



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

1. <i>Introduction</i>	1
1.1 Project Motivation	3
2. <i>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
3. <i>State of the art</i>	12
3.0.1 Reinforcement Learning	13
3.0.2 Deep Reinforcement Learning	16
3.0.3 Problems of Deep Reinforcement Learning in Real-world	24

4. <i>Definition of the Project</i>	26
4.1 Motivation	26
4.2 Objectives	26
4.3 Methodology	27
4.4 Planning and budget	28
4.4.1 Budget	28
5. <i>Developed System</i>	31
5.1 Hardware Architecture	32
5.2 Logical Architecture	34
5.3 ai_manager	41
5.3.1 Definition of the problem	42
5.3.2 Environment.py	42
5.3.3 Rewards	43
5.3.4 Algorithm	43
6. <i>Results Analysis</i>	48
7. <i>Conclusions and Future Work</i>	49
<i>Bibliography</i>	50

LIST OF FIGURES

1.1	Pick and Place Task	2
3.1	Fanuc DNN	13
3.2	Q-Matrix	15
3.3	Deep Q Learning	17
3.4	RL Training	23
4.1	Methodology	28
4.2	Chronograph HW	28
4.3	Chronograph HW	29
4.4	Planing Robot Controller	29
5.1	pieces	31
5.2	Picture of Architecture I	32
5.3	Picture of Architecture II	33
5.4	Logical Architecture	35
5.5	Architecture I	36
5.6	Architecture II	37
5.7	Architecture III	38
5.8	Architecture IV	40

5.9 Node Interaction	41
--------------------------------	----

LIST OF TABLES

4.1 Project's Budget	30
--------------------------------	----

1. INTRODUCTION

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

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

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

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

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

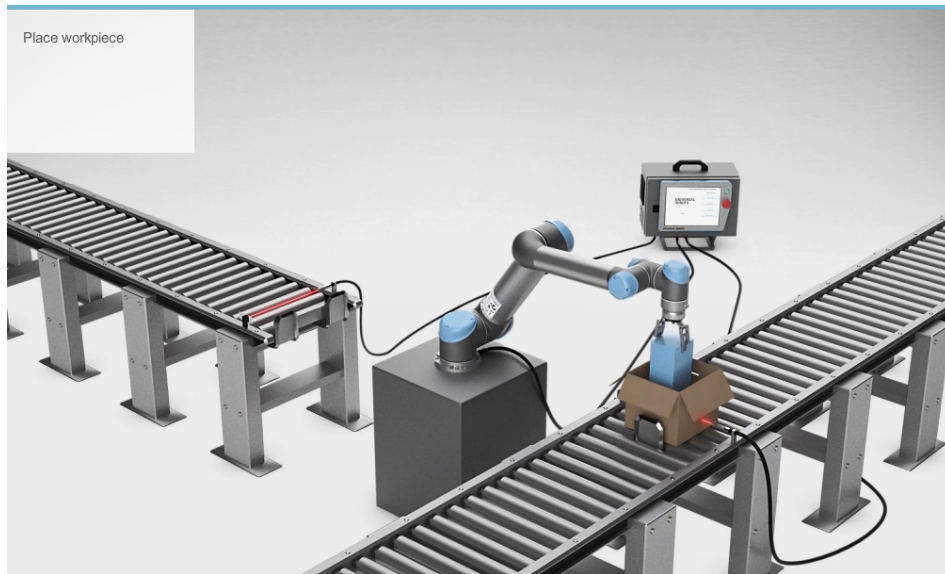


Fig. 1.1: Universal Robots Pick and Place Task

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

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

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

project adding all the advantages of a microservices oriented architecture.

1.1 Project Motivation

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

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

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

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

2. DESCRIPTION OF TECHNOLOGIES

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

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 questions 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 control ROS remotely.
- ROS is a multi-node oriented framework, which allows us to take all the advantages of micro-services. We can split the software by functionality gaining:
 - The possibility of giving more or less computation power to each functionality depending on its needs. In the project we have used 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 separately 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 order to work. We needed the arduino card in order to build our "home made" vacuum 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 camera connected to another node or computer.

PYTORCH

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.

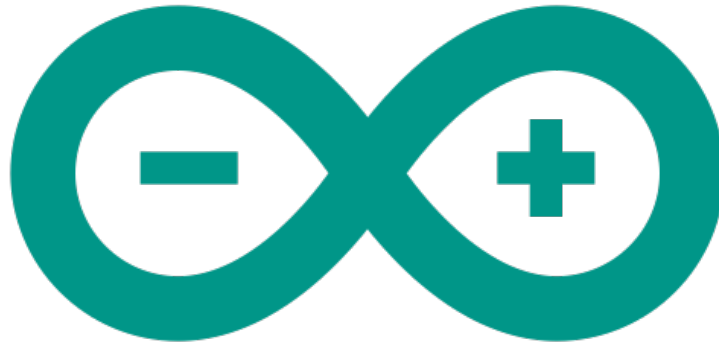
PyTorch has a very simple interface for creating neural networks despite working directly with tensors without the need for a library at a higher level such as Keras for Theano or Tensorflow.

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*

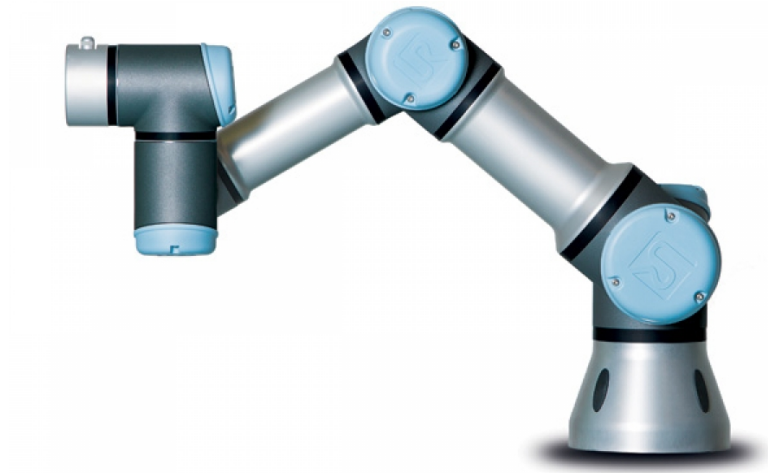


Universal Robots have become a dominant supplier of lightweight, robotic manipulators for industry, as well as for scientific research and education. The Robot Operating System (ROS) has developed from a community-centered movement to a mature framework and quasi standard, providing a rich set of powerful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

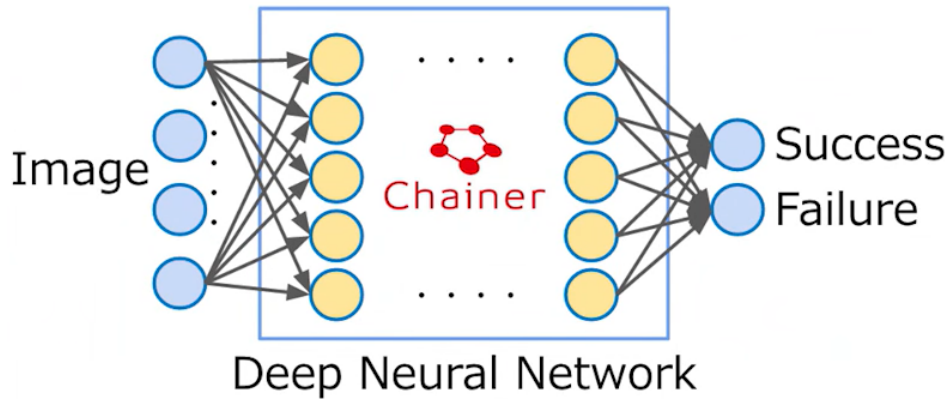


Fig. 3.1: Deep Neural Network of Fanuc solution (taken from the video)

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 th 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** an is calculated as shown in the formula below, where γ is the discount factor, and is used to give a bigger value to the closest steps.

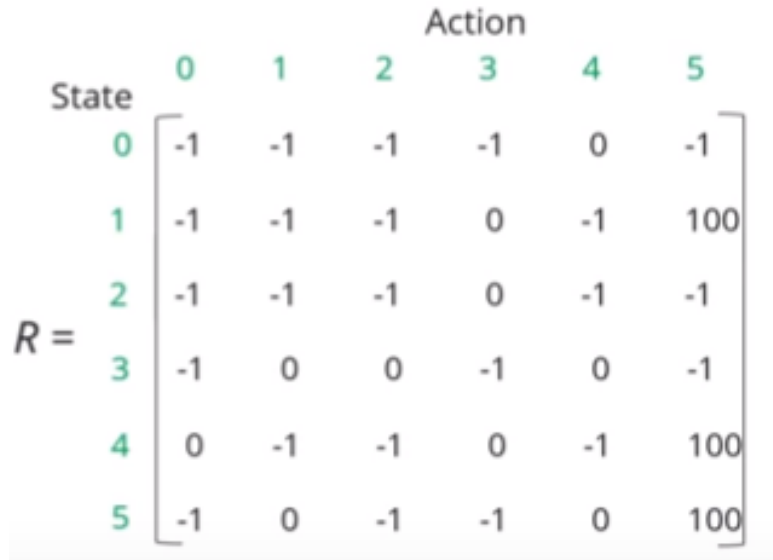
$$G_t = \sum_{k=t} \gamma^{k-t} \cdot R_{k+1} \quad \forall \gamma \in [0, 1]$$

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 $\text{length}(\text{actions}) \times \text{length}(\text{states})$ as shown in the [Figure 3.2](#).



		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

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

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

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

3.0.2 *Deep Reinforcement Learning*

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

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

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

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

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

- Episodes and Steps

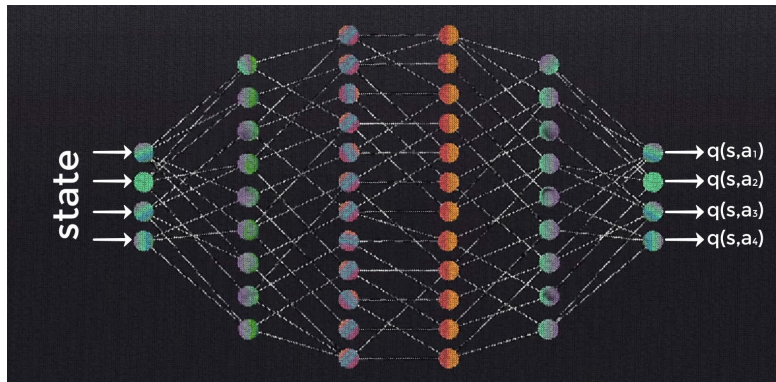


Fig. 3.3: Deep Q Learning Representation with 4 outputs

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

Episodes and Steps

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

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

Exploration vs Exploitation trade-off

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

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

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

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

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

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 hihger 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 calculates 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.

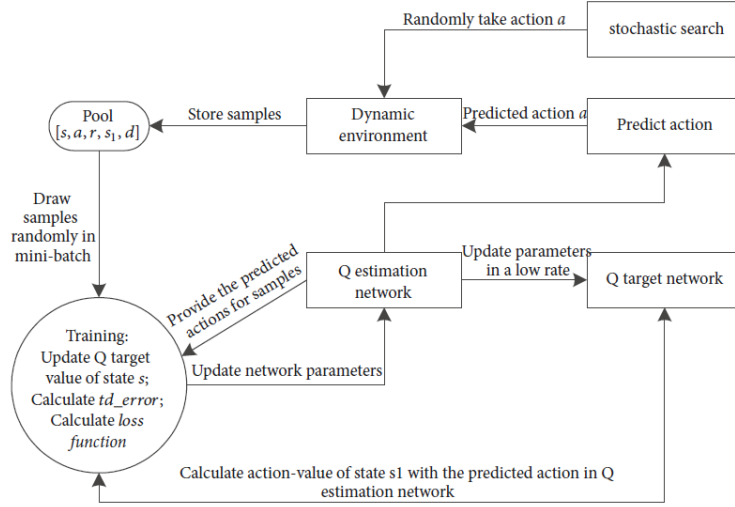


Fig. 3.4: Reinforcement Learning Training Summary

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

Some of the problems they found were the following:

- Slow rate of data-collection, as movements in the real robot are slower than in a simulated environment.
- Partial observability. Sensors cannot retrieve all the information about the environment.
- Noisy sensors will provide inaccurate information.
- Safety of the robot and its surroundings have to be taken in mind.
- Fragility of robot components.
- Delay between an action is requested and the time it is actually performed can affect the training.
- Preparing the robot is a really difficult task:
 - Controlling the robot.
 - Define all aspects of the environment.
 - Difficulties for obtaining random and independent state when episode ends.

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

There are multiple examples of articles on this topic, such as the one Xiaoyun Lei, Zhian Zhang, and Peifang Dong published in September 2018 using a DDQN

approach to solve it [8]. However, there are other solutions as the one proposed by Marco A. Wiering [9], where he introduces some prior knowledge to the model in order to facilitate the learning. His algorithm had problems generalizing the environment, so he introduced some prior information about the model together with a Model-based RL. This made the algorithm more capable to learn without losing 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.

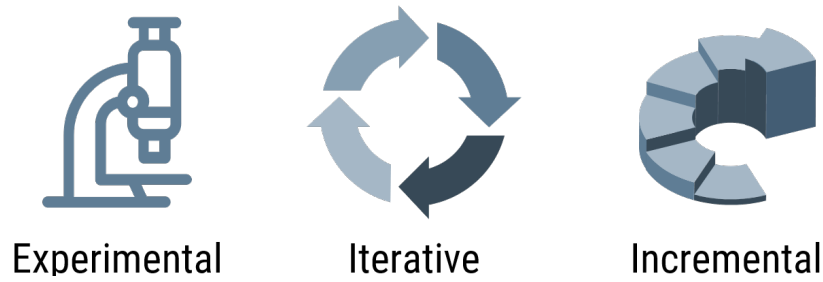


Fig. 4.1: Methodology

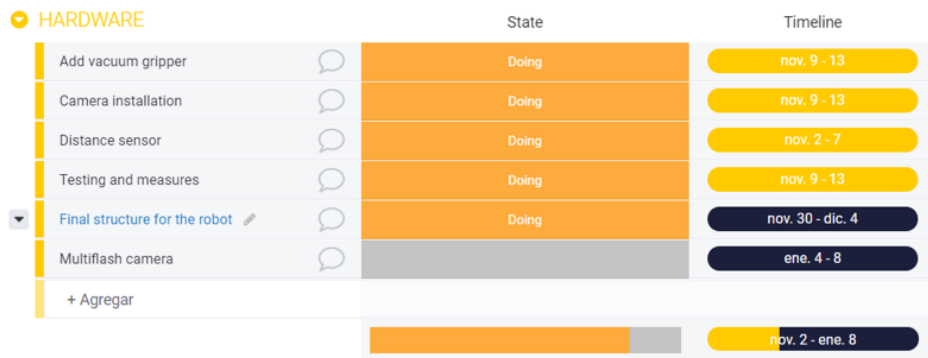


Fig. 4.2: Chronograph of the Hardware implementation

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:

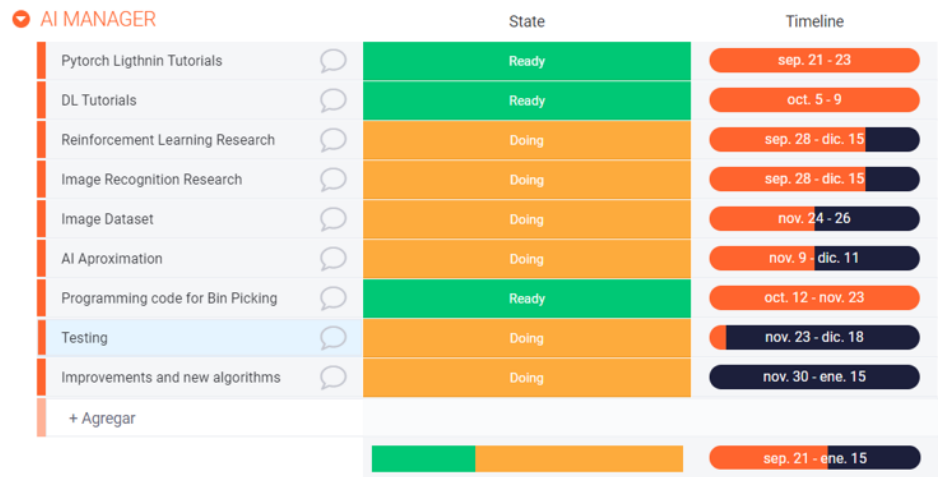


Fig. 4.3: Chronograph of the Hardward implementation

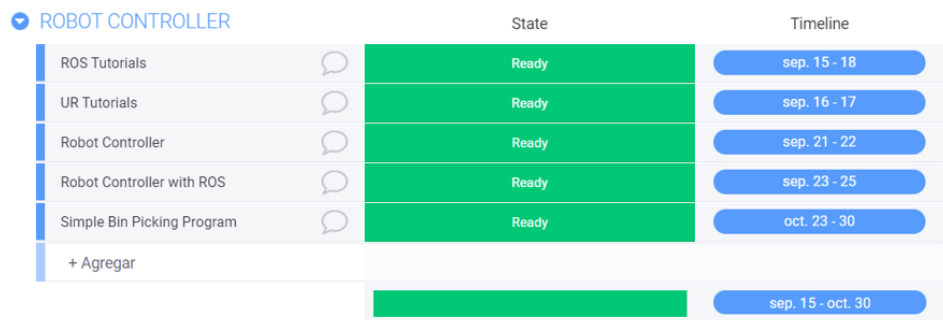


Fig. 4.4: Planning of the Robot Controller implementation

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- \rightarrow 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.

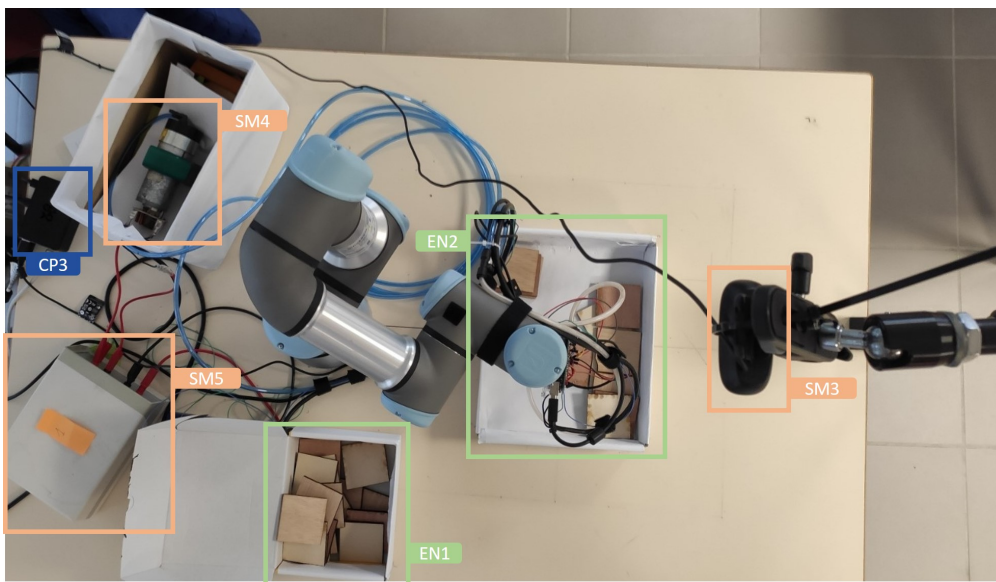


Fig. 5.2: Labelled picture of the Architecture I

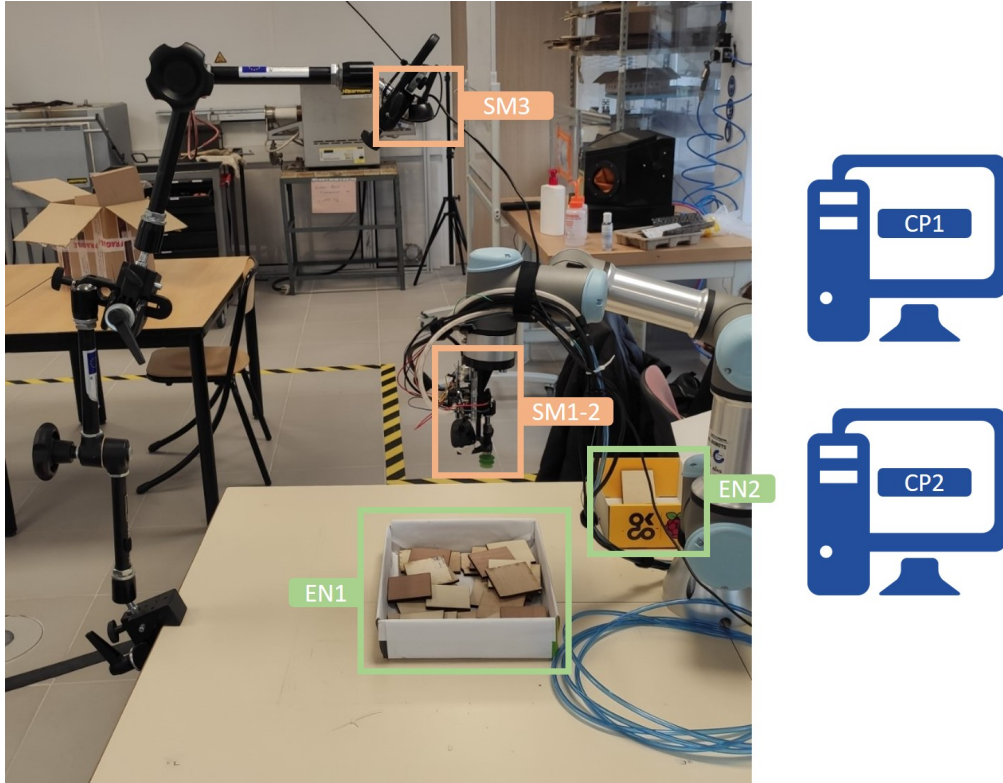


Fig. 5.3: Labelled picture of the Architecture II

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

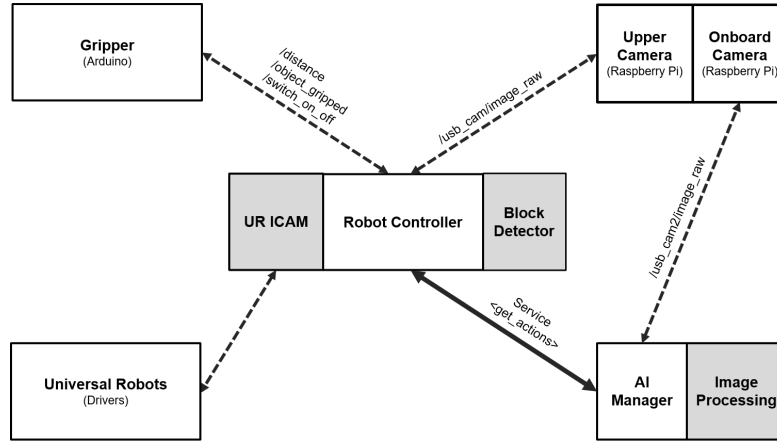


Fig. 5.4: Logical Architecture of the project

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.

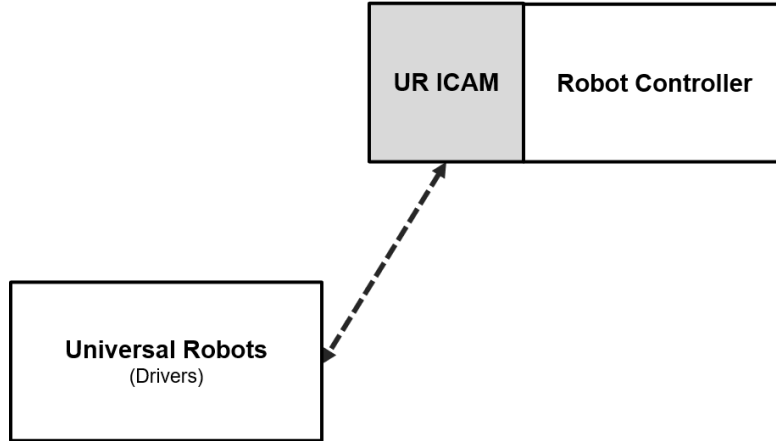


Fig. 5.5: From the simplest to the final Architecture I

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.

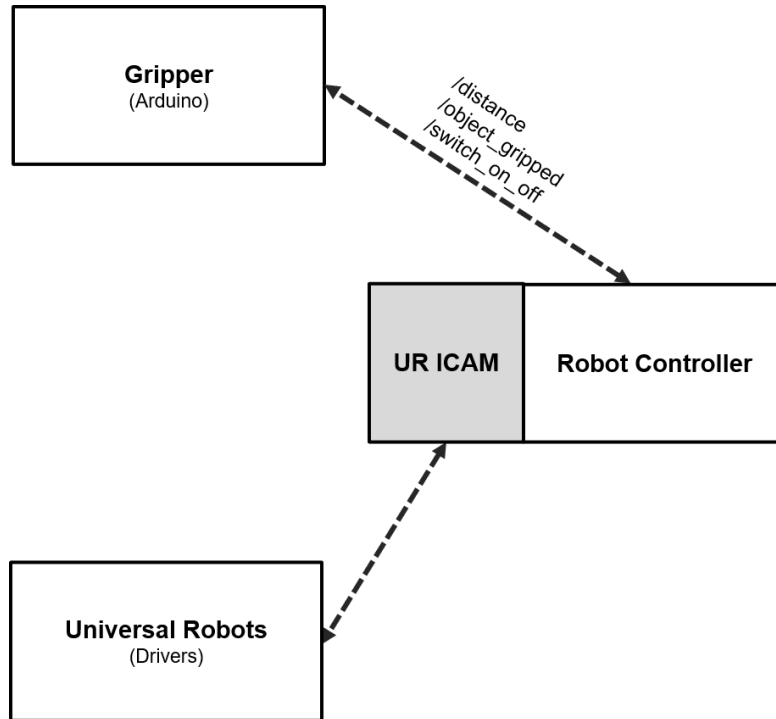


Fig. 5.6: From the simplest to the final Architecture II

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:

- **\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.

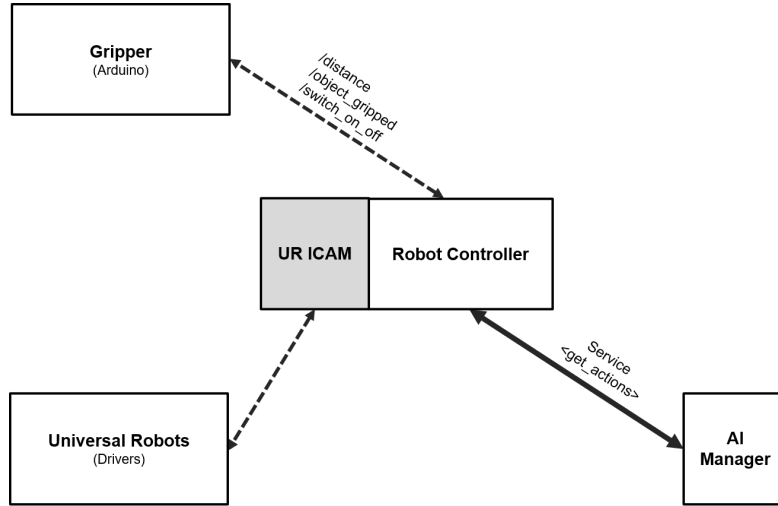


Fig. 5.7: From the simplest to the final Architecture III

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

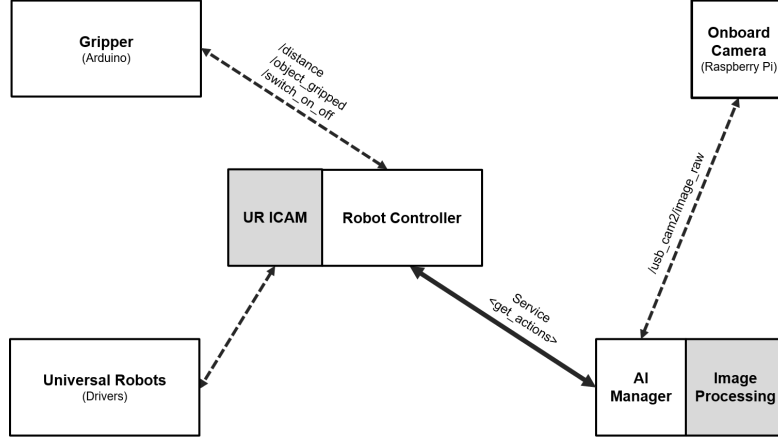


Fig. 5.8: From the simplest to the final Architecture IV

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. This 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.

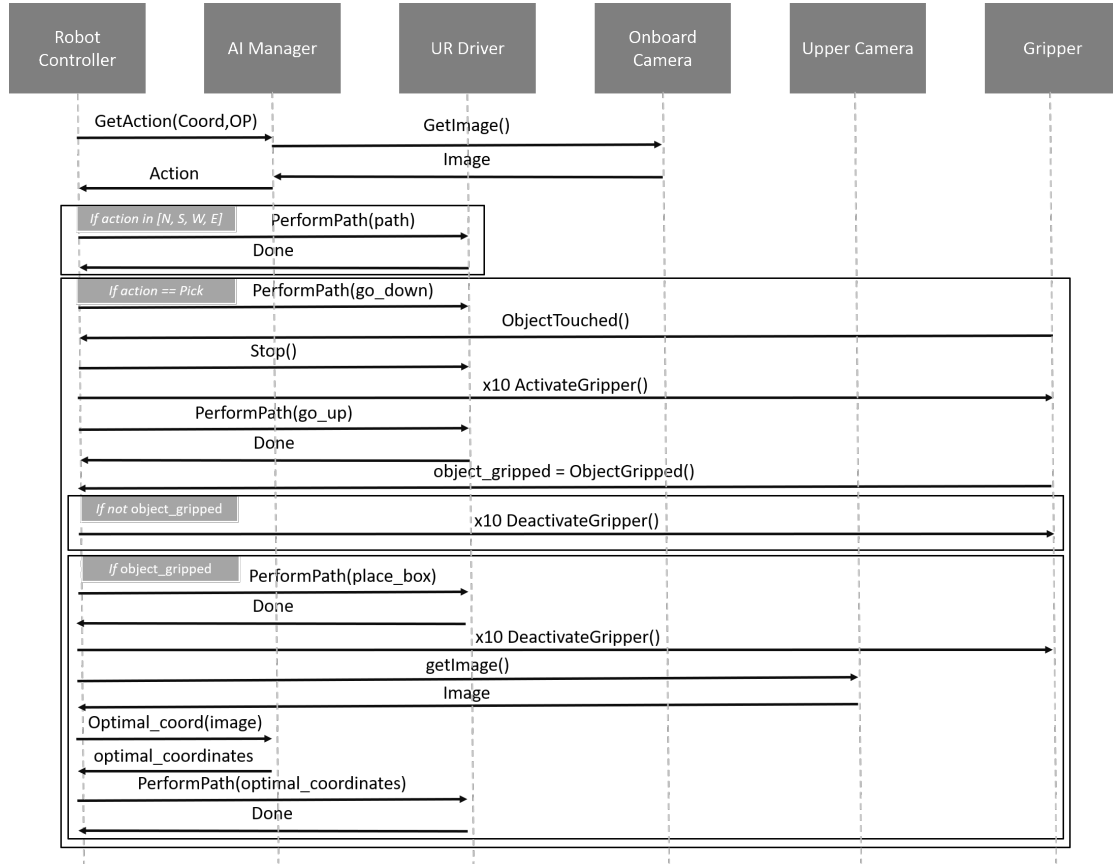


Fig. 5.9: Flow Chart of nodes interacting

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 it 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.

5.3.1 *Definition of the problem*

The objective of the agent is thus the first thing that has to be defined. It 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 have one robot to explore inside the box. To define these 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 `RQAlgorithm.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 they are 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.

6. RESULTS ANALYSIS

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

7. CONCLUSIONS AND FUTURE WORK

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

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_s2xoWNVdDudn51XM8lOuZ_Njv (visited on 11/28/2020).
- [6] OpenAI. *Gym: A toolkit for developing and comparing reinforcement learning algorithms*. URL: <https://gym.openai.com> (visited on 11/30/2020).
- [7] A. Rupam Mahmood et al. “Setting up a Reinforcement Learning Task with a Real-World Robot”. In: *arXiv:1803.07067 [cs, stat]* (Mar. 19, 2018). arXiv: [1803.07067](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).

- [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).