



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

<i>1. Introduction</i>	1
1.1 Project Motivation	3
<i>2. Description of Technologies</i>	4
2.1 ROS + catkin	4
2.2 pytorch	6
2.3 arduino	6
2.4 github	7
2.5 CUDA	8
2.6 moveit	9
2.7 Universal Robots driver for ROS	9
2.8 UR3 robot	10
2.9 anaconda	10
<i>3. State of the art</i>	12
3.0.1 Reinforcement Learning	13
3.0.2 Deep Reinforcement Learning	15
3.0.3 Problems of Deep Reinforcement Learning in Real-world . .	23

<i>4. Definition of the Project</i>	27
4.1 Motivation	27
4.2 Objectives	27
4.3 Methodology	28
4.4 Planning and budget	29
4.4.1 Budget	29
<i>5. Developed System</i>	32
5.1 Hardware Architecture	33
5.2 Logical Architecture	35
5.3 ai_manager	43
5.3.1 Definition of the problem	43
5.3.2 Environment.py	44
5.3.3 Rewards	44
5.3.4 Algorithm	45
5.3.5 Training Flow	48
5.3.6 Image Model	50
5.4 Robot Controller	51
5.4.1 Robot.py	52
5.4.2 Block Detector	54
5.5 Gripper	55
5.6 Algorithm and System optimization	59
5.6.1 Performance	59

5.6.2	Training	60
5.6.3	Why to use Block Detector?	60
6.	<i>Results Analysis</i>	64
7.	<i>Conclusions and Future Work</i>	65
 <i>Appendix</i>		66
.1	Robot Controller	67
.1.1	main.py	67
.1.2	Robot.py	69
.2	Artificial Intelligence Manager	75
.2.1	main.py	75
.2.2	RALgorithm.py	77
.2.3	Environment.py	95
.2.4	ImageController.py	98
 <i>Bibliography</i>		100

LIST OF FIGURES

1.1	Pick and Place Task	2
3.1	Fanuc DNN	13
3.2	Q-Matrix	16
3.3	Deep Q Learning	17
3.4	RL Training	24
4.1	Methodology	29
4.2	Chronograph HW	29
4.3	Chronograph HW	30
4.4	Planing Robot Controller	30
5.1	pieces	32
5.2	Picture of Architecture I	33
5.3	Picture of Architecture II	34
5.4	Logical Architecture	36
5.5	Architecture I	37
5.6	Architecture II	38
5.7	Architecture III	39
5.8	Architecture IV	41

5.9	Node Interction	42
5.10	Training Step	48
5.11	Training Process	49
5.12	Actions	51
5.13	Upper View	54
5.14	Block Detector	55
5.15	Gripper	57
5.16	Robot Heatmap	61
5.17	Heatmap robot II	62

LIST OF TABLES

4.1 Project's Budget	31
5.1 Difference in performance between CPU and GPU trainings	59

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 tasks 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.

2.1 *ROS + catkin*



The first decision we had to make was about the architecture of the project. Was it a good idea to build all the project in the same computer? How should we communicate with the Robot?

We found the best solution for this question in ROS (Robot Operative System), which is a framework for the development of software for robots that provides the functionality of an operating system in a heterogeneous cluster. ROS was originally developed in 2007 under the name switchyard by the Stanford Artificial Intelligence Laboratory to support the Stanford Artificial Intelligence Robot (STAIR) project. Since 2008, development has continued primarily at Willow Garage, a robotic research institute with more than twenty institutions collaborating on a federated development model.

ROS provides the standard services of an operating system such as hardware abstraction, low-level device control, implementation of commonly used functionality, message passing between processes, and package maintenance. It is based on

a graph architecture where the processing takes place in the nodes that can receive, send and multiplex messages from sensors, control, states, schedules and actuators, among others. The library is oriented for a UNIX system (Ubuntu (Linux)) although it is also adapting to other operating systems such as Fedora, Mac OS X, Arch, Gentoo, OpenSUSE, Slackware, Debian or Microsoft Windows, considered as 'experimental'. ROS is free software under BSD license terms. This license allows freedom for commercial and investigative use. Contributions of packages in ros-pkg are under a variety of different licenses.

All of these features made ROS ideal for the project. Specially the following ones:

- Universal Robots drivers for ROS using mooveit make it possible to controll ROS remotelly.
- ROS is a multi-node oriented framework, which allows us to take all the advantages of micro-services. We can split the software by functionallity gaining:
 - The possibility of giving more or less computation power to each functionality depending on its needs. In the project we have useed from computers with the better Nvidia Graphic card and 32 GB of RAM to other mini-computers such as a Raspberry-pi or an Arduino Card.
 - The chance of using a different environment for each functionality. Different versions of python and even different programming languages, different versions of libraries, different Operative Systems, etc.
 - Isolation of each component of the project, allowing us to develop sepratly each functionality without affecting the rest of the functionalities of the project.
- It can work over an Arduino Card. It is not self sufficient, as it needs to be serial connected to a computer (Or Raspberry pi) in orther to work. We needed the arduino card in order to build our "home made" vaccuum gripper.
- It is open source and have a huge community, so we could reuse some already developed solutions such as the usb_cam package, which let us take pictures from a camara connected to another node or computer.



2.2 *pytorch*

PyTorch is a Python package designed to perform numerical calculations using tensor programming. It also allows its execution on GPU to speed up calculations.

Typically PyTorch is used both to replace numpy and process calculations on GPUs and for research and development in the field of machine learning, mainly focused on the development of neural networks. In this case we will use PyTorch in both the Reinforcement Learning algorithm and the image Processing.

PyTorch is a very recent library and despite this it has a large number of manuals and tutorials where to find examples. In addition to a community that grows by leaps and bounds.

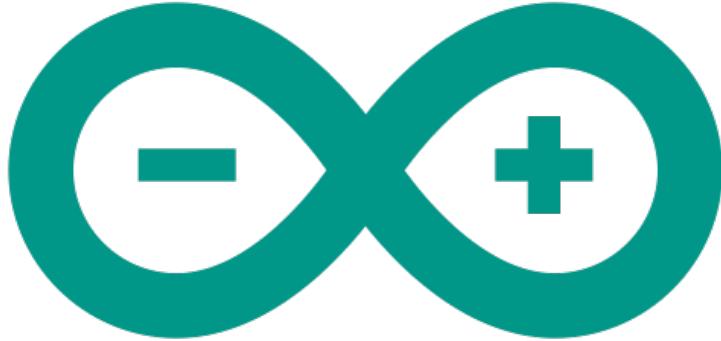
PyTorch has a very simple interface for creating neural networks despite working directly with tensors without the need for a library at a higher level such as Keras for Theano or Tensorflow.

Unlike other packages like Tensorflow, PyTorch works with dynamic graphs instead of static ones. This means that at runtime the functions can be modified and the calculation of the gradient will vary with them. On the other hand, in Tensorflow we must first define the computation graph and then use the session to calculate the results of the tensors, this makes it difficult to debug the code and makes its implementation more tedious.

PyTorch has support to run on graphics cards (GPU), it uses internally CUDA, an API that connects the CPU with the GPU that has been developed by NVIDIA.

2.3 *arduino*

Arduino is an open source electronics creation platform, which is based on free hardware and software, flexible and easy to use for creators and developers. This



platform allows the creation of different types of single-board microcomputers that can be used by the developer community for different types of use.

As commented before, we will use Arduino Card in order to build a Vacuum Gripper for the Robot. It will be connected with all the other nodes using ROS Queues.

2.4 *github*

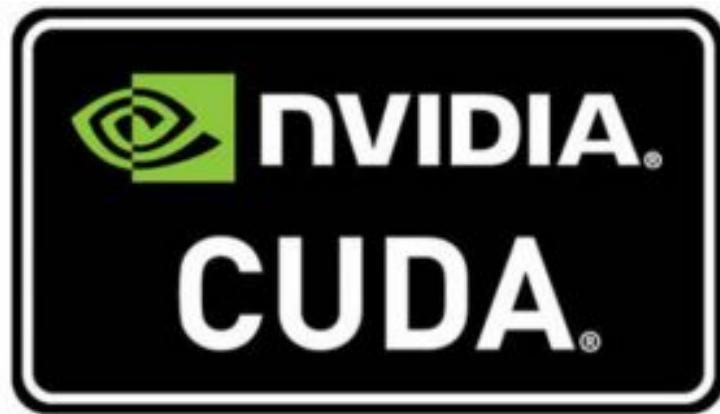


GitHub, Inc. is a provider of Internet hosting for software development and version control using Git. It offers the distributed version control and source code management (SCM) functionality of Git, plus its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, continuous integration and wikis for every project. Headquartered in California, it has been a subsidiary of Microsoft since 2018.

GitHub offers its basic services free of charge. Its more advanced professional and enterprise services are commercial. Free GitHub accounts are commonly used to host open-source projects. As of January 2019, GitHub offers unlimited private repositories to all plans, including free accounts, but allowed only up to three collaborators per repository for free. Starting from April 15, 2020, the free plan allows unlimited collaborators, but restricts private repositories to 2,000 minutes

of GitHub Actions per month. As of January 2020, GitHub reports having over 40 million users and more than 190 million repositories (including at least 28 million public repositories), making it the largest host of source code in the world.

2.5 CUDA



CUDA stands for Compute Unified Device Architecture, which refers to a parallel computing platform including a compiler and a set of development tools created by nvidia that allow programmers to use a variation of the language. C programming for encoding algorithms on nvidia GPUs.

Through wrappers you can use Python, Fortran and Java instead of C / C ++.

Works on all nvidia GPUs from the G8X series onwards, including GeForce, Quadro, ION, and the Tesla line.¹ nvidia claims that programs developed for the GeForce 8 series will also work without modification on all future nvidia cards, thanks to binary compatibility.

CUDA tries to exploit the advantages of GPUs over general purpose CPUs by using the parallelism offered by its multiple cores, which allow the launch of a very high number of simultaneous threads. Therefore, if an application is designed using multiple threads that perform independent tasks (which is what GPUs do when processing graphics, their natural task), a GPU will be able to offer great performance in fields that could range from computational biology to biology, crypto, for example.

The first SDK was released in February 2007 initially for Windows, Linux, and later in version 2.0 for Mac OS. It is currently offered for Windows XP / Vista /

7/8/102, for Linux 32/64 bits3 and for macOS4.

2.6 moveit



MoveIt! is open source software for ROS (Robot Operating System) which is state of the art software for mobile manipulation. In fact, we could say that MoveIt! it is becoming a de facto standard in the field of mobile robotics, as today more than 65 robots use this software, including the latest robots developed by Robotnik.

MoveIt! includes various utilities that speed up the work with robotic arms, and it helps to not be continually “reinventing the wheel”, following the ROS philosophy of code reuse.

2.7 Universal Robots driver for ROS

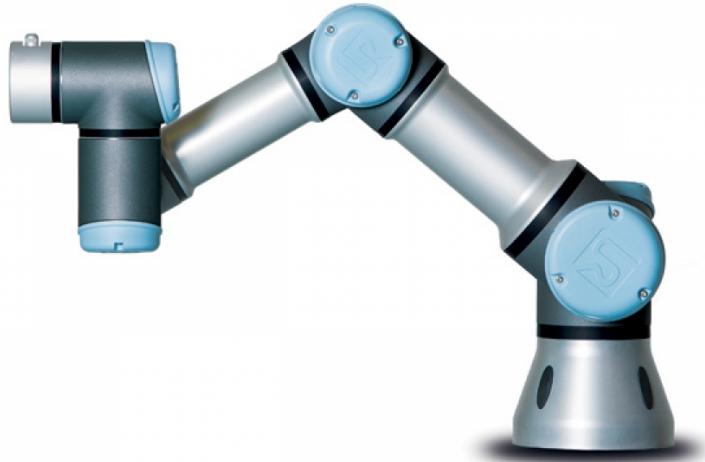


Universal Robots have become a dominant supplier of lightweight, robotic manipulators for industry, as well as for scientific research and education. The Robot Operating System (ROS) has developed from a community-centered movement to a mature framework and quasi standard, providing a rich set of powerful tools for robot engineers and researchers, working in many different domains.

With the release of UR's new e-Series, the demand for a ROS driver that supports the new manipulators and the newest ROS releases and paradigms like ROS-control has increased further. The goal of this driver is to provide a stable and sustainable interface between UR robots and ROS that strongly benefit all parties.

2.8 UR3 robot

The UR3 Universal Robots robot is the smallest cobot of the UR series of Universal. Universal Robots' ultra flexible UR3 provides high precision for the smallest production environments.



It can modulate payloads of up to 3 kg, adding value to scientific, pharmaceutical, agricultural, electronic and technological facilities. Tasks the UR3 excels at include: mounting small objects, gluing, screwing, tool handling, welding and painting.

However, the range of movements of this robot is really limited, and it can only lift up 3 Kilograms so this robot is really good for preparing a prototype, it has the same characteristics than its big brothers, but probably it is not good enough to build a production solution.

2.9 anaconda



Anaconda is a free and open distribution of the Python and R languages, used in data science, and machine learning. This includes processing of large volumes of information, predictive analytics and scientific computations. It is aimed at simplifying the deployment and management of software packages.

The different versions of the packages are managed through the conda package management system, which makes it quite easy to install, run, and update data science and machine learning software such as Scikit-team, TensorFlow and SciPy.

The Anaconda distribution is used by 6 million users and includes more than 250 data science packages valid for Windows, Linux, and MacOS.

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.

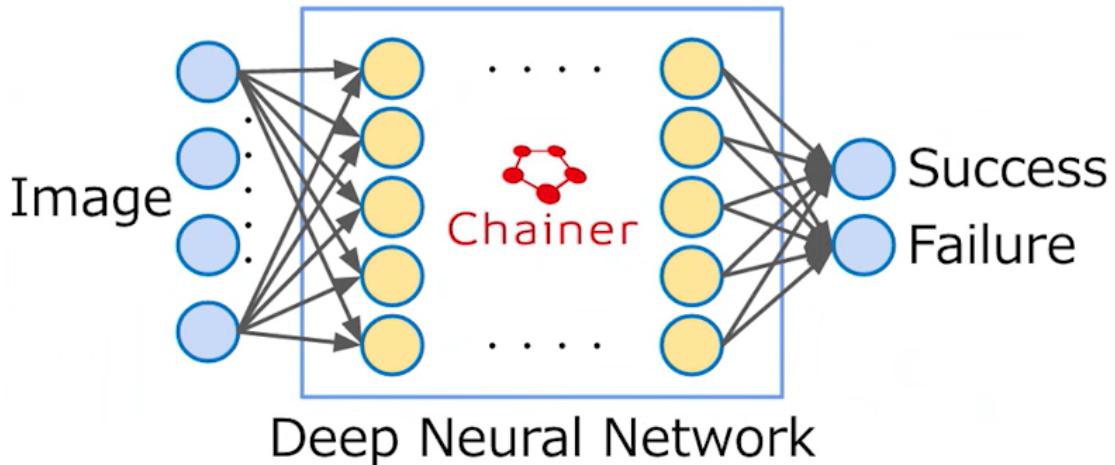


Fig. 3.1: Deep Neural Network of Fanuc solution (taken from the video)

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 the next state: $S_t = S_{t+1}$.
- Environment produces a new reward, which can be represented with the following formula:

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

Agent's performance is calculated in terms of its future accumulated rewards known as return. This is called **expected return** and 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.

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

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](#).

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

		Action					
		0	1	2	3	4	5
State	0	-1	-1	-1	-1	0	-1
	1	-1	-1	-1	0	-1	100
	2	-1	-1	-1	0	-1	-1
	3	-1	0	0	-1	0	-1
	4	0	-1	-1	0	-1	100
	5	-1	0	-1	-1	0	100

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

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 are equivalent to the q-values saw before and so we will call them. To see it graph-

ically, 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.

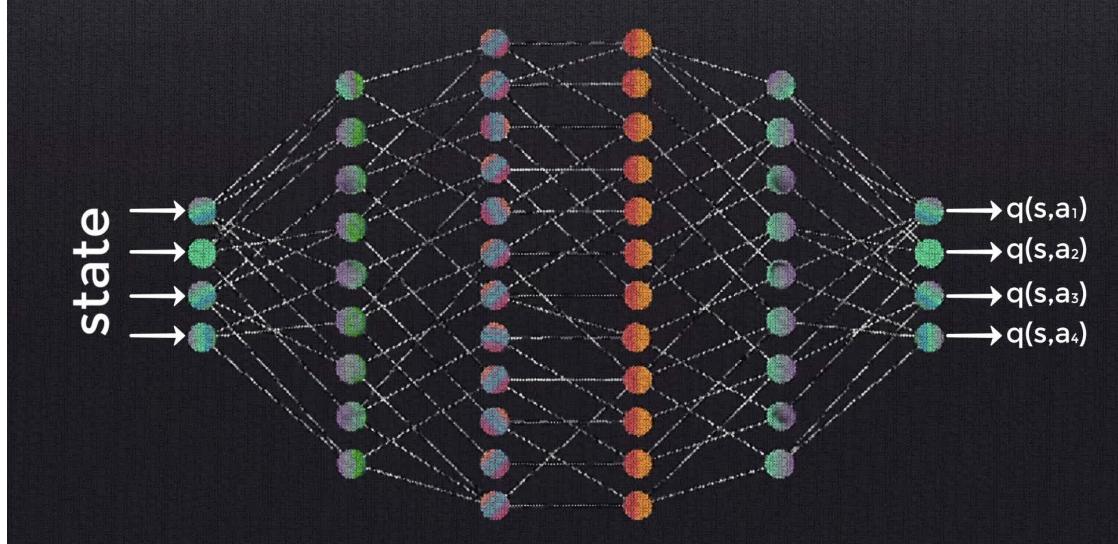


Fig. 3.3: Deep Q Learning Representation with 4 outputs

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
- 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 ($\epsilon = \epsilon - \epsilon_decay$)

Using this strategy, the agent will rather explore or exploit the environment during the training. In the first steps the probability of a random action (exploring) will be much higher than in the last steps of the algorithm. This probability will keep decreasing during the training, until it reaches the minimum exploring rate, which is normally set to 10

Replay Memory

Every time 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 that Reinforcement Learning Networks converge really slowly and variance between consecutive steps is really low. Using consecutive experiences to train the network would result though in a slower and biased learning. Besides, this way of working is better for learning real-world experience,

where there are infinite different states, as the experience gained in previous steps will be used multiple times later to train the network.

Bellman's Equation

As commented before, Deep Reinforcement Learning uses Neural Networks to compute the q-values of each action. The optimal value of these q-values is represented by the Bellman's Equation and is shown below:

$$q_*(s, a) = E[R_t + \gamma \max(q_*(s', a'))]$$

As we can see in the equation, the optimal value depends in both the reward of the action taken and the maximum optimal q-value of the next action. In real life it is impossible to compute this value, because we would be an infinite loop. However, as the most important parameter of the formula is the expected reward ($q_*(s', a')$ is multiplied by the discount factor γ), we can simply use the next action q-value and it will be a good approximation. The formula would stay as follows:

$$q_*(s, a) = E[R_t + \gamma \max(q(s', a'))]$$

With this new formula we will be able to compute the optimal q-value for each experience stored in the Replay Memory (initial state, action, final state and reward). It is important to have in mind for this process that the optimal q-value can only be computed if the action has actually been taken, because we don't know the Reward of non taken actions. But, anyway, why do we need to compute the optimal q-value?

To answer this question, lets take a look to the training process of the neural network:

1. the agent decide which action to take using the policy-network. (action with highest q-value)
2. The agent takes the action and receives a reward from the environment
3. The agent stores all the experience in the Replay Memory

-
4. The training process is started:
- (a) A random batch of experiences is taken from the Replay Memory
 - (b) For all these experiences, its optimal q-value is calculated using the modified Bellman's Equation and target-network
 - (c) For all these experiences, the actual q-value is calculated using the policy-network
 - (d) For all these experiences, the loss is calculated as the difference of both values
 - (e) We use the Neural network optimizer to back-propagate the loss to all the weights

So, to answer the previous question, we need to compute the **optimal q-values in order to calculate the loss** of the neural network for each action taken and train, though, the algorithm.

Retaking here the question answered before about why we needed Replay Memory module, one important reason is that one action taken in the initial steps of the training will affect differently to the neural network in this moment than later, when the network is already trained, and its q-value is though more similar to the optimal q-value. Replay Memory technique allow us to use this information gathered in any step of the training, during a step where the network is more trained.

Target and Policy Networks

In the previous step, we talk about two different networks: policy and target. The target network comes to solve a stability problem of the DRL training. In the next paragraphs I will explain the problem and how target network can help to solve it.

Having in mind the way we calculate the loss of the neural network in the previous section we can realize that we have to pass information through the network twice. Just to remember:

$$loss = R_t + \gamma \max(q(s_{t+1}, a_{t+1})) - q(s_t, a_t)$$

As a spoiler, I can say that $q(s_{t+1})$ and $q(s_t, a_t)$ will not be calculated with the same network. But.. why?

Imagine that we have an experience, which is composed of an initial state, an action that has been taken, the state reached with this action and a reward. Remembering previous section, the loss of the neural network is calculated as the difference between the q-value of the initial state and the action taken and the optimal q-value of the expected cumulative reward (or target value) of the action.

Q-values are calculated using the states as input of Neural Networks. Let's see what could happen if we calculated both of the values with the same network. In this case, once we had the loss calculated, we would use back-propagation to adapt the weights of the Neural Network and make the q-value of the initial state more similar to the target q-value.

The problem here comes because as both q-values are using the same Neural network to be calculated, when we change the weights to move the initial q-value to one direction, the target q-value is moving in the same direction, so we have not reduce the distance between the two values. It is basically like a dog chasing its tail.

To solve this problem we introduce the target-network. This network is basically a frozen clone of the policy network that we only use to calculate the target value of each action. This way, when we gain stability during the training of the RL Algorithm. The target-network is updated periodically after a certain amount of steps, so is always updated.

DQL Training

During the previous sections I have been explaining a lot of concepts about Deep Reinforcement Learning Training. I have explained them and how they affect the training. It is a really complex process so probably a sum-up will help understanding it.

The training basically uses the following schema:

- Initialize replay memory capacity.
- Initialize the policy network with random weights.
- Clone the policy network, and call it the target network.

-
- For each episode:
 - Initialize the starting state.
 - For each time step:
 - Select an action via exploration or exploitation
 - Execute selected action and observe reward and next state.
 - Store experience in replay memory.
 - Sample random batch from replay memory.
 - Preprocess states from batch.
 - Pass batch of preprocessed states to policy network.
 - NN training. Weight back-propagation:
 - Calculate loss between output Q-values and target Q-values.
 - Using both the target and the policy networks to increase stability.
 - Gradient descent updates weights in the policy network to minimize loss.
 - After X time steps or episodes, weights in the target network are updated to the weights in the policy network.

This training can also be explained with [Figure 3.4](#), where the Pool is the Replay Memory that stores the sample (experiences) from real interaction with the environment and feeds the training node of random sample of experiences.

Then, we can see that we use the Q-Network (policy network) to predict actions, but also to calculate the q-value of Replay Memory's batch. Then, we use the target network to predict action-value of state s_1 with the predicted action in Q estimation Network, and this network is updated in a low rate.

Finally, the action taken in the Dynamic Environment can be predicted by the Neural Network or Randomly chosen (Stochastic search). This is the way of representing epsilon-Greedy Strategy.

3.0.3 Problems of Deep Reinforcement Learning in Real-world

Real-world problems introduce some challenges that we will have to manage. In March 2018, A. Rupam Mahmood, Dmytro Korenkevych, Brent J. Komer,

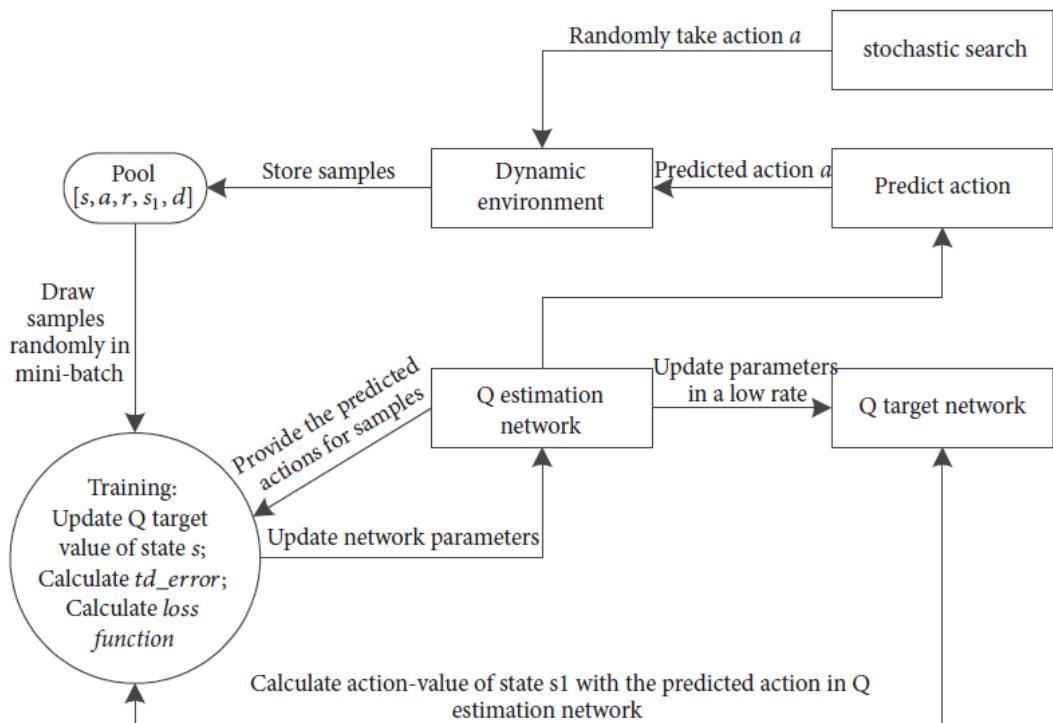


Fig. 3.4: Reinforcement Learning Training Summary

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*

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.
- Analyse the results and test new solutions to improve them.
- 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

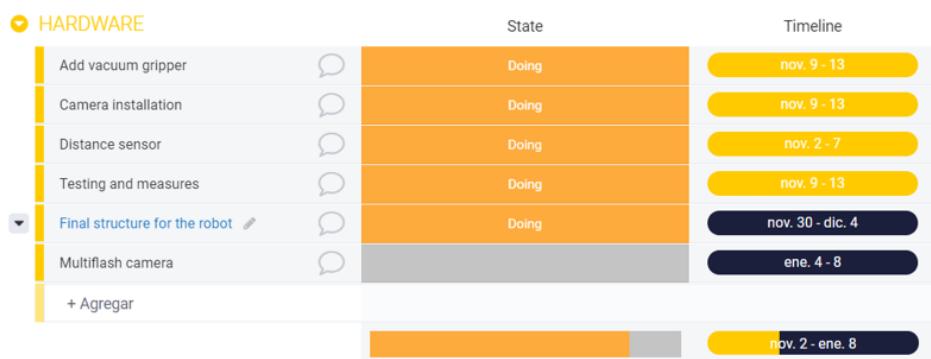


Fig. 4.2: Chronograph of the Hardware implementation

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](#).

4.4.1 Budget

The budget of the project was really limited because it was not sponsored by any company. The cost was mainly due to hardware acquisition. For example, the gripper was design and built by us in order to save thousands of euros.

In the following table we can see the detailed budget of all the project:

4.4. Planning and budget

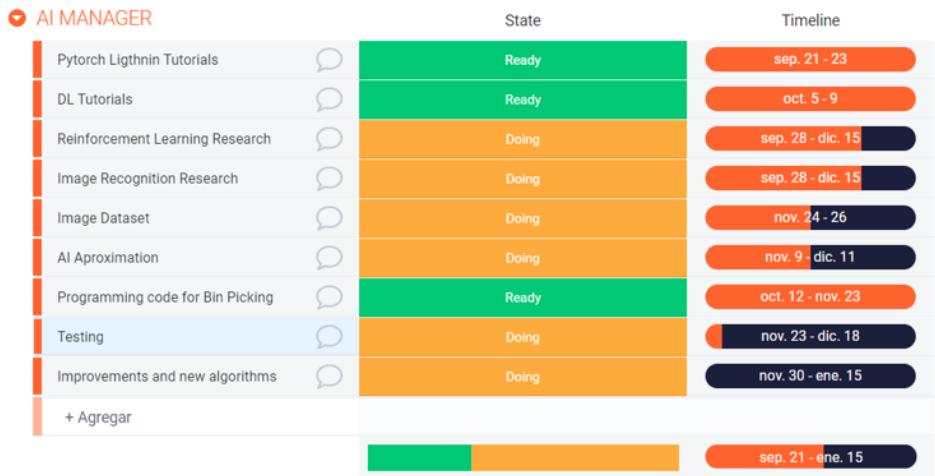


Fig. 4.3: Chronograph of the Hardward implementation

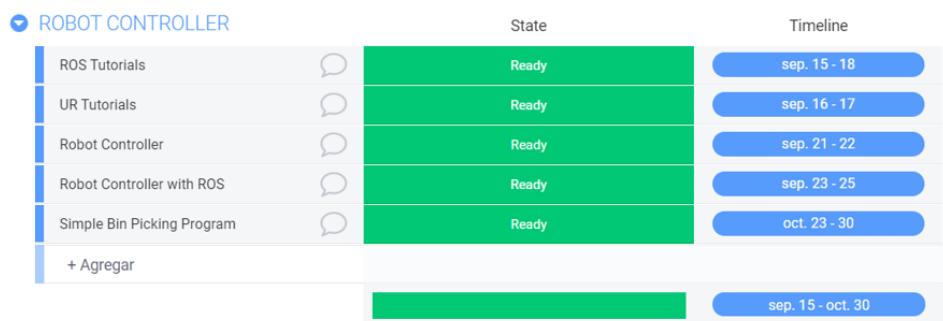


Fig. 4.4: Planning of the Robot Controller implementation

4.4. Planning and budget

ITEM	QUANTITY	PRICE
Nvidia GeForce GTX	2	395,88
Vacuum Gripper	1	53,50
Raspberry Pi4	1	59,05
Electronic Parts (Sensors, transistors..)	30	
Arduino	1	22,64
HDMI->VGA	2	63,31
QWERTY keyboards	2	36,85
Total		661,23

Tab. 4.1: Project's Budget

5. DEVELOPED SYSTEM

In previous sections of this documents, I have explained the main idea of the project, and what technologies are going to be used to develop a fully functional system. Just to remember, the objective of the project is to perform a Pick and Place task using a Universal Robots' UR3 robotical arm.

The objects have to be taken from a box (Environment Box) and placed in another box (Place box). The pieces had to be something light due to the limitations of the UR3 robot that we commented before. As there wasn't any sponsor for the project we could decide the shape of the pieces, and we decided to use 5 cm size wooden squares as the ones showed in [Figure 5.1](#).



Fig. 5.1: Wooden Pieces used in the project

5.1 Hardware Architecture

In order to make this system work, we need a really complex architecture that can be split in three different categories. These categories are:

- Environment
- sensorimotor devices
- Computational devices

To understand it better, we are going to use [Figure 5.2](#) and [Figure 5.3](#), which are labeled pictures of the architecture, where we are going to be able to see how the components of the architecture are like.

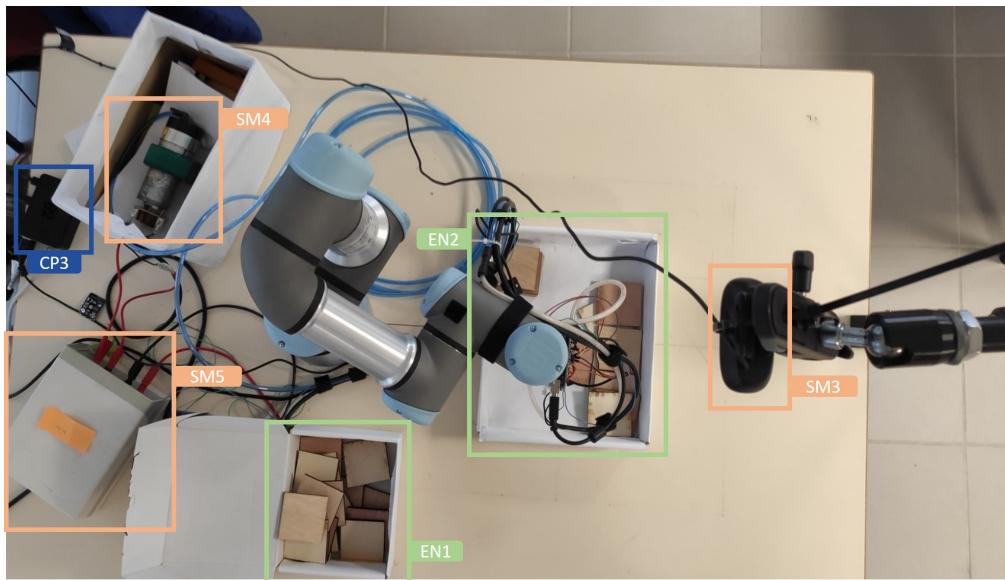


Fig. 5.2: Labelled picture of the Architecture I

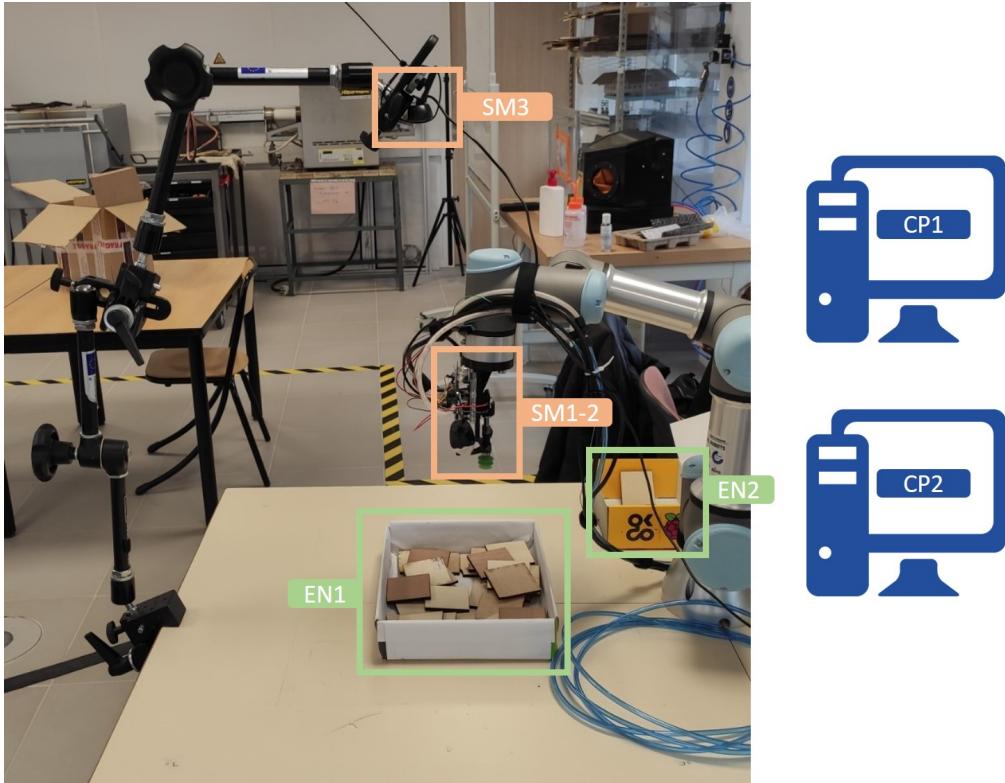


Fig. 5.3: Labelled picture of the Architecture II

In the pictures we can see multiple elements tagged with different colours and labels. Each colour represents a category.

The environment elements, that can be found in green, and labelled with EN, are basically all the things that the robot will have to interact with. The sensorimotor elements, that can be found in orange and with the label SM, are all the elements needed to allow the robot interact with the environment. And finally, in blue and with label CP, we can find the computational elements, which are the ones used to receive all the sensor information, decide which movement to do, and communicate with the robot and the gripper for them to actually perform these actions.

But, what are all these elements? Let's explain them:

- **Environment:** We can see this elements in both images, from different perspective.

- **EN1:** The Environment box where the agent has to take the pieces
- **EN2:** The box where the agent has to place the pieces
- **Sensorimotor devices** Ones are showed in one image, and others in the other.
 - **SM1:** This is the Onboard camera, used for the agent to decide which action to take. It is attached to the gripper, so in the picture they are shown together.
 - **SM2:** Together with the camera, we can see the "Home made" gripper used to pick the pieces.
 - **SM3:** the upper camera, where the agent can pick a global picture of the environment. This picture will be important, but we will explain it later.
 - **SM4:** the pump of the Gripper.
 - **SM5:** Both 12V and 24V power adaptor used to feed the pump and some sensors.
- **Computational devices.** In the picture we cannot see all the computer used in the project, but there are 2 icons used to represent them.
 - **CP1:** This is the ROS Master Node. All of the nodes of the system will be connected to this node. Besides being the master node, robot_controller node and Universal Robots driver will also be running in this computer.
 - **CP2:** This computer is a really powerful one, with one of the best graphical cards in the market and 32 GB of RAM. It will be used to train the algorithm, running the ai_manager node.
 - **CP3:** This mini-computer can be seen in one of the pictures and it's a Raspberry-pi. This computer will be used as a bridge from the arduino card of the gripper and the ROS master node. The cameras will also be attached to the Raspberry-pi.

5.2 Logical Architecture

Once we have seen the physical architecture of the project, let's see how the Software architecture is. The Logical architecture will use all the elements commented

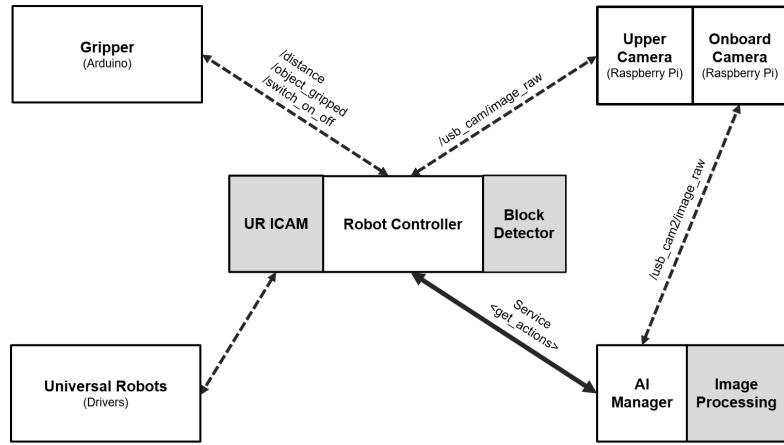


Fig. 5.4: Logical Architecture of the project

on the previous section, and they will work together using ROS (Robot Operative System).

In [Figure 5.4](#) we can see the logical architecture of the application, which is composed of 6 nodes communicating one with each other. We can see that the communication topics are written in the figure and that there are some squares attached one to another, and some of them are grey. All the white squares are ROS nodes, while grey ones are separate pieces of code that the nodes are using, but they are not ROS nodes by themselves. On the other hand, both camera nodes that are together in the upper right corner are two independent nodes, but using the same code to send the cameras images.

To explain briefly what every node does, probably it is easier going step by step from the simplest architecture to the final one, so that we see what every node is doing.

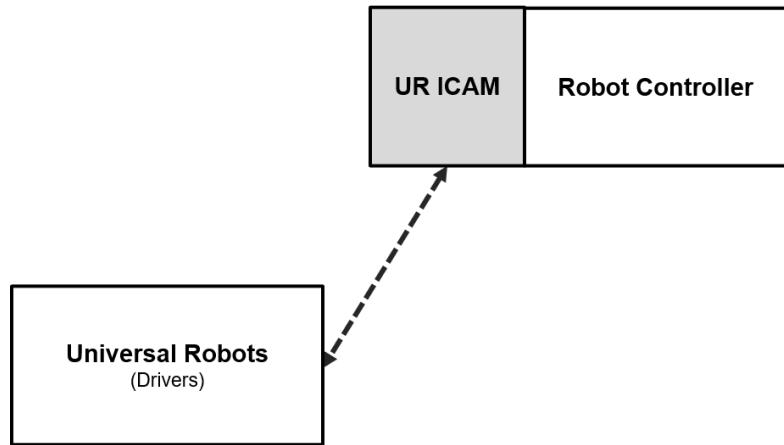


Fig. 5.5: From the simplest to the final Architecture I

In [Figure 5.5](#), there are only three components. However, Both Universal Robots Drivers and UR ICAM were not developed by us and are like black-boxes for us. That is the reason why there are no ROS topics written in the communication the figure.

Universal Robot Drivers is the one that actually communicates with the robot, and provides all the basic tools needed to control it remotely. On the other side, UR ICAM node is a node developed in the university, and it provides us some methods to control easier the robot. These methods are a personalization of the ones provided by MoveIt library, and they make us possible to go to some angular coordinates of the robot without calculating the optimal path to reach this position, or the same thing with some cartesian coordinates.

Finally, the Robot Controller node is the one that actually is communicating with all the othe nodes of the architecture. All the actions that the robot will be able to perform are here, so just with Robot Controller node we could almost be able to implement a silly random agent to perform a pick and place task.

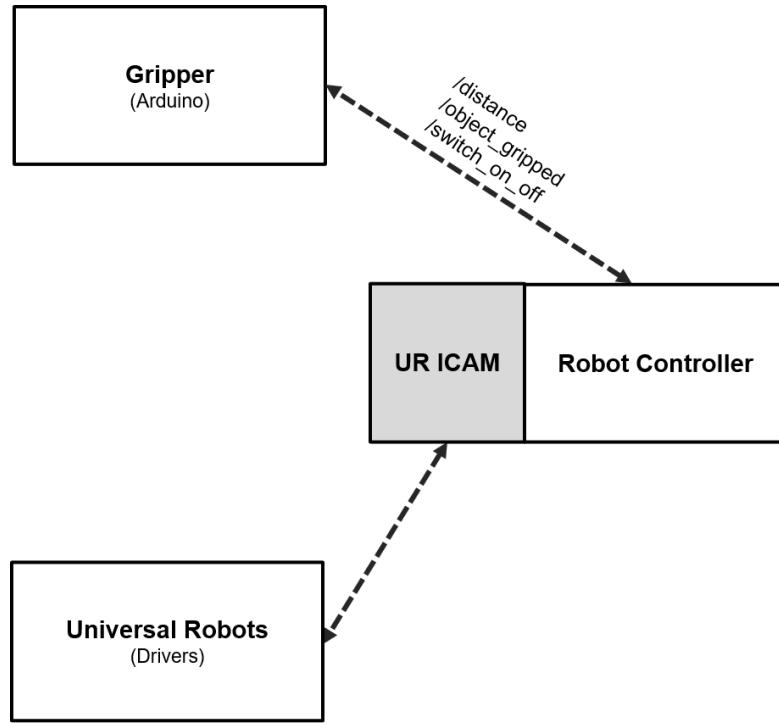


Fig. 5.6: From the simplest to the final Architecture II

I said almost, because to perform a pick and place task we also need the gripper node, as we can see in [Figure 5.6](#). This node is running in an arduino carda communicates by serial port with a Raspberry Pi, which is also connected with the master node. The gripper node uses 3 different topics to communicate with Robot Controller:

- \textbf{distance}: The gripper is publishing continuously if the gripper is being pushed up or not. Robot Controller wants this information to know when to stop during the pick movement. The Robot basically goes down while \textbf{distance} are "False" and stops when they are "True".
- \textbf{switch_on_off}: The gripper listens to this topic. When it receives a "True" message it switch the gripper on, and when it receives a "False" message it switch the gripper off.
- \textbf{object_gripped}: The gripper is publishing continuously if there is an object gripped or not. Robot Controller use this information during the pick action. When this action is finished, robot controller checks if an object has

been picked or not by reading from this topic. If an object has been picked it goes to the box to place the object and, if not, it just finishes the pick action and request AI Manager for a new action.

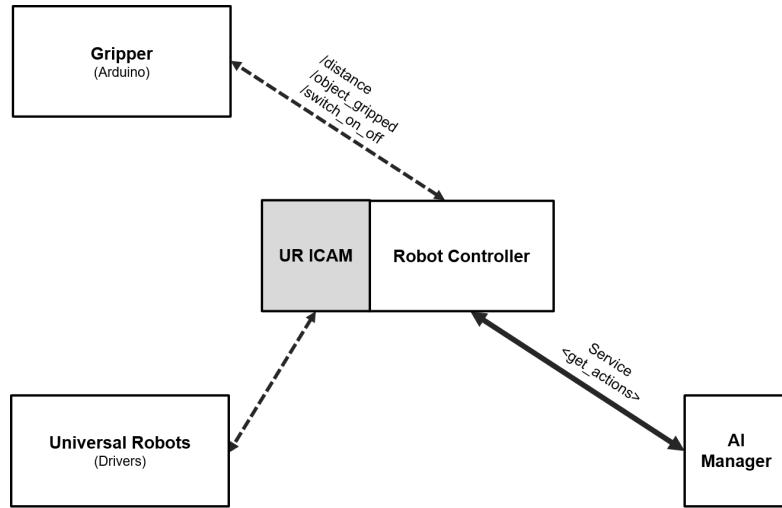


Fig. 5.7: From the simplest to the final Architecture III

The next step in our trip would be adding the AI Manager Node. This Node is the one who decides which action to perform in each time step. It receives the Coordinate, request an image of the environment (Which in this step is still simulated) and trains a Reinforcement Learning Algorithm to decide which action to perform in each step.

As we can see in the [Figure 5.7](#), the arrow representing the communication between Robot Controller and AI Manager is the only continuous line. This is because the communication method is different, in this case we are using ROS Services instead of publishing the messages in topics.

In ROS, the most common way of communicating is using ROS Messages. ROS Messages are simple data structures that are send to a topic, which is basically a queue stored in the Master node. Then, other nodes can be subscribed to this topics so every time that they are free, they ask the master node about the unread messages in the topic. This is called asynchronous communication and, it is probably the best way of sending messages between nodes, because it allows the receiver to adapt its computational needs to the message load received from the topics.

However, in this case asynchronous communication was not possible, because ROS Messages does not ensure the delivery. This was a problem because the Robot Controller could request an action to the AI Manager, and the second one could not receive the message. This is not a problem in this direction because we can solve it by putting a timeout in Robot Controller, and it could make a new request after x time.

Anyway, this could not be a solution because we need to avoid the AI Manager node to receive the same request twice. This is needed because every time that the AI Manager receives a new action, a new step of the training is performed: A reward is given, random probability decreases, Experience is saved, etc.

ROS Services is the way of performing synchronous communication in ROS, and it ensures that every message is delivered once and only once. AI Manager is though a resting node that does nothing until the Robot Controller nod request an action. It then start calculating the action, trains the net and return the selected action.

get_actions services is defined by two structures:

– Request structure:

- **x_coordinate:** X coordinate of the robot used to calculate the reward and training the robot.
- **y_coordinate:** X coordinate of the robot used to calculate the reward and training the robot.
- **object_gripped:** Boolean telling whether the robot has an object gripped or not. It is used to calculate the reward of pick actions.

– Response structure:

- String telling the action selected

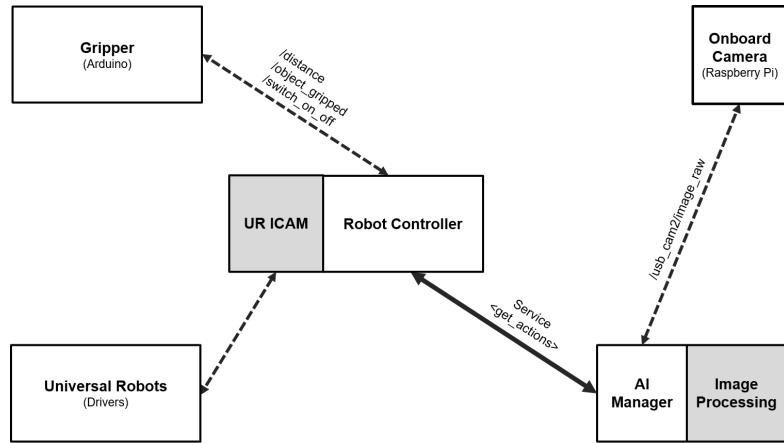


Fig. 5.8: From the simplest to the final Architecture IV

We commented before that AI Manager needs to gather a state image in order to start the training. To gather this image, it just requests a message from **`-usb_cam-image_raw`** topic. Messages of this topic are published by Onboard Camera node, which is basically an instance of `usb_cam` node that publishes with a 30 fps rate the images of the Onboard camera of the robot.

But once the Ai Manager has the image, it has to process it and extract its features, and Ai Manager will do it using the models in Image Processing. We will talk deeply about this later, but it basically means to make some transformation to the image (Changing its shape, color, rotation, etc.) and pass it through a pre-trained Convolutional Neural Network in order to extract some features. These features are actually the ones that will be passed through the Reinforcement Learning Neural Network and the ones that will be stored in the Replay Memory.

Finally, the last two nodes that we have not commented from [Figure 5.4](#) are the Block Detector and the Upper Camera Node. Block Detector is a piece of code used by Robot Controller when it is performing a place action. In this moment, the robot is out of the box and has to decide the initial coordinates of the next episode. The upper camera has, though, a full view of the environment, so the Robot Controller takes the environment picture from the topic **`-usb_cam2-image_raw`** where the Upper Camera node is publishing the images of the upper camera. Then it passes the image to the Block Detector which finally calculates the optimal point of return, avoiding the places of the box where there are no pieces.

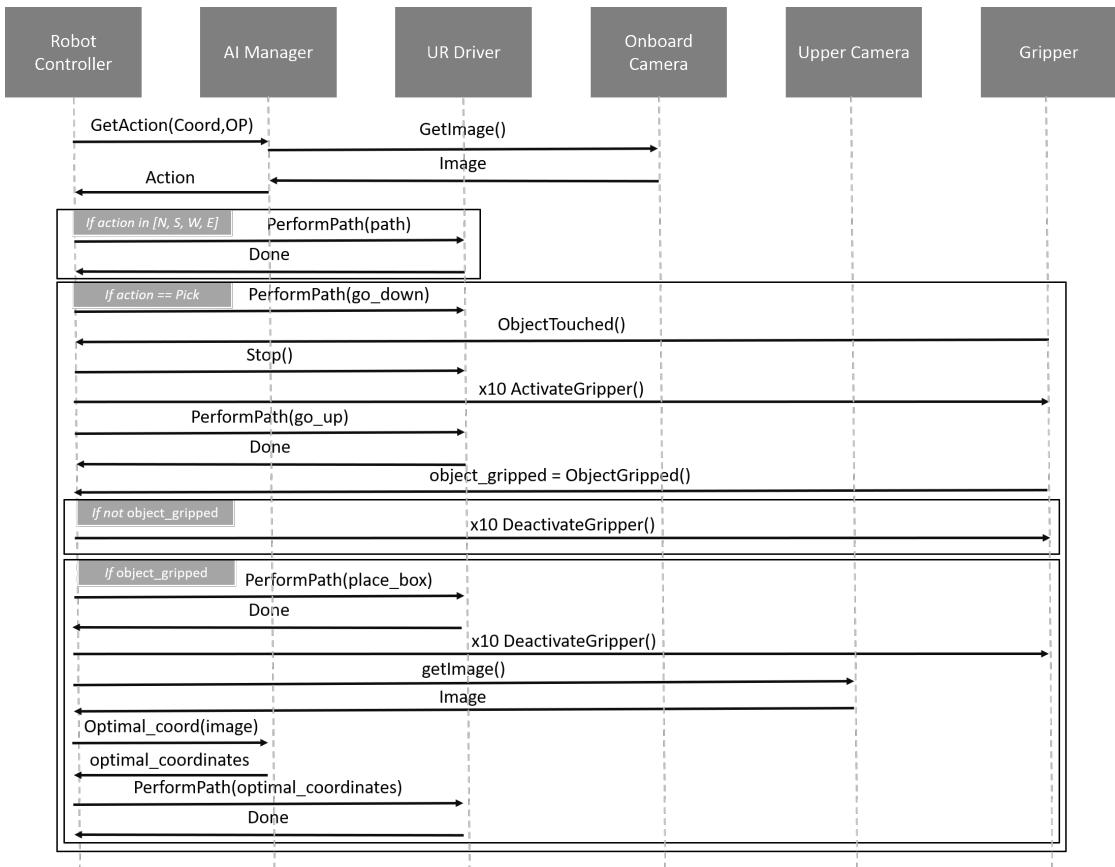


Fig. 5.9: Flow Chart of nodes interacting

To understand better all the architecture, in [Figure 5.9](#) we can find a flow chart showing the iteration between nodes in one step. This flow would be in an infinite loop until the training is over. We can see how the step always starts with Robot Controller Asking AI Manager which action to perform, and AI Manager retrieving a picture from Onboard Camera to decide the action and train the Reinforcement Learning Algorithm. Behind this steps there is a complex process that we will talk about later.

Then, depending on the action to perform, the flow would be really simple or more complex. If action is Pick, the robot has to start an asynchronous downward movement that will only be stopped once Robot Controller receive a True Message in the `\distance` topic, which means that the robot is now in contact with an object. Then, it will activate the gripper, go upwards to the original position and, if the robot has picked an object, perform a place action to put the object in the

place box.

This is a really simplified flow, but its a very good graphical way of understanding how the system works. To go deeply into the system, lets analyse each node separately:

5.3 *ai_manager*

ai_manager module is the "intelligence" of the robot, responsible for making it learn by training a Deep Reinforcement Learning Algorithm. Using this algorithm, the robot (**agent**) will explore the **Environment** by performing a set of **actions**. Once these actions are performed, the agent will receive a **reward** that can be positive, neutral or negative depending on how far the agent is from the objective.

Each time the agent perform an action, it reaches a new **state**. States can be transitional or terminal, when the agent meets the objective or when it gets to a forbidden position. Each time the agent reaches a terminal state, an **episode** is ended, and a new one is started.

The code of the AI Manager can be found in the appendix of this document, where a link to the github repository can also be found.

5.3.1 *Definition of the problem*

The objective of the agent is thus the first thing that has to be defined. In this case is simple: pick a piece.

Then, the environment, the states and the actions have to be defined together. These decisions are conditioned by the hardware and materials available. In our case, as said before, we have a UR3 robot with six different points of movements, and a vacuum gripper. That means that the best way of griping an object is by facing the gripper to the floor and move it vertically until it gets in contact with the object, where the vacuum can be powered on, and we can know if the object has been griped or not.

Having this in mind, we have decided that the robot have to be fixed in a specific height with the gripper facing down. Then, the actions will be "north", "south", "east" or "west" to move the robot through the x-y plane formed by these movements in the selected robot height, "pick", to perform the gripping process

described before and place the object in the box, and "random_state" to move the robot to a new random state when a terminal state is reached.

5.3.2 *Environment.py*

The environment is defined in Environment.py class. There, we can find different parameters and methods. All of them are explained in the code, but we will briefly explain them here. The CARTESIAN_CENTER and the ANGULAR_CENTER represent the same point in the space, but using different coordinates. This point should be the x-y center of the picking box with the robot height defined before as z point. As starting point, we need to use the ANGULAR_CENTER because we want the robot to reach this point with the gripper facing down.

Then, we have to define the edges of the box as terminal states, because we just want the robot to explore inside the box. To define those limits, we use X_LENGTH and Y_LENGTH parameters, which are the X and Y lengths of the box in cm.

Other important parameters to define are the center of the box where we will place all the objects (PLACE_CARTESIAN_CENTER) or the distance that the robot has to move in each action (ACTION_DISTANCE).

Finally, the methods defined in this class are:

- ***generate_random_state(strategy='ncc')***, which is used when the agent reaches a terminal state and needs a new random state.
- ***get_relative_corner(corner)***, which returns the relative coordinates of a corner of the box
- ***is_terminal_state(coordinates, object_gripped)***, which returns a boolean telling whether a given state is terminal or not using the parameters given.

5.3.3 *Rewards*

Rewards are one of the most difficult-to-define parameters. In this case, rewards are defined in the EnvManager inner class of RLAlgorithm.py. The specific value of the rewards are not given here because they are different from one training to another, but we give (positive or negative) rewards for:

- Terminal state after picking a piece.
- Terminal state after exceeding the box limits.
- Non terminal state after a pick action
- Other non terminal states

5.3.4 *Algorithm*

This Deep Q Learning algorithm is implemented in the class RLAlgorithm.py following this schema:

- Initialize replay memory capacity.
- Initialize the policy network with random weights.
- Clone the policy network, and call it the target network.
- For each episode:
 - Initialize the starting state.
 - For each time step:
 - Select an action via exploration or exploitation
 - Execute selected action and observe reward and next state.
 - Store experience in replay memory.
 - Sample random batch from replay memory.
 - Preprocess states from batch.
 - Pass batch of preprocessed states to policy network.
 - NN training. Weight back-propagation:
 - Calculate loss between output Q-values and target Q-values.
 - Using both the target and the policy networks to increase stability.
 - Gradient descent updates weights in the policy network to minimize loss.
 - After X time steps or episodes, weights in the target network are updated to the weights in the policy network.

This schema is a little bit difficult to understand in the first moment, but it is deeply explained in the State of The Art section of this document.

RLAlgorithm.py

RLAlgorithm.py is the most important file of this module because it is the place where the algorithm implementation is done. Several classes have been used to implement the algorithm. Some of these classes are defined inside *RLAlgorithm* (inner classes) and others are normal outer classes. In *RLAlgorithm.py*, we define the *RLAlgorithm* class, which also have several inner classes. These classes are:

- **Agent:** Inner class used to define the agent. The most important thing about this class is the `select_action` method, which is the one used to calculate the action using whether Exploration or Exploitation.
- **DQN:** Inner class used to define the target and policy networks. It defines a neural network that have to be called using the vector of features calculated by passing the image through the feature extractor net.
- **EnvManager:** Inner Class used to manage the RL environment. It is used to perform actions such as calculate rewards or gather the current state of the robot. The most important methods are:
 - **calculate_reward**, which calculates the reward of each action depending on the initial and final state.
 - **calculate_reward**, which calculates the reward of each action depending on the initial and final state.
 - **extract_image_features**, which is used to transform the image to extract image features by passing it through a pre-trained CNN network that can be found in *ImageModel* Module.
- **EpsilonGreedyStrategy:** Inner Class used to perform the Epsilon greedy strategy
- **QValues:** Inner class used to get the predicted q-values from the `policy.net` for the specific state-action pairs passed in. States and actions are the state-action pairs that were sampled from replay memory.
- **ReplayMemory:** Inner Class used to create a Replay Memory for the RL algorithm
- **Environment:** Class where the RL Environment is defined
- **TrainingStatistics:** Class used to store all the training statistics. If it is run separately, It will plot a set of graphs to represent visually the training evolution.

- **ImageModel:** Class used to extract the image features used in the training. You can find this class in this repository, which store another module of this project.
- **ImageController:** Class used to gather and store the relative state images from a ros topic.

In order to implement the algorithm there are two important structures that are defined in the beginning of this file. These structures are:

- **State**, which defines all the things needed to represent a State:
 - Coordinates of the robot.
 - Image of the State.
 - Boolean telling if an object has been gripped.
- **Experience**, which represents the experience of the agent in a given moment:
 - The initial state of the agent (Image).
 - The initial coordinates of the agent.
 - The action taken by the agent.
 - The state reached after taking the action (Image).
 - The coordinates reached after taking the action.
 - The reward obtained for taking this action.
 - Boolean telling whether the final state is terminal or not.

Finally, there are some important methods in RLAlgorithm class that it is important to take into account to understand how this node works:

- **save_training:** Method used to save the training so that it can be retaken later. It uses pickle library to do so and stores the whole RLAlgorithm object because all the context is needed to retake the training. This method also stores a pickle a TrainingStatistics object for them to be accessible easily.
- **recover_training:** Method used to recover saved trainings. If it doesn't find a file with the name given, it creates a new RLAlgorithm object.

- **train_net**: Method used to train both the train and target Deep Q Networks. We train the network minimizing the loss between the current Q-values of the action-state tuples and the target Q-values. Target Q-values are calculated using the Bellman’s equation:

$$q_*(s, a) = E[R_t + \gamma \max(q(s', a'))]$$

- **next_training_step**: This method implements the Reinforcement Learning algorithm to control the UR3 robot. As the algorithm is prepared to be executed in real life, rewards and final states cannot be received until the action is finished, which is the beginning of next loop. Therefore, during an execution of this function, an action will be calculated and the previous action, its reward and its final state will be stored in the replay memory.

5.3.5 Training Flow

This is a really complex process that it is easier to understand watching it graphically.

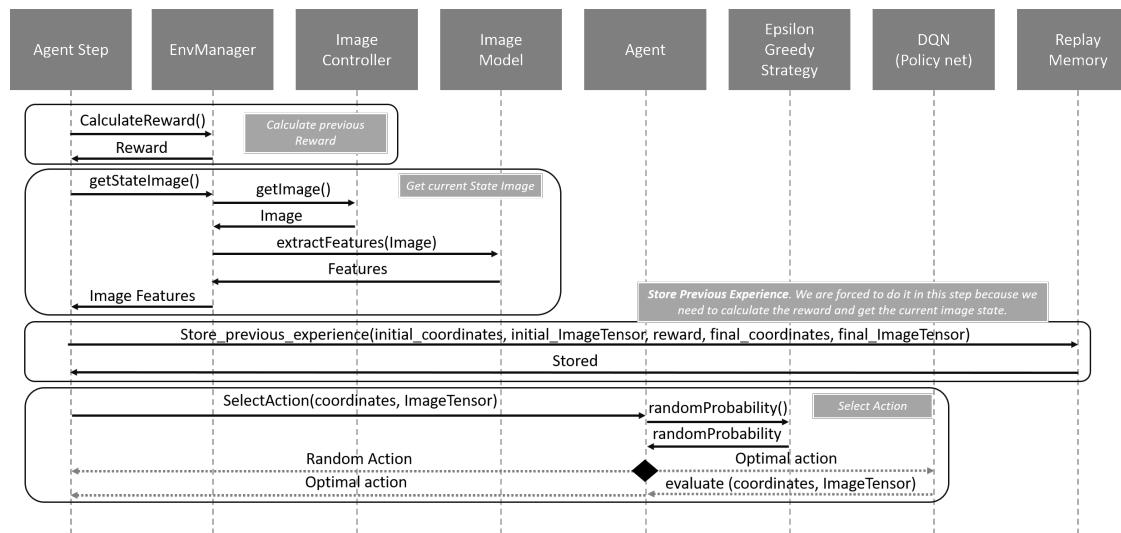


Fig. 5.10: Flow chart explaining training steps

In Figure 5.10, we can see a flow chart explaining what happens during a training step in AI Manager. In the previous section I told that we would explain

deeply what was the training process in AI Manager. We will explain it now with this chart, but, again, it is a simplified flow.

As we can see in the graph, during a training step we have to do basically 2 main tasks. On one hand, we have to calculate the action for the Robot Controller to perform it. The process is simple, we get the current state (Onboard Image and Robot Coordinates), and we pass it through the policy network in order to calculate the Q Value of each action. We get the action with highest Q Value.

On the other hand we have the weird part, which is storing on the Replay Memory the experience of previous step. We have to store the experience of step $t-1$ in step t because experience is composed on both initial and final state, and the reward. The initial state is known in step $t-1$, but the final state is not known until step $t-1$. The reward of action $t-1$ is also calculated in step t because it also depends on the final state of the agent.

Finally, we train the algorithm in every step, following the steps shown on Figure 5.11.

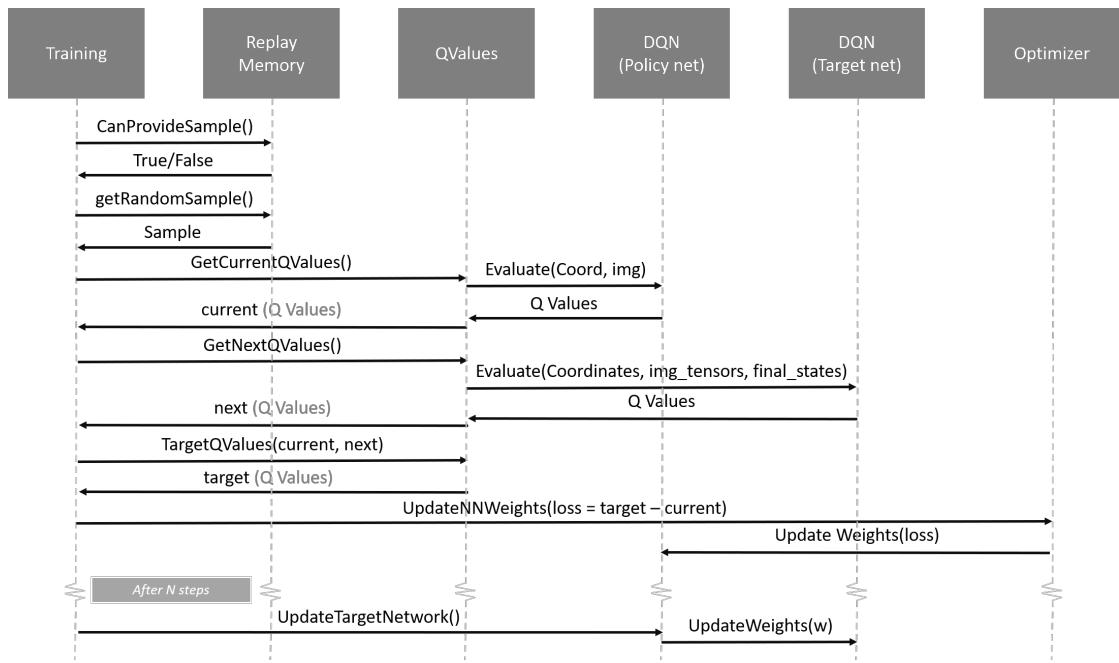


Fig. 5.11: Flow chart explaining the training process

The training process starts asking the Replay Memory if there are enough ex-

perience to supply a sample of size=batch_size. If there are enough experience, then the Reply Memory provide a random sample of experiences and the training actually starts. The next step would be splitting the batch into batches of categories and calculate the current and next Q Values of all the samples of the batch.

To calculate the current Q Values we use the policy network and to calculate the next Q values we use the target network. To understand why we use different networks it is recommended to read and try to understand the Deep Reinforcement Learning Section of the State of the Art of this document.

The process of calculating the Q Values is also a little bit different in both cases, because for the next Q Value we have to first take out (Q Value = 0) all the samples of the batch in which the final state is terminal.

Once we have the next Q Values calculated, we use the Bellman's Equation to calculate the target Q Values and then calculate the loss as the difference between the target Q Values and the current Q Values.

Finally, we back-propagate the loss using the optimizer, to modify the weights of the policy neural network. The weights of the target neural network are frozen and only updated after X steps.

5.3.6 *Image Model*

Image model is the module which is connected to AI Manager and is in charge of extracting the features of the images. It uses some Data augmentation techniques to pre process the image, and then it is passed through a Convolutional Neural network to extract its features.

We will talk about this later, but this is a really important step, not only because a good feature extraction is vital for training any neural network, but also because the size reduction of the image allows us to store a huge amount of experiences in the GPU memory without having to discard any of them because of memory problems.

However, we will not analyse this module more deeply because it was not developed by me and for the aim of this document it works just as a Black Box. The author of this module was Pilar Hernandez, that worked together with me in the project.

5.4 Robot Controller

If the AI Manager is the intelligence of the system, if we continue with the human analogy, the Robot Controller would be the body of the system. As we saw in the [Figure 5.4](#), the Robot Controller is the central node, most of the nodes communicates with it, so it is a really important node

Anyway, its complexity is much lower than the AI Manager complexity. The most important task of Robot Controller is to define the set of movements of all the actions. Just to remember, the actions are the ones showed in [Figure 5.12](#), and Robot Controller will use the UR ICAM MoveIt implementation to perform these actions.

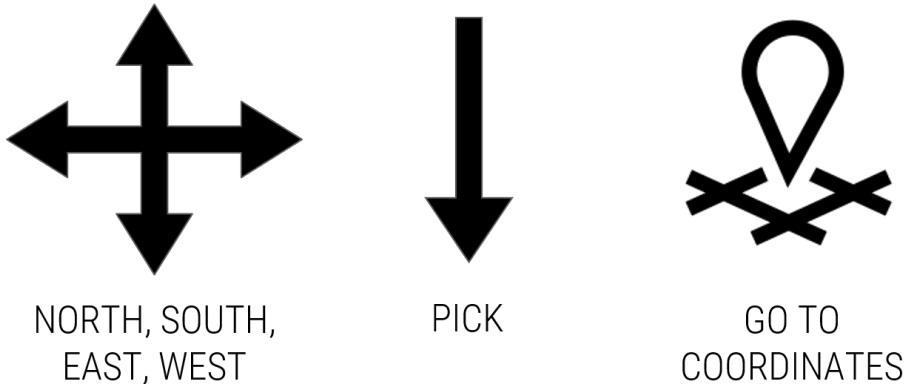


Fig. 5.12: Available actions

With MoveIt, you can tell the robot to go to a specific set of coordinates and it will automatically calculate the best path to reach this position. However, during the implementation of Robot Controller, we found problems executing this kind of movements, because the UR ICAM module wasn't calculating the path correctly, and the robot crashed against itself multiple times.

The solution to this problem was simply stop using this kind of movement, or, at least, to simplify the movement. We realised that we could perform without errors relative movements in every axis, that is to say that we could perform a movement over the axis X, a movement over the axis z but separately, not together. We called this relative move.

Let's see how Robot Controller works. In Robot Controller there are multiple files, but only two important files:

- ***Robot.py***: In this file we use UR ICAM to implement some methods as `relative_move(x, y, z)`, and implement all the actions.
- ***main.py***: In this file we basically implement an infinite loop in which asking the service `get_actions` for actions and executing them.

5.4.1 Robot.py

`Robot.py` is a python Class that implements a set of methods. It is important to say that `Robot.py` is also using the `Environment.py` class that we created in AI Manager, to retrieve all the information about the Environment. We have done it this way to make it easy to change the parameters of training in both AI Manager and Robot Controller at the same time.

The methods implemented are:

- ***relative_move(self, x, y, z)***: this function takes as parameter the distance in meters to move through each axis. The distance can be positive or negative, and the method will calculate and execute the movements to reach this position.
- ***calculate_relative_movement(self, relative_coordinates)***: This method will be used to calculate the relative moves needed to reach certain coordinates. You can see that the parameter passed is called `relative_coordinates`. This is like that because we are talking about the coordinates of our environment and not the coordinates of our robot. This make the calculation more difficult, because in UR ICAM we can only get the current Cartesian coordinates of the robot, but not the position of the robot into our environment.
To calculate the relative moves needed, though, we will have to translate the relative coordinates on robot coordinates as `Environment.CARTESIAN_CENTER - relative_coordinates` and then calculate the difference between the result and the robot current cartesian coordinates.
- ***calculate_current_coordinates(self)***: This method is the opposite problem, we use it to translate from robot Cartesian coordinates to Environment coordinates, and we use it to calculate the position of the robot into our environment.
- ***take_north(self, distance=Environment.ACTION_DISTANCE)***: This is the north action of the model. It is basically a relative movement of size `Environment.ACTION_DISTANCE` in the x axis.

- ***take_south(self, distance=Environment.ACTION_DISTANCE)***: This is the south action of the model. It is basically a negative relative movement of size Environment.ACTION_DISTANCE in the x axis.
- ***take_east(self, distance=Environment.ACTION_DISTANCE)***: This is the east action of the model. It is basically a negative relative movement of size Environment.ACTION_DISTANCE in the y axis.
- ***take_west(self, distance=Environment.ACTION_DISTANCE)***: This is the west action of the model. It is basically a relative movement of size Environment.ACTION_DISTANCE in the y axis.
- ***send_gripper_message(self, msg, timer=2, n_msg=10)***: This is a method used to activate or deactivate the gripper. It sends a burst of messages to ensure that the gripper really switch its state and waits a little bit to allow the robot to pick a piece.
- ***take_pick(self)***: This is the pick action of the model. To perform this action we have to use a new kind of movement: asynchronous relative movement. It is the same movement than before, but we can execute code during the execution of the movement.

We need to execute code during the movement because we do not know how far the pieces are, so we have to start going down and, during the execution, check the distance topic in order to know when to stop. Once the robot is in contact with an object, the gripper will send a message to the distance topic, Robot Controller will receive it and will stop the movement.

Finally, we activate the gripper using `send_gripper_message()`, will go up to the original position, and will check if the robot has picked an object or not. If it hasn't picked an object it will switch the gripper off and would finish the action, but if it has picked an object, it will execute the `take_place()` action.

To understand better this action, you can check the flow diagram showed in [Figure 5.9](#).

- ***take_place(self)***: This is the place action. This is not actually an action of the model, because it theoretically belongs to the pick action, but in the implementation of `Robot.py` we decided to split the method in two different actions.

In this action, the robot goes to the place position and then switch the gripper off. In this moment, it take a picture of the environment from the Upper Camera node. Using this picture and the Block Detector analysis, it

will calculate the point of the box with a bigger amount of pieces, and will send the coordinates to the `take_random_state()` action.

- **`take_random_state(self)`**: This is the action used to reach some coordinates of the box. Probably the name is not the best, if I had to renamed it now I would called it `go_to_initial_state()`, but at the beginning of the implementation the idea was to make this movement randomly and it took the name from it.

5.4.2 Block Detector

This module is used by Robot Controller but wasn't developed by me, so I will only introduce the goal of the module and what it does, because for the aim of this project it is like a black box. The author of this module was Pilar Hernandez, that worked together with me in the project.

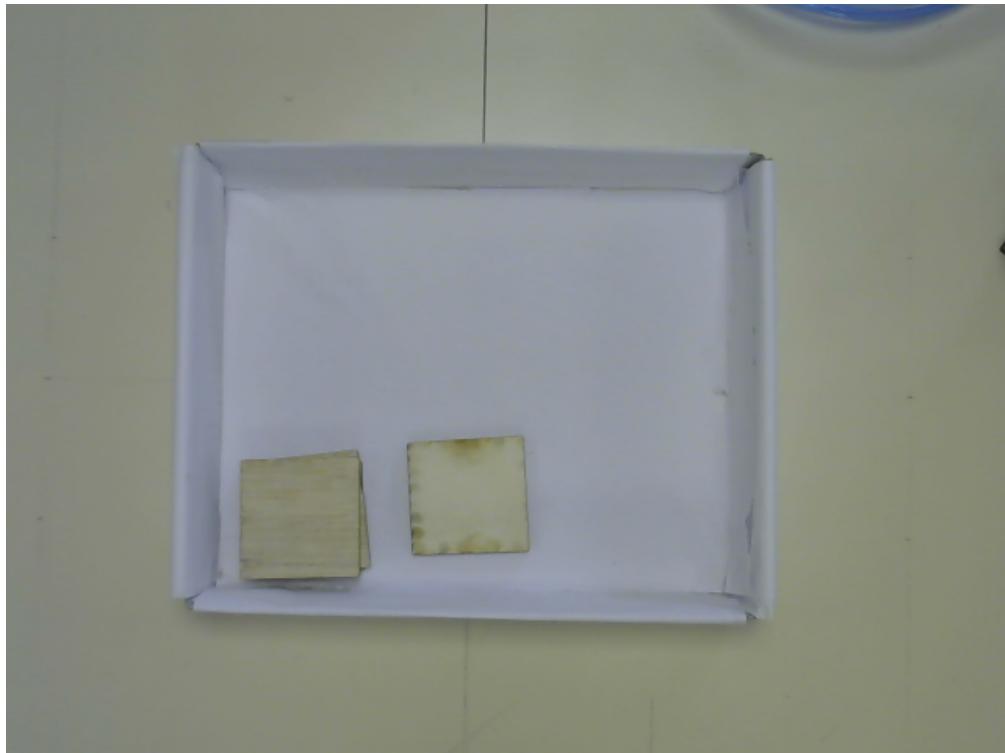


Fig. 5.13: Upper View of the environment

This module is used during the place action in order to know in which point of

the box we should start the next episode. Once the robot had placed the object, it would take a picture from the environment as the one showed in [Figure 5.13](#). Robot Controller would pass the image to the Block detector, which would perform a test similar to the one in [Figure 5.14](#), in which it has calculated all the shapes of the image, and it has calculated the place inside the box with a higher amount of points detected

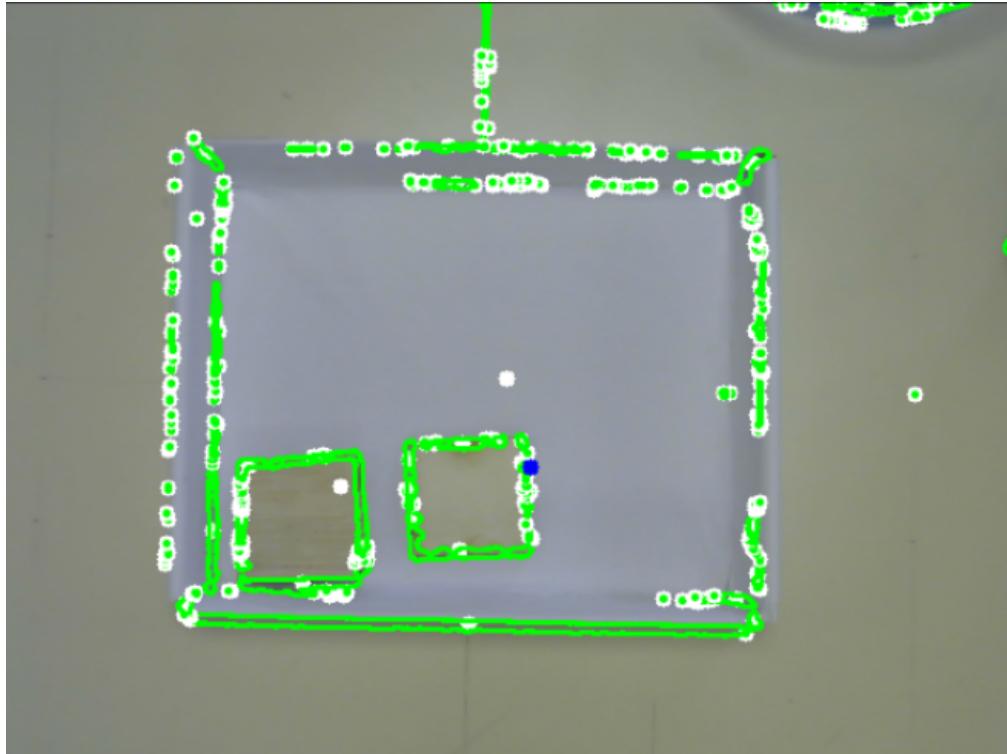


Fig. 5.14: Block Detector working

As we can see in the [Figure 5.14](#), where the blue point would represent the return point of the robot, the block detector haven't probably selected the optimal point, but it is a good one, because it is close to a piece when most of the box is empty.

5.5 Gripper

This node is the last one of our architecture. Just to remember, we decided to use a vacuum gripper because it is really easy to pick object with it. Obviously there

are objects that you cannot pick with it, but when you can, you just have to push your gripper against the object and activate the pump. The gripper used is the one showed in [Figure 5.15](#).

This node is running over an arduino card, which is the controller of the gripper. In the arduino card we have three different peripherals:



57

Fig. 5.15: Front view of the gripper

- **Contact sensor:** This is the one that detects if the gripper is in contact with an object. It isn't really a contact sensor, as we can see in [Figure 5.15](#), the sucker has actually a mechanical system that can go down or up about 1cm. When a robot touches an object, this object pushes up the sucker and, when the bar goes up it pushes a switch, which is actually the sensor.

It was really difficult to design and make this system work, but we will talk later about that.

- **Object gripped sensor:** To detect if an object has been gripped or not, we have used an air flow sensor. to make it work we connected it as a ramification of the main air pipe. This way, it will not detect any air flow when there is no object picked (All the air is taken from outside), but when an object is picked, the air flow from the outside will stop, creating a vacuum in the pipes and activating the sensor.
- **Pump switcher:** the pump has to be activated or deactivated programmatically from the arduino card.

As commented before, to read or control these peripherals we have the following topics:

- **\distance:** The gripper is publishing continuously if the gripper is being pushed up or not. Robot Controller wants this information to know when to stop during the pick movement. The Robot basically goes down while **\distance** are "False" and stops when they are "True".
- **\switch_on_off:** The gripper listens to this topic. When it receives a "True" message it switch the gripper on, and when it receives a "False" message it switch the gripper off.
- **\object_gripped:** The gripper is publishing continuously if there is an object gripped or not. Robot Controller use this information during the pick action. When this action is finished, robot controller checks if an object has been picked or not by reading from this topic. If an object has been picked it goes to the box to place the object and, if not, it just finishes the pick action and request AI Manager for a new action.

5.6 Algorithm and System optimization

Reaching the final version of the system and the algorithm was a really complex and progressive exercise. It is impossible to narrate here all the decisions that we had to make in the process, but I will try to explain the most importants.

5.6.1 Performance

Deep Reinforcement learning can be a really intense CPU and RAM consumer. In fact, during the first steps of the training the time between two actions were between 3 and 5 seconds. This may not look a lot, but having in mind that a Reinforcement Learning training can have more than 10000 steps, we would lose between 30000 and 50000 seconds just calculating actions (We would have to execute them later). In hours, it would be between 8 and a half hours and 13,8 hours.

We had to solve it, so we tried installing a Nvidia GPU in the computer together with the CUDA drivers. After hours of installation, the results were amazing. We designed a load test to test it and these are the results:

	CPU	GPU
raiv2	3:53 minutes	49 minutes
dl	3:14 minutes	36:31 minutes

Tab. 5.1: Difference in performance between CPU and GPU trainings

In the table we can see the time difference between CPU and GPU and between raiv2 and dl. raiv2 is the development computer and dl the execution one, because have a much better GPU and CPU, as it can be seen in the results. It was important to solve this problem, because if we couldn't, we would have to train the network once in each N steps instead of training it in every step.

Another performance problem of Deep Reinforcement Learning has to be with the Replay Memory. In replay Memory we store all the experiences of the agent in order to take smaller samples of experiences to train the algorithm. The problem comes because all the experiences are stored in the GPU memory and, having in mind that in each experience we are saving 2 images, it was impossible to store more than 2000 experiences.

2000 experiences could be enough for the training, but the highest the Replay Memory size, the better. To solve this problem, we decided to store the features extracted from the images instead of the images themselves. With this solution, we introduce a second limitation which is that we have to finish the training with the same feature extractor that we started it.

However, this isn't a project because we have this limitation anyway, it wouldn't make sense to use a different feature extractor, because the algorithm has been trained with the first feature extractor.

5.6.2 Training

One of the most difficult things in Deep Learning Algorithms is tuning the algorithm. In this case we had to tune two neural networks and the Reinforcement Learning Algorithm. The CNN was training using the images of previous trainings. We performed several days of training with a silly Neural Network. As Reinforcement Learning training it was totally useless, but we used to store and classify thousands of pick images.

The initial image state of every pick action performed was saved together with the success or failure information of the movement. With this images and the labels we trained a Neural Network to classify these images, and we used the first layers of this model as feature extractor for the Reinforcement Learning Algorithm.

Then, we had to tune the Reinforcement Learning Algorithm, but we will talk about this in the analysis and results section of this document.

5.6.3 Why to use Block Detector?

During the first steps of the training, we were doing normal training and, every time that an episode was ended, either by reaching the environment limits or by picking a piece, the robot was starting a new episode by going to a new random pair of coordinates. This looked as the best option at the beginning, because it was the simplest solution, and also because we wanted to ensure low correlation between consecutive steps and random decisions are sometimes the best way of reaching low correlation between samples.

However, during these first episodes we observed that the robot was emptying the pieces of the centre of the box, but it was more difficult for it to pick

the pieces that were near to the edges. The coordinates were chosen completely randomly, so we then realized that, although the coordinates probability distribution of the starting point of the episode was uniform, the probability of passing through the centre of the environment during an episode was actually higher than the probability of reaching the box sides.

The explanation of this is simple, when the initial point of the Episode is in the centre, the probability of reaching any of the four sides of the box is the same while, when the initial point of the environment is one of the sides, its much more complicated to reach another side of the box without passing through the centre.

To check this theory we included the coordinates of each step in the training statistics, and the results were the ones showed in [Figure 5.16](#). In this image we can see the heatmap of the movements of the robot and how the probability of reaching the centre of the box was much more higher than the probability of reaching the sides.

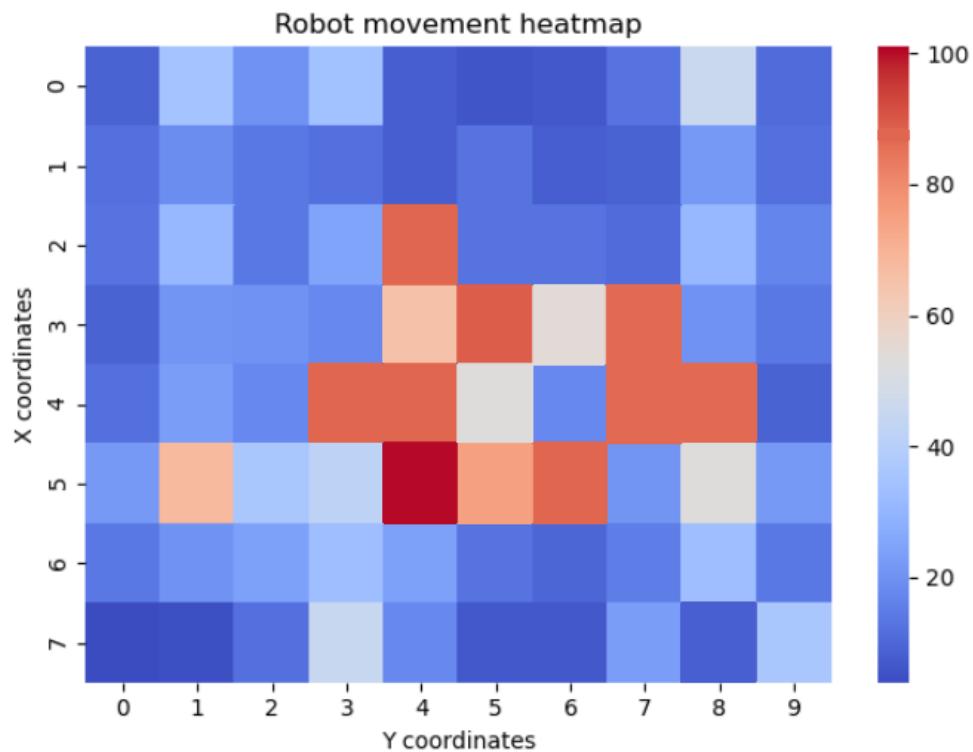


Fig. 5.16: Robot position Heatmap

In an ideal algorithm, this distribution should be uniform, because the distribution of pieces in the box is uniform. To solve this problem, we used a new strategy for calculating the initial point of the environments. In this case, we decided not to send the robot to the centre of the box, that is to say, sending it always to the sides. With this strategy, we knew that the robot was going to pass through the centre of the box, but we were not expecting the distribution of probabilities to be as uniform as it was. The results were really good, as can be seen in the Figure 5.17

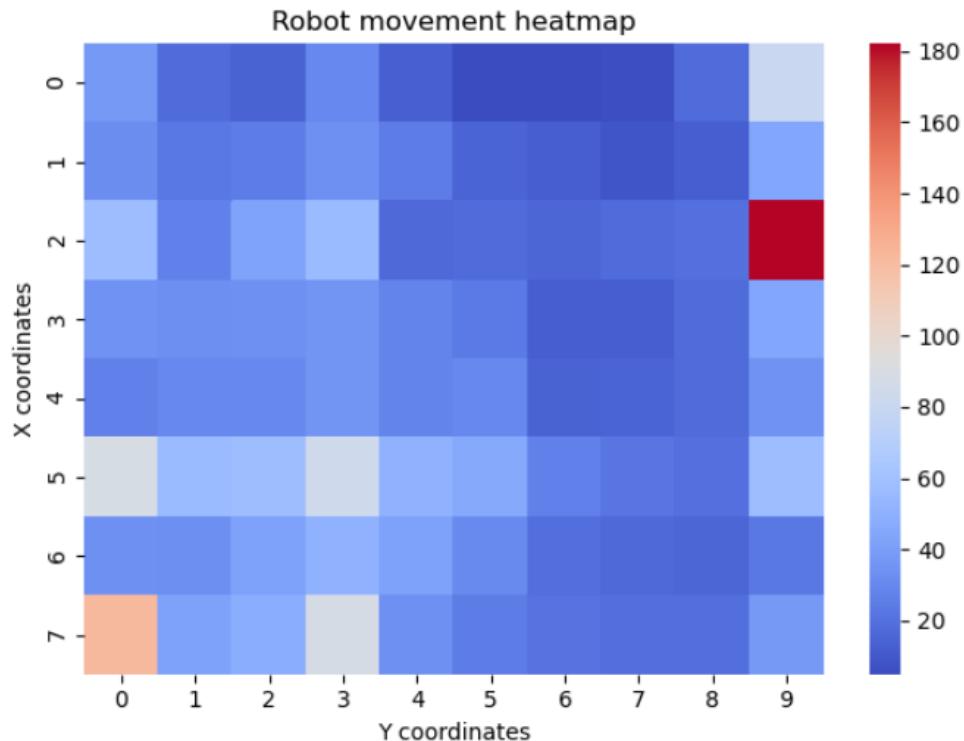


Fig. 5.17: Robot position Heatmap with the new strategy

The robot behaved better with this new strategy, because now it was managing to empty better the sides of the box. However, there was another problem that we had with both strategies. The problem was that, when the box was almost empty, as the initial position of each episode was chosen randomly, most of the times the robot was going to places with no pieces, and it was taking a lot of time to empty the full box. The robot was working correctly with the box full of pieces, but not when there were just few of them.

5.6. Algorithm and System optimization

To solve this problem, we decided to implement the Block Detector. We commented it before, so I will not explain again how it works, but the important thing is that it allowed us to move the robot to the places with highest amount of pieces, making the robot performance great with or without a high amount of pieces. The best thing is that, as the place with highest amount of pieces varies during the training, the distribution of the robot position kept constantly distributed during the whole box.

todos
los
pasos
asta
lle-
gar a
blob
de-
tec-
tor

gripper
de-
sign

6. RESULTS ANALYSIS

rewards

introducción
de
prior-
learning
para
evi-
tar
que
se
den
ac-
ciones
como
por
ejem-
plo
dos
picks
segui-
dos
en el
mismo
lugar

HAblar
de
los
obje-
tivos
ini-
ciales
y si
se
han
cumplido

cuando
el
fondo
es
blanoc
no

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

```
1 #!/usr/bin/env python
2 # coding: utf-8
3
4 """
5 - We need to connect the camera and the nodes
6 roslaunch ur_icam_description webcam.launch
7
8 - We need to establish a connection to the robot with the
9   following comand:
10 rosrun ur_robot_driver ur3_bringup.launch robot_ip
11   :=10.31.56.102 kinematics_config:=${HOME}/Calibration/
12     ur3_calibration.yaml
13
14 - Then, we ned to activate moovit server:
15 rosrun ur3_moveit_config ur3_moveit_planning_execution.launch
16
17 - Activate the talker
18 rosrun ai_manager main.py
19
20 - Finally, we can run the program
21 rosrun robot_controller arduino.py
22
23 """
24
25 import rospy
26
27 from ai_manager.srv import GetActions
28 from Robot import Robot
29
30
31 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],
37                         relative_coordinates[1], object_gripped).action
38     except rospy.ServiceException as e:
39         print("Service call failed: %s"%e)
40
41 # This function defines the movements that robot should make
42 # depending on the action listened
43 def take_action(action, robot):
44     rospy.loginfo("Action received: {}".format(action))
45     object_gripped = False
46     if action == 'north':
47         robot.take_north()
48     elif action == 'south':
49         robot.take_south()
50     elif action == 'east':
51         robot.take_east()
52     elif action == 'west':
53         robot.take_west()
54     elif action == 'pick':
55         object_gripped = robot.take_pick()
56     elif action == 'random_state':
57         robot.take_random_state()
58     return object_gripped
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
70     # of new state
71     object_gripped = take_action('random_state', robot)
72
73     while True:
74         action = get_action(robot, object_gripped)
75         object_gripped = take_action(action, robot)

```

.1.2 Robot.py

```
1 import copy
2 import rospy
3 import time
4
5 from std_msgs.msg import Bool
6 from std_msgs.msg import Float32
7
8 from ai_manager.Environment import Environment
9 from ur_icam_description.robotUR import RobotUR
10
11 """
12 Class used to establish connection with the robot and perform
13 different actions such as move in all cardinal directions
14 or pick and place an object.
15 """
16
17 class Robot:
18     def __init__(self, robot=RobotUR(), gripper_topic='
19         switch_on_off'):
20         self.robot = robot # Robot we want to control
21         self.gripper_topic = gripper_topic # Gripper topic
22         self.gripper_publisher = rospy.Publisher(self.
23             gripper_topic, Bool) # Publisher for the gripper topic
24
25     def relative_move(self, x, y, z):
26         """
27             Perform a relative move in all x, y or z coordinates.
28
29             :param x:
30             :param y:
31             :param z:
32             :return:
33             """
34
35         waypoints = []
36         wpose = self.robot.get_current_pose().pose
37         if x:
38             wpose.position.x -= x # First move up (x)
39             waypoints.append(copy.deepcopy(wpose))
40         if y:
41             wpose.position.y -= y # Second move forward/backwards
42             in (y)
43                 waypoints.append(copy.deepcopy(wpose))
44         if z:
45             wpose.position.z += z # Third move sideways (z)
46             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] -
48             relative_coordinates[0]
49         absolute_coordinates_y = Environment.CARTESIAN_CENTER[1] -
50             relative_coordinates[1]
51
52         current_pose = self.robot.get_current_pose()
53
54         x_movement = current_pose.pose.position.x -
55             absolute_coordinates_x
56         y_movement = current_pose.pose.position.y -
57             absolute_coordinates_y
58
59         return x_movement, y_movement
60
61     def calculate_current_coordinates(self):
62         absolute_coordinate_x = self.robot.get_current_pose().pose.
63             position.x
64         absolute_coordinate_y = self.robot.get_current_pose().pose.
65             position.y
66
67         relative_coordinate_x = Environment.CARTESIAN_CENTER[0] -
68             absolute_coordinate_x
69         relative_coordinate_y = Environment.CARTESIAN_CENTER[1] -
70             absolute_coordinate_y
71
72         return [relative_coordinate_x, relative_coordinate_y]
73
74     # Action north: positive x
75     def take_north(self, distance=Environment.ACTION_DISTANCE):
76         self.relative_move(distance, 0, 0)
77
78     # Action south: negative x
79     def take_south(self, distance=Environment.ACTION_DISTANCE):
80         self.relative_move(-distance, 0, 0)
81
82     # Action east: negative y
83     def take_east(self, distance=Environment.ACTION_DISTANCE):
84         self.relative_move(0, -distance, 0)
85
86     # Action west: positive y
87     def take_west(self, distance=Environment.ACTION_DISTANCE):
88         self.relative_move(0, distance, 0)
89
90     def take_random_state(self):
91         # Move robot to random positions using relative moves. Get
92         coordinates
```



```

126         plan.joint_trajectory.points[i].time_from_start =
127             plan.joint_trajectory.points[
128                 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
133                     [i].velocities[j] * new_speed)
134                     accelerations.append(plan.joint_trajectory.
135                         points[i].accelerations[j] * new_speed)
136                         positions.append(plan.joint_trajectory.points[
137                             i].positions[j])
138
139             point = plan.joint_trajectory.points[i]
140             point.velocities = velocities
141             point.accelerations = accelerations
142             point.positions = positions
143
144             points.append(point)
145
146     new_plan.joint_trajectory.points = points
147
148     return new_plan
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163

```

```

164         communication_problem boolean flag is set to True. It
165         is considered that there is a problem with
166             communications when the robot is not receiving any
167             distance messages during 200 milli-seconds (timeout=0.2)
168
169             :param robot: robot_controller.Robot.py object
170             :return: communication_problem flag
171             """
172
173             distance_ok = rospy.wait_for_message('distance', Bool)
174             .data # We retrieve sensor distance
175             communication_problem = False
176
177             if not distance_ok: # If the robot is already in
178                 contact with an object, no movement is performed
179                     waypoints = []
180                     wpose = robot.robot.get_current_pose().pose
181                     wpose.position.z -= (wpose.position.z) # Third
182                     move sideways (z)
183                     waypoints.append(copy.deepcopy(wpose))
184
185                     (plan, fraction) = robot.robot.move_group.
186                     compute_cartesian_path(
187                         waypoints, # waypoints to follow
188                         0.01, # eef_step
189                         0.0) # jump_threshold
190
191                     plan = change_plan_speed(plan, movement_speed)
192                     robot.robot.move_group.execute(plan, wait=False)
193
194                     while not distance_ok:
195                         try:
196                             distance_ok = rospy.wait_for_message('
197                             distance', Bool, 0.2).data # We retrieve sensor distance
198                         except:
199                             communication_problem = True
200                             rospy.loginfo("Error in communications,
201                             trying again")
202                             break
203
204                         # Both stop and 10 mm up movement to stop the
205                         robot
206                         robot.robot.move_group.stop()
207                         robot.relative_move(0, 0, 0.001)
208
209                         return communication_problem
210
211             communication_problem = True
212             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
242     # episode should finish now
243     self.robot.go_to_joint_state(Environment.ANGULAR_CENTER)
244
245     def go_to_initial_pose(self):
246         target_reached = self.robot.go_to_joint_state(Environment.
247 ANGULAR_CENTER)
248
249         if target_reached:
250             print("Target reached")
251         else:
252             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
4     using Reinforcement Learning and Image Recognition.
5 This code does not perform actions directly into the robot, it
6     just posts actions in a ROS topic and
7 gathers state information from another ROS topic.
8 """
9
10
11 import rospy
12 import torch
13
14
15 from ai_manager.srv import GetActions, GetActionsResponse
16 from RLAlgorithm import RLAlgorithm
17 from Environment import Environment
18
19 rospy.init_node('ai_manager', anonymous=True) # ROS node
20     initialization
```

```

16 # Global Image Controller
17 RL_ALGORITHM = RLAlgorithm.recover_training(batch_size=256, lr
18 =0.0001,
19                                     others='
20                                     optimal_original_rewards_algorithm1901',
21                                     include_pick_prediction=False)
22                                     # others =
23                                     optimal_original_rewards_new_model')
24
25 def handle_get_actions(req):
26     """
27     Callback for each Request from the Robot
28     :param req: Robot requests has 3 elements: object_gripped, x
29     and y elements
30     :return:
31     """
32     object_gripped = req.object_gripped
33     current_coordinates = [req.x, req.y]
34     # Next action is calculated from the current state
35     action = RL_ALGORITHM.next_training_step(current_coordinates,
36     object_gripped)
37
38     # RL_ALGORITHM.plot()
39
40     return GetActionsResponse(action)
41
42
43 def get_actions_server():
44     """
45     Service initialization to receive requests of actions from the
46     robot.
47     Each time that a request is received, handle_get_actions
48     function will be called
49     :return:
50     """
51
52     s = rospy.Service('get_actions', GetActions,
53     handle_get_actions)
54     rospy.loginfo("Ready to send actions.")
55     rospy.spin()
56     rospy.on_shutdown(save_training)
57
58
59 def save_training():
60     RL_ALGORITHM.save_training()
61
62
63 if __name__ == '__main__':
64     try:
65         get_actions_server()

```

```

56     except rospy.ROSInterruptException:
57         pass

```

.2.2 RLAlgorithm.py

```

1 # coding=utf-8
2 import math
3 import random
4 import os
5 import errno
6 import sys
7 from collections import namedtuple
8
9 import matplotlib
10 import matplotlib.pyplot as plt
11 import rospy
12 import torch
13 import torch.nn as nn
14 import torch.nn.functional as F
15 import torch.optim as optim
16 import torchvision.transforms as T
17 from PIL import Image
18
19 from Environment import Environment
20 from TrainingStatistics import TrainingStatistics
21 from ImageProcessing.ImageModel import ImageModel
22 from ImageController import ImageController
23
24 is_ipython = 'inline' in matplotlib.get_backend()
25 if is_ipython: from IPython import display
26
27 import pickle
28
29 State = namedtuple( # State information namedtuple
30     'State',
31     ('coordinate_x', 'coordinate_y', 'pick_probability',
32      'object_gripped', 'image_raw')
33 )
34
35 Experience = namedtuple( # Replay Memory Experience namedtuple
36     'Experience',
37     ('state', 'coordinates', 'pick_probability', 'action',
38      'next_state', 'next_coordinates', 'next_pick_probability',
39      'reward', 'is_final_state')
40 )
41 class RLAlgorithm:

```

```

42     """
43     Class used to perform actions related to the RL Algorithm
44     training. It can be initialized with custom parameters or
45     with the default ones.
46
47     To perform a Deep Reinforcement Learning training , the
48     following steps have to be followed:
49
50         1. Initialize replay memory capacity.
51         2. Initialize the policy network with random weights.
52         3. Clone the policy network, and call it the target
53             network.
54
55         4. For each episode:
56             1. Initialize the starting state.
57             2. For each time step:
58                 1. Select an action.
59                     - Via exploration or exploitation
60                 2. Execute selected action in an emulator or in
61                     Real-life.
62                     3. Observe reward and next state.
63                     4. Store experience in replay memory.
64                     5. Sample random batch from replay memory.
65                     6. Preprocess states from batch.
66                     7. Pass batch of preprocessed states to policy
67                         network.
68                     8. Calculate loss between output Q-values and
69                         target Q-values.
70                         - Requires a pass to the target network for
71                         the next state
72                     9. Gradient descent updates weights in the policy
73                         network to minimize loss.
74                         - After time steps, weights in the target
75                         network are updated to the weights in the policy
76                         network.
77
78     """

```

```

68
69     def __init__(self, object_gripped_reward=10,
70                  object_not_picked_reward=-10, out_of_limits_reward=-10,
71                  horizontal_movement_reward=-1, batch_size=32,
72                  gamma=0.999, eps_start=1, eps_end=0.01, eps_decay=0.0005,
73                  target_update=10, memory_size=100000, lr=0.001,
74                  num_episodes=1000, include_pick_prediction=False,
75                  save_training_others='optimal'):
76
77     """
78
79         :param object_gripped_reward: Object gripped reward
80         :param object_not_picked_reward: Object not picked reward
81         :param out_of_limits_reward: Out of limits reward
82         :param horizontal_movement_reward: Horizontal movement

```

```

reward
    :param batch_size: Size of the batch used to train the
network in every step
    :param gamma: discount factor used in the Bellman equation
    :param eps_start: Greedy strategy epsilon start (
Probability of random choice)
    :param eps_end: Greedy strategy minimum epsilon (
Probability of random choice)
    :param eps_decay: Greedy strategy epsilon decay (
Probability decay of random choice)
    :param target_update: How frequently, in terms of episodes
, target network will update the weights with the
policy network weights
    :param memory_size: Capacity of the replay memory
    :param lr: Learning rate of the Deep Learning algorithm
    :param num_episodes: Number of episodes on training
    :param include_pick_prediction: Use the image model pick
prediction as input of the DQN
    :param self_training_others: Parameter used to modify the
filename of the training while saving
    """
self.batch_size = batch_size
self.gamma = gamma
self.eps_start = eps_start
self.eps_end = eps_end
self.eps_decay = eps_decay
self.target_update = target_update
self.memory_size = memory_size
self.lr = lr
self.num_episodes = num_episodes
self.include_pick_prediction = include_pick_prediction
self.save_training_others = save_training_others

self.current_state = None # Robot current state
self.previous_state = None # Robot previous state
self.current_action = None # Robot current action
self.current_action_idx = None # Robot current action
Index
    self.episode_done = False # True if the episode has just
ended

# This tells PyTorch to use a GPU if its available,
otherwise use the CPU
    self.device = torch.device("cuda" if torch.cuda.
is_available() else "cpu") # Torch devide
    self.em = self.EnvManager(self, object_gripped_reward,
object_not_picked_reward, out_of_limits_reward,
horizontal_movement_reward) #

```

```

Robot Environment Manager
115     self.strategy = self.EpsilonGreedyStrategy(self.eps_start,
116         self.eps_end, self.eps_decay) # Greede Strategy
117     self.agent = self.Agent(self) # RL Agent
118     self.memory = self.ReplayMemory(self.memory_size) #
Replay Memory
119     self.statistics = TrainingStatistics() # Training
statistics

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

```

```

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
153             the information of the current state
154             :param policy_net: DQN object used as policy network
155                 for the RL algorithm
156                 :return:
157                 """
158                 random_action = False
159                 if self.rl_algorithm.episode_done: # If the episode
160                     has just ended we reset the robot environment
161                         self.rl_algorithm.episode_done = False # Put the
162                         variable episode_done back to False
163                             self.rl_algorithm.statistics.new_episode()
164
165                         self.rl_algorithm.current_action = 'random_state'
166                         # Return random_state to reset the robot position
167                         self.rl_algorithm.current_action_idx = None
168                         else:
169                             rate = self.strategy.get_exploration_rate(
170                                 self.rl_algorithm.statistics.current_step) #
171 We get the current epsilon value
172
173         if rate > random.random(): # With a probability =
174             rate we choose a random action (Explore environment)
175                 action = random.randrange(self.num_actions)
176                 random_action = True
177                 else: # With a probability = (1 - rate) we
178                     Exploit the information we already have
179                         try:
180                             with torch.no_grad(): # We calculate the
181                             action using the Policy Q Network
182                                 action = policy_net(state.image_raw,
183                                     torch.tensor(
184                                         [state.coordinate_x, state.
185                                         coordinate_y]), device=self.device),
186                                         state.
187                                         pick_probability, self.rl_algorithm.include_pick_prediction)\ \
188                                         .argmax(dim=1).to(self.device) #
189                                         exploit
190                                         except:
191                                             print("Ha habido un error")
192
193                                         self.rl_algorithm.current_action = self.
194                                         rl_algorithm.em.actions[action]
195                                         self.rl_algorithm.current_action_idx = action
196
197                                         return self.rl_algorithm.current_action, random_action
198                                         # 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,
191                      include_pick_prediction):
192             """
193                 :param image_tensor_size: Size of the input tensor
194                 :param num_actions: Number of actions, which is the
195                     output of the Neural Network
196             """
197
198             super(RLAlgorithm.DQN, self).__init__()
199
200             self.linear1 = nn.Linear(image_tensor_size, int(
201                 image_tensor_size / 2))
202             self.linear2 = nn.Linear(int(image_tensor_size / 2),
203                 int(image_tensor_size / 4))
204             extra_features = 2 # coordinates
205             if include_pick_prediction:
206                 extra_features = 3 # pick prediction
207             self.linear3 = nn.Linear(int(image_tensor_size / 4) +
208                 extra_features, num_actions)
209             self.linear = nn.Linear(image_tensor_size + 2,
210                 num_actions)
211
212             # Called with either one element to determine next action,
213             # or a batch
214             # during optimization. Returns tensor([[left0exp,right0exp]
215             # ...]).
216             def forward(self, image_raw, coordinates, pick_probability =
217                         None, include_pick_probability=False):
218
219                 output = self.linear1(image_raw)
220                 output = self.linear2(output)
221                 if include_pick_probability:
222                     output = torch.cat((output, coordinates,
223                         pick_probability), 1)
224                 else:
225                     output = torch.cat((output, coordinates), 1)
226                 return self.linear3(output)
227
228             class EnvManager:
229                 """
230                     Class used to manage the RL environment. It is used to
231                     perform actions such as calculate rewards or retrieve the
232                     current state of the robot.

```

```

222     """
223
224     def __init__(self, rl_algorithm, object_gripped_reward,
225      object_not_picked_reward, out_of_limits_reward,
226      horizontal_movement_reward):
227         """
228         Initialization of an object
229         :param rl_manager: RLAlgorithm object
230         :param object_gripped_reward: Object gripped reward
231         :param object_not_picked_reward: Object not picked
232         reward
233         :param out_of_limits_reward: Out of limits reward
234         :param horizontal_movement_reward: Horizontal movement
235         reward
236         """
237         self.object_gripped_reward = object_gripped_reward
238         self.out_of_limits_reward = out_of_limits_reward
239         self.object_not_picked_reward =
240         object_not_picked_reward
241         self.horizontal_movement_reward =
242         horizontal_movement_reward
243
244         self.device = rl_algorithm.device # Torch device
245         self.image_controller = ImageController() # ImageController object to manage images
246         self.actions = ['north', 'south', 'east', 'west', 'pick'] # Possible actions of the objects
247         self.image_height = None # Retrieved images height
248         self.image_width = None # Retrieved Images Width
249         self.image = None # Current image ROS message
250         self.image_tensor = None # Current image tensor
251         self.pick_probability = None # Current image tensor
252
253         self.model_name = 'model-epoch=05-val_loss=0.36-
254         weights7y3_unfreeze2.ckpt'
255         # self.model_name = 'resnet50_freezed.ckpt'
256         self.model_family = 'resnet50'
257         self.image_model = ImageModel(model_name=self.
model_family)
258         self.feature_extraction_model = self.image_model.
load_model(self.model_name)
259         self.image_tensor_size = self.image_model.
get_size_features(
260             self.feature_extraction_model) # Size of the
image after performing some transformations
261
262         self.rl_algorithm = rl_algorithm
263         self.gather_image_state() # Retrieve initial state
264         image

```

```

258     def calculate_reward(self, previous_image):
259         """
260             Method used to calculate the reward of the previous
261             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
265             .coordinate_x,
266                             self.rl_algorithm.current_state
267             .coordinate_y] # Retrieve robot's current coordinates
268             object_gripped = self.rl_algorithm.current_state.
269             object_gripped # Retrieve if the robot has an object gripped
270             if Environment.is_terminal_state(current_coordinates,
271             object_gripped): # If is a terminal state
272                 self.rl_algorithm.episode_done = True # Set the
273                 episode_done variable to True to end up the episode
274                 episode_done = True
275                 if object_gripped: # If object_gripped is True,
276                     the episode has ended successfully
277                     reward = self.object_gripped_reward
278                     self.rl_algorithm.statistics.
279                     add_successful_episode(True) # Saving episode successful
280                     statistic
281                     self.rl_algorithm.statistics.increment_picks()
282                     # Increase of the statistics cputer
283                     rospy.loginfo("Episode ended: Object gripped!")
284             )
285             self.image_controller.record_image(
286             previous_image, True) # Saving the failure state image
287             else: # Otherwise the robot has reached the
288             limits of the environment
289                 reward = self.out_of_limits_reward
290                 self.rl_algorithm.statistics.
291                 add_successful_episode(False) # Saving episode failure
292                 statistic
293                 rospy.loginfo("Episode ended: Environment
294                 limits reached!")
295             else: # If it is not a Terminal State
296                 episode_done = False
297                 if self.rl_algorithm.current_action == 'pick': #
298                 if it is not the first action and action is pick
299                     reward = self.object_not_picked_reward
300                     self.image_controller.record_image(
301                     previous_image, False) # Saving the failure state image
302                     self.rl_algorithm.statistics.increment_picks()
303                     # Increase of the statistics counter
304                     else: # otherwise
305                         self.rl_algorithm.statistics.

```

```

288     fill_coordinates_matrix(current_coordinates)
289         reward = self.horizontal_movement_reward
290
290     self.rl_algorithm.statistics.add_reward(reward) # Add
291     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
296             retrieving an image using the image_controller class,
297             which reads the image from the ROS topic specified.
297         """
298
298     previous_image = self.image
299     self.image, self.image_width, self.image_height = self
300     .image_controller.get_image() # We retrieve state image
300     self.image_tensor, pick_probability = self.
301     extract_image_features(self.image)
301     if self.rl_algorithm.include_pick_prediction:
302         self.pick_probability = pick_probability
303
304     return previous_image
305
306     def extract_image_features(self, image):
307         """
308             Method used to transform the image to extract image
308             features by passing it through the image_model CNN
309             network
310             :param image_raw: Image
311             :return:
311         """
312
313     features, pick_prediction = self.image_model.
313     evaluate_image(image, self.feature_extraction_model)
314     features = torch.from_numpy(features)
315     return features.to(self.device), torch.tensor([[math.
315     exp(pick_prediction.numpy()[0][1])]]).to(self.device)
316
317     def num_actions_available(self):
318         """
319             Returns the number of actions available
320             :return: Number of actions available
321         """
322
322     return len(self.actions)
323
324     class EpsilonGreedyStrategy:
325         """
326             Class used to perform the Epsilon greede strategy
327         """
328

```

```

329     def __init__(self, start, end, decay):
330         """
331             Initialization
332             :param start: Greedy strategy epsilon start (
333                 Probability of random choice)
334             :param end: Greedy strategy minimum epsilon (
335                 Probability of random choice)
336             :param decay: Greedy strategy epsilon decay (
337                 Probability decay of random choice)
338             """
339             self.start = start
340             self.end = end
341             self.decay = decay
342
343     def get_exploration_rate(self, current_step):
344         """
345             It calculates the rate depending on the actual step of
346             the execution
347             :param current_step: step of the training
348             :return:
349             """
350             return self.end + (self.start - self.end) * \
351                 math.exp(-1. * current_step * self.decay)
352
353     class QValues:
354         """
355             It returns the predicted q-values from the policy_net for
356             the specific state-action pairs that were passed in.
357             states and actions are the state-action pairs that were
358             sampled from replay memory.
359             """
360
361         @staticmethod
362         def get_current(policy_net, states, coordinates, actions,
363                         pick_probabilities, include_pick_prediction = False):
364             """
365                 With the current state of the policy network, it
366                 calculates the q_values of
367                 :param policy_net: policy network used to decide the
368                     actions
369                     :param states: Set of state images (Preprocessed)
370                     :param coordinates: Set of robot coordinates
371                     :param actions: Set of taken actions
372                     :return:
373                     """
374             return policy_net(states, coordinates,
375                               pick_probabilities, include_pick_prediction).gather(dim=1,
376                               index=actions.unsqueeze(-1))
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2298
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2398
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2497
2497
2498
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2539
2540
2541
2542

```

```

367     @staticmethod
368     def get_next(target_net, next_states, next_coordinates,
369                  next_pick_probabilities, is_final_state,
370                  include_pick_prediction = False):
371         """
372             Calculate the maximum q-value predicted by the
373             target_net among all possible next actions.
374             If the action has led to a terminal state, next reward
375             will be 0. If not, it is calculated using the target
376             net
377             :param target_net: Target Deep Q Network
378             :param next_states: Next states images
379             :param next_coordinates: Next states coordinates
380             :param is_final_state: Tensor indicating whether this
381             action has led to a final state or not.
382             :return:
383             """
384             batch_size = next_states.shape[0] # The batch size is
385             taken from next_states shape
386             # q_values is initialized with a zeros tensor of
387             batch_size and if there is GPU it is loaded to it
388             q_values = torch.zeros(batch_size).to(torch.device(""
389             cuda" if torch.cuda.is_available() else "cpu"))
390             non_final_state_locations = (is_final_state == False)
391             # Non final state index locations are calculated
392             non_final_states = next_states[
393                 non_final_state_locations] # non final state images
394             non_final_coordinates = next_coordinates[
395                 non_final_state_locations] # non final coordinates
396             if include_pick_prediction:
397                 non_final_pick_probabilities =
398                     next_pick_probabilities[non_final_state_locations] # non final
399                     pick probabilities
400             else:
401                 non_final_pick_probabilities = None
402                 # Max q values of the non final states are calculated
403                 # using the target net
404                 q_values[non_final_state_locations] = \
405                     target_net(non_final_states, non_final_coordinates
406                     , non_final_pick_probabilities, include_pick_prediction).max(
407                     dim=1)[
408                         0].detach()
409             return q_values
410
411     class ReplayMemory:
412         """
413             Class used to create a Replay Memory for the RL algorithm
414             """
415

```

```

400     def __init__(self, capacity):
401         """
402             Initialization of ReplayMemory
403             :param capacity: Capacity of Replay Memory
404         """
405         self.capacity = capacity
406         self.memory = [] # Actual memory. it will be filled
407         with Experience namedtuples
408         self.push_count = 0 # will be used to keep track of
409         how many experiences have been added to the memory
410
411     def push(self, experience):
412         """
413             Method used to fill the Replay Memory with experiences
414             :param experience: Experience namedtuple
415             :return:
416         """
417         if len(self.memory) < self.capacity: # if memory is
418             not full, new experience is appended
419             self.memory.append(experience)
420         else: # If its full, we add a new experience and take
421             the oldest out
422             self.memory[self.push_count % self.capacity] =
423             experience
424             self.push_count += 1 # we increase the memory counter
425
426     def sample(self, batch_size):
427         """
428             Returns a random sample of experiences
429             :param batch_size: Number of randomly sampled
430             experiences returned
431             :return: random sample of experiences (Experience
432             namedtuples)
433         """
434         return random.sample(self.memory, batch_size)
435
436     def can_provide_sample(self, batch_size):
437         """
438             returns a boolean telling whether or not we can sample
439             from memory. Recall that the size of a sample
440             we'll obtain from memory will be equal to the batch
441             size we use to train our network.
442             :param batch_size: Batch size to train the network
443             :return: boolean telling whether or not we can sample
444             from memory
445         """
446         return len(self.memory) >= batch_size
447
448     def extract_tensors(self, experiences, include_pick_prediction

```

```

= False):
    """
    Converts a batch of Experiences to Experience of batches
    and returns all the elements separately.
    :param experiences: Batch of Experience objects
    :return: A tuple of each element of a Experience
    namedtuple
    """
    batch = Experience(*zip(*experiences))

    states = torch.cat(batch.state)
    actions = torch.cat(batch.action)
    rewards = torch.cat(batch.reward)
    next_states = torch.cat(batch.next_state)
    coordinates = torch.cat(batch.coordinates)
    next_coordinates = torch.cat(batch.next_coordinates)
    if include_pick_prediction:
        pick_probabilities = torch.cat(batch.pick_probability)
        next_pick_probabilities = torch.cat(batch.
    next_pick_probability)
    else:
        pick_probabilities = None
        next_pick_probabilities = None
        is_final_state = torch.cat(batch.is_final_state)

    return states, coordinates, pick_probabilities, actions,
rewards, next_states, next_coordinates, \
        next_pick_probabilities, is_final_state

    @staticmethod
    def saving_name(batch_size, gamma, eps_start, eps_end,
eps_decay, lr, others=''):
        return 'bs{}_g{}_es{}_ee{}_ed{}_lr_{}_{}.pkl'.format(
            batch_size, gamma, eps_start, eps_end, eps_decay, lr,
others
        )

    def save_training(self, dir='trainings/', others='optimal'):

        filename = self.saving_name(self.batch_size, self.gamma,
self.eps_start, self.eps_end, self.eps_decay, self.lr,
                           self.save_training_others)

    def create_if_not_exist(filename, dir):
        current_path = os.path.dirname(os.path.realpath(
__file__))
        filename = os.path.join(current_path, dir, filename)
        if not os.path.exists(os.path.dirname(filename)):
            try:

```

```

479             os.makedirs(os.path.dirname(filename))
480         except OSError as exc: # Guard against race
481             if exc.errno != errno.EEXIST:
482                 raise
483         return filename
484
485     rospy.loginfo("Saving training...")
486
487     abs_filename = create_if_not_exist(filename, dir)
488
489     self.em.image_model = None
490     self.em.feature_extraction_model = None
491
492     with open(abs_filename, 'wb+') as output: # Overwrites
any existing file.
493         pickle.dump(self, output, pickle.HIGHEST_PROTOCOL)
494
495     rospy.loginfo("Saving Statistics...")
496     print(filename)
497
498     filename = 'trainings/{}_stats.pkl'.format(filename.split(
' .pkl')[0])
499     self.statistics.save(filename=filename)
500
501     rospy.loginfo("Training saved!")
502
503     @staticmethod
504     def recover_training(batch_size=32, gamma=0.999, eps_start=1,
eps_end=0.01,
505                           eps_decay=0.0005, lr=0.001,
506                           include_pick_prediction=False, others='optimal', dir='trainings
/',
):
507         current_path = os.path.dirname(os.path.realpath(__file__))
508         filename = RLAlgorithm.saving_name(batch_size, gamma,
509         eps_start, eps_end, eps_decay, lr, others)
510         filename = os.path.join(current_path, dir, filename)
511         try:
512             with open(filename, 'rb') as input:
513                 rl_algorithm = pickle.load(input)
514                 rospy.loginfo("Training recovered. Next step will
be step number {}"
515                               .format(rl_algorithm.statistics.
current_step))
516
517                 rl_algorithm.em.image_model = ImageModel(
model_name=rl_algorithm.em.model_family)
518                 rl_algorithm.em.feature_extraction_model =
519                 rl_algorithm.em.image_model.load_model(

```

```

517                         rl_algorithm.em.model_name)
518
519                     return rl_algorithm
520             except IOError:
521                 rospy.loginfo("There is no Training saved. New object
522 has been created")
523                 return RLAlgorithm(batch_size=batch_size, gamma=gamma,
524 eps_start=eps_start, eps_end=eps_end,
525                             eps_decay=eps_decay, lr=lr,
526 include_pick_prediction=include_pick_prediction,
527 save_training_others=others)
528
529     def train_net(self):
530         """
531             Method used to train both the train and target Deep Q
532 Networks. We train the network minimizing the loss between
533             the current Q-values of the action-state tuples and the
534 target Q-values. Target Q-values are calculated using
535             the Bellman's equation:
536
537             q*(state, action) = Reward + gamma * max( q*(next_state,
538 next_action) )
539             :return:
540             """
541
542             # If there are at least as much experiences stored as the
543 batch size
544             if self.memory.can_provide_sample(self.batch_size):
545                 experiences = self.memory.sample(self.batch_size) #
546 Retrieve the experiences
547                 # We split the batch of experience into different
548 tensors
549                 states, coordinates, pick_probabilities, actions,
550 rewards, next_states, next_coordinates, \
551                     next_pick_probabilities, is_final_state = self.
552 extract_tensors(experiences, self.include_pick_prediction)
553                 # To compute the loss, current_q_values and
554 target_q_values have to be calculated
555                 current_q_values = self.QValues.get_current(self.
556 policy_net, states, coordinates, actions,
557
558                 pick_probabilities, self.include_pick_prediction)
559                 # next_q_values is the maximum Q-value of each future
560 state
561                 next_q_values = self.QValues.get_next(self.target_net,
562 next_states, next_coordinates,
563
564                 next_pick_probabilities, is_final_state, self.
565 include_pick_prediction)
566                 target_q_values = (next_q_values * self.gamma) +

```

```

    rewards

547         loss = F.mse_loss(current_q_values, target_q_values.
548                         unsqueeze(1)) # Loss is calculated
549         self.optimizer.zero_grad() # set all the gradients to
550         0 (initialization) so that we don't accumulate
551         # gradient throughout all the backpropagation
552         loss.backward(
553             retain_graph=True) # Compute the gradient of the
554             loss with respect to all the weights and biases in the
555             # policy net
556             self.optimizer.step() # Updates the weights and
557             biases with the gradients computed

558     if self.statistics.episode % self.target_update == 0: #
559     If target_net has to be updated in this episode
560         self.target_net.load_state_dict(self.policy_net.
561             state_dict()) # Target net is updated

562     def next_training_step(self, current_coordinates,
563     object_gripped):
564         """
565             This method implements the Reinforcement Learning
566             algorithm to control the UR3 robot. As the algorithm is
567             prepared
568                 to be executed in real life, rewards and final states
569                 cannot be received until the action is finished, which is the
570                 beginning of next loop. Therefore, during an execution of
571                 this function, an action will be calculated and the
572                 previous action, its reward and its final state will be
573                 stored in the replay memory.
574
575                 :param current_coordinates: Tuple of float indicating
576                 current coordinates of the robot
577                 :param object_gripped: Boolean indicating whether or not
578                 an object has been gripped
579                 :return: action taken
580
581         """
582         self.statistics.new_step() # Add new steps statistics
583         self.previous_state = self.current_state # Previous state
584         information to store in the Replay Memory
585         previous_action = self.current_action # Previous action
586         to store in the Replay Memory
587         previous_action_idx = self.current_action_idx # Previous
588         action index to store in the Replay Memory
589         previous_image = self.em.gather_image_state() # Gathers
590         current state image

591
592         self.current_state = State(current_coordinates[0],
593             current_coordinates[1], self.em.pick_probability,

```

```

576                                     object_gripped, self.em.
577
578     image_tensor) # Updates current_state
579
580         # Calculates previous action reward an establish whether
581         # the current state is terminal or not
582         previous_reward, is_final_state = self.em.calculate_reward(
583         previous_image)
584         action, random_action = self.agent.select_action(self.
585         current_state,
586                                     self.
587         policy_net) # Calculates action
588
589         # There are some defined rules that the next action have
590         # to accomplish depending on the previous action
591         action_ok = False
592         while not action_ok:
593             # Its forbidden to perform two cosecutive pick actions
594             # in the same place
595             if action == 'pick' and previous_action != 'pick':
596                 action_ok = True
597             # If previous action was south, it is forbidden to
598             # perform a 'north' action for
599             # The robot not to go back to the original position.
600             elif action == 'north' and previous_action != 'south':
601                 action_ok = True
602             # If previous action was north, it is forbidden to
603             # perform a 'south' action for
604             # The robot not to go back to the original position.
605             elif action == 'south' and previous_action != 'north':
606                 action_ok = True
607             # If previous action was east, it is forbidden to
608             # perform a 'west' action for
609             # The robot not to go back to the original position.
610             elif action == 'west' and previous_action != 'east':
611                 action_ok = True
612             # If previous action was west, it is forbidden to
613             # perform a 'east' action for
614             # The robot not to go back to the original position.
615             elif action == 'east' and previous_action != 'west':
616                 action_ok = True
617             elif action == 'random_state':
618                 action_ok = True
619             else:
620                 action, random_action = self.agent.select_action(
621                 self.current_state,
622
623                 self.policy_net) # Calculates action
624
625         if random_action:

```

```

612         self.statistics.random_action() # Recolecting
613         statistics
614
615             # Random_state actions are used just to initialize the
616             environment to a random position, so it is not taken into
617             # account while storing state information in the Replay
618             Memory.
619             # If previous action was a random_state and it is not the
620             first step of the training
621             if previous_action != 'random_state' and self.statistics.
622             current_step > 1:
623                 self.memory.push( # Pushing experience to Replay
624                 Memory
625                     Experience( # Using an Experience namedtuple
626                         self.previous_state.image_raw, # Initial
627                         state image
628                             torch.tensor([[self.previous_state.
629                             coordinate_x, self.previous_state.coordinate_y]],
630                                         device=self.device), # Initial
631                                         coordinates
632                                         self.previous_state.pick_probability,
633                                         torch.tensor([previous_action_idx], device=
634                                         self.device), # Action taken
635                                         self.current_state.image_raw, # Final state
636                                         image
637                                         torch.tensor([[self.current_state.coordinate_x
638                                         ,
639                                         self.current_state.coordinate_y
640                                         ]],
641                                         device=self.device), # Final
642                                         coordinates
643                                         self.current_state.pick_probability,
644                                         torch.tensor([previous_reward], device=self.
645                                         device), # Action reward
646                                         torch.tensor([is_final_state], device=self.
647                                         device) # Episode ended
648                                         ))
649
650             # Logging information
651             rospy.loginfo("Step: {}, Episode: {}, Previous reward:
652             {}, Previous action: {}".format(
653                 self.statistics.current_step - 1,
654                 self.statistics.episode,
655                 previous_reward,
656                 previous_action))
657
658             self.train_net() # Both policy and target networks
659             gets trained
660
661

```

643 **return** action

.2.3 Environment.py

```

1 """
2 This class defines a RL environment for a pick and place task with
3 a UR3 robot.
4 This environment is defined by its center (both cartesian and
5 angular coordinates), the total length of its x and y axis
6 and other parameters
7 """
8
9 import random
10 from BlobDetector.BlobDetector import BlobDetector
11 from ai_manager.ImageController import ImageController
12 from math import floor
13
14 class Environment:
15     X_LENGTH = 0.175 # Total length of the x axis environment in
16     # meters
17     Y_LENGTH = 0.225 # Total length of the y axis environment in
18     # meters
19
20     CAMERA_SECURITY_MARGIN = 0.035 # As the camera is really
21     close to the gripping point, it needs a security margin
22     X_LIMIT = X_LENGTH - CAMERA_SECURITY_MARGIN # Robot
23     boundaries of movement in axis X
24     Y_LIMIT = Y_LENGTH - CAMERA_SECURITY_MARGIN # Robot
25     boundaries of movement in axis Y
26
27     CARTESIAN_CENTER = [-0.31899288568, -0.00357907370787,
28     0.376611799631] # Cartesian center of the RL environment
29     ANGULAR_CENTER = [2.7776150703430176, -1.5684941450702112,
30     1.299912452697754, -1.3755658308612269,
31     -1.5422008673297327, -0.3250663916217249] #
32     Angular center of the RL environment
33     PLACE_CARTESIAN_CENTER = [0, 0.25, CARTESIAN_CENTER[2]] ##
34     # Cartesian center of the place box
35     ANGULAR_PICTURE_PLACE = [1.615200161933899,
36     -1.235102955495016, 0.739865779876709, -1.2438910643206995,
37     -1.5095704237567347, -0.06187755266298467]
38
39     PICK_DISTANCE = 0.01 # Distance to the object when the robot
40     is performing the pick and place action
41     ACTION_DISTANCE = 0.02 # Distance to the object when the
42     robot is performing the pick and place action

```

```

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
36             Environment defined.
37             To help the robot empty the box, the generated coordinates
38             won't be in the center of the box, because this is
39             the most reachable place of the box.
40
41             :param strategy: strategy used to calculate random_state
42             coordinates
43             :return:
44             """
45     def generate_random_coordinates():
46         coordinate_x = random.uniform((-Environment.X_LIMIT +
47                                         Environment.ENV_BOUNDS_TOLERANCE) / 2,
48                                         (Environment.X_LIMIT -
49                                         Environment.ENV_BOUNDS_TOLERANCE) / 2)
49         coordinate_y = random.uniform((-Environment.Y_LIMIT +
50                                         Environment.ENV_BOUNDS_TOLERANCE) / 2,
51                                         (Environment.Y_LIMIT -
52                                         Environment.ENV_BOUNDS_TOLERANCE) / 2)
52         return coordinate_x, coordinate_y
53
54         # Random coordinates avoiding the ones in the center,
55         # which have a bigger probability of being reached by the
56         # robot.
57         if strategy == 'ncc' or strategy == 'non_centered_coordinates':
58             coordinates_in_center = True
59             while coordinates_in_center:
60                 coordinate_x, coordinate_y =
61                 generate_random_coordinates()
62                 if abs(coordinate_x) > (Environment.X_LIMIT / 4)
63                 or abs(coordinate_y) > (Environment.Y_LIMIT / 4):
64                     coordinates_in_center = False
65                 elif strategy == 'optimal' and image is not None: #
66                     Before going to a random state, we check that there are pieces
67                     in this place
68                     blob_detector = BlobDetector(x_length=Environment.
69                         X_LENGTH, y_length=Environment.Y_LENGTH, columns=4, rows=4)
70                     optimal_quadrant = blob_detector.find_optimal_quadrant
71                     (image)
72                     optimal_point = blob_detector.quadrants_center[
73                     optimal_quadrant]
74
75                     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 =
66         generate_random_coordinates()
67
68     return [coordinate_x, coordinate_y]
69
70     @staticmethod
71     def get_relative_corner(corner):
72         """
73             Function used to calculate the coordinates of the
74             environment corners relative to the CARTESIAN_CENTER.
75
76             :param corner: it indicates the corner that we want to get
77             the coordinates. It's composed by two letters
78             that indicate the cardinality. For example: ne indicates
79             North-East corner
80             :return coordinate_x, coordinate_y:
81             """
82
83         if corner == 'sw' or corner == 'ws':
84             return -Environment.X_LIMIT / 2, Environment.Y_LIMIT /
85
86         if corner == 'nw' or corner == 'wn':
87             return Environment.X_LIMIT / 2, Environment.Y_LIMIT /
88
89         if corner == 'ne' or corner == 'en':
90             return Environment.X_LIMIT / 2, -Environment.Y_LIMIT /
91
92         if corner == 'se' or corner == 'es':
93             return -Environment.X_LIMIT / 2, -Environment.Y_LIMIT /
94
95
96     @staticmethod
97     def is_terminal_state(coordinates, object_gripped):
98         """
99             Function used to determine if the current state of the
100            robot is terminal or not
101            :return: bool
102            """
103
104         def get_limits(length): return length / 2 - Environment.
105             ENV_BOUNDS_TOLERANCE # function to calculate the box boundaries
106             x_limit_reached = abs(coordinates[0]) > get_limits(
107                 Environment.X_LIMIT) # x boundary reached
108             y_limit_reached = abs(coordinates[1]) > get_limits(
109                 Environment.Y_LIMIT) # y boundary reached
110             return x_limit_reached or y_limit_reached or
111             object_gripped # If one or both or the boundaries are reached
112             --> 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(
17         __file__)), image_topic='/usb_cam/image_raw'):
18         self.ind_saved_images = 0 # Index which will tell us the
19         number of images that have been saved
20         self.success_path = "{}/success".format(path) # Path
21         where the images are going to be saved
22         self.fail_path = "{}/fail".format(path) # Path where the
23         images are going to be saved
24         self.image_topic = image_topic
25
26         # If it does not exist, we create the path folder in our
27         workspace
28         try:
29             os.stat(self.success_path)
30         except:
31             os.mkdir(self.success_path)
32
33         # If it does not exist, we create the path folder in our
34         workspace
35         try:
36             os.stat(self.fail_path)
37         except:
38             os.mkdir(self.fail_path)
39
40     def get_image(self):
41         msg = rospy.wait_for_message(self.image_topic, Image)
42
43         return self.to_pil(msg), msg.width, msg.height
44
45     def record_image(self, img, success):
46         path = self.success_path if success else self.fail_path # The path were we want to save the image is

```

```
41     image_path = '{}/img{}.png'.format( # Saving image
42         path, # Path
43         time.time()) # FIFO queue
44
45     img.save(image_path)
46
47     self.ind_saved_images += 1 # Index increment
48
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) #
53         sensor_msg to Image
54         return img
55
56 if __name__ == '__main__':
57     rospy.init_node('image_recorder') # ROS node initialization
58     image_controller = ImageController(path='/home/pilar/Pilar/
59     ros_pictures', image_topic='/usb_cam2/image_raw')
      img, width, height = image_controller.get_image()
      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](https://arxiv.org/abs/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](https://doi.org/10.1109/ICRA.2017.7989381).
- [5] *Reinforcement Learning - Goal Oriented Intelligence*. URL: https://deeplizard.com/learn/playlist/PLZbbT5o_s2xoWNVdDudn51XM8l0uZ_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](https://arxiv.org/abs/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](https://doi.org/10.1155/2018/5781591). URL: <https://www.hindawi.com/journals/jr/2018/5781591/> (visited on 11/30/2020).

Bibliography

- [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).