UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

End-to-End Visual Obstacle Avoidance for a Robotic Manipulator using Deep Reinforcement Learning

Felipe Padula Sanches

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

ICMC USP
SÃO CARLOS

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: ______________________

Felipe Padula Sanches

End-to-End Visual Obstacle Avoidance for a Robotic Manipulator using Deep Reinforcement Learning

Dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP – in accordance with the requirements of the Computer and Mathematical Sciences Graduate Program, for the degree of Master in Science. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Profa. Dra. Roseli Aparecida Francelin Romero

USP – São Carlos
August 2021

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

```
        Sanches, Felipe Padula
S211e      End-to-End Visual Obstacle Avoidance for a
        Robotic Manipulator using Deep Reinforcement
        Learning / Felipe Padula Sanches; orientadora
        Roseli Aparecida Francelin Romero. -- São Carlos,
        2021.
           85 p.


           Dissertação (Mestrado - Programa de Pós-Graduação
        em Ciências de Computação e Matemática
        Computacional) -- Instituto de Ciências Matemáticas
        e de Computação, Universidade de São Paulo, 2021.


           1. motion control. 2. robotic manipulators. 3.
        robot vision. 4. deep reinforcement learning. 5.
        obstacle avoidance. I. Aparecida Francelin Romero,
        Roseli, orient. II. Título.
```

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Felipe Padula Sanches

Desvio de Obstáculo para um Manipulador Robótico utilizando Visão e Aprendizado por Reforço Profundo Ponta-a-Ponta

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientadora: Profa. Dra. Roseli Aparecida Francelin Romero

USP – São Carlos
Agosto de 2021

I dedicate this work to my family,
who helped me to be the person I am today.

ACKNOWLEDGEMENTS

First, I would like to thank my family for all their support throughout the years. Even when being so far away, you were always by my side. I would also like to thank my partner, Mariana, for the love, companionship, and support given during my journey. It is a privilege having you by my side.

I want to thank my advisor, Professor Dr. Roseli Aparecida Francelin Romero, for believing in me and my work since the beginning, giving me support and confidence to pursue my goals.

Also, I would like to thank the members of my qualification exam, Prof. Dr. Valdir Grassi Junior and Researcher Dr. Josué Ramos, for the suggestions and constructive criticism, which were fundamental for this work's development.

I also want to thank my laboratory partners, especially José Pedro and Iury Batista, for their friendship and support, even during this very difficult time of isolation. Our daily conversations brought a bit of the laboratory into my home and made my days more enjoyable.

Additionally, I want to thank the friends I made during my journey at the ICMC, having fond memories that I will carry with me forever. Likewise, my childhood friends, whose bonds of friendship endure despite the time.

And last but not least, I want to thank all the research funding agencies, specially CNPq, CAPES, and FAPESP, for funding this research and many others, essential for the development of science in Brazil.

"I would rather have questions that can't
be answered than answers that can't be questioned."
(Richard Feynman)

ABSTRACT

SANCHES, F. P. **End-to-End Visual Obstacle Avoidance for a Robotic Manipulator using Deep Reinforcement Learning**. 2021. 85 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

Recent changes in industrial paradigms enforce that robots must be intelligent and capable of decision-making. Robotic manipulators need to satisfy many requirements for operating properly. Perhaps the most fundamental one is the capability of operating in its environment without collisions. In this work, we perform visual obstacle avoidance on goal-reaching tasks of a robotic manipulator using an end-to-end Deep Reinforcement Learning model. The motion control policy is responsible for reaching a target position while at the same time avoiding an obstacle positioned randomly in the scene. This policy uses vision and proprioceptive sensor data to operate. We train the reinforcement learning agent using Twin-Delayed DDPG (TD3) algorithm in a simulated environment, utilizing the Unity game engine and the ML-Agents toolkit. Experiments demonstrate that the agent can successfully learn a meaningful policy to avoid obstacles using images.

Keywords: motion control, robot manipulators, robot vision, deep reinforcement learning, obstacle avoidance.

RESUMO

SANCHES, F. P. **Desvio de Obstáculo para um Manipulador Robótico utilizando Visão e Aprendizado por Reforço Profundo Ponta-a-Ponta**. 2021. 85 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2021.

Mudanças recentes nos paradigmas industriais esperam que os robôs sejam inteligentes e capazes de tomar decisões. Os manipuladores robóticos precisam satisfazer muitos requisitos para operar adequadamente. Talvez o mais fundamental seja a capacidade de operar em seu ambiente sem colisões. Neste trabalho, evitamos obstáculos visuais em tarefas de alcance de meta de um manipulador robótico usando um modelo de Aprendizado por Reforço Profundo de ponta-a-ponta. A política de controle de movimento é responsável por atingir uma posição alvo e, ao mesmo tempo, evitar um obstáculo posicionado aleatoriamente na cena. Esta política usa dados de sensores proprioceptivos e de visão para operar. O agente de aprendizagem por reforço foi treinado através do algoritmo Twin-Delayed DDPG (TD3) em um ambiente simulado, utilizando a game engine Unity e o framework ML-Agents. Experimentos demonstram que o agente pode aprender com sucesso uma política significativa para evitar obstáculos usando imagens.

Palavras-chave: controle de movimento, manipuladores robóticos, visão robótica, aprendizado por reforço profundo, desvio de obstáculos.

LIST OF FIGURES

LIST OF CHARTS

LIST OF ALGORITHMS

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

β-VAE	β Variational Auto-Encoder
ANN	Artificial Neural Networks
CNN	Convolutional Neural Networks
DDPG	Deep Deterministic Policy Gradients
DL	Deep Learning
DRL	Deep Reinforcement Learning
IK	Inverse Kinematics
MDP	Markov Decision Processes
ML	Machine Learning
MLP	Multilayer Perceptron
PG	Policy gradient
pHRI	physical Human-Robot Interaction
RL	Reinforcement Learning
SGD	Stochastic Gradient Descent
TD3	Twin-Delayed Deep Deterministic Policy Gradients

CONTENTS

CHAPTER

1

INTRODUCTION

Industry, especially the automotive, has been utilizing robotic manipulators for decades. These types of robots have very appealing characteristics to a production line, such as high precision, repeatability, and speed. However, over the last few years, their usage has increased in other industries and fields. This increase is especially true after Industry 4.0. The 4.0 Industry, or Fourth Industrial Revolution, refers to a new paradigm of production that aims to create smart factories, where all of its elements work in interaction, integrating techniques such as, among others, robotics and artificial intelligence (TJAHJONO *et al.*, 2017). As stated in Benotsmane, Dudás and Kovács (2018):

> "...Industry 4.0 includes the partial transfer of autonomy and autonomous decisions to cyber-physical systems and machines, applying information systems, it is a digital transformation that is changing up the manufacturing business by bringing radical changes not only to systems and processes, but also to business models and the workforce and management styles..."

Thus, it is appealing to use fields of artificial intelligence such as machine learning in robotic manipulators. One fundamental characteristic of these robots is that they must operate in their environment without colliding. These machines usually weigh several hundred, if not thousands of kilograms. As such, collisions pose financial risks by damaging other equipment and health hazards for workers. This situation is aggravated due to the non-traditional paradigm of the 4.0 Industry, which enforces the close/direct cooperation of humans and robots (Physical Human-Robot Interaction) to increase competitiveness (HUBER; WEISS, 2017).

A robotic manipulator located in a well-structured environment can operate without collisions by performing offline motion planning. In offline motion planning, the trajectory is computed before motion execution (SHILLER, 2015). Knowing where objects are positioned in the environment allows the robot to calculate collision-free trajectories in a planning stage.

However, there are many cases in which objects' positions are not known during planning time. For example, when working alongside humans, it is almost impossible to guarantee a priori collision-free behaviour (HADDADIN; LUCA; ALBU-SCHÄFFER, 2017). There are also cases where the robot may acquire new information during execution time. For example, a harvesting robot may operate in a dense canopy and only acquire new objects' positions when moving. In these cases, the robot has to adjust its motion to avoid collisions and perform a more optimal route.

Offline motion planning techniques are computationally expensive, and adapting them to incorporate changes in the environment during execution is still an ongoing research (DIANKOV; KUFFNER, 2008; HADDADIN; LUCA; ALBU-SCHÄFFER, 2017). A possible approach for incorporating a more reactive behavior in robotic motion is by using Reinforcement Learning (RL). This Machine Learning paradigm solves problems modeled as Markov Decision Processes (MDP). In this type of modeling, an agent reacts to an observation of its environment, producing a corresponding reward.

These methods have the advantage of learning a task without being explicitly programmed. Instead, the robot learns how to perform the task by interacting with the environment and receiving a reward signal. If the observations of the environment are in the form of images or high-dimensional data, an extension of RL known as Deep Reinforcement Learning (DRL) is used. DRL is a combination of RL and Deep Learning (DL) and leverages the advances in DL to extract meaningful information from high dimensional data. Tasks such as the ones tackled in (ZENG *et al.*, 2018; AKKAYA *et al.*, 2019; TAN *et al.*, 2018) shows how promising these techniques are. By using large amounts of data to learn, these agents can solve complex tasks that were unfeasible by using classical approaches.

This Master Dissertation's general objective is to use a learning-based method to perform collision avoidance using vision in a goal-reaching task for a robotic manipulator. The task consist in positioning the end-effector at a desired position, while avoiding a single obstacle placed randomly in the scene.

Our main contribution to the field is utilizing an end-to-end DRL approach to control a robotic arm in a goal-reaching task, with obstacle avoidance and using vision. Another contribution is the creation of a simulated environment to validate our approach. This environment is available to the community to be used as a test benchmark.

1.1 Specific objectives

The specific objectives are:

- develop a simulated environment that captures the problem's key features;
- model the system using DRL to perform control from images and proprioceptive sensors to find the goal avoiding obstacles randomly positioned;

- to validate the trained policy in simulation.

1.2 Organization

This work is organized as follows. In Chapter 2, it is presented the theoretical foundations related to this work. In Chapter 3, are discussed some works in literature that are relevant to this research proposal. In Chapter 4, the system proposed is detailed. In Chapter 5, are presented and discussed the results. Finally, in Chapter 6 are presented the conclusion and future works.

CHAPTER

2

THEORETICAL FOUNDATIONS

This chapter presents the theoretical foundations of this Master's Dissertation. We first give an overview of Deep Learning and Convolutional Neural Networks, and how these networks are trained using Stochastic Gradient Descent and Backpropagation. Then, an introduction of Reinforcement Learning and Policy Gradient methods is presented, as well as a type of RL methods called Actor-Critic methods. Finally, we present the concept of Deep Reinforcement Learning, which combines both DL and RL. We also present some often used algorithms, such as DDPG and TD3.

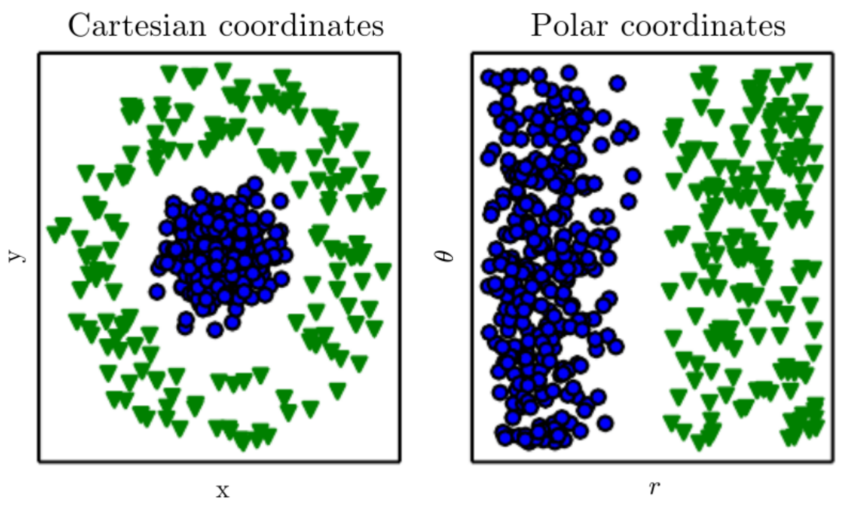
2.1 Deep Learning

Data representation is a fundamental aspect of Machine Learning (ML). It directly affects the effectiveness of an ML's algorithm. For example, consider the Perceptron algorithm, which can only classify data that is linearly separable. If we used it to classify the data from Figure 1 in cartesian form, the algorithm would not be able to classify the data properly. On the other hand, classification is trivial if the data is represented in polar coordinates.

However, it is often a difficult task to determine the appropriate way to represent data. As such, an approach to automatically learning data representations is desired. An example of such system, the Multilayer Perceptron (MLP) can be seen in Figure 2. The transformation observed in Figure 2b is the result of the input data being multiplied by the weights of each hidden neuron and being activated by a sigmoid function. The black line can now linearly separate the data. This algorithm is "the quintessential example of a deep learning model" (GOODFELLOW; BENGIO; COURVILLE, 2016).

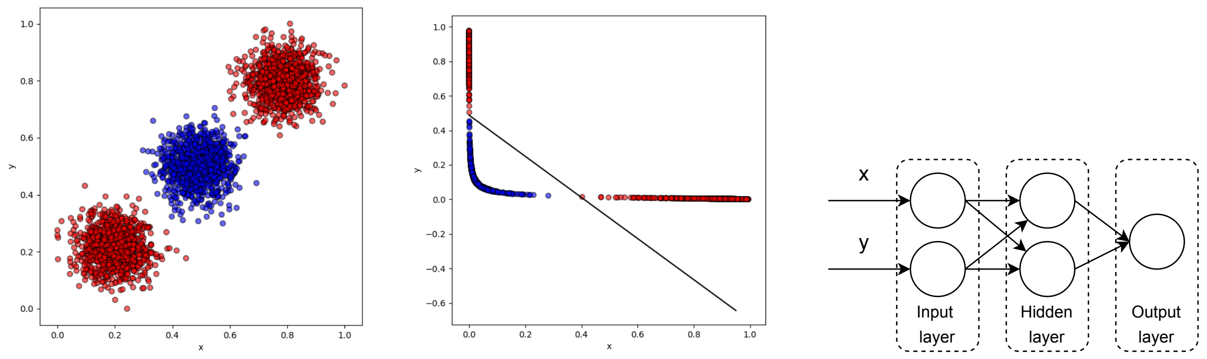
Deep Learning is, in short, a method of learning representations that are derived from other, simpler representations. This approach allows for a computer to build complex concepts incrementally, from much simpler concepts (GOODFELLOW; BENGIO; COURVILLE, 2016).

Figure 1 – Data represented in a Cartesian and polar coordinate system, respectively



Source: Goodfellow, Bengio and Courville (2016).

Figure 2 – Visual representation of Multilayer Perceptron data transformation



(a) Original data. Each color represents a class

(b) Data transformed by Multilayer Perceptron's first hidden layer. The line in black is the decision line

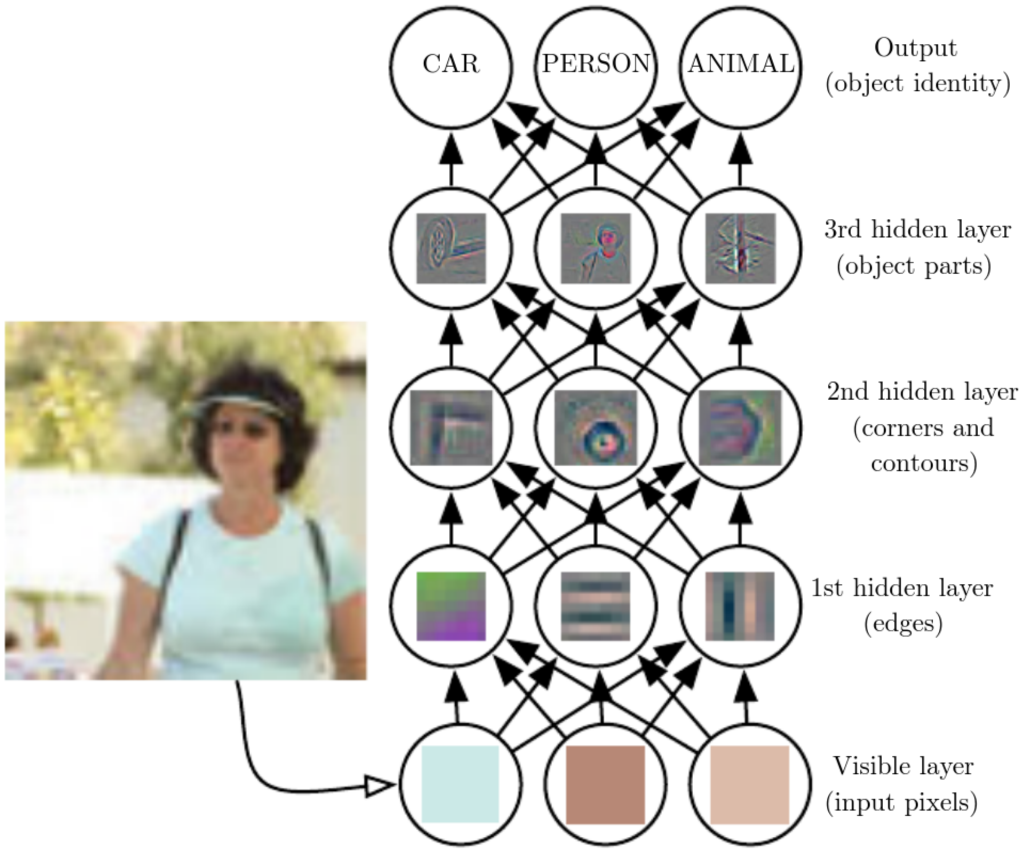
(c) A Multilayer Perceptron architecture. The activation function in the hidden layer is the sigmoid function

Source: Elaborated by the author.

Even though the presented MLP only learns one layer of representations, it is still considered a DL model, as it allows the addition of more hidden layers. Figure 3 illustrate the effect of representations that are made using simpler ones. Note that the initial layers learn representations that are, in a way, primitive. The subsequent layers build up on top of that, and the representations get progressively more complex, to the point where they can represent object parts.

These types of models often require large amounts of data to be trained. In the case of Artificial Neural Networks (ANN), the training occurs by adjusting a set of weights (parameters) so that the outputs of the ANN are closer to the desired output. These adjustments are calculated by expressing a loss function, using the expected output and the output produced by the model. For each weight at a given time step, an adjustment is made to reduce the overall error (calculated

Figure 3 – Example of representations learned in a deep learning model



Source: Goodfellow, Bengio and Courville (2016).

with the loss function). There are different ways to calculate those adjustments, but a popular one is using Stochastic Gradient Descent (SGD).

2.1.1 Stochastic Gradient Descent and Backpropagation

SGD is an extension of Gradient Descent, an optimization algorithm used to find the minimum of a differentiable function. The idea is to minimize a function by taking steps in the inverse direction of its gradient. The problem is that often the cost function used by machine learning algorithms decomposes as a sum over training examples of a per-example loss function (GOODFELLOW; BENGIO; COURVILLE, 2016). For example, the negative conditional log-likelihood cost function $J(\theta)$ over a dataset of size m, with parameter θ is given by:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(\mathbf{x}^{(i)}, y^{(i)}, \theta) \tag{2.1}$$

with L equals to the per-example loss $L(\mathbf{x}, y, \theta) = -\log p(y|x;\theta)$. For an additive cost function like that, gradient descent requires computing:

$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta L(\mathbf{x}^{(i)}, y^{(i)}, \theta) \tag{2.2}$$

In this case, each gradient descent step has a computational cost of $\mathcal{O}(m)$. DL datasets often have millions of examples, rendering Gradient Descent execution time prohibitively high. To mitigate that, SGD treats the gradient as an expectation: by taking a small set of samples (often called mini-batch), the algorithm can make an approximation good enough to perform a step. For a mini-batch of size m', it can be expressed as:

$$g = \frac{1}{m}\nabla_\theta \sum_{i=1}^{m'} L(\mathbf{x}^{(i)}, y^{(i)}, \theta) \tag{2.3}$$

Thus, for fixed model size, the cost for each gradient descent step depends only on m' and not the whole dataset. For calculating the gradient ∇_θ, the algorithm Backpropagation (RUMELHART; HINTON; WILLIAMS, 1986) is often used. It consists of expressing the outputs of a network as a composition of other functions present in the layers preceding the output. By doing this, it is possible to apply the chain rule to calculate the derivatives and therefore calculate ∇_θ. Often in the literature, the Backpropagation algorithm is used to reference the whole learning algorithm, combining both SGD and Backpropagation in just one term (GOODFELLOW; BENGIO; COURVILLE, 2016).

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) is a kind of ANN, initially proposed by LeCun *et al.* (1998) with a DL architecture for processing spatial data. The main idea behind it is the usage of the convolution operation to extract spatial information of data such as time-series data or images. They employ the convolutional operation in at least one of its layers, instead of regular matrix multiplication of regular neural networks (GOODFELLOW; BENGIO; COURVILLE, 2016). The ability to extract spatial information is fundamental in image classification, as the location of a pixel may completely change its meaning depending on where in the image it is located.

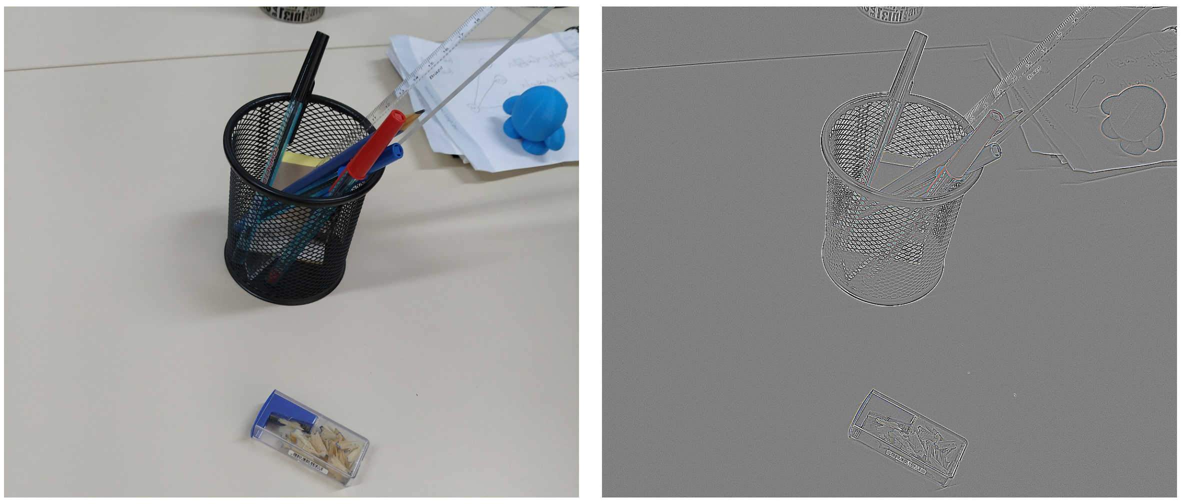
In the context of 2D data, the convolution operation (denoted here with the symbol $*$) can be described as:

$$S(i,j) = (I * J)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.4}$$

where I is a 2D image input, K is a 2D kernel and (i,j) are the pixel coordinates of where the operation is being calculated (GOODFELLOW; BENGIO; COURVILLE, 2016). An example of a convolution operation using the kernel $K = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$ can be seen in Figure 4. In CNNs, the convolution kernels are learned during the training procedure.

As every convolutional layer operates with the output of the previous layers, a composition of convolutions occurs: the first convolutional layer operates in a pixel-domain; the next one, in a convolutional space so on. Figure 3 shows an example of this operation, which results in feature maps that are increasingly more complex.

Figure 4 – Example of a convolutional operation. This kernel can be used to extract edges of an image



(a) (b)

Source: Elaborated by the author.

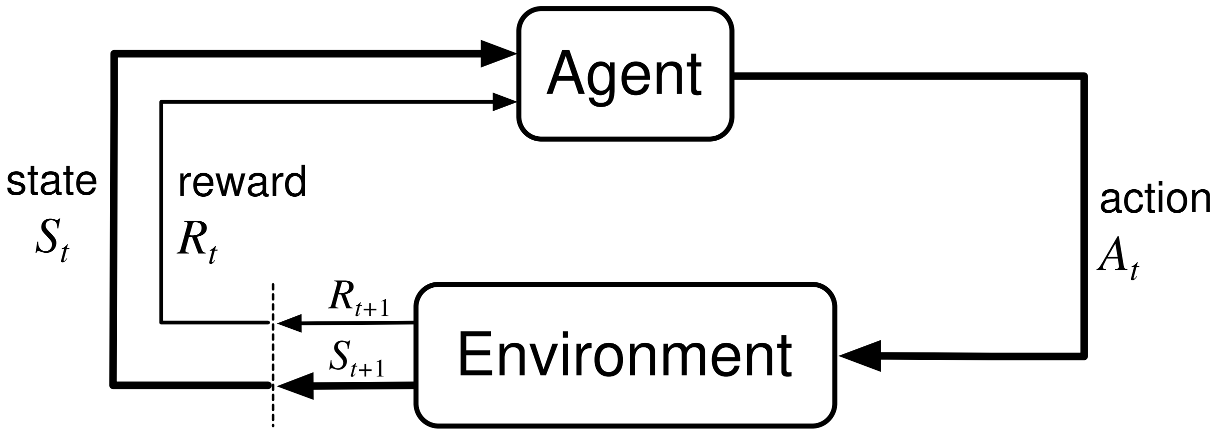
The last layer of a CNN usually is fully connected. This layer can intuitively be seen as a way of determining the role (weighting) of each characteristic (feature map) in deciding the output of the classification. A parallel can be done with the process occurring in Figure 2. Albeit much more complex in the case of CNN, both algorithms are doing the same thing: learning representations of the input data in a more meaningful way to the classification task.

2.2 Reinforcement Learning (RL)

According to Sutton and Barto (2018), "Reinforcement learning is a computational approach to understanding and automating goal-directed learning and decision making". One characteristic that separates it from other computational approaches is the emphasis on the learning by an agent, based on its interactions with the environment, not needing environment models or labeled examples. Biological learning systems inspired it, and from all forms of machine learning, it is the closest to the type of learning animals and humans do.

RL is used to solve tasks modeled as a MDP. MDP is a framework used in processes that learn from interactions in order to achieve a goal. In those processes, at a given time t an agent performs an action A_t given an observation S_t and reward R_t. This action produces a new observation S_{t+1} and reward R_{t+1}. In Figure 5 is illustrated this process. An important characteristic of these systems is that it only depends on the current environment state (S_t) to act. That means that it is reactive by nature.

Figure 5 – The interaction of the agent with an environment following a Markov decision process



Source: Sutton and Barto (2018).

RL is flexible enough to be employed in a system both at a high and low level. For example, an RL algorithm might determine the intensity of a signal that a motor has to receive. On the other hand, another RL agent can decide on what direction a robot has to move. A RL system has 6 main elements (SUTTON; BARTO, 2018):

- **Agent:** The learning and decision-making element, responsible for performing actions to interact with the environment.
- **Environment:** What the agent interacts with, comprising everything but the agent.
- **Policy:** Defines the way an agent behaves at a given time. It defines a mapping from a perceived state of the environment to an action taken in that state.
- **Model of the environment (optional):** A model of the environment allows for inferences of how the environment will behave. Given a state and an action, a model might predict the next state. This model is useful for planning, as the algorithm considers the next possible state to predict future rewards.
- **Reward signal:** The stimulus of an agent when interacting with the environment. This signal is sent to the agent on each time step. The goal of the agent is always to maximize the total reward in the long run.

- **Value function:** While the reward signal is an immediate measure, value functions specify what is good in the long run. A state value function estimates the amount of reward an agent can accumulate by starting in that state and following a determined policy. Similarly, a state-action value function estimates how good it is for an agent to perform a specific action in a state and follow a specific policy afterward. Almost all RL algorithms involve estimating value functions.

The state-value function v_π for a policy π can be defined as:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle|S_t = s\right], \forall s \in \mathcal{S} \tag{2.5}$$

where $\mathcal{S}$ is the set of states, G_t is the expected return, $\gamma \in [0,1]$ is the discount rate and R_t is the reward at time step t. Likewise, the state-action value function q_π can be defined as:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}\middle|S_t = s, A_t = a\right] \tag{2.6}$$

These functions measure quantitatively how good a given state is or how good it is to perform an action at a given state. Many RL algorithms are action-value methods. They learn the values of actions and then select actions based on these values. Their policies would not even exist without the action-value estimates (SUTTON; BARTO, 2018). However, not all algorithms are like that. Some methods instead to learn a parameterized policy that selects actions without relying on a value function. A value function may still be used to learn the policy parameters, but it is unnecessary to select an action.

2.2.1 Policy Gradient Methods

Policy gradient (PG) methods are methods that, instead of estimating value functions and generating a policy, directly estimate a parameterized policy that can select actions without consulting a value function (SUTTON; BARTO, 2018). This parametrized policy is optimized via gradient ascent. However, it is worth noting that these methods may still use a value function to learn the policy parameter, but it is not required for action selection. PG methods map a given state directly into a probability distribution of actions, given a parametrized policy π. The policy parameter vector can be represented by $\theta \in \mathbb{R}$. Therefore, the probability that action a is taken at time t with current environment state s can be written as:

$$\pi(a|s,\theta) = Pr\{A_t = a|S_t = s, \theta_t = \theta\} \tag{2.7}$$

The process of estimating the policy parameters is based on the gradient of a scalar performance measure $J(\theta)$. As the goal is to maximize performance, the updates on the parameter θ are done using gradient ascent in J:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)} \tag{2.8}$$

In this case, $\widehat{\nabla J(\theta_t)}$ is a stochastic estimate whose expectation approximates the gradient of $J(\theta_t)$. Methods that follow these general steps are called policy gradient methods, even if they also estimate a value function on their training. Methods that estimate both policy and value functions are usually called actor-critic methods. PG methods offer a few advantages over conventional value function methods, like Q-Learning, that rely on the ε-greedy policy (which is to select the action that yields the biggest reward, with ε chance of selecting a random action):

1. The approximated parametrized policy can approach a deterministic policy, whereas in ε-greedy there is always a ε chance of taking a random action
2. It enables the selection of actions with arbitrary probabilities. In some problems, the best approximate policy may be stochastic (e.g., when to bluff in Poker). Action-value methods have no natural way of finding such policies, whereas policy approximation can
3. The policy may be a simpler function to approximate than the action-value function
4. The choice of policy parametrization is sometimes a good way of injecting prior knowledge about the desired form of the policy into the RL system.

2.2.1.1 Actor-Critic Methods

The name Actor-Critic stems from the fact that these methods learn both a “actor” and a “critic” function. The critic function is responsible for estimating a value function, either the state-value function (Equation 2.5) or the state-action value function (Equation 2.6). The actor function is the parametrized policy, using the critic function to learn its parameters. The implementation from Sutton and Barto (2018) of the Episodic Actor-Critic algorithm can be seen in Algorithm 1.

Compared to vanilla Policy Gradient Methods, Actor-Critic methods usually learn faster and are easier to implement online or in continuing problems (SUTTON; BARTO, 2018). They are usually more sample efficient and have reduced variance.

2.2.1.2 On-Policy and Off-Policy Methods

There are two approaches for learning in RL: on-policy and off-policy methods. On-policy methods use the same policy to evaluate and select actions while learning. Examples of on-policy methods are SARSA, PPO, and A3C. On the other hand, off-policy methods use

Algorithm 1 – Episodic Actor-Critic

```
1: Input: a differentiable policy parameterization $\pi(a|s,\theta)$
2: Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
3: Parameters: step sizes $\alpha^{\theta} > 0, \alpha^{\mathbf{w}} > 0$
4: Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to 0)
5: while True do                                              ▷ For each episode
6:     $I \leftarrow 1$
7:     while S is not terminal do                             ▷ For each time step
8:         $A \sim \pi(\cdot|S,\theta)$                        ▷ Samples $A$ from the policy
9:         Take action $A$, observe $S', R$
10:        $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$    ▷ If $S'$ is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$
11:        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S,\mathbf{w})$
12:        $\theta \leftarrow \theta + \alpha^{\theta} I \delta \nabla \ln \pi(A|S,\theta)$
13:        $I \leftarrow \gamma I$
14:        $S \leftarrow S'$
15:    end while
16: end while
```

different policies for evaluation and action selection. Examples of off-policy methods are DQN, DDPG, and TD3. Off-policy methods are usually more data-efficient than on-policy methods, which is fundamental in computationally demanding environments.

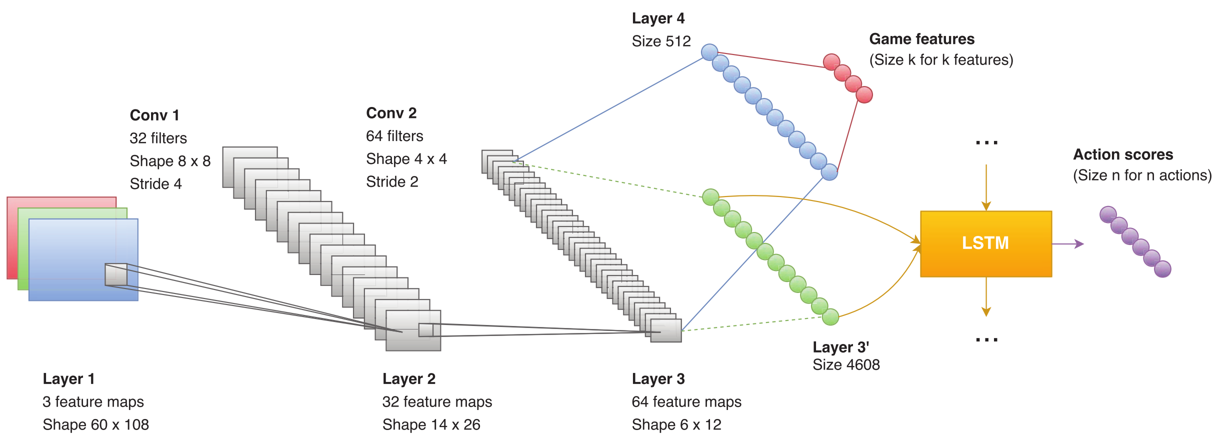
2.3 Deep Reinforcement Learning

DRL is a branch of reinforcement learning that uses DL in function approximation. As discussed in Section 2.1, DL methods can automatically learn representations of data. In an RL context, that means, for instance, that image data can be used to estimate state-value or action-state value functions directly. They can also be used to estimate parametrized policies in Policy Gradient methods. In Mnih *et al.* (2015), the authors employed a DRL architecture to train an agent to play Atari games solely based on image inputs. They used an $84 \times 84 \times 4$ image as an input to a CNN with three convolutional layers to approximate an action-state value function.

The combination of these techniques enables a much more flexible approach to problems. In Lample and Chaplot (2017), uses a combination of CNNs and LSTMs to provide context sensitivity to an RL agent that receives only an image as its input. The network architecture can be seen in Figure 6.

The input image is fed to the network, and its visual features are extracted in layers 2 and 3. Layer $3'$ only does a flattening operation, rearranging the output of the convolutional layers (64 feature maps of size 6x12 totals 4608 elements) to be fed into the LSTM layer. Then, LSTM estimates an action-state value function. One interesting aspect of their architecture is the use of an auxiliary layer: layer 4. This layer has the only function of improving the training time. The

Figure 6 – The architecture used in Lample and Chaplot (2017)



Source: Lample and Chaplot (2017).

idea here is to use values only available during training time in the simulation, like the number of enemies in the image, to force the convolutional layers to learn features that they knew were important. The error signal of the action scores was combined with the errors from the game features so that the CNN would take both into account when training. This flexibility is one of the aspects that turns DRL so powerful.

2.3.1 Deep Deterministic Policy Gradients

Deep Deterministic Policy Gradients (DDPG) is an actor-critic algorithm that uses DL for function approximations that can operate in continuous action spaces being often used for continuous control (LILLICRAP *et al.*, 2016). Its performance is comparable to planning algorithms with full knowledge of the system dynamics and derivatives. With fixed network architecture and hyper-parameters, its vanilla version could solve more than 20 simulated physics tasks, including classic problems such as cart pole swing-up, dexterous manipulation, legged locomotion, and car driving. DDPG can learn control policies directly from pixel data for some of the tasks mentioned.

This algorithm is, in short, an extension of the Deterministic Policy Gradients method (SILVER *et al.*, 2014), but uses ANN as function approximators and methods described in Mnih *et al.* (2015) to mitigate issues that arise with their use.

One of those issues is related to the assumption of most ANN learning algorithms: the samples used in training are sampled independently and identically distributed. In the context of RL, this assumption is broken because of the sequential nature of the agent's exploration of the environment.

To deal with this, DDPG uses what is called a replay buffer, which is a cache of transitions in the form (s_t, a_t, r_t, s_{t+1}) where s, a and r is the state, action, and reward respectively. This cache is filled with transitions sampled from the environment according to the exploration policy.

Both the actor and critic are updated at each timestep by uniformly sampling a mini-batch from this buffer. Another benefit of this approach is the fact that mini-batch learning uses hardware more efficiently than online learning (LILLICRAP *et al.*, 2016).

Another issue of using ANN to estimate value functions directly is that sometimes learning can be unstable. DDPG uses two auxiliary networks that are copies of the originals actor and critic networks to solve this problem, but with a slow tracking of the original network's weights. Consider the actor's network $Q(s,a|\theta^Q)$ with weights (parameters) θ^Q. The copy of this network, represented by $Q'(s,a|\theta^{Q'})$ has its weights updated as $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau \ll 1$. The same applies to the critic network. While this slows down learning considerably, this greatly outweighs the stability of learning (LILLICRAP *et al.*, 2016). Learning in continuous action space can be challenging because of exploration. To mitigate that, DDPG constructs an exploration policy μ' by adding noise sampled from a noise process $\mathcal{N}$ into the actor's policy μ:

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N} \tag{2.9}$$

Where $\mathcal{N}$ is chosen to suit the environment and can be controlled to increase exploration efficiency. DDPG is summarized in Algorithm 2.

Algorithm 2 – DDPG Algorithm

```
1:  Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s,\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$
2:  Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
3:  Initialize replay buffer $R$
4:  for episode = 1, M do
5:      Initialize a random process $\mathcal{N}$ for action exploration
6:      Receive initial observation state $s_1$
7:      for t = 1, T do
8:          Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exp. noise
9:          Execute action $a_t$ and observe reward $r_t$ and new state $s_{t+1}$
10:         Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
11:         Sample a random mini-batch of N transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
12:         Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
13:         Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
14:         Update the actor policy using the sampled policy gradient:
                $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$
15:         Update the target networks:
                $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
                $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
16:     end for
17: end for
```

2.3.2 Twin-Delayed Deep Deterministic Policy Gradients

The Twin-Delayed Deep Deterministic Policy Gradients (TD3) algorithm is an off-policy, actor-critic, RL algorithm that extends DDPG aiming to improve stability by reducing overestimated value estimates (FUJIMOTO; HOOF; MEGER, 2018). It achieves so by using two critics and taking the minimum value between them, building on Double Q-Learning (HASSELT, 2010). They also perform delayed policy updates to further improve performance.

Algorithm 3 – TD3 Algorithm

```
 1: Initialize critic networks $Q_{\theta_1}, Q_{\theta_2}$ and actor network $\pi_\phi$ with random parameters $\theta_1$, $\theta_2$, $\phi$
 2: Initialize target network $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
 3: Initialize replay buffer $\mathcal{B}$
 4: for t = 1, T do
 5:     Select action with exploration noise $a \sim \pi(s) + \varepsilon$, $\varepsilon \sim \mathcal{N}(0, \sigma)$ and observe reward $r$ and new state $s'$
 6:     Store transition tuple $(s, a, r, s')$ in $\mathcal{B}$
 7:     Sample mini-batch of $N$ transitions $(s, a, r, s')$ from $\mathcal{B}$
 8:     Set $\tilde{a} \leftarrow \pi_{\phi'}(s) + \varepsilon$, $\varepsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
 9:     Set $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
10:     Update critics $\theta_i \leftarrow \min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
11:     if $t$ mod $d$ then
12:         Update $\phi$ by the deterministic policy gradient:
                $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
13:         Update target networks:
                $\theta'_i \leftarrow \tau \theta_i + (1 - \tau)\theta'_i$
                $\phi' \leftarrow \tau \phi + (1 - \tau)\phi'$
14:     end if
15: end for
```

Deterministic policies tend to overfit to narrow peaks in value estimate (FUJIMOTO; HOOF; MEGER, 2018). To overcome this, TD3 uses a modifies the target update step, adding a small amount of random noise:

$$y = r + \gamma Q_{\theta'}(s', \pi_{\phi'}(s') + \varepsilon),\ \varepsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c) \tag{2.10}$$

This smooths the value estimate because now the fitting occurs in a small area around the target action. TD3 is summarized in Algorithm 3.

2.4 Final Considerations

This chapter presented the theoretical foundations needed to develop this work. An overview of DL was presented and a DL architecture to process spatial data, the CNNs. Next, RL and some of its core concepts were introduced. Then a class of RL methods called Policy Gradient methods was presented, and in particular, the Actor-Critic methods were discussed more in-depth, alongside the difference of On-Policy and Off-Policy methods. Finally, concepts of DRL were presented, as well as the DDPG and TD3 algorithms, which are mainly used for continuous control.

CHAPTER

3

RELATED WORKS

This chapter presents the related works to this master's dissertation. First, we present works that perform motion control using DRL. Next, we compare and discuss these works. This chapter aims to establish the state-of-art motion control using vision and DRL and determine relevant aspects that will be useful to develop our approach.

3.1 Work Selection

The usage of DRL for obstacle avoidance in robotic manipulators is relatively unexplored. As such, works that address this topic are scarce. As a result, we tried to expand our search by including works that may not completely relate to ours. We did that by selecting works that fitted at least two of the three criteria defined bellow:

1. Perform motion control using vision and DRL;
2. Perform obstacle avoidance using DRL;
3. Use DRL to perform motion control of robotic manipulators.

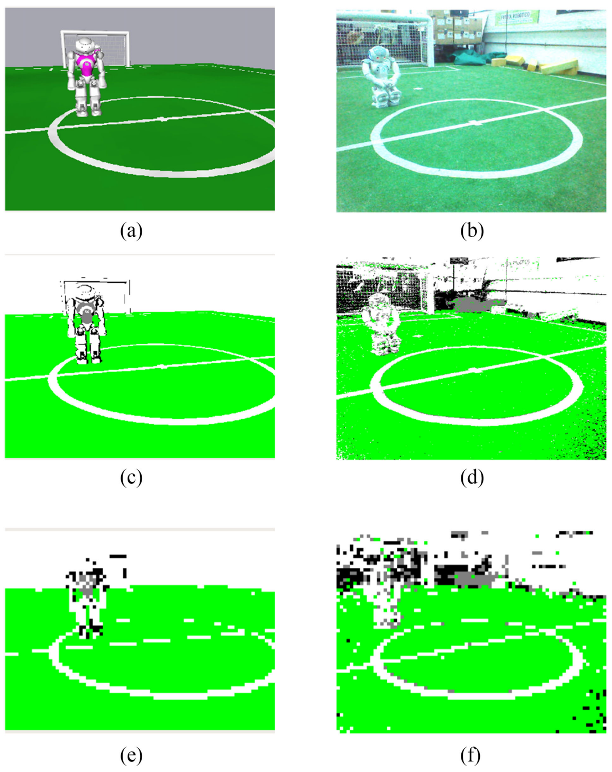
It is worth mentioning there are several works in navigation that fits items **1** and **2**. However, we chose not to include almost all of them because they often use wheeled robots, having very different dynamics when compared to manipulators.

3.2 Motion Control using Deep Reinforcement Learning

In Lobos-tsunekawa, Leiva and solar (2018) the authors used DRL to perform visual navigation of bipedal humanoid robots. As a proof of concept, the authors consider a NAO robot in a simulated robotic soccer environment. The robot's goal is to navigate to a random, predetermined location. The robot uses a camera as its main source of information to avoid obstacles placed at random positions. The camera's image does not cover the whole soccer field, being positioned in the robot's head. As such, the system is partially observed. Consequently, a MDP is not able to fully model the environment.

This issue is addressed by employing a type of recurrent neural network called LSTM (HOCHREITER; SCHMIDHUBER, 1997) to allow the system to have memory of its surroundings. This memory allows the agent to make decisions based on the current state and the previously visited states. They used DDPG as their DRL algorithm. States are formulated as the tuple $s = (s_{image}, s_{sensor}, s_{target}$, where s_{image} is a 80×60 pixels, segmented image of the environment, $s_{sensor} = (\phi_{pan}, c_{left}, c_{right})$ corresponds to sensor information with ϕ_{pan} being the robot's head pan angle, and c_{left} and c_{right} corresponding to collisions in the left and right arm respectively. Finally, s_{target} corresponds to the target's position in polar coordinates. The decision to use segmented images as input reduces the reality gap when transferring the policy trained in simulation to a real robot. By segmenting and downscaling the images passed to the model, the system can make simulation and real images more similar, as seen in Figure 7.

Figure 7 – Comparison between simulated and real images, and the segmentation system. (a) Simulated image. (b) Real image. (c) Simulated image (segmented). (d) Real image (segmented). (e) Simulated image (scaled). (f) Real image (scaled).



Source: Lobos-tsunekawa, Leiva and solar (2018).

Actions are defined as the tuple $a = (v_x, v_y, v_\theta, \phi)$, where v_x and v_y are the robot's speeds in the x and y-axis, v_θ is the rotational speed around the z-axis, and ϕ is the head's desired pan angle. The authors argue that although previous results effectively create motion policies direct from joint control, the difference between simulated and real actuators makes it difficult to transfer to real robots policies learned in simulation. As a result, the RL agent controls the robot using an omnidirectional controller rather than performing low-level joint control.

The reward function $r = -1 + \frac{v_x}{v_x^{max}} cos(\theta_{target}) \left(1 - \frac{|v_\theta|}{v_\theta^{max}}\right)$ reinforces movement towards the target as fast as possible. When a collision happens involving the robot's arm, a reward of $r = -10$ is given. Alternatively, if a body collision is detected, a reward of $r = -200$ is assigned. Finally if the robot moves to an area it cannot see in its field-of-view or if it is close to an obstacle, a reward of $r = -2$ is assigned. The episode ends if the robot reaches it target position, if a body collision is detected of if a maximum number of steps is reached.

The authors use feature augmentation to speed up training by improving data efficiency. This technique consists of using information that is only available during training to make the model's neural networks predict features of the environment in parallel to RL. By doing this, the agent can improve its performance indirectly. Feature augmentation is especially useful when dealing with images, as this helps the convolutional layers extract more meaningful representations of data. In this work, they used depth information from a 240º field of view in front of the robot, with reading every 10º, making a total of 25 augmented features.

The authors consider three evaluation metrics: episode mean reward, the mean overall speed of the agent towards the target, and the episode success rate (an episode ends successfully when the agent reaches the target). Six different experiments (summarized in Chart 1) are run in simulation, using the SimRobot simulator (RÖFER *et al.*, 2016). Here we only report the episode success rate.

Chart 1 – Experiments comparison.

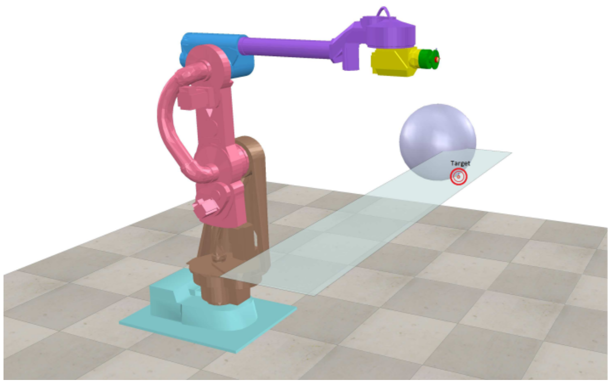
	M1	M2	M3	M4	M5	M6
Head movement	✗	✓	✗	✓	✓	✓
LSTM cells	✗	✗	✓	✓	✓	✓
F. augmentation	✗	✗	✗	✗	combined	interleaved

Source: Lobos-tsunekawa, Leiva and solar (2018).

The best performing models were M5 and M6, achieving a success rate of about 90% and 94%, respectively. The resulting visual navigation policy has an execution time of 20ms being successfully transferred to a real robot. A video showing their experiments can be seen at <https://youtu.be/Zqc6sm76ZCg>.

Sangiovanni *et al.* (2018) proposes a system using DRL to perform motion control of a robot arm with obstacle avoidance. Their goal was to design a model that could be used in physical Human-Robot Interaction (pHRI), where the robot must avoid collisions with humans in its workspace. This is a non-trivial task, as humans are generally an order of magnitude faster than typical high-gear ratio robots (HADDADIN; LUCA; ALBU-SCHÄFFER, 2017). Experiments were conducted in the V-REP simulator (ROHMER; SINGH; FREESE, 2013), where a robotic arm was placed on the floor with the target and obstacles positioned on top of a plane, to emulate a production line where it has to operate, as seen in Figure 8.

Figure 8 – Training scenario in V-REP with the virtual COMAU SMART3-S2, obstacle and target (red circle).



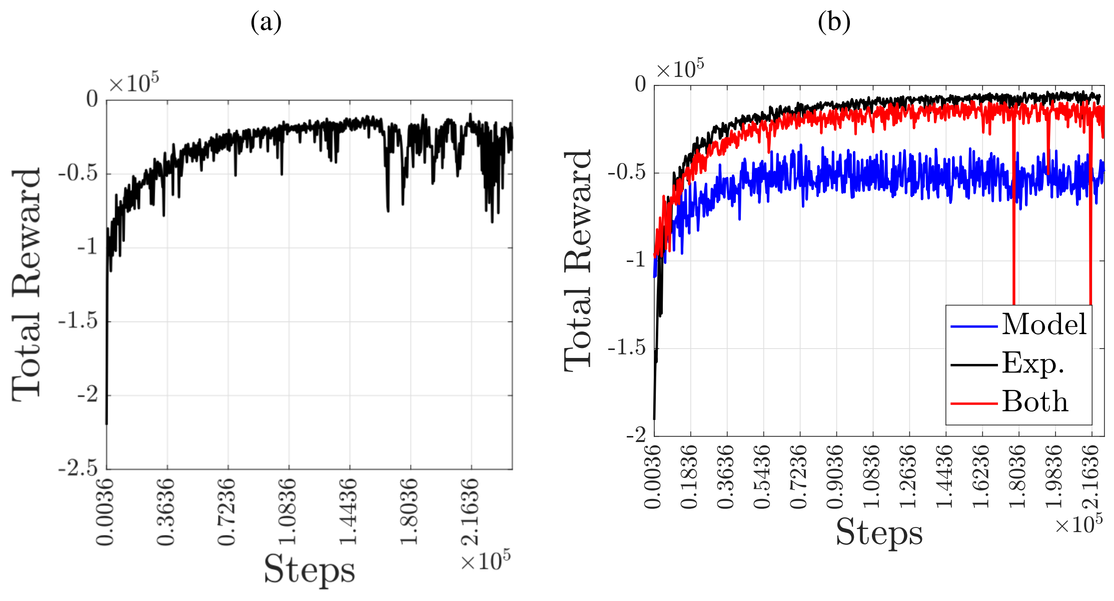
Source: Sangiovanni *et al.* (2018).

The state space is defined as $s = (q, \dot{q}, p_e, p_t, p_o, \dot{p}_o)$, where q and $\dot{q}$ are the joints' positions and velocities respectively, p_e is the end-effector position, p_t is the target position and p_o and $\dot{p}_o$ is the obstacle's positions and velocities respectively. The actions are defined as $a = (\dot{q}_{tar})$, where $\dot{q}_{tar}$ is a target rotational velocity that the joint has to reach at every step, using all available torque. The reward function is a scalar function is defined as $r = c_1R_t + c_2R_A + c_3R_O$, where R_t is the distance between the target and the end-effector, R_A is the magnitude of the actions and R_O is the shortest distance from the obstacle to the robot. They used the DQN RL algorithm with NAF (to allow the usage of continuous actions).

The authors evaluated four different scenarios: i) fixed target, moving obstacle; ii) fixed target, randomly moving obstacle; iii) random target, moving obstacle; iv) random target, random moving obstacle. The used evaluation metrics are the total reward and the distances of the end-effector to the target and the obstacle. The authors also performed experiments to evaluate the effectiveness of transferring knowledge from one task to another (transfer learning). They considered three different approaches: reusing network parameters during initialization of the networks (Model); reusing experience tuples stored in the replay buffer (Experience); parameter and tuples reutilization (Both). Results are summarized in Figure 9. Between the tested approaches, the authors concluded that reusing experience improves performance greatly.

Sangiovanni *et al.* (2020) further extends the work in Sangiovanni *et al.* (2018) by using a hybrid approach to the obstacle avoidance task, using conventional motion planning techniques and DRL. The idea is that if a metric evaluating the risk of collision is below a certain threshold, the system gives the control of the manipulator to a DRL system, which has the task of avoiding

Figure 9 – Total rewards with and without transferring learning. (a) Without transfer learning. (b) With transfer learning.



Source: Sangiovanni *et al.* (2018).

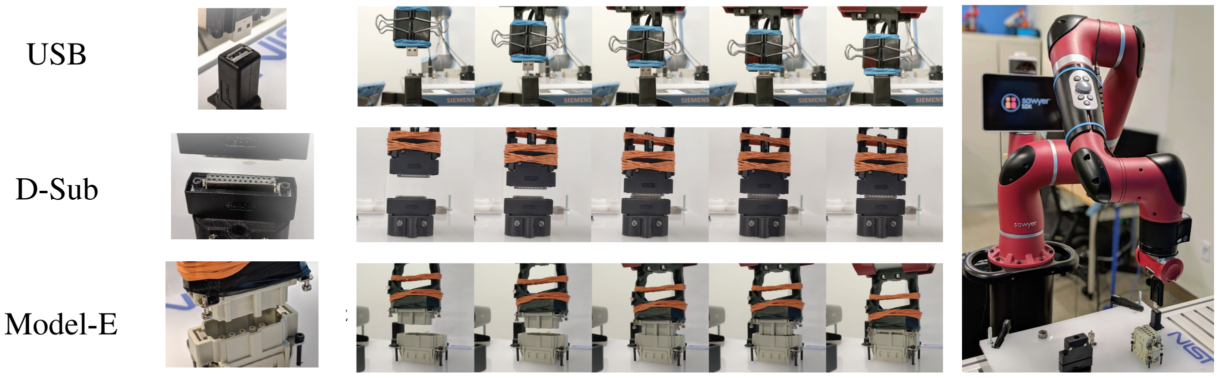
collisions. Once this condition is no longer met, the system returns the manipulator's control to the conventional motion planning module. The used metric is simply the closest distance of the robotic arm to the obstacle.

They use Single-Query Bi-Directional Probabilistic Roadmap Planner with Lazy Collision Checking for the conventional motion planning algorithm. They do not take into account the presence of obstacles, speeding up execution time. As for the RL side, the actions, states, and rewards remain fairly unchanged. A video showing the performance of the model can be seen at <https://www.youtube.com/watch?v=xU43i8mryMY>. Experiments were run in simulation considering three different scenarios: i) single obstacle, planar motion; ii) single obstacle, spatial motion; iii) multiple obstacles of different shapes, planar motion. The reported failure rates are: 0.24% for case (i), 0.31% for case (ii) and 1.35% for case (iii).

Schoettler *et al.* (2020) employs DRL to control a robotic arm for industrial insertion tasks using visual inputs and natural rewards. Connector plug insertion is challenging because errors small as $\pm$1mm can lead to failure. Also, there is significant friction between the plug and the socket, so the robot needs to learn how to apply sufficient force to overcome friction and insert the plug. Figure 10 illustrates this task.

The authors considered three configurations in their experiments. The first one uses only 32$\times$32 grayscale images as its state and specifies an image as its goal. The reward is the pixel-wise L1 distance to the given goal image. This type of reward is compelling, as in this task specifying a goal image is easy. The second one uses sparse rewards and the cartesian coordinates of the end-effector (x_t) and the vertical force currently acting on the end-effector

Figure 10 – The authors train policies directly in the real world to solve connector insertion tasks from raw pixel input and without access to ground-truth state information for reward functions. Left: top- down views of the connectors. Middle: a rollout from a learned policy that successfully completes the insertion task for each connector is shown. Right: a full view of the robot setup. Videos of the results are available at <industrial-insertion-rl.github.io>.



Source: Schoettler *et al.* (2020).

(f_z). The reward, in this case, is equal to 1 if there is an insertion signal and 0 otherwise. Finally, the last configuration uses a dense reward based on the target location x^*, being defined as

$$r_t = -\alpha \cdot \left\|x_t - x^*\right\|_1 - \frac{\beta}{(\|x_t - x^*\|_2 + \varepsilon)} - \varphi \cdot f_z,$$

where $0 < \varepsilon \ll 1$, $\alpha = 100$, $\beta = 0.002$, and $\varphi = 0.1$. States are the same as the second one. For all three configurations, actions are defined as a desired end-effector coordinate.

For the experiments, the authors evaluated the performance of each configuration with and without a ± 1mm noise at the end-effector's position. Both SAC and TD3 algorithms were evaluated. The authors also investigated the usage of Residual RL Johannink *et al.* (2019) and Learning from Demonstrations (LfD) Nair *et al.* (2018) to improve learning. All experiments were conducted on a real robot.

The visual configuration achieved a 100% success rate for the USB connection under no noise (SAC + Residual RL) and 80% with noise (SAC + Residual RL). The sparse configuration achieved a success rate of 100% under no noise (TD3 + Residual RL and SAC + Residual RL) and 84% with noise (SAC + Residual RL). Finally, the dense configuration achieved a 100% success rate under no noise (Residual RL) and 84% with noise (Residual RL).

Aljalbout *et al.* (2020) proposes a system for obstacle avoidance in a goal-reaching task using vision. The authors used a hybrid approach that consists of combining two different policies to perform the task. One policy is a baseline policy, and it is hand-crafted. This policy is responsible for moving the end-effector towards the goal position. The other policy is the reactive policy, modeled using a DRL agent responsible for avoiding obstacles. They use the RMPFlow (CHENG *et al.*, 2018) framework to combine both policies into a desired joint acceleration that is passed to the robot controller. Figure 11 shows an overview of this system.

States are defined as the tuple $s = (q, \dot{q}, x, \dot{x}, z)$, where q is the current joints positions, $\dot{q}$ the joint velocities, x the end-effector position, $\dot{x}$ the end-effector velocity, and z the latent representation of a 128×128 image obtained by a static camera positioned in the environment.

They do not pass the image directly to the RL agent. Instead, they pass a latent representation of this image, obtained using a β Variational Auto-Encoder (β-VAE). This auto-encoder is trained separately from the RL agent, using images collected from the environment. These images are generated using random motion for the arm.

Actions are defined as $a = d_q$, where d_q is a variation in joint angles. Rewards are defined as $r = r_{\text{collide}} + r_{\text{goal}} + r_{\text{dist}} + r_{\text{control}}$, where r_{collide} is a negative reward of -10 given when the robot collides with an obstacle, r_{goal} is a positive incentive of 5 given only when the agent reaches the goal, $r_{\text{dist}} = -1.6||x - x_g|| + 0.75$ is the distance reward (which rises linearly when the distance between the end-effector and the goal position decreases), and $r_{\text{control}} = -0.05||a||$ is a term responsible for pushing large actions. They used TD3 as their DRL algorithm, with Residual RL Johannink *et al.* (2019) and Prioritized Experience Replay (SCHAUL *et al.*, 2016).

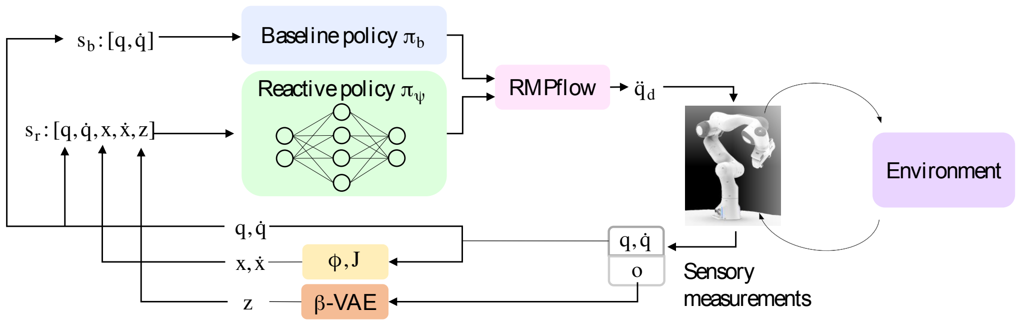
Two experiments are performed in the Gazebo simulator (KOENIG; HOWARD, 2004). Experiment A evaluates the performance of the model in an environment with only one obstacle. Additionally, the authors also compare performance without the baseline policy (Vanilla Learning - VL) and the baseline policy (Residual Policy Learning - RPL). In experiment B, the model is trained in an environment with three obstacles, and then its performance is compared with one and two obstacles. The goal here is to measure how well the trained agent generalizes. The metrics used are average episode return and success rate (when the end-effector reaches the desired goal without collisions and stays there for 5 seconds). Here we will only report success rates.

In experiment A, the proposed model (RPL) achieves a success rate of $84 \pm 6\%$ versus $39 \pm 27\%$ of the VL method. In experiment B, the model achieves a success rate of 66%, 54%, and 72% for one, two, and three obstacles, respectively. If we consider the success as the end-effector being within 7cm of the goal position, the success rates are 96%, 98%, and 94% for one, two, and three obstacles, respectively. A video showing the policies in execution can be seen at <https://drive.google.com/file/d/11C8M5m3NjMGSx_hwqmLqWTJL-etsZ55j/view>.

Zeng *et al.* (2018) uses DRL in the task of pushing and grasping objects to aid robotic manipulation. Often during a manipulation task, it is useful to push or rearrange objects to improve the positioning of the arm and fingers, as seen in Figure 12. However, determining how and when to push/rearrange objects in a non-trivial task. The authors use two fully convolutional neural networks that essentially map visual inputs to actions to solve this. One is used to map the usefulness of push actions, and the other of grasping actions. Then, the most promising action is selected.

The state s_t is modeled as an RGB-D heightmap image, captured from a fixed-mount

Figure 11 – System Overview: At a certain time step, the system receives a visual input o, joint angle measurements q, and joint velocities $\dot{q}$. The image o is encoded using a β-VAE encoder to produce the visual latent z. Using the robot kinematics ϕ and jacobian J, the authors obtain the end-effector position x and velocity $\dot{x}$ from q and $\dot{q}$ respectively. Subsequently, q and $\dot{q}$ are fed into a baseline policy to produce a user-defined behavior. Simultaneously, the authors feed all available information s_r into a reactive policy π_ψ. The latter produces a reactive behavior dependent on the objects in the environment. Both outputs are then composed together based on the RMPflow framework to produce a desired joint acceleration $\ddot{q}_d$. This acceleration is then fed into the robot controller.



Source: Aljalbout *et al.* (2020).

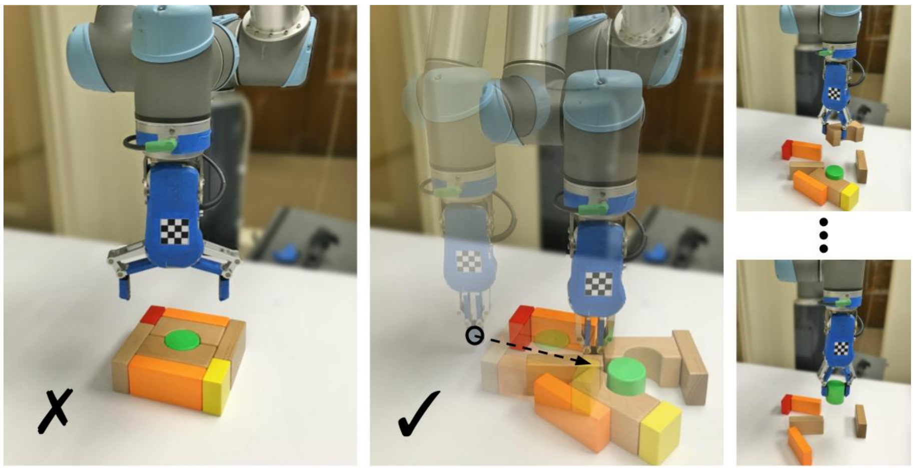
camera, with dimensions 224 ×224. This image represents the scene state at a given time t. The actions are modeled as motion primitives and divided into two types: pushing and grasping. A pushing motion is defined by a 10cm push into 16 possible directions, with the tip of the gripper, from the current end-effector position. A grasping motion consists of a parallel-jaw grasp into 16 possible orientations. During this motion, the gripper moves 3cm below its current position and then closes the gripper. For both motion primitives, IK is solved using Diankov (2010).

The output of the convolutional networks is a dense pixel-wise map of Q-values with the same size and resolution of s_t. These values indicate the future expected reward of executing a determined motion primitive at a given 3D location. To establish the next action, the agent greedily chooses the point that provides the highest Q-value. These networks were trained using an extension of DQN. To improve efficiency, the authors pre-train part of their network in the ImageNet (DENG *et al.*, 2009) dataset.

The reward signal is very simple and straightforward. If, at a given time, the robot performs a successful grasp, a reward of 1 is given. On the other hand, if the robot pushes an object and makes a detectable change in the environment (detected by the sum of differences between heightmaps), a reward of 0.5 is given.

The authors define three different metrics for the experiments. First, they measure the average completion rate over n test runs, which measures the ability of the policy to pick all objects without failing more than ten times. Second, the average grasp success rate per completion. Finally, the action efficiency $\left(\frac{\text{\# of objects in test}}{\text{\# actions before completion}}\right)$ is measured. Results were compared in simulation using two baselines, a reactive grasping-only policy (Grasping-only) and a reactive pushing and grasping policy (P+G Reactive). It is worth noting that both policies do not consider long-horizon strategies. For the real-word experiment, only the Grasping-only policy was used for comparison. The results discussed here relate to the challenging arrangement of objects (for both simulation and the real world).

Figure 12 – Example configuration of tightly packed blocks reflecting the kind of clutter that commonly appears in real-world scenarios (e.g., with stacks of books, boxes), which remains challenging for grasp-only manipulation policies. The authors' model-free system can plan pushing motions that can isolate these objects from each other, making them easier to grasp, improving the overall stability and efficiency of picking.



Source: Zeng *et al.* (2018).

For the simulation, the RL model (VPG) completion rate was 82.7% against 48.2% and 40.6% of the P+G and Grasping-only models, respectively. The VPG model's grasping success was 77.2%, whereas the P+G model was 59% and the Grasping-only 51.7%. Finally, the action efficiency of the VPG model was also superior, with a value of 60.1% against 51.7% from the Grasping-only model and 46.4% from the P+G model.

Real-word results also favored the VPG model. It had a completion rate of 71.4%, while the Grasping-only model had 42.9%. The grasp success for the VPG model was 83.3% (higher than the simulation), whereas Grasping-only was 43.5%. Finally, grasping efficiency was 69% and 43.5% for the VPG and Grasping-only models, respectively.

3.3 Comparison and discussion

The previous section presented works that performed motion control using deep reinforcement learning. Chart 2 summarizes these works. The idea was to select works that fit at least two of the three criteria: perform motion control using vision and DRL; perform obstacle avoidance using DRL; use DRL to perform motion control of robotic manipulators. The usage of DRL for obstacle avoidance in robotic manipulators is relatively unexplored. As such, works that address this topic are scarce. Although there are relatively more works in navigation involving

Chart 2 – Comparison between motion control, DRL systems.

Work	Task	Type of Robot	Type of Approach	Inputs	DRL Algorithm	Simulation Environment	Applied to Real Robot
(LOBOS-TSUNEKAWA; LEIVA; SOLAR, 2018)	Goal reaching w/ obst. avoidance	Bipedal robot	End-to-End	Images and scalar values	DDPG	SimRobot	✓
(SANGIOVANNI *et al.*, 2018)	Goal reaching w/ obst. avoidance	6 DOF robotic arm	End-to-End	Scalar values	DQN+NAF	V-REP	✗
(SANGIOVANNI *et al.*, 2020)	Goal reaching w/ obst. avoidance	6 DOF robotic arm	Hybrid	Scalar values	DQN+NAF	V-REP	✗
(SCHOETTLER *et al.*, 2020)	Industrial insertion	7 DOF robotic arm	End-to-End	Images or scalar values	SAC/TD3	-	✓
(ALJALBOUT *et al.*, 2020)	Goal reaching w/ obst. avoidance	7 DOF robotic arm	Hybrid	Images and scalar values	TD3	Gazebo	✗
(ZENG *et al.*, 2018)	Pushing and grasping objects	6 DOF robotic arm	End-to-End	Images	DQN	V-REP	✓
Ours	Goal reaching w/ obst. avoidance	7 DOF robotic arm	End-to-End	Images and scalar values	TD3	Unity	✗

Source: Research data.

obstacle avoidance and DRL, we chose not to include them as the application domain is quite different from ours.

In this sense, even though Lobos-tsunekawa, Leiva and solar (2018) uses DRL for navigation, their work deals with the navigation of a bipedal robot. Despite not controlling the joints directly, their system dynamics is significantly different from a wheeled robot. Moving in the environment requires the robot to actuate its joints in a controlled manner, to perform a determined walking gait. Moreover, they take into account transferring learning in simulation to the real world, which is crucial to advancing the usage of RL in real-world robots. Finally, they use visual inputs to control continuous variables, aligning with our goal in this work. As a result, we chose to include their work.

In their work, they used image segmentation and image downscaling to address the reality gap in the visual input of the simulated and real agents. They showed that this approach works by successfully transferring learning from simulation to the real robot.

Another interesting aspect of their work is the usage of augmented features to aid training. This technique is especially useful because this type of information can be used in other modules in the robot. During execution time, it can also be used to quantitatively measure if the agent can extract meaningful information from the image. For example, if the augmented feature that the agent is trying to predict is the obstacle position, we can calculate the error with the ground-truth position to measure the prediction quality.

Sangiovanni *et al.* (2018) presents interesting results, and the goal of their work is similar to ours. The main difference is that they do not use vision. However, it is hard to evaluate how well their model performs because the metrics chosen are difficult to interpret, given that they heavily depend on how the environment is modeled. These issues were addressed in their follow-up work, Sangiovanni *et al.* (2020) where they provided not only more meaningful metrics but also videos and images showing execution heatmaps. Their hybrid approach achieved very promising results. On the other hand, it is worth noting that their metric for evaluating when to switch policies (minimum distance between the robot arm and the obstacle) may be difficult to implement in a real-world scenario.

Schoettler *et al.* (2020) presents an important result, as they show a real-world task being solved using DRL. In addition, their pure visual approach is compelling because it does not require extra state information, and many tasks can be easily specified by directly comparing images (at least in an industrial context). They also showed that their method is efficient enough to be directly trained on a real robot, even when dealing with images.

Aljalbout *et al.* (2020) is the work that is mostly similar to ours and the only one that meets all the criteria mentioned at the beginning of this section. They achieve very promising results. However, by using a hybrid model, their approach requires extra modeling when compared to end-to-end.

Zeng *et al.* (2018) can perform a complex, hard-to-model task. Their approach has good results both in simulation and real-world. Their method is also sample efficient, as they could also train their agent in a real robot. However, it would be interesting to transfer knowledge from the simulation domain to the real world.

It is important to note that all works employed at least one strategy to improve training efficiency. Lobos-tsunekawa, Leiva and solar (2018) uses function augmentation to guide the learning of useful features in the convolutional layers. Sangiovanni *et al.* (2018) and Sangiovanni *et al.* (2020) uses transfer learning to pass knowledge from simpler tasks to more complex ones. Schoettler *et al.* (2020) uses Residual RL and Learning from Demonstrations to incorporate prior information into their model. Aljalbout *et al.* (2020) uses a β variational autoencoder to decouple the need for the agent to learn image representations. This autoencoder is trained separately from the RL agent, thus improving efficiency. They also use Prioritized Experience Replay and Residual RL. Finally, Zeng *et al.* (2018) uses transfer learning by pre-training part on the ImageNet (DENG *et al.*, 2009) dataset.

All the works but Zeng *et al.* (2018) rely on the agent reaching a position of interest. The tasks on Lobos-tsunekawa, Leiva and solar (2018), Sangiovanni *et al.* (2018), Sangiovanni *et al.* (2020) and Aljalbout *et al.* (2020) consist of reaching a target position without collisions. The task in Schoettler *et al.* (2020) is not a goal-reaching one; however, it still depends on the robot reaching a determined position before inserting the plug. As a result, the agent's notion of reaching a certain position is necessary, and it is reinforced to the agent via the reward function. This reinforcement can be done by either using a reward signal inversely proportional to the distance between the agent and the target (SANGIOVANNI *et al.*, 2018; SANGIOVANNI *et al.*, 2020; ALJALBOUT *et al.*, 2020; SCHOETTLER *et al.*, 2020) or by stimulating the agent to move with increased speed in the direction of the target (LOBOS-TSUNEKAWA; LEIVA; SOLAR, 2018).

Likewise, the signal to avoid obstacles is also passed to the agent via the reward function. This signal can be sparse, meaning that there is only a signal when the agent collides with the obstacle, or dense, meaning that the agent receives a signal relative to the obstacle before colliding. Usually, it is better and easier to specify sparse signals, as the agent is less prone to be stuck in local optima (ANDRYCHOWICZ *et al.*, 2017; VECERÍK *et al.*, 2017). However, the training of the agent is hindered. In the works involving obstacle avoidance, Lobos-tsunekawa, Leiva and solar (2018) and Aljalbout *et al.* (2020) used a sparse approach while Sangiovanni *et* *al.* (2020) and Sangiovanni *et al.* (2018) used a dense one.

Most of the works did not consider strategies to transfer learning to a real robot (sim-to-real). Even if the methods are efficient enough to be trained in a real robot, it is interesting for an approach to have the capacity of being trained in simulation. This capacity gives flexibility to the system, as well as being safer and cheaper.

By comparing the presented works, we can see that there are no standard simulators used. Likewise, there is no standard environment/setup for testing. This lack of standardization makes comparing results difficult, especially considering that RL is very susceptible to numerical variations in these environments.

3.4 Final considerations

This chapter presented the related works pertinent to this dissertation. We established some of the state-of-art systems that use DRL to perform motion control using vision. We also established some common aspects among these works, such as improving training efficiency, the lack of sim-to-real strategies, and the use of different simulation environments.

As a result, we aim to perform goal reaching with obstacle avoidance for a robotic arm using vision by using an end-to-end approach. We adopt measures to improve training efficiency. Additionally, we take into account techniques to transfer learning from simulation to the real robot. Finally, we choose a simulation environment that can be easily integrated with other tools, thus allowing comparison with future works.

CHAPTER 4

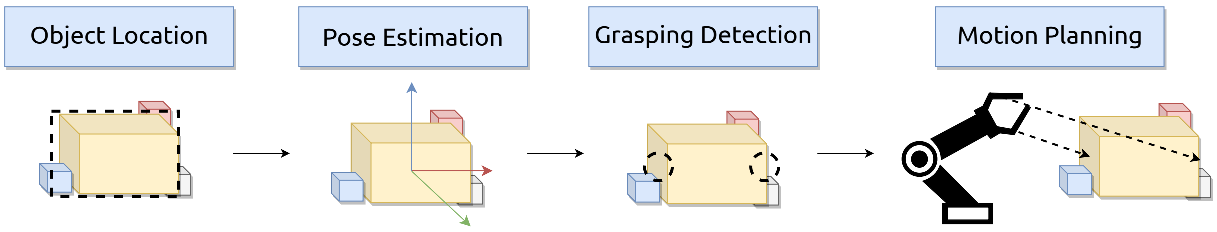
PROPOSED RESEARCH

In this chapter, we present the proposed research of this Master's thesis. Section 4.1 details how the system was modeled from a DRL perspective. Section 4.2 describes how the simulation was setup. Section 4.3 details the techniques and resources used. Finally Section 4.4 makes final considerations.

4.1 Modeling

Before presenting the developed system to control the trajectory of a manipulator avoiding obstacles, it is necessary to introduce an overview of the main tasks involved in the vision-based robotic grasping process. Figure 13 illustrate these tasks.

Figure 13 – Object grasping pipeline.



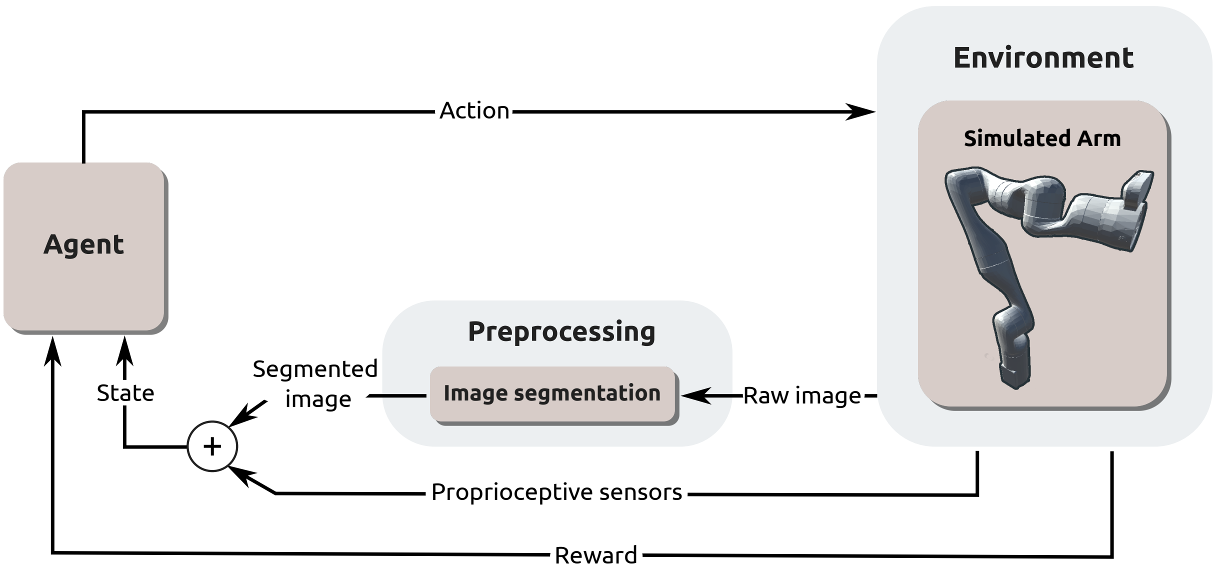
Source: Elaborated by the author.

The four general tasks in vision-based robotic grasping are, in order: object location, pose estimation, grasping detection, and motion planning. Object location consists of locating the object in the input image. Pose estimation involves detecting the position and orientation of the object to grasp. Grasping detection deals with detecting the grasping points or pose of an object in the image. Finally, motion planning deals with actually moving the manipulator's end-effector to the desired grasp points (DU; WANG; LIAN, 2019).

We propose a computational vision model to move the end-effector to a desired location, i.e., a goal-reaching task. This task consists of reaching a random target position with a 7 DOF robotic arm. When the manipulator tries to reach this target position, it needs to avoid a randomly positioned obstacle, using images from a fixed camera in the environment. The input image is segmented to aid training and transferring the learned policy to a real manipulator. Besides the segmented image, it also receives proprioceptive sensor data. The arm's starting pose is fixed, as well as the object's size and color. The camera is positioned so that the obstacle is always located inside its field of view. However, it may still get occluded by the robot arm itself. The task is successful if the end-effector is within a certain tolerance from the target position. A failure happens when a collision is detected in the arm or if a time limit is reached.

The task was formalized as an MDP, as seen in Figure 14. We chose a DRL approach as this MDP had a high-dimensional input since it receives an image from the environment. In the next sections, we describe details and considerations taken during the process of the modeling.

Figure 14 – System overview. An image is taken from environment and segmented. This segmented image, alongside proprioceptive sensor data, are used as the state of the RL agent. The agent then sends an action to the environment, which executes it. The cycle is then repeated.



Source: Elaborated by the author.

4.1.1 State space

Our initial approach was to train a separate network to predict the obstacle's position and feed it to the agent to address the visual aspect of the problem. However, similar to Lample and Chaplot (2017) we observed poor performance, and the agent was not able to learn. We also tried to separately train a β-Variational Autoencoder, as proposed in one of the approaches of Yarats *et al.* (2019) but observed poor results. This bad performance reinforces the notion that the convolutional layers must be shared with the agent, i.e., signals from the RL step must be propagated to the convolutional layers. The authors of both these works also came to this conclusion.

In the early iterations of our model, we used the end-effector's camera as the visual input. However, this leads to several problems. One of them is that the image does not fully capture the environment information, i.e., the camera's image partially observes the environment. The agent would need a type of memorization of the environment to address that. This memorization can be achieved using recurrent networks (HAUSKNECHT; STONE, 2015), at the cost of increased model complexity and training time. Having the camera in the end-effector also meant that the agent would need to learn how to perform actions to look in its environment, adding another layer of complexity to the problem. After many preliminary tests, the agent could not learn a meaningful policy; thus, we opted for a fixed camera in the environment.

In the end, the state space is composed of four main terms, each one modeling a different aspect of the environment. It is defined as:

$$s = (s_{agent}, s_{camera}, p_{target}, s_{arm}) \tag{4.1}$$

where $s_{agent} = (p_{agent}, r_{agent})$ is respectively the agent's position and orientation, s_{camera} is a 64×64 grayscale image captured by a fixed camera in the environment, p_{target} is the target position and $s_{arm} = (p_{ee}, r_{ee}, \theta_{joints})$ is, respectively, the end-effector's position and orientation, and the arm's joint angles. All positions are represented in cartesian coordinates and orientations as quaternions.

Images generated in a simulated environment usually considerably differ from the real world. This difference makes it hard to transfer policies trained using images from simulation to real-world robots. Although it is possible to improve the simulated image's quality to mimic the real world, this usually increases computational costs. Those costs are prohibitive in a DRL context since these methods are already computationally expensive and often data inefficient.

Works like Lobos-tsunekawa, Leiva and solar (2018) try to mitigate this by segmenting and downscaling images before feeding them to the model. These procedures effectively remove information from both the real and simulated images to make them more similar.

Image segmentation also alleviates training by providing class information to the model, removing the necessity of the model to perform any type of internal classification. For example, the agent at first may not know that it has to avoid objects of a certain class, but it already has class information of objects in its visual field. Segmented images can also be represented in simpler and more efficient structures; C different classes can be represented with $\left\lceil \log_2^C \right\rceil$ bits without loss of precision. Segmentation, as well as downscaling, reduces overall execution time for the models.

Thus, aiming for future implementation in a real robot, the camera image is preprocessed before being passed to the model. We argue that performing image segmentation on the real robot is a feasible task, given the impressive performance of recent approaches (MINAEE *et al.*, 2021). In addition, image segmentation is trivial in simulation. Therefore, in our model, the image is reduced to a 64×64 resolution and segmented according to objects classes before being passed to the agent. An example of an image input passed to the agent can be seen in Figure 19b.

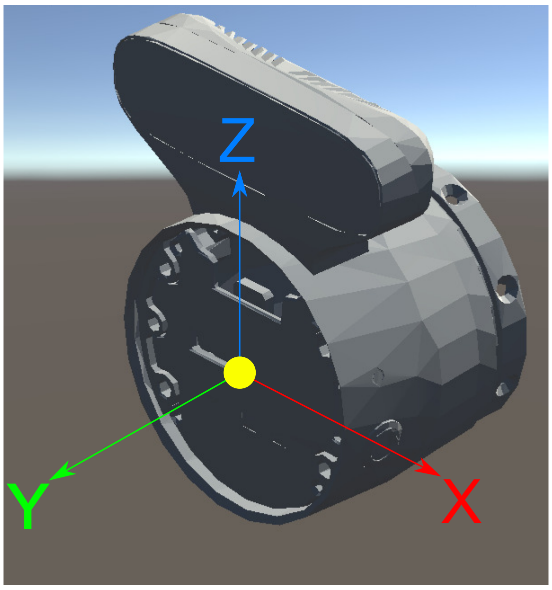
4.1.2 Actions

An action is defined as the tuple:

$$a = (\theta_{yaw}, \theta_{pitch}, \theta_{roll}, \Delta x, \Delta y, \Delta z) \tag{4.2}$$

where θ_{yaw} is a rotation along the agent's Z axis, θ_{pitch} is a rotation along the X axis, θ_{roll} is a rotation along the Y axis, and $\Delta x, \Delta y, \Delta z$ are displacements in the agent's X, Y and Z directions, respectively, with $\Delta x, \Delta y, \Delta z \in [-0.01m, 0.01m]$ and $\theta_{yaw}, \theta_{pitch}, \theta_{roll} \in [-2°, 2°]$. The agent's reference frame can be seen in Figure 15.

Figure 15 – The agent's reference frame.



Source: Elaborated by the author.

The agent and the end-effector are two separate entities. The actions directly affect the agent's position and orientation, but not the end-effector's. After having its position and orientation changed by the action performed, the agent requests its current pose for the robotic arm controller. Figure 16 illustrates this process. There are three possibles outcomes for this request: the controller finds a Inverse Kinematics (IK) solution and reaches the desired pose, the controller finds an IK solution but fails to reach the desired pose, and the controller fails to find an IK solution to the desired pose.

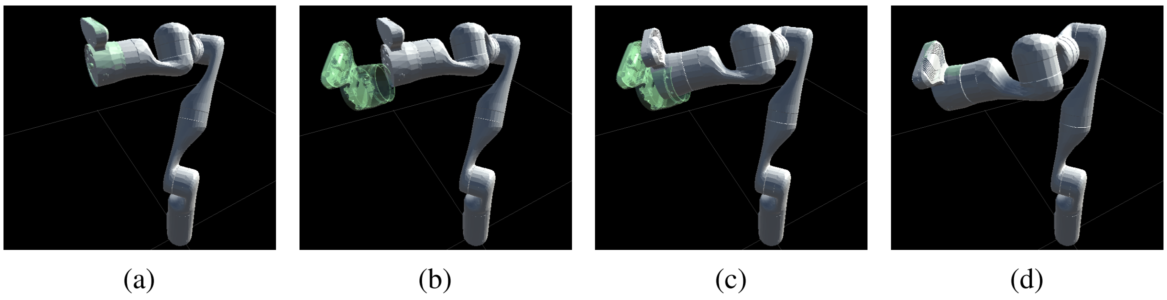
If the controller can find a solution for the desired pose, the environment waits for the arm to reach it before returning a reward and a new observation to the agent. If the arm fails to reach the desired pose by colliding with itself, the floor, or the obstacle, the episode ends.

If no IK solution is found, the agent can continue exploring the environment in a decoupled state while the arm remains in the last valid request pose. This decoupled state means that the agent and the end-effector have different poses. If the agent requests a valid pose while in the decoupled state, the arm will try to reach it.

Preliminary tests showed drastic improvements by enabling the agent to explore while in this state, rather than only allowing it to have valid IK poses. On the other hand, other problems emerged. The arm may collide during its travel if the agent, while in the decoupled state, requests a pose that is too distant (either cartesian distance or joint-space distance) from the current end-effector pose.

Therefore, we limit the maximum distance concerning the agent that the arm is allowed to reach. This limit provides a trade-off between the agent's ability to explore and blind arm's movements (i.e., movements in which the obstacles are not taken into account). Also, this behavior is discouraged by the reward function, as discussed in the next section.

Figure 16 – Agent (represented in green) requesting a pose to the arm controller. (a) Initial pose. (b) Requested pose. (c) The arm moving to target pose. (d) The arm reaches the requested pose.



Source: Elaborated by the author.

4.1.3 Rewards

The reward function is comprised of three terms, each one capturing an essential aspect of the environment. It was mainly based on Sangiovanni *et al.* (2018), with the addition of a new term, $d_{decoupling}$. The function is defined as follows:

$$r = -c_1 R_{target} - c_2 R_{obstacle} - c_3 d_{decoupling} \tag{4.3}$$

Here c_1, c_2 and c_3 are positive constants, and are used to express the importance of each term during training. The term R_{target} is the reward signal the agent receives from the target position and corresponds to the Huber-Loss function between the end-effector and the target position. It is defined as:

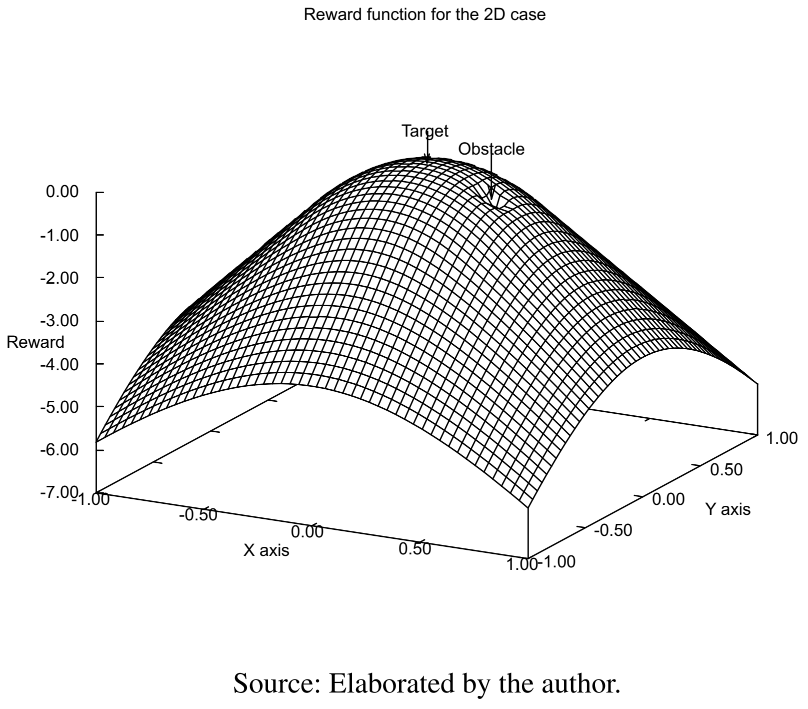
$$R_{target} = L_\delta(d_{target}) = \begin{cases} \frac{d_{target}^2}{2}, & \text{if } |d_{target}| \leq \delta \\ \delta(|d_{target}| - \frac{\delta}{2}), & \text{otherwise} \end{cases} \tag{4.4}$$

where d_{target} is the end-effector's euclidean distance to the target and δ is a positive constant. The term $R_{obstacle}$ relates to the obstacle and is calculated as:

$$R_{obstacle} = \left(\frac{d_{ref}}{d_{obstacle} + d_{ref}} \right)^p \tag{4.5}$$

with d_{ref} and p being positive constants. The term $d_{decoupling}$ is the euclidean distance between the agent and the end-effector. It discourages the agent from exploring too far from the end-effector. Note that if the agent is not in a decoupled state, this term is effectively 0. A simplified version of the reward function can be seen in Figure 17

Figure 17 – Simplified version of the reward function. Here we assume $c_1 = 10$, $c_2 = 1$, $c_3 = 0$, $\delta = 0.5$, $d_{ref} = 0.2$, $p = 8$ and that the agent, target and obstacle lie in the same Z coordinate.



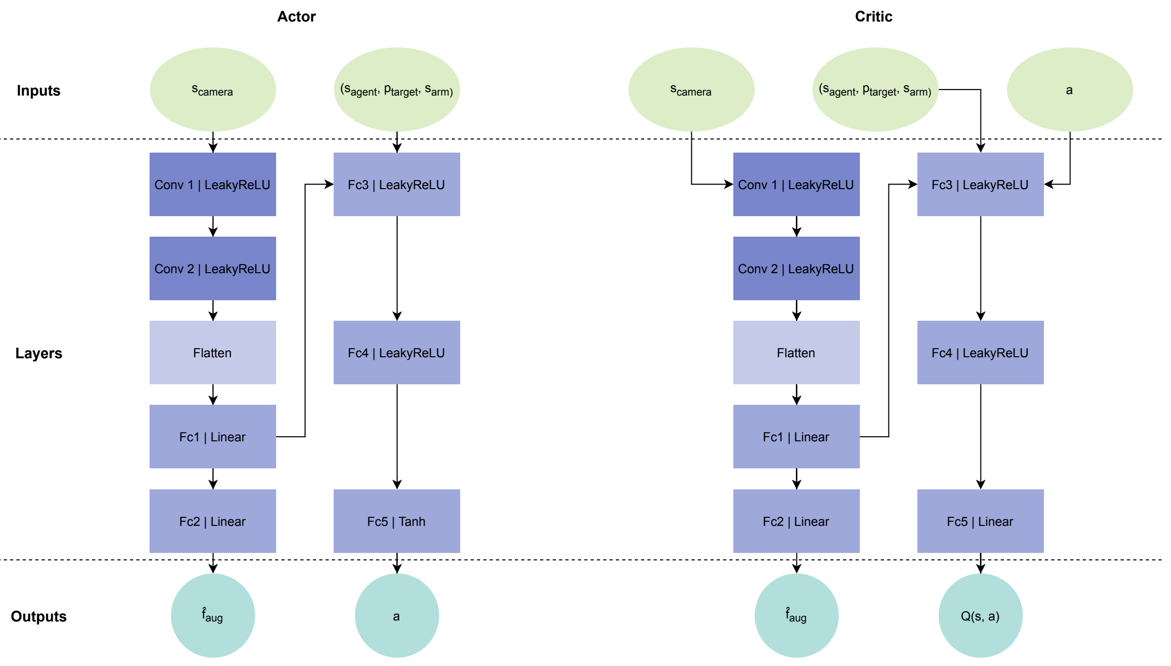
Source: Elaborated by the author.

There are also two special cases for the reward function. In case of a collision with an obstacle, the agent receives a reward of $r = -15$, and the episode ends. Likewise, if the agent can reach the target, it receives a reward of $r = 20$.

4.1.4 Network Architecture

The actor and critic networks' overall architecture is shown in Figure 18. These architectures were loosely based on Lobos-tsunekawa, Leiva and solar (2018). The convolutional layer Conv1 uses a 4×4 kernel, stride 4, and 16 output channels. Conv2 uses a 4×4 kernel, stride 2, and 32 output channels. Both of these layers use the LeakyReLU activation function. The fully connected layer Fc1 consists of 128 units and uses linear activation. The fully connected Fc2 layer has three units and linear activation. Layers Fc3 and Fc4 both have 256 units and LeakyReLU as their activation function. The Fc5 layer in the actor's network has six units and uses the hyperbolic tangent as its action function to normalize its outputs between -1 and 1. The Fc5 layer has 1 unit for the critic network and uses linear activation, as the output values need to vary in range. In total, the actor's network has 315961 parameters and the critic's 316212.

Figure 18 – Illustration of the Actor and Critic network architectures. The image is propagated through the Conv1 and Conv2 layers. The output is fed into the Fc1 layer, then directed to the Fc2 layer to estimate the augmented features. Simultaneously, the Fc1 output is also used as input to the remaining fully connected layers Fc4 and Fc5, alongside the numeric state variables (and the action values in the critic network)



Source: Elaborated by the author.

When receiving a state from the environment, the network first propagates the visual input (s_{camera}) through the convolutional layers Conv1 and Conv2. After the flattening operation, the output of the convolutional layers is fed into the Fc1 layer. The Fc1 layer's output is then concatenated with the rest of the inputs, namely s_{agent}, p_{target} and s_{arm} (and the action a in case of the critic network). Then these values are propagated through the rest of the network's layers, Fc3, Fc4, and Fc5, to generate an output. Note that after propagating the inputs through the Fc1 layer, its output is propagated through the Fc2 layer to generate the estimated values for the function augmentation variables.

4.1.5 Feature Augmentation

Feature augmentation consists of using information that is not directly available to the agent to alleviate training and improve data efficiency, as shown in Lample and Chaplot (2017). It is useful when dealing with images, as it can aid the agent in learning a meaningful representation of visual input. While learning how to estimate the augmented features, the agent indirectly learns representations useful for the problem at hand. The augmented features f_{aug} used in this work consist of the obstacle position p_{obst}.

During training, the agent receives the camera image and propagates it through the convolutional layers Conv1 and Conv2 and then through fully connected Fc1 and Fc2 layers. The Mean Squared Error loss is calculated between the estimated augmented features $\hat{f}_{aug} = \hat{p}_{obst}$ and the augmented features $f_{aug} = p_{obst}$ and the weights of the layers Fc2, Fc1, Conv2, and Conv1 are updated using Backpropagation. These updates are performed before every RL update.

It is worth noting that the agent does not have access to p_{obst}, and this branch of execution is only run during training. However, after the agent has been trained, $\hat{f}_{aug}$ can be used if needed to produce a meaningful estimation of p_{obst}.

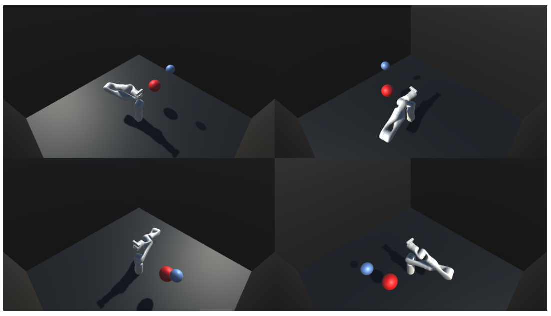
4.2 Simulation Setup

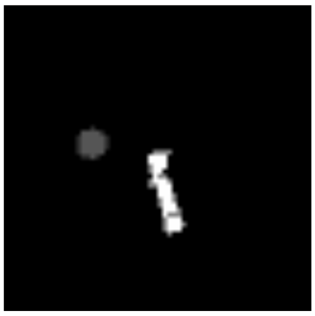
The simulation uses a 7 DOF Kinova Gen3[1] robotic arm, centered on the bottom of a cube with 2m sides. This cube represents the environment of the agent. The arm has fixed initial joint positions, shown in Figure 19a. Both the target's and obstacle's positions are randomized at the start of every episode. However, to ensure that these positions are valid and that the arm can reach its goal without colliding, they must be carefully chosen. The target position is determined using a spherical coordinate system with radius r_{target}, inclination θ_{target} and azimuth φ_{target}. The origin of the system is located on the end-effector's initial position. Table 1 shows the allowed range of values for these variables.

The obstacle is represented by a sphere and has a fixed diameter of 15cm. Initial tests showed that its positioning was directly related to the quality of the learned policy. Our initial approach was to position the obstacle at a random point, with a distance of $d_{obstacle}$ of the agent's initial position. This point is located along the line between the agent's initial position and the target position. If the agent traveled the shortest path possible (a line) between its initial position and the target position, it would always result in a collision. However, this approach resulted in the agent exploiting curves; instead of effectively avoiding the obstacle, the agent would always

[1] <https://www.kinovarobotics.com/en/products/robotic-arms/gen3-ultra-lightweight-robot>

Figure 19 – Environment overview. Note that the blue sphere (target position) is not visible to the camera.


(a) Four different views of the robotic arm.


(b) Agent's visual input.

Source: Elaborated by the author.

perform a curved motion to reach the target position, avoiding completely the region in which the obstacle was positioned.

Although the agent converged to a reasonable solution to the task, this policy was not ideal. To solve this, we created two additional variables to position the obstacle, $\theta_{obstacle}$ and $r_{obstacle}$, as seen in Figure 20. The obstacle is positioned with angle $\theta_{obstacle}$ and distance $r_{obstacle}$ to a pivot point that has distance $d_{obstacle}$ of the agent's initial position. The pivot point always lies in the line that connects the agent's initial position and the target position. These variables are uniformly sampled from a range of allowed values (shown at Table 1) at the beginning of each episode.

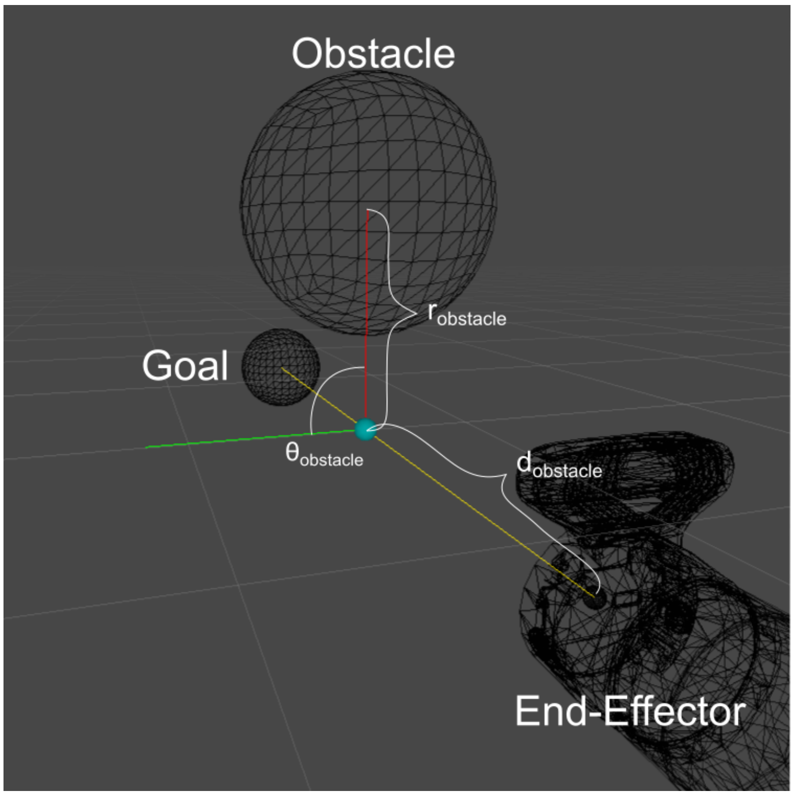
Table 1 – Range of values for the target's and obstacle's positioning variables.

Variable	Minimum Value	Maximum Value
r_{target} (cm)	45	62.5
θ_{target} (degrees)	0	100
φ_{target} (degrees)	0	180
$d_{obstacle}$ (cm)	30	47.5
$r_{obstacle}$ (cm)	0	30
$\theta_{obstacle}$ (degrees)	0	360

Following this positioning procedure, there are cases in which the obstacle may not be in the agent/arm way. If the agent performed a straight path to the goal position, there would be no collision. This lack of collision turned out to be an important scenario, as it allowed the agent to learn that the most optimal path is a straight path, as long as collision is not imminent.

The simulation consists of several environments running in parallel, as shown in Figure 22. This parallelism significantly improves data throughput and speeds up training. Preliminary tests showed that the sample rate (experiences generated per second) peaked at around 530 samples/s, with $n = 25$ parallel environments.

Figure 20 – Variables that determine how the obstacle is positioned concerning the target position and the agent's starting position. The variables $\theta_{obstacle}, r_{obstacle}$ and $d_{obstacle}$ are set to random values at the beginning of the episode. We found that obstacle positioning was imperative in learning a meaningful policy.



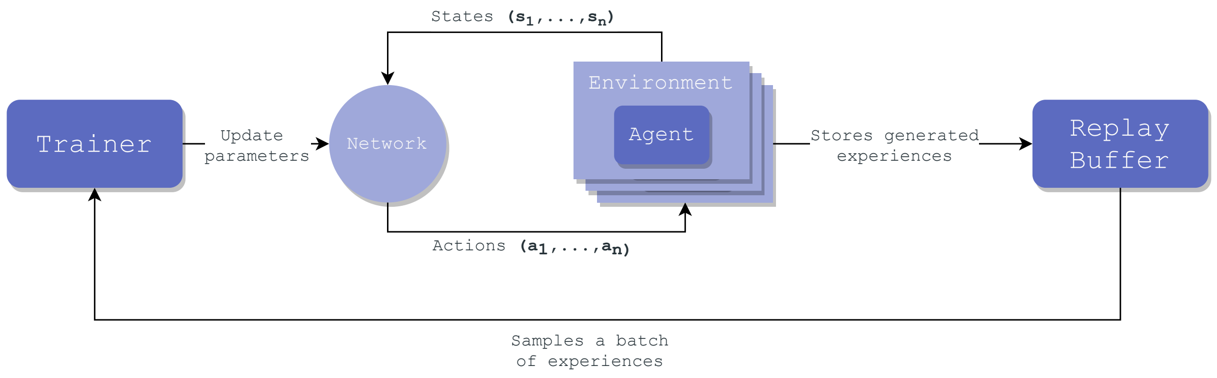
Source: Elaborated by the author.

However, it is worth noting that despite multiples environments (and consequently agents) being used, this is not a multi-agent reinforcement learning system. All agents are under the same policy, generating experience tuples to populate the replay-buffer. This approach speeds up training because the neural network frameworks are optimized to perform batches and GPU processing operations. Figures 21 and 22 illustrate this.

4.3 Techniques and Resources

The majority of this work was implemented using Python 3.7 due to its ease of integration with other tools and fast development. On the other hand, as it is an interpreted language, it may present speed limitations. However, all intensive calculations such as matrix operations were done using highly-optimized libraries such as NumPy (HARRIS *et al.*, 2020), and Pytorch (PASZKE *et al.*, 2019). The implementation of the TD3 algorithm was based on the code provided by Fujimoto, Hoof and Meger (2018).

Figure 21 – Multiple environments diagram. Here n agents propagate through a networks n observations, generating n actions. These actions when applied in the environment produce n new states, which in turn are stored in the replay buffer. Meanwhile, the network is trained using a batch of experiences sampled from the replay buffer. The experience generation and training are asynchronous.

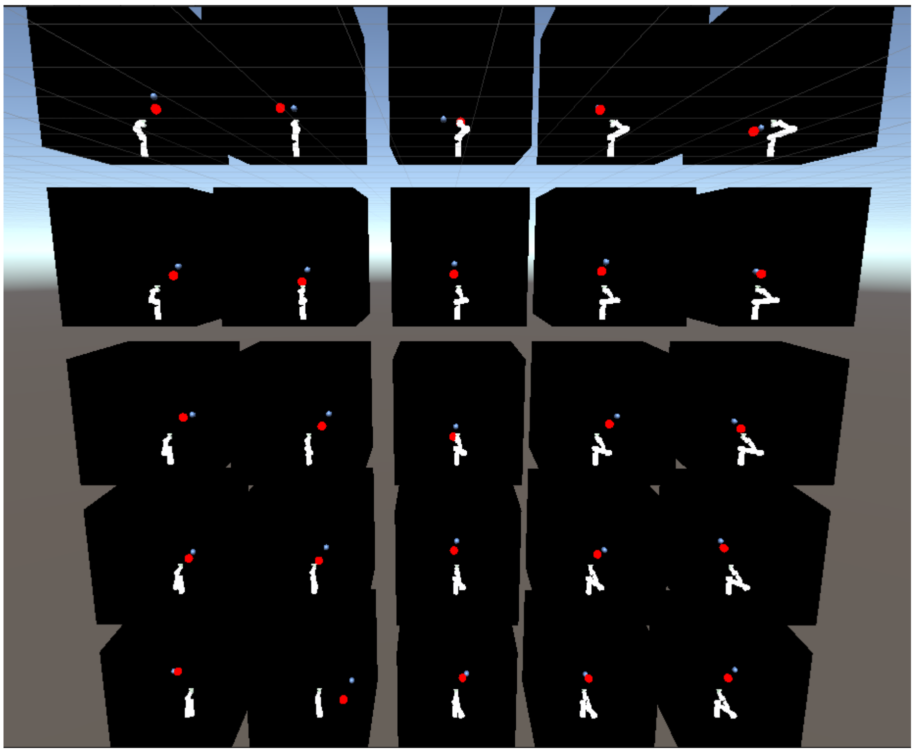


Source: Elaborated by the author.

4.3.1 Simulator

The simulation side was built using Unity (TECHNOLOGIES, 2020). Unity is a free development platform that allows fast prototyping and is extensively used for game development, and more recently, it is gaining space in robotics (TECHNOLOGIES, 2021). It uses the PhysX (SDK, 2021) that is a physics engine to simulate physics, and its latest version supports the Temporal Gauss-Seidel solver. The ML-Agents (JULIANI *et al.*, 2018) framework allows for RL's environments to be easily designed on Unity, and its Python API provides a high-level communication abstraction. This framework also facilitates the integration of the simulation with other tools by providing a Gym (BROCKMAN *et al.*, 2016) interface.

Figure 22 – Multiple environments running inside the simulation. This allows for a significant increase in data throughput.



Source: Elaborated by the author.

Source code 1 – Openai Gym being used as a wrapper for the ML-Agent’s API. Here a random policy is used to control the robotic arm.

```
from gym_unity.envs import UnityToGymWrapper
from mlagents_envs.environment import UnityEnvironment
import numpy as np

# Sample policy that performs random actions, ignoring the
 state.
def sample_policy(_):
  # We select a random number between 0 and 1 and
  # rescale between -1 and 1
  return np.random.random_sample((6,1))*2.0 - 1.0

def main():
  # Simulation path:
  env_location = "./simu_envs/Visualization/scene.x86_64"
  # Loading the unity environment:
  unity_env = UnityEnvironment(env_location)
  # Wrapping with the Gym Wrapper:
    env = UnityToGymWrapper(unity_env)

    # We reset the environment and get our initial state:
    state = env.reset()
    while True:
      # We select an action based on a policy and state:
      action = sample_policy(state)
      # We perform an action in the environment, receiving
      # the next state, the reward and a flag indicating
      # if the episode has ended:
      next_state, reward, ended, _ = env.step(action)
      # If the episode ended, we reset the environment,
      # otherwise we continue execution:
      if ended:
        state = env.reset()
      else:
        state = next_state

  if __name__ == '__main__':
    main()
```

Gym is a toolkit made by OpenAI[2] to develop and compare RL algorithms. It provides an interface to interact with an environment without requiring a fixed structure for the agent. This flexibility allows the designer to test different agents with minimal changes to the code. The Source Code 1 illustrates the simplicity of this tool. The simulation code, as well as the code for the training scripts, can be found at <https://github.com/fpadula/visualcollisionarm>.

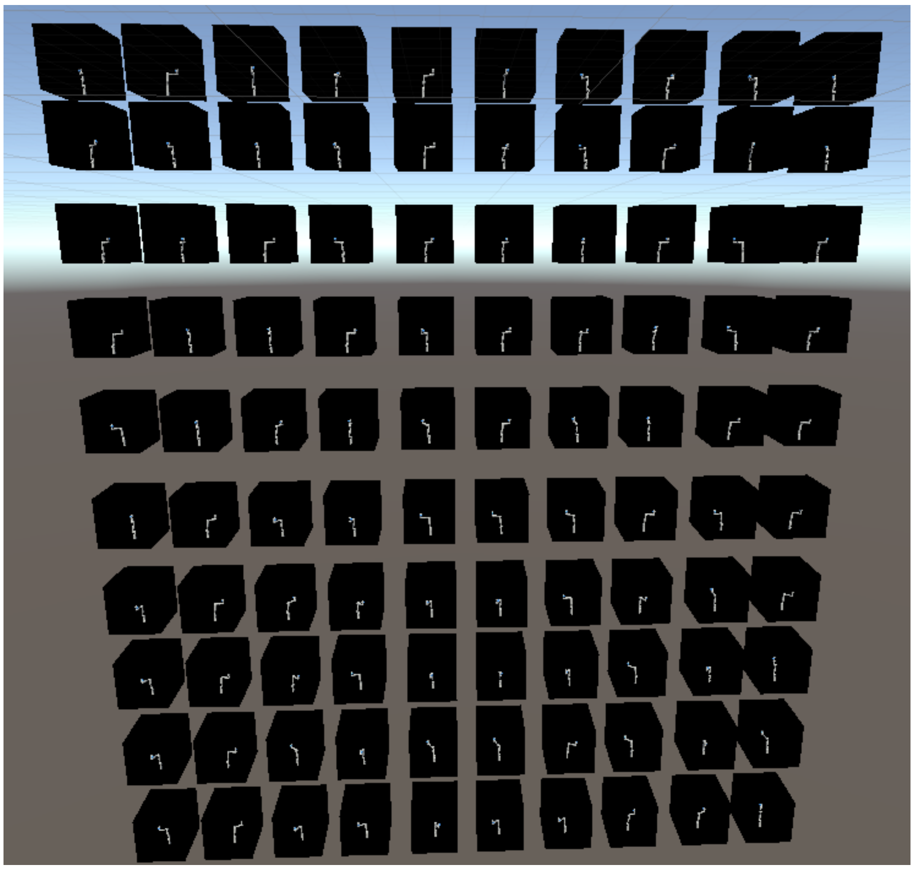
The simulation has to solve the IK for the requested poses as the agent does not perform direct joint control. In addition, as we are running multiple arms in simulation, performance is crucial. Therefore, we chose IKFast (DIANKOV, 2010) as our IK solver. It provides a tool to generate C++ code, which can be compiled and run self-contained without any external tool. Therefore it can be called within Unity by compiling it into a shared library. This approach drastically reduces communication overhead, allowing the IK to be solved without performance drop, as it can be seen in Figure 23. The solver produces at most 16 solutions, and the solution that causes the least spatial displacement of the arm is chosen.

4.4 Final considerations

This chapter presented the proposed research for this work. First, every part of the modeling was discussed and explained: the state and action spaces, actions, rewards, network

[2] <https://gym.openai.com/docs/>

Figure 23 – Demonstration of the IKFast solver being run inside the simulation. By using it as a shared library, communication overhead is greatly reduced. Here 100 instances of the robotic arm are being run in parallel without performance drop. A video can be found at <https://youtu.be/RGtPFAC3T08>.



Source: Elaborated by the author.

architecture, and how we perform feature augmentation. Then, the simulation setup is described. Finally, we explain the techniques and resources used.

CHAPTER 5

EXPERIMENTS AND RESULTS

This chapter details the experiments and results. First, the evaluation metrics are presented. Next, the experiments are detailed. Then the training setup used to train the models and run the tests is described. Finally, results are presented and analyzed.

5.1 Evaluation Metrics

The models are evaluated using three different metrics: success rate, episodic reward, and episode length. The success rate evaluates the ability of the model to reach the target position without collisions. We considered that the model successfully reaches the target position when the end-effector is within 10cm of the target. The episodic reward measures the accumulated reward during an episode. The episode length is measured in agent steps, i.e., the number of times the agent reacted to an environment state.

5.2 Experiment Description

The experiments' goal was to evaluate model performance and capacity to learn from visual inputs. As a result, we evaluated four different policies:

1. Random: the agent uses a policy that performs actions at random, without considering the environment state. These actions are sampled from a uniform distribution. No training was required for this model; the policy was hard-coded. This model's goal was to evaluate how complex the environment was.
2. Naive: the agent moves in a straight line in the direction of the target position without considering the obstacle's position. This policy also aligns the agent to the target position to increase the arm's reach and increase the probability of the arm's controller finding a valid IK solution. This policy was also hard-coded, thus not requiring training. The

goal of this model was to measure the impact of the obstacle not always being in the agent's line-of-sight. A video showing the execution of this policy can be seen at <https://youtu.be/dzpGLlKWfJg>.

3. Scalar: the agent uses only non-visual information of the environment. In this scenario, the agent directly receives the obstacle's position instead of the visual observation of the environment. This effectively replaces the term s_{camera} in Equation 4.1 for the obstacle position $p_{obstacle}$. The neural network used for this model is a modified version of the architecture shown in Figure 18, consisting only of fully connected layers Fc3, Fc4, and Fc5. This model was important to determine if the agent could solve the task if the obstacle position is known and how well learning occurs without images.

4. Visual: the agent uses mixed (visual and non-visual) information of the environment. Here we used the camera image as well as proprioceptive sensor information, as described in Equation 4.1. The neural network architecture used is described in section 4.1.4. No information from the obstacle is explicitly available to the agent; it has to extract this information from the image. A video showing the execution of this policy can be seen at <https://youtu.be/DmQf3LnLj8U>.

The evaluation of different model types was fundamental because we found difficult to compare our work with existing literature directly. This is because there is no standardized environment to run experiments, and metrics like success rate, episodic reward, and episode length are heavily dependent on the task's modeling.

5.3 Training Setup

All training was conducted on the same hardware, an Intel(R) Core(TM) i5-9600K CPU @ 3.70GHz with 32Gb of RAM and a GeForce RTX 2070 Super with 8Gb of VRAM. The hyperparameters used in the experiments are summarized in Table 2.

For every e_{freq} steps, training was interrupted, and the model was evaluated for e_{epi} episodes. The evaluation metrics were then averaged over e_{epi}, and training was resumed. Replay ratio is the number of gradients updates per experience collected. The model performs one gradient update with a replay ratio of 0.04 every time 25 new experiences are collected. This ratio is an important hyperparameter identified by many recent studies (FEDUS *et al.*, 2020). All other hyperparameters are explained in Fujimoto, Hoof and Meger (2018) and in Section 4.1.3.

5.4 Results and Analysis

Evaluation data were collected during training for the scalar and the mixed models. As there was no training involved in the random and naive models, they were instead evaluated for

Table 2 – Hyperparameters used in training

Parameter	Value
Evaluation frequency (e_{freq})	50000
Evaluation episodes (e_{epi})	100
Replay ratio	0.04
Maximum timesteps	5000000
Max. episode length	250
Exploration noise	0.2
Policy noise	0.2
Batch size	128
Discount factor (γ)	0.99
Noise clip	0.5
Policy update frequency	2
Actor learning rate	0.0001
Critic learning rate	0.0001
Random seeds	0, 1, 2, 3, 4
c_1	10
c_2	1
c_3	1
δ	0.5
d_{ref}	0.2
p	8

1000 episodes. All experiments were run five times with a different seed each time. The same set of seeds were used across experiments and are shown in Table 2. The results are summarized in Figure 24 and Table 3.

Table 3 – Training Results. Results are averaged after 5 runs.

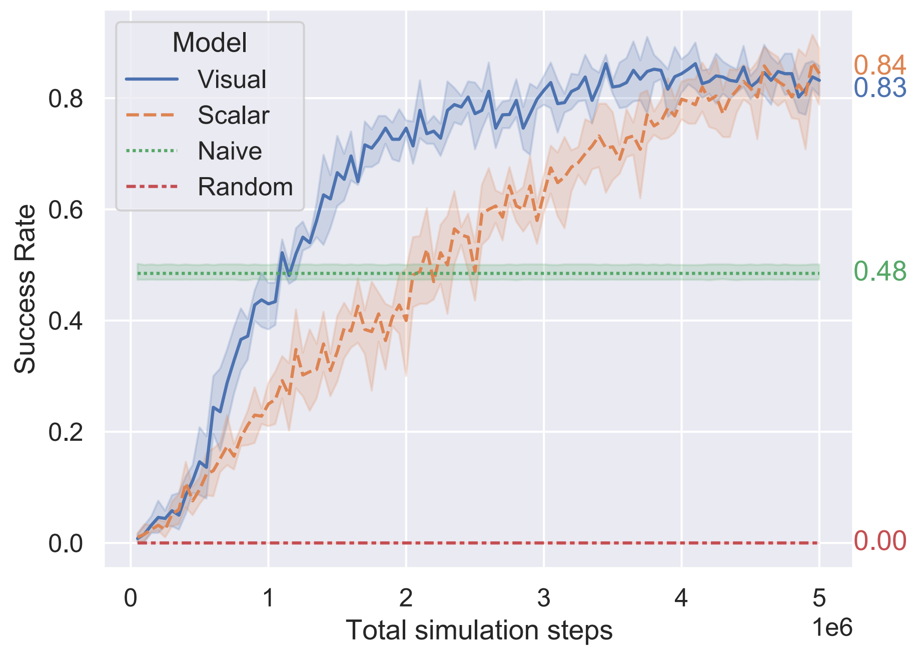
Model Name	Actor # of Parameters	Critic # of Parameters	Success Rate (%)	Episodic Reward	Episode Length	Training Time
Random	-	-	$0_{\pm(0)}$	$-370.45_{\pm(3.08)}$	$248.15_{\pm(0.75)}$	-
Naive	-	-	$48.48_{\pm(1.6)}$	$-41.49_{\pm(0.62)}$	$44.82_{\pm(0.76)}$	-
Scalar	**74246**	**74497**	$\mathbf{84.4}_{\pm(6.58)}$	$-15.12_{\pm(4.92)}$	$53.02_{\pm(10.61)}$	**2h 28m**
Visual	315961	316212	$83.2_{\pm(3.56)}$	$\mathbf{-12.06}_{\pm(3.01)}$	$\mathbf{44.22}_{\pm(4.9)}$	5h 16m

The random model was, as expected, the worst-performing model. The environment is relatively complex for a random policy to be effective; thus, it never reached the target, with a success rate of $0 \pm 0\%$. This bad performance is also reflected in the episodic reward; as the agent could not get closer to the target, its rewards remained low, with a value of -370.45 ± 3.08. The episode length of 248.15 ± 0.75 indicates that almost all episodes ended in timeout, as the maximum number of steps per episode was 250.

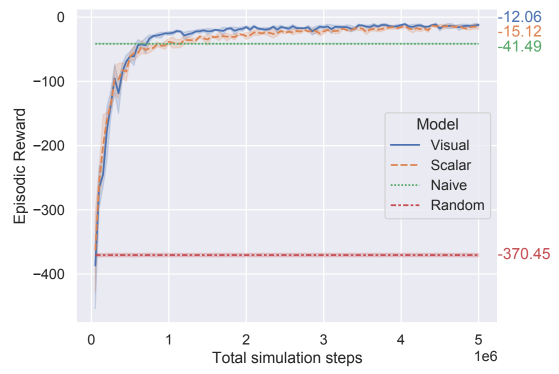
The naive model achieved a success rate of $48.48 \pm 1.6\%$. The reason for the success rate being reasonably high is the way the object is placed in the scene (refer to Figure 20). By

Figure 24 – Training results using non-visual information (Scalar) and using mixed-information (Vision), with random and naive policies as baselines. The results are averaged after 5 runs. (a) Comparison of the success rates. (b) Comparison of episodic rewards. (c) Comparison of episode lengths.

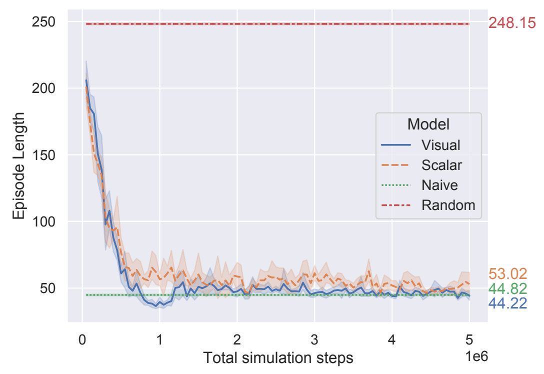
(a)



(b)



(c)



Source: Elaborated by the author.

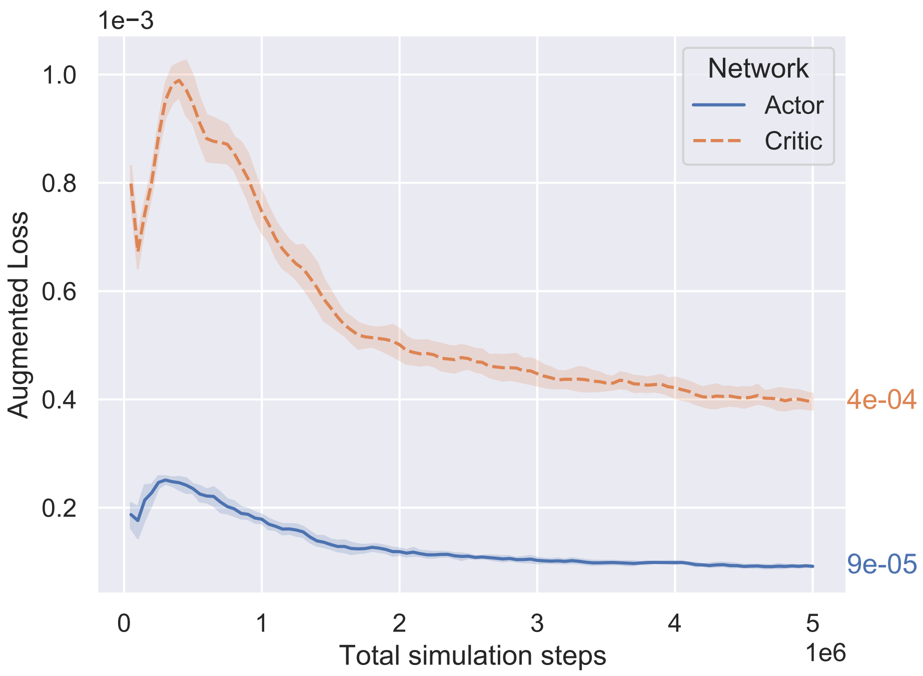
allowing $r_{obstacle}$ between 0 and 30cm, the obstacle has a chance of being outside the agent's motion range, thus not generating a collision.

The scalar model had a success rate of $84.4 \pm 6.58\%$. This rate is significantly higher than the other two previous models. This model's episode length of 53.02 ± 10.61 is similar to the naive model's 44.82 ± 0.76, indicating that the agent reaches the target position in a reasonable amount of steps. This model reached a better success rate than the naive model at around 2.1m steps.

Finally, the visual model achieved a success rate of $83.2 \pm 3.56\%$, practically the same as the scalar model. This rate shows that the visual model was able to extract the obstacle's information from the image. Its episode length was also very similar to the naive model, being ahead by just a negligible margin. The training time was considerably larger than the scalar model, as expected, as the visual model has approximately four times more parameters than the scalar model. There is also more communication overhead between the simulation and the Python code as the amount of information passed increases.

When comparing the scalar and visual models' success rate curves, one fact that caught our attention was that the visual model had better performance throughout almost the whole training procedure. This fact was unexpected because theoretically, the visual model should take much longer to converge, as it has much more free parameters. Therefore, it should take longer for it to start performing as well as the scalar model that directly receives the obstacle's position.

Figure 25 – Augmented function loss (MSE) for the actor and critic networks in the visual model.



Source: Elaborated by the author.

We hypothesize that the agent extracts more than just the obstacle's position from the image. As the arm is also in the camera's field of view (Figure 19b), the agent can observe the arm's positioning relative to the obstacle as it moves through the environment. Therefore, it can react before a collision happens on the arm. However, in the scalar model, the agent has to learn how to relate joint angles and world coordinates to figure out the arm's position in relation to the obstacle.

Another factor that explains the early performance of the visual model is that the actor and critic networks can predict the augmented features with low error very early in the training process. The augmented features loss remains very low throughout training, as it is shown in Figure 25. This low loss effectively means that the agent can estimate the obstacle's position with good precision very early in the training process.

By analyzing both the scalar and visual model executions, we noticed that the agents moved in a straight line to the goal position when the obstacle was not in the route of collision. This behavior means that the agents were only correcting their movements when strictly needed, as opposed to exploiting curved motions (as explained in Section 4.2). That is an important conclusion, as works such as (ALJALBOUT *et al.*, 2020) do not take into account this phenomenon. By spreading the possible obstacle's location, we effectively forced the agent to learn how to react to it and not just avoid the region that it is most likely located (as did the agents trained in our preliminary tests).

This spread of the obstacle's location may raise questions of whether or not the obstacle is, in fact, an obstacle. However, by analyzing the naive policy's success rate, we can argue that if an agent ignored the obstacle's position and exploited linear movements towards the goal, its success rate would be around 48%. Instead, both the scalar and visual policies had around 35% more success than the naive policy, indicating that they were actively avoiding the obstacle. We find this result important because it shows that the agent is not performing actions that exploit the obstacle's placement region, such as avoiding a region to avoid collisions.

5.5 Final considerations

This section presented the experiments and results from this work. The metrics used to evaluate performance were presented, as well as the details of the experiments. Then, the training setup (hardware and hyperparameters) was described. Finally, the results were presented and discussed.

Four policies were tested: a random policy, a naive policy, a scalar policy, and a visual policy. We showed that the visual policy had a final performance almost as good as the scalar policy and learned the task faster by using elements present in the camera's image.

Our model was able to use relatively low resolution, segmented images to accomplish its task. We argue that this is important because this means that this policy can theoretically be transferred more easily to a real robot. However, testing on a real robot was outside the scope of this work.

CHAPTER

6

CONCLUSION AND FUTURE WORK

This work presented an end-to-end, deep reinforcement learning approach to perform obstacle avoidance for a robotic arm using computational vision. Our model performed motion control of a 7 DOF arm directly from images and proprioceptive sensors. The system developed uses Deep Reinforcement Learning (DRL) to perform control from images and proprioceptive sensors to find the goal avoiding obstacles randomly positioned. The main difference of our approach from existing works is that it is end-to-end, meaning that it requires minimal modeling and a low resolution (64×64) segmented image as the visual input of the system. This type of visual input makes the policy more likely to be transferred to a real robotic arm, as shown in the related works. However, as no experiments were performed on a real robotic arm, we can not guarantee that it would be transferred successfully.

We also developed a simulated environment to conduct experiments and validate the proposed policy. This environment has the advantage of being created using tools that value ease of use and flexibility, allowing future researchers to use it in their research by using well-established interfaces and APIs. This simulated environment was used to validate our model. In total, we evaluated four policies: a random policy; a naive policy that moved straight to the goal position, ignoring the obstacle; a scalar policy that uses the obstacle position and proprioceptive sensor data, but not the camera's image; and finally, the visual policy, that uses the camera's image, as well as proprioceptive sensor data, but not the obstacle's position. The random and naive policies served as a baseline for comparison for the other two policies. Both scalar and visual policies obtained a success rate of about 84% in reaching the goal position without collisions. By achieving performance comparable to the scalar model, we demonstrated that the visual model could extract obstacle information from the image.

This dissertation covered a broad range of topics and techniques. Much effort was made to ensure that all the parts of the system worked together and without errors. In addition, modeling a task using DRL proved quite challenging. We spent many hours testing different network architectures, hyperparameters, and environment setups. Perhaps the most challenging aspect of developing such a system was the difficulty of tracking down errors. From our experience, learning proved to be quite sensitive to changes in the environment or hyperparameters, and determining which aspect was hindering the agent was quite difficult.

Furthermore, the knowledge acquired during this dissertation contributed to the following results:

- The writing and submission of the paper:
  - SANCHES, F. P. ; ROMERO, R. A. F. , "End-to-End Visual Obstacle Avoidance for a Robotic Manipulator using Deep Reinforcement Learning" to the 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2021). Submitted in March, 2021.
- Participation in the Brazilian Robotics Competition (CBR) in the Category: @HomeCBR, in which our team (Warthog) achieved second place in 2019 and 2020.
- The writing and publication of the paper:
  - BELO, J. P. R. ; SANCHES, F. P. ; ROMERO, R. A. F. . Facial Recognition Experiments on a Robotic System Using One-Shot Learning. In: LARS/SBR 2019, 2019, RIO GRANDE - RS, Proceedings IEEE LARS/SBR 2019. Palo Alto - CA: IEEE, 2019.
- A chapter in the book "Aplicações de Estratégias Básicas de Busca em um Ambiente Real", SANCHES, F. P. ; ROMERO, R. A. F., p. 312, submitted to publication by the USP Publisher.

We showed using the proposed experiments that our model could perform the task with a reasonable success rate. However, there is still much room for improvement. Below we list some aspects that we believe would greatly improve our work and that are part of future works:

Adding multiple obstacles: in a real-world scenario, it is much more likely to find environments with multiple obstacles rather than just one. As such, it is desirable that the agent reacts and avoid them properly;

Obstacles of different sizes and shapes: objects appear in different sizes and shapes in the real world. An agent that is capable of reacting to them is much more suited to perform real-world tasks;

Obstacle occlusion: often objects are hidden or partially occluded by other objects. As such, it would be interesting if the agent considered that when moving;

Integrate multiple image sources: it is often the case that there are multiple cameras present in the robot's environment. By using all the available information, the robot would have a better understanding of its surroundings, reacting more appropriately to it;

Incorporate prior information to the model: prior information can be injected into the model to improve training efficiency. Methods such as learning from demonstrations and residual RL are a step in this direction;

Integration with a real robot: to fully evaluate and understand the limitations of our approach, tests with a real robot are needed;

Applying to an existing problem: our approach can be used in several different areas. For example, we could employ our approach to harvesting robots to perform obstacle avoidance while reaching for fruit.

All these improvements aim to increase the model's robustness and make it more likely to be used in a real-world scenario.

BIBLIOGRAPHY

AKKAYA, I.; ANDRYCHOWICZ, M.; CHOCIEJ, M.; LITWIN, M.; MCGREW, B.; PETRON, A.; PAINO, A.; PLAPPERT, M.; POWELL, G.; RIBAS, R. *et al.* Solving rubik's cube with a robot hand. **arXiv preprint arXiv:1910.07113**, 2019. Citation on page 28.

ALJALBOUT, E.; CHEN, J.; RITT, K.; ULMER, M.; HADDADIN, S. Learning vision-based reactive policies for obstacle avoidance. In: **Conference on Robot Learning**. [S.l.: s.n.], 2020. Citations on pages 50, 52, 54, 55, and 76.

ANDRYCHOWICZ, M.; WOLSKI, F.; RAY, A.; SCHNEIDER, J.; FONG, R.; WELINDER, P.; MCGREW, B.; TOBIN, J.; ABBEEL, P.; ZAREMBA, W. Hindsight experience replay. **Advances in Neural Information Processing Systems**, v. 2017-Decem, n. Nips, p. 5049–5059, 2017. ISSN 10495258. Citation on page 55.

BENOTSMANE, R.; DUDÁS, L.; KOVÁCS, G. Collaborating robots in industry 4.0 conception. In: IOP PUBLISHING. **IOP Conference Series: Materials Science and Engineering**. [S.l.], 2018. v. 448, n. 1, p. 012023. Citation on page 27.

BROCKMAN, G.; CHEUNG, V.; PETTERSSON, L.; SCHNEIDER, J.; SCHULMAN, J.; TANG, J.; ZAREMBA, W. Openai gym. **arXiv preprint arXiv:1606.01540**, 2016. Citation on page 67.

CHENG, C.-A.; MUKADAM, M.; ISSAC, J.; BIRCHFIELD, S.; FOX, D.; BOOTS, B.; RATLIFF, N. Rmpflow: A computational graph for automatic motion policy generation. In: SPRINGER. **International Workshop on the Algorithmic Foundations of Robotics**. [S.l.], 2018. p. 441–457. Citation on page 50.

DENG, J.; DONG, W.; SOCHER, R.; LI, L.-J.; LI, K.; FEI-FEI, L. Imagenet: A large-scale hierarchical image database. In: IEEE. **2009 IEEE conference on computer vision and pattern recognition**. [S.l.], 2009. p. 248–255. Citations on pages 52 and 55.

DIANKOV, R. **Automated construction of robotic manipulation programs**. Phd Thesis (PhD Thesis) — Carnegie Mellon University, Pittsburgh, PA, 2010. Citations on pages 52 and 69.

DIANKOV, R.; KUFFNER, J. Openrave: A planning architecture for autonomous robotics. **Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34**, v. 79, 2008. Citation on page 28.

DU, G.; WANG, K.; LIAN, S. Vision-based robotic grasping from object localization, pose estimation, grasp detection to motion planning: A review. **arXiv preprint arXiv:1905.06658**, 2019. Citation on page 57.

FEDUS, W.; RAMACHANDRAN, P.; AGARWAL, R.; BENGIO, Y.; LAROCHELLE, H.; ROWLAND, M.; DABNEY, W. Revisiting fundamentals of experience replay. In: PMLR. **International Conference on Machine Learning**. [S.l.], 2020. p. 3061–3071. Citation on page 72.

FUJIMOTO, S.; HOOF, H.; MEGER, D. Addressing function approximation error in actor-critic methods. In: PMLR. **International Conference on Machine Learning**. [S.l.], 2018. p. 1587–1596. Citations on pages 42, 66, and 72.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. **Deep Learning**. [S.l.]: MIT Press, 2016. <http://www.deeplearningbook.org>. Citations on pages 31, 32, 33, and 34.

HADDADIN, S.; LUCA, A. D.; ALBU-SCHÄFFER, A. Robot collisions: A survey on detection, isolation, and identification. **IEEE Transactions on Robotics**, IEEE, v. 33, n. 6, p. 1292–1312, 2017. Citations on pages 28 and 48.

HARRIS, C. R.; MILLMAN, K. J.; WALT, S. J. van der; GOMMERS, R.; VIRTANEN, P.; COURNAPEAU, D.; WIESER, E.; TAYLOR, J.; BERG, S.; SMITH, N. J.; KERN, R.; PICUS, M.; HOYER, S.; KERKWIJK, M. H. van; BRETT, M.; HALDANE, A.; R'ıO, J. F. del; WIEBE, M.; PETERSON, P.; G'eRARD-MARCHANT, P.; SHEPPARD, K.; REDDY, T.; WECKESSER, W.; ABBASI, H.; GOHLKE, C.; OLIPHANT, T. E. Array programming with NumPy. **Nature**, Springer Science and Business Media LLC, v. 585, n. 7825, p. 357–362, Sep. 2020. Available: <https://doi.org/10.1038/s41586-020-2649-2>. Citation on page 66.

HASSELT, H. Double q-learning. **Advances in neural information processing systems**, Citeseer, v. 23, p. 2613–2621, 2010. Citation on page 42.

HAUSKNECHT, M. J.; STONE, P. Deep recurrent q-learning for partially observable mdps. In: **2015 AAAI Fall Symposia, Arlington, Virginia, USA, November 12-14, 2015**. [S.l.]: AAAI Press, 2015. p. 29–37. Citation on page 59.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. **Neural Computation**, v. 9, n. 8, p. 1735–1780, 1997. Available: <https://doi.org/10.1162/neco.1997.9.8.1735>. Citation on page 46.

HUBER, A.; WEISS, A. Developing human-robot interaction for an industry 4.0 robot: How industry workers helped to improve remote-hri to physical-hri. In: **Proceedings of the Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction**. [S.l.: s.n.], 2017. p. 137–138. Citation on page 27.

JOHANNINK, T.; BAHL, S.; NAIR, A.; LUO, J.; KUMAR, A.; LOSKYLL, M.; OJEA, J. A.; SOLOWJOW, E.; LEVINE, S. Residual reinforcement learning for robot control. In: IEEE. **2019 International Conference on Robotics and Automation (ICRA)**. [S.l.], 2019. p. 6023–6029. Citations on pages 50 and 51.

JULIANI, A.; BERGES, V.-P.; VCKAY, E.; GAO, Y.; HENRY, H.; MATTAR, M.; LANGE, D. Unity: A general platform for intelligent agents. **arXiv preprint arXiv:1809.02627**, 2018. Citation on page 67.

KOENIG, N.; HOWARD, A. Design and use paradigms for gazebo, an open-source multi-robot simulator. In: IEEE. **2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)**. [S.l.], 2004. v. 3, p. 2149–2154. Citation on page 51.

LAMPLE, G.; CHAPLOT, D. S. Playing fps games with deep reinforcement learning. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2017. v. 31, n. 1. Citations on pages 15, 39, 40, 58, and 64.

LECUN, Y.; BOTTOU, L.; BENGIO, Y.; HAFFNER, P. Gradient-based learning applied to document recognition. **Proceedings of the IEEE**, Ieee, v. 86, n. 11, p. 2278–2324, 1998. Citation on page 34.

LILLICRAP, T. P.; HUNT, J. J.; PRITZEL, A.; HEESS, N.; EREZ, T.; TASSA, Y.; SILVER, D.; WIERSTRA, D. Continuous control with deep reinforcement learning. In: BENGIO, Y.; LECUN, Y. (Ed.). **4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings**. [S.l.: s.n.], 2016. Citations on pages 40 and 41.

LOBOS-TSUNEKAWA, K.; LEIVA, F.; SOLAR, J. Ruiz-del. Visual navigation for biped humanoid robots using deep reinforcement learning. **IEEE Robotics and Automation Letters**, IEEE, v. 3, n. 4, p. 3247–3254, 2018. Citations on pages 45, 46, 47, 54, 55, 59, and 63.

MINAEE, S.; BOYKOV, Y. Y.; PORIKLI, F.; PLAZA, A. J.; KEHTARNAVAZ, N.; TERZOPOULOS, D. Image segmentation using deep learning: A survey. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, IEEE, 2021. Citation on page 60.

MNIH, V.; KAVUKCUOGLU, K.; SILVER, D.; RUSU, A. A.; VENESS, J.; BELLEMARE, M. G.; GRAVES, A.; RIEDMILLER, M.; FIDJELAND, A. K.; OSTROVSKI, G.; PETERSEN, S.; BEATTIE, C.; SADIK, A.; ANTONOGLOU, I.; KING, H.; KUMARAN, D.; WIERSTRA, D.; LEGG, S.; HASSABIS, D. Human-level control through deep reinforcement learning. **Nature**, Nature Publishing Group, v. 518, n. 7540, p. 529–533, 2015. ISSN 14764687. Available: <http://dx.doi.org/10.1038/nature14236>. Citations on pages 39 and 40.

NAIR, A.; MCGREW, B.; ANDRYCHOWICZ, M.; ZAREMBA, W.; ABBEEL, P. Overcoming exploration in reinforcement learning with demonstrations. **Proceedings - IEEE International Conference on Robotics and Automation**, p. 6292–6299, 2018. Citation on page 50.

PASZKE, A.; GROSS, S.; MASSA, F.; LERER, A.; BRADBURY, J.; CHANAN, G.; KILLEEN, T.; LIN, Z.; GIMELSHEIN, N.; ANTIGA, L.; DESMAISON, A.; KOPF, A.; YANG, E.; DEVITO, Z.; RAISON, M.; TEJANI, A.; CHILAMKURTHY, S.; STEINER, B.; FANG, L.; BAI, J.; CHINTALA, S. Pytorch: An imperative style, high-performance deep learning library. In: WALLACH, H.; LAROCHELLE, H.; BEYGELZIMER, A.; ALCHé-BUC, F. d'; FOX, E.; GARNETT, R. (Ed.). **Advances in Neural Information Processing Systems 32**. [S.l.]: Curran Associates, Inc., 2019. p. 8024–8035. Citation on page 66.

RÖFER, T.; LAUE, T.; KUBALL, J.; LÜBKEN, A.; MAASS, F.; MÜLLER, J.; POST, L.; RICHTER-KLUG, J.; SCHULZ, P.; STOLPMANN, A. *et al.* **B-human: Team Report and Code Release 2016**. [S.l.]: Deutschen Forschungszentrums für Künstliche Intelligenz (DFKI) GmbH, 2016. Citation on page 47.

ROHMER, E.; SINGH, S. P.; FREESE, M. V-rep: A versatile and scalable robot simulation framework. In: IEEE. **2013 IEEE/RSJ International Conference on Intelligent Robots and Systems**. [S.l.], 2013. p. 1321–1326. Citation on page 48.

RUMELHART, D. E.; HINTON, G. E.; WILLIAMS, R. J. Learning representations by back-propagating errors. **nature**, Nature Publishing Group, v. 323, n. 6088, p. 533–536, 1986. Citation on page 34.

SANGIOVANNI, B.; INCREMONA, G. P.; PIASTRA, M.; FERRARA, A. Self-configuring robot path planning with obstacle avoidance via deep reinforcement learning. **IEEE Control Systems Letters**, IEEE, v. 5, n. 2, p. 397–402, 2020. Citations on pages 48, 54, 55, and 56.

SANGIOVANNI, B.; RENDINIELLO, A.; INCREMONA, G. P.; FERRARA, A.; PIASTRA, M. Deep reinforcement learning for collision avoidance of robotic manipulators. In: IEEE. **2018 European Control Conference (ECC)**. [S.l.], 2018. p. 2063–2068. Citations on pages 47, 48, 49, 54, 55, 56, and 61.

SCHAUL, T.; QUAN, J.; ANTONOGLOU, I.; SILVER, D. Prioritized experience replay. In: BENGIO, Y.; LECUN, Y. (Ed.). **4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings**. [S.l.: s.n.], 2016. Citation on page 51.

SCHOETTLER, G.; NAIR, A.; LUO, J.; BAHL, S.; OJEA, J. A.; SOLOWJOW, E.; LEVINE, S. Deep reinforcement learning for industrial insertion tasks with visual inputs and natural rewards. In: **IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2020, Las Vegas, NV, USA, October 24, 2020 - January 24, 2021**. [S.l.]: IEEE, 2020. p. 5548–5555. Citations on pages 49, 50, 54, and 55.

SDK, P. **Unity Public Relations**. 2021. Available: <https://developer.nvidia.com/physx-sdk>. Accessed: 03/04/2021. Citation on page 67.

SHILLER, Z. Off-line and on-line trajectory planning. **Motion and Operation Planning of Robotic Systems**, Springer, p. 29–62, 2015. Citation on page 27.

SILVER, D.; LEVER, G.; HEESS, N.; DEGRIS, T.; WIERSTRA, D.; RIEDMILLER, M. Deterministic policy gradient algorithms. In: . [S.l.: s.n.], 2014. Citation on page 40.

SUTTON, R. S.; BARTO, A. G. **Reinforcement learning: An introduction**. [S.l.]: MIT press, 2018. Citations on pages 35, 36, 37, and 38.

TAN, J.; ZHANG, T.; COUMANS, E.; ISCEN, A.; BAI, Y.; HAFNER, D.; BOHEZ, S.; VANHOUCKE, V. Sim-to-real: Learning agile locomotion for quadruped robots. In: KRESS-GAZIT, H.; SRINIVASA, S. S.; HOWARD, T.; ATANASOV, N. (Ed.). **Robotics: Science and Systems XIV, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, June 26-30, 2018**. [S.l.: s.n.], 2018. Citation on page 28.

TECHNOLOGIES, U. **Unity Public Relations**. 2020. Available: <https://unity3d.com/public-relations>. Accessed: 03/04/2021. Citation on page 67.

____. **Robotic Simulation - Unity**. 2021. Available: <https://unity.com/solutions/automotive-transportation-manufacturing/robotics>. Accessed: 03/04/2021. Citation on page 67.

TJAHJONO, B.; ESPLUGUES, C.; ARES, E.; PELAEZ, G. What does industry 4.0 mean to supply chain? **Procedia manufacturing**, Elsevier, v. 13, p. 1175–1182, 2017. Citation on page 27.

VECERÍK, M.; HESTER, T.; SCHOLZ, J.; WANG, F.; PIETQUIN, O.; PIOT, B.; HEESS, N.; ROTHÖRL, T.; LAMPE, T.; RIEDMILLER, M. A. Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. **CoRR**, abs/1707.08817, 2017. Citation on page 55.

YARATS, D.; ZHANG, A.; KOSTRIKOV, I.; AMOS, B.; PINEAU, J.; FERGUS, R. Improving sample efficiency in model-free reinforcement learning from images. **arXiv preprint arXiv:1910.01741**, 2019. Citation on page 58.

ZENG, A.; SONG, S.; WELKER, S.; LEE, J.; RODRIGUEZ, A.; FUNKHOUSER, T. Learning synergies between pushing and grasping with self-supervised deep reinforcement learning. **IEEE International Conference on Intelligent Robots and Systems**, p. 4238–4245, 2018. ISSN 21530866. Citations on pages 28, 51, 53, 54, and 55.

ICMC USP
SÃO CARLOS