Reinforcement Learning to perform a pick and place task with a robotic arm

Master's Degree in Big Data Technologies and Advanced Analytics
Pablo Iglesia Fernández-Tresguerres
Directed by: Philippe Juhel

February 10, 2021

Abstract

I. Objectives

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.

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.

Besides, 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.

II. 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.

Figure 1: *Wooden Pieces used in the project*

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.

III. System Developed

The objects have to be taken from a box (Environment Box) and placed in another box (Place box). The pieces had to be something light due to the limitations of the UR3 robot that we commented before. As there wasn't any sponsor for the project we could decide the shape of the pieces, and we decided to use 5 cm size wooden squares as the ones showed in Figure 1.

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

- Environment
- sensorimotor devices
- Computational devices

To understand it better, we are going to use Figure 2 and Figure 3, which are labeled pictures of the architecture, where we are going to be able to see how the components of the architecture are like.

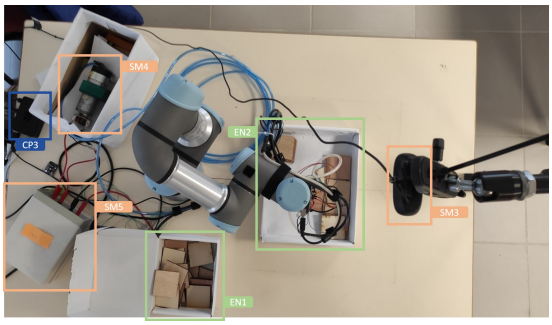

Figure 2: *Labelled picture of the Architecture I*

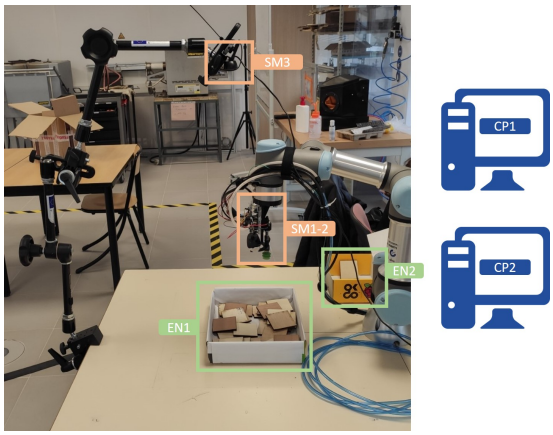

Figure 3: *Labelled picture of the Architecture II*

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

- **Environment**: We can see this elements in both images, from different perspective.
  - **EN1**: The Environment box where the agent has to take the pieces
  - **EN2**: The box where the agent has to place the pieces
- **Sensorimotor devices** Ones are showed in one image, and others in the other.
  - **SM1**: This is the Onboard camera, used for the agent to decide which action to take. It is attached to the gripper, so in the picture they are shown together.

  - **SM2**: Together with the camera, we can see the "Home made" gripper used to pick the pieces.
  - **SM3**: the upper camera, where the agent can pick a global picture of the environment. This picture will be important, but we will explain it later.
  - **SM4**: the pump of the Gripper.
  - **SM5**: Both 12V and 24V power adaptor used to feed the pump and some sensors.
- **Computational devices**. In the picture we cannot see all the computer used in the project, but there are 2 icons used to represent them.
  - **CP1**: This is the ROS Master Node. All of the nodes of the system will be connected to this node. Besides being the master node, robot_controller node and Universal Robots driver will also bo running in this computer.
  - **CP2**: This computer is a really powerful one, with one of the bes graphical cards in the market an 32 GB of RAM. It will be used to train the algorithm, running the ai_manager node.
  - **CP3**: This mini-computer can be seen in one of the pictures and its a Raspberry-pi. This computer will be used as a bridge form the arduino card of the gripper and the ROS master node. The cameras will also be attached to the Raspberry-pi.

IV. Logical Architecture

Once we have seen the physical architecture of the project, let's see how the Software architecture is. The Logical architecture will use all the elements commented on the previous section, and they will work together using ROS (Robot Operative System).

In Figure 4 we can see the logical architecture of the application, which is composed of 6 nodes communicating one with each other. We can see that the communication topics are written in the figure and that there are some

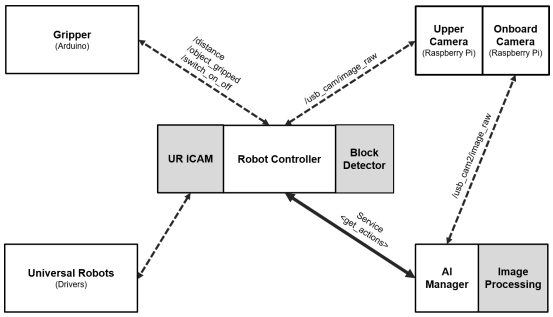

Figure 4: *Logical Architecture of the project*

squares attached one to another, and some of them are grey. All the white squares are ROS nodes, while grey ones are separate pieces of code that the nodes are using, but they are not ROS nodes by themselves. On the other hand, both camera nodes that are together in the upper right corner are two independent nodes, but using the same code to send the cameras images.

- Universal Robots Driver: Drivers to control the robot remotely.
- Robot Controller: It is the central node of the system. Here all the actions are implemented and it is dedicated to repeating over and over again the actions that the AI Manager indicates.
- Gripper: It is a node implemented in an Arduino board and that is in continuous contact with the Robot controls to execute the actions.
- Upper Camera: Node that constantly publishes the images captured by the upper camera on a topic.
- Onboard Camera: Node that constantly publishes the images captured by the onboard camera on a topic.
- **AI Manager: It is the intelligence of the system. Here the Reinforcement Learning algorithm is implemented.** The Robot Controller represents the agent, who takes an action and asks the AI Manager to calculate the next action to maximize the reward. The AI Manager collects the state image from the Onboard Camera node and calculates the action.

i. Algorithm

The chosen AI algorithm was AI or Deep Q Learning (DQL). DQL or Double Deep Q Learning (DDQN) depending on the implementation, is the approach of mixing both image recognition and Reinforcement Learning, 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 [**?**]. Another really interesting example is this robot [**?**], 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 [**?**]. In this tutorial they explain different versions of the algorithm and how to implement them in python to solve different OpenAI gym environments [**?**].

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 5 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:

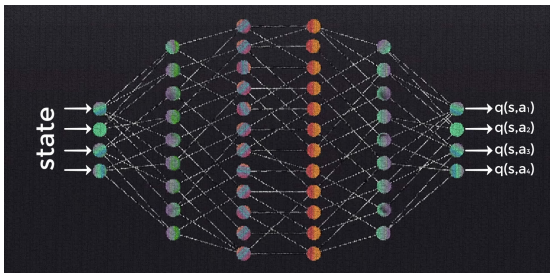

Figure 5: *Deep Q Learning Representation with 4 outputs*

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

i.1 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.

i.2 Exploration vs Exploitation trade-off

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

In Deep Reinforcement Learning, the agent exploit the information gathered by using the pre-trained Neural Network to decide next action. On the other hand, the agent explore the environment by deciding next action randomly. We use exploration mainly in the beginning

of the training because we want the agent to gather as much information of the environment as possible before starting training.

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

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

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

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

i.3 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.

i.4 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:

$$q_*(s,a) = E[R_t + \gamma max(q_*(s',a'))] \quad (1)$$

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:

$$q_*(s, a) = E[R_t + \gamma max(q(s', a')] \quad (2)$$

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.

i.5 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:

$$loss = R_t + \gamma max(q(s_{t+1}, a_{t+1})) - q(s_t, a_t) \quad (3)$$

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

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

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

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

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

i.6 DQL Training

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

The training basically uses the following schema:

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

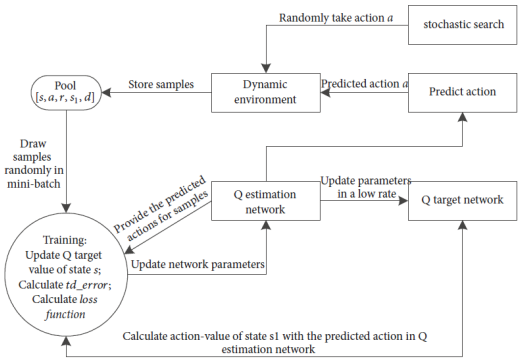

Figure 6: *Reinforcement Learning Training Summary*

This training can also be explained with Figure 6, where the Pool is the Replay Memory that stores the sample (experiences) from real interaction with the environment and feeds the training node of random sample of experiences.

Then, we can see that we use the Q-Network (policy network) to predict actions, but also to calculate the q-value of Replay Memory's batch. Then, we use the target network to predict action-value of state s1 with the predicted

action in Q estimation Network, andthis network is updated in a low rate.

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

ii. Deep Reinforcement Learning Implementation

The system developed is a really big implementation and there were multiple nodes involved. However, for the aim of this article, I will only comment the AI Manager node preented in previous subsections, as this is the node qere the artificial intelligence is implemented.

V. ai_manager

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

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

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

i. Definition of the problem

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

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

Having this in mind, we have decided that the robot have to be fixed in a specific height with the gripper facing down. Then, the actions will be "north", "south", "east" or "west" to move the robot through the x-y plane formed by these movements in the selected robot height, "pick", to perform the gripping process described before and place the object in the box, and "random_state" to move the robot to a new random state when a terminal state is reached.

ii. Environment.py

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

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

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

Finally, the methods defined in this class are:

- ***generate_random_state(strategy='ncc')***, which is used when the agent reaches a

terminal state and needs a new random state.
- ***get_relative_corner(corner)***, which returns the relative coordinates of a corner of the box
- ***is_terminal_state(coordinates, object_gripped)***, which returns a boolean telling whether a given state is terminal or not using the parameters given.

iii. Rewards

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

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

iv. Algorithm

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

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

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

iv.1 RLAlgorithm.py

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

- **Agent:** Inner class used to define the agent. The most important thing about this class is the select_action method, which is the one used to calculate the action using whether Exploration or Exploitation.
- **DQN:** Inner class used to define the target and policy networks. It defines a neural network that have to be called using the vector of features calculated by passing the image through the feature extractor net.
- **EnvManager:** Inner Class used to manage the RL environment. It is used to perform actions such as calculate rewards or gather

the current state of the robot. The most important methods are:

  - **calculate_reward**, which calculates the reward of each action depending on the initial and final state.
  - **calculate_reward**, which calculates the reward of each action depending on the initial and final state.
  - **extract_image_features**, which is used to transform the image to extract image features by passing it through a pre-trained CNN network that can be found in ImageModel Module.

- **EpsilonGreedyStrategy**: Inner Class used to perform the Epsilon greede strategy
- **QValues:** Inner class used to get the predicted q-values from the policy_net for the specific state-action pairs passed in. States and actions are the state-action pairs that were sampled from replay memory.
- **ReplayMemory:** Inner Class used to create a Replay Memory for the RL algorithm
- **Environment:** Class where the RL Environment is defined
- **TrainingStatistics:** Class used to store all the training statistics. If it is run separately, It will plot a set of graphs to represent visually the training evolution.
- **ImageModel:** Class used to extract the image features used in the training. You can find this class in this repository, which store another module of this project.
- **ImageController:** Class used to gather and store the relative state images from a ros topic.

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

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

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

- **save_training:** Method used to save the training so that it can be retaken later. It uses pickle library to do so and stores the whole RLAlgorithm object because all the context is needed to retake the training. This method also stores a pickle a TrainingStatistics object for them to be accessible easily.
- **recover_training:** Method used to recover saved trainings. If it doesn't find a file with the name given, it creates a new RLAlgorithm object.
- **train_net:** Method used to train both the train and target Deep Q Networks. We train the network minimizing the loss between the current Q-values of the action-state tuples and the target Q-values. Target Q-values are calculated using the Bellman's equation:

$$q_*(s,a) = E[R_t + \gamma max(q(s',a')] \quad (4)$$

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

action will be calculated and the previous action, its reward and its final state will be stored in the replay memory.

v. Training Flow

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

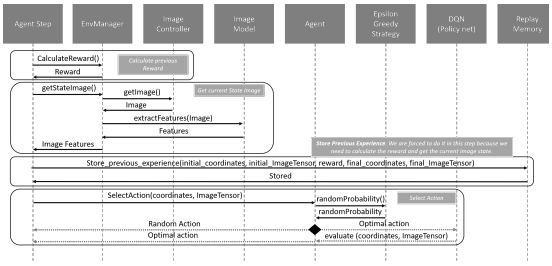

Figure 7: *Flow chart explaining training steps*

In Figure 7, we can see a flow chart explaining what happens during a training step in AI Manager. In the previous section I told that we would explain deeply what was the training process in AI Manager. We will explain it now with this chart, but, again, it is a simplified flow.

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

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

Finally, we train the algorithm in every step, following the steps shown on Figure 8.

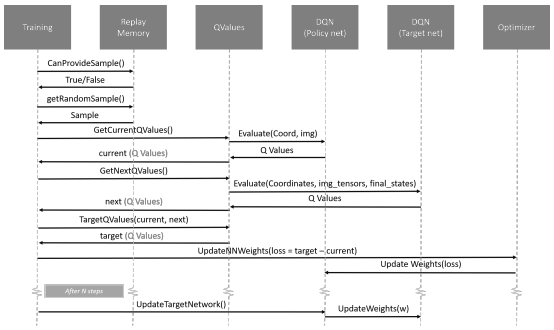

Figure 8: *Flow chart explaining the training process*

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

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

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

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

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

vi. Image Model

Image model is the module which is connected to AI Manager and is in charge of extracting the

features of the images. It uses some Data augmentation techniques to pre process the image, and then it is passed through a Convolutional Neural network to extract its features.

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

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

vii. Rewards

Rewards Is one of the most important and difficult things to do in Reinforcement Learning Algorithms. In this case we used several different configuration of Rewards.

The final combination is the following:

- **Successful pick actions:** 10
- **Unsuccessful pick actions:** -10
- **Robot out of boundaries:** -10
- **All the other actions:** -1

We used this rewards because we noticed that, if we didn't give negative reward for the normal actions, the robot was never performing pick actions, because the probability of failing was higher than the probability of succeeding.

Besides, we added some prior-knowledge to the algorithm. Prior knowledge technique is used in really complex Reinforcement Learning Problems as the one we were facing, and consist on limiting some actions depending on other conditions.

For example, in this algorithm we forbid to perform two consecutive pick actions in the same place without moving, because if the first pick action wasn't successful, the second one will not be successful either.

Results Analysis

In Reinforcement Learning, results vary a lot depending on the parameters of the training. To understand how we are going to compare the different algorithms, we have to first define some metrics. In this case, the metrics are:

- **The number of pick actions per episode**: When the agent performs a pick action and picks an object, the episode is ended. So, in this case, **the lowest the better**.
- **Reward per episode**: In each episode, the agent is trying to maximize the total reward, so, in this case, **the highest the better**
- **Number of steps per episode:** We want to minimize the time between picks. **The lower the better**
- **Evolution of successful episodes during the training:** As we already know, the episodes can be ended because the agent has pick an object (success) or when the robot has reached the environment limits (not success). The number of unsuccessful episodes should decrease during the training.
- **Random actions per episode:** This is not actually a measure of performance but a measure of control. If the training is going well, when the number of random actions decreases, the rest of the measures should improve.

Once that we have the measures, we should understand how to compare this information. For example I have decided that, for the first 3 measures, we shouldn't take into account the information of unsuccessful episodes, because that could result on a false sense of improvement. Let's imagine the "Number of steps per episode" metric, if we took into account the non successful episodes, we could have an episodes of one steps thinking that we are improving, while we are actually worsen. The same could happen with the number of picks and even with the reward metric, because, although the reward for reaching the environment limits is really low, the reward of a failed pick is as low as the other, so we could also be having a false

sense of improving.

On the other hand, in order to have softer and more readable graphs, I have decided to plot the mean of the last N values instead of the values themselves. I probably got inspiration from the "Theory of Communications" course.

The results gotten can be seen in the following set of figures. In Figure 9 we can see how the pick actions per episode decreased during the training. The mean was between 1.2 and 1.4 which are goods results having in mind that we are only showing the successful episodes. The optimal value of this metric would be 1 pick per successful episode, so this results shows that there are much more successful picks than unsuccessful.

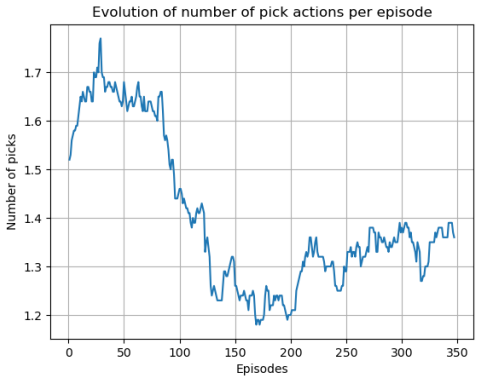

Figure 9: *Evolution of pick actions in episodes*

In Figure 10, we can see how rewards evolve during the training. As we can see, the reward is increasing during the whole training. The reward is still negative but it is normal, because all the actions have negative rewards but the successful pick action, that has a positive reward.The objective of the training would be having positive rewards, that would mean picking a piece in the first 10 steps of the episode without failing any other pick. This is a performance that, unfortunately is still far from happening, although our performance is good enough to reach our objectives, as we will see later.

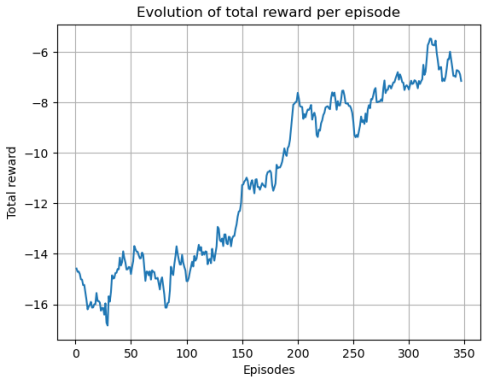

Figure 10: *Evolution of rewards through episodes*

In Figure 11, we can see the evolution of steps per episode. This metric has improve a lot, as we can see that the agent is picking pieces much faster than in the beginning.

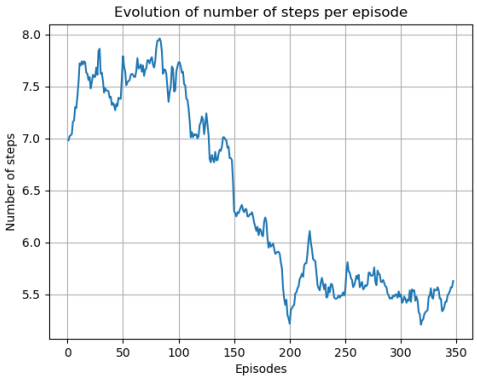

Figure 11: *Evolution of steps through episodes*

As we said before, the improvement in the other metrics is important, but we have to check if this is happening when we reduce the number of random actions. If it doesn't, it can just be coincidence. In Figure 12 we can see how the random actions kept decreasing during the training, showing that the good impressions of the training are well founded.

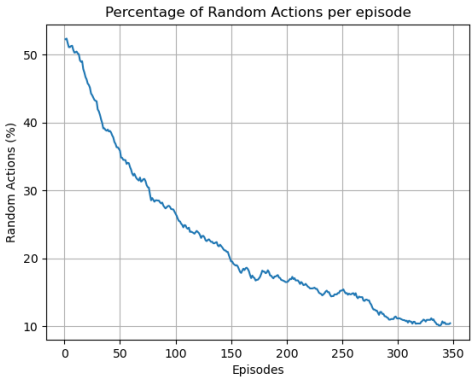

Figure 12: *Evolution of random actions per episode*

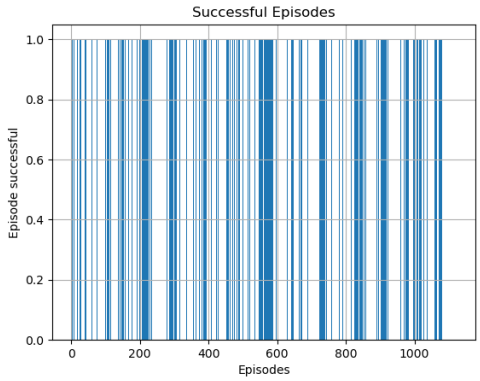

Figure 13: *Evolution of successful episodes during training*

Until now, all the news regarding were good. However, we cannot be completely happy with the results showed in Figure 13. In this image we can see how successful and non successful episodes have evolved during training. Having in mind that the big white holes (set of unsuccessful episodes) can be considered as accidents during the training (forgetting to refill the box with pieces, or failures of the system while calculating the robot coordinates), it seems like the rate of successful vs unsuccessful episodes improve over the training.

However, This is probably the worst metric in the training, and it is something to improve in the next iterations. If I had to say the reason, I would say that sometimes, when the box is partially empty, the agent sees reaching the environment limits as a tool for picking a piece, because every time that the robot reaches the limits, the optimal point of starting is calculated.

Conclusions and Future Work

In the previous section we talked about the results, but the most important thing that we have to do when analysing the results of a project is compare the results with the objectives. Just to remember, the objectives of the project were the following:

- 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.
- Create a functional system that can be continued and that delivers the first results.
- The system should empty the box with a rhythm of 2 pieces per minute.

It is important to have in mind that the project is not a fully functional system. We couldn't probably sell the product to anyone in this state, but taking into account the complexity of the project, that we were starting from zero and th results obtained, I would say that we have reached the expectations.

Analysing the objectives, we have implemented a simple and complex bin picking solution, studying and adding new technologies such as the Block Detector or Prior Knowledge. Besides, we have implemented a functional system that has to be improved, but it works in a good way, improving the objective of picking 2 pieces per minute in almost 20 seconds.

This last objective may not seem a lot, but it is important to take into account that the system has been implemented in a real robot, and just the pick and place action can take about 30 seconds to be taken.

During the whole project we have passed through a lot of adversities. It was difficult to make the communication between ROS nodes work, the development of the home made gripper took more time than expected, there were

Table 1: *Example table*

Name		
First name	Last Name	Grade
John	Doe	7.5
Richard	Miles	2

a national lock down in France that last over 2 months... In every moment we knew that we were on a project and that all these things could happen. If we were locked i some task, we managed to find a complementary task in order to not being stopped.

In my opinion, we have done a really good job and we have proven that Reinforcement Learning is a good solution for this kind of problems. The solution implemented is still far from its final version, but we have created a really good starting point for other people. I am glad that the project will be followed in the following months and it will probably reach a much better state.

The solution implemented was working, but I think that there is still a lot of work to do. For example, the agent was learning that when the background of the images was white (No pieces in the box), he shouldn't perform a pick action. We tested for a long time, and if there were no pieces in the box, the robot was rarely performing pick actions, coinciding with the minimum random action ratio.

However, in my opinion, the agent wasn't really detecting the best place for taking a pick action, but the worst, and it was performing pick actions in the rest of the places. There is a lot of work to do on fine tuning the Algorithm to find its optimal state. Probably with hours and hours of training it will be possible.

There are other important things to do in the project, as trying to generalize the model in order to be able to pick not only the wooden squares but any other kind of objects. In order to do so, it will take a lot of hours of training with some different objects.

$$e = mc^2 \tag{5}$$

References

[Figueredo and Wolf, 2009] Figueredo, A. J. and Wolf, P. S. A. (2009). Assortative pairing and life history strategy - a cross-cultural study. *Human Nature*, 20:317–330.