DuckieBot Following: Trailing Robots based on Computer Vision

António Ribeiro
Faculty of Engineering
University of Porto
Porto, Portugal
up201906761@edu.fe.up.pt

Filipe Campos
Faculty of Engineering
University of Porto
Porto, Portugal
up201905609@edu.fe.up.pt

Francisco Cerqueira
Faculty of Engineering
University of Porto
Porto, Portugal
up201905337@edu.fe.up.pt

***Abstract*—We propose an approach for enabling a robotic system to autonomously trail a DuckieBot, leveraging the tracking of its movements through the utilization of ArUco markers or object detection techniques, and validating it in the Duckietown simulation environment. Our contribution extends to the introduction of a novel object detection architecture, DOLO, which augments the YOLOv5 architecture with two supplementary ordinal regression tasks designed to forecast the distance and rotational orientation of the targeted object. We attain an autonomous agent proficient in trailing a designated DuckieBot through multiple intersections while consistently upholding an optimal distance.**

***Index Terms*—robotics and automation, navigation, simulation, object detection**

I. Introduction

In recent years, the field of Computer Vision has witnessed remarkable advancements, particularly in the realm of robotics and autonomous systems. One intriguing application that has garnered attention is the DuckieTown project, a simulated environment designed for testing and developing autonomous robotic systems (Figure 1).

The primary objective of this research is to develop an autonomous agent tasked with following another robot, called a Duckiebot, within this simulation environment. This agent should demonstrate proficiency in road following, adherence to road lines, and precise movement at intersections. More importantly, the study aims to investigate the follower robot's capability to identify and track the guide robot using two distinct methodologies—ArUco [1] markers and a novel Object Detection architecture based on YOLOv5 [2], which, as must perform the additional task of predicting both the distance and rotation of the objects detected.

II. Related Work

The need to identify reference points in an image is present in a plethora of computer vision tasks. To do so, fiducial markers are commonly used, providing well-known reference points in an image.

In robotics, there are two main types of markers commonly used. AprilTags [3] and ArUco [1]

Object Detection based on Deep-Neural Networks is currently a key research area in computer vision due to its applicability to autonomous driving applications. This task consists of, on an image, detecting the bounding box which encloses each individual object, and determining its class.

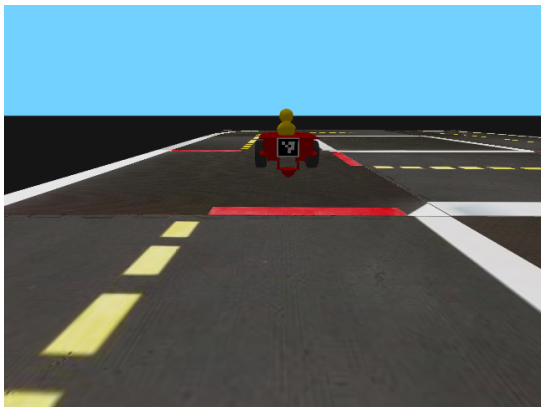

Fig. 1: Screenshot of the camera view from the perspective of the controllable robot.

The first approach to perform Object Detection is based on Deep Learning (DL), using Region-Based Convolutional Neural Networks (R-CNN) [4]. Throughout the years, multiple improvements have been proposed to either improve detection quality or increase inference speed, such as Fast R-CNN [5] or Faster R-CNN [6]. In particular, for embedded devices, models need to be as fast as possible while retaining high detection accuracy. The family of YOLO (You Only Look Once) models, initially proposed by Redmond *et al.* [7], provides a more time-efficient approach. After successive new versions, there are currently two main widespread revisions, YOLOv5 [2] and YOLOv7 [8], with the main difference being that, for the same parameter count, YOLOv7 is more accurate while YOLOv5 is faster.

A regression problem, such as identifying the angle or distance of a vehicle, can be converted into an ordinal regression problem by dividing continuous values into a set of discreet values. The most common approaches to solve ordinal regression tasks using Deep Neural Networks are: modelling it as a regression problem, discreetizing the outputs and using ordinal metrics; treating it as a classification problem or employing ordinal loss functions [9] such as the Cumulative Link Loss [10].

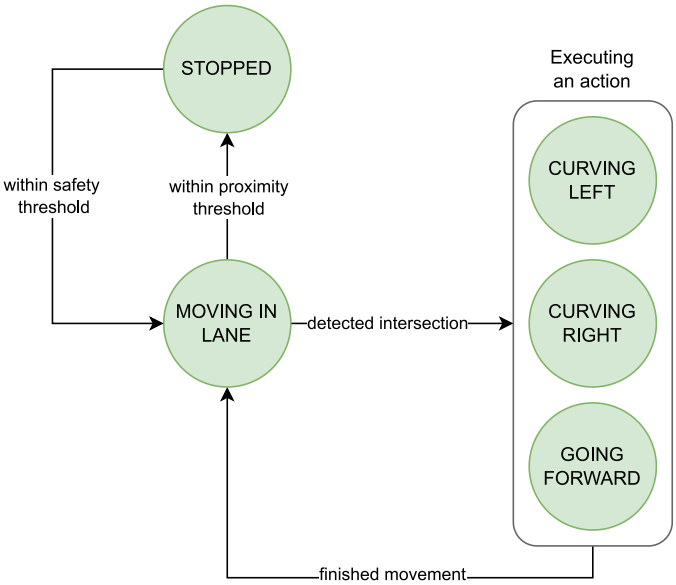

Fig. 2: Duckiebot's state machine diagram.

Lane Following is a common robotics task. State of the art approaches are ever evolving and increasingly complex, using techniques such as Reinforcement Learning to great success. Yet, the main focus of this work does not lie in obtaining the best lane-following possible, instead using simpler approaches based on classical computer vision operations.

III. DuckieBot Implementation

A. Description and Architecture

The controllable robot has a two-wheel differential drive and incorporates a reactive architecture model. The movement is controlled by a 2D vector (v_1, v_2), which executes the longitudinal and lateral movement, respectively.

To guide the robot at each particular instance, we devised a state machine (pictured in Figure 2) that indicates the current state of the movement and transitions according to the perceived environment context. The state machine initiates in the *MOVING IN LANE* state, which presumes the robot spawns in the correct location. If the distance to the detected guide robot is not within a given safety limit, the state transitions to *STOPPED* until it is possible to resume the movement. Throughout the path, upon detecting a movement from the guide robot that needs to be replicated, such as turning right or left and going straight, the algorithm stores it in an action queue that memorizes what to execute in the upcoming intersection.

These actions were implemented with a simple event loop that performs the path described by a bezier curve when turning and moves longitudinally when going straight. Lane following centers the Duckiebot in the right lane by making angle adjustments when needed.

B. Lane Detection and Following

Yellow Lines: The middle of the road is represented by a dashed yellow line. Instead of employing a traditional line detector, which performs poorly due to the gaps in the line. Instead, we individually detect each segment and identify its center point $P_i = (x_i, y_i)$. Based on this, each line can be identified by two points (P_i, P_{i+1}). Yet, with this approach, we also identify duckies as line segments. To fix this, we remove all lines that are too large, $||P_i, P_{i+1}||_2 \geq$ threshold

Red Lines: The starting point of an intersection is determined by a red line at the end of the lane. We apply a mask to detect all yellow pixels in the image, to which we employ two morphological operations: erosion and dilation. We use the *Hough* transform to retrieve the lines representing the yellow shapes, remove all non-horizontal lines (irrelevant to our detection), and select the closest one. With the length of this line, we can measure how close the Duckiebot is to the end of the lane and start to execute the buffered action. Our experiments found that the optimal threshold for determining the intersection's line length is 300 pixels.

White Lines: We repeat the aforementioned steps to detect the white lines in our camera view, with a white color mask, erosion, and dilation, followed by the *Hough* transform. We took the average line from this set and used it to direct the movement.

Lane following: This algorithm takes the detected lines and determines their angle. With this information, we can execute minor velocity adjustments that compensate for angle differences around a reference value, ensuring that the Duckiebot travels in the center of the lane. We defined multiple angle intervals that determine how impactful the adjustment is and edge cases that require further action, such as detecting a line by itself (a sharper turn) and detecting a line on the opposite side of the correct screen position (going in the wrong direction).

IV. ArUco Marker Detection

To keep track of a DuckieBot, we first employed a 4×4 ArUco marker attached to the robot's rear. The algorithm captures the corners of the identifier, retrieving rotation and translation vectors, which are then used to determine the distance and orientation of the robot. This proved to be a simple and efficient approach to detect the robot's location and estimate its pose.

Yet, a limitation present in this approach is the fact that the marker can easily become invisible to the robot's camera whenever a DuckieBot performs a sharp curve or a similar manoeuvre.

V. Object Detection

We developed an Object Detection model as an alternative to using an Aruco marker to detect the DuckieBot. This model identifies both DuckieBots and duckies, providing their distance and orientation estimations.

A. Dataset

Due to the absence of a suitable Duckietown simulator dataset meeting our requirements, we created a custom dataset.

TABLE I: Dataset collection statistics. Number of samples per map per class.

Map	Empty	Duckie	Duckiebot	Total
ETH_large_intersect	78	45	42	165
ETH_small_intersect	39	45	39	123
loop_dyn_duckiebots	48	70	56	174
loop_obstacles	53	60	56	169
regress_4way_adam	53	39	64	156
udem1	71	47	60	178
Total	342	306	317	965

The data collection process spanned various maps, each catering to specific classes, as outlined in Table I.

This process involved using the simulator to capture regular screenshots, registering the target class, bounding box, and the position and orientation relative to the controllable robot. Only one target class instance was placed on the map to ensure ease of position and orientation retrieval in the simulation environment. However, the extraction of bounding boxes required a post-processing workflow:

1) Application of a mask using the specific target class color (yellow for duckies, red for duckiebots) on the original image, followed by a conversion to grayscale. Fine-tuning color parameters was necessary because other map elements like road lanes or traffic signs share similar colors.
2) Thresholding was applied to generate a binary image.
3) Morphological operations were performed to eliminate potential noise from other detected objects.
4) Acquiring the contours of individual image segments, whereby the segment displaying the largest area is deemed the most likely candidate for the target object.

B. Model

As a typical object detection model, YOLOv5 has three distinct tasks. For each proposed anchor, it must predict whether or not an object is present (obj classification), perform regression to determine the bounding box's coordinates (box regression), and determine the class of the object contained within the said box (cls classification). These changes are reflected in the loss functions used, which combine a Binary Cross-Entropy loss with a sigmoid function. Consider y, the real label, and $\hat{y}$ the predicted label

$$\begin{aligned}\mathcal{L}_{obj}(y,\hat{y}) &= -w_n\left[\hat{y}_n \cdot \log\sigma(y_n) + (1-\hat{y}_n)\cdot\log(1-\sigma(y_n))\right]\\ \mathcal{L}_{cls}(y,\hat{y}) &= -w_n\left[\hat{y}_n \cdot \log\sigma(y_n) + (1-\hat{y}_n)\cdot\log(1-\sigma(y_n))\right]\end{aligned} \tag{1}$$

and for the regression task

$$\begin{aligned}\mathcal{L}_{box}(b,\hat{b}) &= 1 - IoU(b,\hat{b})\\ IoU(b,\hat{b}) &= \frac{|b\cap\hat{b}|}{|b\cup\hat{b}|}\end{aligned} \tag{2}$$

where both b and $\hat{b}$ denote bounding boxes.

Yet, YOLOv5 does not support situations where there are multiple classification tasks, which we require. To solve this issue, we present a novel architecture, **DOLO** - Duckies Only Look Once. For our new architecture, we expand upon this by coupling two new classification tasks. The first will predict the distance an object is from the driver (in 5 discreet steps: very close, close, fine, far, very far), and the second will predict the rotation of an object (once again in 5 steps: very left, left, middle, right, very right). As a baseline loss function, we employ a Cross-Entropy loss function:

$$\text{BCE}_n = -\sum_{c=1}^{C} w_c \log\frac{\exp(x_n,c)}{\sum_{i=1}^{C}\exp(x_n,i)}y_{n,c} \tag{3}$$

Yet, this approach considers that there is no relationship between each class when, in fact, for both problems, there is an ordinal scale. To make use of this information, we also tested different loss functions that turn the initial classification problem into an **ordinal regression**. First, we employ a Cumulative Link Loss [10] which is defined as

$$\psi_{\text{CL}}(y,\alpha) = \begin{cases} -\log(\sigma(\alpha_1)) & \text{if } y = 1\\ -\log(\sigma(\alpha_y) - \sigma(\alpha_{y-1})) & \text{if } 1 < y < k\\ -\log(\sigma(1-\sigma(\alpha_{k-1})) & \text{if } y = k\end{cases} \tag{4}$$

Therefore, the new loss functions are defined as such,

$$\begin{aligned}\mathcal{L}_{dist}(y,\alpha) &= \psi_{\text{CL}}(y,\alpha)\\ \mathcal{L}_{rot}(y,\alpha) &= \psi_{\text{CL}}(y,\alpha)\end{aligned} \tag{5}$$

The final loss function is obtained as such

$$\mathcal{L} = w_1\cdot\mathcal{L}_{box} + w_2\cdot\mathcal{L}_{obj} + w_3\cdot\mathcal{L}_{cls} + w_4\cdot\mathcal{L}_{dist} + w_5\cdot\mathcal{L}_{rot} \tag{6}$$

where $W = \{w_1, w_2, ..., w_5\}$ is a set of weights meant to balance the value of each individual loss function. These values depend on the loss functions used since they may have different orders of magnitude. Table II specifies the values we used during our experiments.

We propose a training process (Figure 3) with the following three steps:

1) **Pre-train the YOLOv5-small on COCO dataset**. The authors of the model have already performed this step. Therefore, we can use the model weights which have been published. This step is important to achieve better generalizability and reduce the computational requirements for training the remainder of the steps.
2) **Perform Domain Adaptation**. We can leverage existing datasets collected from the Duckietown simulator (without ordinal classification labels) and use them to adapt the model to the simulation environment. During this step, we keep the model backbone frozen to avoid overfitting and reducing the computational requirements of the training.
3) **Fine tune to our dataset**. Finally, while maintaining the frozen backbone, we fine-tuned our smaller dataset, which included ordinal classification labels.

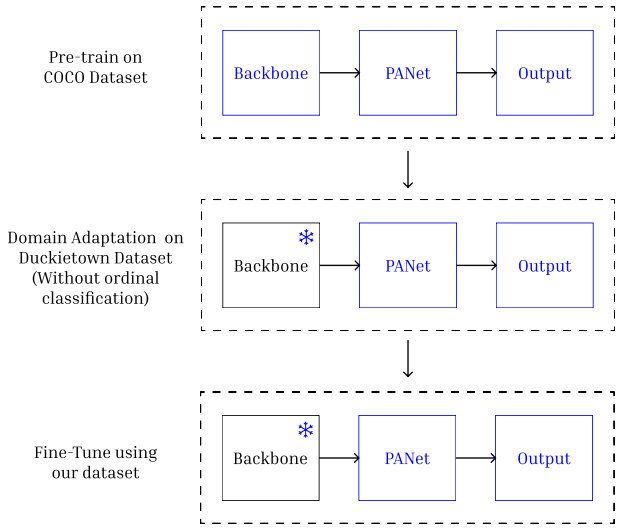

Fig. 3: Training process for the DOLO model. We start with a YOLOv5 model pre-trained on the COCO dataset, perform domain adaptation on a Duckietown simulator dataset, and then use our own dataset to perform fine-tuning. These last two steps have a frozen backbone to ensure it does not overfit our data and to reduce the computational requirements for training.

TABLE II: Loss functions utilized during training and their respective weights.

Loss function	w_1	w_2	w_3	w_4	w_5
Cross Entropy	0.05	0.16	0.025	0.1	0.1
Cumulative Link Loss	0.05	0.16	0.025	0.001	0.001

VI. Results

During the model's training process, we discovered that performing the Domain Adaptation step instead of improving the experimental results led to worse detection quality, as shown in Table III. Another aspect that negatively contributed to the results was the use of ordinal loss functions, which promote unimodality and are consistently outperformed by the commonly used cross-entropy loss. Therefore, our best model was achieved by training without previous domain adaptation

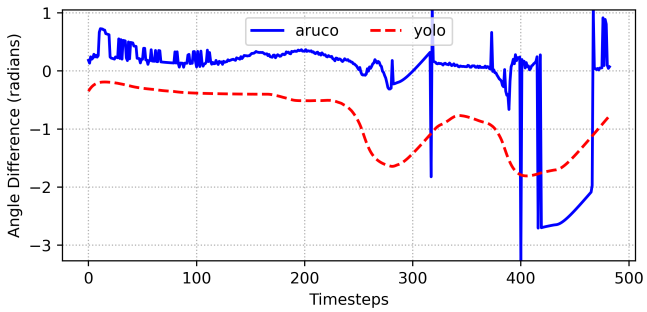

Fig. 4: Deviation of detected angles from real angles.

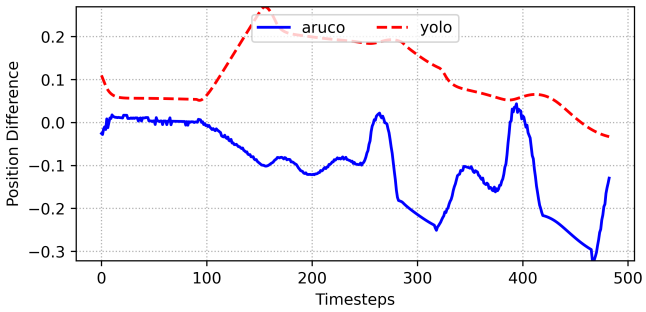

Fig. 5: Deviation of detected distances from real distances.

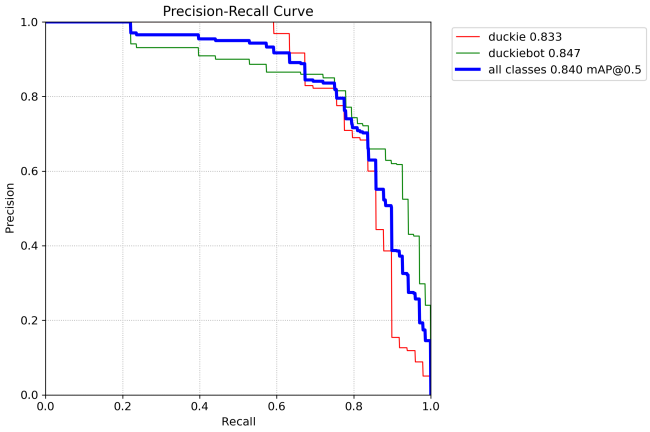

Fig. 6: Precision-Recall Curve for predictions made by DOLO model

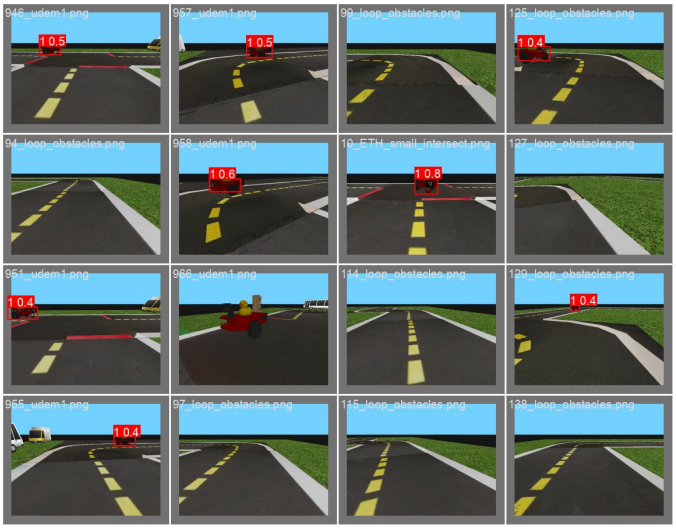

Fig. 7: Examples of predictions made by the DOLO model

TABLE III: Experimental results of the DOLO model. We show the Mean Average Precision (mAP) at different Intersection over Union (IoU) levels and, for each task, the precision (P) and recall (R) values.

Name	Loss	Epoch	mAP@0.5	mAP@0.5:0.95	Class		Distance		Rotation	
					P	R	P	R	P	R
DOLO (No Domain Adaptation)	CE	100	0.84	0.5	0.82	0.72	0.36	0.11	0.12	0.14
DOLO (No Domain Adaptation)	CLL	50	0.994	0.802	0.82	0.722	0.121	0.139	0.36	0.107
DOLO (Domain Adaptation)	CE	25 + 50	0.7041	0.26364	0.53	0.59	0.07	0.11	0.13	0.15

for 100 epochs and employing the Cross-Entropy loss function for both the distance and rotation tasks. In Figure 7, we showcase some predictions for images in the test set; it performs admirably in most instances, with the exemption of an image where the DuckieBot is facing the camera. Since the dataset was collected with the following task in mind, there is a slight bias towards having more images with a DuckieBot facing away from the camera instead of toward it. Finally, we show the precision-recall curve obtained in Figure 6.

In our experiments, we measured the angle and distance deviations between both detection strategies and the distance and angle computed through internal simulation values, as depicted in Figure 4 and Figure 5. From both analyses, we can conclude that the ArUco detection algorithm performs better when initiating the movement, given that the robot starts directly behind the guide but quickly declines upon losing track of the marker. On the other hand, DOLO manages to counteract this limitation, given that the guide robot can be detected from all possible camera angles, revealing its location.

We also found that the lane following algorithm constraints the overall movement duration, as it accumulates errors and, on longer path iterations, steers the robot out of map bounds.

The simulator's field of view holds a substantial influence over the performance of the detection algorithms. When conducting a sharp turn, it's significantly easier for the robot to lose track of the guide bot, deterring its movement until another detection occurs.

VII. Conclusion

In this work, we describe an approach for DuckieBot trailing which is functional and reliable by employing ArUco marker or object detection tracking mechanisms coupled with a lane-following algorithm based on classic computer vision operations such as masking and morphological operations. We presented a novel architecture DOLO, which included two additional ordinal regression tasks from which we discovered that simpler approaches can prevail over more complex ones which attempt to imbue domain knowledge; this occurred with the ordinal regression loss functions, which were consistently outperformed by their classification loss counterpart, Cross Entropy.

In future work, the lane following system could be enhanced by integrating PID or reinforcement learning to achieve a more robust navigation. Additionally, expanding the dimensions of the training dataset by incorporating diverse environmental conditions, scenarios, and edge cases would make object detection more adaptive and resilient. Finally, exploring novel methods to train the model, such as incorporating self-supervised learning, could further refine the system's ability to navigate complex and dynamic road environments efficiently.

References

[1] S. Garrido-Jurado, R. Muñoz-Salinas, F. Madrid-Cuevas, and M. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, p. 2280–2292, 06 2014.

[2] G. Jocher, "Ultralytics yolov5," 2020.

[3] E. Olson, "AprilTag: A robust and flexible visual fiducial system," in *2011 IEEE International Conference on Robotics and Automation*, pp. 3400–3407, May 2011.

[4] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," Oct. 2014.

[5] R. Girshick, "Fast R-CNN," Sept. 2015.

[6] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," Jan. 2016.

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," May 2016.

[8] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," *arXiv preprint arXiv:2207.02696*, 2022.

[9] J. S. Cardoso, R. Cruz, and T. Albuquerque, "Unimodal distributions for ordinal regression," 2023.

[10] A. Tewari and P. L. Bartlett, "On the consistency of multiclass classification methods," *Journal of Machine Learning Research*, vol. 8, no. 36, pp. 1007–1025, 2007.