COMILLAS

UNIVERSIDAD PONTIFICIA

ICAI

Máster en ingeniería de telecomunicaciones

Proyecto de fin de máster

Identificación y recogida de objetos con un brazo robótico utilizando técnicas de reinforcement learning

Autor

Pablo Iglesia Fernández-Tresguerres

Dirigido por

Philippe Juhel

Madrid

Enero 2021

ABSTRACT

Abstract content

Thank yous

And other important information

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Robotics and real life are worlds destined to meet. Today everyone has seen robots trying to behave like human beings. Many of them even look similar to a person and try to imitate the way we walk, talk or, ultimately, interact with the environment around us.

Robots, Artificial Intelligence or other concepts such as Machine Learning have crept into our lives in just a few years. In fact, until recently, only a few visionaries like Marvin Minsky or Isaac Asimov used to speak of these concepts, and it was as part of science fiction novels. Nowadays, series like Black Mirror bring this technologies closer to the general public and make us reflect on how the future could be.

However, robots, and artificial intelligence in general, are still far from the vision that is told in the novels. They are not capable of understanding the environment around them, of learning or generalizing as we humans do. Companies and researchers are working on getting better generalization of the algorithms, but the truth is that, so far, Artificial Intelligence is only able to perform specific tasks for which they are programmed.

This project is one of those cases. The goal is to control a UR3 arm robot using Artificial Intelligence in order to pick disordered objects from a box and place them in a point of delivery. This task seems trivial, because we are used to see machines performing pick and place actions in industrial processes, but in fact, these kind of processes are normally just repeating the same action or the same rule over and over again. They are able to perform this tasks because they know apriori where these objects are or how they are placed, but they are not capable of generalizing the workflow.

For instance, in Universal Robot free e-Learning course [1], they expose the following example of an industry pick and place task. In Figure 1.1 we can see how the robot is placing an object in a box located in a conveyor belt. The robot is using an infrared sensor to know that a box has arrived, and this box will always

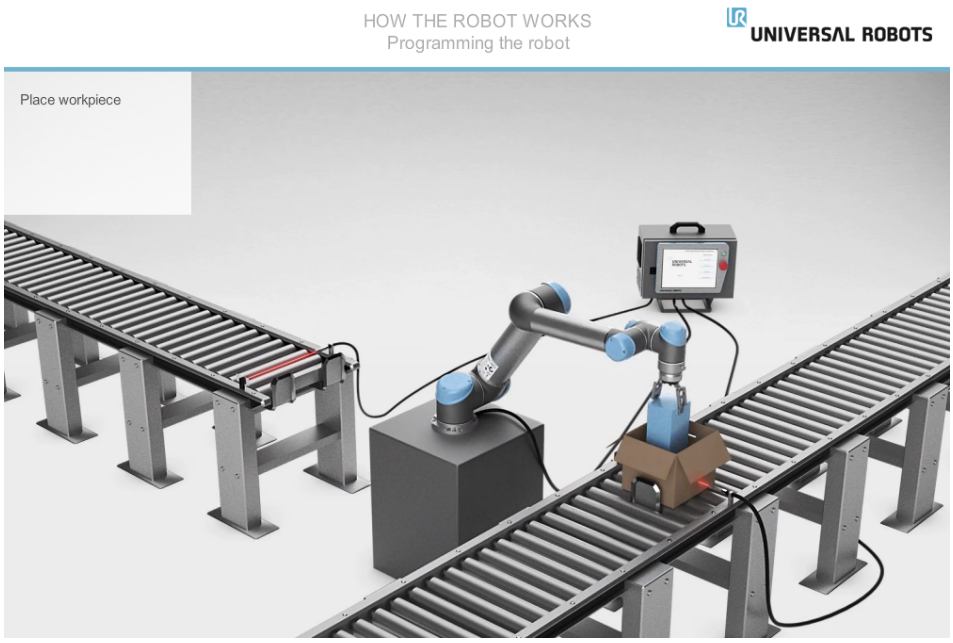

Fig. 1.1: Universal Robots Pick and Place Task

be in the same place because there is a stopper in the conveyor belt which doesn't allow the box to keep moving. On the other hand, the object is picked from the other conveyor belt using the same system to detect the arrival of a new object. The whole task is using a complex architecture, but the robot is performing the same chain of movements in a loop and the only intelligence that the robot has to have is waiting for the object and the box to come.

To achieve generalization in this project, Reinforcement Learning (RL) together with Image Recognition techniques have been used. This algorithms give the robot the ability of calculating, for each time step, the optimal action to achieve the final goal of picking all the objects from the box and placing them in the objective point. To compute this action the robot needs to gather information about the environment such as its relative position over the box or how the pieces are distributed. This information together is called state, and the robot computes each action depending on it.

To perform this project, a distributed architecture with multiple nodes has been created. Each of them takes care of a different activity. For example, some nodes are used to control the robot, others to gather information about the current state, and others are used to train the Artificial Intelligence algorithm. This architecture has been created using ROS (Robot Operative System) and contributes to the

project adding all the advantages of a microservices oriented architecture.

1.1 Project Motivation

The fourth industrial revolution is here, and it will change the way that goods are produced, raising efficiency by increasing the amount of automated processes. This will lead to a faster production and a reduction of errors, as machines have the ability to decide and act in fractions of seconds without making mistakes. Furthermore, machines can also be working 24 hours per day stopping just for maintenance checks, which would help to increase the productivity factor without increasing the expense in human resources.

We have been hearing about industry 4.0 since 2011, but the truth is that it is not a reality yet. We are just in the beginning, and it will take decades to perform such a big change in the industry. There are some factors to take in mind in order to analyse the evolution of the industry in the following years. The improvement on the telecommunications with the arrival of 5G networks, the moral dilemma of substituting workers for machines and the impact that this could have in the society or the improvement and implementation of AI technologies are just some of these factors.

We have seen a lot of Artificial Intelligence algorithms applied to the industry, but the truth is that these technologies are not fully developed yet and just big companies can afford to use them in their supply chain. Besides, there are some task that are now performed by humans and cannot be done by machines due to its complexity or its importance in the whole production chain.

The motivation of this project is to contribute to the industry change providing an open source solution to a complex problem such as disordered pick and place task. This open source solution does not currently exist in the industry and would add value being a good starting point for bigger projects in the future.

2. DESCRIPTION OF TECHNOLOGIES

Describir las tecnologías, protocolos, herramientas específicas, etc. que se vayan a tratar durante el proyecto para facilitar su lectura y comprensión. Hablar de Java no procede aquí porque todo el mundo sabe lo que es, pero si en el proyecto hablo continuamente del protocolo Baseband, debo especificar en este capítulo qué es y para qué sirve.

- ROS + catkin
- pytorch
- arduino
- github
- CUDA
- moveit
- UR driver
- UR3 robot
- gazebo
- arduino
- Reinforcement learning
- anaconda

3. STATE OF THE ART

The pick and place task that is intended to be performed in this thesis is really useful for a lot of applications into the industrial world because it would bring a lot of flexibility for these processes. A example of this applications could be an assembly line, where robotic arms could be picking all the different pieces to assemble in the product using always the same algorithm.

Big companies are developing a lot of Artificial intelligence use cases in the industry, and they try to contribute to the AI community by publishing scientific articles on how they managed to use AI for their specific tasks. Unfortunately, although some companies have already developed their own solutions for our specific pick and place task, none of them have published a scientific article on the subject, making it difficult to study the way they have achieved it.

One of the companies that has already developed a pick and place task is the Japanese automation company Fanuc, which has developed an AI-based solution together with Preferred Networks. As commented before, they have not published any scientific article about the topic but we can see the system working in a video they have posted on YouTube [2]. That means that we have to gather all the possible information from the video, where we could find that they have not used a Reinforcement Learning algorithm but just a Deep Neural Network (DNN) with image recognition.

To train the net, they have collected "success" or "fail" labelled images by making pick actions in random places of the box. Once they gathered a big enough dataset of images, they have trained a Deep Neural Network as the one we see in Figure 3.1, where we can also see that the Neural Net has been trained to predict whether the robot is going to success in a pick action in a specific place or not. Using that net, they can make a heat map of the whole box, predicting the points of maximum probabilities of succeed. As usually, they noticed that the bigger the image dataset, the higher the success ratio. In eight hours, they reach 90% of success, which they say is bigger than the human success ratio.

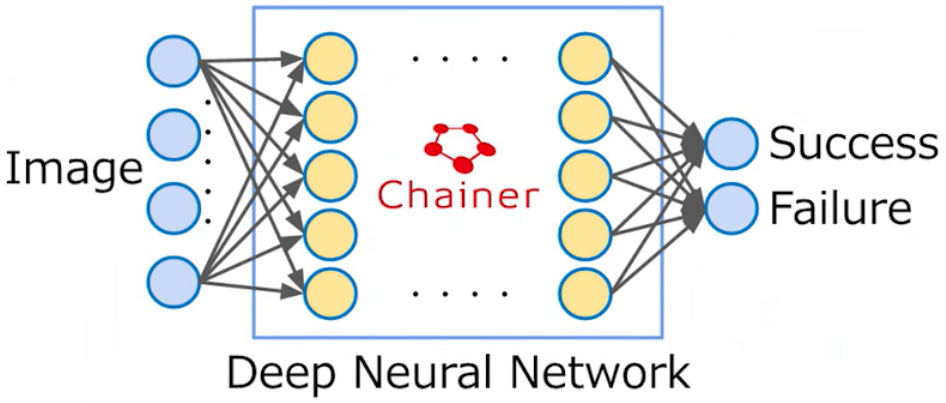

Fig. 3.1: Deep Neural Network of Fanuc solution (taken from the video)

3.0.1 Reinforcement Learning

The idea of the project is to keep using image recognition techniques but, in our case, applied to a **Reinforcement Learning** Algorithm which is an area of machine learning inspired by psychology behavioural. Its goal is to determine what actions a software agent should choose in a given environment in order to maximize some notion of "reward" or accumulated prize.

Explained easily, RL is used to make an **agent** (the robot) learn how to interact with a **environment** in order to perform a task. To achieve this, Markov Decision Process (MDP) which provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker.

Markov Decision Process (MD)

In MDP, the environment is what we are actually trying to simulate with the MDP. The agent will interact with it to learn how to perform the task, so these are the attributes of the environment:

- **Agent:** The agent is the most important piece of the algorithm because it represents the objects that we want to become smarter.
- **Actions (A):** The agent can interact with the environment by performing a set of actions which is normally finite.

– **States (S):** Each time the agent performs an action, it moves to a new state. States are basically the set of information that differentiates the situation of the agent before and after performing an action. States can be transitional or terminal, when the agent meets the objective or when it gets to a forbidden position.

– **Rewards (R):** Each time an action is performed, the agent receives a reward. This reward can be positive, negative or null depending on the impact of the action to achieve the objective.

– **Policy (π):** The policy is used to define the optimal action for each step. It gives a punctuation for all of the actions in the current step as shown in the following formula. The agent takes the action with highest punctuation.

$$\pi(a|s) = P_r\{A_t = a|S_t = s\}$$

The MDP is divided in discrete timesteps (t), where each timestep does not have to last the same time as the previous step. Each timestep, the agent uses the policy π to decide the next action.

Once the action is taken:

– The environment transits to th next state: $\boldsymbol{S_t = S_{t+1}}$.

– Environment produces a new reward, which can be represented with the following formula:

$$\boldsymbol{P(s', r, s, a) = P_r\{S_{t+1} = s', R_{t+1} = r, S_t = s, A_t = a\}}$$

Agent's performance is calculated in terms of its future accumulated rewards known as return. This is called **expected return** an is calculated as shown in the formula below, where γ is the discount factor, and is used to give a bigger value to the closest steps.

$$G_t = \sum_{k=t} \gamma^{k-t} \cdot R_{k+1} \ \forall \ \gamma \in [0, 1]$$

Q-Learning

Now that we know all these concepts, we have to learn what Reinforcement Learning Algorithms do to learn. Basically, **the goal of the agent is to find a policy that maximizes the expected return** . This can be done using different strategies as:

- **Q-Learning:** Estimating action values using Q Tables or other methods
- **TRPO:** Parametrizing the policy and optimizing its parameters

Basic Q-Learning is based on the assumption that both actions and states are limited and that the same action in the same state always drives to the same new state. Having this in mind, Q-learning algorithms build two matrices of shape **length(actions) x length(states)** as shown in the Figure 3.2.

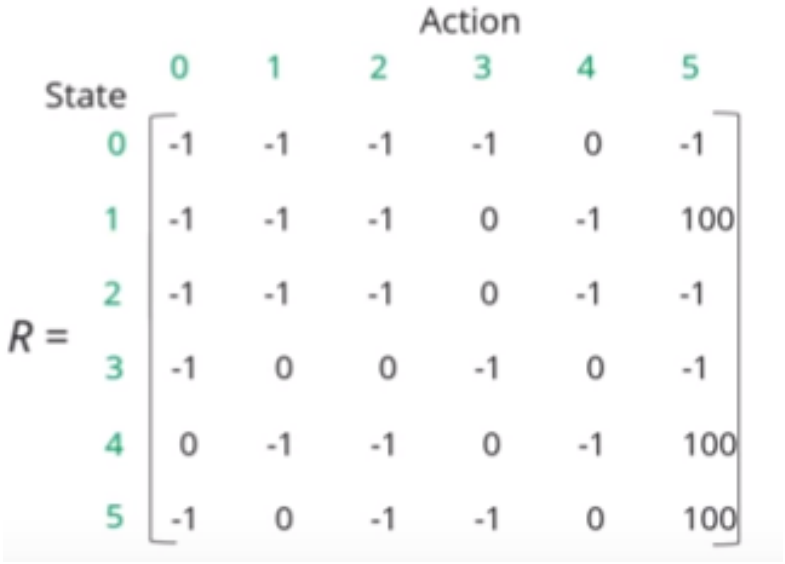

Fig. 3.2: Reward and Q Matrix shape in Basic Reinforcement Learning

In these two matrices, Q-Learning algorithm stores in the R matrix the reward for the pair of action-state while in the Q matrix they store cumulated reward for this same pair. The Q matrix is the one used to decide which action to perform in each state and R matrix the one used to calculate the reward of each action.

However, for the aim of this project, the states of the agent can be different in each timestep. The state would actually be partially formed by images, so the

number of states can be infinite. We need a more complex version of Reinforcement Learning.

3.0.2 Deep Reinforcement Learning

The approach of mixing both image recognition and RL is called Deep Q Learning (DQN) or Double Deep Q Learning (DDQN) depending on the implementation and uses Neural Networks in two different stages of the algorithm. Firstly, a Convolutional Neural Network (CNN) is used to extract image features, and then, a Deep Neural Network (DNN) is used to calculate the q value of each independent action and select the next one using these values.

DQL was proposed in 2012, and, since then, it has been used for a lot of different purposes. For example, Guillaume Lample and Devendra Singh Chaplot demonstrated back in 2017 that a RL agent could play FPS Games using as inputs just game scores and pixels from the screen [3]. Another really interesting example is this robot [4], which is capable of moving around a house looking for an objective and avoiding obstacles using DDQL.

A good resource to understand how Reinforcement Learning really works is Deeplizard's tutorial [5]. In this tutorial they explain different versions of the algorithm and how to implement them in python to solve different OpenAI gym environments [6].

Deep Reinforcement Learning is though an union between RL and image recognition, but let's see how it actually works. The main idea is to replace the Q-table that we saw before for a Dense Neural Network that uses as input another Neural Network, a Convolutional Neural Network (CNN). The full algorithm would have as many outputs as allowed actions. Therefore, simplifying, these outputs re equivalent to the q-values saw before and so we will call them. To see it graphically, when the agent wanted to take an action, he would pass the state image through the Neural network represented in Figure 3.3 and would take the action with higher q-value.

When I said "simplifying" in the previous paragraph, I meant "simplifying a lot" in the next paragraphs I will explain all the intermediate steps in the algorithm and why they are important:

– Episodes and Steps

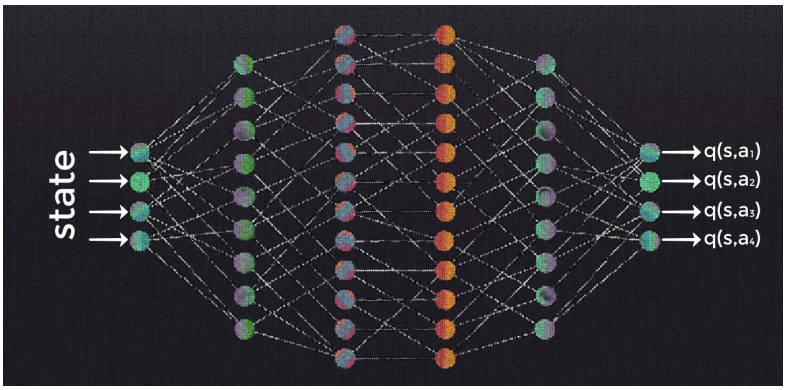

Fig. 3.3: Deep Q Learning Representation with 4 outputs

- Exploration vs Exploitation trade-off
- Replay Memory
- Bellman's Equation
- Target and Policy Networks

Episodes and Steps

RL training is divided in Episodes. One Episode is the sequence of actions needed to reach a terminal State. Each time the agent reaches a terminal state, an episode is ended, and a new one is started.

On the other hand, steps represents every time that a new action is taken, so the number of steps taken by the agent during training is infinite. Later on, we will use as metric of performance the number of steps per episode, as they must decrease during the training.

Exploration vs Exploitation trade-off

In Reinforcement Learning there are two important concepts that are **Explore** and **Exploit**. To explore is basically gather new information about the environment and to exploit is to make the best decision with the information that we already have.

In Deep Reinforcement Learning, the agent exploit the information gathered by using the pre-trained Neural Network to decide next action. On the other hand, the agent explore the environment by deciding next action randomly. We use exploration mainly in the beginning of the training because we want the agent to gather as much information of the environment as possible before starting training.

When the agent uses exploitation, it is also gathering information about the environment. However, we could not let the agent explore this way because during the exploration phase we want all the actions to be performed with the same probability and neural network bias can cause some actions to be performed much more than others.

So, how do we decide when the agent must explore or exploit? To decide it we can use multiple techniques, but the most common one is the Epsilon-Greedy Strategy. This strategy basically consist on setting a probability of exploring and keep decreasing it slowly during the training. It works this way:

1. **We set the initial exploring probability (ϵ)**
2. **We set the per-step epsilon decay, (ϵ_decay)**
3. **For each step:**
   (a) **With probability $p = \epsilon$, the agent explores the environment (takes a random action). If not, it exploit the information by deciding the action using the NN.**
   (b) **Whether the agent has explore or not, we decrease the probability of exploring the environment in the next step ($\epsilon = \epsilon - \epsilon\_decay$)**

Replay Memory

Everytime that the agent performs an action, either by exploring or exploiting, the agent lives an experience. For the purpose of training the algorithm, we will store all these experiences.

Experiences are formed by the initial state, the action taken, the state reached (final state) and the reward gotten and they are stored in the Replay Memory.

Then, every time that an action is taken, the algorithm is trained following this steps:

1. Replay Memory checks if the number of experiences is higher than the batch size
2. If there are enough experiences:
   (a) Replay Memory supplies a random set of experiences of size=batch_size.
   (b) With this set of experiences, the target network is trained.

Optimizing Replay Memory can be a challenge, because, if we are using a Graphic Card in the training, we would be storing all the experiences in its memory. But, why do we need to store all the experiences? We could also be using the last N experiences to train the network and it would be a less memory-consumption demanding solution. The answer to this question is tha

Bellman's Equation

Target and Policy Networks

3.0.3 Problems of Deep Reinforcement Learning in Real-world

Real-world problems introduces some challenges that we will have to manage. In march 2018, A. Rupam Mahmood, Dmytro Korenkevych, Brent J. Komer, and James Bergstra explained the problems they found while implementing a RL algorithm in a UR5 robotic arm [7].

Some of the problems they found were the following:

- Slow rate of data-collection, as movements in the real robot are slower than in a simulated environment.
- Partial observability. Sensors cannot retrieve all the information about the environment.
- Noisy sensors will provide inaccurate information.
- Safety of the robot and its surroundings have to be taken in mind.

- Fragility of robot components.
- Delay between an action is requested and the time it is actually performed can affect the training.
- Preparing the robot is a really difficult task:
  - Controlling the robot.
  - Define all aspects of the environment.
  - Difficulties for obtaining random and independent state when episode ends.

Another problem that can be found in our project is that, as objects are randomly placed, the environment that the agent will have to face will be completely different each time. In fact, the robot can interact with the environment, as it can move the pieces trying to pick them, so we are facing a dynamic environment RL problem. A good example of a dynamic environment problem is the path planning of a self driven car, where each time the agent takes an action the environment will change and, furthermore, obstacles do not have to be static, but they can also move.

There are multiple examples of articles on this topic, such as the one Xiaoyun Lei, Zhian Zhang, and Peifang Dong published in September 2018 using a DDQN approach to solve it [8]. However, there are other solutions as the one proposed by Marco A. Wiering [9], where he introduces some prior knowledge to the model in order to facilitate the learning. His algorithm had problems generalizing the environment, so he introduced some prior information about the model together with a Model-based RL. This made the algorithm more capable to learn without loosing a lot of trainable capability.

4. DEFINITION OF THE PROJECT

4.1 Motivation

The project motivation is the natural continuation of a previous project performed at ICAM University. This project was part of the assembly line of a car manufacturing process and its objective was to pick some specific plastic pieces and place them into the product. To achieve this, the system used opencv image processing, so it was recognising a specific shape given apriori.

This project was totally functional and the robot could perform the task with a high successful rate. However, the lack of generalization of the system makes it hard to introduce changes as using it for another part of the assembly line. Each time that this happened someone would have to introduce the shape of the pieces to the system and to calibrate the camera to the new environment.

The motivation of the project is to create from scratch a new solution for performing the picking of the pieces. This time, the project will not be sponsored by any company, so there will be less resources to use.

With this new approach, the idea is to use all the knowledge of previous documented projects on Artificial Intelligence in the industry and make a little contribution to the huge advance of industry 4.0. In fact, the idea is to make the project completely replicable so that anyone could use this project as a starting point for new applications.

4.2 Objectives

Revisar objetivos y adaptarlos al proyecto

The objectives of the project are five and are listed and explained below:

– Implement a bin picking simple solution. A basic one, without Artificial

Intelligence.

- Improve the performance using RL and Image Recognition.
- Study the usage of new technologies to add information to the system. Adding physics or new image recognition techniques.
- Test different tools such as Multiflash or 3D images to improve performance.
- Add value to the scientific community making the solution available if the previous objectives are reached.

4.3 Methodology

This project will be performed using an agile methodology, which is one of the simplest and effective processes to turn a vision for a business need into software solutions. Agile is a term used to describe software development approaches that employ continual planning, learning, improvement, team collaboration, evolutionary development, and early delivery. It encourages flexible responses to change [10].

In the case of this project, the team is just formed by two workers and a project manager. This make necessary to make some changes to the typical agile methodology. For example, daily meetings are substituted by constant communication between both workers and weekly meetings with the project director. With this approach, all the members of the project are updated about how it is going and have clear objectives.

Likewise, the methodology of this project is based in three fundamental principles as it is shown in Figure 4.1. The first two principles are highly related, as the project is iterative because it is experimental. That means that the way of working is perform little sprints with new functionalities, test them (experimental) and, depending on the results, define the new sprint, execute it and test again (iterative). Besides, the project is also incremental because the idea is starting implementing the simplest possible solution and keep adding new improvements to it in a iterative loop until the optimal configuration is reached.

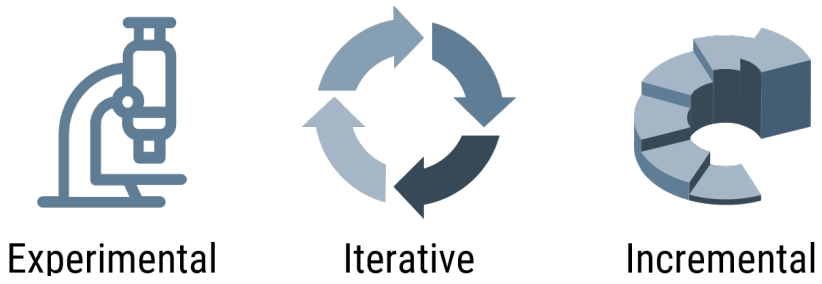

Fig. 4.1: Methodology

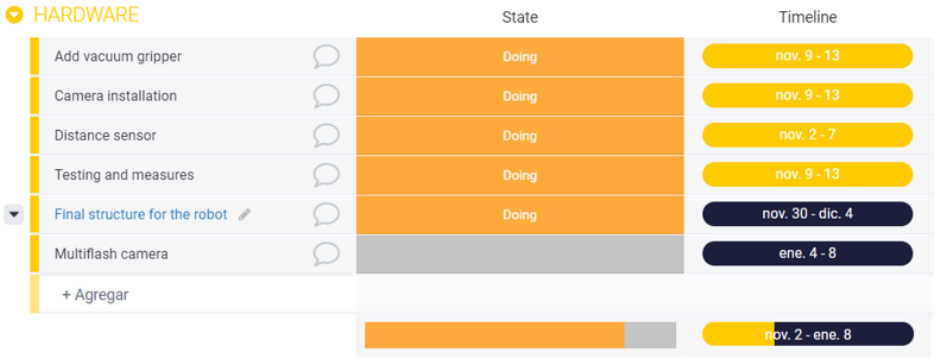

Fig. 4.2: Chronograph of the Hardward implementation

4.4 Planning and budget

Regarding the planning, there are some really important functionalities that have been defined since the beginning of the project, as they are needed. These functionalities are split in three different groups: Hardware implementation, Artificial Intelligence Implementation and Robot controller implementation. The tasks related to these three groups are shown in Figure 4.2, Figure 4.3 and Figure 4.4.

Añadir presupuesto

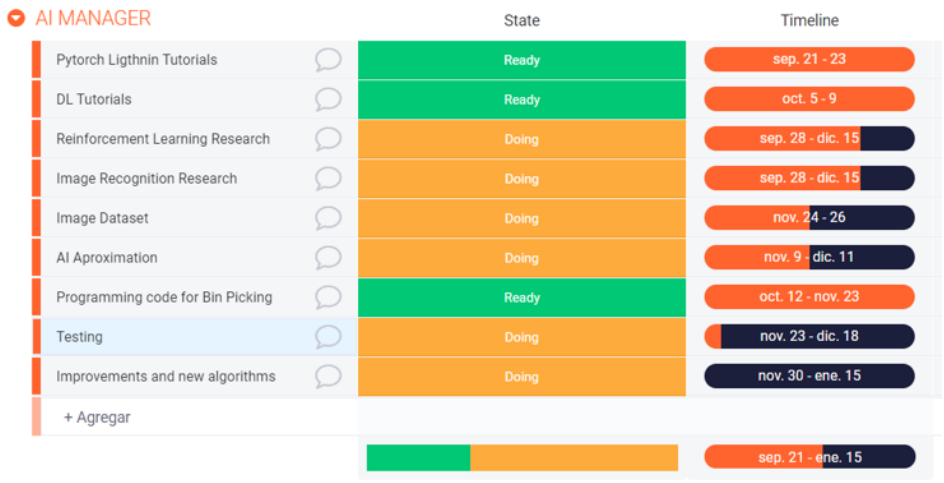

Fig. 4.3: Planning of the Artificial Intelligence implementation

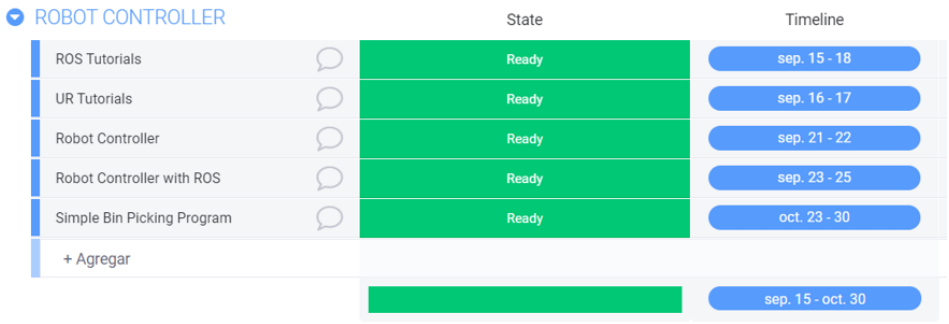

Fig. 4.4: Planning of the Robot Controller implementation

5. DEVELOPED SYSTEM

En este capítulo es donde el alumno debe describir su proyecto. En función del tipo de proyecto la estructura interna variará. El título del mismo, así como sus apartados, son sólo una sugerencia que cada alumno deberá adaptar particularmente a su proyecto.

A su vez, este capítulo podrá extenderse en varios capítulos más, por ejemplo si mi proyecto consta de varias fases o módulos, lo lógico sería tener varios capítulos donde se describa el desarrollo del proyecto:

- Capítulo 5. Implantación y configuración de la plataforma
- Capítulo 6. Desarrollo del sistema

5.1 Análisis del sistema

...

5.2 Diseño

...

5.3 Implementación

...

6. RESULTS ANALYSIS

Destacar los resultados más relevantes del proyecto y hacer un análisis crítico de los mismos. También es un capítulo obligatorio y clave.

7. CONCLUSIONS AND FUTURE WORK

Comentar las conclusiones del proyecto, destacando lo que se ha hecho, dejando claros qué objetivos se han cubierto y cuáles son las aportaciones hechas.

APPENDIX

.1 Robot Controller

En esta sección se muestra el código de algunos de los principales elementos del módulo Robot Controller, implementado en la arquitectura del proyecto.

Este código también está disponible en el siguiente repositorio de github:

https://github.com/pabloiglesia/robot_controller

.1.1 main.py

```
#!/usr/bin/env python
# coding: utf-8

"""
- We need to connect the camera and the nodes
roslaunch ur_icam_description webcam.launch

- We need to establish a connection to the robot with the
    following comand:
roslaunch ur_robot_driver ur3_bringup.launch robot_ip
    :=10.31.56.102 kinematics_config:=${HOME}/Calibration/
    ur3_calibration.yaml

- Then, we ned to activate moovit server:
roslaunch ur3_moveit_config ur3_moveit_planning_execution.launch

- Activate the talker
rosrun ai_manager main.py

- Activate the node
rosrun robot_controller arduino.py

- Finally, we can run the program
rosrun robot_controller main.py

"""

import rospy

from ai_manager.srv import GetActions
from Robot import Robot


def get_action(robot, object_gripped):
```

```
32      relative_coordinates = robot.calculate_current_coordinates()
33      rospy.wait_for_service('get_actions')
34      try:
35          get_actions = rospy.ServiceProxy('get_actions', GetActions
        )
36          return get_actions(relative_coordinates[0],
        relative_coordinates[1], object_gripped).action
37      except rospy.ServiceException as e:
38          print("Service call failed: %s"%e)
39
40
41  # This function defines the movements that robot should make
        depending on the action listened
42  def take_action(action, robot):
43      rospy.loginfo("Action received: {}".format(action))
44      object_gripped = False
45      if action == 'north':
46          robot.take_north()
47      elif action == 'south':
48          robot.take_south()
49      elif action == 'east':
50          robot.take_east()
51      elif action == 'west':
52          robot.take_west()
53      elif action == 'pick':
54          object_gripped = robot.take_pick()
55      elif action == 'random_state':
56          robot.take_random_state()
57      return object_gripped
58
59
60  if __name__ == '__main__':
61
62      rospy.init_node('robotUR')
63
64      robot = Robot()
65
66      # Test of positioning with angular coordinates
67      robot.go_to_initial_pose()
68
69      # Let's put the robot in a random position to start, creation
        of new state
70      object_gripped = take_action('random_state', robot)
71
72      while True:
73          action = get_action(robot, object_gripped)
74          object_gripped = take_action(action, robot)
```

.1.2 Robot.py

```
import copy
import rospy
import time

from std_msgs.msg import Bool
from std_msgs.msg import Float32

from ai_manager.Environment import Environment
from ur_icam_description.robotUR import RobotUR

"""
Class used to establish connection with the robot and perform different actions such as move in all cardinal directions
or pick and place an object.
"""


class Robot:
    def __init__(self, robot=RobotUR(), gripper_topic='switch_on_off'):
        self.robot = robot  # Robot we want to control
        self.gripper_topic = gripper_topic  # Gripper topic
        self.gripper_publisher = rospy.Publisher(self.gripper_topic, Bool)  # Publisher for the gripper topic

    def relative_move(self, x, y, z):
        """
        Perform a relative move in all x, y or z coordinates.

        :param x:
        :param y:
        :param z:
        :return:
        """
        waypoints = []
        wpose = self.robot.get_current_pose().pose
        if x:
            wpose.position.x -= x  # First move up (x)
            waypoints.append(copy.deepcopy(wpose))
        if y:
            wpose.position.y -= y  # Second move forward/backwards in (y)
            waypoints.append(copy.deepcopy(wpose))
        if z:
            wpose.position.z += z  # Third move sideways (z)
            waypoints.append(copy.deepcopy(wpose))

```

```
44          self.robot.exec_cartesian_path(waypoints)
45
46      def calculate_relative_movement(self, relative_coordinates):
47          absolute_coordinates_x = Environment.CARTESIAN_CENTER[0] -
        relative_coordinates[0]
48          absolute_coordinates_y = Environment.CARTESIAN_CENTER[1] -
        relative_coordinates[1]
49
50          current_pose = self.robot.get_current_pose()
51
52          x_movement = current_pose.pose.position.x -
    absolute_coordinates_x
53          y_movement = current_pose.pose.position.y -
    absolute_coordinates_y
54
55          return x_movement, y_movement
56
57      def calculate_current_coordinates(self):
58          absolut_coordinate_x = self.robot.get_current_pose().pose.
    position.x
59          absolut_coordinate_y = self.robot.get_current_pose().pose.
    position.y
60
61          relative_coordinate_x = Environment.CARTESIAN_CENTER[0] -
    absolut_coordinate_x
62          relative_coordinate_y = Environment.CARTESIAN_CENTER[1] -
    absolut_coordinate_y
63
64          return [relative_coordinate_x, relative_coordinate_y]
65
66      # Action north: positive x
67      def take_north(self, distance=Environment.ACTION_DISTANCE):
68          self.relative_move(distance, 0, 0)
69
70      # Action south: negative x
71      def take_south(self, distance=Environment.ACTION_DISTANCE):
72          self.relative_move(-distance, 0, 0)
73
74      # Action east: negative y
75      def take_east(self, distance=Environment.ACTION_DISTANCE):
76          self.relative_move(0, -distance, 0)
77
78      # Action west: positive y
79      def take_west(self, distance=Environment.ACTION_DISTANCE):
80          self.relative_move(0, distance, 0)
81
82      def take_random_state(self):
83          # Move robot to random positions using relative moves. Get
        coordinates
```

```
84          relative_coordinates = Environment.generate_random_state()
85          # Calculate the new coordinates
86          x_movement, y_movement = self.calculate_relative_movement(
    relative_coordinates)
87          # Move the robot to the random state
88          self.relative_move(x_movement, y_movement, 0)
89
90      def send_gripper_message(self, msg, timer=2, n_msg=10):
91          """
92          Function that sends a burst of n messages of the
    gripper_topic during an indicated time
93          :param msg: True or False
94          :param time: time in seconds
95          :param n_msg: number of messages
96          :return:
97          """
98          time_step = (timer/2)/n_msg
99          i=0
100         while(i <= n_msg):
101             self.gripper_publisher.publish(msg)
102             time.sleep(time_step)
103             i += 1
104
105         time.sleep(timer/2)
106
107     # Action pick: Pick and place
108     def take_pick(self):
109         # In this function we should read the distance to the
    object
110         # up_distance = 0  # Variable were we store the distance
    that we have move the robot so that we can go back to the
111         # original pose
112
113         def change_plan_speed(plan, new_speed):
114             """
115             Function used for changing Robot velocity of a
    cartesian path once the movement have been planned.
116             :param plan: RobotTrajectory object. For example, the
    one calculated by compute_cartesian_path() MoveGroup function.
117             :param new_speed: speed factor of the robot, been 1
    the original speed and 0 the minimum.
118             :return: RobotTrajectory object (new plan).
119             """
120             new_plan = plan
121             n_joints = len(plan.joint_trajectory.joint_names)
122             n_points = len(plan.joint_trajectory.points)
123
124             points = []
125             for i in range(n_points):
```

```
126                 plan.joint_trajectory.points[i].time_from_start =
    plan.joint_trajectory.points[
127
        i].time_from_start / new_speed
128                 velocities = []
129                 accelerations = []
130                 positions = []
131                 for j in range(n_joints):
132                     velocities.append(plan.joint_trajectory.points
    [i].velocities[j] * new_speed)
133                     accelerations.append(plan.joint_trajectory.
    points[i].accelerations[j] * new_speed)
134                     positions.append(plan.joint_trajectory.points[
    i].positions[j])
135
136                 point = plan.joint_trajectory.points[i]
137                 point.velocities = velocities
138                 point.accelerations = accelerations
139                 point.positions = positions
140
141                 points.append(point)
142
143             new_plan.joint_trajectory.points = points
144
145             return new_plan
146
147         def back_to_original_pose(robot):
148             """
149             Function used to go back to the original height once a
     vertical movement has been performed.
150             :param robot: robot_controller.Robot.py object
151             :return:
152             """
153             distance = Environment.CARTESIAN_CENTER[2] - robot.
    robot.get_current_pose().pose.position.z
154             robot.relative_move(0, 0, distance)
155
156         def down_movement(robot, movement_speed):
157             """
158             This function performs the down movement of the pick
    action.
159
160             It creates an asynchronous move group trajectory
    planning. This way the function is able to receive distance
161             messages while the robot is moving and stop it once
    the robot is in contact with an object.
162
163             Finally, when there is any problems with the
    communications the movement is stopped and
```

```
164             communication_problem boolean flag is set to True. It
    is considered that there is a problem with
165             communications when the robot is not receiving any
    distance messages during 200 milli-seconds (timeout=0.2)
166 
167             :param robot: robot_controller.Robot.py object
168             :return: communication_problem flag
169             """
170 
171             distance_ok = rospy.wait_for_message('distance', Bool)
    .data  # We retrieve sensor distance
172             communication_problem = False
173 
174             if not distance_ok:  # If the robot is already in
    contact with an object, no movement is performed
175                 waypoints = []
176                 wpose = robot.robot.get_current_pose().pose
177                 wpose.position.z -= (wpose.position.z)  # Third
    move sideways (z)
178                 waypoints.append(copy.deepcopy(wpose))
179 
180                 (plan, fraction) = robot.robot.move_group.
    compute_cartesian_path(
181                     waypoints,  # waypoints to follow
182                     0.01,  # eef_step
183                     0.0)  # jump_threshold
184 
185                 plan = change_plan_speed(plan, movement_speed)
186                 robot.robot.move_group.execute(plan, wait=False)
187 
188                 while not distance_ok:
189                     try:
190                         distance_ok = rospy.wait_for_message('
    distance', Bool, 0.2).data  # We retrieve sensor distance
191                     except:
192                         communication_problem = True
193                         rospy.loginfo("Error in communications,
    trying again")
194                         break
195 
196                 # Both stop and 10 mm up movement to stop the
    robot
197                 robot.robot.move_group.stop()
198                 robot.relative_move(0, 0, 0.001)
199 
200             return communication_problem
201 
202         communication_problem = True
203         while communication_problem:  # Infinite loop until the
```

```
movement is completed
204             communication_problem = down_movement(self,
movement_speed=0.2)
205
206         self.send_gripper_message(True, timer=4)  # We turn on the
 gripper
207
208         back_to_original_pose(self)  # Back to the original pose
209
210         object_gripped = rospy.wait_for_message('object_gripped',
Bool).data
211         if object_gripped:  # If we have gripped an object we
place it into the desired point
212             self.take_place()
213         else:
214             self.send_gripper_message(False)  # We turn off the
gripper
215
216         return object_gripped
217
218     # Function to define the place for placing the grasped objects
219     def take_place(self):
220         # First, we get the cartesian coordinates of one of the
corner
221         x_box, y_box = Environment.get_relative_corner('se')
222         x_move, y_move = self.calculate_relative_movement([x_box,
y_box])
223         # We move the robot to the corner of the box
224         self.relative_move(x_move, y_move, 0)
225         # We calculate the trajectory for our robot to reach the
box
226         trajectory_x = self.robot.get_current_pose().pose.position
.x - Environment.PLACE_CARTESIAN_CENTER[0]
227         trajectory_y = self.robot.get_current_pose().pose.position
.y - Environment.PLACE_CARTESIAN_CENTER[1]
228         trajectory_z = - Environment.CARTESIAN_CENTER[2] +
Environment.PLACE_CARTESIAN_CENTER[2]
229         # We move the robot to the coordinates desired to place
the object
230         self.relative_move(0, 0, trajectory_z)
231         self.relative_move(0, trajectory_y, 0)
232         self.relative_move(trajectory_x, 0, 0)
233         # Then, we left the object
234         self.relative_move(0, 0, -0.05)
235         # Then, we switch off the vacuum gripper so the object can
 be placed
236         self.send_gripper_message(False)
237         # Wait some seconds, in order to the msg to arrive to the
gripper
```

```
238         time.sleep(2)
239         # Then the robot goes up
240         self.relative_move(0, 0, 0.05)
241         # Final we put the robot in the center of the box, the
    episode should finish now
242         self.robot.go_to_joint_state(Environment.ANGULAR_CENTER)
243 
244     def go_to_initial_pose(self):
245         target_reached = self.robot.go_to_joint_state(Environment.
    ANGULAR_CENTER)
246 
247         if target_reached:
248             print("Target reachead")
249         else:
250             print("Target not reached")
```

.2 Artificial Intelligence Manager

En esta sección se muestra el código de algunos de los principales elementos del módulo AI Manager, implementado en la arquitectura del proyecto.

Este código también está disponible en el siguiente repositorio de github:

https://github.com/pabloiglesia/ai_manager

.2.1 main.py

```
1 #!/usr/bin/env python
2 """
3 Code used to train the UR3 robot to perform a pick and place task
    using Reinforcement Learning and Image Recognition.
4 This code does not perform actions directly into the robot, it
    just posts actions in a ROS topic and
5 gathers state information from another ROS topic.
6 """
7 
8 # import rospy
9 # import torch
10 
11 # from ai_manager.srv import GetActions, GetActionsResponse
12 from RLAlgorithm import RLAlgorithm
13 from Environment import Environment
14 
15 # rospy.init_node('ai_manager', anonymous=True)  # ROS node
    initialization
```

```
16 # Global Image Controller
17 RL_ALGORITHM = RLAlgorithm.recover_training(batch_size=256, lr
       =0.0001,
18                                            others='
       optimal_original_rewards')
19                                            # others = '
       optimal_original_rewards_new_model')
20
21 def handle_get_actions(req):
22     """
23     Callback for each Request from the Robot
24     :param req: Robot requests has 3 elements: object_gripped, x
       and y elements
25     :return:
26     """
27     object_gripped = req.object_gripped
28     current_coordinates = [req.x, req.y]
29     # Next action is calculated from the current state
30     action = RL_ALGORITHM.next_training_step(current_coordinates,
       object_gripped)
31
32     # RL_ALGORITHM.plot()
33
34     return GetActionsResponse(action)
35
36
37 def get_actions_server():
38     """
39     Service initialization to receive requests of actions from the
       robot.
40     Each time that a request is received, handle_get_actions
       function will be called
41     :return:
42     """
43     s = rospy.Service('get_actions', GetActions,
       handle_get_actions)
44     rospy.loginfo("Ready to send actions.")
45     rospy.spin()
46     rospy.on_shutdown(save_training)
47
48
49 def save_training():
50     RL_ALGORITHM.save_training()
51
52
53 if __name__ == '__main__':
54     save_training()
```

.2.2 RLAlgorithm.py

```
# # coding=utf-8
import math
import random
import os
import errno
import sys
from collections import namedtuple

import matplotlib
import matplotlib.pyplot as plt
# import rospy
# import torch
# import torch.nn as nn
# import torch.nn.functional as F
# import torch.optim as optim
# import torchvision.transforms as T
# from PIL import Image

from Environment import Environment
from TrainingStatistics import TrainingStatistics
# from ImageProcessing.ImageModel import ImageModel
# from ImageController import ImageController

is_ipython = 'inline' in matplotlib.get_backend()
if is_ipython: from IPython import display

import pickle

State = namedtuple(  # State information namedtuple
    'State',
    ('coordinate_x', 'coordinate_y', 'pick_probability', 'object_gripped', 'image_raw')
)

Experience = namedtuple(  # Replay Memory Experience namedtuple
    'Experience',
    ('state', 'coordinates', 'pick_probability', 'action', 'next_state', 'next_coordinates', 'next_pick_probability',
     'reward', 'is_final_state')
)


class RLAlgorithm:
    """
    Class used to perform actions related to the RL Algorithm training. It can be initialized with custom parameters or
    with the default ones.
```

```
45
46      To perform a Deep Reinforcement Learning training, the
    following steps have to be followed:
47
48          1. Initialize replay memory capacity.
49          2. Initialize the policy network with random weights.
50          3. Clone the policy network, and call it the target
    network.
51          4. For each episode:
52             1. Initialize the starting state.
53              2. For each time step:
54                  1. Select an action.
55                      - Via exploration or exploitation
56                  2. Execute selected action in an emulator or in
    Real-life.
57                  3. Observe reward and next state.
58                  4. Store experience in replay memory.
59                  5. Sample random batch from replay memory.
60                  6. Preprocess states from batch.
61                  7. Pass batch of preprocessed states to policy
    network.
62                  8. Calculate loss between output Q-values and
    target Q-values.
63                      - Requires a pass to the target network for
    the next state
64                  9. Gradient descent updates weights in the policy
    network to minimize loss.
65                      - After time steps, weights in the target
    network are updated to the weights in the policy network.
66
67      """
68
69      def __init__(self, object_gripped_reward=10,
    object_not_picked_reward=-10, out_of_limits_reward=-10,
70                   horizontal_movement_reward=-1, batch_size=32,
    gamma=0.999, eps_start=1, eps_end=0.01, eps_decay=0.0005,
71                   target_update=10, memory_size=100000, lr=0.001,
    num_episodes=1000, include_pick_prediction=False,
72                   save_training_others='optimal'):
73          """
74
75          :param object_gripped_reward: Object gripped reward
76          :param object_not_picked_reward: Object not picked reward
77          :param out_of_limits_reward: Out of limits reward
78          :param horizontal_movement_reward: Horizontal movement
    reward
79          :param batch_size: Size of the batch used to train the
    network in every step
80          :param gamma: discount factor used in the Bellman equation
```

```
81          :param eps_start: Greedy strategy epsilon start (Probability of random choice)
82          :param eps_end: Greedy strategy minimum epsilon (Probability of random choice)
83          :param eps_decay: Greedy strategy epsilon decay (Probability decay of random choice)
84          :param target_update: How frequently, in terms of episodes, target network will update the weights with the
85          policy network weights
86          :param memory_size: Capacity of the replay memory
87          :param lr: Learning rate of the Deep Learning algorithm
88          :param num_episodes:  Number of episodes on training
89          :param include_pick_prediction: Use the image model pick prediction as input of the DQN
90          :param self_training_others: Parameter used to modify the filename of the training while saving
91          """
92
93          self.batch_size = batch_size
94          self.gamma = gamma
95          self.eps_start = eps_start
96          self.eps_end = eps_end
97          self.eps_decay = eps_decay
98          self.target_update = target_update
99          self.memory_size = memory_size
100         self.lr = lr
101         self.num_episodes = num_episodes
102         self.include_pick_prediction = include_pick_prediction
103         self.save_training_others = save_training_others
104
105         self.current_state = None  # Robot current state
106         self.previous_state = None  # Robot previous state
107         self.current_action = None  # Robot current action
108         self.current_action_idx = None  # Robot current action Index
109         self.episode_done = False  # True if the episode has just ended
110
111         # This tells PyTorch to use a GPU if its available, otherwise use the CPU
112         self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # Torch devide
113         self.em = self.EnvManager(self, object_gripped_reward, object_not_picked_reward, out_of_limits_reward,
114                                   horizontal_movement_reward)  # Robot Environment Manager
115         self.strategy = self.EpsilonGreedyStrategy(self.eps_start, self.eps_end, self.eps_decay)  # Greede Strategy
116         self.agent = self.Agent(self)  # RL Agent
```

```
117         self.memory = self.ReplayMemory(self.memory_size)  # Replay Memory
118         self.statistics = TrainingStatistics()  # Training statistics
119 
120         self.policy_net = self.DQN(self.em.image_tensor_size,
121                                    self.em.num_actions_available(),
122                                    self.include_pick_prediction).to(self.device)  # Policy Q Network
123         self.target_net = self.DQN(self.em.image_tensor_size,
124                                    self.em.num_actions_available(),
125                                    self.include_pick_prediction).to(self.device)  # Target Q Network
126         self.target_net.load_state_dict(self.policy_net.state_dict())  # Target net has to be the same as policy network
127         self.target_net.eval()  # Target net has to be the same as policy network
128         self.optimizer = optim.Adam(params=self.policy_net.parameters(), lr=self.lr)  # Q Networks optimizer
129 
130         print("Device: ", self.device)
131 
132     class Agent:
133         """
134         Class that contains all needed methods to control the agent through the environment and retrieve information of
135         Its state
136         """
137 
138         def __init__(self, rl_algorithm):
139             """
140 
141             :param self: RLAlgorithm object
142             """
143             self.strategy = rl_algorithm.strategy  # Greedy Strategy
144             self.num_actions = rl_algorithm.em.num_actions_available()  # Num of actions available
145             self.device = rl_algorithm.device  # Torch device
146             self.rl_algorithm = rl_algorithm
147 
148         def select_action(self, state, policy_net):
149             """
150             Method used to pick the following action of the robot
151             Method used to pick the following action of the robot
152             :param state: State RLAlgorithm namedtuple with all the information of the current state
```

```
153             :param policy_net: DQN object used as policy network
    for the RL algorithm
154             :return:
155             """
156             random_action = False
157             if self.rl_algorithm.episode_done:  # If the episode
    has just ended we reset the robot environment
158                 self.rl_algorithm.episode_done = False  # Put the
    variable episode_done back to False
159                 self.rl_algorithm.statistics.new_episode()
160 
161                 self.rl_algorithm.current_action = 'random_state'
     # Return random_state to reset the robot position
162                 self.rl_algorithm.current_action_idx = None
163             else:
164                 rate = self.strategy.get_exploration_rate(
165                     self.rl_algorithm.statistics.current_step)  #
    We get the current epsilon value
166 
167                 if rate > random.random():  # With a probability =
     rate we choose a random action (Explore environment)
168                     action = random.randrange(self.num_actions)
169                     random_action = True
170                 else:  # With a probability = (1 - rate) we
    Explote the information we already have
171                     try:
172                         with torch.no_grad():  # We calculate the
    action using the Policy Q Network
173                             action = policy_net(state.image_raw,
    torch.tensor(
174                                 [[state.coordinate_x, state.
    coordinate_y]], device=self.device),
175                                                 state.
    pick_probability).argmax(dim=1).to(
176                                 self.device)  # exploit
177                     except:
178                         print("Ha habido un error")
179 
180                 self.rl_algorithm.current_action = self.
    rl_algorithm.em.actions[action]
181                 self.rl_algorithm.current_action_idx = action
182 
183             return self.rl_algorithm.current_action, random_action
      # We return the action as a string, not as int
184 
185     # class DQN(nn.Module):
186     #     """
187     #     Class to create a Deep Q Learning Neural Network
188     #     """
```

```
189     #
190     #     def __init__(self, image_tensor_size, num_actions,
        include_pick_prediction):
191     #         """
192     #
193     #         :param image_tensor_size: Size of the input tensor
194     #         :param num_actions: Number of actions, which is the
        output of the Neural Network
195     #         """
196     #         super(RLAlgorithm.DQN, self).__init__()
197     #
198     #         self.linear1 = nn.Linear(image_tensor_size, int(
        image_tensor_size / 2))
199     #         self.linear2 = nn.Linear(int(image_tensor_size / 2),
         int(image_tensor_size / 4))
200     #         extra_features = 2  # coordinates
201     #         if include_pick_prediction:
202     #             extra_features = 3  # pick prediction
203     #         self.linear3 = nn.Linear(int(image_tensor_size / 4)
        + extra_features, num_actions)
204     #         self.linear = nn.Linear(image_tensor_size + 2,
        num_actions)
205     #
206     #     # Called with either one element to determine next
        action, or a batch
207     #     # during optimization. Returns tensor([[left0exp,
        right0exp]...]).
208     #     def forward(self, image_raw, coordinates,
        pick_probability):
209     #
210     #         output = self.linear1(image_raw)
211     #         output = self.linear2(output)
212     #         if pick_probability:
213     #             output = torch.cat((output, coordinates,
        pick_probability), 1)
214     #         else:
215     #             output = torch.cat((output, coordinates), 1)
216     #         return self.linear3(output)
217
218     class EnvManager:
219         """
220         Class used to manage the RL environment. It is used to
        perform actions such as calculate rewards or retrieve the
221         current state of the robot.
222         """
223
224         def __init__(self, rl_algorithm, object_gripped_reward,
        object_not_picked_reward, out_of_limits_reward,
225                      horizontal_movement_reward):
```

```
226             """
227             Initialization of an object
228             :param rl_manager: RLAlgorithm object
229             :param object_gripped_reward: Object gripped reward
230             :param object_not_picked_reward: Object not picked
    reward
231             :param out_of_limits_reward: Out of limits reward
232             :param horizontal_movement_reward: Horizontal movement
     reward
233             """
234             self.object_gripped_reward = object_gripped_reward
235             self.out_of_limits_reward = out_of_limits_reward
236             self.object_not_picked_reward =
    object_not_picked_reward
237             self.horizontal_movement_reward =
    horizontal_movement_reward
238 
239             self.device = rl_algorithm.device  # Torch device
240             self.image_controller = ImageController()  #
    ImageController object to manage images
241             self.actions = ['north', 'south', 'east', 'west', '
    pick']  # Possible actions of the objects
242             self.image_height = None  # Retrieved images height
243             self.image_width = None  # Retrieved Images Width
244             self.image = None  # Current image ROS message
245             self.image_tensor = None  # Current image tensor
246             self.pick_probability = None  # Current image tensor
247 
248             self.model_name = 'model-epoch=05-val_loss=0.36-
    weights7y3_unfreeze2.ckpt'
249             # self.model_name = 'resnet50_freezed.ckpt'
250             self.model_family = 'resnet50'
251             self.image_model = ImageModel(model_name=self.
    model_family)
252             self.feature_extraction_model = self.image_model.
    load_model(self.model_name)
253             self.image_tensor_size = self.image_model.
    get_size_features(
254                 self.feature_extraction_model)  # Size of the
    image after performing some transformations
255 
256             self.rl_algorithm = rl_algorithm
257             self.gather_image_state()  # Retrieve initial state
    image
258 
259         def calculate_reward(self, previous_image):
260             """
261             Method used to calculate the reward of the previous
    action and whether it is a final state or not
```

```
262             :return: reward, is_final_state
263             """
264             current_coordinates = [self.rl_algorithm.current_state
    .coordinate_x,
265                                    self.rl_algorithm.current_state
    .coordinate_y]  # Retrieve robot's current coordinates
266             object_gripped = self.rl_algorithm.current_state.
    object_gripped  # Retrieve if the robot has an object gripped
267             if Environment.is_terminal_state(current_coordinates,
    object_gripped):  # If is a terminal state
268                 self.rl_algorithm.episode_done = True  # Set the
    episode_done variable to True to end up the episode
269                 episode_done = True
270                 if object_gripped:  # If object_gripped is True,
    the episode has ended successfully
271                     reward = self.object_gripped_reward
272                     self.rl_algorithm.statistics.
    add_succesful_episode(True)  # Saving episode successful
    statistic
273                     self.rl_algorithm.statistics.increment_picks()
      # Increase of the statistics cpunter
274                     rospy.loginfo("Episode ended: Object gripped!"
    )
275                     self.image_controller.record_image(
    previous_image, True)  # Saving the falure state image
276                 else:  # Otherwise the robot has reached the
    limits of the environment
277                     reward = self.out_of_limits_reward
278                     self.rl_algorithm.statistics.
    add_succesful_episode(False)  # Saving episode failure
    statistic
279                     rospy.loginfo("Episode ended: Environment
    limits reached!")
280             else:  # If it is not a Terminal State
281                 episode_done = False
282                 if self.rl_algorithm.current_action == 'pick':  #
    if it is not the first action and action is pick
283                     reward = self.object_not_picked_reward
284                     self.image_controller.record_image(
    previous_image, False)  # Saving the falure state image
285                     self.rl_algorithm.statistics.increment_picks()
      # Increase of the statistics counter
286                 else:  # otherwise
287                     self.rl_algorithm.statistics.
    fill_coordinates_matrix(current_coordinates)
288                     reward = self.horizontal_movement_reward
289
290             self.rl_algorithm.statistics.add_reward(reward)  # Add
     reward to the algorithm statistics
```

```
291             return reward, episode_done
292 
293         def gather_image_state(self):
294             """
295             This method gather the relative state of the robot by retrieving an image using the image_controller class,
296             which reads the image from the ROS topic specified.
297             """
298             previous_image = self.image
299             self.image, self.image_width, self.image_height = self.image_controller.get_image()  # We retrieve state image
300             self.image_tensor, pick_probability = self.extract_image_features(self.image)
301             if self.rl_algorithm.include_pick_prediction:
302                 self.pick_probability = pick_probability
303             return previous_image
304 
305         def extract_image_features(self, image):
306             """
307             Method used to transform the image to extract image features by passing it through the image_model CNN
308             network
309             :param image_raw: Image
310             :return:
311             """
312             features, pick_prediction = self.image_model.evaluate_image(image, self.feature_extraction_model)
313             features = torch.from_numpy(features)
314             return features.to(self.device), pick_prediction[1].to(self.device)
315 
316         def num_actions_available(self):
317             """
318             Returns the number of actions available
319             :return: Number of actions available
320             """
321             return len(self.actions)
322 
323     class EpsilonGreedyStrategy:
324         """
325         Class used to perform the Epsilon greede strategy
326         """
327 
328         def __init__(self, start, end, decay):
329             """
330             Initialization
331             :param start: Greedy strategy epsilon start (Probability of random choice)
332             :param end: Greedy strategy minimum epsilon (
```

```
    Probability of random choice)
333             :param decay: Greedy strategy epsilon decay (
    Probability decay of random choice)
334             """
335             self.start = start
336             self.end = end
337             self.decay = decay
338 
339         def get_exploration_rate(self, current_step):
340             """
341             It calculates the rate depending on the actual step of
     the execution
342             :param current_step: step of the training
343             :return:
344             """
345             return self.end + (self.start - self.end) * \
346                    math.exp(-1. * current_step * self.decay)
347 
348     class QValues:
349         """
350         It returns the predicted q-values from the policy_net for
    the specific state-action pairs that were passed in.
351         states and actions are the state-action pairs that were
    sampled from replay memory.
352         """
353 
354         @staticmethod
355         def get_current(policy_net, states, coordinates, actions,
    pick_probabilities):
356             """
357             With the current state of the policy network, it
    calculates the q_values of
358             :param policy_net: policy network used to decide the
    actions
359             :param states: Set of state images (Preprocessed)
360             :param coordinates: Set of robot coordinates
361             :param actions: Set of taken actions
362             :return:
363             """
364             return policy_net(states, coordinates,
    pick_probabilities).gather(dim=1, index=actions.unsqueeze(-1))
365 
366         @staticmethod
367         def get_next(target_net, next_states, next_coordinates,
    next_pick_probabilities, is_final_state):
368             """
369             Calculate the maximum q-value predicted by the
    target_net among all possible next actions.
370             If the action has led to a terminal state, next reward
```

```
 will be 0. If not, it is calculated using the target
371             net
372             :param target_net: Target Deep Q Network
373             :param next_states: Next states images
374             :param next_coordinates: Next states coordinates
375             :param is_final_state: Tensor indicating whether this
    action has led to a final state or not.
376             :return:
377             """
378             batch_size = next_states.shape[0]  # The batch size is
     taken from next_states shape
379             # q_values is initialized with a zeros tensor of
    batch_size and if there is GPU it is loaded to it
380             q_values = torch.zeros(batch_size).to(torch.device("
    cuda" if torch.cuda.is_available() else "cpu"))
381             non_final_state_locations = (is_final_state == False)
     # Non final state index locations are calculated
382             non_final_states = next_states[
    non_final_state_locations]  # non final state images
383             non_final_coordinates = next_coordinates[
    non_final_state_locations]  # non final coordinates
384             non_final_pick_probabilities = next_pick_probabilities
    [
385                 non_final_state_locations]  # non final pick
    probabilities
386             # Max q values of the non final states are calculated
    using the target net
387             q_values[non_final_state_locations] = \
388                 target_net(non_final_states, non_final_coordinates
    , non_final_pick_probabilities).max(dim=1)[
389                     0].detach()
390             return q_values
391
392     class ReplayMemory:
393         """
394         Class used to create a Replay Memory for the RL algorithm
395         """
396
397         def __init__(self, capacity):
398             """
399             Initialization of ReplayMemory
400             :param capacity: Capacity of Replay Memory
401             """
402             self.capacity = capacity
403             self.memory = []  # Actual memory. it will be filled
    with Experience namedtuples
404             self.push_count = 0  # will be used to keep track of
    how many experiences have been added to the memory
405
```

```
406         def push(self, experience):
407             """
408             Method used to fill the Replay Memory with experiences
409             :param experience: Experience namedtuple
410             :return:
411             """
412             if len(self.memory) < self.capacity:  # if memory is
    not full, new experience is appended
413                 self.memory.append(experience)
414             else:  # If its full, we add a new experience and take
     the oldest out
415                 self.memory[self.push_count % self.capacity] =
    experience
416             self.push_count += 1  # we increase the memory counter
417 
418         def sample(self, batch_size):
419             """
420             Returns a random sample of experiences
421             :param batch_size: Number of randomly sampled
    experiences returned
422             :return: random sample of experiences (Experience
    namedtuples)
423             """
424             return random.sample(self.memory, batch_size)
425 
426         def can_provide_sample(self, batch_size):
427             """
428             returns a boolean telling whether or not we can sample
     from memory. Recall that the size of a sample
429             we'll obtain from memory will be equal to the batch
    size we use to train our network.
430             :param batch_size: Batch size to train the network
431             :return: boolean telling whether or not we can sample
    from memory
432             """
433             return len(self.memory) >= batch_size
434 
435     def extract_tensors(self, experiences):
436         """
437         Converts a batch of Experiences to Experience of batches
    and returns all the elements separately.
438         :param experiences: Batch of Experienc objects
439         :return: A tuple of each element of a Experience
    namedtuple
440         """
441         batch = Experience(*zip(*experiences))
442 
443         states = torch.cat(batch.state)
444         actions = torch.cat(batch.action)
```

```
445         rewards = torch.cat(batch.reward)
446         next_states = torch.cat(batch.next_state)
447         coordinates = torch.cat(batch.coordinates)
448         next_coordinates = torch.cat(batch.next_coordinates)
449         pick_probabilities = torch.cat(batch.pick_probability)
450         next_pick_probabilities = torch.cat(batch.
    next_pick_probability)
451         is_final_state = torch.cat(batch.is_final_state)
452 
453         return states, coordinates, pick_probabilities, actions,
    rewards, next_states, next_coordinates, \
454                next_pick_probabilities, is_final_state
455 
456     @staticmethod
457     def saving_name(batch_size, gamma, eps_start, eps_end,
    eps_decay, lr, others=''):
458         return 'bs{}_g{}_es{}_ee{}_ed{}_lr_{}_{}.pkl'.format(
459             batch_size, gamma, eps_start, eps_end, eps_decay, lr,
    others
460         )
461 
462     def save_training(self, dir='trainings/', others='optimal'):
463 
464         filename = self.saving_name(self.batch_size, self.gamma,
    self.eps_start, self.eps_end, self.eps_decay, self.lr,
465                                     self.save_training_others)
466 
467         def create_if_not_exist(filename, dir):
468             current_path = os.path.dirname(os.path.realpath(
    __file__))
469             filename = os.path.join(current_path, dir, filename)
470             if not os.path.exists(os.path.dirname(filename)):
471                 try:
472                     os.makedirs(os.path.dirname(filename))
473                 except OSError as exc:  # Guard against race
    condition
474                     if exc.errno != errno.EEXIST:
475                         raise
476             return filename
477 
478         rospy.loginfo("Saving training...")
479 
480         abs_filename = create_if_not_exist(filename, dir)
481 
482         self.em.image_model = None
483         self.em.feature_extraction_model = None
484 
485         with open(abs_filename, 'wb+') as output:  # Overwrites
    any existing file.
```

```
486             pickle.dump(self, output, pickle.HIGHEST_PROTOCOL)
487
488         rospy.loginfo("Saving Statistics...")
489         print(filename)
490
491         filename = 'trainings/{}_stats.pkl'.format(filename.split(
    '.pkl')[0])
492         self.statistics.save(filename=filename)
493
494         rospy.loginfo("Training saved!")
495
496     @staticmethod
497     def recover_training(batch_size=32, gamma=0.999, eps_start=1,
    eps_end=0.01,
498                          eps_decay=0.0005, lr=0.001, others='
    optimal', dir='trainings/', ):
499         current_path = os.path.dirname(os.path.realpath(__file__))
500         filename = RLAlgorithm.saving_name(batch_size, gamma,
    eps_start, eps_end, eps_decay, lr, others)
501         filename = os.path.join(current_path, dir, filename)
502         # try:
503         with open(filename, 'rb') as input:
504             rl_algorithm = pickle.load(input)
505                 # rospy.loginfo("Training recovered. Next step
    will be step number {}"
506                 #               .format(rl_algorithm.statistics.
    current_step))
507                 #
508                 # rl_algorithm.em.image_model = ImageModel(
    model_name=rl_algorithm.em.model_family)
509                 # rl_algorithm.em.feature_extraction_model =
    rl_algorithm.em.image_model.load_model(
510                 #     rl_algorithm.em.model_name)
511                 #
512             return rl_algorithm
513         # except IOError:
514         #     rospy.loginfo("There is no Training saved. New
    object has been created")
515         #     return RLAlgorithm(batch_size=batch_size, gamma=
    gamma, eps_start=eps_start, eps_end=eps_end,
516         #                        eps_decay=eps_decay, lr=lr,
    save_training_others=others)
517
518     def train_net(self):
519         """
520         Method used to train both the train and target Deep Q
    Networks. We train the network minimizing the loss between
521         the current Q-values of the action-state tuples and the
    target Q-values. Target Q-values are calculated using
```

```
522         thew Bellman's equation:
523 
524         q*(state, action) = Reward + gamma * max( q*(next_state,
    next_action) )
525         :return:
526         """
527         # If there are at least as much experiences stored as the
    batch size
528         if self.memory.can_provide_sample(self.batch_size):
529             experiences = self.memory.sample(self.batch_size)  #
    Retrieve the experiences
530             # We split the batch of experience into different
    tensors
531             states, coordinates, pick_probabilities, actions,
    rewards, next_states, next_coordinates, \
532                 next_pick_probabilities, is_final_state = self.
    extract_tensors(experiences)
533             # To compute the loss, current_q_values and
    target_q_values have to be calculated
534             current_q_values = self.QValues.get_current(self.
    policy_net, states, coordinates, actions,
535 
    pick_probabilities)
536             # next_q_values is the maximum Q-value of each future
    state
537             next_q_values = self.QValues.get_next(self.target_net,
     next_states, next_coordinates,
538 
    next_pick_probabilities, is_final_state)
539             target_q_values = (next_q_values * self.gamma) +
    rewards
540 
541             loss = F.mse_loss(current_q_values, target_q_values.
    unsqueeze(1))  # Loss is calculated
542             self.optimizer.zero_grad()  # set all the gradients to
     0 (initialization) so that we don't accumulate
543             # gradient throughout all the backpropagation
544             loss.backward(
545                 retain_graph=True)  # Compute the gradient of the
    loss with respect to all the weights and biases in the
546             # policy net
547             self.optimizer.step()  # Updates the weights and
    biases with the gradients computed
548 
549         if self.statistics.episode % self.target_update == 0:  #
    If target_net has to be updated in this episode
550             self.target_net.load_state_dict(self.policy_net.
    state_dict())  # Target net is updated
551 
```

```
552     def next_training_step(self, current_coordinates, object_gripped):
553         """
554         This method implements the Reinforcement Learning algorithm to control the UR3 robot.  As the algorithm is prepared
555         to be executed in real life, rewards and final states cannot be received until the action is finished, which is the
556         beginning of next loop. Therefore, during an execution of this function, an action will be calculated and the
557         previous action, its reward and its final state will be stored in the replay memory.
558         :param current_coordinates: Tuple of float indicating current coordinates of the robot
559         :param object_gripped: Boolean indicating whether or not ann object has been gripped
560         :return: action taken
561         """
562         self.statistics.new_step()  # Add new steps statistics
563         self.previous_state = self.current_state  # Previous state information to store in the Replay Memory
564         previous_action = self.current_action  # Previous action to store in the Replay Memory
565         previous_action_idx = self.current_action_idx  # Previous action index to store in the Replay Memory
566         previous_image = self.em.gather_image_state()  # Gathers current state image
567 
568         self.current_state = State(current_coordinates[0], current_coordinates[1], self.em.pick_probability,
569                                    object_gripped, self.em.image_tensor)  # Updates current_state
570 
571         # Calculates previous action reward an establish whether the current state is terminal or not
572         previous_reward, is_final_state = self.em.calculate_reward(previous_image)
573         action, random_action = self.agent.select_action(self.current_state,
574                                                          self.policy_net)  # Calculates action
575 
576         # There are some defined rules that the next action have to accomplish depending on the previous action
577         action_ok = False
578         while not action_ok:
579             # Its forbidden to perform two cosecutive pick actions in the same place
580             if action == 'pick' and previous_action != 'pick':
```

```
581                 action_ok = True
582             # If previous action was south, it is forbidden to
    perform a 'north' action for
583             # The robot not to go back to the original position.
584             elif action == 'north' and previous_action != 'south':
585                 action_ok = True
586             # If previous action was north, it is forbidden to
    perform a 'south' action for
587             # The robot not to go back to the original position.
588             elif action == 'south' and previous_action != 'north':
589                 action_ok = True
590             # If previous action was east, it is forbidden to
    perform a 'west' action for
591             # The robot not to go back to the original position.
592             elif action == 'west' and previous_action != 'east':
593                 action_ok = True
594             # If previous action was west, it is forbidden to
    perform a 'east' action for
595             # The robot not to go back to the original position.
596             elif action == 'east' and previous_action != 'west':
597                 action_ok = True
598             elif action == 'random_state':
599                 action_ok = True
600             else:
601                 action, random_action = self.agent.select_action(
    self.current_state,
602
    self.policy_net)  # Calculates action
603
604         if random_action:
605             self.statistics.random_action()  # Recolecting
    statistics
606
607         # Random_state actions are used just to initialize the
    environment to a random position, so it is not taken into
608         # account while storing state information in the Replay
    Memory.
609         # If previous action was a random_state and it is not the
    first step of the training
610         if previous_action != 'random_state' and self.statistics.
    current_step > 1:
611             self.memory.push(  # Pushing experience to Replay
    Memory
612                 Experience(  # Using an Experience namedtuple
613                     self.previous_state.image_raw,  # Initial
    state image
614                     torch.tensor([[self.previous_state.
    coordinate_x, self.previous_state.coordinate_y]],
615                                  device=self.device),  # Initial
```

```
    coordinates
616                     self.previous_state.pick_probability,
617                     torch.tensor([previous_action_idx], device=
    self.device),  # Action taken
618                     self.current_state.image_raw,  # Final state
    image
619                     torch.tensor([[self.current_state.coordinate_x
    ,
620                                    self.current_state.coordinate_y
    ]],
621                                   device=self.device),  # Final
    coordinates
622                     self.current_state.pick_probability,
623                     torch.tensor([previous_reward], device=self.
    device),  # Action reward
624                     torch.tensor([is_final_state], device=self.
    device)  # Episode ended
625                 ))
626
627             # Logging information
628             rospy.loginfo("Step: {}, Episode: {}, Previous reward:
     {}, Previous action: {}".format(
629                 self.statistics.current_step - 1,
630                 self.statistics.episode,
631                 previous_reward,
632                 previous_action))
633
634             self.train_net()  # Both policy and target networks
    gets trained
635
636         return action
```

.2.3 Environment.py

```
1 """
2 This class defines a RL environment for a pick and place task with
      a UR3 robot.
3 This environment is defined by its center (both cartesian and
    angular coordinates), the total length of its x and y axis
4 and other parameters
5 """
6
7 import random
8 # from BlobDetector.BlobDetector import BlobDetector
9 # from ai_manager.ImageController import ImageController
10 # from math import floor
11
12
```

```
13 class Environment:
14     X_LENGTH = 0.175  # Total length of the x axis environment in
       meters
15     Y_LENGTH = 0.225  # Total length of the y axis environment in
       meters
16 
17     CAMERA_SECURITY_MARGIN = 0.035  # As the camera is really
       close to the gripping point, it needs  a security marging
18     X_LIMIT = X_LENGTH - CAMERA_SECURITY_MARGIN  # Robot
       boundaries of movement in axis X
19     Y_LIMIT = Y_LENGTH - CAMERA_SECURITY_MARGIN  # Robot
       boundaries of movement in axis Y
20 
21     CARTESIAN_CENTER = [-0.31899288568, -0.00357907370787,
       0.376611799631]  # Cartesian center of the RL environment
22     ANGULAR_CENTER = [2.7776150703430176, -1.5684941450702112,
       1.299912452697754, -1.3755658308612269,
23                       -1.5422008673297327, -0.3250663916217249]  #
        Angular center of the RL environment
24     PLACE_CARTESIAN_CENTER = [0, 0.25, CARTESIAN_CENTER[2]]  #
       Cartesian center of the place box
25     ANGULAR_PICTURE_PLACE = [1.615200161933899,
       -1.235102955495016, 0.739865779876709, -1.2438910643206995,
       -1.5095704237567347, -0.06187755266298467]
26 
27     PICK_DISTANCE = 0.01  # Distance to the object when the robot
       is performing the pick and place action
28     ACTION_DISTANCE = 0.02  # Distance to the object when the
       robot is performing the pick and place action
29 
30     ENV_BOUNDS_TOLERANCE = 0
31 
32     @staticmethod
33     def generate_random_state(image=None, strategy='ncc'):
34         """
35         Calculates random coordinates inside the Relative
       Environment defined.
36         To help the robot empty the box, the generated coordinates
        won't be in the center of the box, because this is
37         the most reachable place of the box.
38 
39         :param strategy: strategy used to calculate random_state
       coordinates
40         :return:
41         """
42         def generate_random_coordinates():
43             coordinate_x = random.uniform((-Environment.X_LIMIT +
       Environment.ENV_BOUNDS_TOLERANCE) / 2,
44                                           (Environment.X_LIMIT -
```

```
Environment.ENV_BOUNDS_TOLERANCE) / 2)
45          coordinate_y = random.uniform((-Environment.Y_LIMIT +
Environment.ENV_BOUNDS_TOLERANCE) / 2,
46                                        (Environment.Y_LIMIT -
Environment.ENV_BOUNDS_TOLERANCE) / 2)
47          return coordinate_x, coordinate_y
48
49      # Random coordinates avoiding the ones in the center,
which have a bigger probability of being reached by the
50      # robot.
51      if strategy == 'ncc' or strategy == '
non_centered_coordinates':
52          coordinates_in_center = True
53          while coordinates_in_center:
54              coordinate_x, coordinate_y =
generate_random_coordinates()
55              if abs(coordinate_x) > (Environment.X_LIMIT / 4)
or abs(coordinate_y) > (Environment.Y_LIMIT / 4):
56                  coordinates_in_center = False
57      elif strategy == 'optimal' and image is not None:  #
Before going to a random state, we check that there are pieces
in this place
58          blob_detector = BlobDetector(x_length=Environment.
X_LENGTH, y_length=Environment.Y_LENGTH, columns=4, rows=4)
59          optimal_quadrant = blob_detector.find_optimal_quadrant
(image)
60          optimal_point = blob_detector.quadrants_center[
optimal_quadrant]
61
62          coordinate_x = optimal_point[0] * 0.056
63          coordinate_y = optimal_point[1] * 0.056
64      else: # Totally random coordinates
65          coordinate_x, coordinate_y =
generate_random_coordinates()
66
67      return [coordinate_x, coordinate_y]
68
69   @staticmethod
70   def get_relative_corner(corner):
71      """
72      Function used to calculate the coordinates of the
environment corners relative to the CARTESIAN_CENTER.
73
74      :param corner: it indicates the corner that we want to get
 the coordinates. It' s composed by two letters
75      that indicate the cardinality. For example: ne indicates
North-East corner
76      :return coordinate_x, coordinate_y:
77      """
```

```
78          if corner == 'sw' or corner == 'ws':
79              return -Environment.X_LIMIT / 2, Environment.Y_LIMIT /
     2
80          if corner == 'nw' or corner == 'wn':
81              return Environment.X_LIMIT / 2, Environment.Y_LIMIT /
    2
82          if corner == 'ne' or corner == 'en':
83              return Environment.X_LIMIT / 2, -Environment.Y_LIMIT /
     2
84          if corner == 'se' or corner == 'es':
85              return -Environment.X_LIMIT / 2, -Environment.Y_LIMIT
    / 2
86
87      @staticmethod
88      def is_terminal_state(coordinates, object_gripped):
89          """
90          Function used to determine if the current state of the
    robot is terminal or not
91          :return: bool
92          """
93          def get_limits(length): return length / 2 - Environment.
    ENV_BOUNDS_TOLERANCE  # functon to calculate the box boundaries
94          x_limit_reached = abs(coordinates[0]) > get_limits(
    Environment.X_LIMIT)  # x boundary reached
95          y_limit_reached = abs(coordinates[1]) > get_limits(
    Environment.Y_LIMIT)  # y boundary reached
96          return x_limit_reached or y_limit_reached or
    object_gripped # If one or both or the boundaries are reached
    --> terminal state
```

.2.4 ImageController.py

```
1 #!/usr/bin/env python
2 # coding: utf-8
3
4 import os
5 import time
6
7 import rospy
8 from PIL import Image as PILImage
9 from sensor_msgs.msg import Image
10
11 """
12 This class is used to manage sensor_msgs Images.
13 """
14
15 class ImageController:
16     def __init__(self, path=os.path.dirname(os.path.realpath(
```

```
    __file__)), image_topic='/usb_cam/image_raw'):
17          self.ind_saved_images = 0  # Index which will tell us the
    number of images that have been saved
18          self.success_path = "{}/success".format(path)  # Path
    where the images are going to be saved
19          self.fail_path = "{}/fail".format(path)  # Path where the
    images are going to be saved
20          self.image_topic = image_topic
21
22          # If it does not exist, we create the path folder in our
    workspace
23          try:
24              os.stat(self.success_path)
25          except:
26              os.mkdir(self.success_path)
27
28          # If it does not exist, we create the path folder in our
    workspace
29          try:
30              os.stat(self.fail_path)
31          except:
32              os.mkdir(self.fail_path)
33
34      def get_image(self):
35          msg = rospy.wait_for_message(self.image_topic, Image)
36
37          return self.to_pil(msg), msg.width, msg.height
38
39      def record_image(self, img, success):
40          path = self.success_path if success else self.fail_path  #
     The path were we want to save the image is
41
42          image_path = '{}/img{}.png'.format(  # Saving image
43              path,  # Path
44              time.time())  # FIFO queue
45
46          img.save(image_path)
47
48          self.ind_saved_images += 1  # Index increment
49
50      def to_pil(self, msg, display=False):
51          size = (msg.width, msg.height)  # Image size
52          img = PILImage.frombytes('RGB', size, msg.data)  #
    sensor_msg to Image
53          return img
54
55  if __name__ == '__main__':
56      rospy.init_node('image_recorder')  # ROS node initialization
57      image_controller = ImageController(path='/home/pilar/Pilar/
```

```
    ros_pictures', image_topic='/usb_cam2/image_raw')
58      img, width, height = image_controller.get_image()
59      image_controller.record_image(img, True)
```

BIBLIOGRAPHY

[1] *Formación en línea de CB3*. URL: https://academy.universal-robots.com/es/formacion-en-linea-gratuita/formacion-en-linea-de-cb3/ (visited on 11/15/2020).

[2] Preferred Networks, Inc. *Bin-picking Robot Deep Learning*. 2015. URL: https://www.youtube.com/watch?v=ydh_AdWZflA&ab_channel=Pickit3D (visited on 11/27/2020).

[3] Guillaume Lample and Devendra Singh Chaplot. "Playing FPS Games with Deep Reinforcement Learning". In: *arXiv:1609.05521 [cs]* (Jan. 29, 2018). arXiv: 1609.05521. URL: http://arxiv.org/abs/1609.05521 (visited on 11/30/2020).

[4] Y. Zhu et al. "Target-driven visual navigation in indoor scenes using deep reinforcement learning". In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*. 2017 IEEE International Conference on Robotics and Automation (ICRA). May 2017, pp. 3357–3364. DOI: 10.1109/ICRA.2017.7989381.

[5] *Reinforcement Learning - Goal Oriented Intelligence*. URL: https://deeplizard.com/learn/playlist/PLZbbT5o_s2xoWNVdDudn51XM8lOuZ_Njv (visited on 11/28/2020).

[6] OpenAI. *Gym: A toolkit for developing and comparing reinforcement learning algorithms*. URL: https://gym.openai.com (visited on 11/30/2020).

[7] A. Rupam Mahmood et al. "Setting up a Reinforcement Learning Task with a Real-World Robot". In: *arXiv:1803.07067 [cs, stat]* (Mar. 19, 2018). arXiv: 1803.07067. URL: http://arxiv.org/abs/1803.07067 (visited on 11/30/2020).

[8] Xiaoyun Lei, Zhian Zhang, and Peifang Dong. "Dynamic Path Planning of Unknown Environment Based on Deep Reinforcement Learning". In: *Journal of Robotics* 2018 (Sept. 18, 2018), pp. 1–10. ISSN: 1687-9600, 1687-9619. DOI: 10.1155/2018/5781591. URL: https://www.hindawi.com/journals/jr/2018/5781591/ (visited on 11/30/2020).

[9] Marco Wiering. "Reinforcement Learning in Dynamic Environments using Instantiated Information". In: (Aug. 27, 2001).

[10] *Agile Methodology: What is Agile Software Development Model?* URL: https://www.guru99.com/agile-scrum-extreme-testing.html (visited on 12/02/2020).