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

1. *Introduction* 1

1.1 Project Motivation 3

2. *Description of Technologies* 4

3. *State of the art* 5

3.0.1 Reinforcement Learning 6

3.0.2 Deep Reinforcement Learning 9

3.0.3 Problems of Deep Reinforcement Learning in Real-world . . 16

4. *Definition of the Project* 18

4.1 Motivation 18

4.2 Objectives 18

4.3 Methodology 19

4.4 Planning and budget 20

5. *Developed System* 22

5.1 Análisis del sistema 22

5.2 Diseño 22

5.3 Implementación 22

6. Results Analysis 23

7. Conclusions and Future Work 24

Appendix 25

.1 Robot Controller 26

.1.1 main.py 26

.1.2 Robot.py 28

.2 Artificial Intelligence Manager 34

.2.1 main.py 34

.2.2 RLAlgorithm.py 36

.2.3 Environment.py 54

.2.4 ImageController.py 57

Bibliography 59

LIST OF FIGURES

1.1 Pick and Place Task 2

3.1 Fanuc DNN 6

3.2 Q-Matrix 8

3.3 Deep Q Learning 10

4.1 Methodology 20

4.2 Chronograph HW 20

4.3 Planning AI 21

4.4 Planing Robot Controller 21

LIST OF TABLES

1. INTRODUCTION

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

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

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

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

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

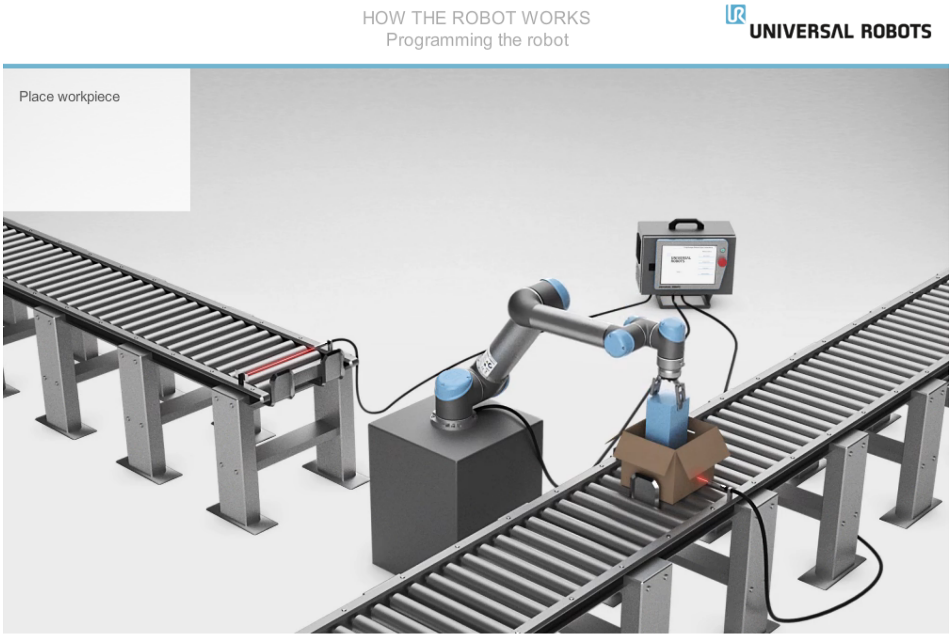

Fig. 1.1: Universal Robots Pick and Place Task

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

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

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

1.1 Project Motivation

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

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

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

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

2. DESCRIPTION OF TECHNOLOGIES

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

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

3. STATE OF THE ART

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

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

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

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

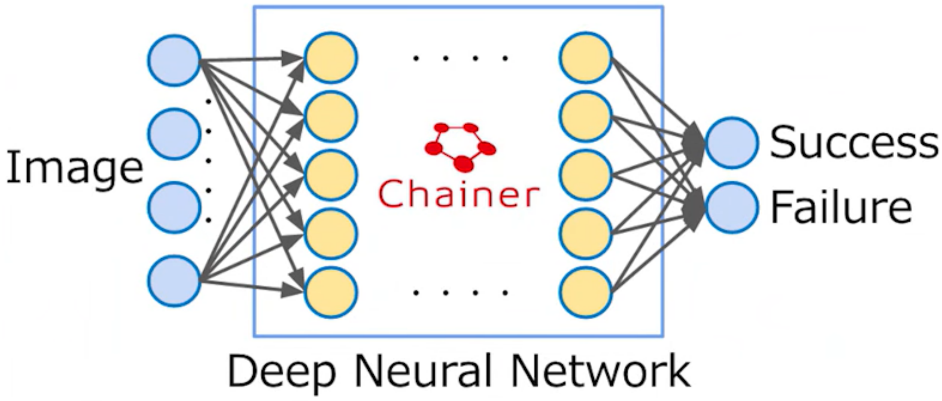

Fig. 3.1: Deep Neural Network of Fanuc solution (taken from the video)

3.0.1 Reinforcement Learning

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

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

Markov Decision Process (MD)

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

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

- **States (S):** Each time the agent performs an action, it moves to a new state. States are basically the set of information that differentiates the situation of the agent before and after performing an action. States can be transitional or terminal, when the agent meets the objective or when it gets to a forbidden position.
- **Rewards (R):** Each time an action is performed, the agent receives a reward. This reward can be positive, negative or null depending on the impact of the action to achieve the objective.
- **Policy (π):** The policy is used to define the optimal action for each step. It gives a punctuation for all of the actions in the current step as shown in the following formula. The agent takes the action with highest punctuation.

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

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

Once the action is taken:

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

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

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

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

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.

$$\begin{array}{c} \\ \text{State} \\ \\ \\ R = \\ \\ \\ \end{array}
\begin{array}{c} \\ \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\left[\begin{array}{cccccc}
-1 & -1 & -1 & -1 & 0 & -1 \\
-1 & -1 & -1 & 0 & -1 & 100 \\
-1 & -1 & -1 & 0 & -1 & -1 \\
-1 & 0 & 0 & -1 & 0 & -1 \\
0 & -1 & -1 & 0 & -1 & 100 \\
-1 & 0 & -1 & -1 & 0 & 100
\end{array}\right] \end{array}$$

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

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

However, for the aim of this project, the states of the agent can be different in each timestep. The state would actually be partially formed by images, so the number of states can be infinite. We need a more complex version of Reinforcement Learning.

3.0.2 Deep Reinforcement Learning

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

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

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

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

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

– Episodes and Steps

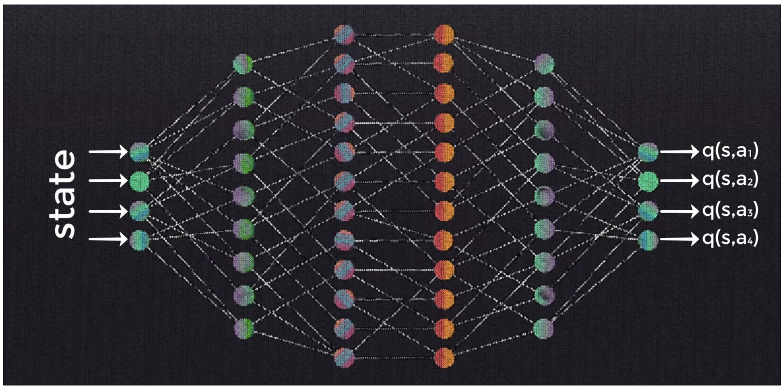

Fig. 3.3: Deep Q Learning Representation with 4 outputs

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

Episodes and Steps

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

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

Exploration vs Exploitation trade-off

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

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

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

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

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

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?

Let's see what could happen if we calculated both of the values with the same network. If we used the same network, we would calculate the loss of the neural network as the difference between two consecutive

We do the first pass to calculate the Q-value for the relevant action, and then we do a second pass in order to calculate the target Q-value for this same action. Our objective is to get the Q-value to approximate the target Q-value.

Remember, we don't know ahead of time what the target Q-value even is, so we attempt to approximate it with the network. This second pass occurs using the same weights in the network as the first pass.

Given this, when our weights update, our outputted Q-values will update, but so will our target Q-values since the targets are calculated using the same weights. So, our Q-values will be updated with each iteration to move closer to the target Q-values, but the target Q-values will also be moving in the same direction.

As Andong put it in the comments of the last video, this makes the optimization appear to be chasing its own tail, which introduces instability. As our Q-values move closer and closer to their targets, the targets continue to move further and further because we're using the same network to calculate both of these values.

Well, here's a perfect time to introduce the second network that we mentioned earlier. Rather than doing a second pass to the policy network to calculate the target Q-values, we instead obtain the target Q-values from a completely separate network, appropriately called the target network.

The target network is a clone of the policy network. Its weights are frozen with the original policy network's weights, and we update the weights in the target network to the policy network's new weights every certain amount of time steps. This certain amount of time steps can be looked at as yet another hyperparameter that we'll have to test out to see what works best for us.

So now, the first pass still occurs with the policy network. The second pass, however, for the following state occurs with the target network. With this target network, we're able to obtain the Q-value for the next state, and again, plug this value into the Bellman equation in order to calculate the target Q-value for the first state.

This is all we use the target network for — To find the value of this term so that we can calculate the target Q-value for any state passed to the policy network. As it turns out, this removes much of the instability introduced by using only one network to calculate both the Q-values, as well as the target Q-values. We now have something fixed, i.e. fixed Q-targets, that we want our policy network to approximate. So, we no longer have the dog-chasing-it's-tail problem.

As mentioned though, these values don't stay completely fixed the entire time. After amount of time steps, we'll update the weights in the target network with the weights from our policy network, which will in turn, update the target Q-values with respect to what it's learned over those past time steps. This will cause the policy network to start to approximate the updated targets.

3.0.3 Problems of Deep Reinforcement Learning in Real-world

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

Some of the problems they found were the following:

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

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

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

4. DEFINITION OF THE PROJECT

4.1 Motivation

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

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

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

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

4.2 Objectives

Revisar objetivos y adaptarlos al proyecto

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

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

Intelligence.

– Improve the performance using RL and Image Recognition.

– Study the usage of new technologies to add information to the system. Adding physics or new image recognition techniques.

– Test different tools such as Multiflash or 3D images to improve performance.

– Add value to the scientific community making the solution available if the previous objectives are reached.

4.3 Methodology

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

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

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

Fig. 4.1: Methodology

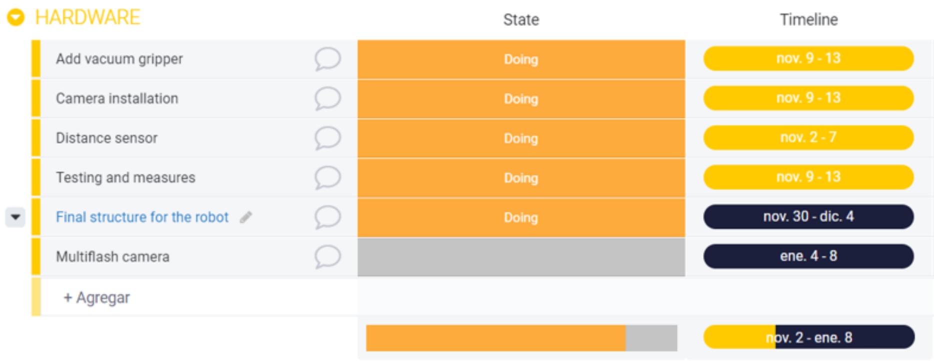

Fig. 4.2: Chronograph of the Hardward implementation

4.4 Planning and budget

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

Añadir presupuesto

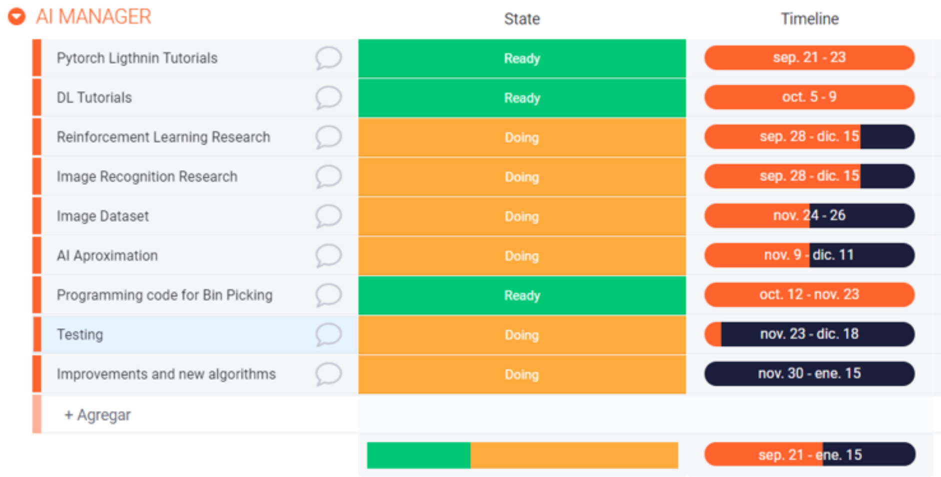

Fig. 4.3: Planning of the Artificial Intelligence implementation

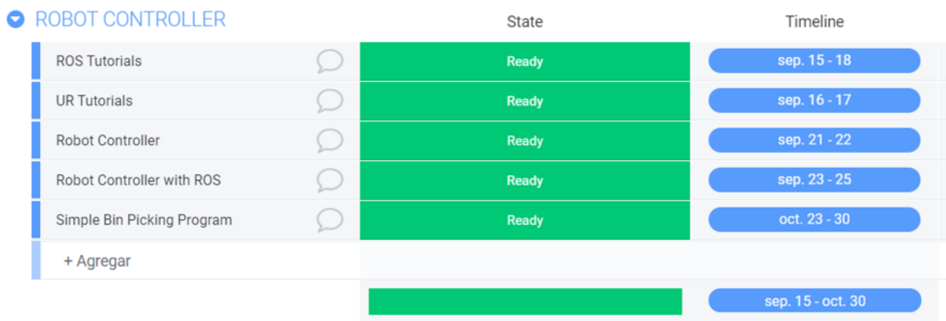

Fig. 4.4: Planning of the Robot Controller implementation

5. DEVELOPED SYSTEM

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

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

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

5.1 Análisis del sistema

...

5.2 Diseño

...

5.3 Implementación

...

6. RESULTS ANALYSIS

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

7. CONCLUSIONS AND FUTURE WORK

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

APPENDIX

.1 Robot Controller

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

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

https://github.com/pabloiglesia/robot_controller

.1.1 main.py

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

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

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

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

- Activate the talker
rosrun ai_manager main.py

- Activate the node
rosrun robot_controller arduino.py

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

"""

import rospy

from ai_manager.srv import GetActions
from Robot import Robot


def get_action(robot, object_gripped):
```

```
    relative_coordinates = robot.calculate_current_coordinates()
    rospy.wait_for_service('get_actions')
    try:
        get_actions = rospy.ServiceProxy('get_actions', GetActions)
        return get_actions(relative_coordinates[0], relative_coordinates[1], object_gripped).action
    except rospy.ServiceException as e:
        print("Service call failed: %s"%e)


# This function defines the movements that robot should make depending on the action listened
def take_action(action, robot):
    rospy.loginfo("Action received: {}".format(action))
    object_gripped = False
    if action == 'north':
        robot.take_north()
    elif action == 'south':
        robot.take_south()
    elif action == 'east':
        robot.take_east()
    elif action == 'west':
        robot.take_west()
    elif action == 'pick':
        object_gripped = robot.take_pick()
    elif action == 'random_state':
        robot.take_random_state()
    return object_gripped


if __name__ == '__main__':

    rospy.init_node('robotUR')

    robot = Robot()

    # Test of positioning with angular coordinates
    robot.go_to_initial_pose()

    # Let's put the robot in a random position to start, creation of new state
    object_gripped = take_action('random_state', robot)

    while True:
        action = get_action(robot, object_gripped)
        object_gripped = take_action(action, robot)
```

.1.2 Robot.py

```
import copy
import rospy
import time

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

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

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


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

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

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

```

```
        self.robot.exec_cartesian_path(waypoints)

    def calculate_relative_movement(self, relative_coordinates):
        absolute_coordinates_x = Environment.CARTESIAN_CENTER[0] - relative_coordinates[0]
        absolute_coordinates_y = Environment.CARTESIAN_CENTER[1] - relative_coordinates[1]

        current_pose = self.robot.get_current_pose()

        x_movement = current_pose.pose.position.x - absolute_coordinates_x
        y_movement = current_pose.pose.position.y - absolute_coordinates_y

        return x_movement, y_movement

    def calculate_current_coordinates(self):
        absolut_coordinate_x = self.robot.get_current_pose().pose.position.x
        absolut_coordinate_y = self.robot.get_current_pose().pose.position.y

        relative_coordinate_x = Environment.CARTESIAN_CENTER[0] - absolut_coordinate_x
        relative_coordinate_y = Environment.CARTESIAN_CENTER[1] - absolut_coordinate_y

        return [relative_coordinate_x, relative_coordinate_y]

    # Action north: positive x
    def take_north(self, distance=Environment.ACTION_DISTANCE):
        self.relative_move(distance, 0, 0)

    # Action south: negative x
    def take_south(self, distance=Environment.ACTION_DISTANCE):
        self.relative_move(-distance, 0, 0)

    # Action east: negative y
    def take_east(self, distance=Environment.ACTION_DISTANCE):
        self.relative_move(0, -distance, 0)

    # Action west: positive y
    def take_west(self, distance=Environment.ACTION_DISTANCE):
        self.relative_move(0, distance, 0)

    def take_random_state(self):
        # Move robot to random positions using relative moves. Get coordinates
```

```
        relative_coordinates = Environment.generate_random_state()
        # Calculate the new coordinates
        x_movement, y_movement = self.calculate_relative_movement(
    relative_coordinates)
        # Move the robot to the random state
        self.relative_move(x_movement, y_movement, 0)

    def send_gripper_message(self, msg, timer=2, n_msg=10):
        """
        Function that sends a burst of n messages of the
    gripper_topic during an indicated time
        :param msg: True or False
        :param time: time in seconds
        :param n_msg: number of messages
        :return:
        """
        time_step = (timer/2)/n_msg
        i=0
        while(i <= n_msg):
            self.gripper_publisher.publish(msg)
            time.sleep(time_step)
            i += 1

        time.sleep(timer/2)

    # Action pick: Pick and place
    def take_pick(self):
        # In this function we should read the distance to the
    object
        # up_distance = 0  # Variable were we store the distance
    that we have move the robot so that we can go back to the
        # original pose

        def change_plan_speed(plan, new_speed):
            """
            Function used for changing Robot velocity of a
    cartesian path once the movement have been planned.
            :param plan: RobotTrajectory object. For example, the
    one calculated by compute_cartesian_path() MoveGroup function.
            :param new_speed: speed factor of the robot, been 1
    the original speed and 0 the minimum.
            :return: RobotTrajectory object (new plan).
            """
            new_plan = plan
            n_joints = len(plan.joint_trajectory.joint_names)
            n_points = len(plan.joint_trajectory.points)

            points = []
            for i in range(n_points):
```

```
                plan.joint_trajectory.points[i].time_from_start = plan.joint_trajectory.points[
    i].time_from_start / new_speed
                velocities = []
                accelerations = []
                positions = []
                for j in range(n_joints):
                    velocities.append(plan.joint_trajectory.points[i].velocities[j] * new_speed)
                    accelerations.append(plan.joint_trajectory.points[i].accelerations[j] * new_speed)
                    positions.append(plan.joint_trajectory.points[i].positions[j])

                point = plan.joint_trajectory.points[i]
                point.velocities = velocities
                point.accelerations = accelerations
                point.positions = positions

                points.append(point)

            new_plan.joint_trajectory.points = points

            return new_plan

        def back_to_original_pose(robot):
            """
            Function used to go back to the original height once a vertical movement has been performed.
            :param robot: robot_controller.Robot.py object
            :return:
            """
            distance = Environment.CARTESIAN_CENTER[2] - robot.robot.get_current_pose().pose.position.z
            robot.relative_move(0, 0, distance)

        def down_movement(robot, movement_speed):
            """
            This function performs the down movement of the pick action.

            It creates an asynchronous move group trajectory planning. This way the function is able to receive distance
            messages while the robot is moving and stop it once the robot is in contact with an object.

            Finally, when there is any problems with the communications the movement is stopped and
```

```python
        communication_problem boolean flag is set to True. It is considered that there is a problem with
        communications when the robot is not receiving any distance messages during 200 milli-seconds (timeout=0.2)

        :param robot: robot_controller.Robot.py object
        :return: communication_problem flag
        """

        distance_ok = rospy.wait_for_message('distance', Bool).data  # We retrieve sensor distance
        communication_problem = False

        if not distance_ok:  # If the robot is already in contact with an object, no movement is performed
            waypoints = []
            wpose = robot.robot.get_current_pose().pose
            wpose.position.z -= (wpose.position.z)  # Third move sideways (z)
            waypoints.append(copy.deepcopy(wpose))

            (plan, fraction) = robot.robot.move_group.compute_cartesian_path(
                waypoints,  # waypoints to follow
                0.01,  # eef_step
                0.0)  # jump_threshold

            plan = change_plan_speed(plan, movement_speed)
            robot.robot.move_group.execute(plan, wait=False)

            while not distance_ok:
                try:
                    distance_ok = rospy.wait_for_message('distance', Bool, 0.2).data  # We retrieve sensor distance
                except:
                    communication_problem = True
                    rospy.loginfo("Error in communications, trying again")
                    break

            # Both stop and 10 mm up movement to stop the robot
            robot.robot.move_group.stop()
            robot.relative_move(0, 0, 0.001)

        return communication_problem

    communication_problem = True
    while communication_problem:  # Infinite loop until the
```

```
movement is completed
            communication_problem = down_movement(self, movement_speed=0.2)

        self.send_gripper_message(True, timer=4)  # We turn on the gripper

        back_to_original_pose(self)  # Back to the original pose

        object_gripped = rospy.wait_for_message('object_gripped', Bool).data
        if object_gripped:  # If we have gripped an object we place it into the desired point
            self.take_place()
        else:
            self.send_gripper_message(False)  # We turn off the gripper

        return object_gripped

    # Function to define the place for placing the grasped objects
    def take_place(self):
        # First, we get the cartesian coordinates of one of the corner
        x_box, y_box = Environment.get_relative_corner('se')
        x_move, y_move = self.calculate_relative_movement([x_box, y_box])
        # We move the robot to the corner of the box
        self.relative_move(x_move, y_move, 0)
        # We calculate the trajectory for our robot to reach the box
        trajectory_x = self.robot.get_current_pose().pose.position.x - Environment.PLACE_CARTESIAN_CENTER[0]
        trajectory_y = self.robot.get_current_pose().pose.position.y - Environment.PLACE_CARTESIAN_CENTER[1]
        trajectory_z = - Environment.CARTESIAN_CENTER[2] + Environment.PLACE_CARTESIAN_CENTER[2]
        # We move the robot to the coordinates desired to place the object
        self.relative_move(0, 0, trajectory_z)
        self.relative_move(0, trajectory_y, 0)
        self.relative_move(trajectory_x, 0, 0)
        # Then, we left the object
        self.relative_move(0, 0, -0.05)
        # Then, we switch off the vacuum gripper so the object can be placed
        self.send_gripper_message(False)
        # Wait some seconds, in order to the msg to arrive to the gripper
```

```
        time.sleep(2)
        # Then the robot goes up
        self.relative_move(0, 0, 0.05)
        # Final we put the robot in the center of the box, the
    episode should finish now
        self.robot.go_to_joint_state(Environment.ANGULAR_CENTER)

    def go_to_initial_pose(self):
        target_reached = self.robot.go_to_joint_state(Environment.
    ANGULAR_CENTER)

        if target_reached:
            print("Target reachead")
        else:
            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

```
#!/usr/bin/env python
"""
Code used to train the UR3 robot to perform a pick and place task
    using Reinforcement Learning and Image Recognition.
This code does not perform actions directly into the robot, it
    just posts actions in a ROS topic and
gathers state information from another ROS topic.
"""

import rospy
import torch

from ai_manager.srv import GetActions, GetActionsResponse
from RLAlgorithm import RLAlgorithm
from Environment import Environment

rospy.init_node('ai_manager', anonymous=True)  # ROS node
    initialization
```

```
# Global Image Controller
RL_ALGORITHM = RLAlgorithm.recover_training(batch_size=256, lr
    =0.0001,
                                            others='
    optimal_original_rewards_algorithm1901',
    include_pick_prediction=False)
                                            # others = '
    optimal_original_rewards_new_model')

def handle_get_actions(req):
    """
    Callback for each Request from the Robot
    :param req: Robot requests has 3 elements: object_gripped, x
    and y elements
    :return:
    """
    object_gripped = req.object_gripped
    current_coordinates = [req.x, req.y]
    # Next action is calculated from the current state
    action = RL_ALGORITHM.next_training_step(current_coordinates,
    object_gripped)

    # RL_ALGORITHM.plot()

    return GetActionsResponse(action)


def get_actions_server():
    """
    Service initialization to receive requests of actions from the
     robot.
    Each time that a request is received, handle_get_actions
    function will be called
    :return:
    """
    s = rospy.Service('get_actions', GetActions,
    handle_get_actions)
    rospy.loginfo("Ready to send actions.")
    rospy.spin()
    rospy.on_shutdown(save_training)


def save_training():
    RL_ALGORITHM.save_training()


if __name__ == '__main__':
    try:
        get_actions_server()
```

```
    except rospy.ROSInterruptException:
        pass
```

.2.2 RLAlgorithm.py

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

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

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

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

import pickle

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

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


class RLAlgorithm:
```

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

    To perform a Deep Reinforcement Learning training, the following steps have to be followed:

        1. Initialize replay memory capacity.
        2. Initialize the policy network with random weights.
        3. Clone the policy network, and call it the target network.
        4. For each episode:
            1. Initialize the starting state.
             2. For each time step:
                 1. Select an action.
                     - Via exploration or exploitation
                 2. Execute selected action in an emulator or in Real-life.
                 3. Observe reward and next state.
                 4. Store experience in replay memory.
                 5. Sample random batch from replay memory.
                 6. Preprocess states from batch.
                 7. Pass batch of preprocessed states to policy network.
                 8. Calculate loss between output Q-values and target Q-values.
                     - Requires a pass to the target network for the next state
                 9. Gradient descent updates weights in the policy network to minimize loss.
                     - After time steps, weights in the target network are updated to the weights in the policy network.

    """

    def __init__(self, object_gripped_reward=10, object_not_picked_reward=-10, out_of_limits_reward=-10,
                 horizontal_movement_reward=-1, batch_size=32, gamma=0.999, eps_start=1, eps_end=0.01, eps_decay=0.0005,
                 target_update=10, memory_size=100000, lr=0.001, num_episodes=1000, include_pick_prediction=False,
                 save_training_others='optimal'):
        """

        :param object_gripped_reward: Object gripped reward
        :param object_not_picked_reward: Object not picked reward
        :param out_of_limits_reward: Out of limits reward
        :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
        self.strategy = self.EpsilonGreedyStrategy(self.eps_start, self.eps_end, self.eps_decay)  # Greede Strategy
        self.agent = self.Agent(self)  # RL Agent
        self.memory = self.ReplayMemory(self.memory_size)  # Replay Memory
        self.statistics = TrainingStatistics()  # Training statistics

        self.policy_net = self.DQN(self.em.image_tensor_size,
                                   self.em.num_actions_available(),
                                   self.include_pick_prediction).to(self.device)  # Policy Q Network
        self.target_net = self.DQN(self.em.image_tensor_size,
                                   self.em.num_actions_available(),
                                   self.include_pick_prediction).to(self.device)  # Target Q Network
        self.target_net.load_state_dict(self.policy_net.state_dict())  # Target net has to be the same as policy network
        self.target_net.eval()  # Target net has to be the same as policy network
        self.optimizer = optim.Adam(params=self.policy_net.parameters(), lr=self.lr)  # Q Networks optimizer

        print("Device: ", self.device)

    class Agent:
        """
        Class that contains all needed methods to control the agent through the environment and retrieve information of
        Its state
        """

        def __init__(self, rl_algorithm):
            """

            :param self: RLAlgorithm object
            """
            self.strategy = rl_algorithm.strategy  # Greedy Strategy
            self.num_actions = rl_algorithm.em.num_actions_available()  # Num of actions available
            self.device = rl_algorithm.device  # Torch device
            self.rl_algorithm = rl_algorithm

        def select_action(self, state, policy_net):
            """
```

```
        Method used to pick the following action of the robot
        Method used to pick the following action of the robot
        :param state: State RLAlgorithm namedtuple with all the information of the current state
        :param policy_net: DQN object used as policy network for the RL algorithm
        :return:
        """
        random_action = False
        if self.rl_algorithm.episode_done:  # If the episode has just ended we reset the robot environment
            self.rl_algorithm.episode_done = False  # Put the variable episode_done back to False
            self.rl_algorithm.statistics.new_episode()

            self.rl_algorithm.current_action = 'random_state'  # Return random_state to reset the robot position
            self.rl_algorithm.current_action_idx = None
        else:
            rate = self.strategy.get_exploration_rate(
                self.rl_algorithm.statistics.current_step)  # We get the current epsilon value

            if rate > random.random():  # With a probability = rate we choose a random action (Explore environment)
                action = random.randrange(self.num_actions)
                random_action = True
            else:  # With a probability = (1 - rate) we Explote the information we already have
                try:
                    with torch.no_grad():  # We calculate the action using the Policy Q Network
                        action = policy_net(state.image_raw, torch.tensor(
                            [[state.coordinate_x, state.coordinate_y]], device=self.device),
                                            state.pick_probability, self.rl_algorithm.include_pick_prediction)\
                            .argmax(dim=1).to(self.device)  # exploit
                except:
                    print("Ha habido un error")

            self.rl_algorithm.current_action = self.rl_algorithm.em.actions[action]
            self.rl_algorithm.current_action_idx = action

        return self.rl_algorithm.current_action, random_action  # We return the action as a string, not as int
```

```

 class DQN(nn.Module):
     """
     Class to create a Deep Q Learning Neural Network
     """

     def __init__(self, image_tensor_size, num_actions,
include_pick_prediction):
         """

         :param image_tensor_size: Size of the input tensor
         :param num_actions: Number of actions, which is the
output of the Neural Network
         """
         super(RLAlgorithm.DQN, self).__init__()

         self.linear1 = nn.Linear(image_tensor_size, int(
image_tensor_size / 2))
         self.linear2 = nn.Linear(int(image_tensor_size / 2),
int(image_tensor_size / 4))
         extra_features = 2  # coordinates
         if include_pick_prediction:
             extra_features = 3  # pick prediction
         self.linear3 = nn.Linear(int(image_tensor_size / 4) +
extra_features, num_actions)
         self.linear = nn.Linear(image_tensor_size + 2,
num_actions)

     # Called with either one element to determine next action,
 or a batch
     # during optimization. Returns tensor([[left0exp,right0exp
]...]).
     def forward(self, image_raw, coordinates, pick_probability
=None, include_pick_probability=False):

         output = self.linear1(image_raw)
         output = self.linear2(output)
         if include_pick_probability:
             output = torch.cat((output, coordinates,
pick_probability), 1)
         else:
             output = torch.cat((output, coordinates), 1)
         return self.linear3(output)

 class EnvManager:
     """
     Class used to manage the RL environment. It is used to
perform actions such as calculate rewards or retrieve the
     current state of the robot.
```

```
    """

    def __init__(self, rl_algorithm, object_gripped_reward, object_not_picked_reward, out_of_limits_reward,
                 horizontal_movement_reward):
        """
        Initialization of an object
        :param rl_manager: RLAlgorithm object
        :param object_gripped_reward: Object gripped reward
        :param object_not_picked_reward: Object not picked reward
        :param out_of_limits_reward: Out of limits reward
        :param horizontal_movement_reward: Horizontal movement reward
        """
        self.object_gripped_reward = object_gripped_reward
        self.out_of_limits_reward = out_of_limits_reward
        self.object_not_picked_reward = object_not_picked_reward
        self.horizontal_movement_reward = horizontal_movement_reward

        self.device = rl_algorithm.device  # Torch device
        self.image_controller = ImageController()  # ImageController object to manage images
        self.actions = ['north', 'south', 'east', 'west', 'pick']  # Possible actions of the objects
        self.image_height = None  # Retrieved images height
        self.image_width = None  # Retrieved Images Width
        self.image = None  # Current image ROS message
        self.image_tensor = None  # Current image tensor
        self.pick_probability = None  # Current image tensor

        self.model_name = 'model-epoch=05-val_loss=0.36-weights7y3_unfreeze2.ckpt'
        # self.model_name = 'resnet50_freezed.ckpt'
        self.model_family = 'resnet50'
        self.image_model = ImageModel(model_name=self.model_family)
        self.feature_extraction_model = self.image_model.load_model(self.model_name)
        self.image_tensor_size = self.image_model.get_size_features(
            self.feature_extraction_model)  # Size of the image after performing some transformations

        self.rl_algorithm = rl_algorithm
        self.gather_image_state()  # Retrieve initial state image
```

```

    def calculate_reward(self, previous_image):
        """
        Method used to calculate the reward of the previous action and whether it is a final state or not
        :return: reward, is_final_state
        """
        current_coordinates = [self.rl_algorithm.current_state.coordinate_x,
                               self.rl_algorithm.current_state.coordinate_y]  # Retrieve robot's current coordinates
        object_gripped = self.rl_algorithm.current_state.object_gripped  # Retrieve if the robot has an object gripped
        if Environment.is_terminal_state(current_coordinates, object_gripped):  # If is a terminal state
            self.rl_algorithm.episode_done = True  # Set the episode_done variable to True to end up the episode
            episode_done = True
            if object_gripped:  # If object_gripped is True, the episode has ended successfully
                reward = self.object_gripped_reward
                self.rl_algorithm.statistics.add_succesful_episode(True)  # Saving episode successful statistic
                self.rl_algorithm.statistics.increment_picks()  # Increase of the statistics cpunter
                rospy.loginfo("Episode ended: Object gripped!")
                self.image_controller.record_image(previous_image, True)  # Saving the falure state image
            else:  # Otherwise the robot has reached the limits of the environment
                reward = self.out_of_limits_reward
                self.rl_algorithm.statistics.add_succesful_episode(False)  # Saving episode failure statistic
                rospy.loginfo("Episode ended: Environment limits reached!")
        else:  # If it is not a Terminal State
            episode_done = False
            if self.rl_algorithm.current_action == 'pick':  # if it is not the first action and action is pick
                reward = self.object_not_picked_reward
                self.image_controller.record_image(previous_image, False)  # Saving the falure state image
                self.rl_algorithm.statistics.increment_picks()  # Increase of the statistics counter
            else:  # otherwise
                self.rl_algorithm.statistics.
```

```
fill_coordinates_matrix(current_coordinates)
                reward = self.horizontal_movement_reward

        self.rl_algorithm.statistics.add_reward(reward)  # Add reward to the algorithm statistics
        return reward, episode_done

    def gather_image_state(self):
        """
        This method gather the relative state of the robot by retrieving an image using the image_controller class,
        which reads the image from the ROS topic specified.
        """
        previous_image = self.image
        self.image, self.image_width, self.image_height = self.image_controller.get_image()  # We retrieve state image
        self.image_tensor, pick_probability = self.extract_image_features(self.image)
        if self.rl_algorithm.include_pick_prediction:
            self.pick_probability = pick_probability

        return previous_image

    def extract_image_features(self, image):
        """
        Method used to transform the image to extract image features by passing it through the image_model CNN
        network
        :param image_raw: Image
        :return:
        """
        features, pick_prediction = self.image_model.evaluate_image(image, self.feature_extraction_model)
        features = torch.from_numpy(features)
        return features.to(self.device), torch.tensor([[math.exp(pick_prediction.numpy()[0][1])]]).to(self.device)

    def num_actions_available(self):
        """
        Returns the number of actions available
        :return: Number of actions available
        """
        return len(self.actions)

class EpsilonGreedyStrategy:
    """
    Class used to perform the Epsilon greede strategy
    """

```

```
    def __init__(self, start, end, decay):
        """
        Initialization
        :param start: Greedy strategy epsilon start (
Probability of random choice)
        :param end: Greedy strategy minimum epsilon (
Probability of random choice)
        :param decay: Greedy strategy epsilon decay (
Probability decay of random choice)
        """
        self.start = start
        self.end = end
        self.decay = decay

    def get_exploration_rate(self, current_step):
        """
        It calculates the rate depending on the actual step of
 the execution
        :param current_step: step of the training
        :return:
        """
        return self.end + (self.start - self.end) * \
               math.exp(-1. * current_step * self.decay)

 class QValues:
    """
    It returns the predicted q-values from the policy_net for
the specific state-action pairs that were passed in.
    states and actions are the state-action pairs that were
sampled from replay memory.
    """

    @staticmethod
    def get_current(policy_net, states, coordinates, actions,
pick_probabilities, include_pick_prediction = False):
        """
        With the current state of the policy network, it
calculates the q_values of
        :param policy_net: policy network used to decide the
actions
        :param states: Set of state images (Preprocessed)
        :param coordinates: Set of robot coordinates
        :param actions: Set of taken actions
        :return:
        """
        return policy_net(states, coordinates,
pick_probabilities, include_pick_prediction).gather(dim=1,
index=actions.unsqueeze(-1))

```

```python
    @staticmethod
    def get_next(target_net, next_states, next_coordinates, next_pick_probabilities, is_final_state, include_pick_prediction = False):
        """
        Calculate the maximum q-value predicted by the target_net among all possible next actions.
        If the action has led to a terminal state, next reward will be 0. If not, it is calculated using the target
        net
        :param target_net: Target Deep Q Network
        :param next_states: Next states images
        :param next_coordinates: Next states coordinates
        :param is_final_state: Tensor indicating whether this action has led to a final state or not.
        :return:
        """
        batch_size = next_states.shape[0]  # The batch size is taken from next_states shape
        # q_values is initialized with a zeros tensor of batch_size and if there is GPU it is loaded to it
        q_values = torch.zeros(batch_size).to(torch.device("cuda" if torch.cuda.is_available() else "cpu"))
        non_final_state_locations = (is_final_state == False)  # Non final state index locations are calculated
        non_final_states = next_states[non_final_state_locations]  # non final state images
        non_final_coordinates = next_coordinates[non_final_state_locations]  # non final coordinates
        if include_pick_prediction:
            non_final_pick_probabilities = next_pick_probabilities[non_final_state_locations]  # non final pick probabilities
        else:
            non_final_pick_probabilities = None
        # Max q values of the non final states are calculated using the target net
        q_values[non_final_state_locations] = \
            target_net(non_final_states, non_final_coordinates, non_final_pick_probabilities, include_pick_prediction).max(dim=1)[
                0].detach()
        return q_values

class ReplayMemory:
    """
    Class used to create a Replay Memory for the RL algorithm
    """

```

```
    def __init__(self, capacity):
        """
        Initialization of ReplayMemory
        :param capacity: Capacity of Replay Memory
        """
        self.capacity = capacity
        self.memory = []  # Actual memory. it will be filled with Experience namedtuples
        self.push_count = 0  # will be used to keep track of how many experiences have been added to the memory

    def push(self, experience):
        """
        Method used to fill the Replay Memory with experiences
        :param experience: Experience namedtuple
        :return:
        """
        if len(self.memory) < self.capacity:  # if memory is not full, new experience is appended
            self.memory.append(experience)
        else:  # If its full, we add a new experience and take the oldest out
            self.memory[self.push_count % self.capacity] = experience
        self.push_count += 1  # we increase the memory counter

    def sample(self, batch_size):
        """
        Returns a random sample of experiences
        :param batch_size: Number of randomly sampled experiences returned
        :return: random sample of experiences (Experience namedtuples)
        """
        return random.sample(self.memory, batch_size)

    def can_provide_sample(self, batch_size):
        """
        returns a boolean telling whether or not we can sample from memory. Recall that the size of a sample
        we'll obtain from memory will be equal to the batch size we use to train our network.
        :param batch_size: Batch size to train the network
        :return: boolean telling whether or not we can sample from memory
        """
        return len(self.memory) >= batch_size

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 Experienc 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:
```

```
                    os.makedirs(os.path.dirname(filename))
                except OSError as exc:  # Guard against race condition
                    if exc.errno != errno.EEXIST:
                        raise
            return filename

        rospy.loginfo("Saving training...")

        abs_filename = create_if_not_exist(filename, dir)

        self.em.image_model = None
        self.em.feature_extraction_model = None

        with open(abs_filename, 'wb+') as output:  # Overwrites any existing file.
            pickle.dump(self, output, pickle.HIGHEST_PROTOCOL)

        rospy.loginfo("Saving Statistics...")
        print(filename)

        filename = 'trainings/{}_stats.pkl'.format(filename.split('.pkl')[0])
        self.statistics.save(filename=filename)

        rospy.loginfo("Training saved!")

    @staticmethod
    def recover_training(batch_size=32, gamma=0.999, eps_start=1, eps_end=0.01,
                         eps_decay=0.0005, lr=0.001, include_pick_prediction=False, others='optimal', dir='trainings/', ):
        current_path = os.path.dirname(os.path.realpath(__file__))
        filename = RLAlgorithm.saving_name(batch_size, gamma, eps_start, eps_end, eps_decay, lr, others)
        filename = os.path.join(current_path, dir, filename)
        try:
            with open(filename, 'rb') as input:
                rl_algorithm = pickle.load(input)
                rospy.loginfo("Training recovered. Next step will be step number {}"
                              .format(rl_algorithm.statistics.current_step))

                rl_algorithm.em.image_model = ImageModel(model_name=rl_algorithm.em.model_family)
                rl_algorithm.em.feature_extraction_model = rl_algorithm.em.image_model.load_model(
```

```
                    rl_algorithm.em.model_name)

                return rl_algorithm
        except IOError:
            rospy.loginfo("There is no Training saved. New object has been created")
            return RLAlgorithm(batch_size=batch_size, gamma=gamma, eps_start=eps_start, eps_end=eps_end,
                                eps_decay=eps_decay, lr=lr, include_pick_prediction=include_pick_prediction, save_training_others=others)

    def train_net(self):
        """
        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
        thew Bellman's equation:

        q*(state, action) = Reward + gamma * max( q*(next_state, next_action) )
        :return:
        """
        # If there are at least as much experiences stored as the batch size
        if self.memory.can_provide_sample(self.batch_size):
            experiences = self.memory.sample(self.batch_size)  # Retrieve the experiences
            # We split the batch of experience into different tensors
            states, coordinates, pick_probabilities, actions, rewards, next_states, next_coordinates, \
                next_pick_probabilities, is_final_state = self.extract_tensors(experiences, self.include_pick_prediction)
            # To compute the loss, current_q_values and target_q_values have to be calculated
            current_q_values = self.QValues.get_current(self.policy_net, states, coordinates, actions,
                                                        pick_probabilities, self.include_pick_prediction)
            # next_q_values is the maximum Q-value of each future state
            next_q_values = self.QValues.get_next(self.target_net, next_states, next_coordinates,
                                                  next_pick_probabilities, is_final_state, self.include_pick_prediction)
            target_q_values = (next_q_values * self.gamma) +
```

```
rewards

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

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

    def next_training_step(self, current_coordinates,
object_gripped):
        """
        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.
        :param current_coordinates: Tuple of float indicating
current coordinates of the robot
        :param object_gripped: Boolean indicating whether or not
ann object has been gripped
        :return: action taken
        """
        self.statistics.new_step()  # Add new steps statistics
        self.previous_state = self.current_state  # Previous state
 information to store in the Replay Memory
        previous_action = self.current_action  # Previous action
to store in the Replay Memory
        previous_action_idx = self.current_action_idx  # Previous
action index to store in the Replay Memory
        previous_image = self.em.gather_image_state()  # Gathers
current state image

        self.current_state = State(current_coordinates[0],
current_coordinates[1], self.em.pick_probability,
```

```
                                          object_gripped, self.em.image_tensor)  # Updates current_state

        # Calculates previous action reward an establish whether the current state is terminal or not
        previous_reward, is_final_state = self.em.calculate_reward(previous_image)
        action, random_action = self.agent.select_action(self.current_state,
                                                         self.policy_net)  # Calculates action

        # There are some defined rules that the next action have to accomplish depending on the previous action
        action_ok = False
        while not action_ok:
            # Its forbidden to perform two cosecutive pick actions in the same place
            if action == 'pick' and previous_action != 'pick':
                action_ok = True
            # If previous action was south, it is forbidden to perform a 'north' action for
            # The robot not to go back to the original position.
            elif action == 'north' and previous_action != 'south':
                action_ok = True
            # If previous action was north, it is forbidden to perform a 'south' action for
            # The robot not to go back to the original position.
            elif action == 'south' and previous_action != 'north':
                action_ok = True
            # If previous action was east, it is forbidden to perform a 'west' action for
            # The robot not to go back to the original position.
            elif action == 'west' and previous_action != 'east':
                action_ok = True
            # If previous action was west, it is forbidden to perform a 'east' action for
            # The robot not to go back to the original position.
            elif action == 'east' and previous_action != 'west':
                action_ok = True
            elif action == 'random_state':
                action_ok = True
            else:
                action, random_action = self.agent.select_action(self.current_state,
                                                                 self.policy_net)  # Calculates action

        if random_action:
```

```
            self.statistics.random_action()  # Recolecting statistics

        # Random_state actions are used just to initialize the environment to a random position, so it is not taken into
        # account while storing state information in the Replay Memory.
        # If previous action was a random_state and it is not the first step of the training
        if previous_action != 'random_state' and self.statistics.current_step > 1:
            self.memory.push(  # Pushing experience to Replay Memory
                Experience(  # Using an Experience namedtuple
                    self.previous_state.image_raw,  # Initial state image
                    torch.tensor([[self.previous_state.coordinate_x, self.previous_state.coordinate_y]],
                                 device=self.device),  # Initial coordinates
                    self.previous_state.pick_probability,
                    torch.tensor([previous_action_idx], device=self.device),  # Action taken
                    self.current_state.image_raw,  # Final state image
                    torch.tensor([[self.current_state.coordinate_x,
                                   self.current_state.coordinate_y]],
                                 device=self.device),  # Final coordinates
                    self.current_state.pick_probability,
                    torch.tensor([previous_reward], device=self.device),  # Action reward
                    torch.tensor([is_final_state], device=self.device)  # Episode ended
                ))

            # Logging information
            rospy.loginfo("Step: {}, Episode: {}, Previous reward: {}, Previous action: {}".format(
                self.statistics.current_step - 1,
                self.statistics.episode,
                previous_reward,
                previous_action))

            self.train_net()  # Both policy and target networks gets trained

```

```
        return action
```

.2.3 Environment.py

```
"""
This class defines a RL environment for a pick and place task with
    a UR3 robot.
This environment is defined by its center (both cartesian and
    angular coordinates), the total length of its x and y axis
and other parameters
"""

import random
from BlobDetector.BlobDetector import BlobDetector
from ai_manager.ImageController import ImageController
from math import floor


class Environment:
    X_LENGTH = 0.175  # Total length of the x axis environment in
    meters
    Y_LENGTH = 0.225  # Total length of the y axis environment in
    meters

    CAMERA_SECURITY_MARGIN = 0.035  # As the camera is really
    close to the gripping point, it needs  a security marging
    X_LIMIT = X_LENGTH - CAMERA_SECURITY_MARGIN  # Robot
    boundaries of movement in axis X
    Y_LIMIT = Y_LENGTH - CAMERA_SECURITY_MARGIN  # Robot
    boundaries of movement in axis Y

    CARTESIAN_CENTER = [-0.31899288568, -0.00357907370787,
    0.376611799631]  # Cartesian center of the RL environment
    ANGULAR_CENTER = [2.7776150703430176, -1.5684941450702112,
    1.299912452697754, -1.3755658308612269,
                      -1.5422008673297327, -0.3250663916217249]  #
     Angular center of the RL environment
    PLACE_CARTESIAN_CENTER = [0, 0.25, CARTESIAN_CENTER[2]]  #
    Cartesian center of the place box
    ANGULAR_PICTURE_PLACE = [1.615200161933899,
    -1.235102955495016, 0.739865779876709, -1.2438910643206995,
    -1.5095704237567347, -0.06187755266298467]

    PICK_DISTANCE = 0.01  # Distance to the object when the robot
    is performing the pick and place action
    ACTION_DISTANCE = 0.02  # Distance to the object when the
    robot is performing the pick and place action

```

```python
    ENV_BOUNDS_TOLERANCE = 0

    @staticmethod
    def generate_random_state(image=None, strategy='ncc'):
        """
        Calculates random coordinates inside the Relative Environment defined.
        To help the robot empty the box, the generated coordinates won't be in the center of the box, because this is
        the most reachable place of the box.

        :param strategy: strategy used to calculate random_state coordinates
        :return:
        """
        def generate_random_coordinates():
            coordinate_x = random.uniform((-Environment.X_LIMIT + Environment.ENV_BOUNDS_TOLERANCE) / 2,
                                          (Environment.X_LIMIT - Environment.ENV_BOUNDS_TOLERANCE) / 2)
            coordinate_y = random.uniform((-Environment.Y_LIMIT + Environment.ENV_BOUNDS_TOLERANCE) / 2,
                                          (Environment.Y_LIMIT - Environment.ENV_BOUNDS_TOLERANCE) / 2)
            return coordinate_x, coordinate_y

        # Random coordinates avoiding the ones in the center, which have a bigger probability of being reached by the
        # robot.
        if strategy == 'ncc' or strategy == 'non_centered_coordinates':
            coordinates_in_center = True
            while coordinates_in_center:
                coordinate_x, coordinate_y = generate_random_coordinates()
                if abs(coordinate_x) > (Environment.X_LIMIT / 4) or abs(coordinate_y) > (Environment.Y_LIMIT / 4):
                    coordinates_in_center = False
        elif strategy == 'optimal' and image is not None:  # Before going to a random state, we check that there are pieces in this place
            blob_detector = BlobDetector(x_length=Environment.X_LENGTH, y_length=Environment.Y_LENGTH, columns=4, rows=4)
            optimal_quadrant = blob_detector.find_optimal_quadrant(image)
            optimal_point = blob_detector.quadrants_center[optimal_quadrant]

            coordinate_x = optimal_point[0] * 0.056
```

```
            coordinate_y = optimal_point[1] * 0.056
        else: # Totally random coordinates
            coordinate_x, coordinate_y = generate_random_coordinates()

        return [coordinate_x, coordinate_y]

    @staticmethod
    def get_relative_corner(corner):
        """
        Function used to calculate the coordinates of the environment corners relative to the CARTESIAN_CENTER.

        :param corner: it indicates the corner that we want to get the coordinates. It's composed by two letters
        that indicate the cardinality. For example: ne indicates North-East corner
        :return coordinate_x, coordinate_y:
        """
        if corner == 'sw' or corner == 'ws':
            return -Environment.X_LIMIT / 2, Environment.Y_LIMIT / 2
        if corner == 'nw' or corner == 'wn':
            return Environment.X_LIMIT / 2, Environment.Y_LIMIT / 2
        if corner == 'ne' or corner == 'en':
            return Environment.X_LIMIT / 2, -Environment.Y_LIMIT / 2
        if corner == 'se' or corner == 'es':
            return -Environment.X_LIMIT / 2, -Environment.Y_LIMIT / 2

    @staticmethod
    def is_terminal_state(coordinates, object_gripped):
        """
        Function used to determine if the current state of the robot is terminal or not
        :return: bool
        """
        def get_limits(length): return length / 2 - Environment.ENV_BOUNDS_TOLERANCE  # functon to calculate the box boundaries
        x_limit_reached = abs(coordinates[0]) > get_limits(Environment.X_LIMIT)  # x boundary reached
        y_limit_reached = abs(coordinates[1]) > get_limits(Environment.Y_LIMIT)  # y boundary reached
        return x_limit_reached or y_limit_reached or object_gripped # If one or both or the boundaries are reached --> terminal state
```

.2.4 ImageController.py

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

import os
import time

import rospy
from PIL import Image as PILImage
from sensor_msgs.msg import Image

"""
This class is used to manage sensor_msgs Images.
"""

class ImageController:
    def __init__(self, path=os.path.dirname(os.path.realpath(
    __file__)), image_topic='/usb_cam/image_raw'):
        self.ind_saved_images = 0  # Index which will tell us the
    number of images that have been saved
        self.success_path = "{}/success".format(path)  # Path
    where the images are going to be saved
        self.fail_path = "{}/fail".format(path)  # Path where the
    images are going to be saved
        self.image_topic = image_topic

        # If it does not exist, we create the path folder in our
    workspace
        try:
            os.stat(self.success_path)
        except:
            os.mkdir(self.success_path)

        # If it does not exist, we create the path folder in our
    workspace
        try:
            os.stat(self.fail_path)
        except:
            os.mkdir(self.fail_path)

    def get_image(self):
        msg = rospy.wait_for_message(self.image_topic, Image)

        return self.to_pil(msg), msg.width, msg.height

    def record_image(self, img, success):
        path = self.success_path if success else self.fail_path  #
     The path were we want to save the image is
```

```

        image_path = '{}/img{}.png'.format(  # Saving image
            path,  # Path
            time.time())  # FIFO queue

        img.save(image_path)

        self.ind_saved_images += 1  # Index increment

    def to_pil(self, msg, display=False):
        size = (msg.width, msg.height)  # Image size
        img = PILImage.frombytes('RGB', size, msg.data)  # sensor_msg to Image
        return img

if __name__ == '__main__':
    rospy.init_node('image_recorder')  # ROS node initialization
    image_controller = ImageController(path='/home/pilar/Pilar/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. URL: http://arxiv.org/abs/1609.05521 (visited on 11/30/2020).

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

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

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

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

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

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

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