Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Planning smooth and safe arm movements using reinforcement learning

Master's Thesis

2023

Bc. Tomáš Janeta

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Planning smooth and safe arm movements using reinforcement learning

Master's Thesis

Study Programme: Computer Science
Field of Study: Computer Science
Department: Department of Computer Science
Supervisor: prof. Ing. Igor Farkaš, Dr.

Bratislava, 2023
Bc. Tomáš Janeta

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Tomáš Janeta
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Planning smooth and safe arm movements using reinforcement learning
Plánovanie hladkých a bezpečných pohybov ramena pomocou učenia posilňovaním

Anotácia: Učenie posilňovaním (RL) je najbežnejším (zvyčajne) bezmodelovým prístupom pri učení sa sekvenčného správania, napr. v robotických systémoch. RL možno porovnať s klasickými algoritmami plánovania pohybu, ktoré sú založené na tvorbe grafov a hľadaní cesty. Medzi pretrvávajúce výzvy v RL patrí plynulé a bezpečné (bezkolízne) správanie, najmä v prípade viacerých stupňov voľnosti, ktoré je potrebné ovládať.

Cieľ: 1. Oboznámte sa s robotickým simulátorom CoppeliaSim a implementujte prispôsobené prostredie RL pre robotické rameno.
2. Porovnajte vybrané klasické algoritmy s prístupmi založenými na RL na vybranej úlohe dosiahnutia cieľa (aj s uvažovaním prekážok) pomocou robotického ramena.
3. Vyhodnoťte výsledky všetkých experimentov s následnou diskusiou.

Literatúra:
Kim M.S. et al. (2019). Motion planning of robot manipulators for a smoother path using a twin delayed Deep Deterministic Policy Gradient with Hindsight Experience Replay. Applied Sciences, 10, 575; doi:10.3390/app10020575
Busoniu L. et al. (2018). Reinforcement learning for control: Performance, stability, and deep approximators. Annual Reviews in Control, 46, https://doi.org/10.1016/j.arcontrol.2018.09.005

Vedúci: prof. Ing. Igor Farkaš, Dr.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 14.12.2021

Dátum schválenia: 18.12.2021

prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.... študent

.... vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Tomáš Janeta
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Planning smooth and safe arm movements using reinforcement learning

Annotation: Reinforcement learning (RL) is a most common (typically) model-free approach in learning sequential behavior, e.g. in robotic systems. RL can be contrasted with classical motion planning algorithms that are based on graph formation and path finding. Among remaining challenges in RL is smooth and safe (collision-free) behavior, especially in case of more degrees of freedom to be controlled.

Aim: 1. Get hands-on experience with a robotic simulator CoppeliaSim and implement the customized RL environment for a robotic arm.
2. Compare selected classical motion algorithms with RL-based approaches on a selected reaching task (also considering obstacles) using a robotic arm.
3. Evaluate and discuss the results of all experiments.

Literature: Kim M.S. et al. (2019). Motion planning of robot manipulators for a smoother path using a twin delayed Deep Deterministic Policy Gradient with Hindsight Experience Replay. Applied Sciences, 10, 575; doi:10.3390/app10020575
Busoniu L. et al. (2018). Reinforcement learning for control: Performance, stability, and deep approximators. Annual Reviews in Control, 46, https://doi.org/10.1016/j.arcontrol.2018.09.005

Supervisor:	prof. Ing. Igor Farkaš, Dr.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. RNDr. Tatiana Jajcayová, PhD.
Assigned:	14.12.2021

Approved: 18.12.2021 prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

....

Student Supervisor

Acknowledgments: I would like to thank my supervisor prof. Ing. Igor Farkaš, Dr. for his help and guidance. I would like to also thank my family and friends, who were always there for me.

// Abstrakt

Učenie s posilňovaním je oblasť strojového učenia, zaoberajúca sa optimálnou stratégiou pre agenta nachádzajúceho sa v prostredí s možnosťou vykonávania rôznych akcií. Plánovanie trasy robota je známy problém, pri ktorom sa snažíme pre robota nájsť čo najefektívnejšiu cestu, ktorá sa vyhne všetkým prekážkam. Klasické algoritmy pre plánovanie trasy robota sú založené na grafovom prístupe, a nie sú dostatočne efektívne v zložitom a dynamickom prostedí s mnohými stupňami voľnosti. V práci implementujeme prostredie pre simulovanú robotickú ruku. Pomocou tohto prostredia a algoritmov učenia s odmenou trénujeme agenta na rôznych úlohách plánovania trasy, pričom sme sa snažili dosiahnuť čo najväčšiu efektívnosť a bezpečnosť. Najlepšie natrénované modely sú porovnateľné z hľadiska efektívnosti s klasickými algoritmami na plánovanie trasy.

Kľúčové slová: robotická ruka, učenie s posilňovaním, neurónová sieť, plánovanie trasy robota

Abstract

Reinforcement learning is a machine learning area, studying optimal strategy for an agent in an environment with the possibility of taking various actions. Robotic path planning is a well-known problem in which we are trying to find an effective path for a robot that avoids all the obstacles. Classical algorithms for robotic path planning rely on graph algorithms and are not effective enough in complex and dynamic environments with many degrees of freedom. In the thesis, we implement a reinforcement learning environment for a simulated robotic arm. We train the agent using reinforcement learning algorithms on various path planning tasks, while trying to reach optimal smoothness and safety. The best models are comparable in terms of smoothness and effectiveness with classical motion planning algorithms.

Keywords: robotic arm, reinforcement learning, neural network, path planning

Contents

Introduction **1**

1 Background **2**
1.1 Motion planning . . . 2
1.2 Related work . . . 2
1.3 Technical details . . . 3

2 Motion planning algorithms **4**
2.1 Probabilistic Roadmaps (PRM) . . . 5
2.2 Rapid-exploring random trees (RRT) . . . 7
2.3 Complications . . . 9

3 Reinforcement learning **10**
3.1 Basic concepts . . . 10
3.2 Deep Deterministic Policy Gradient . . . 14
3.3 Twin Delayed DDPG . . . 15
3.4 Proximal Policy Optimization . . . 17
3.5 Soft Actor–Critic . . . 18

4 Implementation **20**
4.1 Reinforcement learning tools . . . 20
4.2 Evaluation . . . 23
4.3 Installation and deployment . . . 24

5 Results **26**
5.1 Evaluating smoothness and effectiveness . . . 26
5.2 Evaluating safety . . . 31

Conclusion **35**

Príloha A **40**

List of Figures

1.1 Physical Panda robotic arm (left) and its model in the CoppeliaSim simulator (right) [Emika, 2023]. 3

2.1 Output of the PRM and Lazy PRM algorithm [Garg, 2023] in 2D space. In the second image, created collisions can be seen. 7

2.2 Output of the RRT algorithm [Garg, 2023]. 8

3.1 One of the possible inputs for the Frozen Lake problem 14

4.1 Atari games environments from the gym package [OpenAI, a]: Atlantis and bank heist environment. 20

4.2 MuJoCo robotic environments from the gym package [OpenAI, b]: humanoid and half cheetah environment. 21

5.1 Averaged reward after 10000 steps from training with R_{joint} and R_{const} reward functions using `max_speed=0.2` 27

5.2 Averaged reward after 10000 steps from training with R_{cartes} and $R_{quatern}$ reward functions using `max_speed=0.2` 27

5.3 Averaged reward after 10000 steps from training with R_{joint} and R_{const} reward functions using `max_speed=1.0` 29

5.4 Averaged reward after 10000 steps from training with R_{cartes} and $R_{quatern}$ reward functions using `max_speed=1.0` 29

5.5 Dependence of $\rho_{joint}, \rho_{cartesian}, \rho_{find}$ metrics on `max_speed` parameter . 30

5.6 Averaged reward after 10000 steps with `max_speed=0.1` 31

5.7 Visualization of the simulated Panda robot traversing a path found by the TD3 model. The red ball represents the target. 32

5.8 Visualization of the experiments with non-variable obstacles. The obstacles are highlighted in blue color. 32

5.9 Visualization of the experiments with non-variable obstacles. The obstacles are highlighted in blue color. 34

List of Algorithms

1 The first phase of the PRM algorithm 6
2 The RRT algorithm 8
3 The Q-learning algorithm 13
4 DDPG 16
5 PPO 18
6 Pseudocode for step and reset methods 22

Introduction

Reinforcement learning (RL) is an area of machine learning concerned with finding optimal behavior for an intelligent agent placed in an environment. It has experienced massive growth in recent years and is now applied to many tasks like games, autonomous driving, and trading systems while achieving or even passing human-level precision.

Robotics has drawn enormous attention lately, as robots have been becoming able to take on more and more functions formerly considered only for humans. With the dawn of intelligent factories and Industry 4.0 technologies, the need for automation is increasing rapidly. The need of effective robotic path planning algorithms has grown with the increasing effort to automate as many industrial tasks as possible.

Classical path planning methods use graph algorithms to find the shortest path in a tree structure created from nodes sampled from the joint space. The problem with this approach is that generating a dense enough graph may require sampling many points, especially in the case of a robot with a high number of degrees of freedom. Moreover, as the sampling process needs to be repeated every time the environment changes, time complexity becomes an issue.

In this thesis, we applied reinforcement learning on a robotic arm and compared the results with classical algorithms. Our hypothesis is that RL methods can compete in performance with classical motion planning algorithms. We implemented a RL environment for a simulated robotic arm and tested it using multiple training algorithms.

In Chapter 1, we explain the background of the thesis. In Chapter 2, we present classical methods for path planning and their drawbacks. In Chapter 3, we explain the basic theory behind reinforcement learning. In Chapter 4, we describe our implementation and technical challenges. In Chapter 5, we present our results.

Chapter 1

Background

1.1 Motion planning

As technology evolves, the importance and impact of autonomous robotic systems in everyday life increases. Many industrial tasks have been successfully automated, achieving higher precision and productivity. While performing common industrial tasks, robots must move in an environment containing various obstacles, such as boxes, machines, or other robots. Therefore, the robot needs to be capable of finding a path that is both smooth and safe. Safety of the path ensures that the robot will not collide with any of the obstacles and cause damage to itself or the environment. Smoothness is essential for effectiveness since we want to find the shortest path possible. Solving this problem is called motion planning. The input of the motion planning algorithm is the robot's initial state, the target's position in the 3D space, and a description of obstacles, if there are any. Desired output is a path that has described qualities.

Robots can be divided into various classes. In the thesis, we focus solely on robotic arms. A robotic arm is a type of robot with functions similar to a human arm. A typical robotic arm comprises joints, motors, grippers, and manipulators. Robotic arms are often used in factories, performing tasks like box picking, assembling, and packing.

1.2 Related work

Motion planning is a well-known problem with many applications in the industry. In this thesis, we present a motion planning algorithm that uses deep neural networks and reinforcement learning and its application to the problem with many degrees of freedom. Classical motion planning algorithms, which do not use reinforcement learning, rely on generating random graphs and graph algorithms for shortest path finding. Two of the most common algorithms are Probabilistic Roadmaps [Kavraki et al., 1996] and Rapid-exploring Random Trees [LaValle, 1998].

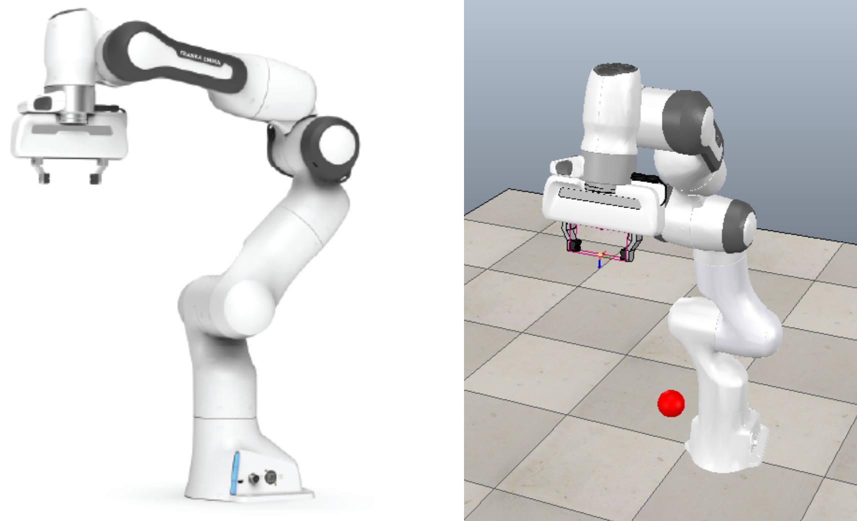

Figure 1.1: Physical Panda robotic arm (left) and its model in the CoppeliaSim simulator (right) [Emika, 2023].

Reinforcement learning is a machine learning paradigm, alongside supervised and unsupervised learning, studying how to choose optimal actions for an agent placed in an environment so that the agent receives the maximum reward. Reinforcement learning is a closely studied area that has utilization in many areas, such as chess engines [Silver et al., 2017], chatbots [Haristiani, 2019], and autonomous cars [Fayjie et al., 2018]. Reinforcement learning models provide good results when applied to robots with 2 or 3 degrees of freedom [Raajan et al., 2020, Kim et al., 2020, Yu et al., 2020].

1.3 Technical details

To control and manipulate the robots, we decided to use CoppeliaSim robotic simulator. CoppeliaSim [Rohmer et al., 2013, James et al., 2019] is a robotic simulator, the successor of V-REP, developed by Swiss company Coppelia Robotic. The core of the simulator uses mainly the programming language C++, but APIs from languages like Java, Python, or Lua are available along with good documentation and many helpful tutorials. The simulator is very widely utilized in industry, education, and research.

One of the many robots CoppeliaSim supports out of the box is Franka Emika Panda[1] [Gaz et al., 2019]. It is a robotic arm with seven joints and a gripper. It can be operated via a programming interface directly from a computer.

[1]from now on, we will refer to it as only Panda

Chapter 2

Motion planning algorithms

Suppose we have a robotic arm with n joints and a gripper. Assume the following notation

- S is a fixed cartesian system
- $\boldsymbol{\theta}_j$ is the vector of joint angles after timestep j.
- $\boldsymbol{g}_j = (x_{g,j}, y_{g,j}, z_{g,j})$ is the vector of coordinates of the gripper in S after timestep j
- $\boldsymbol{q}_j = (q_{x,j}, q_{y,j}, q_{z,j}, q_{w,j})$ is the quaternion of the gripper in the coordinate system S in time j
- $\boldsymbol{t} = (x_t, y_t, z_t)$ is the position of the static target in S

Further, denote Q, the set of all possible values of the $\boldsymbol{\theta_j}$, which in robotics is called a *joint space*. Q can be divided into two disjoint subsets $Q = Q_{free} \cup Q_{collide}$. Q_{free} represents the set of all states of the robotic arm that it can enter without causing a collision with obstacles or itself. On the other hand, $Q_{collide}$ represents the set of states the arm cannot enter without colliding.

For now, suppose there are no obstacles in the environment. Then input of a motion planning algorithm is $(\boldsymbol{\theta_{start}}, t, d)$, where $\boldsymbol{\theta_{start}}$ is a vector of initial joint angles. The position of the gripper is not included in the input, as the joint angle vector determines it. d is a distance threshold. The output of the algorithm is a sequence of configurations $\boldsymbol{\theta_1}, ...\boldsymbol{\theta_m}$, such that

- $\boldsymbol{\theta_1} = \boldsymbol{\theta_{start}}$
- $|\boldsymbol{g_m} - \boldsymbol{t}| \leq d$
- $\forall j, 1 \leq j \leq m : \boldsymbol{\theta_j} \in Q_{free}$

The first condition means that the path will start from the initial state. The second condition means that at the end of the path, the gripper will be close enough to the target. The third condition ensures safety, as all states along the path must be from the collision-free space. Assume we have already found a path $p = (\boldsymbol{\theta_1}, ...\boldsymbol{\theta_m})$ satisfying the conditions above. Define the following functions from the set of all possible paths to $\mathbb{R}$.

$$JD(p) = \sum_{i=1}^{m-1} \|\boldsymbol{\theta_i} - \boldsymbol{\theta_{i+1}}\| \tag{2.1}$$

$$CD(p) = \sum_{i=1}^{m-1} \|\boldsymbol{g_i} - \boldsymbol{g_{i+1}}\| \tag{2.2}$$

$$OC(p) = \sum_{i=1}^{m-1} \cos^{-1}(\boldsymbol{q_i}^T.\boldsymbol{q_{i+1}}) \tag{2.3}$$

These are the metrics most often used for measuring the effectiveness and smoothness of the path. The first one measures joint distance - the sum of all angles all joints had to turn at. The second one measures the Euclidean distance the gripper had to pass. The third one measures how much the orientation of the gripper changed along the path. A path found by the motion planning algorithm should minimize one of these measures to be effective.

This chapter explains a few most common methods for the presented problem. Explained methods use graph algorithms on a graph of nodes randomly generated from the joint space.

2.1 Probabilistic Roadmaps (PRM)

PRM [Kavraki et al., 1996, Dale and Amato, 2001] is a well-known motion planning algorithm. It has two additional parameters n, h. The first represents the number of vertices of the generated graph, and the second is the maximal distance of two connected nodes. The algorithm consists of 2 phases. The first phase generates a graph of random nodes from Q_{free}. Edges between two nodes represent the path between the corresponding configurations. This phase can be described by the Algorithm 1.

In the second phase, the algorithm tries to find a path in the generated graph from $\boldsymbol{\theta_{start}}$ using any graph algorithm for the shortest path finding. It is important to notice that the goal node is not known. Therefore, the graph must also save information about the distance of the gripper to the target for each node. The algorithm is very popular because of its straightforward implementation, but it is not guaranteed to find the optimal solution.

Algorithm 1 The first phase of the PRM algorithm

```
1: function GenerateGraph(θ_start, n, h)
2:     G ← empty graph
3:     add node θ_start to G
4:     for i from 1 to n do
5:         θ_new ← randomly generated node
6:         while θ_new ∉ Q_free do
7:             θ_new ← randomly generated node
8:         end while
9:         add θ_new to G
10:        for every vertex θ ∈ G do
11:            if θ ≠ θ_new then
12:                if |θ_new, θ| ≤ h then
13:                    if segment (θ_new, θ) lies in Q_free then
14:                        add edge (θ_new, θ) to G
15:                    end if
16:                end if
17:            end if
18:        end for
19:    end for

20:    return G
21: end function
```

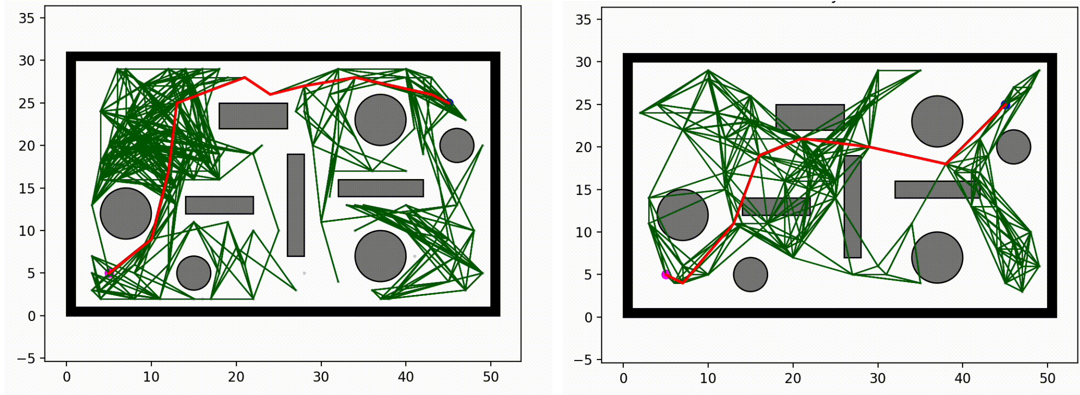

Figure 2.1: Output of the PRM and Lazy PRM algorithm [Garg, 2023] in 2D space. In the second image, created collisions can be seen.

Lazy PRM [Bohlin and Kavraki, 2000] is a variant of the basic algorithm that does not check for collisions in the first phase. Instead, it generates a graph, tries to find the path in the graph, assuming all nodes and edges are collision-free, and removes respective nodes and edges from the graph if it finds a collision. This variant has been shown to be faster in most cases than the classic PRM. Other variants of the PRM algorithm are available, offering smoother paths in exchange for increased computational complexity.

2.2 Rapid-exploring random trees (RRT)

RRT [LaValle, 1998, Lavalle and Kuffner, 2000] is the second sampling algorithm presented in this chapter. In contrast with PRM, RRT has only one phase. The algorithm takes two additional parameters n, h - the maximal number of nodes in the resulting graph and step.

We used a few special functions in Algorithm 2 of the RRT procedure. The function *nearest* takes a node $\boldsymbol{\theta}$ and a graph G as an input and returns a node from the G closest to the $\boldsymbol{\theta}$. The *newnode* function has two configurations $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2$, and step h as an input and returns configuration, created by moving distance h from $\boldsymbol{\theta}_2$ in the direction of $\boldsymbol{\theta}_1$. The function *path* finds the path between two nodes in a tree structure. The algorithm is very popular and has many variants. In some implementations, $\boldsymbol{\theta}_{rand}$ is used instead of the $\boldsymbol{\theta}_{new}$. A version of the algorithm called RRT*, which creates a very dense graph, has been shown to converge to the optimal solution, creating a very smooth path. However, this variant is potentially computationally costly.

Algorithm 2 The RRT algorithm

1: **function** RRT($\boldsymbol{\theta_{start}}, d, n$)
2: $G \leftarrow$ empty graph
3: add node $\boldsymbol{\theta_{start}}$ to G
4: **for** i from 1 to n **do**
5: $\boldsymbol{\theta_{rand}} \leftarrow$ randomly generated node
6: $\boldsymbol{\theta_{near}} \leftarrow nearest(\boldsymbol{\theta_{rand}}, G)$
7: $\boldsymbol{\theta_{new}} \leftarrow newnode(\boldsymbol{\theta_{rand}}, \boldsymbol{\theta_{new}}, h)$
8: **if** $\boldsymbol{\theta_{new}} \in Q_{free}$ and segment $(\boldsymbol{\theta_{new}}, \boldsymbol{\theta_{near}})$ lies in Q_{free} **then**
9: **if** $\boldsymbol{\theta_{new}}$ is close enough to the target **then return** $path(\boldsymbol{\theta_{start}}, \boldsymbol{\theta_{new}}, G)$
10: **end if**
11: add node $\boldsymbol{\theta_{new}}$ to G
12: add edge $(\boldsymbol{\theta_{new}}, \boldsymbol{\theta_{near}})$ to G
13: **end if**
14: **end for**
15: **end function**

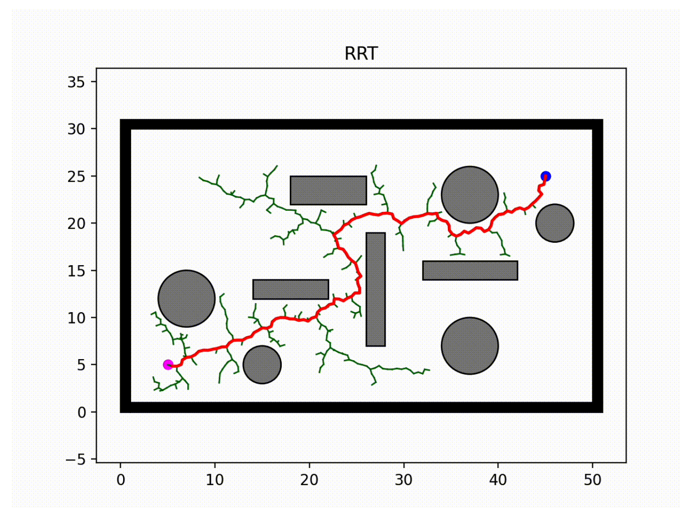

Figure 2.2: Output of the RRT algorithm [Garg, 2023].

2.3 Complications

Motion planning algorithms relying on sampling are handy in an environment with static obstacles. But if the obstacles change, configurations from the previous sampling are useless, and we need to generate a new graph. That can be very computationally expensive, especially when the robot has many degrees of freedom. Also, for a path to be relatively smooth, the searched graph must be dense, which increases computational cost.

Chapter 3

Reinforcement learning

3.1 Basic concepts

Reinforcement learning (RL) is an area of machine learning that studies how intelligent agents ought to take actions in an environment to maximize cumulative reward. RL can be applied to a wide range of tasks, starting with games such as chess [Silver et al., 2017] and Go [Silver et al., 2016] and ending with self-driving cars [Fayjie et al., 2018].

Suppose an agent is placed in an environment. We will be using the following notation.

- S is the set of all possible states of the agent, also called the observation space, and s_T the terminal state we want to reach (in general, there can be multiple terminal states)
- s_j is the state of the agent at time step j
- A is the set of all possible actions, called the action space
- $P : S \times A \times S \to [0, 1]$ is a probabilistic distribution function determining how likely the agent is to make a transition from state s to s' using action a
- $R : S \times A \times S \to \mathbb{R}$ is the reward function, meaning after making the transition from state s to s using action a, the agent will receive reward $R(s, a, s')$

Assume the agent uses a policy π to find the next action and the reward the agent receives while following policy π after the step i is R_i. The probability of choosing an action a in the state s while following policy π is denoted $\pi(a|s)$. The state-action value function $Q^\pi : S \times A \to R$ determines the value of a particular state-action tuple. Analogically, $V^\pi : S \to R$ is a state value function evaluating how good a particular state is.

$$Q^{\pi}(s,a) = \mathbb{E}\Big[\sum_{k=0}^{\infty}\gamma^k R_k | s_k = s, a_k = a\Big] \tag{3.1}$$

$$V^{\pi}(s) = \mathbb{E}\Big[\sum_{i=0}^{\infty}\gamma^k R_k | s_k = s\Big] \tag{3.2}$$

where $\gamma \in (0,1)$. The motivation for introducing this constant is that we want the agent to focus less on the rewards that are far in the future and more on the rewards achievable in a few steps. Also, the constant T can be chosen instead of the infinite sum. In such a case, the learning happens in episodes of at most T steps.

The relationship between these two functions can be expressed as

$$V^{\pi}(s) = \mathop{E}_{a \in A}\big[Q(s,a)\big]. \tag{3.3}$$

The optimal policy is usually marked as π^*. Optimal value functions are then

$$Q^*(s,a) = Q^{\pi^*}(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{3.4}$$

$$V^*(s) = Q^{\pi^*}(s) = \max_{\pi} V^{\pi}(s) \tag{3.5}$$

The following equation, the Bellman equation, can be derived for the state-action value and value-value functions

$$Q^{\pi}(s,a) = \mathop{E}_{s' \in S}\big[R(s,a,s') + \gamma \max_{a' \in A} Q^{\pi}(s',a')\big] \tag{3.6}$$

$$V^{\pi}(s) = \mathop{E}_{a \in A, s' \in S}\big[R(s,a,s') + \gamma V^{\pi}(s')P(s,a,s')\big] \tag{3.7}$$

The Bellman equation is one of the key formulas in RL theory. It is clear that if Q^* was known, the problem would be solved. In each state s, we would simply choose the best possible action $\mathop{\mathrm{argmax}}_{a \in A} Q^*(s,a)$. If the observation and action spaces are finite, the task to find the optimal policy is much more simple. The well-known algorithm for this type of problem is called Q-learning [Watkins and Dayan, 1992] and is extremely straightforward. Suppose π_k is our learned policy after step k, $\varepsilon \in (0,1)$ and T is the episode length. In each step, the algorithm learns from the experience obtained from the environment, and updates the policy accordingly. The details of the algorithm are explained in procedure 3. The purpose of the ε constant is to decide whether the agent should explore new actions or use the current policy to determine the best possible action. This is called the exploration vs exploitation trade-off [Wang et al., 2019] because the agent has to choose between exploiting an immediate certain reward, achievable using the current policy, or exploring new actions, which can bring even greater reward. Finally, α is the learning rate, determining how fast the agent should converge to the optimal policy.

RL algorithms can be classified in two ways. The first classification is between model-based algorithms, which assume a mathematical model behind the environment, and model-free algorithms, which work without such an assumption. The second classification is between on-policy and off-policy algorithms. The off-policy algorithms do not use the learned policy to find the best action. The opposite are the algorithms from the on-policy class. Q-learning belongs to the off-policy class of algorithms because, during each step, it uses the new updated policy. It is also from the model-free class of algorithms because it makes no assumptions about the environment besides the finiteness of both action and state space. The Markov chain [Yang, 2020] is an example of a model-based algorithm since it assumes probabilistic distribution behind the environment. SARSA [van Seijen et al., 2009] is an example of an algorithm from the on-policy class.

All modern reinforcement learning algorithms make use of a technique called replay buffer. For example, when the agent performs a transition from the state s to s' using action a and achieving reward r, the tuple (s, a, r, s', d) will be stored in the buffer, where $d = 1$ if s' is a terminal state, and 0 otherwise. Experience from the buffer is later replayed and used for learning the policy.

Now we illustrate the Q-learning algorithm on a simple example called The Frozen Lake problem. Suppose the agent is a person on a frozen lake. The lake is represented by a $n \times n$ grid, with the agent starting at the field marked with S and trying to reach the goal field marked with G while avoiding holes in the ice marked with H. The input of the algorithm is a map of the lake. The task is to find a policy that will safely lead the agent from the starting state to the goal state. This task has two parts. In the first part, we have to choose a reward function that will punish the agent for falling in the holes and reward him for reaching the goal state, preferring the shortest possible path. The second part includes training a policy using the chosen reward function. The order of the field agent represents the state of the agent is currently standing in. Possible actions are $0, 1, 2, 3$ representing step left, down, right, and up respectively. A version of this problem exists, in which after taking a step in a specific direction, the agent is not guaranteed to move in that direction due to the slippery surface of the lake and can move in any direction with probability p. One possible input is in Figure 3.1. The possible reward function is

$$R(s, a, s') = \begin{cases} 1 & s \in S_{goal} \\ -10 & s' \in S_{hole} \\ -1 & s' \notin S_{goal} \cup S_{hole} \end{cases}$$

where S_{goal} is a the set of states marked with G, and S_{hole} is a set of states with character H. This kind of reward function is called a sparse reward since the agent only receives a positive reward when he reaches the goal state. Q-learning has been

Algorithm 3 The Q-learning algorithm

1: **function** QLearning
2: $\forall s \in S, a \in A : Q^{\pi_0}(s, a) = 0$
3: **for** each episode **do**
4: observe the initial state s_0
5: **for** k from 1 to T **do**
6: **if** s_k is terminal **then**
7: reset the environment
8: break
9: **end if**
10: $x \sim R(0, 1)$
11: **if** $x < \varepsilon$ **then**
12: choose action a_k randomly
13: **else**
14: choose action a_k according to the current policy π_k
15: **end if**
16: play the action a_k
17: observe the next state s_{k+1}
18: update the policy as

$$Q^{\pi_{k+1}}(s_k, a_k) = Q^{\pi_k}(s_k, a_k) + \alpha(R(a_k, s_k) + \gamma \max_{a \in A} Q^{\pi_k}(s_{k+1}, a) - Q^{\pi_k}(s_k, a_k)) \quad (3.8)$$

19: **end for**
20: **end for**
21: **end function**

Figure 3.1: One of the possible inputs for the Frozen Lake problem

shown to provide good results when applied to this problem. Although it performs well on simple problems, the Q-learning algorithm is unsuitable for complex problems, as it cannot handle a problem where action or observation space is continuous. In the following sections, we present two algorithms that deal with this problem.

3.2 Deep Deterministic Policy Gradient

Deep Deterministic Policy gradient (DDPG) [Silver et al., 2014, Lillicrap et al., 2019] algorithm is from the class off-policy model-free algorithms. It can be used on the type of problems where the action space is continuous. Its approach is similar to the Q-learning algorithm. Using the Bellman equation, the algorithm concurrently learns a Q function and a policy. The Bellman equation for the optimal policy π^* is

$$Q^*(s,a) = \underset{s' \in S}{E}\big[R(s,a) + \gamma \max_{a' \in A} Q^*(s,a')\big] \tag{3.9}$$

Suppose the optimal value function Q^* is approximated by a neural network Q_θ with parameters θ, and in the replay buffer B we have a set of transitions (s,a,s',r,d). For training the Q_θ network, the Mean Squared Bellman Error function (MSBE) is used. The equation for the MSBE function is

$$L(\theta, B) = \underset{(s,a,s',r,d) \in B}{E}\left[\Big(Q_\theta(s,a) - (r + \gamma(1-d)\max_{a' \in A} Q_\theta(s',a'))\Big)^2\right]. \tag{3.10}$$

The term $(r + \gamma(1-d)\max_{a' \in A} Q_\theta(s',a')$ is called the target since we want to minimize the difference between $Q_\theta(s,a)$ and the target. However, the MSBE function can not directly train Q_θ, as both Q_θ and the target depend on the parameter θ, and this would destabilize the training. Therefore, another neural network $Q_{\theta_{tgt}}$, called the target network, is used for computing the error function. However, when computing $\max_{a' \in A} Q_{\theta_{tgt}}(s',a')$ in equation 3.11, another problem arises. As the action space is not a finite set, we cannot simply iterate through all possible actions, and since this

subroutine is used every time we are looking for the optimal action, it needs to be fast, so grid search is not suitable. Since the action space is infinite, it is possible to use another neural network μ as a function approximator for the function $\arg\max_{a' \in A} Q(s', a')$. Also, for similar reasons as with the $Q_{\theta_{tgt}}$ network, the target policy network μ_{tgt} is used. The target networks are then updated using parameters from the main networks by Polyak averaging formula. In RL terminology, the policy network is often called the actor since it finds the best possible action, and the Q network is called the critic since it evaluates the actions taken in each state.

The error function 3.10 rewritten using μ_{tgt} and $Q_{\theta_{tgt}}$ is

$$L(\theta, B) = \underset{(s,a,s',r,d) \in B}{E}\left[\left(Q_\theta(s,a) - (r + \gamma(1-d)\max_{a' \in A} Q_{\theta_{tgt}}(s', \mu_{tgt}(s')))\right)^2\right] \tag{3.11}$$

The loss function for the policy network is

$$L_p(\mu, B) = \frac{1}{|B|}\sum_{s \sim B} Q_\theta(s, \mu(s). \tag{3.12}$$

Denote as $\text{clip}(x, u, v)$ the function that returns a vector x clipped into interval (u, v), and ε is a random variable with normal distribution. ε is added to the chosen action, as it helps the algorithm explore a much bigger space of actions. The pseudocode is available as Algorithm 4. Hyperparameters n_0, n_1, ρ represent the maximal size of the replay buffer, maximal number of executed steps, and update rate from Polyak averaging.

3.3 Twin Delayed DDPG

The Twin Delayed DDPG (TD3) algorithm [Fujimoto et al., 2018] is a direct successor of DDPG. While DDPG achieves great performance in most cases, on some types of problems, it fails to learn the optimal policy due to overestimating Q values and exploiting these errors. The TD3 algorithm addresses this issue by implementing a few critical improvements. Firstly, the action is selected using the formula

$$a(s) = \text{clip}(\mu_{tgt}(s) + \text{clip}(c, -c, \varepsilon) + a_{low}, a_{high})$$

where ε is a random variable with normal distribution and c is a hyperparameter. This technique, called target policy smoothing, helps the algorithm avoid exploiting sharp peaks in the approximated Q function. The second improvement is called double Q-learning [van Hasselt et al., 2015]. The algorithm learns two Q functions concurrently instead of one and chooses the smaller one to compute the target term in the Bellman equation. Again, this helps to circumvent overestimation. The TD3 algorithm contains two main Q networks $Q_{\theta_0}, Q_{\theta_1}$ and two target Q networks $Q_{\theta_{tgt,0}}, Q_{\theta_{tgt,1}}$.

Algorithm 4 DDPG

1: **function** DDPG
2: initialize target network parameters $\theta_{tgt} \leftarrow \theta, \mu_{tgt} \leftarrow \mu$
3: $n \leftarrow 0$
4: **while do**$n < n_0$
5: $n \leftarrow n + 1$
6: observe state s, select action $a = \text{clip}(\mu(s) + \varepsilon, a_{low}, a_{high})$
7: execute a in the environment
8: observe new state s', reward r, and flag d signalling whether s' is terminal
9: store the transition (s, a, r, s', d) in the replay buffer
10: if s' is terminal, reset the environment
11: **if** the replay buffer has sufficient size **then**
12: **for** i from 1 to n_1 **do**
13: sample a batch of transition B_1 from B
14: update Q_θ network by gradient descent and error function from 3.11
15: update μ network using gradient ascent and loss function from 3.12
16: update target networks

$$\theta_{tgt} \leftarrow \rho\theta_{tgt} + (1 - \rho)\theta, \mu_{tgt} \leftarrow \rho\mu_{tgt} + (1 - \rho)\mu$$

17: **end for**
18: **end if**
19: **end while**
20: **end function**

The target term is computed as

$$y(r, s', d) = r + \gamma(1 - d) \min_{i \in \{0,1\}} Q_{\theta_{tgt,i}}(s', \mu_{tgt}(s'))$$

and both networks are trained by gradient descent toward this target. The error functions are

$$L(\theta_i), D) = \underset{(s,a,s',r,d) \in B}{E} \left[\left(Q_{\theta_i}(s, a) - y(r, s', d)) \right)^2 \right]. \tag{3.13}$$

Then the policy is updated using learned by maximizing Q_{θ_1}, using the loss function

$$L_p(\mu, B) = \frac{1}{|B|} \sum_{s \sim B} Q_{\theta_1}(s, \mu(s)). \tag{3.14}$$

The last technique used in TD3 is delayed policy updates. While in DDPG, the target networks are updated every time the main networks are updated, in TD3, the target networks are updated less frequently, leading to better performance. The paper [Fujimoto et al., 2018] recommends one update for every two Q-function updates. Other than that, the algorithm is the same as DDPG.

3.4 Proximal Policy Optimization

The Proximal policy optimization algorithm (PPO) belongs to the class of on-policy algorithms. Instead of using a replay buffer, it explores by sampling actions using the latest version of the stochastic policy π_θ, where θ is a neural network approximator. Define as $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$ the advantage function computed using policy π and

$$L(s, a, \theta', \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta'}(a|s)} A^{\pi_{\theta'}(s,a)}, g(\varepsilon, A^{\pi_{\theta'}}(s, a), \right) \tag{3.15}$$

where

$$g(\varepsilon, A) = \begin{cases} (1 + \varepsilon)A & A \geq 0 \\ (1 - \varepsilon)A & A < 0 \end{cases}$$

and ε is a hyperparameter.

PPO updates the policy via

$$\theta_{k+1} = \arg\max_{\theta} \underset{s,a \sim \theta_k}{E} \left[L(a, s, \theta_k, \theta) \right] \tag{3.16}$$

By means of the stochastic gradient ascent, PPO also uses a second approximator ϕ to learn the V function, which is then used to compute the advantage function. The pseudocode is provided as Algorithm 5.

Algorithm 5 PPO

1: **function** PPO
2: initialize parameters of the neural networks θ_0, ϕ_0
3: **for** i in 1...n_0 **do**
4: collect set of trajectories D_k by running policy π_{θ_k} in the environment
5: compute expected reward estimates $\hat{R}_t$
6: compute advantage estimates based on the current value function V_{ϕ_k}
7: update the policy using the rule 3.16
8: update the value function network by gradient descent using the loss function

$$L_{loss}(\phi, D_k) = \frac{1}{|D_k|T} \sum_{\tau \in D_k} \sum_{t=0}^{T} (V_\phi(s) - \hat{R}_t)^2$$

where T is the episode length
9: **end for**
10: **end function**

3.5 Soft Actor–Critic

The Soft Actor–Critic algorithm (SAC) [Haarnoja et al., 2018, Haarnoja et al., 2019] uses a rather different approach than DDPG and TD3, although it incorporates some techniques from these algorithms. The distinction is that SAC employs the entropy regularization idea. While trying to find the optimal policy, it also considers how much information will reaching a specific state provide. This helps the agent to explore the state space more and can give better results in the long-term horizon. The amount of information is measured by a quantity called entropy. For a random variable x with probability density function P, the entropy H of x is defined as

$$H(x) = -\mathbb{E}\big[\log P(x)\big].$$

After each step, the agent gets a bonus reward proportional to the entropy of the policy at that timestep. The optimal policy is then defined as

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \beta \underset{a \in A}{H}(a|s_t)\right)\right]$$

where β is a hyperparameter called the trade-off coefficient. Increasing β means the agent will try to explore more. In the case of episodic learning, the infinity would be replaced by the episode length.

The equations for value functions become

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \beta \underset{a' \in A}{H}(a'|s_t)\right) \middle| s_0 = s\right]$$

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \left(R(s_t, a_t, s_{t+1}) + \beta\gamma \underset{a' \in A}{H}(a'|s_t) \right) \middle| s_0 = s, a_0 = a \right].$$

The Bellman equation for the state-action value function is

$$Q^{\pi}(s,a) = \mathbb{E}\left[R(s,a,s') + \gamma V^{\pi}(s') \right].$$

SAC employs the double Q-learning trick from TD3 and updates the target networks using Polyak averaging. The error function is MSBE, with the target term expressed as

$$y(r, s', d) = r + \gamma(1 - d)(\min_{i \in 0,1} Q_{tgt,i}(s, a') - \beta \log \pi(a'|s'))$$

where π is the current learned policy and $a' \sim \pi$. While optimizing the policy, the SAC algorithm makes use of the reparametrization trick, meaning the action is computed by the formula

$$a' = f(\varepsilon, s)$$

where f is a deterministic function, and ε is a random noise with normal distribution. For f, authors of the algorithm recommend the hyperbolic tangent. The loss function optimized by the policy network is

$$L_p(\mu, B) = \frac{1}{|B|} \sum_{s \sim B} \big(\min_{i \in \{0,1\}} Q_{\theta_{tgt,i}}(s, a') - \beta \log \pi_{\mu}(a', s) \big). \tag{3.17}$$

Chapter 4

Implementation

4.1 Reinforcement learning tools

For the implementation, we chose the programming language Python because of its broad ecosystem of libraries and well-documented API for controlling the CoppeliaSim simulator. All the details regarding versions of the libraries, the programming language and the simulator are available in the appendix with source code and data.

The reinforcement learning environments we implemented inherit from the Env class from the gym package. Gym [Brockman et al., 2016] is a library for creating and implementing various RL environments containing many environments ready to use out of the box. The team maintaining the gym package has moved all future development into Gymnasium, an even more extensible framework that supports features like multi-agent environments. Examples of environments from the gym package can be seen in Figures 4.1 and 4.2.

We used the implementation of RL algorithms from the library stable-baselines3 [Raffin et al., 2021]. The library offers a wide variety of highly parameterized training

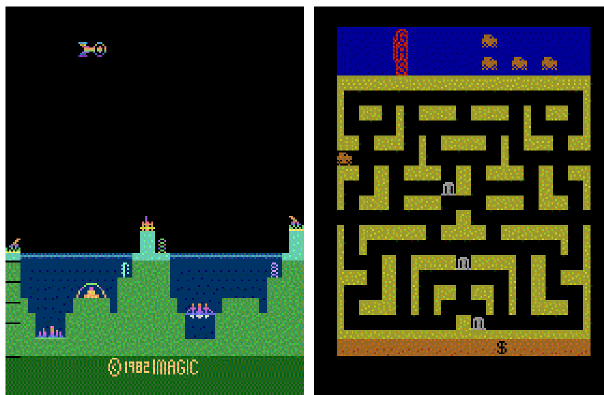

Figure 4.1: Atari games environments from the gym package [OpenAI, a]: Atlantis and bank heist environment.

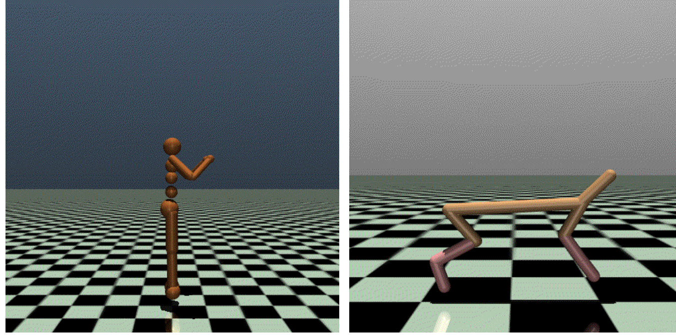

Figure 4.2: MuJoCo robotic environments from the gym package [OpenAI, b]: humanoid and half cheetah environment.

algorithms. Extensive documentation and tutorials are also very helpful. For neural network management, the library employs PyTorch [Paszke et al., 2019], a deep learning library founded by Facebook. PyTorch is written almost entirely in C++ but has Python bindings, ensuring great performance and convenience.

All environments derived from the Env class must implement step, reset, render, and close methods and define observation and action space, both derived from the Space class from the gym package.

We created an environment that contains obstacles to train a safe path planning model. All obstacles we used in training are cuboids and can be represented by a vector of 6 numbers - position and lengths of edges. We implemented two types of obstacles - non-variable and variable. Non-variable obstacles do not change during the training procedure and are not included in the state of the environment. Variable obstacles change state after each episode. The new state for each obstacle is randomly chosen from a uniform distribution passed as a parameter. Then, the state of variable obstacles is incorporated into the state of the environment. State of the environment in the timestep j is then expressed as

$$\boldsymbol{s}_j = (\boldsymbol{t}, \boldsymbol{\theta}_j, \boldsymbol{v})$$

where $\boldsymbol{t}$ is the target position in the 3D space, $\boldsymbol{\theta}_j$ is a vector of joint values and $\boldsymbol{v}$ is vector representing state of all variable obstacles during the current episode.

Step method takes action as a parameter, executes the action in the environment, and returns the tuple (s, r, d, i) where s is the next state, r is the respective reward, d is a boolean variable signaling whether the episode has ended, and i is a dictionary with debugging information. The episode may end in two ways - the maximum number of steps has been exceeded, or the target has been reached. An action in our environment represents a vector of joint velocities during the next timestep. In case of a collision, punishment constant `collision_reward` is added to the classical reward function. The reset method resets the environment to a random configuration and returns the

new state. It was not useful to implement the render method, as the rendering update happens in the step function in this case. The close method should release any resources owned by the environment, such as buffers, file descriptors, or child processes. In our case, it means closing the simulator. The pseudocode for the step and reset methods is available here 6. The number of generated random actions and the maximum number of steps per episode are parameters provided to the environment constructor.

Algorithm 6 Pseudocode for step and reset methods

1: **function** STEP(a)
2: increase number of steps
3: set the joint velocities from a
4: execute timestep in the simulator
5: observe a new state s, and a reward r
6: observe d
7: observe info i (no debugging information is provided)
8: return (s, r, d, i)
9: **end function**
10: **function** RESET
11: set the number of steps to 0
12: reset joint values to initial position
13: generate p random actions
14: reset the position of the target
15: reset the position of the obstacles
16: execute the generated actions
17: return the current state
18: **end function**

We implemented 4 reward functions, all based on the idea of a sparse reward, which means the agent receives small punishment for every step, that does not lead to a goal state, and a higher reward $f(p)$ after reaching the goal state, where f is a function and p is the current path of the robot. The shape of the f function may help the agent to learn better and choose more effective paths. We will call f the boost function.

The boost functions that we proposed are

$$f_{const}(p) = c \tag{4.1}$$

$$f_{joint}(p) = c - JD(p) \tag{4.2}$$

$$f_{cartes}(p) = c - CD(p) \tag{4.3}$$

$$f_{orient}(p) = c - OC(p) \tag{4.4}$$

where c is a hyperparameter.

We will denote the sparse reward function created using boost function f_{const} as R_{const}, analogically for f_{joint}, f_{cartes}, f_{orient}.

The implemented environment is extensible and generic, so for any robotic arm supported by PyRep, it is possible to create an RL environment. All that is needed is to derive from the ArmEnv class and pass the desired robotic arm class as a parameter to the parent constructor.

4.2 Evaluation

Suppose our RL model has found a path. To determine this path's effectiveness, we need to know another, preferably optimal path, which could serve as a baseline. The Pyrep package, used for controlling the simulator, incorporates several motion planning algorithms, including PRM and RRT, described earlier. The problem is that algorithms from this package have another parameter, determining the final rotation of the gripper. Euler angles can represent this parameter. Thus, functionality from the PyRep package cannot be directly used for finding the shortest path.

Suppose *getpath* is a function with parameters $(\boldsymbol{v}, \boldsymbol{t}, \boldsymbol{e})$, where $\boldsymbol{v}$ is a vector of joint angles, $\boldsymbol{t} = (t_x, t_y, t_z)$ is the position of the target, and $\boldsymbol{e}$ is a vector of Euler angles, that finds a path to the target using the chosen algorithm. Suppose we want to find the path that minimizes the cartesian distance function 2.1. Then it is possible to perform a grid search on a space of Euler angles. Define $E = \left\{ \frac{2k\pi}{m} \,|\, 1 \leq k \leq m \right\}, m \in \mathbb{N}$, where $\frac{2k\pi}{m}$ is a search step. Then the shortest path found via grid search is

$$getpath(v, t, e_{min})$$

where

$$e_{min} = \arg\min_{e \in E} CD(getpath(v, t, e)).$$

The complexity of the search grows with the term m^3. Another problem is that after a certain number of calls *getpath*, the CoppeliaSim simulator gets stuck. We have been unable to resolve this issue. Therefore we implemented a script that restarts the search if it has been more than 10 seconds from the last call *getpath*.

We generated $N_{cfg} = 500$ random configurations and, for each, found the shortest path using the approach described above and both PRM and RRT. For possible values of m, we chose $m = 12$, and $m = 24$. This increase in the grid density brought 10% improvement in the path length on average. Paths found using this approach were used to evaluate the effectiveness of trained models.

Assume a trained model. For each of the N_{cfg} random configurations, the model is used to find a path for the robotic arm. We repeated this process 10 times for each configuration.

Suppose p_i, r_i are the shortest path the PRM and RRT algorithms find, respectively, for the random configuration number i. Further, suppose M is the trained model and m_i is the mean length of the paths found by the model for the random configuration number i. Analogically, m'_i is the mean length of the paths of the gripper for the random configuration number i. Then, define the following measures for measuring smoothness.

$$\rho_{joint} = \frac{1}{2N_{cfg}} \left(\sum_{i=1}^{N_{cfg}} \frac{JD(p_i)}{m_i} + \sum_{i=1}^{N_{cfg}} \frac{JD(r_i)}{m_i} \right) \tag{4.5}$$

$$\rho_{cartesian} = \frac{1}{2N_{cfg}} \left(\sum_{i=1}^{N_{cfg}} \frac{CD(p_i)}{m'_i} + \sum_{i=1}^{N_{cfg}} \frac{CD(r_i)}{m'_i} \right) \tag{4.6}$$

Greater values of these measures signalize the model is better at finding effective paths. Moreover, define

$$\rho_{find} = \frac{n}{m} \tag{4.7}$$

where n is the number of successful attempts and m is the number of all attempts. This value tells how often the model is able to find a path. The optimal value for this metric is 1.

For measuring safety, we will generate 1000 random configurations and use a trained model to find a path for these configurations. We define a measure

$$\rho_{collide} = \frac{n_{collide}}{n_{all}} \tag{4.8}$$

where n_{all} is the number of epidsodes and $n_{collide}$ is the number of collisions.

4.3 Installation and deployment

To run training on a desktop computer, install necessary dependencies from the file `requirements/dev.txt` in the current Python environment, download CoppeliasSim, and add lines

```
export COPPELIASIM_ROOT=PATH/TO/COPPELIASIM/INSTALL/DIR
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$COPPELIASIM_ROOT
export QT_QPA_PLATFORM_PLUGIN_PATH=$COPPELIASIM_ROOT
```

to your bash profile, and reload bash. After that, you can simply run `src/main.py` script.

For deployment to be as straightforward as possible, we decided to use Docker. Docker is a platform designed to help build, share, and deploy all kinds of applications. It is available on all major platforms. Many software tools use Docker to run painlessly because of its scalability and flexibility. In addition, Docker is completely open-source.

Key concepts are image and container. Image can be thought of as a read-only template containing instructions on how to run containers. Image can contain any operating system supported by the host platform. All build instructions are written in the file called Dockerfile. Docker allows developers to create custom images, save and share them via a Docker repository service. A container is an instance of a particular image while ensuring virtualization, which means faults inside the container cannot harm the host. Docker also offers port mapping, directory mapping, and communication between containers.

To run the training and evaluation process, we chose Ubuntu 22.04 operating system. The Dockerfile to build the image for training the Panda robot is supplied with the source code. The CoppeliaSim simulator runs a graphical user interface, which means it cannot be run directly from the server command line. Instead, we used a software tool called xvfb, X virtual frame buffer, a display server implementing X11 protocol in virtual memory. Xvfb can run a GUI program without display.

To deploy on a server, create a Python virtual environment and install packages from the file `requirements/production.txt`. Then run

```
$ bash scripts/start-docker.sh
```

.

Chapter 5

Results

We tested the implemented environment using 4 training algorithms from the stable-baselines3 library, namely DDPG, TD3, SAC, and PPO. Results were compared in terms of effectiveness with the results of RRT and PRM methods using the defined metrics. We also trained models to optimize safety. Trained models with the best performance are available in the source code repository along with the test datasets. All used neural networks share the same internal architecture. Specifically, each has two hidden layers with 100 neurons. In addition, we used hyperbolic tangent as an activation function.

5.1 Evaluating smoothness and effectiveness

We used 4 training algorithms from the stable-baselines3 library, namely DDPG, TD3, PPO, and SAC. We have used all implemented reward functions and trained every combination of an algorithm and a reward function. All models were trained using 500000 training steps. The maximum episode length value was set to 50 steps. Initially, the parameter `max_speed`, representing the maximal joint velocity of a robot, was set to 0.2. In figures 5.1 and 5.2 are plotted averaged rewards from the training. DDPG, TD3, and SAC algorithms converged to approximately similar results. The PPO model failed to explore enough and achieve a positive reward. The probable cause of this is the fact that PPO is an on-policy algorithm, and more steps are needed to achieve positive reward with setting `max_speed=0.2`.

Table 5.1 shows all results of the training with setting `max_speed=0.2`. The PPO algorithm performed consistently the worst with every reward function. TD3 and SAC performed the best, achieving value $\rho_{find} = 1$ while finding relatively effective paths. It can also be observed that $\rho_{joint} < \rho_{cartesian}$ is true for all models. The probable cause of this is that ρ_{joint} incorporates accumulated loss from all joints, while $\rho_{cartesian}$ contains only loss from the gripper. The best model was trained using the SAC algorithm and

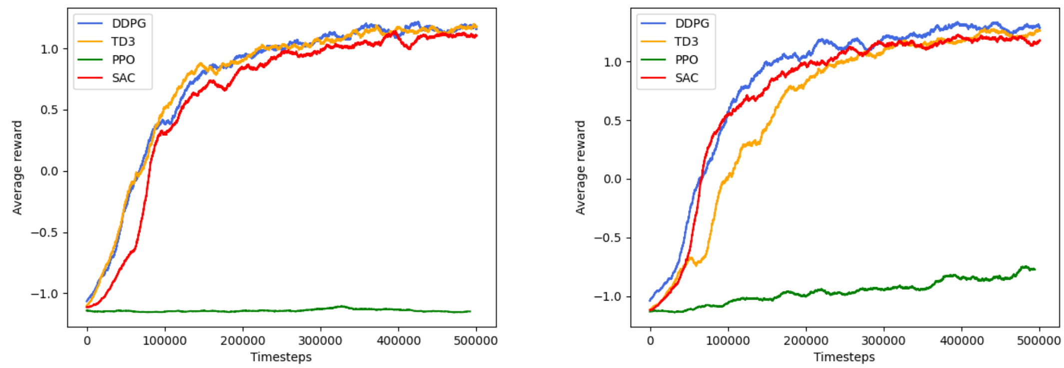

Figure 5.1: Averaged reward after 10000 steps from training with R_{joint} and R_{const} reward functions using `max_speed=0.2`

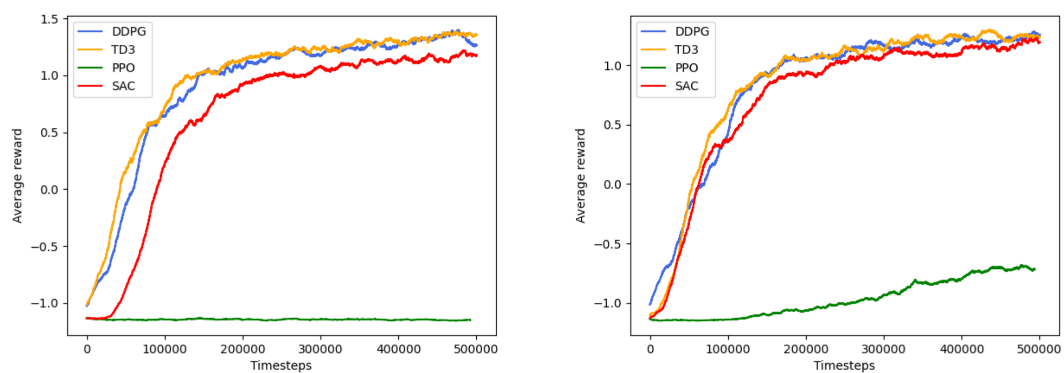

Figure 5.2: Averaged reward after 10000 steps from training with R_{cartes} and $R_{quatern}$ reward functions using `max_speed=0.2`

Algorithm	reward function	ρ_{joint}	$\rho_{cartesian}$	ρ_{find}
DDPG	R_{const}	0.325	0.742	0.995
TD3	R_{const}	0.366	0.744	1.0
SAC	R_{const}	0.449	0.787	1.0
PPO	R_{const}	0.212	0.659	0.513
DDPG	R_{joint}	0.397	0.771	0.965
TD3	R_{joint}	0.419	0.755	1.0
SAC	R_{joint}	0.394	0.743	0.999
PPO	R_{joint}	0.121	0.447	0.359
DDPG	$R_{quatern}$	0.320	0.747	0.998
TD3	$R_{quatern}$	0.333	0.731	1.0
SAC	$R_{quatern}$	0.414	0.752	0.997
PPO	$R_{quatern}$	0.159	0.622	0.266
DDPG	R_{cartes}	0.341	0.766	0.996
TD3	R_{cartes}	0.374	0.764	1.0
SAC	R_{cartes}	0.448	0.757	1.0
PPO	R_{cartes}	0.100	0.292	0.002

Table 5.1: Obtained measures of smoothness from training with `max_speed=0.2`

R_{const} reward function.

We also researched the possibility of achieving better performance by tuning the value of the `max_speed` parameter. Values of metrics of models trained with the setting `max_speed=1` are in Table 5.2. Averaged rewards from this experiment are in Figures 5.4 and 5.3.

As we can see, higher speed helped the PPO algorithm to learn to reach the target. However, models trained with this setting have significantly worse values of the $\rho_{cartesian}$ metric. We also explored the dependence of $\rho_{joint}, \rho_{cartesian}$ values on `max_speed` when trained with 300000 training steps. Results are shown in Figure 5.5.

As the choice of a reward function had little effect on performance, we will use only the R_{joint} reward function from now on. Since PPO performed badly, we decided to continue further training only with DDPG, TD3, and SAC. All algorithms achieved low values of metrics near `max_speed=0`. One of our hypotheses was this could be improved by increasing the episode length and the number of training steps. We trained DDPG, TD3, and SAC algorithms using 1500000 training steps with `max_speed=0.1` and episode length 200 steps. As can be seen from Table 5.3, the SAC model reached optimal ρ_{find} value while achieving $\rho_{cartesian} = 0.91$. On the other hand, DDPG and TD3 models did not improve so significantly. The probable reason for this is the exploration encouragement in the SAC algorithm. Results of this experiment are in

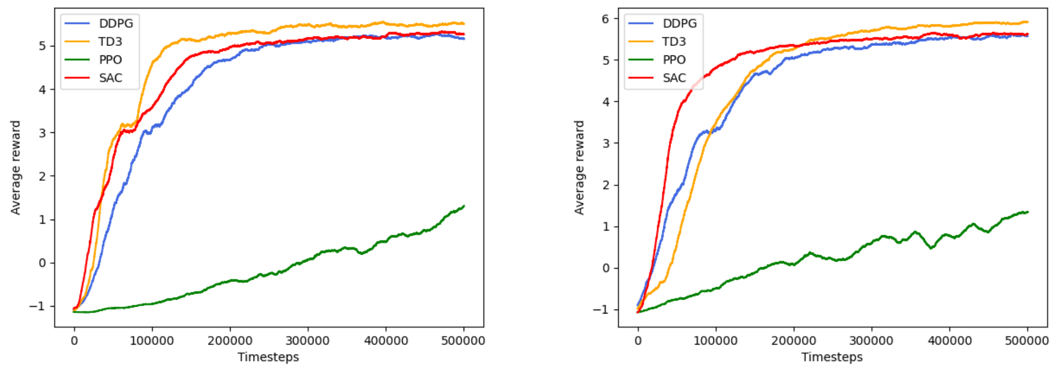

Figure 5.3: Averaged reward after 10000 steps from training with R_{joint} and R_{const} reward functions using `max_speed=1.0`

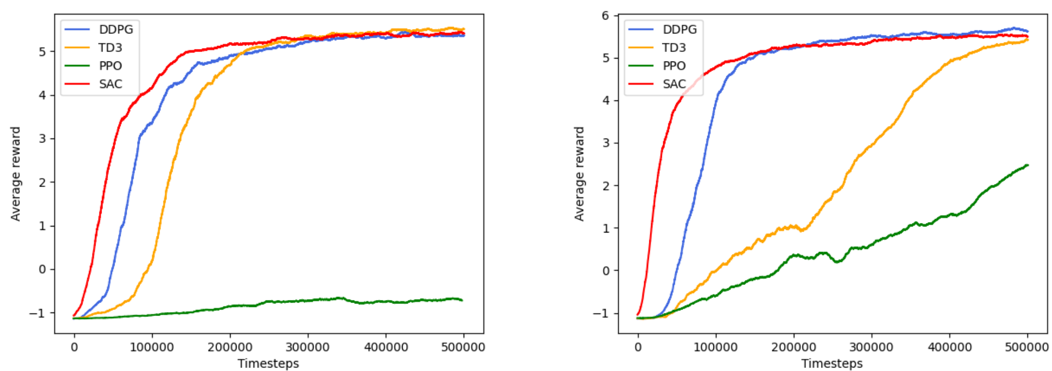

Figure 5.4: Averaged reward after 10000 steps from training with R_{cartes} and $R_{quatern}$ reward functions using `max_speed=1.0`

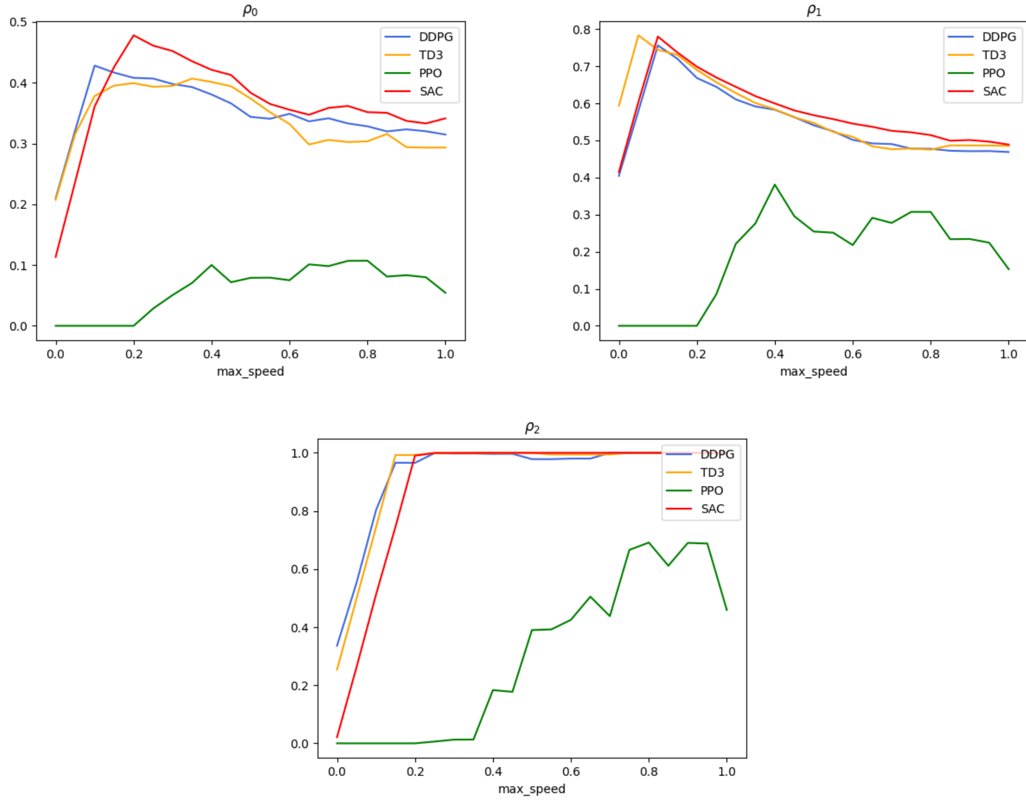

Figure 5.5: Dependence of $\rho_{joint}, \rho_{cartesian}, \rho_{find}$ metrics on `max_speed` parameter

Algorithm	reward function	ρ_{joint}	$\rho_{cartesian}$	ρ_{find}
DDPG	R_{const}	0.321	0.4914	0.9878
TD3	R_{const}	0.216	0.617	1.0
SAC	R_{const}	0.281	0.562	0.994
PPO	R_{const}	0.143	0.462	0.9972
DDPG	R_{joint}	0.255	0.478	0.1.0
TD3	R_{joint}	0.271	0.524	1.0
SAC	R_{joint}	0.285	0.476	0.985
PPO	R_{joint}	0.163	0.622	0.999
DDPG	$R_{quatern}$	0.274	0.524	0.1.0
TD3	$R_{quatern}$	0.298	0.486	1.0
SAC	$R_{quatern}$	0.277	0.483	1.0
PPO	$R_{quatern}$	0.120	0.371	0.83
DDPG	R_{cartes}	0.284	0.494	0.997
TD3	R_{cartes}	0.269	0.529	1.0
SAC	R_{cartes}	0.295	0.489	1.0
PPO	R_{cartes}	0.168	0.678	1.0

Table 5.2: Obtained measures of smoothness from training with `max_speed=1.0`

Algorithm	reward function	ρ_{joint}	$\rho_{cartesian}$	ρ_{find}
DDPG	R_{joint}	0.338	0.821	0.8246
TD3	R_{joint}	0.349	0.846	0.998
SAC	R_{joint}	0.379	0.912	1.0

Table 5.3: Obtained measures of smoothness from training with `max_speed=0.1`

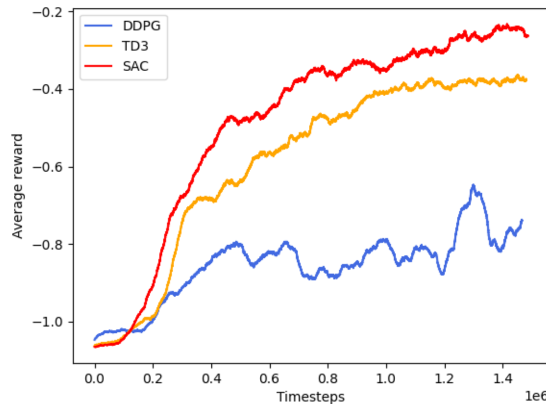

Figure 5.6: Averaged reward after 10000 steps with `max_speed=0.1`

Table 5.3. Averaged rewards from this experiment are plotted in Figure 5.6.

5.2 Evaluating safety

We implemented four experiments with non-variable obstacles and two with variable obstacles. In each experiment, we tried to train the robot to reach the target while avoiding obstacles using DDPG, TD3, and SAC training algorithms. In the first experiment is one non-variable obstacle, representing a cuboid. In the second example, there is a non-variable obstacle representing a pole. In the third example, there is an obstacle representing a wall. The target is located behind the wall. Visualization of these experiments can be seen in figure 5.2. During all experiments, we used `collision_reward=-1000`.

Results of the experiments are in Table 5.4. As can be seen from the results, all three algorithms were able to learn policy in the first experiment but failed in the second and the third one. This may be caused by wrong hyperparameter values, such as the wall being too high, which would make the agent unable to reach the target. This hypothesis is supported by the fact that whenever a model has a poor value of $\rho_{collide}$, it also has a bad value of ρ_{find}.

In the first experiment with variable obstacles, the was one obstacle representing a dynamic stick. The stick is located in the space between the target and the robot with a

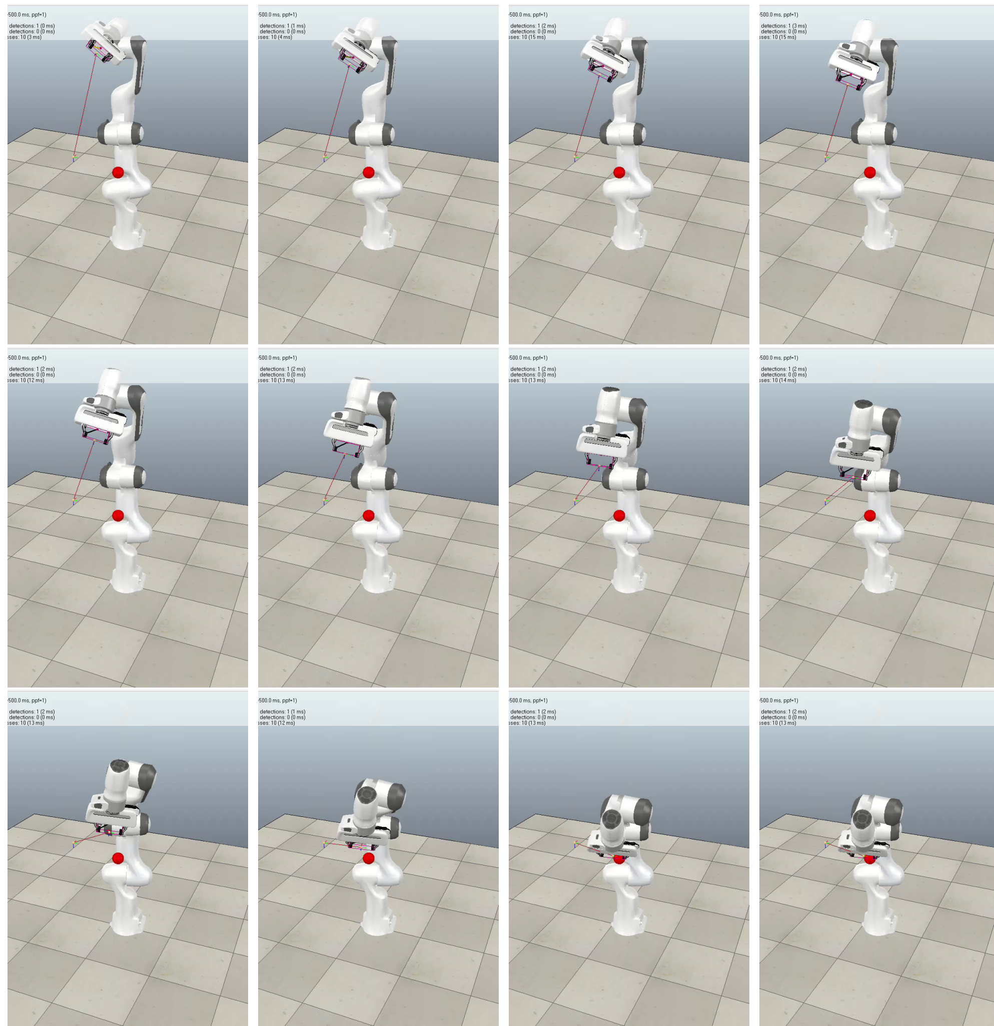

Figure 5.7: Visualization of the simulated Panda robot traversing a path found by the TD3 model. The red ball represents the target.

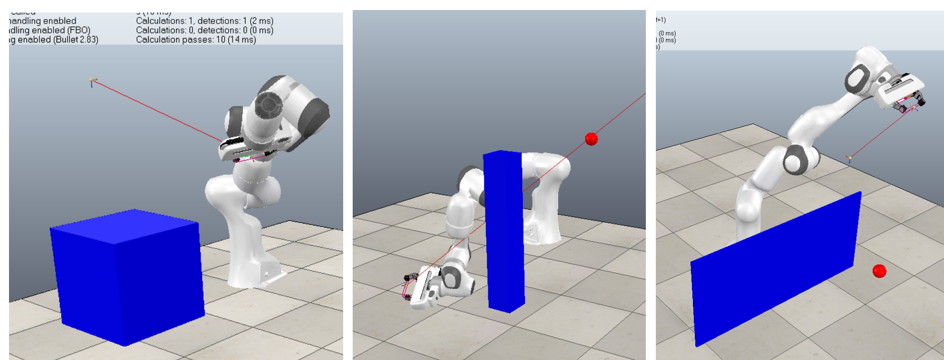

Figure 5.8: Visualization of the experiments with non-variable obstacles. The obstacles are highlighted in blue color.

Algorithm	experiment	ρ_{find}	$\rho_{collide}$
DDPG	cuboid	1.0	0.119
TD3	cuboid	1.0	0.178
SAC	cuboid	1.0	0.236
DDPG	pole	0.291	5.837
TD3	pole	0.279	5.578
SAC	pole	0.016	6.902
DDPG	wall	0.251	14.654
TD3	wall	0.035	14.592
SAC	wall	0.287	12.473

Table 5.4: Table containing values of metrics from the experiments with non-variable obstacles

Algorithm	experiment	ρ_{find}	$\rho_{collide}$
DDPG	variable stick	0.073	7.795
TD3	2 variable stick	0.077	6.435
SAC	variable stick	0.213	11.192
DDPG	2 variable poles	0.045	21.144
TD3	2 variable poles	24.784	0.141
SAC	2 variable poles	26.491	0.085

Table 5.5: Table containing values of metrics from the experiments with variable obstacles

variable z axis. In the fourth experiment, there were two poles again, but their position was changing during the training process. Visualization of these two experiments can be seen in Figure 5.2.

Results of these experiments are in Table 5.5. As can be seen, all three algorithms failed in both experiments.

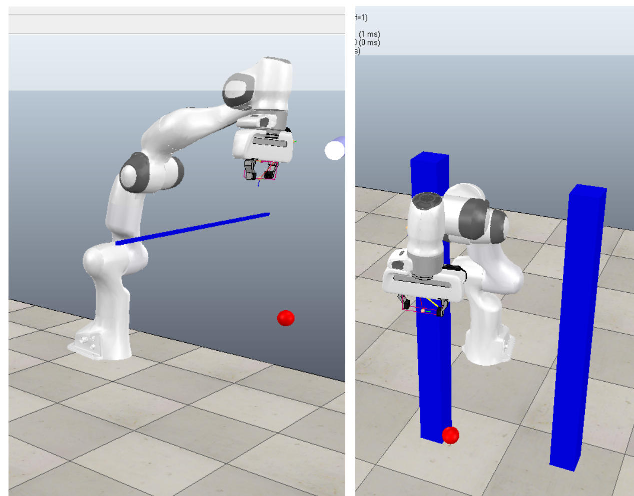

Figure 5.9: Visualization of the experiments with non-variable obstacles. The obstacles are highlighted in blue color.

Conclusion

In the thesis, we implemented a reinforcement learning environment for a robotic arm using the robotic simulator CoppeliaSim. The source code is generic and parameterized and enables anybody to create a custom RL environment for any robot supported by CoppeliaSim. We also implemented a pipeline to easily deploy training and evaluation procedures on a server.

We trained the environment with four RL algorithms. Trained models were evaluated in terms of performance and safety and compared against classical motion planning algorithms. We also researched the tuning of performance against various parameters. For example, we found out that models trained with lower maximal joint velocity require more training steps and longer episodes but find much smoother paths. However, it is impossible to directly compare our models with classical algorithms built into CoppeliaSim, as these algorithms also consider the final orientation of the gripper. Still, some of the trained models performed very well and are capable of finding smooth and effective paths.

As for the safety part, although we were able to train models capable of avoiding non-variable obstacles in some cases, in general success of the training is not guaranteed. One of the complications is that the learning process is very sensitive to changes in the state of obstacles, which means it is very difficult to find a position for obstacles in the critical area that also enables reaching a positive experience.

Future work

There are many gaps in the area of robotic reinforcement learning, and the space for improvement is extensive. The key places where the future research should be directed are

- Implementation of a vision-based agent. Implementing an agent whose input comes from a camera could significantly improve safety results
- Implementation of an RL environment for different types of robots from CoppeliaSim.
- Using the knowledge from the classical motion planning algorithms to try and improve the smoothness and safety results of the trained models. The experience

from optimal paths found by the RRT and PRM algorithms might be useful, if utilized in the right way.

Bibliography

[Bohlin and Kavraki, 2000] Bohlin, R. and Kavraki, L. (2000). Path planning using lazy prm. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, volume 1, pages 521–528 vol.1.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

[Dale and Amato, 2001] Dale, L. and Amato, N. (2001). Probabilistic roadmaps - putting it all together. volume 2, pages 1940 – 1947 vol.2.

[Emika, 2023] Emika, F. (2023). Franka emika.

[Fayjie et al., 2018] Fayjie, A. R., Hossain, S., Oualid, D., and Lee, D.-J. (2018). Driverless car: Autonomous driving using deep reinforcement learning in urban environment. In *2018 15th International Conference on Ubiquitous Robots (UR)*, pages 896–901.

[Fujimoto et al., 2018] Fujimoto, S., van Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods.

[Garg, 2023] Garg, S. (2023). Motion planning algorithms for robots.

[Gaz et al., 2019] Gaz, C., Cognetti, M., Oliva, A., Robuffo Giordano, P., and De Luca, A. (2019). Dynamic identification of the franka emika panda robot with retrieval of feasible parameters using penalty-based optimization. *IEEE Robotics and Automation Letters*, 4(4):4147–4154.

[Haarnoja et al., 2018] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor.

[Haarnoja et al., 2019] Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., and Levine, S. (2019). Soft actor-critic algorithms and applications.

[Haristiani, 2019] Haristiani, N. (2019). Artificial intelligence (ai) chatbot as language learning medium: An inquiry. *Journal of Physics: Conference Series*, 1387(1):012020.

[James et al., 2019] James, S., Freese, M., and Davison, A. J. (2019). Pyrep: Bringing v-rep to deep robot learning. *arXiv preprint arXiv:1906.11176*.

[Kavraki et al., 1996] Kavraki, L., Svestka, P., Latombe, J.-C., and Overmars, M. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580.

[Kim et al., 2020] Kim, M., Han, D.-K., Park, J.-H., and Kim, J.-S. (2020). Motion planning of robot manipulators for a smoother path using a twin delayed deep deterministic policy gradient with hindsight experience replay. *Applied Sciences*, 10(2).

[Lavalle and Kuffner, 2000] Lavalle, S. and Kuffner, J. (2000). Rapidly-exploring random trees: Progress and prospects. *Algorithmic and computational robotics: New directions*.

[LaValle, 1998] LaValle, S. M. (1998). Rapidly-exploring random trees : a new tool for path planning. *The annual research report*.

[Lillicrap et al., 2019] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2019). Continuous control with deep reinforcement learning.

[OpenAI, a] OpenAI. Openai atari environments.

[OpenAI, b] OpenAI. Openai mujoco environments.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library.

[Raajan et al., 2020] Raajan, J., Srihari, P. V., Satya, J. P., Bhikkaji, B., and Pasumarthy, R. (2020). Real time path planning of robot using deep reinforcement learning. *IFAC-PapersOnLine*, 53(2):15602–15607. 21st IFAC World Congress.

[Raffin et al., 2021] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8.

[Rohmer et al., 2013] Rohmer, E., Singh, S. P. N., and Freese, M. (2013). Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS).*

[Silver et al., 2016] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

[Silver et al., 2017] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815.

[Silver et al., 2014] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, page I–387–I–395. JMLR.org.

[van Hasselt et al., 2015] van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning.

[van Seijen et al., 2009] van Seijen, H., Van Hasselt, H., Whiteson, S., and Wiering, M. (2009). A theoretical and empirical analysis of expected sarsa. pages 177 – 184.

[Wang et al., 2019] Wang, H., Zariphopoulou, T., and Zhou, X. (2019). Exploration versus exploitation in reinforcement learning: a stochastic control approach.

[Watkins and Dayan, 1992] Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

[Yang, 2020] Yang, X. (2020). *Markov Chain and Its Applications.* PhD thesis.

[Yu et al., 2020] Yu, J., Su, Y., and Liao, Y. (2020). The path planning of mobile robot by neural networks and hierarchical reinforcement learning. *Frontiers in Neurorobotics*, 14.

Príloha A

The appendix contains the source code, trained models and the the testing datasets. Source code is also available in the Github repository.