UNIVERSITÀ DEGLI STUDI DI SALERNO

Information and Electrical Engineering Department and Applied Mathematics

Master's degree course in computer engineering

Cognitive Robotics

FINAL PROJECT

Social Pepper

Fina Vittorio
Puzo Giovanni
Russomanno Vincenzo
Ventre Salvatore

Problem Description

A brief introduction to the problem is deemed necessary to understand the context in which we are moving.

1. The Task

The main task is to implement a ROS[1] architecture that makes the robot Pepper[2] capable of recognize objects in three scenes and say a sentence that summarizes everything it has seen. The different scene must be captured by the robot by rotating its head and look around for objects. In particular, the architecture must provide *4 nodes*, each of which is assigned to fulfil a certain purpose, and which will need to communicate with one or more of the remaining nodes. Here follows a brief description of the nodes requested:

- **Node 1**: Pepper needs to acquire the video stream from its camera.
- **Node 2**: Pepper rotates the head in order to see on the front, on the left and then on the right.
- **Node 3**: Pepper must recognize the object in the scene.
- **Node 4**: Pepper must be able to speak and say whatever it has just seen.

The implementation of the third node shall be carried out using an *Object Detector* trained on the *COCO Dataset*.

2. The Environment

Briefly, we describe the technologies used to develop the problem are described below, including ROS and the framework made available for integration with the Pepper system. All the elements used were fundamental for the cooperation and the achievement of the objective set by the task.

The first element referred to is ROS, the operating system used for robotic programming. ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management [1]. Its use was fundamental to make the architecture usable with the framework made available for interaction with Pepper.

We are talking about **PyNaoQi**, a framework written in *Python* that implements the interfaces and methods necessary to create Pepper applications in the ROS environment. NaoQi is the main module and includes objects such as *ALProxy*, *ALBroker* and *ALModule*. The first one, in particular, was used in this project to interface the application created with the robot's speech modules, for example.

The requests for the development of our solution are *Python*, *ROS Melodic* and the *PyNaoQi* framework.

[1] ROS: The Robot Operating System is a set of libraries and tools to build a robot application.
[2] For further information visit http://doc.aldebaran.com/2-5/home_pepper.hmtl

Implemented architecture

Below is a detailed report on the implementation choices made and the description of the architecture applied to develop the solution to the problem posed.

1. Architecture Design

The developed architecture foresees the use of *4 fundamental nodes*, of which a first description was given in the previous section. Some of them cooperate through the use of topic and *publisher/subscriber* communication technique to allow Pepper to perform a given task. We can therefore distinguish the nodes as follows:

- **Node 1**: the `camera_acquisition` node has the goal of managing the aspects related to Pepper's front camera, whether it is image acquisition or showing the video stream. It contains two executables and makes use of a custom message created for interacting with the node responsible for the movement of the head. Among the two executables, the main one is `image_acquire` which takes care of capturing the image useful for performing object detection on it. The custom message `ImageWithPose.msg` represents the image captured by the camera and the pose related as a String. Essentially the node communicates via three topics. The first is essential to receive the images published by the front camera of the robot. The second to know when the pepper head is in the right position to retrieve the image and the third to transmit the image on which to operate the detection. They are respectively: `/pepper_robot/camera/front/camera/image_raw`, `/pepper_pose` and `image_capture`.
- **Node 2**: the `head_movement` node has the goal of managing the aspects related to Pepper's head movements and communicate with the topic responsible for joints movement and let Pepper look around himself. It includes only one executable where the movements are managed by publishing messages on the topic `/pepper_robot/pose/joint_angles`, sending `JointAnglesWithSpeed` messages contained in the `naoqi_bridge_msgs` package.
- **Node 3**: the `animated_say` node has the goal of managing the aspects related to Pepper's capabilities related to speaking. It contains two executables that interact with the *ALAnimatedSpeech* proxy to handle Pepper's speech. The first one, `wait_say`, takes care of making Pepper interact by voice while waiting for the detection node to be ready to operate on the images that are captured. The second one, which has the same operating principle as the first, is responsible for forming a meaningful sentence, with all the objects classified by the detector, to be pronounced by the robot. Both refer to the `/pepper_detective` topic which communicates them when the detector has performed the classification of the objects on the images received in the dedicated topic and you can then let the robot speak.
- **Node 4**: the `pepper_detector` node has the goal of managing the aspects related to Pepper's capabilities related to the detection of the objects in a scene. A detailed description of the detector used will be given in the next section, but we will briefly describe the operations that the running node performs. The custom DetectionWithPose.msg message has been created to represent and include in the message all the classifications related to a given pose. The executable interacts with the topics `/pepper_detective` and `/image_acquire`. The first communicates customized messages containing the detections and the first contains Image type messages to retrieve the images on which to perform the detections.

The architecture just described moves and interacts with the ROS core according to the following figure.

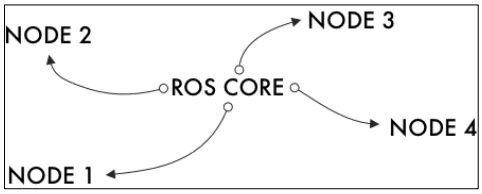

Figure 1. Implemented Architecture

Now we can give a detailed description of the communication protocols used to make the created nodes interact, including *publishing and subscribing services* with related *topics*. As regards the node that takes care of acquiring the video stream from the camera, the node implemented by us will be shown in the figure of the architecture, however, due to problems due to the execution of this node on the machine connected to Pepper, we have chosen to have the launch file the pre-installed ROS node `/rqt_image_view`.

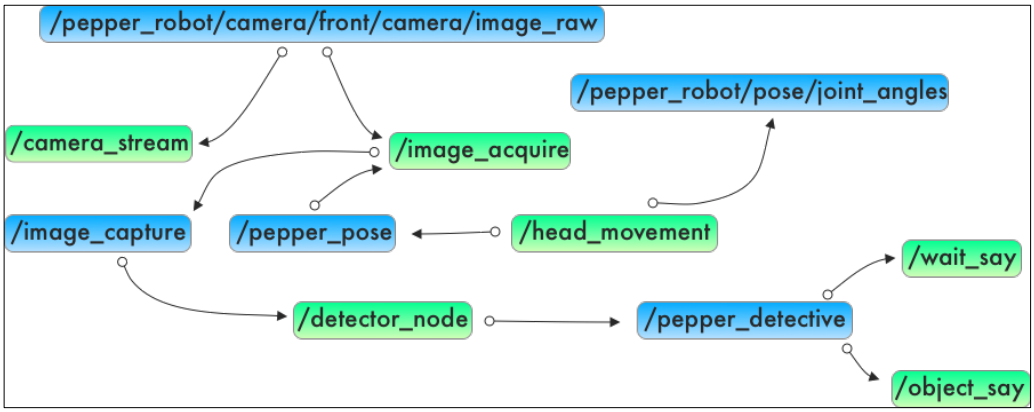

Figure 2. Communication protocol between nodes

In the picture you can see the running nodes in *green* and the topics that allow interactions in *blue*. An arrow going from a topic to a node says that that node is initialized as a *subscriber*, in the opposite case as a *publisher*. As you can see, we have chosen to implement only **publisher/subscriber** architectures through the use of some custom topics in addition to the standard ones provided for Pepper management. This is because the task carried out does not expect to receive an instant response and the designed architecture has not been designated as a client-server structure (for the detector for example), but we have taken care to first store the images due to the rotation of Pepper's head and then apply the detection.

The reason in support of this choice is Pepper's architecture, which envisages loading times of the model which are not very fast and therefore the impossibility of being able to apply a service protocol that reacts immediately upon request by processing and sending a response.

We therefore describe the topics and the interaction with the nodes that can be seen in the previous figure. The first of these is the topic prepared for Pepper's camera stream: `/pepper_robot/camera/front/camera/image_raw`. Both nodes `/camera_stream` and `/image_acquire` both register as subscribers to this topic, the first to show the video stream and the second to acquire the image to be managed by the detector. The second topic already prepared for the task is `/pepper_robot/pose/joint_angles` on which messages are published to request the movement of Pepper's head.

Now let's discuss the custom topics that have been made for the purpose. The first of these is `/image_capture`, the topic on which the images are published and then retrieved by the

/detector_node node, initialized as a subscriber to this topic. The node just mentioned, on the other hand, interacts with the /pepper_detective topic by sending custom messages that include the pose and the detection recovered from the relative image. The two nodes /wait_say and /object_say then retrieve information from this topic, where in particular the second one reconstructs the sentence that includes the objects found to make the robot pronounce it. The last topic created ad hoc is /pepper_pose with which the /image_acquire node interacts to know the reference position of a captured image and the /head_movement node interacts as a publisher to communicate the current pose of the robot.

Object Detector

We have chosen to use one from a collection of detection models pre-trained on the COCO 2017 dataset[3]. These models can be useful for out-of-the-box inference if you are interested in categories already in those datasets. They are also useful for initializing your models when training on novel datasets, but in our project, simply, we used a model pre-trained for the detections.
The choice of the model was based on the one that could seem the most performing, compromising with the processing time and the score in terms of COCO mAP. Another aspect to underline, is the loading time of the model when the "startup" signal is given to Pepper. In fact, the size of the model is a discriminating feature for the detector's choice as it takes some minutes to load it into the robot.
Finally, the interest also focused on the output given by the model, as for the task to be solved it is appropriate to use the *Boxes*. Based on the above, two object recognition models have been considered: **EfficienDet-D0** and **EfficienDet-D1**.

Model	test-dev AP	test-dev AP_{50}	test-dev AP_{75}	val AP	Params
EfficientDet-D0 (512)	**34.6**	**53.0**	**37.1**	**34.3**	**3.9M**
YOLOv3 [34]	33.0	57.9	34.4	-	-
EfficientDet-D1 (640)	**40.5**	**59.1**	**43.7**	**40.2**	**6.6M**
RetinaNet-R50 (640) [24]	39.2	58.0	42.3	39.2	34M
RetinaNet-R101 (640)[24]	39.9	58.5	43.0	39.8	53M

*Figure 3. EfficientDet performance results on COCO – **test-dev** is the COCO test set and **val** is the validation set.*

1. EfficientDet D1

In the end, the model chosen was the ***EfficienDet-D1-coco17*** with 32 TPUs (tensor processing unit) and with these significant features:

Model Name	Speed (ms)	COCO mAP	Input size	Outputs
EfficienDet-D1	54	38.4	640x640	Boxes

The reasons of why it was the most convincing model for the development of the task, will be now explained. First, although Efficientdet-D0 has a faster execution speed of 15ms, Efficiendet-D1 has a superior precision mAP (Mean Average Precision) with a larger input size.
Subsequently, even the study on the architectural features of the model confirmed the choice taken previously, as the network follows the one-stage detectors paradigm, it was employed Imagenet-pretrained EfficientNets as the backbone network and BiFPN Layer for the features network.

[3] https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md

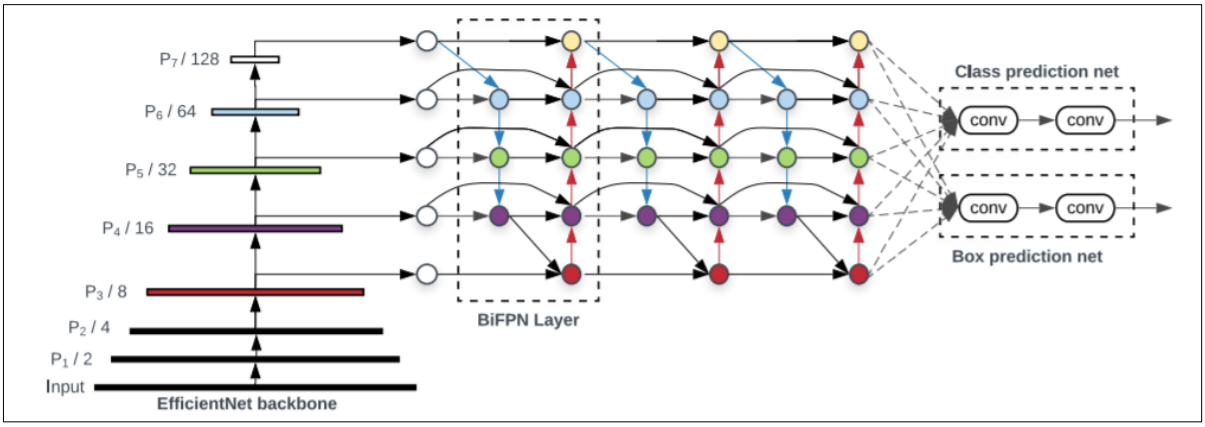

Figure 4. EfficientDet architecture.

Moreover, the features are fed to a class and box network to produce object class and bounding box predictions, respectively. Also, the class and box network weights are shared across all levels of features.

Bibliography

[1] «ROS Introduction,» [Online]. Available: https://wiki.ros.org/ROS/Introduction.

[2] T. Mingxing e Q. V. L. Ruoming Pang, «EfficientDet: Scalable and Efficient Object Detection,» 27 Luglio 2020. [Online]. Available: https://arxiv.org/pdf/1911.09070.pdf.