Research and Adaption of Efficient Autonomous Driving methods in Duckietown

Programming with fully autonomous navigation

Ayush Kumar (ge95xod) - *BIE*
Parth Bharatbhai Karkar (ge32cap) - *BIE*
Servesh Khandwe (ge95luw) - *BIE*

Supervision

– Prof. Dr. Amr Alanwar (alanwar@tum.de)

– Ahmad Hafez (ahmad.hafez@tum.de)

Background

In recent years, the field of autonomous vehicles has witnessed unprecedented advancements, with research and development accelerating at an astonishing pace. The integration of artificial intelligence and robotics has opened up new frontiers, pushing the boundaries of what was once deemed impossible. Among the various domains of autonomous systems, Duckietown stands out as an innovative platform that not only fosters research and experimentation but also serves as an engaging educational tool.

Duckietown, conceived as an environment for testing and refining autonomous vehicle algorithms, provides a miniature urban setting where robotic vehicles, aptly named Duckiebots, navigate through a network of roads and intersections. This simulation serves as a controlled yet challenging arena for testing the capabilities of autonomous vehicles, mirroring the complexities of real-world urban scenarios.

Project Overview

The goal of our project is to design, implement, and test an autonomous driving system within the Duckietown framework. This endeavor involves integrating a multitude of technologies, ranging from computer vision and machine learning to robotics and control systems. As we delve into this project, our primary objectives include achieving reliable lane following, navigating intersections, and demonstrating an overall robust autonomous behavior in the Duckietown environment.

Object Detection Exploration: Early Attempts with SSD Model

In our pursuit of robust object detection within the Duckietown environment, we embarked on an initial implementation utilizing the Single Shot Multibox Detector (SSD) model. SSD stands out for its prowess in real-time object detection, combining high accuracy with operational efficiency. It operates on a unified architecture, eliminating the need for multiple stages, which is particularly advantageous in resource-constrained environments.

SSD achieves object detection by predicting bounding boxes and class probabilities directly from feature maps at multiple scales. This multi-layered approach enables the model to capture objects of various sizes effectively. Furthermore, the model integrates non-maximum suppression during inference, streamlining the process of identifying and classifying objects in real-time which is very ideal for our Duckietown environment.

Model Integration and Framework Utilization

Adopting an existing repository as a reference, our focus was on seamlessly incorporating the SSD model into the Duckietown environment. Leveraging its capabilities in efficient real-time object detection, SSD presented a robust framework that aligned with the demands of our miniature urban setting.

The dataset we used was meticulously curated for Duckietown and encapsulates the complexities of the miniature urban environment. Comprising annotated images, the dataset captures a diverse range of scenarios, including Duckiebots, traffic signs, and obstacles. The diversity in the dataset is instrumental in training the SSD model to discern and classify objects in real-world scenarios. Annotations of the dataset were already done by the authors, so it saved us a lot of time from the meticulous labeling of objects in the images. It really provides the model with ground truth information during the training process. The richness of our dataset is a critical component in imbuing the SSD model with the contextual awareness required for navigating Duckietown's intricate roadways.

Training Successes and Model Proficiency

During the training phase, the SSD model exhibited commendable proficiency in recognizing and categorizing objects within the Duckietown dataset. Our dataset size was around 3400 annotated images, and we ran more than 9000 training loops on the dataset. The model demonstrated an apt understanding of the environment, laying the groundwork for effective object detection, which can be seen in the figure below.

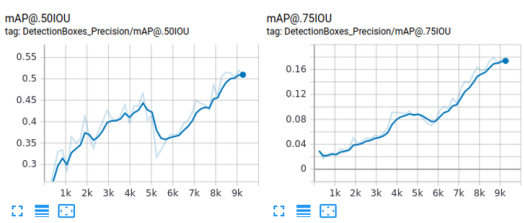

Figure 1: Model mAP

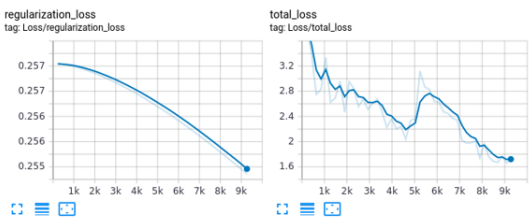

Figure 2: Model Loss

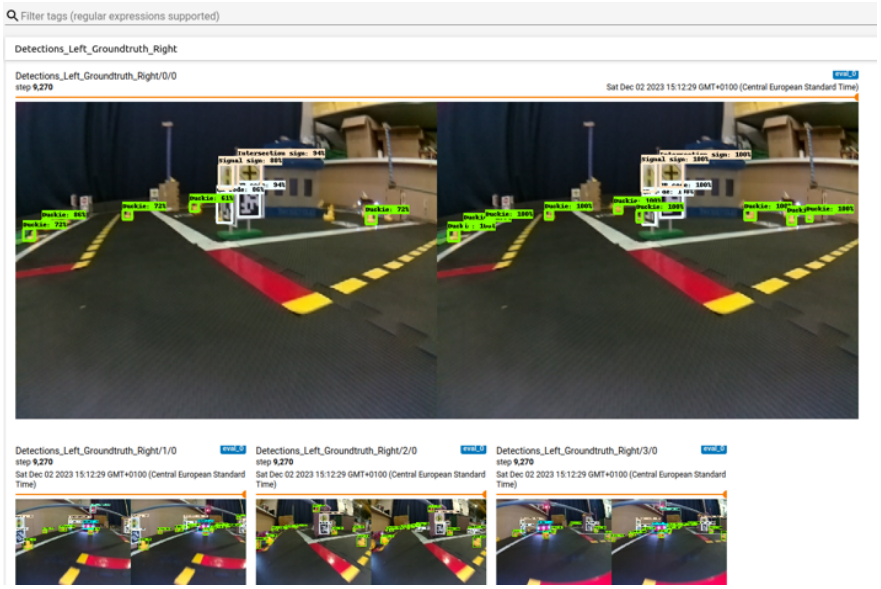

Figure 3: Model performance on duckietown environment

ROS Integration Hurdles

We tried to run the inference with the bot camera, and the main challenges emerged during the integration phase, notably in the form of ROS publisher-subscriber errors in the camera node. The model was running very well with images passed through the arguments, but in the Duckiebot, we could not really read the camera images due to some versioning and ROS errors. These obstacles posed impediments to the seamless deployment of the SSD model onto live Duckiebots. A thorough investigation into the ROS communication also didn't help solve the issue, so we decided to move on with a different approach.

Transitioning to YOLO Model: In-Depth Exploration and Implementation

Following the initial exploration of the SSD model and the hurdles we faced there, we transitioned to the You Only Look Once (YOLO) model for our object detection endeavors within Duckietown. At the heart of our implementations lies YOLOv3, a third iteration of the YOLO model known for its speed and accuracy. The YOLO architecture is distinctive for its unified approach, addressing object detection as a regression problem to predict bounding boxes and class probabilities directly from the image.

Anchor Boxes: YOLOv3 introduces the concept of anchor boxes, aiding the model in accurately predicting bounding box dimensions. These anchor boxes act as templates, enabling the model to adapt to objects of varying scales within the image.

Feature Pyramid Networks (FPNs): To capture object features at multiple scales, YOLOv3 incorporates Feature Pyramid Networks. This hierarchical approach enhances the model's ability to detect objects of different sizes within a given image, contributing to its robustness.

Detection Process: YOLOv3 employs a grid-based approach to divide the image into cells, and each cell predicts bounding boxes and class probabilities. The model operates on three scales, utilizing feature maps at different resolutions. The predictions are then refined through non-maximum suppression to yield the final set of detected objects.

Data Collection: Addressing YOLO Format Requirements

The YOLO model operates with a specific data format, demanding a distinct annotation schema compared to the SSD model. To align our dataset with YOLO's requirements, we initiated a comprehensive data collection effort, we can be able utilize the row images taken by anyone in the Duckietown environment but to run the training on YOLO the annotation needs to be in a very specific format like it involves specifying bounding boxes and class labels in a certain format, catering to the unique structure of the YOLO model.

This dataset, specifically tailored for YOLO, encompassed diverse scenarios encountered in Duckietown. Traffic signs, Duckiebots, and obstacles were meticulously annotated, ensuring that the YOLO model would comprehend the intricacies of the urban environment.

Implementation with YOLO Model

Our implementation utilizes the YOLOv3 architecture. The YOLO model was integrated into the Duckietown framework, leveraging the existing infrastructure for seamless deployment on Duckiebots. The training phase involves fine-tuning the YOLOv3 model on our YOLO-formatted dataset. Hyperparameter adjustments, transfer learning, and data augmentation techniques were employed, drawing upon the lessons learned from our previous SSD model exploration; after the fine-tuning is done for the pre-trained YOLO, the results of the model performance are given in below figures.

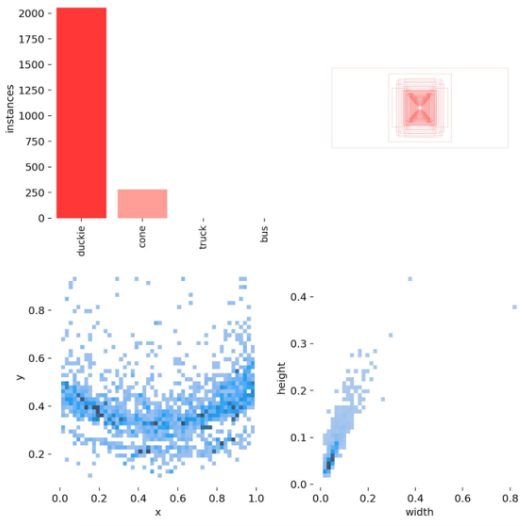

Figure 4: Labels of the baseline Model

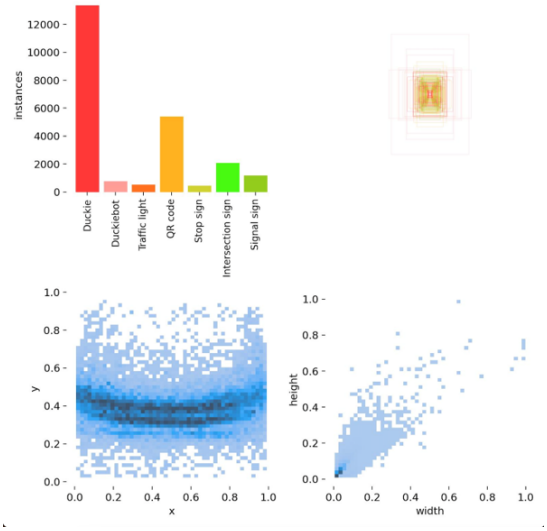

Figure 5: Labels of the new fine-tuned Model

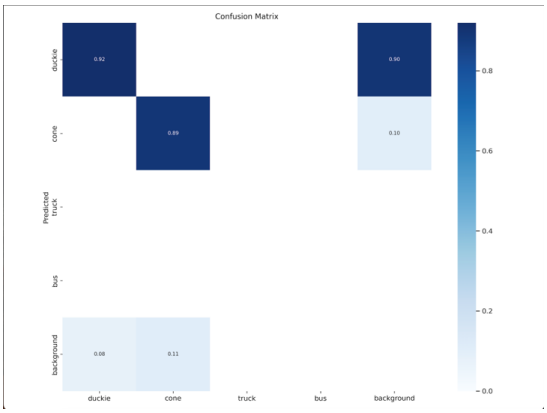

Figure 6: Confusion Matrix of the baseline Model

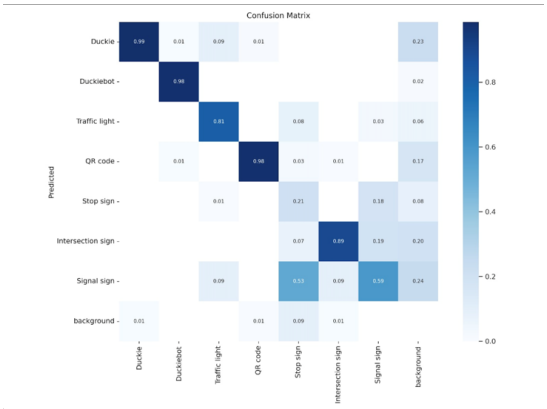

Figure 7: Confusion Matrix of the new fine-tuned Model

Real-World Adaptability: Challenges and Considerations

Our YOLO exploration also involved assessing the model's adaptability to real-world deployment on Duckiebots. Challenges were encountered, particularly in addressing real-time constraints and the nuances of the Duckietown environment. Considerations for deployment strategies and runtime optimizations were actively being pursued to make the model run on a simulated Duckietown environment.

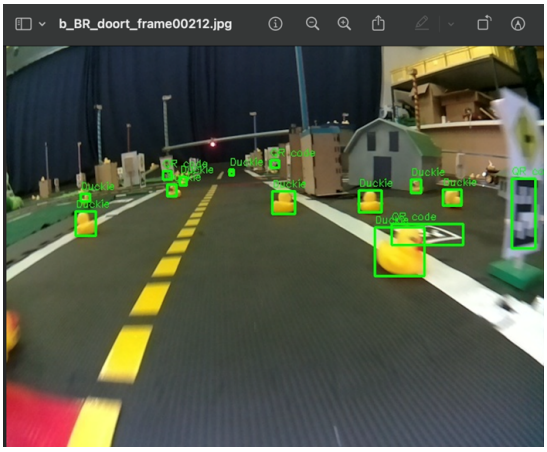

Figure 8: Model Performance on real Duckietown environment

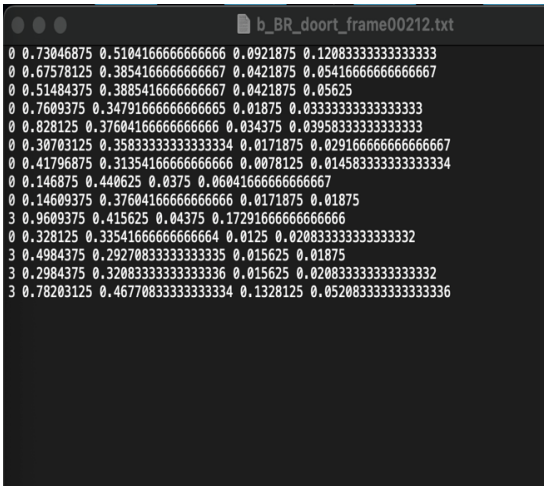

Figure 9: Annotations format used by YOLO for output

After we had completed the object detection working model, we shifted our focus then to making the Duckiebot run and adapt to the Duckietown environment and freely move around the roads intelligently, being aware of the external surroundings. A few of the main parts of this implementation would be making sure that the Duckiebot does correct lane following, also can detect the stop lines and other external hurdles on the road, and intelligently decide the turn direction at the intersection, which can be done using April tags on the road.

Autonomous Navigation in Duckietown: A Comprehensive Implementation

In the pursuit of realizing a fully autonomous Duckiebot capable of navigating the intricacies of Duckietown, our focus shifted to the seamless integration of essential modules. This endeavor involves harmonizing lane following, stop line detection, intersection navigation, and other critical components to enable a comprehensive and efficient navigation system. Leveraging the insights and foundations laid in our prior work with object detection using the SSD and YOLO models, we now embark on a detailed exploration of our efforts to make the Duckiebot autonomously navigate the Duckietown environment.

Lane Following: The Foundation of Navigation

Lane following forms the bedrock of our autonomous navigation system, providing the Duckiebot with the ability to stay within designated lanes. Building upon our previous work with object detection, the Duckiebot processes camera inputs to detect lane markings, ensuring precise alignment and trajectory control. The implementation we tried was first from the Duckietown demos, and it was not performing as per the expectation and deviating a lot for a simple lane following task, and that is why we decided to go with the Pure Pursuit Algorithm for Lane Following.

Pure Pursuit Algorithm for Lane Following

The implementation of lane following in autonomous vehicles plays a pivotal role in achieving accurate and robust navigation. One prominent algorithm that has demonstrated significant promise in this context is the Pure Pursuit Algorithm. Developed at the Robotics Institute and extensively tested on various platforms, including the Terragator, NavLab, and NavLab II (HMMWV), the Pure Pursuit Algorithm has shown remarkable potential as a general-purpose tracking algorithm.

Originating as a method to calculate the arc necessary to bring a robot back onto a path, the Pure Pursuit Algorithm has undergone refinement and validation over the years. Initially applied to the Terragator in the early 80s, it later transitioned to the NavLab project. Comparative studies with alternative path-tracking algorithms, such as the Quintic Polynomial approach and a "Control Theory" approach, consistently highlighted the robustness and reliability of the Pure Pursuit method.

Principle Theory and Implementation

The Pure Pursuit Algorithm is a tracking algorithm designed to calculate the curvature needed to guide a vehicle from its current position to a predetermined goal position. The core idea revolves around choosing a goal position located at a specific distance ahead of the vehicle on the path. The algorithm is aptly named as the vehicle is envisioned to be pursuing a point on the path ahead of it, simulating the way humans drive by focusing on a point in front of the vehicle. The theoretical underpinning of the Pure Pursuit Algorithm involves a geometric derivation to determine the curvature required to drive the vehicle to the chosen goal point. The critical parameter in this algorithm is the lookahead distance, denoted by '1'. The algorithm calculates the curvature based on the inverse square of the lookahead distance, creating a proportional controller-like relationship.

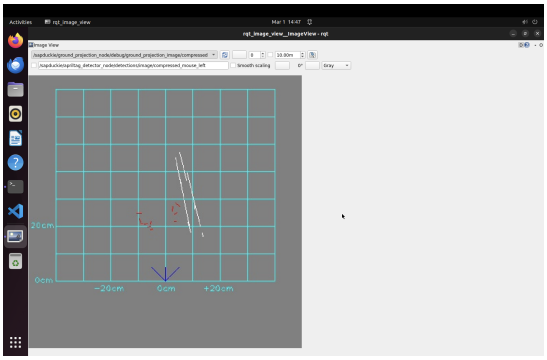

Figure 10: Ground Projection

Figure 11: Lane Edge detection

The algorithm explained above is more general and made to work in all environments, but to adapt to our Duckietown environment, we found some modification that needs to be done.

Modifications for better adaptability

Finding the Target Point : In the standard Pure Pursuit Algorithm, the calculation of the target point involves complex computations based on the defined path.

In our modification: We directly estimate our target point by offsetting ground-projected yellow lane points to the right. If the yellow lane is not visible, we offset ground-projected white lane points to the left. The average of these offset points is taken as our estimate of the target point. Consideration is given to the average direction of line segments, accommodating scenarios where yellow line segments are perpendicular to the robot.

Varying Speed and Omega Gain: To optimize the performance of our robot on different path segments the robot dynamically adjusts its speed and omega gain based on detected path characteristics (left turn, right turn, or straight path). Gradual speed-up occurs on straight paths, accompanied by a reduction in the omega gain to avoid jerky movements at high speeds. The robot gradually slows down at turns, with an increase in the omega gain for smooth, sharp turns. A second-degree polynomial governs the velocity/omega gain changes, ensuring a gradual speed increase after turns and quicker slowdowns at turns for safety.

Modified Lane Filter: Our lane filter package underwent a modification to enhance its adaptability, At each update step, we calculate the time elapsed since the last update. Based on this elapsed time, we dynamically scale the variance of the Gaussian used for smoothing the belief. This adaptive scaling is particularly useful when dealing with varying frames per second (FPS), ensuring optimal belief smoothing regardless of FPS variance.

Stop Line Detection: Enhancing Intersection Safety

Stop line detection plays a pivotal role in ensuring safe and efficient intersection navigation. The Duckiebot, equipped with camera sensors, identifies stop lines and halts appropriately, adhering to traffic regulations and avoiding collisions at intersections. this is mainly done by identifying the color coordinates of the line ahead of the camera; if we see a red line, then we should stop at a safe distance.

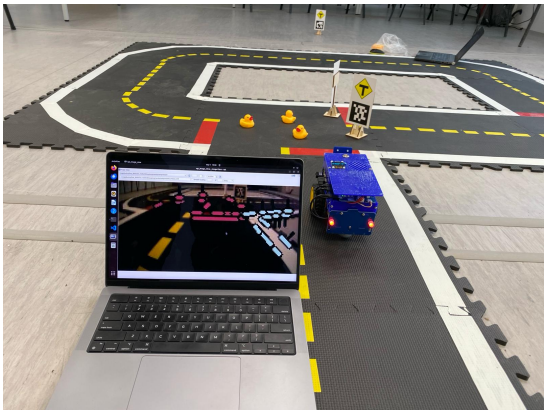

Figure 12: Color detector for stop line filter

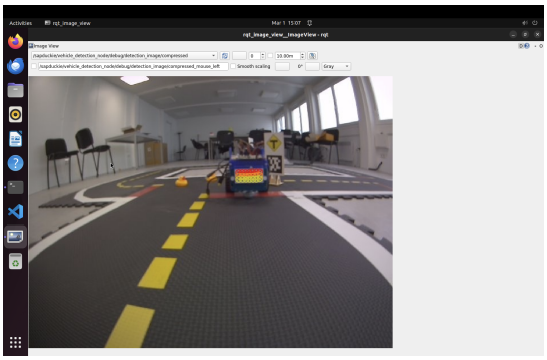

Figure 13: Bot detector for Collision avoidance

Similarly, we need to stop when we have a bot in front of us to avoid colliding with that, so bot detection is also very crucial for that.

Intersection Navigation: Navigating Duckietown's Crossroads

We leveraged the stop line filter to detect stop lines. By incorporating this package into our codebase, our DuckieBot now effectively slows down and stops at stop lines during lane following; the stop lines are always placed at the intersection, so now Duckiebot needs to decide which directions to move after stop, and this decision is taken by reading the April tags at the intersection.

Controller Adaptation To accommodate the sleep duration required during turns, we made adjustments to the controller logic. The Control Action method was transformed into the drive method, and the drive method now contains all the necessary logic for computing actions to control the DuckieBot at the intersection.

Intersection Detection using AprilTags

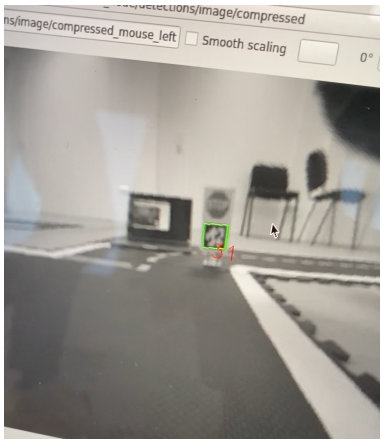

Figure 14: April Tag sign detection

AprilTags were employed to identify legal directions at intersections. We used the April tags package from dtcore, which integrates seamlessly with our codebase. We introduced a dictionary, mapping AprilTag IDs to legal turns (0 for left, 1 for straight, and 2 for right) from which we track the action that needs to be taken when the respective tag number is identified. When the turn is being taken, LED signals are implemented to indicate the intended direction at intersections. We defined three signals for left, straight, and right turns. The LED parameter controls whether LED signals are activated, allowing flexibility in testing and real-world scenarios.

Nine turning parameters, encompassing velocities, angular velocities, and sleep durations, were introduced for left, straight, and right turns. These parameters can be tuned dynamically during runtime, facilitating a straightforward tuning process without requiring code modification.

Harmonizing All the Modules

The integration of lane following, stop line detection, and intersection navigation posed challenges in terms of real-time synchronization and seamless collaboration between modules. To make sure that all are running parallel, we need to create an individual package for each of them with a correct launch file in it, additionally, we need to have an overall project launch file which then launches all the packages together, the main performance starts to kick in when we have all the packages running, we have noticed quite a few times that the memory and CPU usage was hitting the maximum and that was slowing down and causing latency in decision making by the bot and it seemed confused on what to do when having an overload of processing power but the individual modules we tested were performing overall good.

Conclusion

In summary, the report details the development and implementation of key components for the DuckieBot's autonomous navigation system. We dwelled deeper into object detection using Duckiebot first with SSD and YOLO model and tried to achieve near best performance, then we shifted focus to overall autonomous driving like The lane following module, inspired by the modified pure pursuit controller, efficiently tracks lanes by estimating a target point based on ground-projected lane information. Variations in speed and omega gain are dynamically adjusted to optimize performance during straight paths and turn, enhancing the DuckieBot's responsiveness and stability. The intersection navigation strategy simplifies the complex task by relying on April tags to identify legal turn directions. The implementation involves stopping at intersections, detecting April tags, and selecting turns based on available legal options. LED signals are employed to indicate turn directions, enhancing the DuckieBot's visibility and communication in shared spaces. Turning maneuvers are achieved by applying specific velocity and angular velocity parameters, facilitating smooth and controlled navigation through intersections. at last, we also tried to combine all the modules and run them together for an autonomous driving feature, which was partially working, and the main problem we identified was due to over usage of the CPU capacity.

References

[1] S. Macenski, S. Singh, F. Martin, and J. Gines, "Regulated pure pursuit for robot path tracking," 2023.

[2] "duckietown/dt-core." https://github.com/duckietown/dt-core?tab=readme-ov-file.

[3] yahsiuhsieh, "pure-pursuit." https://github.com/yahsiuhsieh/pure-pursuit.

[4] saryazdi, "Duckietown-object-detection-lfv." https://github.com/saryazdi/Duckietown-Object-Detection-LFV.

[5] samuelfneumann, "Robotics/duckietown - intersection navigation and vehicle avoidance/following," 2023.

[6] samuelfneumann, "Robotics/duckietown - predicting apriltags using general value functions," 2023.

[7] duckietown ethz, "proj-lfivop-ml." https://github.com/duckietown-ethz/proj-lfivop-ml.

[8] duckietown, "duckietown-lx." https://github.com/duckietown/duckietown-lx/tree/mooc2022/object-detection.