Task-based control of a humanoid robot in the SPL RoboCup environment

Emanuele Musumeci, 1653885

I. Introduction

In modern Robotics, mostly due to the gap between hardware efficiency (slowly reaching its peak) and software complexity (lagging behind), it is difficult to come up with applications that are able to make advanced technologies, such as autonomous humanoid robots, readily interactive and interesting to the final user, set aside the many research applications currently being explored.
This project tries to find a way to deliver a Human-Robot Interaction flavour to the SPL RoboCup Soccer environment.

II. Background

A. RoboCup Standard Platform League (SPL)

The Standard Platform League (SPL) [1] is a branch of the RoboCup Soccer tournament, taking place in the context of RoboCup, an international event that involves universities and research institutions from all over the world, in different competitions, mainly featuring Robotics applications. The SPL features teams of 5 NAO robots, considered as a standard platform, competing in soccer matches with a specialized rule book.
The human is not considered an active part of these matches, apart from the referee, that ensures rules are followed throughout the games.
The passive role of the public is probably one of the obstacles in building hype and interest in this unusual application of advanced robotics technologies.

B. Motivation

The purpose of this project is to experiment with a plausible way of interacting with the soccer-playing NAO robots through a simple task-based control paradigm:

a) A library of known tasks is implemented in the robot control software through high-level FSM-based behaviors, that are reactive to the current conditions on the playing field, depending on the robot's Knowledge Base, constantly updated by lower-level software (described in the following sections).
b) The human is enabled to give tasks to the robot and control their execution using a simple GUI on an external device (that can be a laptop, tablet or phone).
c) The robot will engage the human and attract its attention with simple verbal or gesture-based interaction (e.g. pointing at its destination or at the target where the ball has to be kicked), while keeping it informed of the current state of execution.

Please notice that at the current state of development, such an application is still not yet enabled for a close-contact interaction: a certain safety distance has to be kept from the robot (even though, given its dimensions and weight, it can rarely pose any danger to an adult).

III. Robotic platform

The NAO v6 robot is a programmable humanoid robot resembling a child, developed by Aldebaran Robotics, a French robotics company, subsequently acquired by SoftBank Group and rebranded as SoftBank Robotics. The robot's dimensions are approximately 57.4 x 29 x 273 cm, weighing approx 5.48 kg. It's provided with an Intel Atom E3845 Quad Core CPU running at 1.91 GHz, 4GB DDR3 RAM and a 32 GB SSD. The robot has 25 degrees of freedom (which in the control framework are managed as 23 physical joints + 1 virtual joint for both hips). It is provided with a big variety of sensors. The most important ones for this project are the following:

- 36 MR Encoders to detect joints positions
- An IMU featuring a 3-axis gyrometer and 3-axis accelerometer
- 8 Force Sensitive Resistors located on the bottom of the robot's feet to detect ground contact
- 2 Cameras both with a maximum resolution of 640 x 480 at 30 fps and a vertical FOV of 43.7°:
  - An **upper camera** located on the *forehead*, horizontal wrt to the ground if the head is perfectly aligned

– A **lower camera** located on the *chin*, tilted by 39.7° wrt to the upper camera

IV. Project setup

The project structure can be easily divided into a *backend*, the robot control framework running on the embedded computing systems of the robot, and a *frontend*, comprising the GUI, an HTML5 and JavaScript-based interface and the network infrastructure acting as a middleware between the robot and the GUI. A more in-depth description follows in Section V, Section VI and Section VII, (backend) and Section VIII (frontend).

V. Robot control framework

The Robot control framework is based on the framework used by the SPQRTeam (RoCoCo Lab, Sapienza University of Rome) [2] during the latest (June 2021) SPL RoboCup edition. This framework is in turn based on the B-Human framework by the B-Human SPL RoboCup team from the University of Bremen [3].

The software framework is structured as a collection of modules, arranged in four different threads (each one run on a different CPU core of the robot):

- The **Upper** and **Lower** threads respectively deal with images coming from the upper and lower camera. Frames are segmented and information is extracted based also on data from the **Cognition** thread.
- The **Cognition** thread computes and updates a world model, determined by data received from the **Upper** and **Lower** threads and by encoder data coming from the **Motion** thread. This thread also uses the *world model* to determine high-level motion commands to send to the *Motion* thread.
- The **Motion** thread controls the 25 robot joints according to the motion commands received by the **Cognition** and exchanges sensor data with the rest of the framework.

The shared memory where all these threads exchange data is called the ”***blackboard***” and is accessed by the framework ***modules***, which store data as serializable data structures called ”***representations***”.

The framework is implemented in C++, with a heavy usage of macros to correctly define *Modules*, *Representations* and other serializable data structures. The serializability is crucial to ensure the streaming of data over network between robots.

In fact the framework is run in a distributed fashion, where each robot takes decisions autonomously in the team, towards the collective strategy. Mutual exclusion of roles taken by robots in the team (e.g. *goalie*, *striker*, *defender*) is enforced by a shared context coordination algorithm. Such an algorithm determines robot roles in the team in a mutually exclusive way depending on a utility value computed locally by each robot for each other robot in the team, based on the current perceived world model, both locally (determined by the observation of the local robot) and globally (determined by the observations shared by each robot over the network), in such a way that the assigned roles match on all the instances of this distributed algorithm.

A. Behavior Control system

Most importantly, the *Cognition* thread runs the **Behavior control system**, a system based on Finite-State Machines, that models the *behavior* of the robot, the reactive part of the framework that determines the actions executed by the robot based on its knowledge-base and the current state of the world.
Each robot role has a different collections of behaviors, called a ”***decks***” (one for each robot role), from which a software module called the ”***card dealer***” extracts a behavior called a ”***card***”. The extraction criterion is: start from the top of the deck and extract the first behavior with true *pre-conditions*.
Each *Card* is structured as a **Finite-State Machine**, with *states* and *transitions* triggered by certain conditions. *States*, in particular, are blocks of code that can schedule the execution of a wide range of *actions*, called ”***skills***”, including joint control actions or more high-level motion commands, that are also structured as Finite-State Machines. *Cards* only have initial states but no terminal states: they are triggered by their *pre-conditions* and are terminated when *pre-conditions* are not met anymore or their *post-conditions* are met instead.
Skills, on the other hand, have to be called and have initial and terminal states.

VI. Perception pipeline

Particular attention has to be given to the vision pipeline, given that most of the information used in the *Cognition* modules comes from the world model. The robot cameras and foot pressure sensors are the only exteroceptive sensors used by the framework (set aside microphones used to detect the referee’s whistle). In particular the cameras are crucial for the robot’s ability to self-localize in the field, detect the ball and the other robots.

A. Cameras setup

The two cameras are tilted in such a way that their fields-of-view intersect, but not enough to use this intersection for stereo vision. Moreover, the frames received by these cameras are not synchronized on hardware level. Therefore both cameras are used separately as monocular sensors.

Frames are received at a fixed rate of 30 fps, in YUV422 color format. While frames from the upper camera are potentially used to detect features in the whole field, therefore a resolution of 640 x 480 is required, frames from the lower camera are used for detection of nearby objects, therefore a 320 x 240 resolution is enough.
The calibration process can be performed using specific modules of the framework but it is rarely necessary: in fact the camera matrix used is the one distributed with the framework itself. The only useful calibration process that is repeated every time lighting conditions change considerably is the Color Calibration, that changes thresholds used in the segmentation process (more on this later).

B. Image color management

1) Color space conversion: The frames are sent by the *Video for Linux* drivers in YUV422 color space, which can be imagined as a cube, where the *Y* component constitues the *luminance* channel, being the height of a "*color plane*" in the color space, and the *U* and *V* components are instead coordinates on this plane, called *chrominance*, determining the actual color at the given luminance.
Frames from this color space are converted into YHS2 color space, that can be instead imagined as a cylinder, where still the *Y* luminance component is the height of the color plane, and the *H* hue and *S* saturation components are now polar coordinates on the color plane, with H being the angle and S the distance from the origin.

2) Color classification: After this color conversion the image is color-classified, meaning that each pixel color is converted to a fixed color value, among Black, White, Green or "None" (displayed as Gray). The algorithm used in this process is described in Figure 1. All thresholds are derived from the manual color calibration process, which should be repeated every time the lighting conditions change considerably.

C. Coordinate Systems and Image clipping

The coordinate systems used in the framework are three:

- A **global** coordinate system, centered in the center of the field
- A **local** c. s., centered in the center of the robot's shadow, with the x-axis outgoing from the front of the robot, the y-axis parallel to the robot's body, oriented to its left and the z-axis vertical, oriented upwards.
- An **image** c. s., which is derived by a two step process:
  1) First the horizon is found in every frame, using a vertical scan-line approach
  2) Then the transformation matrix between the robot frame of reference and the image f.o.r. is computed using the camera matrix.

Using this last transformation, it is possible to clip out the robot's body (which has a known mesh) from the images.

D. Segmentation

Features are extracted from the image following a multiple step segmentation process:

1) In order to find the *field boundary* a *scan-line approach* is used: pixels are scanned vertically along vertical scan-lines whose density increases with the distance (based on the *Image Coordinate System*). The field boundary is detected whenever there is a discontinuity in the classified image color by accumulating a "score" along each scan-line, increased for each green pixel and decreased otherwise: all spots where the peak for this score is reached are considered "candidates". Scanning starts at the lower border of the image (also excluding the robot's body) and end at the position in the image the field boundary would appear if the robot were located in one corner of the field looking at the other corner). Then the most fitting alignment of candidate spots is chosen using a RANSAC-based algorithm [4].
2) Finding *field lines* follows a similar approach.
3) Finding the *ball* uses a *scan-line approach* only to find *candidate spots* in the image that might possibly contain the ball, while the actual classification of these spots (to determine if they contain a ball) and the estimation of the ball position is handled by a neural network. This process is applied to images coming from both cameras (more on this later).
4) Finding obstacles is a process that varies between the two cameras: images coming from the *upper* camera are fed as input to a neural-based object detector, while the lower camera

uses a simple scan-line approach (more on this later).

E. Ball detector

The ball detector used follows a two-step approach:

1) Scan the image vertically using scan lines at two different resolutions (areas in the image that are supposed to be further are scanned with tighter scan lines). Lines are searched for a distribution of white "spots" whose horizontal extension is compatible with that of a ball.
2) A grayscale square patch of the image is extracted with an edge length of 3.5 times the radius of the ball at that distance. This patch is then downscaled to a 32 x 32 pixel patch and fed to the neural network shown in Figure 2 and Figure 3.
   This neural network performs two different downstream tasks:
   - A **Ball Classifier** that gives a prediction value (between 0 and 1) determining the probability p that the input spot contains a ball. In particular:
     a) If $p \geq 0.9$: the spot is determined to contain a ball
     b) If $p \in [0.8, 0.9)$: the patch is saved in a set of "probable balls". If no patch is found in category *(a)* the ball is assumed to be in the "best" patch of this category
     c) If $p \in [0.7, 0.8)$: the patch is treated similarly to category *(b)* (so if no patch is found in a better category, the best one of this category is returned).
     a) otherwise the ball is discarded
   - A **Position Estimator** that, given the patch corresponding to a successfully classified ball, infers the 2 coordinates determining the ball position in the local reference frame.

F. Obstacle detector

The detection method used in the upper camera is different by the one in the lower camera. Only the detection method for the upper camera is particularly of interest for the current project, as the obstacles detected in the lower one rarely contribute to the obstacle avoidance (as they are already too close to not have been already detected by the upper cameras).

1) Pre-processing: The grayscale (luminance) image is used in the detection process (as the main features of the robot are of white and gray color), passed as a whole to the neural network, after being resized to a 80 x 60 resolution and having been transformed through a 2% min-max normalization (for a better robustness to lighting conditions).

2) Neural Network: The network represented by the table in Figure 4 then predicts the coordinates of the bounding boxes for all robots in the input image, which are then scaled back to normal resolution.

3) Bounding box post-processing: Given that the bounding boxes are constructed using anchor boxes, the re-scaling increases the error in the bounding box size wrt to the actual detected object. To correct this, a post-processing step is applied using scan-lines from the middle of the bounding box to the lower border of it, to find the correct left and right boundaries of the bounding boxes.
Additional post-processing steps can be applied to detect the jersey color but this is not crucial for the current project.

VII. Additions to the framework

The framework structure described up until now is used as is or slightly tuned with some minor changes. The custom additions to the framework, that pertain to this project and constitue the "back-end" part, are instead reported below (for a more comprehensive list of the files that have been heavily edited or added to the framework check Appendix A).

A. Ball carrier

The basic framework, lacks behaviors, because the strategy is generally developed by each RoboCup team and is updated and tuned each year, trying to integrate new technologies or improve from the mistakes of the previous competition. One of the additions to our framework for RoboCup 2021 was a "*ball carrier*" behavior, allowing the robot to move the ball on the field while keeping it controlled (e.g. defended against the contrasts of opponent robots).
The idea behind this behavior can be explained as follows:

1) Given the currently seen obstacles, detect the *most accessible targetable area on the opponent goal line*, by projecting all obstacles on the goal line, then assigning a utility value to each targetable gap (computed on the distance with goal posts and other obstacles) and selecting the one with the highest utility.
2) Then compute a *path from the ball to the goal target* (selected in the previous step), avoiding obstacles with a radius that varies dynamically

based on the robot position (e.g. if the robot is very near to the goal, this radius decays to zero) and the distance to the nearest obstacle. The path planning algorithm used in this step is the same one used by the walk path planner in the framework [5], wrapped in an higher level layer that allows customizing the planning process.
3) Then select a waypoint along this path (generally the end of the first segment of this polygonal path) and use it as an immediate target for the ball.
4) Compute an approach point aligned with this target and the ball.
5) Perform all motion commands necessary to align the robot's foot that is nearest to the nearest obstacle and that will be used to kick, with the robot on the approach point computed in the previous step (with tunable offsets between the ball and the foot).
6) Finally perform the motion commands to walk through the ball (propelling it forward).
7) Repeat these steps continuously, everytime with the most recently updated world model. For example, the robot might see a new obstacle or might re-localize, having seen another landmark which corrects the guesses on its position.

To implement this reasoning process, the behavior comprises two main parts:

- The ball carrier **representation** and **module**, provide at all times the position to correctly approach the ball and its immediate target (basically, steps 1-4).
- The ball carrier **behavior**, implemented as a FSM, to actually perform the planned approach to the ball (steps 5-6). A representation of the FSM related to this behavior can be seen in Figure 5

B. Task handling

The main focus of this project is to provide a way to give ***tasks*** for the robot to complete autonomously. While the network "middleware", implemented to deliver these orders to the robots, will be described in Section VI, a socket-based communication protocol has been implemented for the framework to receive and store these orders.

Each robot is assigned a known static IP when the framework is copied on it. Moreover, each robot is statically assigned a team number.
Knowing this, the ports for the receiving and the sending socket respectively are computed, univocally for each robot, based on the robot's team number (in a way that is repeatable in the other nodes of the network as well).
The next node along the communication infrastructure is a controller server implemented in Python.
The communication protocol is very simple and based on strings. The following kinds of messages are exchanged:

- A *keepalive message* is exchanged between the robot and the controller server at regular intervals of time, such that the controller server can determine if for any reason the robot is inactive (e.g. the framework crashed for some bug or the robot ran out of battery).
- A *task queue*, which is usually provided by the Python server (and is deleted whenever the robot becomes inactive). If the server crashed instead or it is started when the robot already has assigned tasks (which basically means it crashed before, as the only way to assign tasks to the robot is through this server), the server will request the current task queue to the robot and update its local copy accordingly.
- Data on the *ball position*, *robot position and orientation* and *obstacles positions* are streamed at regular intervals of time by the robot to the Python server (which then propagates it to the GUI).
- A command to *reset the task list*, to *delete a single task* as well as to *repeat the initial instructions speech* (more on this in the section about interaction).

A **task** is a queue of **actions** to execute in succession, each action optionally specifying a *robot destination pose* and a *ball destination pose* (depending on the action type).
Behaviors are associated to actions almost in a one-to-one correspondence, requiring a specific action type in their pre-conditions.
Actions can be considered the atomic executable unit of this system and are completed depending on pre-defined conditions (different for each action type): when the robot reaches a position, when the ball reaches a position or simply when the underlying behavior notifies this system that the action has been completed (calling a specific function of this library).
Two classes of tasks can be issued: *Simple* tasks which require the completion of one or more basic actions (e.g. "reaching a certain position"), *Complex* tasks, which are composed by several simple tasks.
A comprehensive list follows:

- **Simple** tasks:
  - *Go to position*: the robot will move to the requested position
  - *Kick ball to position*: the robot will kick the ball to the requested position
  - *Carry ball to position*: the robot will carry the ball to the requested position
- **Complex** tasks:
  - *Score a goal*: as of now, the only assignable complex task requires the robot to score a goal. It is composed by the "Reach ball" simple task and then the "Kick to goal" simple task (which are not selectable as a simple tasks).

The task queue streamed by the Python server contains a list of tasks, each one with a unique ID. The robot keeps track of the last *received* task ID as well as the last *completed* task ID (which is the ID of the last task that has been carried out) Whenever the robot receives a new task queue, it will update its local copy of the task queue by adding the leading tasks whose ID is higher than the one of the last completed task.

C. Interactions

The RoboCup framework doesn't really offer flexibility in terms of Human-Robot interactivity, as its prime purpose is to adhere perfectly to the official rule book of the SPL RoboCup, therefore the Human-Robot interaction part of this framework is very simple. There are four kinds of interaction:

- **Instructions speech**: at the beginning of the HRI routine, the robot will perform a speech telling the user how the interaction routine works and telling instructions on how to use the GUI. A smaller version of this speech can be played by pressing the "Tell instructions" blue button on the GUI. When performing this speech, the robot will turn its body and head to the user's position.
- **Position-based action announcement**: whenever the robot begins an action that requires it to reach a position or move the ball to a position, it will announce so by saying an action-specific line (e.g. "I'm going there" for the "Go to position" task) while pointing with its arm at the destination and looking at the user (only the head is facing the user while the body is aligned with the target).
- **Non position-based action announcement**: for any other action that doesn't require the user to issue a position (e.g. "Kick to goal", which is not selectable by the user and instead is issued automatically when issuing a "Score a goal" complex task, implicitly uses the position returned by the utility-based goal target selection), the robot will turn its head looking at the user and say the action-specific line, without pointing at a destination.
- **Generic announcement**: announcements that are not task or action-specific but refer to the overall HRI routine execution only require the robot to say something (e.g. "Task completed" when a task is finished).

Notice that the interactions are hard-coded in the Finite-State Machines of the robot behaviors.
The speech is pre-synthesized, using the **gTTS** ("Google Text-To-Speech") Python package, saved as audio files. The arm pointing is obtained as a motion "*Skill*".

VIII. Frontend

The *frontend* comprises a small network infrastructure to allow delivering orders from the GUI, made of several nodes for increased modularity and extendability, and an HTML5/Javascript-based GUI.

A. Network infrastructure

The network infrastructure comprises three nodes (in addition to the robots), as can be seen in Figure 6:

- A **Python server**, that directly communicates with robots, keeping track of their task execution. In particular, a different write and read socket is associated to each robot (the port is computed in a repeatable way in both the robots and the server, based on the robot team number) and the alive state of each robot is checked constantly through the exchange of *keepalive* messages. Whenever a robot goes down, its task queue is reset. If instead a robot has a non-empty task queue when the server is activated (which means that the server previously crashed), the server asks the robot for that task queue and stores it locally.
- A **NodeJS server**, communicating with the Python server and the GUI clients. While it "repeats" messages on both ends, it also keeps track of which clients are connected: multiple clients are allowed but only one client is given control for each robot, while clients that can not be matched to a robot will be disabled (the GUI page just turns into an empty white

page). The communication with clients happens through *websockets*.

- **GUI clients** on devices connected to the same LAN network of the NodeJS server, implemented in HTML5 and Javascript, constantly exchange information with the NodeJS server to update their current view of the field and deliver user commands. Notable parts of this GUI are:
  - A **field view**, representing a 2D top-down view of the field, where the robot position and orientation is represented as well as the ball position and eventual obstacles in the field. Issued tasks are represented here with colored "targets".
  - A **task preview area**, a box located below the field view containing as many boxes as there are tasks in the task queue. Each one of these boxes contains informations on the task and is colored as the target on the field view corresponding to that task. By clicking on one box, the corresponding task is deleted from the task list.
  - A **task selection panel**, located to the right of the field view, contains two tabs: one for the simple tasks and one for the complex ones. Each tab contains one button for each issuable task: when clicking on one of these buttons, if the task is position based, the user will have to click on the field view to select the target position, otherwise the task will be issued straight away. Moreover, each tab always contains one red button to empty the task list and one blue button to have the robot tell instructions to the user.

Not all mobile devices support *websockets* so the usage is currently restricted. The webpage is provided through *http* and is adaptive to the device screen size.

IX. Conclusions

This project is a very basic demonstration of the possibilities given by exploring HRI routines based on autonomous completion of assigned tasks. Even though the execution of the tasks is strictly domain dependent (e.g. robot behaviors), the concept has been interesting for me to explore. A plausible future work, possibly for my thesis, is to improve the library of available tasks and create a different frontend to provide advanced interaction techniques that allow more reasoning on the given tasks and also a more streamlined interaction framework, with the possibility of adding spoken human-robot interaction (or other advanced interaction modes) to improve the usability of this system.

References

[1] Robocup standard platform league (spl). [Online]. Available: https://spl.robocup.org/

[2] Spqr team website. [Online]. Available: http://spqr.diag.uniroma1.it/

[3] Bhuman robocup spl team, university of bremen. [Online]. Available: https://b-human.de/

[4] Bhuman code release 2019. [Online]. Available: https://github.com/bhuman/BHumanCodeRelease/blob/master/CodeRelease2019.pdf

[5] Bhuman code release 2019. [Online]. Available: https://b-human.de/downloads/publications/2018/Winner2017.pdf

Algorithm 1: Color classification from the YHS2 frame to the ECImage color-classified format

```
Data: Input frame $I$ in YHS2 as a collection of pixels $p$, where every pixel has a saturation, a
      luminance and a hue value; A saturation threshold value $S_t$; A luminance threshold $L_t$; A
      minimum green threshold $G_t^{min}$ and a maximum one $G_t^{max}$
Result: Color-classified input frame $I^c$, where each pixel is of a class among Black, White, Green
        and None
foreach $p_i$ in $I$ do
    if p.saturation $\leq S_t$ then
        if p.luminance $\leq L_t$ then                                                          end
            $I_i^c$ = Black
        else
            $I_i^c$ = White
        end
    else
        if p.hue $\in [G_t^{min}, G_t^{max}]$
        then
            $I_i^c$ = Green
        else
            $I_i^c$ = None
        end
    end
```

Fig. 1: In this algorithm, we consider three thresholds: one for the saturation, one for the luminance and one for the hue of the pixel. First the saturation is examined: if the pixel has a saturation below the threshold value, then it is considered a "non-color": it is Black or White based on its luminance value, compared to the luminance threshold. Otherwise (pixel saturated enough) the hue is examined: if it is in the correct range, it is translated to Green, otherwise it is considered None (Gray), as in not useful towards the segmentation. Notice that the thresholds are decided with a color calibration process that is repeated manually every time the lighting conditions change considerably.

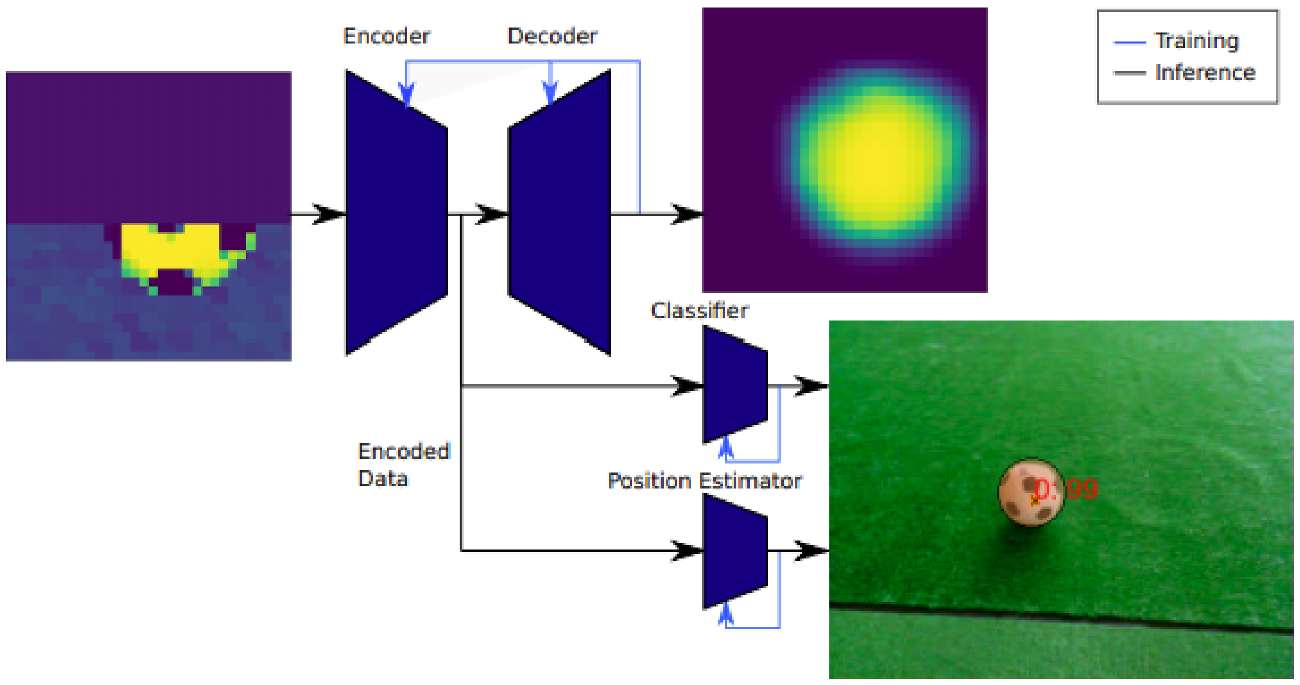

Fig. 2: Scheme representing the Ball perceptor neural network: it's possible to see the Encoder part acting as a feature extractor (the Decoder is just used during training and is of no interest for this project) and the two downstream heads, the Ball classifier and the Position estimator.

Layer Type	Output Size
Input	32x32x1
Convolutional	32x32x8
Batch Normalization	32x32x8
ReLU Activation	32x32x8
Max Pooling	16x16x8
Convolutional	16x16x16
Batch Normalization	16x16x16
ReLU Activation	16x16x16
Max Pooling	8x8x16
Convolutional	8x8x16
Batch Normalization	8x8x16
ReLU Activation	8x8x16
Max Pooling	4x4x16
Convolutional	4x4x32
Batch Normalization	4x4x32
ReLU Activation	4x4x32
Max Pooling	2x2x32

(a) Encoder

Layer Type	Output Size
Input	2x2x32
Flatten	128
Dense + Batch Norm + ReLU	32
Dense + Batch Norm + ReLU	64
Dense + Batch Norm + ReLU	16
Dense + Batch Norm + Sigmoid	1

(b) Ball Classifier

Layer Type	Output Size
Input	2x2x32
Flatten	128
Dense + Batch Norm + ReLU	32
Dense + Batch Norm + ReLU	64
Dense + Batch Norm + ReLU	3

(c) Position Estimator

Fig. 3: Layers of the Ball perceptor neural network.

Layertype	Filters	Filter Size	Strides	Padding	Output
BNorm	-	-	-	-	80×60
Conv	16	3×3	1	Same	80×60
SConv	24	3×3	2	Same	40×30
BNorm	-	-	-	-	40×30
SConv	16	3×3	1	Same	40×30
SConv	20	3×3	1	Same	40×30
SConv	20	3×3	2	Same	20×15
BNorm	-	-	-	-	20×15
SConv	20	3×3	1	Same	20×15
SConv	20	3×3	1	Same	20×15
SConv	24	3×3	1	Same	20×15
SConv	24	3×3	2	Same	10×8
BNorm	-	-	-	-	10×8
Conv	24	3×3	1	Same	10×8
Conv	24	3×3	1	Same	10×8
Conv	24	3×3	1	Same	10×8
Conv	24	3×3	1	Same	10×8
Conv	20	1×1	1	Same	10×8

Fig. 4: Layers of the Obstacle detector neural network.

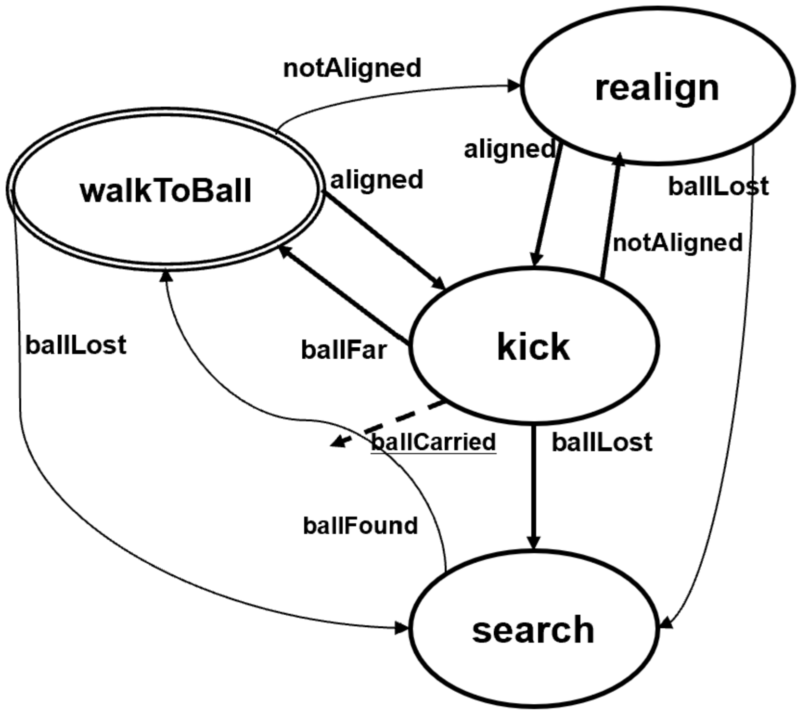

Fig. 5: Representation of the FSM equivalent to the Ball carrier behavior.

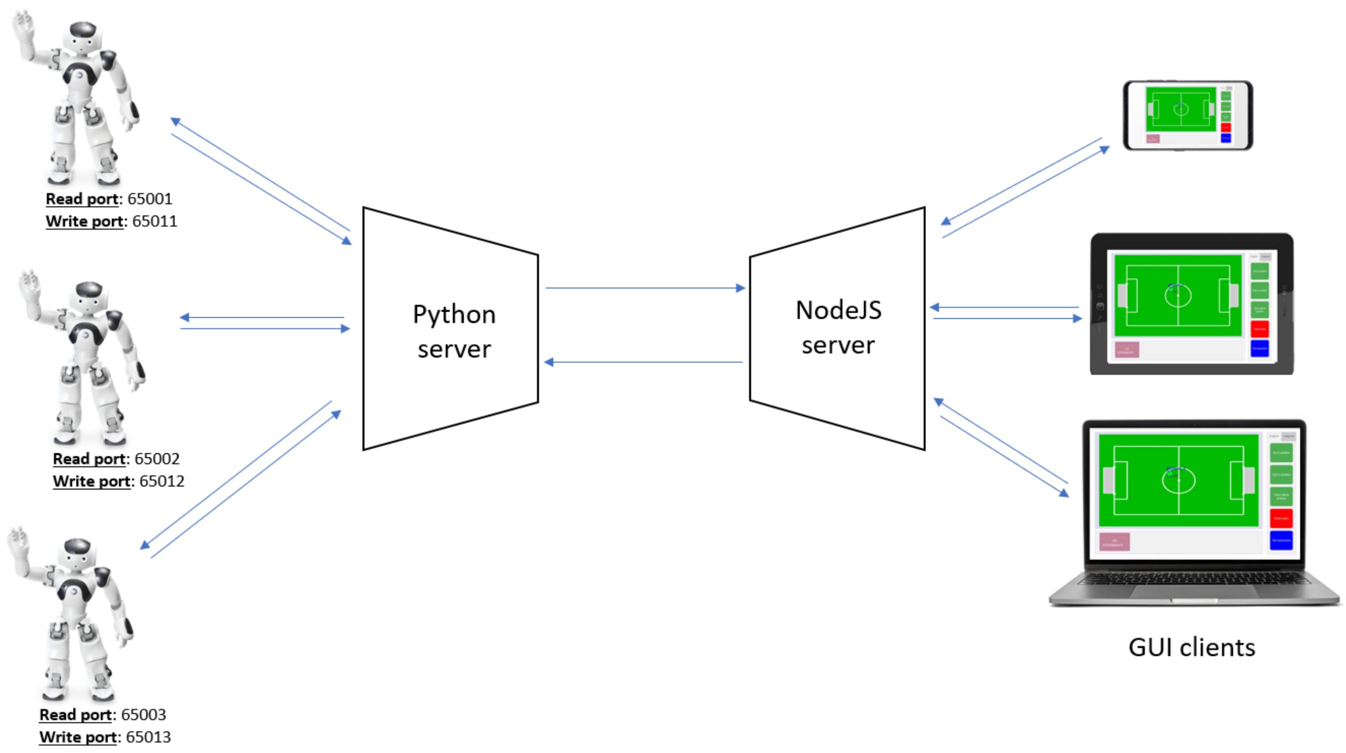

Fig. 6: Scheme of the network communication.