.2 Weight Initialization

Before training begins, all weights and biases must be initialized. There are several standard techniques, each designed to serve a different purpose. The most widely used methods are random initialization, Xavier initialization [26], and He initialization [31].

Random initialization It is the most common way to initialize weights. It samples weights randomly from a normal distribution independently.

$$\mu(w) = 0 \quad \text{and} \quad Var(w) \sim N(0, I)$$

Xavier initialization [26] This initialization, also called Glorot initialization, is designed to avoid gradient explosion or vanishing gradient for sigmoid or tanh acti-

vation functions.

$$\mu(w) = 0 \quad \text{and} \quad Var(w) \sim N(0, \sigma^2) \; ; \; \sigma = \sqrt{\frac{2}{dim_{in} + dim_{out}}}$$

He initialization [31] This technique is similar to Xavier initialization [26], but it can prevent gradient problems for ReLU rather than sigmoid or tanh activation functions.

$$\mu(w) = 0 \quad \text{and} \quad Var(w) \sim N(0, \sigma^2) \; ; \; \sigma = \sqrt{\frac{2}{dim_{in}}}$$

2.3.3 Training Configuration

Training configuration refers to hyperparameters which need to be defined before training, such as learning rate, batch size, number of epochs, optimizer, and loss function [86]. This section will describe only learning rate, batch size, and number of epochs, while optimizer and loss function will be discussed in section 2.3.5.

Learning rate This parameter defines how fast the model learns by being the multiplier in the parameters update equation $\theta_{t+1} = \theta_t - \eta \cdot V$, for θ, η, V representing the model's parameters, learning rate, and updating value, which will be discussed in the same section as optimizer and loss function. The model is considered converged when its loss function is small enough – the local/global minima of the function. Fig. 2.4 depicts simple, emulated loss function graphs, with red dots demonstrating how the values of the loss function flow. The model never knows how the loss function graph looks like because of the non-linear property of deep learning, so defining learning rate, as well as other hyperparameters, needs further adjustment and training multiple times to obtain the best result.

Epochs Epochs (or "number of epochs") refer to the times the entire training dataset is passed through the model during training. Too many epochs can cause a model to overfit—losing generalization—while too few epochs may prevent the model from converging.

Batch size It controls how much data is used in each weight update during training. Smaller batch sizes introduce more gradient noise, which can help with generalization and provide more frequent updates. In comparison, larger batch sizes reduce gradient noise and make better use of GPU parallelism, leading to faster computation but potentially weaker generalization. An example of batching a full dataset into three batches is shown in Fig. 2.5.

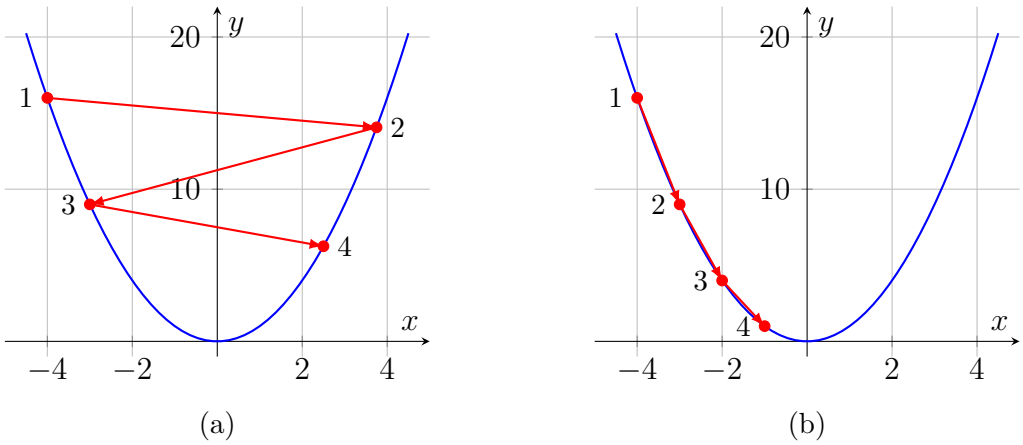


Figure 2.4: Effect of learning rate on convergence: (a) with a learning rate that is too large, the loss oscillates and fails to converge; (b) with an appropriate learning rate, the loss smoothly decreases toward a local minimum.

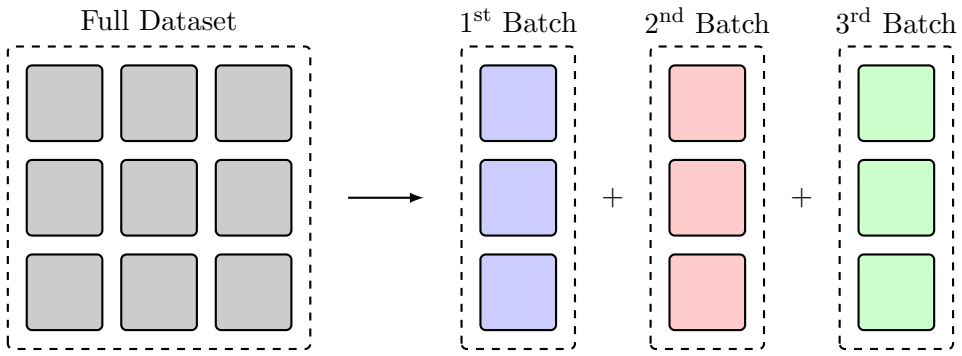


Figure 2.5: Illustration of a full dataset separated into three batches.

2.3.4 Forward Propagation

Forward propagation is the process of feeding input data to a neural network and producing outputs. Each layer of an MLP contains a different number of neurons, indicating a different dimension. In order to feed input data through the network, an affine transformation is required to map two dimensions. The affine transformation, defined as $f(x) = Wx + b$, consists of a linear transformation, $f(x) = Wx$, followed by a translation (shift by b). In addition to mapping to two dimensions, an activation function (σ) is applied to create non-linearity.

An activation function is an important key that allows a deep network to learn complex real-world tasks through its non-linearity property [86]. There are three main types of activation functions: Rectified Linear Unit (ReLU), Sigmoid, and hyperbolic tangent.

ReLU ReLU is the most widely used activation function for modern deep learning and artificial intelligence applications. It introduces non-linearity from its piecewise linear function, defined as $f(x) = \max(0, x)$. Unlike saturating functions such as sigmoid or tanh, ReLU allows unbounded positive output, which enables strong gradient flow for positive inputs, resulting in a faster and more effective training method. A graph of ReLU and its derivative is visualized in Fig. 2.6.

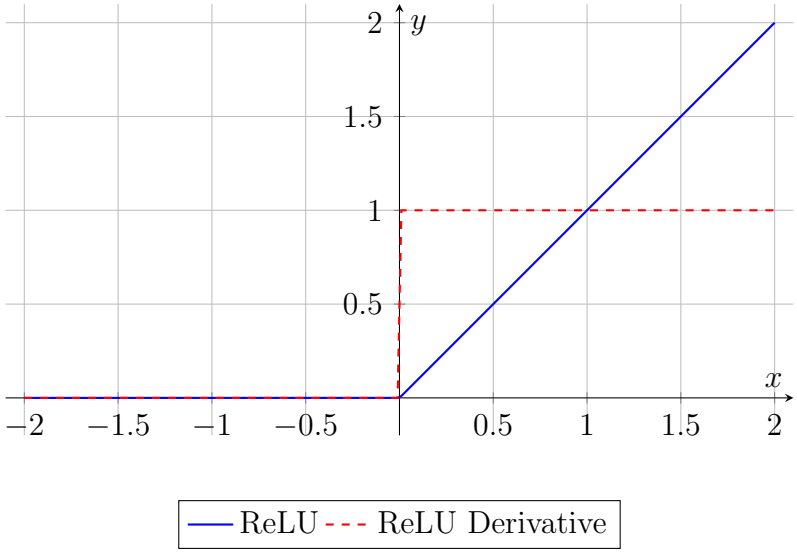


Figure 2.6: Graph of ReLU and its derivative.

Sigmoid This function is an S-shaped function ranging between 0 and 1, defined as $f(x) = \frac{1}{1+e^{-x}}$. It serves to map input features into probabilistic outputs. Thus, it is usually only used at the prediction or classification layer to predict which class targets belong. The sigmoid function and its derivative are visualized in Fig. 2.7.

Hyperbolic tangent Hyperbolic tangent or tanh, defined as $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, has a similar S-shaped curve, like the sigmoid function. However, its slope is steeper, allowing more gradient flow, which can avoid vanishing gradient in a larger deep network, such as a large language model (LLM). However, unlike the sigmoid function, it is not an ideal option for a classification task because of its output range between -1 and 1. Fig. 2.8 illustrates the hyperbolic tangent function and its derivative.

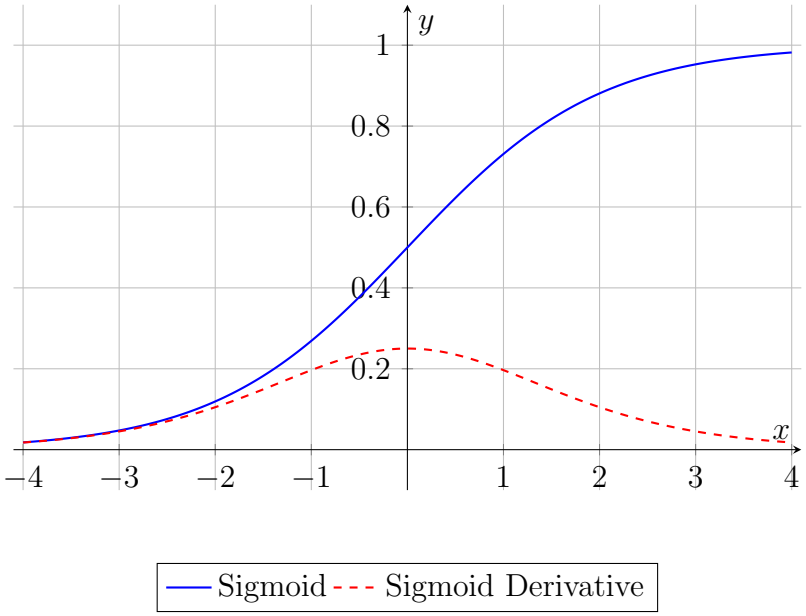


Figure 2.7: Graph of sigmoid and its derivative.

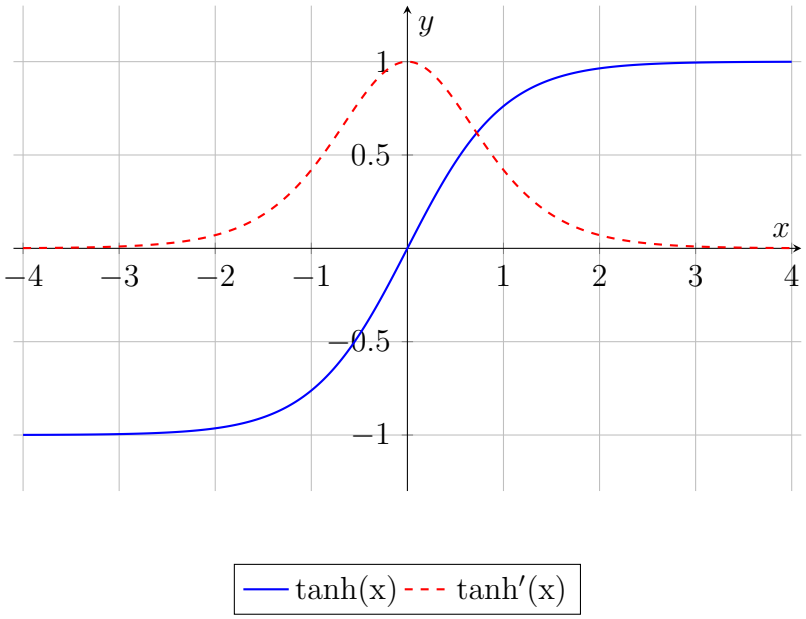


Figure 2.8: Graph of hyperbolic tangent and its derivative.

2.3.5 Backward Propagation and Training Mechanics

Backward propagation is when the model's predicted output is compared with the desired output, and the result is propagated throughout the model to update weights and biases [70]. For simplicity, this section is divided into three parts: loss function, gradient descent, and optimization.

Loss Function

As aforementioned, the predicted output of a model is compared with the expected output at the beginning of the backward propagation phase. The comparison policy is called **loss function**, which we aim to minimize. To avoid misunderstanding, the model's accuracy is not the same as the loss function. Minimizing the loss function helps the model converge, leading to more stable training. In contrast, accuracy measures how well the model performs, typically evaluated after the model has been trained by minimizing the loss. Among the many loss functions used in deep learning [76], three of the most common are mean absolute error (MAE), mean squared error (MSE), and cross-entropy (CE).

MAE loss The function, called L1 loss, is applied to minimize the error, which is the sum of all the absolute differences between the true value and the predicted value. MAE loss is applied when a dataset contains many outliers or noisy information.

$$\mathcal{L}_{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

MSE loss It, also known as L2 loss, is used to minimize the error, which is the sum of all the squared differences between the true value and the predicted value. MSE loss is the most popular for most applications because it provides a smooth gradient, allowing more gradual updates to weights and biases. Unlike MAE loss, MSE loss can barely handle outliers, since the squared value results in a large penalty, making the model harder to converge.

$$\mathcal{L}_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

CE loss This loss function, alias log loss, is used to minimize the error, which is the sum of the products of all the true probabilities and the logarithm of the predicted probabilities. The idea of CE loss is intricate, being based on information theory and maximum likelihood estimation (MLE). Nevertheless, it is considered the most frequently used function for probabilistic prediction.

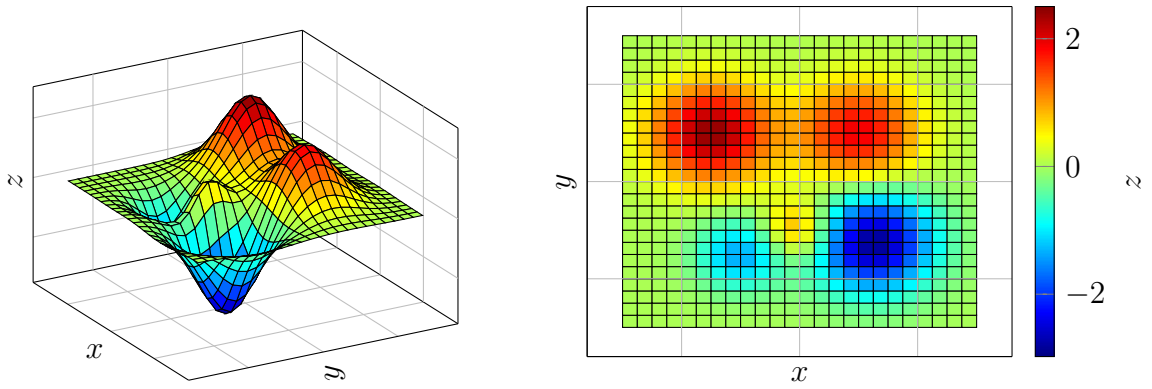
$$\mathcal{L}_{CE} = -\frac{1}{n}\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i)$$

Another notable loss function is binary cross-entropy (BCE), the specialized variant of CE loss. BCE is employed in binary classification tasks, where the objective is to distinguish between two distinct classes.

$$
\begin{aligned}
\mathcal{L}_{CE} &= -\frac{1}{n}\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i) \\
&= -\frac{1}{2}\left[y_1 \cdot \log(\hat{y}_1) + y_2 \cdot \log(\hat{y}_2)\right] \\
&= -\frac{1}{2}\left[y_1 \cdot \log(\hat{y}_1) + (1-y_1) \cdot \log(1-\hat{y}_1)\right];\ y_2 = 1 - y_1 \\
\therefore \mathcal{L}_{BCE} &= -\frac{1}{2}\left[y \cdot \log(\hat{y}) + (1-y) \cdot \log(1-\hat{y})\right];\ y_1 = y
\end{aligned}
$$

Gradient Descent

The gradient descent [69], a first-order iterative optimization algorithm, is implemented to propagate the loss value to find the minimum of a function. Fig. 2.9 shows a 3D illustration of a simplified hypothetical loss function, where the optimization process aims to reach a local or global minimum for model stability. In addition, Fig. 2.10 depicts the mathematical computation, with the forward pass as a solid line and the backward pass as a dashed line.



(a) (b)

Figure 2.9: Comparison of a simplified loss function plotted in 3D: (a) angled perspective, (b) top-down view with colorbar where red indicates high values and blue indicates low values.

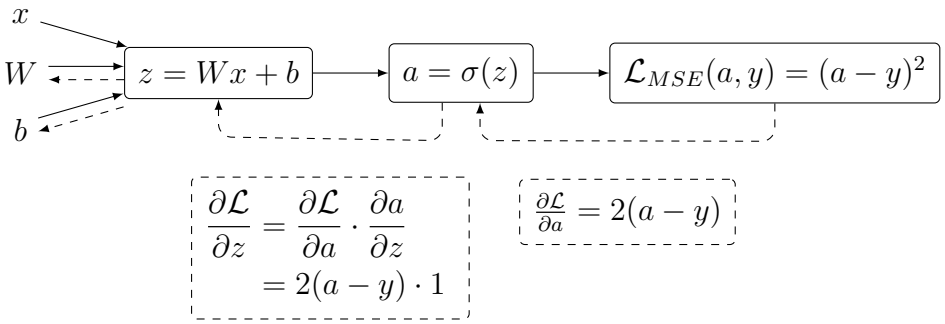


Figure 2.10: Illustration of the simplified deep learning process, where solid lines represent forward propagation and dashed lines represent backpropagation. The function $\sigma(z)$ denotes ReLU(z), with $z \geq 0$.

The following is an example of weight and bias update with ReLU ($f(x) = \max(0, x)$) during back propagation.

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial W} \qquad \frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial b}$$
$$= 2(a - y) \cdot 1 \cdot x \qquad = 2(a - y) \cdot 1 \cdot 1$$
$$\therefore \frac{\partial \mathcal{L}}{\partial W} = 2x(a - y) \qquad \therefore \frac{\partial \mathcal{L}}{\partial b} = 2(a - y)$$

In simple terms, the update of weights and biases can be discerned as follows – for η is the model's learning rate (how fast the model can learn):

$$w_{t+1} = w_t - \eta \frac{\partial \mathcal{L}}{\partial w_t} \qquad b_{t+1} = b_t - \eta \frac{\partial \mathcal{L}}{\partial b_t} \tag{1}$$

The equation (1) can also be compressed into a general form of θ, where θ are parameters of the model and $J_{\text{total}}(\theta)$ is the loss over all training examples. The following equation is the compressed version called Batch Gradient Descent [69].

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta J_{\text{total}}(\theta) \tag{2}$$

Optimizer

As illustrated in equation (1), both weights and biases get updated until the training ends. However, the naive strategy is inefficient, making finding local/global minima harder. There are five fundamental optimizers: Gradient Descent (Batch Gradient Descent) [69], Stochastic Gradient Descent (SGD) [66], Momentum [59], Root Mean Square Propagation (RMSprop) [32], Adam [40]. The batch gradient descent has already been mentioned in equation (2), which inherently combines all losses for

each batch before completing backward propagation, and SGD [66] is the variant of batch gradient descent by applying backward propagation every batch in lieu of aggregating and propagating in one shot.

Momentum [59] It is introduced to solve SGD [66]'s instability and slowness by employing the idea of a velocity vector. Instead of using the current gradient, the history of past gradients is applied.

$$\begin{aligned} v_t &= \gamma v_{t-1} + (1-\gamma)\nabla_\theta J(\theta) \\ \theta &= \theta - \eta v_t \end{aligned}$$

RMSprop [32] This technique applies gradients with different magnitudes, especially in recurrent neural networks (RNNs) or sparse features, and introduces adaptive learning rates per parameter. RMSprop [32] solves the gradient fluctuation problem: parameters with large gradients get scaled down, and parameters with small gradients get relatively larger updates.

$$\begin{aligned} E[g^2]_t &= \beta E[g^2]_{t-1} + (1-\beta)(\nabla_\theta J(\theta))^2 \\ \theta &= \theta - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \nabla_\theta J(\theta) \end{aligned}$$

Adam [40] Adam [40] combines both momentum and RMSprop [32], introducing bias correction. The bias correction of m_t and v_t helps training at an early stage to have a greater update because they have not yet accumulated meaningful gradient history. Hence, Adam [40] solves all fundamental problems, making it one of the most popular optimizers for modern deep learning tasks.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta J(\theta) \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta J(\theta))^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\ \theta &= \theta - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \end{aligned}$$

2.3.6 Regularization and Training Techniques

All previously mentioned hyperparameters can vary the performance in deep learning training; however, overfitting is one problem that hinders researchers from converging on models while keeping generalized performance. Thus, this section explains some practical, modern, and widely used techniques to avoid that problem.

Overfitting

Overfitting is a state where a model is stuck in an inescapable situation of not achieving better performance, no matter how many epochs are given. This circumstance exists because its parameters have learned too much, or learned a trivial solution—the solution that a model finds is an overly simple way to solve the training set. In contrast, this simple method is unsuitable for the test set [86]. See Fig. 2.11 for an example of different fits.

Figure 2.11: Illustration of noised input data with different fit.

Regularization

Regularization refers to techniques used during training that help a model generalize better to unseen data – by preventing overfitting, especially when learning complex patterns. Using a smaller batch size is one method to help the model generalize training data [86]. There are three main regularization methods: dropout [74], batch normalization [36], and L1/L2 regularization [51].

Dropout [74] This strategy drops certain neurons during the training process, which forces the network to not rely too heavily on any single neuron. See Fig. 2.12 for the dropout visualization.

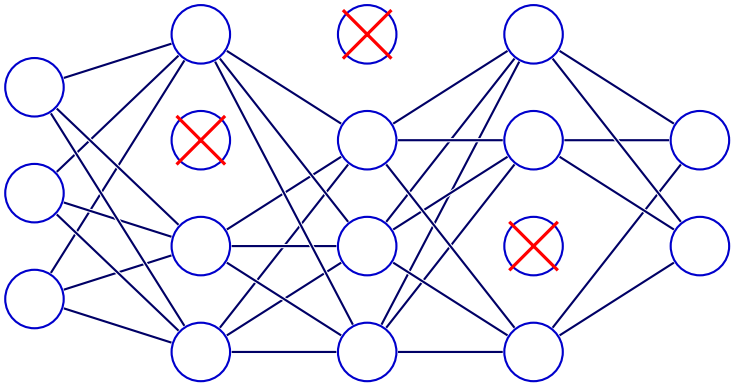


Figure 2.12: Illustration of MLP with randomly dropped neurons.

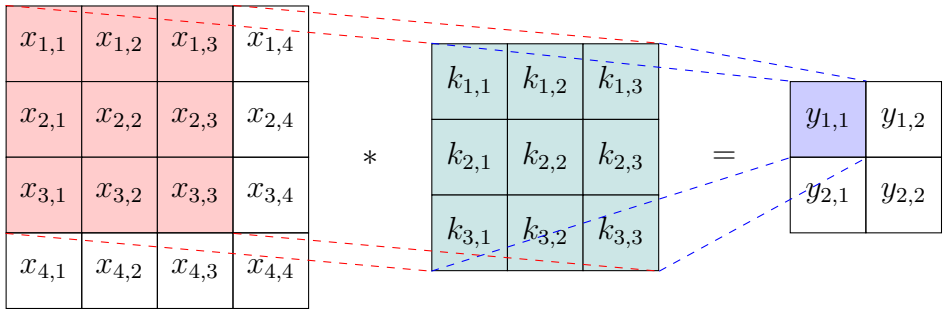
Batch normalization [36] It uses statistics (mean and variance) computed from each mini-batch, which vary across training steps. This introduces stochasticity into the network, acting as a mild regularizer. In addition, in some cases of positive input, like images, without batch normalization, all of the updates of weights that feed into a node will have the same sign, resulting in all weights being either increased or decreased, making the model learns more slowly and inefficiently. Thus, adding batch normalization is an uncomplicated method to alleviate this problem.

L1/L2 regularization [51] This method adds a penalty term to the loss function to discourage overfitting – encouraging weight sparsity in the case of L1 norm, or penalizing large weights in the case of L2 norm, also known as weight decay.

$$\mathcal{L}_{\text{total}} = \mathcal{L}(y, \hat{y}) + \underbrace{\lambda_1 \sum_{i=0}^{n} |W_i|}_{\text{L1 norm}} + \underbrace{\lambda_1 \sum_{i=0}^{n} W_i^2}_{\text{L2 norm}}$$

2.4 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a special neural network for visual pattern recognition [44]. Instead of using a naive MLP that treats all input features globally and equally, a CNN extracts local features through convolutional filters, allowing the model to learn spatial hierarchies before aggregating global information. Unlike an MLP, a CNN uses shared weights across feature maps, enabling lower computational resources. A CNN can be seen as another variant of the linear projection of shared parameters, represented as a Toeplitz matrix [71]. A simplified CNN and its associated Toeplitz matrix are shown in Fig. 2.13.



$$\begin{bmatrix} K & 0 & 0 & 0 & 0 & 0 \\ 0 & K & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & K & 0 \\ 0 & 0 & 0 & 0 & 0 & K \end{bmatrix} \cdot \begin{bmatrix} x_{1,1} \\ x_{1,2} \\ \vdots \\ x_{4,3} \\ x_{4,4} \end{bmatrix} = \begin{bmatrix} y_{1,1} \\ y_{1,2} \\ y_{2,1} \\ y_{2,2} \end{bmatrix}$$

$$\text{For } K = \begin{bmatrix} k_{1,1} & k_{1,2} & k_{1,3} & 0 & k_{2,1} & k_{2,2} & k_{2,3} & 0 & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$$

Figure 2.13: Illustration of a simplified CNN and its corresponding Toeplitz matrix.

The input features usually contain multiple channels, such as RGB data or inside an intermediate layer. The exact number of filter matrices as the inputs is required to map the input features to the next layer. While this refined version in Fig. 2.14 shows a multi-channel input with only one output channel, there are typically multiple output channels in real implementations.

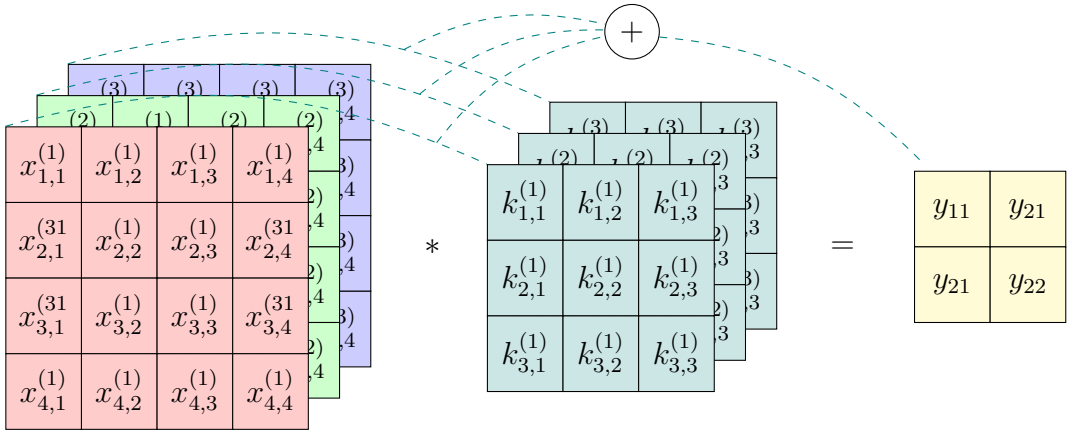


Figure 2.14: Illustration of a convolutional neural network (CNN) extracting input features from three channels into a single output channel.

For CNN training, two widely used metrics are intersection over union (IoU) and

mean average precision (mAP) [21, 47]. IoU is defined as the intersection area over the union between the predicted and ground-truth bounding boxes. The equation below shows how the mAP metric is formed.

$$\text{mAP} = \frac{1}{|C|} \sum_{c \in C} \frac{|TP_c|}{|FP_c| + |TP_c|}$$

For C is a set of all classes, TP_c and FP_c are true positive and false positive of class c, respectively. To determine whether a prediction is a TP_c or FP_c, the predicted bounding box is compared with ground truth using IoU and class label matching. If the predicted class matches the ground-truth class and the IoU is greater than a threshold (e.g., 0.6), it is counted as TP_c; otherwise, it is counted as FP_c. Widely-adopted mAP are mAP_{50} and mAP_{50-95}. mAP_{50} refers to mAP of IoU > 0.5. While mAP_{50-95} is computed from $\frac{1}{N} \sum_{\tau \in \{0.05, 0.55, \ldots, 0.95\}} \text{AP@IoU} = \tau$.

For most object detection models, three loss components are commonly used: box loss, objectness loss, and classification loss [76]. Since classification loss is discussed in the previous section, the following focuses only on box and objectness loss.

Box loss It computes how well the predicted bounding boxes fit the ground-truth bounding boxes regardless of the predicted class.

$$\begin{aligned}
\mathcal{L}_{\text{box}} &= 1 - \text{CIoU}(\mathbf{b_p}, \mathbf{b_{gt}}) \\
\text{where} \quad \text{CIoU} &= \text{IoU} - \frac{\rho^2(\mathbf{b_p}, \mathbf{b_{gt}})}{c^2} - \alpha v, \\
v &= \frac{4}{\pi^2} \left(\arctan \frac{w_{gt}}{h_{gt}} - \arctan \frac{w_p}{h_p} \right)^2, \\
\alpha &= \frac{v}{(1 - \text{IoU}) + v}
\end{aligned}$$

where $\rho(\mathbf{b_p}, \mathbf{b_{gt}})$ is Euclidean distance between box centers, c is diagonal length of the enclosing box, v is aspect ratio consistency term, and α is weighting factor.

Objectness loss It measures how well the model predicts whether a bounding box contains an object or not.

$$\begin{aligned}
\mathcal{L}_{obj} &= \frac{1}{N} \sum_{i=1}^{N} \mathbf{BCE}(o_i, t_i), \\
t_i &= \begin{cases} \text{IoU}(\mathbf{b_p}, \mathbf{b_{gt}}), & \text{if } i \text{ is a positive anchor,} \\ 0, & \text{if } i \text{ is a negative anchor.} \end{cases}
\end{aligned}$$

where o_i is predicted objectness score and t_i is target objectness score.

2.5 Real-Time Object Detection Constraints

In a real-time object detection system, there are some constraints for the system design: limited onboard computational resources, prediction latency, low camera resolution, and the speed and accuracy trade-off [34, 58].

Some constraints are obviously unavoidable, such as limited onboard computational resources or low camera resolution. Nonetheless, a low camera resolution can be alleviated by training a more sophisticated model, yet traded with more computational resources required [23, 52, 80].

3 Related Work

Existing object detection can be categorized into two types: two-stage and single-stage object detection.

Two-Stage Object Detection Two-stage object detection first extracts feature maps from the input image, then the Region Proposal Network (RPN) is employed to select anchors that are most likely to contain objects. These proposals are further refined and classified in the second stage to produce the final predictions. Two-stage object detection is the pioneering object detection algorithm to use the convolutional neural network (CNN), and was first introduced in R-CNN [25] by applying selective search to candidate proposals and Support Vector Machines (SVMs) to classify. Fast R-CNN [24] attempts to alleviate computational cost from excessive CNN by passing the entire input image once through CNN, then applying selective search on a shared feature map. Faster R-CNN [65] introduces RPN to replace selective search with learnable parameters, allowing more generalization. More modern methods [7, 89, 57] adapt the two-stage framework with advanced modules to achieve higher precision and efficiency. Despite their high precision, these methods are computationally expensive and slower due to the second stage.

Single-Stage Object Detection As two-stage detection tolerates its trade-off between speed and precision, single-stage object detection seeks to remove a bottleneck from RPN and enhance its performance through the refinement of the backbone architecture. One of the most prominent single-stage object detection models, OverFeat [72], adopts a backbone from AlexNet [42] by extending the classification and bounding box regression head. YOLOv1 [61] directly processes a feature map to generate final predictions in a single array, and SSD [48] integrates VGG-16 [73] backbone with anchors to create multi-scale single-stage object detection. Modern attempts [15, 8, 75] introduce novel architectures and training strategies that improve accuracy, and modern YOLO models: YOLOv8-YOLOv12 [64, 85, 82, 38, 77], are designed for both real-time and large-scale detection tasks. Regardless of the lack of performance in the former single-stage models, recent models demonstrate a prominent capability to deal with the trade-off.

Preference for Duckietown Applications In this project, we prioritize a real-time system by selecting single-stage models because each model is deployed on less powerful hardware, unlike full-scale autonomous vehicles that can utilize multiple

high-performance GPUs. Hence, we test YOLOv8-YOLOv12 [64, 85, 82, 38, 77], and evaluate them regarding their precision and weight, while selecting some candidates to deploy in a Duckiebot to investigate their speed to ensure the feasibility in a real-time system.

4 Approach

This chapter introduces and explains approaches for an object detection system, as well as investigates their feasibility.

4.1 Analysis of Object Detection Systems

The main goal of this thesis is to implement a system that can detect and identify interested objects in a real-time manner. This project has two onboard sensors: a ToF sensor and a camera sensor, leading to different algorithmic designs for an object detection system.

4.1.1 Problems of the Existing Object Detection Systems

The ToF sensor-based detection captures precise, sparse spatial information without RGB data, and publishes to a `front_center_tof_driver_node/range`. The Obstacle Detection Node later performs a simple logic condition: stop if the sensor captures something within 20 cm, and publish the result as `obstacle_detection_node/obstacle_detected`, then subscribed by the Lane Controller Node to operate the lane following system. However, according to the ToF sensor evaluation from [43], it reflects that its distance measurement preciseness is acceptable ($< 5\%$ relative error) only when an object is within 40 cm between field of view (FOV) of -5° and 5° and 40–50 cm in front of the bot's camera. Moreover, the ToF sensor can only detect objects within the FOV of 25°, and no dataset for the object detection task exists whose input is from the ToF sensor.

Conversely, visual-based detection proposed by [43] uses a camera, and attempts to solve the limited range of the ToF sensor by introducing a dynamic region of interest, allowing a Duckiebot to distinguish a Duckie and a yellow road center line strip. However, it cannot recognize other objects in the Duckietown environment, such as a Duckiebot and a stop sign.

4.1.2 Approach to the Object Detection Systems

Following the previously mentioned problems, this project proposes a new idea by adopting CNN-based object detection. This method can detect objects in a more fixed classes manner, meaning that a CNN-based model needs to be trained to understand what kind of objects should be detected, see 2.3 and 2.4 for more explanation.

Regarding a CNN-based method, we select YOLOv8-YOLOv12 [64, 85, 82, 38, 77] to be our case studies for our Duckietown environment because of two reasons: 1. these models are trained on the COCO [47] dataset, large-scale of 330K images, resulting in their well-prepared weights for fine-tuning and 2. they are maintained and supported by Ultralytics [79] and ONNX [14] – see Section 4.3 for libraries analysis, making these models perfect for deploying in real-world environments. The following are succinct explanations for each model.

YOLOv8 [64] YOLOv8 introduces the first unified backbone framework, which means main tasks, including object detection, image segmentation, or pose estimation, unlike other previous YOLO models [61, 62, 63, 5, 37, 45, 83] that can only perform object detection.

YOLOv9 [85] YOLOv9 proposes the Programmable Gradient Information (PGI) and Generalized Efficient Layer Aggregation Network (GELAN). The former introduces auxiliary representation to help the model prevent information loss, while the latter combines CSPNet [84], ensuring gradient path, and ELAN [83], encouraging deeper aggregation.

YOLOv10 [82] YOLOv10 employs Consistent Dual Assignments during training, providing clean predictions without the Non-Max Suppression (NMS). This drastically lowers inference latency and simplifies the deployment workflow. Additionally, the model architecture has been improved from top to bottom using holistic efficiency-accuracy driven design to obtain a balance trade-off between efficiency – inference time – and accuracy.

YOLOv11 [38] YOLOv11 attempts to improve a model backbone by enhancing feature extraction capabilities, allowing the model to handle more complex tasks, while using fewer parameters.

YOLOv12 [77] YOLOv12 presents Area Attention, which focuses on specific areas from feature maps, allowing lower computational consumption while maintaining strong feature representation, unlike general attention mechanisms that compute all possible connections. The work also introduces Residual Efficient Layer Aggregation Networks (R-ELAN), which address the vanishing gradient problem while simultaneously preserving the deep network.

All models are considered recent and well-developed. Moreover, whether a model is a strong candidate or not does not only depend on the evidence from COCO [47] evaluation but also on many factors: fine-tuning dataset or training setup, since each model's architecture is dissimilar, the final result tested on our environment can be different from the result from COCO [47].

4.2 Analysis of Dataset

There are two main ways to train a model in deep learning: unsupervised and supervised learning. Examples of unsupervised learning are Generative Adversarial Networks [27] (GANs) or Autoencoders [41], which specialize in finding patterns and generating creative output. Supervised learning has many applications, including object detection, classification, and image segmentation. While unsupervised learning can be trained without labeled data, it cannot explicitly perform the prediction of the fixed detection of detection classes, unlike supervised learning where we apply a loss function to labeled and predicted output to update weights and biases. So, this section focuses on the analysis of the dataset.

One of the most important factors for supervised learning to indicate how well the performance of the model would be is a dataset, including training, validation, and testing sets; however, generally only the performance of the testing set is shown in the report. Additionally, the two most essential factors in selecting a dataset are the size and quality. While the size of a dataset can intuitively imply how much information a model has to learn, the quality of a dataset can describe in many ways, in computer vision, for example, blurriness and brightness of images, or how generalized data is collected. Too generalized data can adapt to any environment but may not perform well, while too specialized data can perform well in the environment where it is collected but is unlikely to perform well in other environments. In this thesis, we select an open-source dataset from Duckietown [19] to train our model, which is created explicitly for object detection, containing input images and output annotations as labeled data. For the analysis of the open-source dataset, we categorize it into two parts: the measurable and immeasurable characteristics of the open-source dataset.

Measurable characteristics The measurable characteristics include the number of image-annotation pairs and the number of objects for each class. In the open-source dataset, there are 1824 image-annotation pairs which is enough to train an object detection model [67], and seven classes with their counts: Duckie (10381), Duckiebot (552), intersection sign (1555), QR code (3973), signal sign (714), stop sign (468), and traffic light (493). While intersection sign, QR code, signal sign, and traffic light are not the main concern for this project, the large number of Duckies shows how crucial Duckies or pedestrians are to be detected in the real-world environment. Nevertheless, the imperturbable classes are still used to train and prepare a fully trained model in any Duckietown environment in case of further development.

For more refined image investigation, see Table 4.1, there are only 823 images that align with the lighting conditions in our environment – dim and bright. Conversely, despite 1001 images from the open-source dataset being less related, we use them to interpret variation and to enable more generalized learning through physical augmentation, while preparing for the future lab setup.

Table 4.1: Number of images with different lighting setup categorized by color. The final column (Match) refers whether the lighting condition matches our environment.

Color	Counts	Match
Red	184	No (1001)
Blue	146	
Green	315	
Dark	356	
Dim	416	Yes (823)
Bright	407	
Total	1824	

Immeasurable characteristics For immeasurable characteristics, two things might affect a model's fidelity: the Duckiebot's design and the environment setup. The Duckiebot from the open-source dataset, see Fig. 4.1, has a totally different design, especially color, which can adversely affect our environment, since a CNN learns to recognize patterns such as color and design. For the environment setup, the size or complexity of the environment can also have impact on the performance. Training a CNN with diverse data from the open-source dataset can be beneficial. However, we must be aware that if the model's capability is unstatisfactory, one factor can be the different environment dataset.

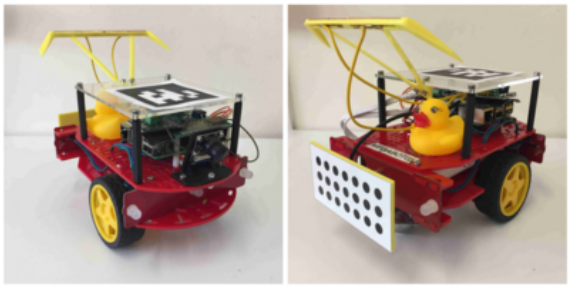


Figure 4.1: Front and rear views of a red Duckiebot [16].

4.2.1 Problems of the Open-Source Duckietown Dataset

As mentioned in Section 4.2, the primary concern is the difference between the training dataset, which is the open-source dataset, and our lab environment. There are several measures to solve the problem: gathering a new dataset for training and testing, or fine-tuning the model trained with the open-source dataset with our custom dataset. However, both methods take too much manual effort, making them

unsuitable for this thesis. Hence, this thesis decides to collect only a small testing dataset as a simple evaluation, while maintaining the diversity of the dataset.

4.2.2 Approach to the Evaluation Dataset

Since the open-source dataset differs from our setup, this thesis proposes an idea of collecting a custom dataset. The custom dataset is only applied for evaluation because collecting a custom dataset for other tasks, like training, might require more than 1000 images to convince a model to understand our environment, as it has learned from the open-source dataset, which is a time-consuming task. We also aim to use around 100 image-annotation pairs as a simple evaluation dataset.

The custom dataset is collected through a Duckiebot's camera by capturing an image every 1 second. This method allows us to continuously obtain new images without physically interacting with the Duckiebot. Despite the method's simplicity, it collects an excessive amount of images; hence, we develop an algorithm to group coherent images and pick a fixed number of candidates, see Section 5.2. Then, we use a pseudo-labeling method by using an initially trained model to generate labels on new data, helping reduce the effort of manual labeling.

4.3 Analysis of Hardware

This project uses the NVIDIA Jetson Nano 4GB for computation. According to [53], it integrates an NVIDIA Maxwell GPU with 128 CUDA cores and a quad-core ARM Cortex-A57 MPCore CPU. As reported in [42], GPUs are generally preferred for deep learning tasks because they can execute highly parallel operations more efficiently than CPUs, leading to significant acceleration in training and inference. In addition, according to [11], the GPU outperforms the CPU regarding instruction throughput and memory bandwidth while maintaining a similar price and power envelope. The GPU and CPU have different capabilities because they are designed to achieve different goals. While the CPU is designed to excel at executing a sequence of operations known as a thread as quickly as possible and can run a few tens of these threads in parallel, the GPU is designed to run thousands of them concurrently.

Besides theoretical aspects, the basic Duckiebot's system solely utilizes the CPU to operate, and our team has also heavily relied on the CPU for researching and developing ADS, see Table 6.8 and 6.12 for details on the CPU and GPU evaluation. For this reason, this project aims to utilize the GPU.

4.4 Analysis and Selection of Deep Learning Libraries and Frameworks

For GPU applications in the deep learning world, as mentioned in [12], many widely-used deep learning frameworks, such as PyTorch [55], TensorFlow [2], and Jax [6], rely on GPU-accelerated libraries provided by NVIDIA. Regardless of those prominent libraries, the NVIDIA Jetson Nano 4GB only supports small ones, which information can be found on [20]. Also, upon utilizing NVIDIA Jetson Nano 4GB, we need to consider the compatibility with Duckietown software, Duckietown-certified versions can be found in [17]. This leads to the selection of some deep learning frameworks/libraries supported by Duckietown, including PyTorch [55], TensorFlow [2], ONNX Runtime [14], and TensorRT [13], of which all are described in the following sections.

Before beginning the library analysis, there are two main parts during model run time: inference and processing. The inference step is where a deep learning model is trained and used, and it relies on different libraries depending on the usage purpose. For example, PyTorch [55] or TensorFlow [2] are used for training or simple deployment, but ONNX Runtime [14] or TensorRT [13] are employed for efficient deployment for real-time systems. Contrarily, in the processing phase, including preprocessing, where a system prepares data for the inference, and postprocessing, where raw prediction from the inference is processed to get the desired format, relies on libraries that allow a user to flexibly perform matrix operations such as NumPy [30], PyTorch [55], and TensorFlow [2]. The following part describes the advantages and disadvantages of each of the earlier-mentioned libraries, and their limitation specifically for this project.

PyTorch [55] PyTorch [55] is one of the most widely used deep learning frameworks because it seamlessly integrates with CPU, GPU, or Metal Performance Shaders (MPS). Also, PyTorch provides rich computer vision tools, like TorchVision [49], that allow a user to edit and process images before deep learning operations, and can be used to form a complicated deep network design. Our project faced a peculiar problem while using PyTorch on a GPU with ONNX Runtime [14]. Although there is no conflict between PyTorch and ONNX Runtime, the model appeared to hang during loading in ONNX Runtime on the GPU, taking forever without producing an error. So, we only stick to PyTorch on CPU.

TensorFlow [2] TensorFlow was created by Google as an open-source framework for machine learning. It supports deep learning, neural networks, and general numerical computations, similar to PyTorch [55], that can be operated on various computational hardware. Nevertheless, its lower popularity compared to PyTorch [55], as shown by Google Trends [28], likely contributes to slower development and fewer

available features, given the smaller community and industry support. Also, TensorFlow conflicts with the Duckietown setup. Therefore, we avoid TensorFlow [2] in this project.

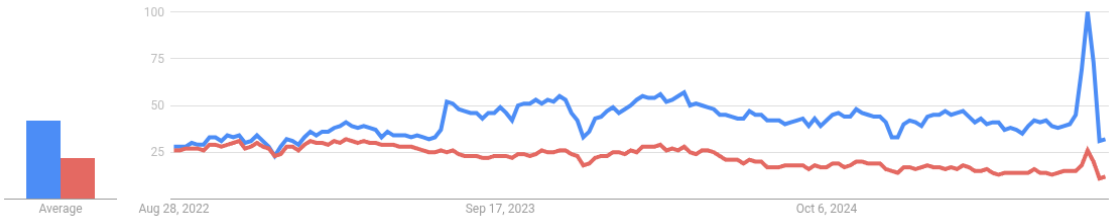


Figure 4.2: Graphs of the popularity of PyTorch [55] (blue) and TensorFlow [2] (red) from September 1, 2022 to September 1, 2025. Obtained from Google Trends [28].

NumPy [30] NumPy is commonly used for CPU array operations such as pre- and postprocessing. Moreover, from the Stack Overflow 2024 survey [54], NumPy [30] is ranked in the Top-3 for most used Python libraries and Top-1 for programming learners. Therefore, it is a top choice for a simple and accessible library for anyone.

ONNX Runtime [14] ONNX Runtime is a high-performance inference engine for running models stored in ONNX [22] format, an open standard file format for representing machine learning models. It parses a model into a directed acyclic graph (DAG) representation and optimizes a model according to the execution providers, for example, CPU or GPU via CUDA – a low-level parallel computing platform and programming model designed for direct communication with NVIDIA GPUs. In the implementation, we use Ultralytics [79] to load the model and fine-tune it, then convert it into ONNX [22] format and run it on ONNX Runtime. We never run it with Ultralytics [79] because the model has not been optimized, making it too heavy for our limited hardware, the NVIDIA Jetson Nano 4GB.

TensorRT [13] TensorRT is one of the most optimal options for a deep network deployment because it generates hardware-specific fused kernels tuned for the exact GPU and can run with mixed precision seamlessly, maximizing GPU throughput. Nonetheless, although the inference model is unchanged for any libraries or frameworks, we found that the raw output prediction results differ from the original model. We have two hypotheses for this situation: 1. the TensorRT version is too old and therefore incompatible with the ONNX [22] version, and 2. there could be bugs in graph optimization due to the newer YOLO architecture and obsolete TensorRT version, which might require a more modern version of TensorRT. However, Duckietown does not support newer versions of TensorRT, see [17].

Eventually, the current exploration of deep learning model setups, including candidate frameworks, libraries is consolidated and visualized in Fig. 4.3.

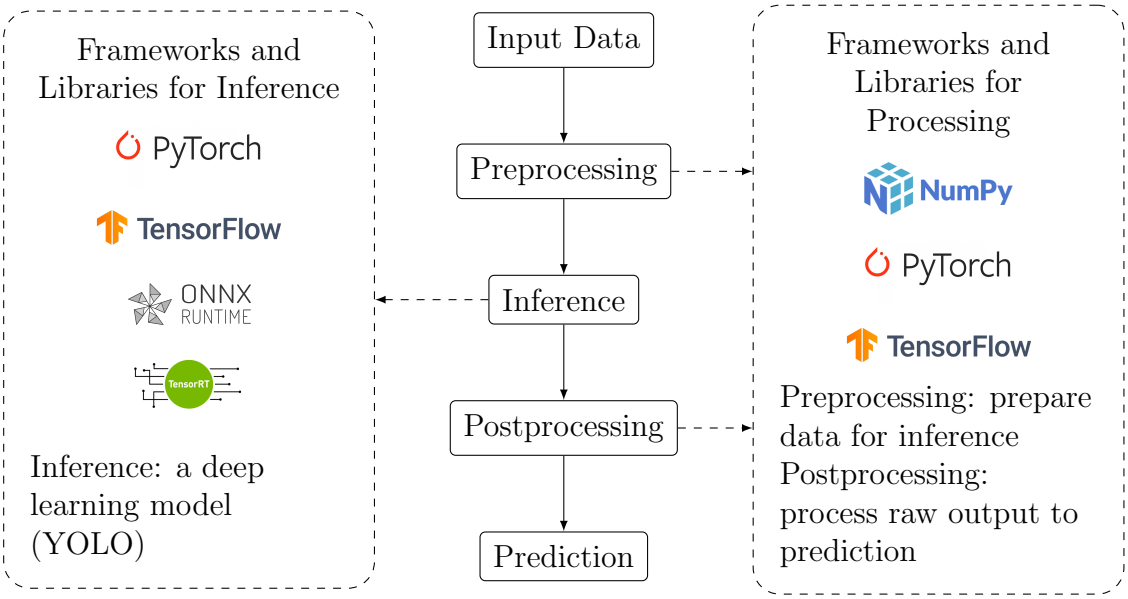


Figure 4.3: Illustration of data flow during deployment. All images are from [30, 55, 2, 14, 13].

After the full libraries analysis, we select PyTorch [55] and NumPy [30] for processing, and ONNX Runtime [14] for the inference, see Section 6 for the evaluation of each framework. During fine-tuning, we use Ultralytics [79] because it gives flexible computer vision tasks training or testing configurations. See Table 4.2 for the summary table.

Table 4.2: Comparison of Deep Learning Frameworks and Libraries

Framework/Library	Application Domain	Hardware Support	Remarks
NumPy [30]	Numerical and matrix operations	CPU	-
PyTorch [55]	Numerical computation and deep neural network modeling	CPU, GPU, limited TPU support	In our lab, only CPU setup is used due to GPU setup conflicts with ONNX Runtime
TensorFlow [2]	Numerical computation and deep neural network modeling	CPU, GPU, TPU, mobile/embedded	Less popular for model AI; conflicts with Duckietown setup
ONNX Runtime [14]	Deep neural network deployment	CPU, GPU	Lightweight runtime
TensorRT [2]	Deep neural network deployment	GPU (hardware-specific models)	Limited adoption; too obsolete for modern YOLO

In addition, we select only the nano/tiny versions of YOLOv8–YOLOv12 [64, 85, 82, 38, 77] to be fine-tuned for lightweight purposes. They are trained for 100 epochs with a learning rate of 10^{-4} and a batch size of 16, following general YOLO setups, and each model is evaluated using our custom dataset.

4.5 Approach to the Fundamental Decision Making

This work focuses on the basic implementation of the reaction to prediction results from a CNN-based model rather than a full functionality setup, like [43]. We propose a simple approach aiming to stop when a Duckiebot sees other Duckiebots or Duckies, and to wait for a period of time when seeing a stop sign before running again. Our design can be visualized with a finite state machine (FSM), see Fig. 4.4 and 4.5. To enhance realism, the system employs simple policies that utilize the location and dimensions of detected objects.

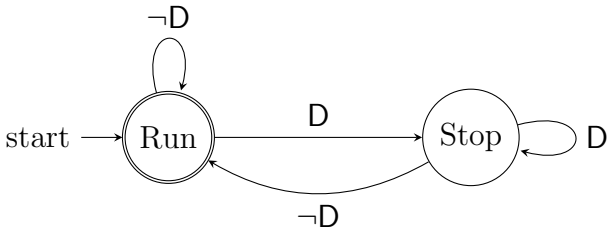


Figure 4.4: Finite state machine for Duckie/Duckiebot detection (D ≡ Duckie / Duckiebot detected).

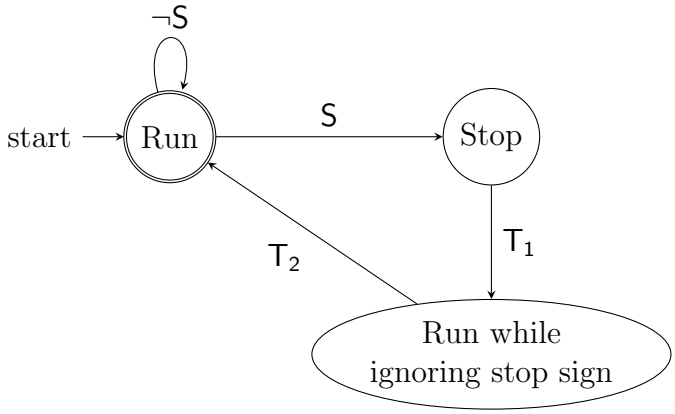


Figure 4.5: Finite state machine for stop sign detection (S ≡ stop sign detected, T_1 ≡ 3-second timer expires, T_2 ≡ 5-second timer expires.

5 Implementation

This chapter details the implementation of the main approaches: object detection system, data collection of a custom dataset, and a fundamental decision-making system.

5.1 Development of Object Detection System

This section discusses the implementation of the object detection system. The system relies on RoboFlow [67], an online platform for data labeling and other computer vision tasks; YOLO-based models, an open-source computer vision CNN-based models provided by Ultralytics [79]; Kaggle [1], an online platform for machine learning offering computational resources; and ONNX Runtime [14], an open-source inference engine with graph optimizations to improve performance.

The process starts by preparing the training/validation/testing dataset, which includes removing unannotated images and resizing images into a YOLO-compatible resolution of 640x640 pixels (commonly referred to as input size of 640). In other tasks, images can be augmented by, for example, adding noise or applying transformations (translation, rotation, or flipping), which helps generate more data from the original images. As a result, we obtained 1824 images from the previous data preparation process using the Duckietown open-source dataset [19]. Next, the nano versions of YOLOv8-YOLOv12 [64, 85, 82, 38, 77] are imported from Ultralytics [79], and prepared for the fine-tuning on Kaggle [1]. For simplicity, the training configuration of each model is set similarly by setting 100 epochs and a learning rate of 10^{-4} as a default setup recommended by Ultralytics [79]. After fine-tuning, each model is then tested with the open-source testing dataset to evaluate; in addition, each model is assessed with our custom dataset – the dataset collection is described in Section 5.2 – for the final model fidelity evaluation for our environment.

5.2 Custom Dataset

In our lab, there are two possible ways to gather images from a Duckiebot camera: 1. capturing or downloading an image manually, which is the common practice, and 2. setting up an automatic system to collect images and pull all images at once to our local PC. This project uses the second method because the first method is inefficient compared to the second method, which can capture hundreds of images at a time,

so we set the system to collect an image every second. Despite the efficiency of the second method in image capturing, it cannot distinguish images, meaning that the system does not know how similar the current image is to previous ones. Without further discrimination processes, this can lead to poor fine-tuning results. Hence, we acquire the concept of embedding from LLMs to guide the first stage of image selection, helping reduce 1000+ images into 200 images without heavy manual labor.

An embedding is a low-dimensional, numerical vector representation of data, such as words, images, or audio. It is popular in LLMs because it represents extracted information to enhance reasoning capabilities. It is obtained from the immediate feature, and generally as a form of a single-dimensional vector. A similar idea is applied in an object detection model to distinguish and select images based on their embeddings, indicating the features before their prediction layer. Inspired by [87], we process each image into YOLO embeddings followed by L2-norm, and apply K-means to find the centroid of each cluster. The only key difference is that [87] is trained to force all embeddings to be uniformly distributed, unlike YOLO and other object detection models. Instead, we rely on embeddings from a well-trained YOLO model on open-source datasets, as these provide non-trivial representations. In other words, the model's weights and biases are already well-optimized, which helps prevent poor clustering results. The visualization of our method is shown in Fig. 5.1.

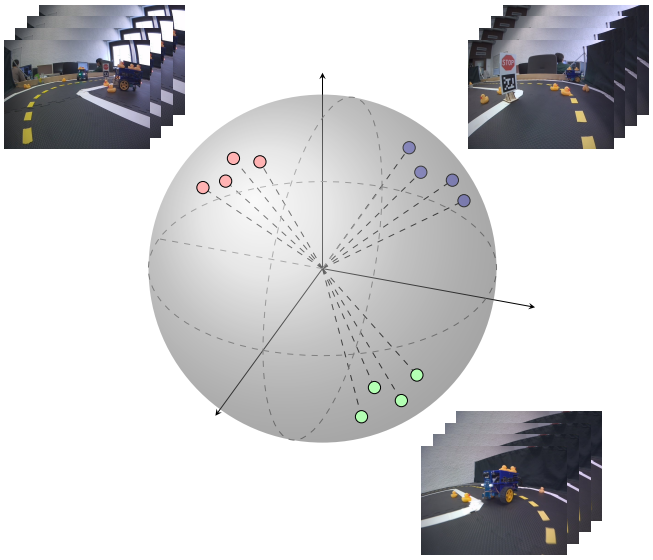


Figure 5.1: Visualization of the L_2 norm of hidden features projected onto a unit hypersphere. The figure illustrates how different images are distributed across the hypersphere.

Through this process, 119 images are obtained from 1000+ images before feeding each image into the YOLO algorithm and storing annotation results in YOLO format

as pseudo-labeling. In practice, pseudo-labeling can be used to train or test directly, but to make them more accurate, the labels are processed again in RoboFlow [67] through manual checking. Lastly, we obtain four classes with their sample counts: Duckie (477), Duckiebot (94), QR code (43), and stop sign (42).

5.3 Fundamental Decision Making

As mentioned in Section 4.5, we implement a simple algorithm that allows a Duckiebot to stop when detecting Duckies or another Duckiebot, and to wait for 3 seconds when observing a stop sign, then run again while ignoring a stop sign for 5 seconds. This allows us to perceive the basic concept of the real-world implementation of an object detection system while performing logical operations in a lightweight manner. Algorithm 1 demonstrates that the Duckiebot stops when the normalized Duckie width or height (with respect to the image width) exceeds 0.08, which indicates a potentially dangerous proximity; or when the centroid of a Duckie is located below its predefined horizontal threshold (normalized image height = 0.6), or the centroid of another Duckiebot is below its threshold (normalized image height = 0.4), indicating that the object is too close. In addition, Algorithm 2 illustrates the Duckiebot's stop-and-wait logic when detecting a stop sign. The vehicle stops if the normalized stop sign width or height, relative to the image width, exceeds 0.1. It first runs until it detects a stop sign and waits for 3 seconds. After that, it continues running again while ignoring any future stop signs for 5 seconds, helping the Duckiebot operate smoothly and imitate a real-world scenario where a car would wait and go. Fig.5.2 illustrates the threshold lines used for Duckie (yellow) and Duckiebot (blue) detection, and presents the normalized object widths and heights relative to the image width.

Algorithm 1 Duckie/Duckiebot detection decision-making algorithm (derived from Fig. 4.4). Defined image regions $\mathbf{R_D}$ and $\mathbf{R_{DB}}$ apply the rule-based method by defining a horizontal line on an image, and it returns True if an object is below the line. The CheckSize() command checks if either a normalized object width or height is over 0.08 (near to a Duckiebot's camera).

Input: Image **I**
Output: Duckiebot action **A**
Variables: Duckie dimension $\mathbf{D_d}$; Duckie centroid $\mathbf{D_c}$; Duckiebot location $\mathbf{DB_c}$; Defined image region for Duckie $\mathbf{R_D}$; Defined image region for Duckie $\mathbf{R_{DB}}$
if ($\mathbf{D_c} \in \mathbf{R_D}$ **or** $\mathbf{DB_c} \in \mathbf{R_{DB}}$) **or** CheckSize($\mathbf{D_d}$) **then**
    $\mathbf{A} \leftarrow$ stop
else
    $\mathbf{A} \leftarrow$ run
end if

Algorithm 2 Stop sign detection decision-making algorithm (derived from Fig. 4.5). The CheckSize() command checks if either a normalized object width or height is over 0.08 (near to a Duckiebot's camera).

Input: Image **I**
Output: Duckiebot action **A**
Variables: Wait flag **W**; Wait delay $\mathbf{T_1}$; Detection delay $\mathbf{T_2}$; Stop sign dimension $\mathbf{S_d}$
if $\mathbf{T_1}$ expired **then**
    W $\leftarrow$ False
end if
if ($\neg$**W**) **and** ($\mathbf{T_2}$ expired) **then**
    W $\leftarrow$ CheckSize($\mathbf{S_d}$)
    if W = True **then**
        Start timer $\mathbf{T_1}$
        Start timer $\mathbf{T_2}$
    end if
end if
if W = True **then**
    A $\leftarrow$ stop
else
    A $\leftarrow$ run
end if

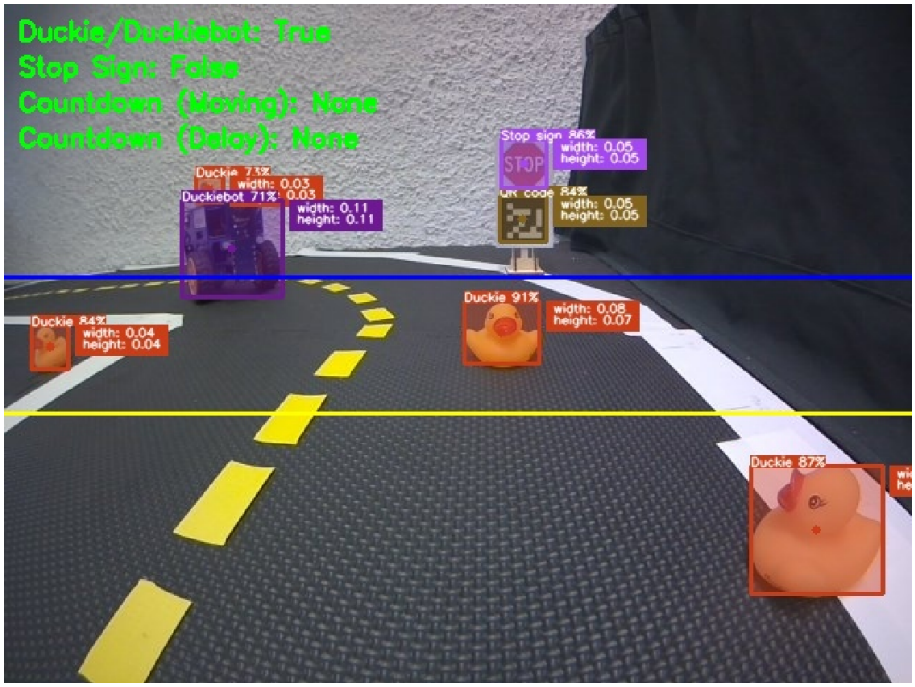


Figure 5.2: Picture of a Duckietown lane with threshold lines for Duckies (yellow) and Duckiebots (blue), along with normalized width and height relative to the image width labeled for each object.

6 Evaluation

This chapter evaluates the performance of different YOLO versions over different image resolutions in terms of mean average precision, output rate, and hardware utilization. In addition, a case study is conducted to reflect real-world scenarios.

6.1 Model Size and FLOPs

The number of parameters refers to the number of weights and biases. While a larger number of parameters can often imply better performance, it highly affects inference time. On the other hand, floating-point operations, known as FLOPs, are needed by a model to measure how heavy a model is. However, FLOPs does not imply that when a model is deployed on a GPU, which has the ability to parallelize, a model's runtime will conform to FLOPs. Hence, the following tables can only be seen as an overview of the model ponderosity.

Table 6.1: Model Parameters and FLOPs [79].

Model	Params (M)	FLOPs (B)	mAP_{50-95} on COCO [47]
YOLOv8n	3.2	8.7	37.3
YOLOv9t	2.0	7.7	38.3
YOLOv10n	2.3	6.7	38.5
YOLOv11n	2.6	6.5	39.5
YOLOv12n	2.6	6.5	40.6

6.2 Performance Comparison of YOLO Models

This project selects only YOLO nano versions from version 8 to version 12 [64, 85, 82, 38, 77] for two reasons. First, YOLO nano is the lightweight option. Another reason is that YOLO version 8 is the most widely used model due to the full support from Ultralytics [79] and its ability to operate in an industrial real-time system; in addition, the newer versions are worth investigation because they can perform better on the COCO dataset [47]. The evaluation is conducted by training all YOLO nano models from all modern versions with the open-source dataset with different input

sizes: 640, 480, 320, 160, then testing with both the open-source dataset and our custom dataset. It is remarked that YOLOv9 [85] does not have a nano version; instead, it offers an equivalent version, called a tiny version.

Table 6.2: Performance evaluation of YOLO-nano models with input size 640.

Model	Open-Source		Custom		ΔmAP$_{50}$	ΔmAP$_{50-95}$
	mAP$_{50}$	mAP$_{50-95}$	mAP$_{50}$	mAP$_{50-95}$		
YOLOv8n	90.98	58.84	**44.57**	**30.41**	-46.41	-28.43
YOLOv9t	90.12	58.05	44.34	28.40	-45.78	-29.65
YOLOv10n	90.02	58.52	43.20	28.62	-46.82	-29.90
YOLOv11n	91.02	59.49	44.36	29.57	-46.66	-29.92
YOLOv12n	**92.65**	**60.02**	40.43	24.74	-52.22	-35.28

Table 6.3: Performance evaluation of YOLO-nano models with input size 480.

Model	Open-Source		Custom		ΔmAP$_{50}$	ΔmAP$_{50-95}$
	mAP$_{50}$	mAP$_{50-95}$	mAP$_{50}$	mAP$_{50-95}$		
YOLOv8n	**91.17**	58.30	41.08	25.57	-50.09	-32.73
YOLOv9t	90.50	57.65	43.20	27.27	-47.30	-30.38
YOLOv10n	89.21	57.82	41.57	27.77	-47.64	-30.05
YOLOv11n	90.64	58.18	**45.54**	**29.50**	-45.10	-28.68
YOLOv12n	91.12	**58.68**	41.30	25.37	-49.82	-33.31

Table 6.4: Performance evaluation of YOLO-nano models with input size 320.

Model	Open-Source		Custom		ΔmAP$_{50}$	ΔmAP$_{50-95}$
	mAP$_{50}$	mAP$_{50-95}$	mAP$_{50}$	mAP$_{50-95}$		
YOLOv8n	**87.32**	**54.19**	37.89	**23.08**	-49.43	-31.11
YOLOv9t	85.99	52.67	**37.93**	21.78	-48.06	-30.89
YOLOv10n	85.58	53.55	37.06	22.73	-48.52	-30.82
YOLOv11n	85.59	53.26	37.78	22.42	-47.81	-30.84
YOLOv12n	86.14	52.53	35.19	19.94	-50.95	-32.59

Table 6.5: Performance evaluation of YOLO-nano models with input size 160.

Model	Open-Source		Custom		ΔmAP$_{50}$	ΔmAP$_{50-95}$
	mAP$_{50}$	mAP$_{50-95}$	mAP$_{50}$	mAP$_{50-95}$		
YOLOv8n	**59.77**	32.22	**24.75**	**13.62**	-35.02	-18.60
YOLOv9t	57.85	30.79	21.19	10.65	-36.66	-20.14
YOLOv10n	57.50	31.57	20.62	10.65	-36.88	-20.92
YOLOv11n	59.60	**32.95**	22.87	11.31	-36.73	-21.64
YOLOv12n	57.50	31.55	22.62	12.02	-34.88	-19.53

As seen in Table 6.2, 6.3, 6.4, 6.5, we inspect a modest performance drop from 640 to 480 input size, and a more slight descent from 480 to 320 input size for all models, while all models on 160 input resolution have the worst performance. We choose only YOLOv11n with input sizes of 640 and 480 for further evaluation because YOLOv11n on 480 input size outperforms all other models on the custom dataset, and YOLOv11n on 640 input size achieves almost a similar level to other models. In addition, this model selection helps study the impact of image size and computation performance.

6.3 YOLO Deployment Performance

Following the YOLO training performance evaluation, this section explores object detection performance while stationary and running (lane following) of YOLOv11n with two input sizes: 480 and 640. Some results are compared with a regular lane following system to understand the impact of an object detection model on hardware. Also, Table 6.7 and 6.11, showing publishing rate, use annotation instead of the real ROS topic name to avoid confusion. The annotations are Image In: `/duckiebot/camera_node/image/compressed`, a formatted image from a camera; Detection

Array: `/duckiebot/object_detection_node/detection_result/array`, a matrix containing detection results; and Car CMD: `/duckiebot/lane_controller/car_cmd`, a car command defining the linear and angular velocity of the Duckiebot. The publishing rates were measured using the ROS command `rostopic hz /duckiebot/topic_name`, while hardware performance was evaluated using the `tegrastats` utility provided by NVIDIA.

6.3.1 While Stationary

Table 6.6: Pre-, post-processing, inference, and total times are reported in milliseconds and averaged using a 30-step simple moving average while stationary. Inference may run on either CPU or GPU, as indicated, while pre- and post-processing always run on CPU regardless of the inference device.

Framework	Model	Preprocessing		Inference		Postprocessing		Total	
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
NumPy	YOLOv11n@480	50.00	53.70	1615.93	**142.67**	1.02	1.94	1666.95	**198.31**
	YOLOv11n@640	68.12	127.14	2627.56	179.10	2.34	**0.54**	2698.02	306.78
PyTorch	YOLOv11n@480	**46.56**	64.51	1761.40	150.35	25.39	13.98	1833.35	228.84
	YOLOv11n@640	62.18	107.52	2535.40	198.31	19.52	13.76	2617.10	319.59

Table 6.7: Publishing frequencies (Hz) of the primary stages in the object detection pipeline while stationary.

Framework	Model	Image In		Detection Array		Car CMD	
		CPU	GPU	CPU	GPU	CPU	GPU
No Detection		9.6		0		**14.2**	
NumPy	YOLOv11n@480	**9.8**	7.7	1.0	3.5	12.0	7.8
	YOLOv11n@640	9.0	7.7	0.5	**4.2**	10.5	11.5
PyTorch	YOLOv11n@480	7.5	7.5	0.9	4.7	8.5	11.5
	YOLOv11n@640	8.2	7.0	0.6	2.7	12.0	9.5

Table 6.8: Comparison of RAM, CPU, and GPU utilization while stationary.

Framework	Model	RAM (%)		CPU (%)		GPU (%)	
		CPU	GPU	CPU	GPU	CPU	GPU
No Detection		**40.89**		**96.48**		0	
Numpy	YOLOv11n@480	62.88	60.60	99.30	98.72	0	26.35
	YOLOv11n@640	80.80	79.78	99.38	98.05	0	29.86
PyTorch	YOLOv11n@480	43.32	79.44	99.38	98.04	0	**19.75**
	YOLOv11n@640	75.88	94.14	99.45	98.08	0	33.89

Table 6.9: Comparison of CPU and GPU temperatures while stationary.

Framework	Model	CPU Temp (°C)		GPU Temp (°C)	
		CPU	GPU	CPU	GPU
No Detection		33.64		34.43	
Numpy	YOLOv11n@480	**33.13**	33.73	33.84	**32.83**
	YOLOv11n@640	33.96	34.48	34.46	33.45
PyTorch	YOLOv11n@480	34.40	34.60	35.04	33.11
	YOLOv11n@640	33.98	37.17	34.86	35.89

6.3.2 While Running

Table 6.10: Pre-, post-processing, inference, and total times are reported in milliseconds and averaged using a 30-step simple moving average while running. Inference may run on either CPU or GPU, as indicated, while pre- and post-processing always run on CPU regardless of the inference device.

Framework	Model	Preprocessing		Inference		Postprocessing		Total	
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
NumPy	YOLOv11n@480	**40.81**	53.19	1443.25	**131.73**	2.27	1.85	1486.33	**186.77**
	YOLOv11n@640	69.00	75.55	2506.90	162.58	**1.02**	1.41	2576.92	239.54
PyTorch	YOLOv11n@480	66.05	56.16	1477.23	140.62	14.76	22.42	1558.04	219.20
	YOLOv11n@640	64.82	91.02	2261.47	189.28	12.77	7.38	2339.06	287.68

Table 6.11: Publishing frequencies (Hz) of the primary stages in the object detection pipeline while running.

Framework	Model	Image In		Detection Array		Car CMD	
		CPU	GPU	CPU	GPU	CPU	GPU
No Detection		**14.0**		0		**21.3**	
NumPy	YOLOv11n@480	13.5	10.3	0.9	**4.9**	10.5	11.2
	YOLOv11n@640	11.5	9.6	0.7	3.8	11.5	12.8
PyTorch	YOLOv11n@480	11.5	7.3	1.1	4.5	9.5	12.5
	YOLOv11n@640	10.0	8.5	0.5	3.6	12.0	12.5

Table 6.12: Comparison of RAM, CPU, and GPU utilization while running.

Framework	Model	RAM (%)		CPU (%)		GPU (%)	
		CPU	GPU	CPU	GPU	CPU	GPU
No Detection		74.85		**93.84**		0	
Numpy	YOLOv11n@480	88.04	94.01	98.97	97.96	0	31.22
	YOLOv11n@640	**42.27**	80.36	99.08	97.40	0	40.83
PyTorch	YOLOv11n@480	79.66	92.40	99.02	97.66	0	**25.97**
	YOLOv11n@640	60.49	55.87	98.89	97.50	0	34.92

Table 6.13: Comparison of CPU and GPU temperatures while running.

Framework	Model	CPU Temp (°C)		GPU Temp (°C)	
		CPU	GPU	CPU	GPU
No Detection		**33.87**		34.97	
Numpy	YOLOv11n@480	34.06	35.18	34.59	**34.37**
	YOLOv11n@640	34.79	36.15	35.37	35.19
PyTorch	YOLOv11n@480	34.85	35.69	35.59	34.57
	YOLOv11n@640	34.96	37.66	35.33	36.48

As seen from all tables, it is evident that running a model on a GPU truly outperforms a model on a CPU for total run time, especially in mode inference, for both, see Table 6.6 and 6.10. For pre- and postprocessing, the former shows that processing time at input size of 480 is likely faster than at 640 because preprocessing performs resizing on an input image, whose processing time depends on input

dimension. Postprocessing, in ideal, should execute with a similar run time, but in the PyTorch [55] case, the recorded postprocessing values are very varied, showing a sign of inferior PyTorch [55] optimization for CPU. For hardware utilization, see Table 6.8 and 6.12. In most cases, RAM utilization is expected to rise when using a GPU for inference because the GPU of the NVIDIA Jetson Nano 4GB does not have VRAM, indicating shared RAM with the whole system.

However, several aspects of the evaluation raise questions about hardware design and deep learning concepts. As seen from Table 6.6 and 6.10, even though most values are logical, as mentioned before, the inference time during running is lower than during stationary for all cases, which is counter-intuitive since running is an extra operation compared to stationary, adding more workload on the hardware. Nevertheless, our hypothesis is that when comparing Table 6.8 and 6.12 on RAM and GPU, most values from RAM and all values from GPU increase when running, which is reasonable because when the Duckiebot is running, all input images are highly dissimilar, making the GPU non-idle. This higher GPU utilization might help its pipeline run smoothly and work more consistently; also, it affects publishing frequencies for some ROS topics. Lastly, peculiar results are given from the temperature measurements. Even though it is rational that the temperature of CPU and GPU is higher during running, when looking closely at both stationary and running, the temperature of CPU is higher when a model is on CPU, while the temperature of GPU is lower. Our intuition would be that if we deploy a model on a GPU, the GPU temperature should be higher, but the results truly contradict it. Our assumption is that memory traffic is smoother when a GPU is operating, while the CPU is working harder to capture an image and feed it to a GPU. One last thing to be remarked is the publishing rate of Detection Array from Table 6.11, as a model inference runs at 1558.04 ms, the publishing rate should return as $\frac{1}{1558.04\text{ ms}} \approx 0.6$ Hz. Despite this fact, all values are recorded as average values and from different periods, as it is impossible to record all of them at the same exact time. Therefore, the reported numbers should be interpreted as approximate indicators of system behavior rather than exact synchronized measurements. Eventually, regardless of unpredictable hardware behavior, a model's performance when deployed on the GPU, with a prominently high publishing rate of the Detection Array topic, is good enough for a real-time application.

6.4 Case Study: Confidence Score Evaluation

The confidence score (conf_k) is the product of $\sigma(z_{\text{obj}})$ and $\sigma(z_{\text{cls},k})$, where $\sigma(z_{\text{obj}})$ denotes the objectness—the model's probability that a predicted box contains any object, irrespective of class—and $\sigma(z_{\text{cls},k})$ denotes the probability that the box contains class k. To obtain extra results from raw prediction, we first filter the raw results by strictly selecting results with `conf_k` greater than the confidence thresh-

old, then pick the results with the highest `conf_k` for each result. After that, all proposed results are iteratively compared with other boxes with the same class to obtain the final prediction. The comparison involved IoU computation among other boxes within the same class, and selected boxes with an IoU less than the predefined threshold. The default values of confidence score and IoU threshold are 0.25 and 0.7; however, we want to filter only strong candidates, so we set confidence score and IoU threshold to 0.5 and 0.3. This section uses the confidence score for our case study to demonstrate two evaluation scenarios: nominal and safety-critical cases.

6.4.1 Nominal Cases

Nominal case refers to cases we expect to see in a real-world situation. This section defines nominal cases as follows.

Duckie This project conducts a case study by placing Duckies with different facing directions (front, back, left, right) and placing them at different distances and angles from the center: at 25 cm: 0°, 30°, 60°; 50 and 75 cm: 0°, 15°, 30°; 100 cm: 0°, 15°. However, this project uses a top camera to map and draw desired points instead of physically measuring distance and angle. Fig. 6.1 shows the locations of the Duckies test cases.

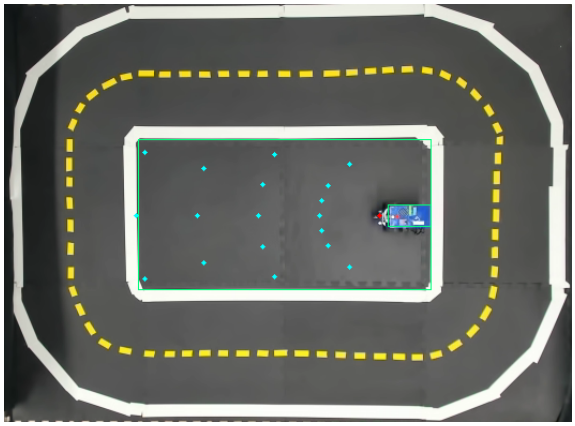


Figure 6.1: Picture of the locations of Duckies test cases from the top view.

Duckiebot Similar to a Duckie, it is placed in different facing directions (front, back, left, right) but placed at different distances without any angular rotation: at 25 cm, 50 cm, 75 cm, and 100 cm. Fig. 6.2 shows location of test cases for a Duckiebot.

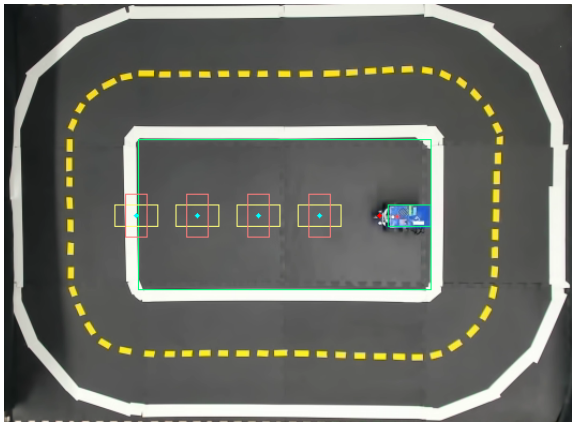


Figure 6.2: Picture of the locations of Duckiebot test cases from the top view.

Stop sign For a stop sign, we only place it at 25 cm, 50 cm, 75 cm, and 100 cm because it is a standard practice in the real world to place a stop sign in a way that faces a driver. Fig. 6.3 shows location of test cases for a stop sign.

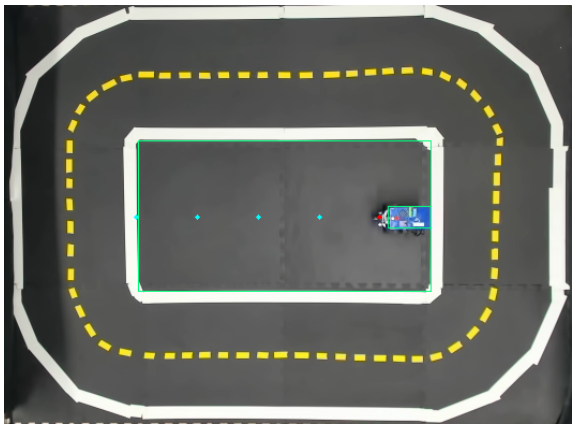


Figure 6.3: Picture of the locations of a stop sign test cases from the top view.

6.4.2 Conclusion of Nominal Cases Studies

As seen from Fig. 6.4, both Duckie and stop sign detection align with intuitive expectation – the further the distance, the lower the confidence score. However, although Duckiebot detection results show a similar trend, its performance is drastically lower than that of other classes. This is because this project has trained object detection models with the open-source dataset provided by Duckietown [19]. However, the dataset contains different environmental features, as mentioned in Section 4.2. Consequently, it also answers why some results from Duckiebot are not visualized since

they are not considered good candidates with a lower confidence score and have been removed during the process.

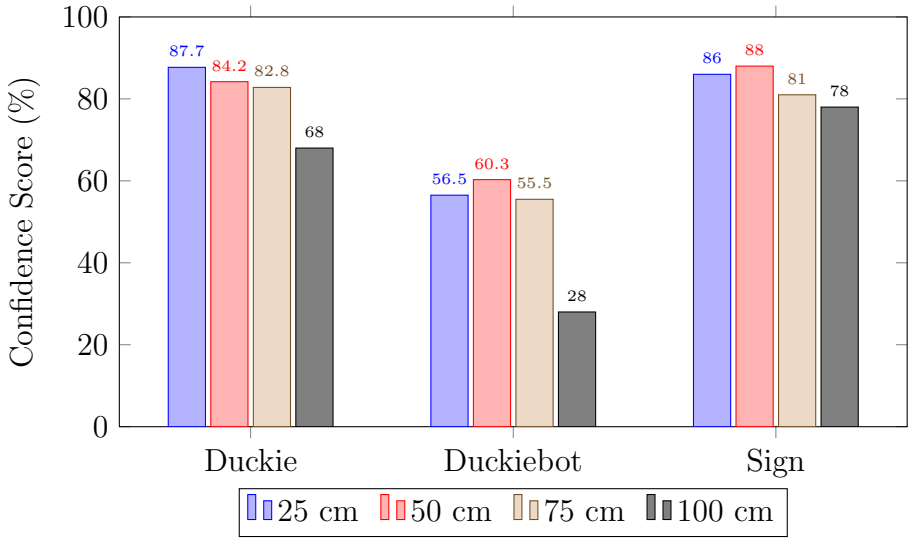


Figure 6.4: Histogram of overall result categorized by distance and objects for nominal cases, see Appendix for images. Note that the results from Duckie and Duckiebot shown in the histogram are from the average of four directions at each distance.

6.4.3 Safety-Critical Cases

Safety-critical cases are dangerous and unlikely to occur: occluded or too near objects.

Duckie A Duckie is tested at 25 cm facing the camera and being occluded for 25%, 50%, and 75% from left to right and vice versa, also when the Duckie is close to camera (5 cm to 10 cm).

Duckiebot As seen in Fig. 8.5a, a Duckiebot facing the camera is not detected, so our setup for this case only concerns the side and back of a Duckiebot placed at 25 cm from the camera.

6.4.4 Conclusion of Safety-Critical Cases Studies

From Appendix 8.2, results show that a Duckie can be detected for most cases, unless a Duckie and its red beak are blocked at 75% occlusion. This shows how a CNN works, since a CNN or other deep learning models are trained to understand

patterns from a training dataset. Thus, the results demonstrate that our model has learned to understand the key feature, i.e., Duckie's beak, while other yellow parts are also learned but less manifested. Similarly, from Appendix 8.2, a model has learned to understand pivotal components of a Duckiebot from the open-source dataset, which is a red and different design. However, we inspect that as a model is trained to understand patterns, results from Fig. 8.18 show the highlighted areas on an AprilTag, a white board with black dots at the back of the Duckiebot, and the Duckiebot's cables, indicating shared features between the old and new versions of the Duckiebot.

7 Conclusion and Outlook

Conclusion This thesis focuses on a real-time object detection system for Duckietown, a scaled-down autonomous driving simulation, and shifting from the previous work on a visual-based algorithm [43] to a deep learning approach through a convolutional neural network (CNN). In addition, this project explores the potential of using a graphics processing unit (GPU) for object detection instead of a central processing unit (CPU) because of two reasons: 1. our lab research has already heavily utilized a CPU, see Table 6.8 and 6.12 for hardware utilization, and 2. an object detection system contains many parameters and requires parallelization to accelerate run time.

So, we select modern single-stage object detection models from Ultralytics, including nano/tiny versions of YOLO from version 8 to 12, the current latest version, and fine-tune with the Duckietown open-source dataset [19]. However, the open-source dataset does not contain similar data to our environment, so we propose a custom dataset for our lab evaluation. Moreover, after all model selection details, we implement a fundamental decision-making algorithm as a simple key to autonomous driving.

The initial model evaluation results from Table 6.2, 6.3, 6.4, and 6.5 indicate higher performance when using input sizes of 640 and 480 for all models. However, we select only two candidates, including YOLOv11n with input size 640 and 480, because YOLOv11n with input size of 480 performs best among others with the same resolution (mAP_{50-95} of YOLOv11@480 (29.50 mAP_{50-95}) differed from the Top-1 (30.41 mAP_{50-95}) only 0.91 mAP_{50-95}) while YOLOv11n with input size of 640 returns almost indistinguishable performance among others with the same resolution. With this setup, we also explore the performance between two distinct input resolutions during deployment on the NVIDIA Jetson Nano 4GB.

For the evaluation during deployment, this project investigates the performance when the Duckiebot is stationary and running. Although it should be remarked that the collected results are varied over time, the model runtime, see Table 6.6 and 6.10, shows models on GPU highly outperform models on CPU, notably more than 10 times faster during inference. These results indicate the capability of the GPU on the NVIDIA Jetson Nano 4GB for model deployment. Nevertheless, results show that the NVIDIA Jetson Nano 4GB RAM utilization often reaches almost 100%, indicating a model contains many temporary layers or non-quantized parameters that rely on too much RAM, which can lead to future research opportunities. Overall, an object detection system using a GPU is fast enough for a real-time application.

Our custom dataset can be found at `https://universe.roboflow.com/duckietown-for-yolo/lis-duckietown-dataset-p0cfx`, and our fine-tuning setup at `https://www.kaggle.com/code/krittinch/yolovn-fine-tuning-testing`.

Outlook One important aspect found during evaluation is that RAM utilization is greater when a model runs on a GPU. This is likely because of the model architecture, which is hardly compromiseable; however, another research possibility lies in model quantization, referring to forcing a model's parameters to be integers, which can decrease RAM over-consumption on a small device like the NVIDIA Jetson Nano 4GB. Another aspect is the integration of language into a model, returning a prediction in the form of text or action. This is known as a vision-language model (VLM) or a vision-language action model (VLA), as opposed to producing only object detection results. Both tasks are a current emerging area in robotics [81, 92, 10, 39, 35, 33, 9, 90, 91], and especially in autonomous driving [88, 29, 93, 94, 50, 4]. However, since this is a very modern research field, there are not many datasets yet; as a result, some works [29, 50, 4] have to introduce new datasets to align with their research.

Confirmation

Herewith I, Krittin Chaowakarn, confirm that I independently prepared this work. No further references or auxiliary means except those declared in this document have been used.

Munich, September 9, 2025

Krittin Chaowakarn