Object Detection-Based Behaviour using YOLO on Thymio

Francesco Saverio Zuppichini[1] and Alessia Ruggeri[2]

***Abstract*— The ability to classify objects in both space and domain is encoded in an enormous variety of species on the planet. Recently, deep learning approaches have been developed to make machines able to mimic the same task with interesting results. With this paper, we propose a minimal architecture to archive the same goal using a cheap two wheels robots, the Thymio, with a mounted frontal camera in order to make it act based on the objects detected in the surrounding.**

I. Introduction

Finding objects in space by identifying their identity and their coordinates is a basic task that almost every animal is able to accomplish. The ability to solve this problem is extremely important in robotics, where a machine needs to act based on its surrounding. We implemented an architecture running on ROS, that is able to find selected targets object by using YOLO, a real-time object detection model, by only using a cheap camera.

Specifically, our robot will wait for an user-specified class, e.g. *dog*, and, when found, it moves towards the object by keeping it in the middle of its view. We also implemented a PID based algorithm to avoid obstacles and properly stop in front of the founded item.

The paper is structured as follows: we will first describe the architecture by showing each individual part of the topology and how they interact; then, we will discuss the challenges during the development process and finally, we will show the results.

II. Architecture overview

We used the Robotic Operating System,[**ROS**], since its *de facto* the state of the art. Our architecture is composed by:

1) A robot with a mounted camera with a ROS node running on the on-board computer
2) A web server running the Object Detection model
3) A ROS node running on a local machine that controll the robot

As a robot, we used the *MightyThymio* [2], which is a robot developed on top of the Thymio, a small mass-produced mobile robot usually used in primary and secondary education. The MightyThymio has a powerful Odroid C1 on board, a 1GB quad-core ARM Linux machine, with a HD camera mounted on top. The authors also implemented a ROS node to communicate with it and open-sourced the code.

The MightyThymio Robot

In our project, the MightyThymio has the same configuration proposed in [2]. Briefly, the onboard computer exposes the Thymios sensors through ROS. Thus, another ROS node can publish and subscribe to it in order to communicate with its sensors. For example, we can subscribe to the *camera sensors* to get information from the proximity sensors to avoid hitting obstacles. Also, we can simply subscribe to the camera's topic to get both compressed and uncompressed images.

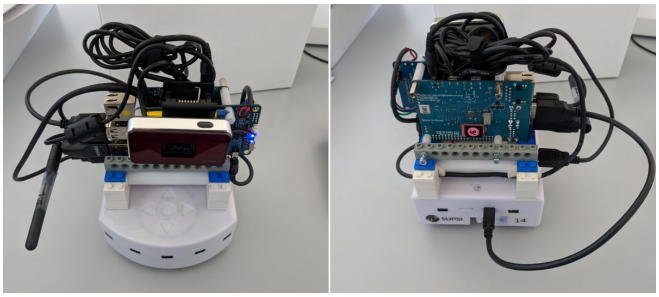

Fig. 1. The MightyThymio

Local ROS node

We created a new ROS node to communicate with the running node on the MightyThymio to control it accordingly with a specified behavior based on the object detected by the model. The user specifies a set of target classes, for example *dog* and *cat*. Then our node, after having subscribed to the camera topic of the MightyThymio, sends the fetched image to the model hosted on a Web Server through an HTTP request and gets a response detected classes and their bounding boxes. In the case there is no target object, the robot enters in `exploration` mode and starts spinning around. When one or more of the target classes are found simultaneously, the robot will go towards the target with the biggest bounding box area while keeping it in the center of the camera in order to follow it. In addition, we have an obstacle-avoiding algorithm in order to properly stop the Thymio before it hits the target.

Object Detection REST-API Web Server

We used pre-trained Keras implementation[1] of YOLO[4] model, the state-of-the-art real-time object detection model, with TensorFlow[3] backend to predict the classes in the image and their bounding boxes. A bounding box is a rectangle in which the predicted class is contained. For example, in the following picture we can see the bounding boxes for a *dog*, a *bicycle* and a *truck*. Each box is labeled with the class name and its score.

[1]https://github.com/qqwweee/keras-yolo3

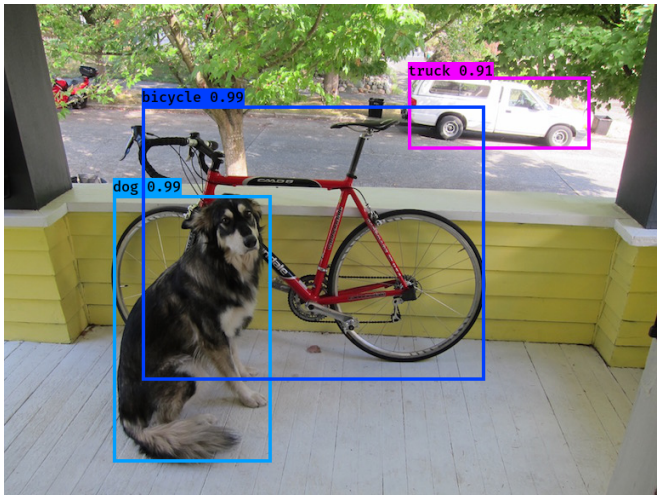

Fig. 2. Yolo Prediction Example

The model was trained on the COCO(Common Object in Context)[1] Dataset, that contains 80 classes.

On top of that, we built a REST-API Web Server using Python3 and Flask[2], a micro-framework to create Web Servers, to make the model accessible from the Web. We decided to decouple the logic behind the robot behavior and the model in order to make our architecture more scalable. For instance, if needed, we can deploy our model in the cloud while keeping the running ROS code intact.

We following endpoints are available:

```
GET /model
POST /prediction −−image −−size −−compress
```

Each of them returns a JSON serialised response. The `/model` endpoint is used to get informations about the model. These informations are the classes that can be predicted and the colour associated to each one of them. It follows a response sample:

```
{
    ”classes”: [ ”person”, ”bicycle”, ”car”, ”motorbike”, ”
        aeroplane”, ... ]
    ”colors”: [[255, 38, 0], [0, 63, 255], ... ]
 }
```

The client can get a prediction by sending an image to `/prediction`. It is also possible to specify whatever the image is compressed or no and if the server should resize it before feeding it to the model. In our case, we send a compressed image directly taken from the MightyThymio after subscribing to the `/camera/compressed` topic and we define the resize size to (190,190). The small size is due to low computational power since we had no access to a GPU. By sanding the same image showed in Figure II we get the following response.

```
{’res’: [{’boxes’: [96.21580505371094,
                    219.6134033203125,
                    437.66387939453125,
                    594.16748046875],
          ’class’: ’bicycle’,
          ’class_idx’: 1,
          ’score’: 0.48406994342803955}, ...
         ]
 ’time’: 0.174407958984375}
```

For each detected class we return the bounding boxes, the *boxes* key of the JSON, the class name, its index, and the score, the confidence of the model. Each box is composed of the top, left, bottom and the right offsets of the bounding box with respect to the left of the image and the top. In the above example, the bicycle's bounding starts 96 pixels from the left and 219 from the top.

III.

IV. Implementation

We design our ROS node to find a specified object and follow it until it is close enough. In addition, we also implemented a PID based algorithm to avoid obstacles and to stop before crash into the target object. From a high-level view, the robot can be represented as a state machine with three states.

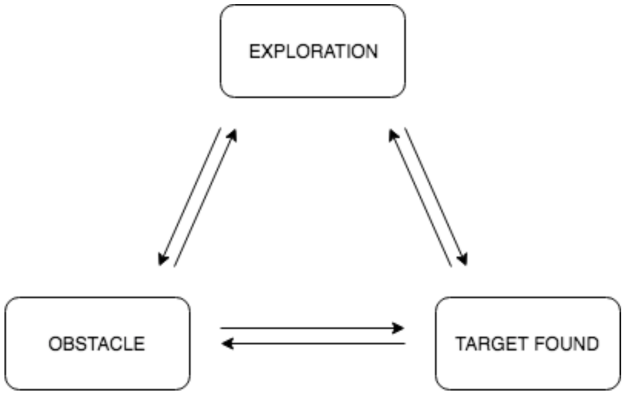

Fig. 3. State Machine

Thus, at any time the machine can be **only** in one of the three states: *exploration*, *obstacle* and *target found*.

a) Exploration: The robot starts to spin on itself with a fixed angular velocity to scan the surrounding until it finds the target or it detect an obstacle.

b) Target Found: When a target object is found, the robot starts moving towards it by keeping its bounding box center in the camera's center. We used a Proportional Integral Derivative, PID, controller to select the correct angular velocity. Given the camera's image and the bounding box, we define the error at time t as:

$$e_t = (l_b + \frac{w_b}{2}) - \frac{w_c}{2} \quad (1)$$

Where, w_c is the camera's image width, in our case 640, l_b and w_b are the left component and the width of target's bounding box respectively. To find w_b we need to just subtract the left and right component of the box. We are

[2]http://flask.pocoo.org/

not taking the absolute value of the error since the sign is used to decide in which direction to turn. Before feeding the error to the PID controller we normalize it between zero and one by dividing by w_c, otherwise, we should have defined a very small proportional parameter for the PID.

After computing the error for the current box, we pass it as parameter to a PID with a proportional parameter set to 0.5, and integral to 0 and a derivative to 3. However, even if we did not do any hyperparameter search, we notice that even with smaller values of the derivative component of the PID the robot works properly.

The following figure shows the error for each frame given different position of the target class *bottle*.

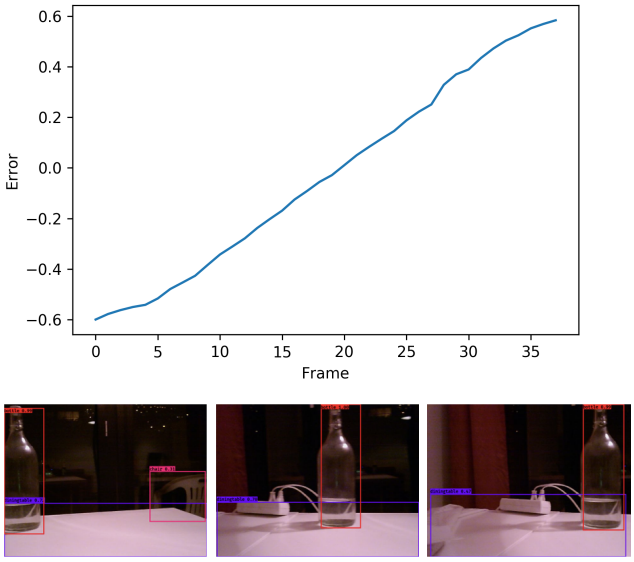

Fig. 4. Bounding Box Error

The absolute value of the error is bigger when the bottle is on the left on the right and it is zero when it is in the center of the image. We used it as angular velocity in order to keep the robot pointed to the target object. We move it forward with a fixed linear velocity of 0.1.

With these settings we were able to smoothly guide the robot to the target, in the following picture we shown some example frame when the Thymio first see a *cup* on the left and then it moves towards it by keeping it in the middle.

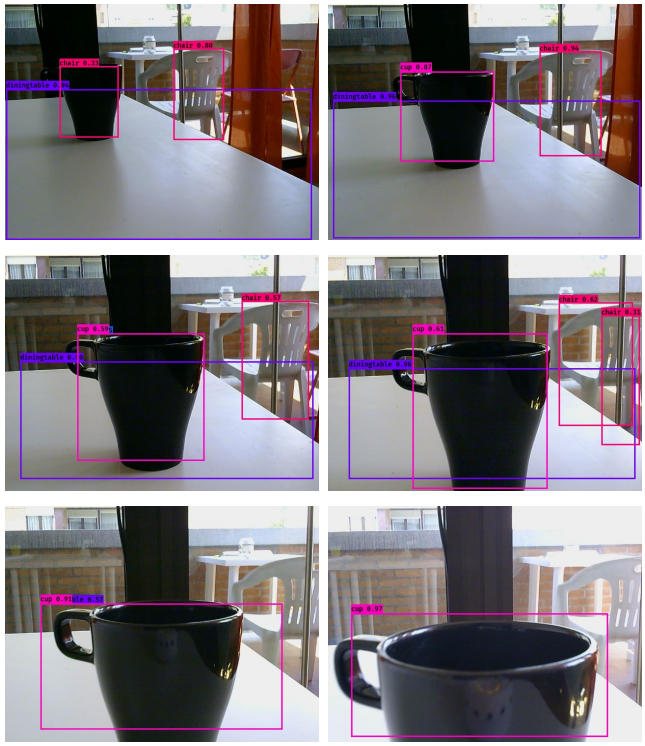

Fig. 5. Thymio finds and go to a *cup*

c) Obstacle: When proximity sensors, five in the front and two in the back, detect something, the robot stops any activity and goes into the *Obstacle* state. We implemented a simple yet effective PID controllers based algorithm to move the Thymio await from any object in front or behind it.

To accomplish this task, we defined two PIDs, one for the linear velocity and one for the angular. The errors are e_l for the linear PID and e_a for the angular PID, defined as follow. We denote the front sensor set as S_f and the rear sensor set as S_r, where each element in the set is a double representing the value from the sensor. In our case the cardinality of each set are $|S_f| = 5$ and $|S_r| = 2$.

$$e_l = \sum_{i=0}^{|S_f|} S_f(i) - \sum_{i=0}^{|R_f|} R_f(i) \tag{2}$$

$$e_a = (\sum_{i=0}^{\frac{|S_f|}{2}} S_f(i) - \sum_{i=\frac{|S_f|}{2}}^{|S_f|} S_f(i)) - (\sum_{i=0}^{\frac{|R_f|}{2}} R_f(i) - \sum_{i=\frac{|R_f|}{2}}^{|R_f|} R_f(i)) \tag{3}$$

The linear error, e_l, is just the sum of all the frontal sensors values minus the rear sensors values. The angular error is defined as the difference between each pair of sensors on the left and the right. In booth equation, we subtract the frontal values from the rear in order to invert the sign of the PID output so when there is an obstacle in front of the robot, the velocity is negative and vice-versa.

As user feedback, we turn on the five frontal Thymio's LEDs. The number and position of the LEDs are based on the width of target's bounding box. For example, a big object on the left side will turn on two LEDs, while a smaller only one. The following figure shows an example where the robot is looking for a *cup*:

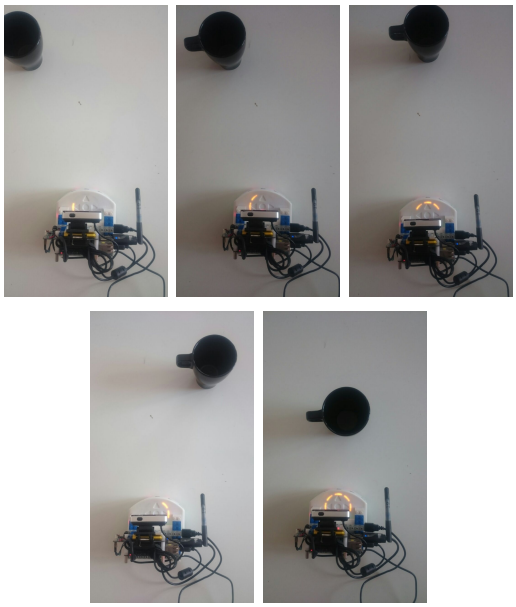

Fig. 6. On board LEDs

V. Conclusion

In this paper, we presented a light and scalable architecture that uses a real-time Object Detection model to correctly find objects in the surrounding environment and to perform actions based on them. Moreover, because the only hardware constraint is the camera, our work can be used into similar robots running ROS to extend their functionalities. Furthermore, since we used YOLO to perform the classification task, the model can be easily re-trained with other classes to increase the context domain. This symbiosis between Machine Learning and Robotic highlights the power of both areas by creating new interesting use cases and scenarios in which both disciplines can shine.

References

[1] Tsung-Yi Lin Michael Maire Serge Belongie Lubomir Bourdev Ross Girshick James Hays Pietro Perona Deva Ramanan C. Lawrence Zitnick Piotr Dollar. "Microsoft COCO: Common Objects in Contex". In: 2015.

[2] Jerome Guzzi et al. "Mighty Thymio for Higher-Level Robotics Education". In: *AAAI*. 2018.

[3] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: `https://www.tensorflow.org/`.

[4] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: 2016.