

,



SAPIENZA
UNIVERSITÀ DI ROMA

Multi-point Path Planning for the Unicycle via Deep Reinforcement Learning

Faculty of Information Engineering, Informatics, and Statistics
Master Degree in Control Engineering

Arturo Maiani

ID number 1738271

Advisor

Prof. Giuseppe Oriolo

Academic Year 2021/2022

Multi-point Path Planning for the Unicycle via Deep Reinforcement Learning
Master Thesis. Sapienza University of Rome

© 2022 Arturo Maiani. All rights reserved

This thesis has been typeset by L^AT_EX and the Sapthesis class.

Author's email: maiani.1738271@studenti.uniroma1.it

“L'uomo tanto può quanto sa”
Sir Francis Bacon

Ringraziamenti

Ringrazio i miei genitori. Probabilmente ci vorrà del tempo per comprendere fino in fondo la fortuna che ho avuto nell'essere cresciuto da voi. L'amore incondizionato che avete mostrato nei miei confronti è un dono raro, senza di voi non sarei riuscito a raggiungere questo piccolo traguardo. Avete sempre appoggiato i miei interessi e incoraggiato le mie passioni, nonostante molte volte foste stanchi avendo lavorato tutto il giorno, mi avete ascoltato mentre parlavo di ciò che avevo trovato interessante all'università e anche se sembra una cosa da poco, non lo è stata.

Durante la triennale ho avuto la fortuna di conoscere dei compagni di banco che ho imparato a chiamare amici. Siete riusciti a strapparmi una risata anche durante gli innumerevoli momenti di disperazione mentre studiavamo in Boaga oppure in qualche auletta semivuota. Siete stati una seconda famiglia e porterò il ricordo di quelle giornate nel cuore per sempre.

Menzione speciale va a Michela, ti ringrazio per aver creduto in me e avermi aiutato ad uscire dal guscio in cui mi trovavo quando mi hai conosciuto.

Ho avuto il privilegio di passare due anni di magistrale con Chiara, Federico, Giacomo ed Edoardo. Senza di voi non sarei riuscito ad affrontare le sfide che si sono parate dinanzi a me e probabilmente non avrei più capelli attaccati al cuoio capelluto. Lavorando in gruppo con voi ho osservato la vostra concretezza, trasparenza e determinazione e ne ho fatto tesoro.

Un ringraziamento speciale va a Federico, oltre che un collega e un amico sei stato un fratello maggiore, dandomi consigli preziosi che vanno oltre lo studio. Ti ringrazio per avermi coinvolto nel progetto di "AllenaMente", certo che il suo sviluppo futuro sia in ottime mani.

Ringrazio il team di Pixies per avermi accolto durante la mia tesi, in particolare ho apprezzato moltissimo la disponibilità di Michele e di Matteo nell'aiutarmi ad entrare nei meandri di ROS.

Un sincero ringraziamento al mio gruppo di amici del liceo, con il quale ho trascorso cinque anni indimenticabili e che continua a far parte della mia vita.

Ringrazio i miei fratelli Lorenzo e Marco, i miei nonni, zii e zie (sia in cielo che in terra), le mie cugine grandi e piccole, Giovanni e Margherita. Inoltre ringrazio gli amici Ciccio, Oliver, Enrico, Riccardo Mattia, Giovanni, Simone, Filippo, Giulio, Pola e Riccardo Mattei, la cui forza d'animo è una grande fonte di ispirazione per me.

List of acronyms

WMR Wheeled Mobile Robots

RL Reinforcement Learning

DRL Deep Reinforcement Learning

DQN Deep Q-Network

MLP Multi Layer Perceptron

CNN Convolutional Neural Network

TSP Traveling Salesman Problem

TRPO Trust Region Policy Optimization

PPO Proximal Policy Optimization

SGD Stochastic Gradient Descent

Abstract

This thesis proposes a Deep Reinforcement Learning (DRL) based method aimed at finding a trajectory for the unicycle model in order for the robot to visit some goal points on the map in presence of static obstacles. These points are given a-priori and represent places where presence of litter has been statistically detected. Instead of defining an heuristic method to choose the visiting order of the points of interest and connecting them using classical control methods like Artificial Potential Fields, this work proposes a global trajectory connecting all points of interest, allowing the learning algorithm to choose the best visiting order.

This problem falls in the class of Non-Euclidean Traveling Salesman Problems. These problems are NP-hard to solve and require to deal with sophisticated optimization methods, which take into account the dynamical model and the presence of obstacles. In general, these methods must be solved for each specific configuration of goal points but are really fast. On the contrary, the proposed approach develops a policy network which requires only an initial slow training phase, but which is easier to be implemented thanks to the diffusion of DRL frameworks and libraries, such as OpenAI Gym and Stable Baselines. For simplicity reasons, the unicycle model moves according to Dubin's curves, i.e. going straight or steering left or right with fixed linear speed and angular velocity.

In addition to the task of finding an optimal trajectory, this work focuses on the effect that the reward function has on the learning problem, in particular analyzing the fact that the use of negative rewards when hitting obstacles leads to the policy remaining stuck in some local minima, preventing exploration. In addition to that, the effects of some key features of the learning algorithm are investigated.

The obtained trajectory must be considered as an approximate guideline for the robot, but does not take into account the actual presence of litter around the robot itself. This obliges to switch to a subroutine when the robot finds itself in the proximity of litter, which in a real-world scenario could be detected and classified through Computer Vision techniques. The proposed subroutine is an heuristic method based on the combination of a Traveling Salesman Problem and a cubic interpolation task.

Abstract

Questa tesi propone un metodo basato sul Deep Reinforcement Learning mirato a pianificare una traiettoria per un robot uniciclo in modo che esso visiti alcuni punti di interesse su una mappa che contiene alcuni ostacoli statici. Questi punti sono dati a priori e nel caso in questione rappresentano delle regioni su cui è stata statisticamente determinata presenza di rifiuti. Invece di pianificare una traiettoria che connette coppie di punti, come ad esempio mediante gli "Artificial Potential Fields", si propone un metodo in grado di generare traiettorie che vadano a connettere insiemi di punti, lasciando libero l'algoritmo di decidere l'ordine con cui visitarli. Questo problema rientra nella classe di "Non-Euclidean Traveling Salesman Problems", di difficoltà NP. Tali problemi richiedono l'utilizzo di sofisticati metodi di ottimizzazione matematica i quali devono gestire la presenza di ostacoli e di dinamiche nonlineari. La soluzione di tali problemi deve essere calcolata per ogni specifica configurazione di punti di interesse ma è ricavabile in tempi molto brevi. Al contrario, il metodo proposto in questa tesi mira ad ottenere una rete la quale viene allenata solamente in una fase iniziale, con tempistiche significative, ma che non richiede ulteriori modifiche. Inoltre la sua implementazione è più semplice rispetto a quella di un "Traveling Salesman Problem" grazie alla recente diffusione di framework e librerie riguardanti il Deep Reinforcement Learning come OpenAI Gym e Stable Baselines. Per mantenere una certa semplicità nell'approccio si è scelto di utilizzare come modello per l'uniciclo quello di Dubins.

In aggiunta all'obiettivo presentato finora, questo lavoro analizza gli effetti della funzione ricompensa sul processo di learning. In particolare osservando che se si utilizza una ricompensa negativa quando il robot va a sbattere contro gli ostacoli, la "policy" rimane bloccata in alcuni minimi locali, impedendo al robot di esplorare la mappa a sufficienza. Vengono inoltre analizzati gli effetti di altri parametri importanti del processo di learning come il "learning rate" oppure la dimensione della rete.

Tale traiettoria ottenuta deve essere considerata come una linea guida e non come una traiettoria esatta. Infatti in uno scenario realistico il robot sarebbe in grado di individuare se intorno a sé siano presenti dei rifiuti mediante algoritmi di Computer Vision e quindi abbandonare momentaneamente la traiettoria principale e passare ad una "subroutine" la quale sia in grado di far raccogliere al robot i rifiuti specifici. Tale legge locale si basa sulla combinazione di un Traveling Salesman Problem e di una legge di controllo per l'inseguimento di traiettoria.

Contents

1	Introduction	1
1.1	Motivations	1
1.2	Literature review	2
1.3	Thesis Structure	6
2	Pixies WMR State of The Art	7
2.1	Hardware	7
2.2	Software	9
2.3	Navigation	9
2.4	Computer Vision	10
3	Wheeled mobile robots modeling and control	11
3.1	Wheel types	11
3.2	Kinematic structures	12
3.3	Non-holonomy	12
3.4	Discretization	14
3.5	Classic Path Planning techniques	16
3.6	Trajectory tracking and Cartesian regulation	18
3.6.1	Input-Output Exact Linearization	18
3.6.2	Posture Regulation	19
4	Deep Reinforcement Learning	21
4.0.1	Reinforcement learning	21
4.0.2	Markov Decision Process (MDP)	23
4.1	Deep Q Network	25
4.1.1	Machine Learning	25
4.2	Policy Gradient Algorithms	27
4.2.1	Policy gradient derivation	27
4.2.2	Fixes that lead to real-world practicality	29
4.2.3	Variance reduction by Discounting	30
4.2.4	Vanilla Policy Gradient algorithm - pseudocode	31
4.3	TRPO and PPO	32
4.3.1	Trust Region Policy Optimization (TRPO)	32
4.3.2	Proximal Policy Optimization (PPO)	33

5 Proposed Environment Structures	35
5.1 Structure 1	35
5.1.1 State	35
5.1.2 State normalization	36
5.1.3 Action	36
5.1.4 State transition function	36
5.1.5 Reward	37
5.2 Structure 2	38
5.2.1 State	38
5.2.2 State normalization	38
5.2.3 Action	39
5.2.4 State transition function	39
5.2.5 Reward	40
5.2.6 Voronoi diagram generalization	40
6 Subroutine Control Law	43
7 Simulation Results	47
7.1 Main Routine: Structure 1	47
7.1.1 PPO-DQN-A2C Comparison	48
7.1.2 Negative Reward Comparison	50
7.1.3 Learning rate schedule comparison	51
7.2 Main Routine: Structure 2	53
7.2.1 Variable number of Goal States (17×20 map)	53
7.2.2 Neural Network Architecture Comparison (12×12 map)	56
7.2.3 ROS simulation	60
7.3 Simulations about Subroutine Control law	61
8 Conclusions and Future Works	65
8.1 Future works	65
8.2 Conclusions	65
9 Code	67
9.1 Structure 2	67
9.2 Subroutine Code	75
9.3 ROS code	80
Bibliography	89

Chapter 1

Introduction

This thesis is titled "Multi-point Path Planning for the unicycle via Deep Reinforcement Learning". This name can be explained as follows: "Multi-point" means that the objective is to let the robot visit not only one final goal state, but needs to visit many different locations on the map. As will be explained in section 3, the Unicycle is a very common way to model an autonomous mobile robot and can be intended as a robot with two actuated wheels and a third balancing wheel. Finally, "Deep Reinforcement Learning" is a broad range of algorithms developed in order to let an agent learn how to behave in an environment, in order to maximize some discounted future reward signal, defined specifically for the desired task.

1.1 Motivations

Even though Path Planning solutions for wheeled Mobile Robots (WMR) have been extensively investigated, they all tend to consider a single goal state in the task. The benefits that a Deep Reinforcement Learning (DRL) based approach for such problem is that it can be applied to an environment where some multiple goal states have been scattered, and need to be visited. DRL does not require to specify some heuristic visiting for the goal states but everything is left to the agent to learn. In addition to that, the study of a simple low dimensional task like this one allows to easily visualize the effect that a certain reward function on the learning process, for instance with respect to the agent being stuck in local minima and exhibiting nonsense behaviours.

As previously stated the main idea is that of defining a policy composed of a main routine and a subroutine. The main routine is that of finding a trajectory which allows the robot to visit all the points of interest on the map. Such map needs to be large enough for the main routine to be required, meaning that the robot cannot see all parts of the map through its camera, hence needs to be guided by an algorithm which has knowledge of the whole state of the map and not simply by a Computer Vision based planner.

If the robot detects some litter pieces during the main routine, it switches to the subroutine, which allows the robot to behave in a local deterministic way and to collect all detected litter pieces by means of a feedback linearization trajectory

tracking controller, together with artificial potential fields.

1.2 Literature review

Deep Reinforcement Learning is a broad research area which has been deeply investigated for the past seven years. This field has become of great interest after the pioneeristic article by Mnih et al. [1]. They have indeed proven that using neural networks to solve Complex Reinforcement Learning problems represented a very smart and efficient solution. As stated in the abstract of such article: "The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them."

Since then, many different variations of such algorithm have been introduced, all of them aiming at regularizing the learning process, for instance reducing the variance or reducing overestimation of certain actions. A very important work in this direction is represented by the "Double Deep Q Learning", introduced by Van Hasselt et al. [2]

The previously presented works deal with estimating the future discounted rewards in a discrete action space. This is not of interest for continuous action space problems, for instance in robotic manipulators. Therefore a very promising research direction, Policy Gradient Methods, has been investigated by defining the actor as a Neural Network, hence allowing to optimize in a continuous action space.

This methods originate from the work of [3] by R.J. Williams, [4] and has evolved into current state of the art algorithms such as Trust Region Policy Optimization (TRPO)[5] and its optimized version, Proximal Policy Optimization (PPO) [6] by J. Schulman et al. In particular the latter represents the most interesting technique, which has been developed by OpenAI labs. This method has proven to be very effective in HalfCheetah, Hopper, InvertedDoublePendulum, Reacher and Walker2d, outperforming the current state of the art algorithms such as trust region policy optimization , cross-entropy method and vanilla policy gradient with adaptive stepsize. These environments have been developed by MuJoCo (Multi-Joint dynamics with Contact) which is a proprietary physics engine for detailed, efficient rigid body simulations with contacts. MuJoCo can be used to create environments with continuous control tasks such as walking or running. Thus, many policy gradient methods (TRPO, PPO) have been tested on various MuJoCo environments.

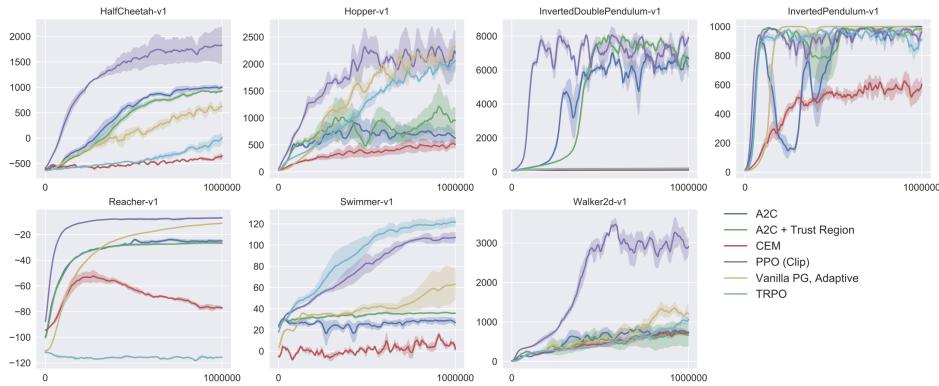


Figure 1.1. Performance of PPO versus other algorithms. [6]

DRL based trajectory planning has been analyzed by many researchers, for instance [7] developed a policy based on CNN-LSTM networks which allows the agent to navigate inside a maze by receiving as input the image from the virtual camera produced by a 3D simulator.

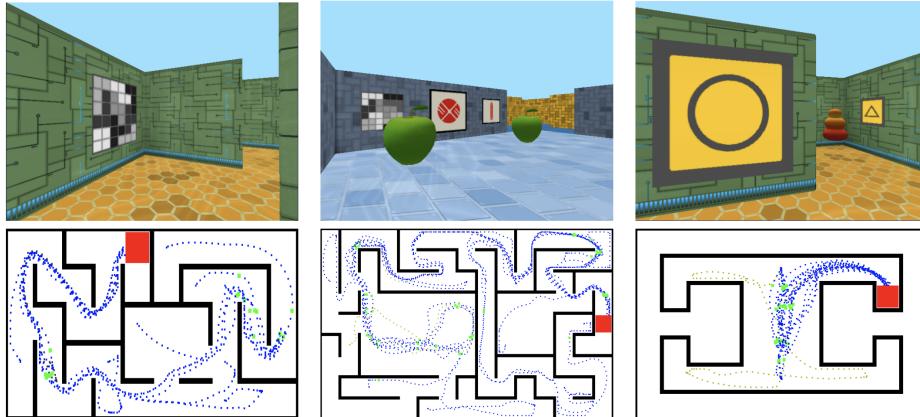


Figure 1.2. “Learning to navigate in complex environments” [7]

The work by Lei et al. [8] called "Dynamic Path Planning of Unknown Environment Based on Deep Reinforcement Learning" has investigated in a simpler fashion this research direction, proposing a solution to a path planning problem in presence of moving obstacles. This work only aims at planning for a single point at a time.

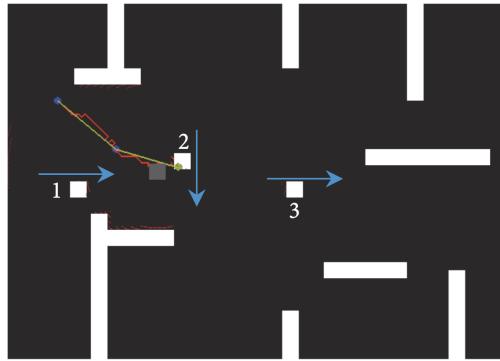


Figure 1.3. "Dynamic Path Planning of Unknown Environment Based on Deep Reinforcement Learning" [8]

In a simpler fashion [9] has developed a policy based only on a CNN to navigate an agent in a time varying grid-world, meaning that it could perform only *up, down, left, right* actions. A work that is much closer to the purpose of this master thesis has been done by [10], they developed a DRL based planning algorithm for a Unicycle model, connecting the starting position of a robot with a desired target goal, by defining a reward which increases if the agent gets closer to the goal state.

This master thesis follows this research direction but considering static environments, meaning that the position of points of interest does not vary during the cleaning task. In addition to that, the simpler network architecture of MLP has been used instead of Complex CNN or even more complex CNN-LSTM models.

The subroutine used in this thesis is based on [11], called "Traveling salesperson problems for the dubins vehicle" which solved the Dubins Traveling Salesman Problem by means of a heuristic alternating algorithm. They defined a visiting order for a set of points by solving the Euclidean Traveling Salesman Problem and then connect each ordered point with a Dubins Path. In this work, in order to avoid the complexity of finding the optimal solution among the admissible Dubins Paths a simple Cubic Interpolation is used in order to generate the local reference trajectory, which can then be tracked by the unicycle using the Exact Linearization method, i.e. by controlling a point displaced by a small amount along the sagittal axis, instead of controlling the actual center of the robot. This choice is justified by the fact that the point of the robot which needs to collect litter is placed on the anterior part, displaced w.r.t. the proper center of the robot.

Dubins Traveling Salesman Problem with Neighborhoods

This method represents the best approach found in the literature to solve this kind of problems, and has been developed in [12]. It is defined as a TSP in presence of Dubin's like kinematics, and considers a point to be visited if the agent finds itself in the proximity of it, not exactly on it. This method belongs to the area of Operations Research. The solution produced by this method are specific to the goals states distribution and needs to be computed every time. The computational time is very low, less than one second even with many goal states. The drawback

is that, in order for the method to be feasible, a suboptimal solution is proposed, which presents some non-ideal behaviours, for instance seeing the robot passing inside obstacles. This behaviour could be easily handled with the use of trajectory tracking controllers, together with Artificial Potential Fields (Vortex Fields).

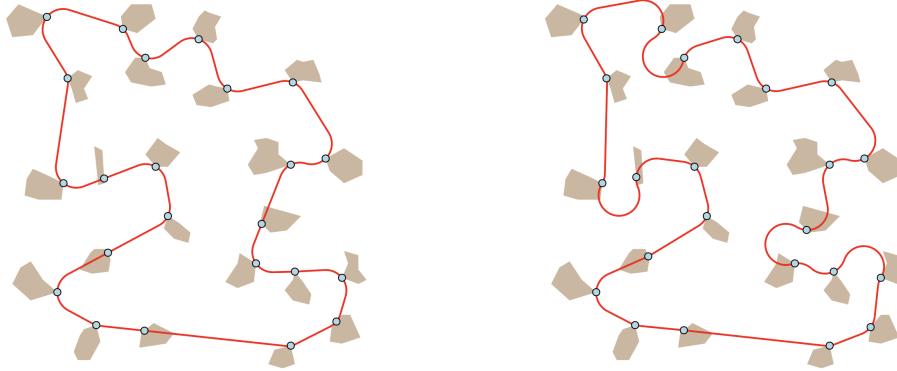


Figure 1.4. DTSPN succesful trajectories

The approach proposed in the current master thesis is incredibly slow if compared with such methods, but with the advantage of not computing a specific solution to each problem. The main purpose of this thesis is not that of beating the computational efficiency of this methods, which in unmatched, but that of investigating if DRL can solve this class of problems and which are its performances. DRL is much simpler in terms of theoretic background and can make use of many standard Python libraries, in particular Stable Baselines 2 and 3 [13].

1.3 Thesis Structure

The remainder of the thesis is organized as follows:

- 2) **Pixies WMR State of The Art:** The second chapter gives a detailed overview of the robot developed by Pixies, concerning Hardware, Software, Navigation, Computer Vision and Artificial Intelligence.
- 3) **Wheeled mobile robots modeling and control:** The third chapter concerns classical results concerning nonlinear systems and control theory applied to Wheeled Mobile Robots.
- 4) **Deep Reinforcement Learning:** This chapter deals with the theoretic aspects regarding Deep Reinforcement Learning algorithms, both Value-based and Policy Gradient.
- 5) **Proposed Environment Structures:** Next, the fifth chapter discusses the proposed environment definitions for the DRL task. Two different approaches are presented.
- 6) **Subroutine Control Law:** The sixth chapter discusses the proposed subroutine, based on Cubic Interpolation and Trajectory Tracking based on Input-Output Exact Linearization.
- 7) **Simulation results:** The seventh chapter presents the results of the simulations regarding the two proposed methods. In particular, using environment structure 1, a comparison between PPO-DQN-A2C is shown, the effects of both negative rewards and learning rate schedule are investigated. A comparison of different neural network architectures is performed using the second structure.
- 8) **Conclusions and Future Works:** The last chapter summarizes the main conclusions about this Master Thesis and proposed some new research direction in order to optimize this task.
- 9) **Code:** This chapter contains the python code used both for DRL tasks and for Subroutine Trajectory tracking.

Chapter 2

Pixies WMR State of The Art

This chapter is divided into 4 main sections as follows:

- Hardware
- Software
- Navigation
- Computer Vision

2.1 Hardware

The wheeled mobile robot built by Pixies is composed by four driving wheels, proprioceptive and exteroceptive sensors, a control board and a cleaning setup composed of two counter-rotating brushes. The power-supply is based on lithium-ions batteries which grant a 48V voltage.

The tractor mobile base is composed of four ZLTech wheels of diameter 20 cm actuated by brushless HUB SERVO MOTOR controlled by an Odrive board through Pulse Width Modulation(PWM). The wheels are provided with encoders, which allow the extraction of odometric data and diagnostics regarding the proper working

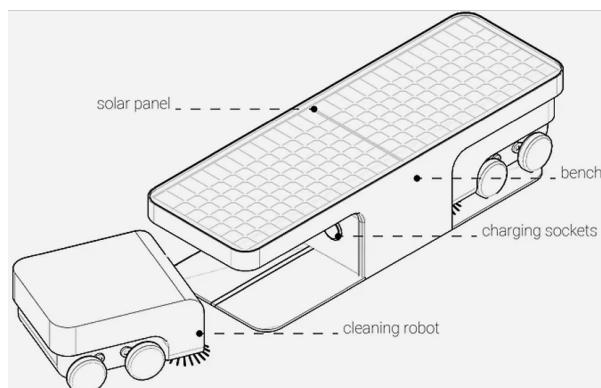


Figure 2.1. Pixies Smart Bench

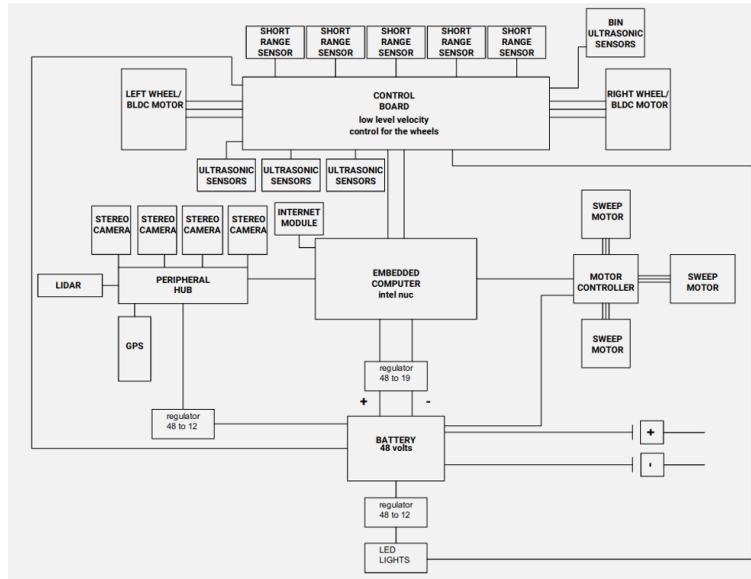


Figure 2.2. High level Scheme

of the wheels.

The set of sensors is composed of four stereo depth cameras IntelRealsense D435i which grant a stream of rgb video data provided with depth information. The technical requirements for this setup are: 60 frames per second video frequency, HD resolution, a minimum detection range of 10 cm and maximum one of 6 meters, finally an horizontal field of view of ± 90 degrees.

An other fundamental sensor is the Lidar 2D. This is a rotating laser sensor which consists of an emitter and a receiver which allow to detect obstacles and objects around the robot with a 10 Hz frequency. The chosen hardware piece is the RPLIDAR S1 360° Laser Scanner (40 m) suitable for outdoor use. The blind spots are covered thanks to other lidar sensors which analyze the proximity of the robot.

Furthermore, the GPS is fundamental since it allows the robot to locate itself in an absolute way on the map.

The internal computer which handles the communication of the robot with all its exteroceptive sensors is the mini-pc Intel Nuc. The chosen configuration is an i7 composed by 12 cores, SSD by 1TB and RAM of 16 GB.



Figure 2.3. Harware elements

2.2 Software

ROS technology is based on libraries of common programming languages e.g. Python and C++ and is meant for robot control. Thanks to ROS it is possible to develop software for position or velocity control and test it on the Gazebo Simulator, which is basically a Digital Twin of the Robot. Thanks to this simulate environment it is possible to test the robot on outdoor-like scenarios, even crowded. In addition, Computer Vision software are developed using OpenCV, Python, PCL, C++.

2.3 Navigation

The navigation software allow the robot to autonomously move inside a known environment. This is based on an accurate reconstruction of the map in which the robo is moving. This is done for instance through "gmapping", which returns a 2D map using the Lidar and odometric measures of the wheels. This is based on SLAM (Simultaneous localization and mapping) methods.

Global Planners The path planning task from a global point of view is performed by using the Dijkstra Algorithm which computes the minimum length path, using the function "navfn".

Local Planners They deal with the generation of trajectories in real time, which allow the robot to overcome obstacles. For instance one of this methods is called "

mpc_local_planner".

Recover Behaviours are the ones that the robot puts in action when he cannot find a solution for the local planner. An example of this methods sees the robot rotate on itself hoping to find some way out of the position where he is stuck.

Docking This task uses Visual Servoing techniques in order to let the robot come back under the Smart Bench in order to recharge. In order to render easier this task , AprilTag fiducials have been chosen to be mounted on the target point, in order to let the robot estimate its pose in a more confident way. The control method which has been used aims at controlling both the final position but also the orientation. The control method is called Exact Input Output Linearization and aims at controlling the trajectory of a point put on the front of the robot.

2.4 Computer Vision

The Computer Vision department is responsible for detecting litter using the multiple cameras of the robot. This detection is done through CNN's and then applies a bounding box around the profile of the detected robot on the image. This information is then used to estimate the coordinates of the litter piece relative to the robot. Then by using Visual Servoing thecniques the robot is brought near the litter piece in order to collect it.

The object detection task has been solved by using a Yolo CNN, training its last layers with a technique called transfer learning, training on a dataset which contained only images of litter pieces of interest. This task is not responsible for locating the object on the image but just for detecting if there is a litter piece.



Figure 2.4. Bottle detection and localization.

Chapter 3

Wheeled mobile robots modeling and control

This chapter deals with some standard results in modeling and control of WMR's which are necessary to give a context of the domain in which this thesis is being conducted. These results are extrapolated from the book by Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani and Giuseppe Oriolo [14].

3.1 Wheel types

A wheeled mobile robot can present three main wheel typologies:

- **Fixed wheel:** This kind of wheels has a fixed orientation and are usually used for actuation.
- **Steerable wheel:** The orientation of the wheel can be controlled, typically this is not a passive element but is rather actuated through electric motors.
- **Caster wheel:** This represents a usually passive kind of wheel, its structure can be seen in the following picture.

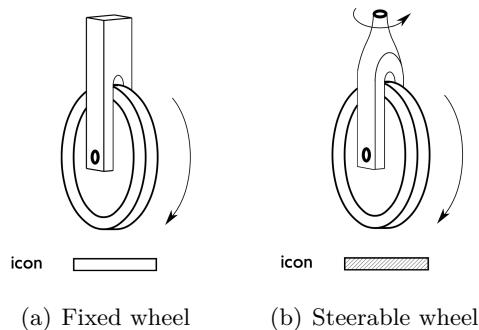


Figure 3.1. Fixed and Steerable wheels.

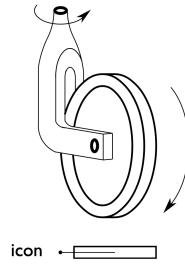


Figure 3.2. Caster wheel.

3.2 Kinematic structures

- **Differential Drive:** This robot is composed by two active wheels and a passive Caster wheel. This robot can be easily actuated with the following control signals, where b is the distance between the wheels:

$$\begin{aligned} v &= v_r + v_l \\ \omega &= b(v_r - v_l) \end{aligned}$$

- **Synchro Drives:** This is a three or four wheeled mobile robot actuated by two-motors, one rotating all the wheels with the same angle and the other one controlling the linear speed.
- **Tricycle/Car-like:** This robot is actuated by three motors, two for the rear wheels, as in the differential drive model, and one additional motor to control the orientation of the front wheel(s).
- **Omnidirectional:** This kind of robots have the possibility to steer three or more wheels, for instance of caster type. In this way a generic velocity vector for the center of mass can be generated. In this class of robots fall the ones actuated by Mecanum wheels.

3.3 Non-holonomy

Wheeled mobile robots share a peculiar mechanical characteristic, which is called non-holonomy and is strongly related with geometric analysis of the nonlinear system of differential equations that describe the kinematics of the robot, as follows.

Geometric constraint This class of constraint is only function of positional quantities:

$$h_i(q) = 0 \quad i = 0, 1, \dots, n \quad (3.1)$$

Kinematic constraint This class of constraint is function of both positional and velocity quantities. Mathematically defined as:

$$h_i(q, \dot{q}) = 0 \quad i = 0, 1, \dots, n \quad (3.2)$$

A special class of kinematic constraints are Pfaffian Constraints, linear in the velocity and defined as:

$$h_i(q, \dot{q}) = A^T \dot{q} = 0$$

Note that a geometric constraint always implies Pfaffian constraint:

$$h_i(q) = 0 \rightarrow \frac{d}{dt}(h_i(q) = 0) = \frac{\partial h}{\partial q} \dot{q} = 0$$

Of great importance is the fact that only those velocity vectors which are contained in the $\text{span}(\text{Ker}(A^T))$ satisfy the generic Pfaffian constraint. This fact imposes a limitation in the generic allowed instantaneous motion for the WMR.

Integrability of a Distribution: Frobenius Theorem

Frobenius Theorem: A distribution is integrable if and only if it is involutive. Given a non-holonomic constraint of the form:

$$A(q)^T \dot{q} = 0 \quad q \in \mathcal{R}^n$$

Define the $\text{Ker}(A(q)^T) = \Delta(q) = \text{span}(v_1, \dots, v_d)$, then in order for $\Delta(q)$ to be integrable it must be involutive, which means that:

$$[v_i, v_j] \in \Delta(q) \quad \forall i, j = 1, \dots, d \quad (3.3)$$

Where the Lie Bracket $[v, w](x)$ is defined as:

$$[v, w](x) = \frac{\partial w}{\partial x} v - \frac{\partial v}{\partial x} w$$

Frobenius theorem can be proved by the fact that [15]:

$$L_{v_i} L_{v_j} \lambda - L_{v_j} L_{v_i} \lambda = L_{[v_i, v_j]} \lambda$$

And since $L_{v_j} \lambda = 0, L_{v_i} \lambda = 0$ by hypothesis, then it means that $L_{[v_i, v_j]} \lambda = 0$, hence $d\lambda \perp [v_i, v_j] = 0$.

If a distribution is not integrable, then is possible with a combination of some maneuvers to allow the state transfer of the system from two arbitrary configurations.

Application to the unicycle The unicycle model can be derived from its corresponding non-holonomic constraint:

$$\cos(\theta)\dot{y} - \sin(\theta)\dot{x} = 0$$

By solving for the kernel of this linear operator one obtains:

$$\begin{cases} \dot{x} = \cos(\theta)v \\ \dot{y} = \sin(\theta)v \\ \dot{\theta} = \omega \end{cases} \quad (3.4)$$

Therefore the distribution which must be analyzed is:

$$\Delta(\theta) = (v_1, v_2) = \begin{pmatrix} \cos(\theta) & 0 \\ \sin(\theta) & 0 \\ 0 & 1 \end{pmatrix}$$

$$[v_1, v_2] = \begin{pmatrix} \sin(\theta) \\ -\cos(\theta) \\ 0 \end{pmatrix}$$

Which is always linearly independent from the other two. Hence this robot is controllable.

An example of a motion along the Lie bracket for the unicycle can be shown as follows:

$$\begin{cases} u_1(t) = 1 & u_2(t) = 0 \quad t \in [0, \epsilon) \\ u_1(t) = 0 & u_2(t) = 1 \quad t \in [\epsilon, 2\epsilon) \\ u_1(t) = -1 & u_2(t) = 0 \quad t \in [2\epsilon, 3\epsilon) \\ u_1(t) = 0 & u_2(t) = -1 \quad t \in [3\epsilon, 4\epsilon) \end{cases}$$

Then if one applies this kind of control law the corresponding state at $t = 4\epsilon$ becomes:

$$\begin{aligned} z &= (x, y, z)^T \\ z(4\epsilon) &= z_0 + \epsilon^2[v_1(z), v_2(z)] + \mathbf{o}(\epsilon^3) \end{aligned}$$

3.4 Discretization

Euler method This method is a forward method, meaning that the state of the system at time $t + 1$ is determined only with information from time t . This implies that if the discretization time T is too large, then a huge error will appear in the predicted state at $t + 1$. This Method is very used for small T since it is simple and gives satisfactory results.

$$\begin{cases} x_{k+1} = x_k + \cos(\theta_k)v_k T \\ y_{k+1} = y_k + \sin(\theta_k)v_k T \\ \theta_{k+1} = \theta_k + \omega_k T \end{cases} \quad (3.5)$$

Runge Kutta method This method is more refined w.r.t. Euler's Method. The prediction at time $t + 1$ is based on information both from t and $t + 1$. This is done by considering the average orientation of the robot in order to update the state variables x, y . Still not an exact method but for low T performs really well.

$$\begin{cases} x_{k+1} = x_k + \cos(\theta_k + \omega_k \frac{T}{2})v_k T \\ y_{k+1} = y_k + \sin(\theta_k + \omega_k \frac{T}{2})v_k T \\ \theta_{k+1} = \theta_k + \omega_k T \end{cases} \quad (3.6)$$

Exact integration method This method is the one which has been chosen for this kind of task since it is an exact method. In our application a $T = 1 \div 2s$ is used, hence approximations cannot be tolerated.

Since the angular velocity appears at the denominator, we need to take into account two cases. The first one is the non problematic one, since the solution does not present computational problems, i.e. division by zero.

$$\begin{cases} x_{k+1} = x_k + \frac{v_k}{\omega_k} (\sin(\theta_{k+1}) - \sin(\theta_k)) \\ y_{k+1} = y_k - \frac{v_k}{\omega_k} (\cos(\theta_{k+1}) - \cos(\theta_k)) \\ \theta_{k+1} = \theta_k + \omega_k T \end{cases} \quad (3.7)$$

In the case in which the commanded angular velocity becomes zero, one simply resorts to Euler integration method:

$$\begin{cases} x_{k+1} = x_k + \cos(\theta_k) v_k T \\ y_{k+1} = y_k + \sin(\theta_k) v_k T \\ \theta_{k+1} = \theta_k \end{cases} \quad (3.8)$$

3.5 Classic Path Planning techniques

Among the classical techniques for path planning, the following ones are the most applied. In particular the concepts covered in the following section are:

- Differential Flatness and the Chained Form
- Plan for the flat outputs
- Use parametrized inputs

Differential Flatness A generic affine nonlinear system $\dot{x} = f(x) + g(x)u$ is said to be differentially flat if exists a set of outputs y such that:

$$x = x(y, \dot{y}, \dots, y^{(r)}) \quad (3.9)$$

$$u = u(y, \dot{y}, \dots, y^{(r)}) \quad (3.10)$$

For the case of the unicycle model, the flat outputs are x, y :

$$\theta(s) = \text{Atan2}(y'(s), x'(s)) + k\pi \quad k = 0, 1 \quad (3.11)$$

$$\tilde{v}(s) = \pm \sqrt{(x'(s))^2 + (y'(s))^2} \quad (3.12)$$

$$\tilde{\omega}(s) = \frac{y''(s)x'(s) - x''(s)y'(s)}{(x'(s))^2 + (y'(s))^2} \quad (3.13)$$

This kind of method is not feasible when the cartesian trajectory degenerates into a cusp or in general, when $(x'(s))^2 + (y'(s))^2 = 0$. In addition, the choice of k is determined by the motion is forward ($k = 0$) or backward ($k = 1$).

Generalization: Chained Form A chained form is a particular structure for a nonlinear driftless system, in the form:

$$\begin{cases} \dot{z}_1 = v_1 \\ \dot{z}_2 = v_2 \\ \dot{z}_3 = z_2 v_1 \\ \dots \\ \dot{z}_n = z_{n-1} v_1 \end{cases} \quad (3.14)$$

From this structure it can be proved that this class of systems always have flat outputs, which are z_1, z_n . Therefore in a similar way the control inputs v_1, v_2 can be obtained as a function of the desired trajectory and its time derivatives.

The unicycle model can be put in this form by means of a change of coordinates:

$$z_1 = \theta \quad (3.15)$$

$$z_2 = x \cos(\theta) + y \sin(\theta) \quad (3.16)$$

$$z_3 = x \cos(\theta) - y \sin(\theta) \quad (3.17)$$

$$v_1 = \omega \quad (3.18)$$

$$v_2 = v - z_3 v_1 \quad (3.19)$$

This structure allows to use simple considerations in order to perform path planning.

Parametrized inputs Then chained form can be exploited also for an other control technique, which consists of proposing a special structure for the control inputs, function of some parameters, which must be tuned in order for the trajectory of the flat outputs to interpolate two desired points.

$$\begin{cases} \dot{z}_1 = \tilde{v}_1 \\ \dot{z}_2 = \tilde{v}_2 \\ \dot{z}_3 = z_2 \tilde{v}_1 \\ \dots \\ \dot{z}_n = z_{n-1} \tilde{v}_1 \end{cases}$$

$$\tilde{v}_1 = sgn(\Delta) \quad (3.20)$$

$$\tilde{v}_2 = c_0 + c_1 s + \dots + c_{n-2} s^{n-2} \quad (3.21)$$

with $\Delta = z_{1,f} - z_{1,i}$ and $s \in [0, |\Delta|]$. Then this problem can be solved by inverting a matrix $D(\Delta)$, which must be nonsingular.

$$D(\Delta) \begin{pmatrix} c_0 \\ c_1 \\ \dots \\ c_{n-2} \end{pmatrix} = d(z_i, z_f, \Delta) \quad (3.22)$$

Necessary condition for the inversion of $D(\Delta)$ is $\Delta \neq 0$

3.6 Trajectory tracking and Cartesian regulation

3.6.1 Input-Output Exact Linearization

In order for the generated trajectory to be followed , the method of I-O Exact Linearization is used. This choice is justified since the point which needs to track the reference trajectory is the opening in the front of the robot which contains the rotating brushes.

To achieve this, a change of coordinates is performed:

$$x_b = x + b\cos(\theta) \quad (3.23)$$

$$y_b = y + b\sin(\theta) \quad (3.24)$$

In such new output is derived w.r.t. time, it yields:

$$\begin{pmatrix} \dot{x}_b \\ \dot{y}_b \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -b\sin(\theta) \\ \sin(\theta) & b\cos(\theta) \end{pmatrix} \begin{pmatrix} v \\ \omega \end{pmatrix}$$

Being the matrix invertible and given the desired instantaneous velocities, the controller becomes:

$$\begin{pmatrix} v \\ \omega \end{pmatrix} = \begin{pmatrix} \cos(\theta) & -b\sin(\theta) \\ \sin(\theta) & b\cos(\theta) \end{pmatrix}^{-1} \begin{pmatrix} v_x^d + k_x e_x \\ v_y^d + k_y e_y \end{pmatrix}$$

This proportional controller yields an asymptotic convergence to zero steady state error in a decoupled and linear way, as follows:

$$\dot{x}_b - v_x^d = k_x(x_b - x^d) \quad (3.25)$$

$$\dot{y}_b - v_y^d = k_y(y_b - y^d) \quad (3.26)$$

Where $k_x, k_y < 0$

3.6.2 Posture Regulation

This control strategy is based on a polar coordinate system for the unicycle, as shown in the following picture:

$$\rho = \sqrt{x^2 + y^2} \quad (3.27)$$

$$\gamma = \text{Atan2}(y, x) - \theta + \pi \quad (3.28)$$

$$\delta = \gamma + \theta \quad (3.29)$$

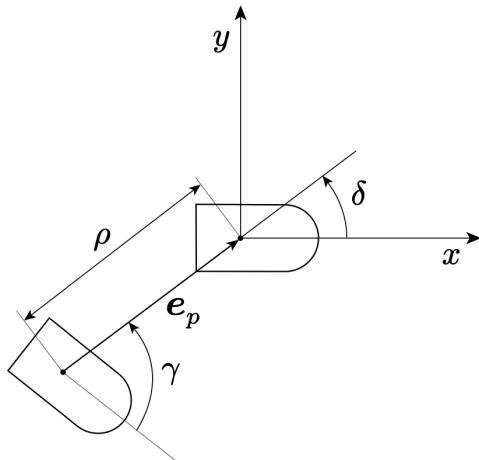


Figure 3.3. Definition of polar coordinates for Unicycle [14]

The time derivative of the state of the system takes the form:

$$\dot{\rho} = -v \cos \gamma \quad (3.30)$$

$$\dot{\gamma} = \sin \gamma v - \omega \quad (3.31)$$

$$\dot{\delta} = \frac{\sin \gamma}{\rho} v \quad (3.32)$$

The proposed control law which guarantees global asymptotic stability is the following:

$$v = k_1 \rho \cos \gamma \quad (3.33)$$

$$\omega = k_2 \gamma + k_1 \frac{\sin \gamma \cos \gamma}{\gamma} (\gamma + k_3 \delta) \quad (3.34)$$

$k_1, k_2 > 0$ This result can be proved by means of Lyapunov-like considerations. Define an Energy storage function V , positive definite, and compute its time derivative, substituting the proposed control law.

$$V = \frac{1}{2} (\rho^2 + \gamma^2 + k_3 \delta^2) \quad (3.35)$$

$$\dot{V} = -k_1 \cos^2(\gamma) \rho^2 - k_2 \gamma^2 \quad (3.36)$$

Then one needs to resort to the result given by the Barbalat Leamma, as follows.

Lemma di Barbalat If the following three conditions hold:

- 1) $V(x, t)$ is lower bounded.
- 2) $\dot{V}(x, t) \leq 0$
- 3) $\ddot{V}(x, t)$ is bounded, which implies that \dot{V} is uniformly continuous.

Then $\lim_{t \rightarrow \infty} \dot{V}(x, t) = 0$.

As it is easy to show for the former system, \ddot{V} is bounded and therefore Barbalat lemma holds. This result states that in our case, ρ, γ tend to zero. Further analysis of the closed loop system leads to the conclusion that also δ tends to zero [14].

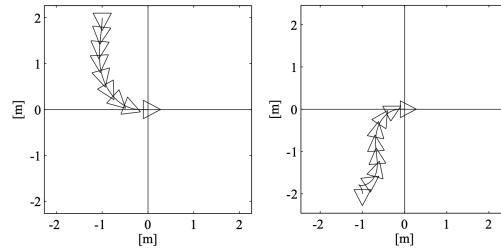


Figure 3.4. Posture regulation simulation [14]

Chapter 4

Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) stems from [1], in which Mnih et al. have introduced deep neural networks in order to learn a control policy. In order to understand why this was necessary and why it has awoken such interest we must develop a short introduction regarding reinforcement learning must be done

4.0.1 Reinforcement learning

As stated in [16] consider the fact that "A gazelle calf struggles to its feet minutes after being born. Half an hour later it is running at 20 miles per hour. This action involves interaction between an active decision-making agent and its environment, within which the agent seeks to achieve a goal despite uncertainty about its environment." The agent's actions affect the future state of the environment, thereby influencing the actions and opportunities available to the agent at later times. Correct choice requires taking into account delayed consequences of actions, and thus may require foresight or planning. The effects of actions cannot be predicted in a deterministic way, thus the agent must monitor its environment frequently and react appropriately. The gazelle must observe how each single movement of its body has produced a good effect, in the sense of allowing itself to run away from dangerous animals.

A policy represents the learning agent's way of responding to inputs and can be defined as a mapping from perceived states of the environment to actions to be taken. In simple problems a lookup table could be enough, whereas in other cases one may need to resort to neural networks or search algorithms. Policies could be stochastic, yielding probabilities for each action.

A reward signal represents the mathematical way of defining the goal of a reinforcement learning problem. The agent's purpose is to maximize the total reward it receives over a finite or infinite prediction horizon. Therefore the reward signal contains information whether events are good for the agent or not. Even reward signals could be affected by noise and in general one could not have a precise model for it. Finally a value function must be defined. The value of a state is the amount of reward that an agent could expect to gather over the future steps, starting from the current state. In simpler terms value functions represent long-term reward and for instance even if a nearby state could yield a low reward, from then on one could

be on a good path for future higher rewards.

In order to mathematically define the problem one must consider that at each time step the agent senses a state s_t which could belong both to a discrete or continuous space S and accordingly could choose an action a_t which in the same way could belong to a continuous or discrete space A .

The way in which this choice is done is through a mapping $\pi(s, a)$. The way in which $\pi(s, a)$ is defined is through an optimization problem, whose objective is that to maximize the cumulative discounted reward function:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \quad (4.1)$$

where γ is a discount factor which belongs to $(0, 1]$. Define $V(s) = E[R_t]$, which is called a State Value function, and represents the expected value of the cumulative discounted reward. When $\gamma = 1$ it means that we do not take into account that future actions could be affected by noise and therefore not be so reliable. A nice $\gamma = 0.99$ can instead weight more near future actions w.r.t. far in the future ones. An other aspect to take into account is that of stochastic state transition function $P(s_{t+1}|s, a)$, in many cases we cannot predict the exact behavior of a system mostly since we do not have access to the exact model of it. In our case of interest the problem is simplified as deterministic, therefore a probabilistic transition must not be taken into account. Indeed we consider that if we tell the robot to go straight then it will go straight exactly according to its kinematic model. A more accurate description would be that of considering a probabilistic distribution for an error on the next state, maybe caused by an imperfect or non flat ground profile.

A problem which must be taken into account when dealing with such algorithms is called "Trade-off between Exploration and Exploitation". If up until the current policy a certain portion of the state space has been explored then the agent could think that it has found the best policy, but it doesn't know that he is still ignorant about other possible global minima. In order to prevent such scenario one may think to introduce a random choice every once in a while, thus making the agent take a non deterministic action. This simple reasoning proves to be sufficient to let the agent explore unknown portions of the state space. This procedure takes the name of " ϵ - greedy" since with probability ϵ the agent takes random actions. Vice versa in the Exploitation phase (which occurs with $1 - \epsilon$ probability) the agent takes the actions which, according to its past experience, will yield the highest cumulative discounted reward. Thanks to the empiric nature of such algorithms, one does not know a-priori which kind of ϵ is best suited for a generic RL problem, this renders some trial and error necessary in order to find out which one to choose in order to not favor too much exploration, which corresponds to a completely random policy, or too much exploitation which represents a very conservative and not-willing-to-change policy.

Last but not least, as will be better explained in the next section about Markov Decision Processes, the value functions that are used in order to solve a RL problem are of two different (but related) kinds: As we have previously told, $V(s)$ represents the expected value of the cumulative discounted reward. An action-specific Value

Function can be defined as $Q(s, a) = E[R_t|s, a]$ which represents the expected discounted reward starting from state s **and choosing** action a . The goal of finding the best policy will automatically find the optimum solution for both Value functions.

Why going deep? As we will shortly see, in order to explore the state space in a complete manner, one should visit each and every possible combination of states and each action taken from those states. The cardinality of this set is:

$$\mathcal{C} = n^d a$$

where n is the number of possible state values, d is the dimension of the state space and a in the number of actions that can be taken. Consider for instance the Cart-pole balancing problem, which has $d = 4$ since the state is composed by cart position and speed and rod angular position and speed. If we want to discretize each component with 10 points, then, considering we can only have two discrete actions, right or left, then the cardinality becomes $10^4 \times 2$. If we choose $d = 100$ then it can be easily seen that this problem can become rapidly unfeasible w.r.t. the discretization interval.

4.0.2 Markov Decision Process (MDP)

A Markov Decision Process can be described by a tuple ($\langle S, A, P, R \rangle$). In order to understand what each element represents the **Markov State** is first described. The fundamental property is called the "Markov Property" and is represented by the fact that the state of the system encapsulates all the meaningful information for the environment. The current state S_t contains all the information about the past, hence the state transition function does not need to know what the previous states were in order to produce the next one. Such systems are said to be "Memoryless". The tuple ($\langle S, A, P, R \rangle$) is described as follows:

$$S = \text{finite set of states} \quad (4.2)$$

$$A = \text{finite set of actions} \quad (4.3)$$

$$P_{ss'}^a = P(S_{t+1} = s' | S_t = s, A_t = a) \quad (4.4)$$

$$R_s^a = E[R_{t+1} | S_t = s, A_t = a] \quad (4.5)$$

At the core of a MDP there is the policy function, which maps states into actions:

$$\pi(a|s) = P(A_t = a | S_t = s) \quad (4.6)$$

In order to obtain a mathematical solution to this problem, the definition of two value functions is required: the State-value function and the Action-value function. The State-value represents the expected return starting from state s , then following policy π . The Action-value represents the expected return starting from state s taking action a , then following policy π :

$$V_\pi(s) = E_\pi[F_t | S_t = s] \quad (4.7)$$

$$Q_\pi(s) = E_\pi[F_t | S_t = s, A_t = a] \quad (4.8)$$

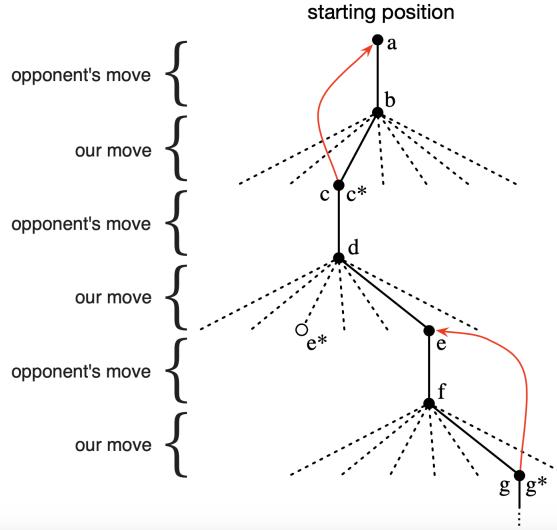


Figure 4.1. Example of a MDP in the case of a Chess Game. Taken from [16]

Where G_t is the discounted cumulative reward from time t to infinity, and mathematically defined as an infinite sum.

$$F_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (4.9)$$

Each reward is multiplied by γ^k which is a geometric decreasing sequence which tends to zero as time tends to infinity. This machinery produces the effect of weighting more near future rewards w.r.t. ones appearing far in the future.

The expression of G_t can be decomposed in a recursive form, hence allowing to rewrite (3.11) and (3.12) into the Bellman equations:

$$V_{\pi}(s) = \pi(a|s) \sum_{a \in A} (R_s^a + \sum_{s' \in S} P_{ss'}^a v_{\pi}(s)) \quad (4.10)$$

$$Q_{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(s'|a') q_{\pi}(s', a') \quad (4.11)$$

The idea, at the end, is that this MDP problem is considered solved when we achieved the optimal value for the two value function, which means:

$$V_*(s) = \max_{a \in A} Q_*(s, a) \quad (4.12)$$

$$Q_*(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V^*(s') \quad (4.13)$$

$$a^* = \arg \max_{a \in A} Q(s, a) \quad (4.14)$$

4.1 Deep Q Network

4.1.1 Machine Learning

Machine learning is a field of inquiry devoted to understanding and building methods that leverage data to improve performance on some set of tasks and is seen as a part of artificial intelligence. Machine learning models are based on sample data, known as training data, and have as purpose to predict without being explicitly programmed to do so. Machine learning algorithms are used in a wide variety of applications, such as in medicine, email filtering, speech recognition, and computer vision, where it is difficult or unfeasible to develop conventional algorithms to perform the needed tasks. This field can be divided in three categories:

- Supervised: The dataset is labeled and can then be used for Regression and classification which are two of the most encountered supervised learning problems.
- Unsupervised: the dataset is not labeled. K-means is one of the most used methods belonging to this area.
- Reinforcement Learning: The dataset is not labeled, since an action cannot be judged as good or bad simply by looking at its immediate effect. This method is based on MDP's

The main difference is that Reinforcement Learning (RL) is far-sighted and considers long-term cumulative reward.

Deep Learning.

classic techniques from Machine learning, such as Support Vector Machines(SVM) are helpful in obtaining interesting and effective models for simple tasks where the dataset is almost linearly separable. If the main goal is that of classifying photos of cats and dogs, then SVM are useless, since the task is too nonlinear and complex. Deep has proved to be extremely effective in solving such problems since it produced incredibly nonlinear models which are able to approximate any function, this result has been proved in [17], an article called "Approximation by superpositions of a sigmoidal function".

The typical definition of a Deep Learning Regression problem is the following, and is mainly defined by a Loss Function, which could be defined as a Mean Squared Error Loss Function or in many other ways. This loss function contains usually also a regularization term weighted by a parameter λ . This term prevents the network from "overfitting", which in practical terms means that the network memorizes the dataset and then when tested on unseen data it fails because it has not achieved generalization power.

The second step consists in a simple gradient descent in which the parameters are updated in a way to minimize the loss function. This second step is actually performed through the "Backpropagation algorithm" which is a geometrically efficient way to perform gradient descent.

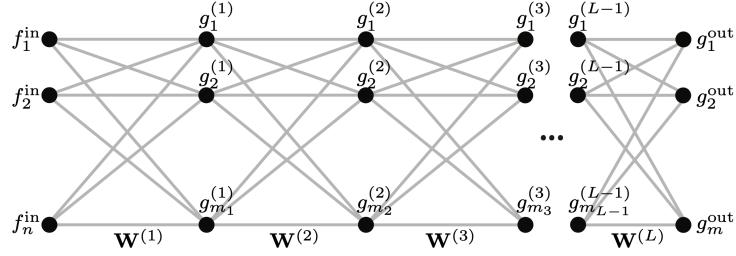


Figure 4.2. Deep Neural Network

$$L(\theta, \mathbf{x}) = \sum_{i=0}^n (y_i - f(x_i, \theta))^2 + \lambda \theta^2 \quad (4.15)$$

$$\theta \leftarrow \theta - \eta \nabla_{\theta} L(\mathbf{x}, \theta) \quad (4.16)$$

Learning rate scheduling A fundamental parameter to be tuned is the learning rate. In general a schedule is used in order to speed up the learning at the beginning and refine the policy once it has been generally defined. Some common schedules could be linear and exponential:

$$\begin{aligned} \eta^t &= \left(1 - \frac{t}{r}\right) \eta^0 + \frac{t}{r} \eta^{final} \\ \eta^t &= \frac{1}{1 + rt} \eta^0 \end{aligned}$$

The process of tuning the learning rate is by definition empirical and usually also a multi-step learning rate could be defined.

Optimizers In order to speed up the learning process, Stochastic gradient Descent has been introduced. This technique uses only a portion of the dataset in order to compute the gradient at each optimization step:

$$\theta \leftarrow \theta - \eta \nabla_{\theta} \left(\sum_{i=0}^B (y_i - f(x_i, \theta))^2 + \lambda \theta^2 \right) \quad (4.17)$$

where B is the so called "batch size". The update rule of a standard gradient descent, unrolled, takes the form:

$$\begin{aligned} x^{(1)} &= x^{(0)} - \alpha \nabla f(x^{(0)}) \\ x^{(2)} &= x^{(1)} - \alpha \nabla f(x^{(1)}) = x^{(0)} - \alpha \nabla f(x^{(0)}) - \alpha \nabla f(x^{(1)}) \\ &\dots \\ x^{(t+1)} &= x^{(0)} - \alpha \sum_{i=1}^t \nabla f(x^{(i)}) \end{aligned}$$

Momentum The idea behind Momentum, developed by [18], is to not discard past gradients but rather to accumulate them. This can be interpreted with physics like arguments, indeed this method tried to preserve some kind of momentum v . The variation of momentum is affected by $-\alpha \nabla f(x^{(t)})$ which can be seen as an acceleration, caused by the objective landscape.

$$v^{(t+1)} = \lambda v^{(t)} - \alpha \nabla f(x^{(t)}) \quad (4.18)$$

$$x^{(t+1)} = x^{(t)} + v^{(t+1)} \quad (4.19)$$

The unrolled expression for Gradient descent update using momentum becomes:

$$x^{(t+1)} = x^{(0)} + \alpha \sum_{i=1}^t \frac{1 - \lambda^{t+1-i}}{1 - \lambda} \nabla f(x^{(i)})$$

This form generalizes to more complex and fancy optimization methods like ADAM, AdaGrad, RMSprops. In particular ADAM [19] has proven to be the best option to go with, and PPO is based on it.

4.2 Policy Gradient Algorithms

Many results found in the literature prove the effectiveness of Policy Gradient (PG) algorithms w.r.t. Value based ones like DQN. For instance [20] shown that A3C, particular PG algorithm, is better than DQN in Beamrider, Breakout, Pong, Q*bert, Space invaders. These games, mostly belonging to the Atari Company have been chosen by the RL community as a baseline on which test DRL methods.

4.2.1 Policy gradient derivation

The following section aims at introducing the policy Gradient Theorem derivation, and is based on [4], pioneeristic work done by Sutton et al. In addition, a survey about Policy Gradient methods can be found in [21], which focuses in particular on robotics applications. This article [22] by Tedrake et al. shown how even with very early methods like A2C a biped robot can learn to walk.

We let τ denote a state-action sequence $s_0, u_0, \dots, s_H, u_H$. For a more compact notation we define $R(\tau) = \sum_{t=0}^H R(s_t, u_t)$. Now define the objective function that we want to maximize as:

$$U(\theta) = \mathbb{E} \left[\sum_{t=0}^H R(s_t, u_t) | \pi_\theta \right] \quad (4.20)$$

Taking the gradient w.r.t. θ gives:

$$\begin{aligned}\nabla_{\theta} U(\theta) &= \nabla_{\theta} \sum_{\tau} P(\tau, \theta) R(\tau) = \\ &= \sum_{\tau} \nabla_{\theta} P(\tau, \theta) R(\tau) = \\ &= \sum_{\tau} \frac{P(\tau, \theta)}{P(\tau, \theta)} \nabla_{\theta} P(\tau, \theta) R(\tau) = \\ &= \sum_{\tau} P(\tau, \theta) \frac{\nabla_{\theta} P(\tau, \theta)}{P(\tau, \theta)} R(\tau) = \\ &= \sum_{\tau} P(\tau, \theta) \nabla_{\theta} \log P(\tau, \theta) R(\tau)\end{aligned}$$

Such expression is equivalent to an expected value of the form:

$$\nabla_{\theta} U(\theta) = E[\nabla_{\theta} \log P(\tau, \theta) R(\tau)] \quad (4.21)$$

and can be approximated by the empirical estimate for m sample paths under policy π_{θ} :

$$\nabla_{\theta} U(\theta) \approx \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} \log P(\tau^{(i)}, \theta) R(\tau^{(i)}) \quad (4.22)$$

This expression is useful also for the fact that the gradient is not taken w.r.t. the reward function, which means that no restriction is required on the continuity properties of such reward.

In addition to that, an intuitive explanation for such gradient estimate can be given. Suppose that "good" trajectories lead to a positive cumulative reward and "bad" trajectories to negative reward, then it is clear that the overall gradient will tend to move parameters so to increase the probability of trajectories which have lead to high rewards and viceversa.

$$\begin{aligned}\nabla_{\theta} \log P(\tau^{(i)}, \theta) &= \nabla_{\theta} \log \left[\prod_{t=0}^H P(s_{t+1}^{(i)} | s_t^{(i)}, u_t^{(i)}) \pi_{\theta}(u_t^{(i)} | s_t^{(i)}) \right] = \\ &= \nabla_{\theta} \left[\sum_{t=0}^H \log P(s_{t+1}^{(i)} | s_t^{(i)}, u_t^{(i)}) + \sum_{t=0}^H \log \pi_{\theta}(u_t^{(i)} | s_t^{(i)}) \right] = \\ &= \nabla_{\theta} \sum_{t=0}^H \log \pi_{\theta}(u_t^{(i)} | s_t^{(i)}) = \\ &= \sum_{t=0}^H \nabla_{\theta} \log \pi_{\theta}(u_t^{(i)} | s_t^{(i)})\end{aligned}$$

Since the expression aims to increase the probability of a certain trajectory, and since the dynamics does not depend on the parameters θ , then the expression of the dynamics disappears when taking the gradient.

4.2.2 Fixes that lead to real-world practicality

- Baseline subtraction
- Value function estimation
- Advantage Estimation (A2C/A3C/GAE)

Baseline subtraction

Define a baseline b . The subtraction from the reward doesn't make the estimator biased, as long as the bias does not depend on parameters θ [16],[3].

$$\nabla_\theta U(\theta) \approx \frac{1}{m} \sum_{i=1}^m \nabla_\theta \log P(\tau^{(i)}, \theta) (R(\tau^{(i)}) - b) \quad (4.23)$$

Indeed the contribution to the expected value is:

$$\begin{aligned} & E[\nabla_\theta \log P(\tau^{(i)}, \theta) b] = \\ &= \sum_{\tau} P(\tau, \theta) \nabla_\theta \log P(\tau, \theta) b = \\ &= \sum_{\tau} P(\tau, \theta) \frac{\nabla_\theta P(\tau, \theta)}{P(\tau, \theta)} b = \\ &= \sum_{\tau} \nabla_\theta P(\tau, \theta) b = \\ &= \nabla_\theta \left(\sum_{\tau} P(\tau, \theta) b \right) = \\ &= b \nabla_\theta \left(\sum_{\tau} P(\tau, \theta) \right) = b \times 0 \end{aligned}$$

Where the identity $\sum_{\tau} P(\tau, \theta) = 1$ has been used since probabilities sum up to 1. The main reason for baseline introduction is variance reduction for the estimator. In addition to that it must be pointed out that in the gradient approximation expression the gradient of the log probability of the trajectory is multiplied with the reward of such trajectory. The expression as it is implies that each term inside the sum $\sum_{t=0}^H \nabla_\theta \log \pi_\theta(u_t^{(i)} | s_t^{(i)})$ gets multiplied for the reward of the whole trajectory from time $t = 0$ to $t = H - 1$. This observation has led to a modification in the gradient approximation:

$$\begin{aligned} \nabla_\theta U(\theta) &\approx \frac{1}{m} \sum_{i=1}^m \nabla_\theta \log P(\tau^{(i)}, \theta) (R(\tau^{(i)}) - b) = \\ &= \frac{1}{m} \sum_{i=1}^m \left(\sum_{t=0}^H \nabla_\theta \log \pi_\theta(u_t^{(i)} | s_t^{(i)}) \right) (R(\tau^{(i)}) - b) = \\ &= \frac{1}{m} \sum_{i=1}^m \sum_{t=0}^H \nabla_\theta \log \pi_\theta(u_t^{(i)} | s_t^{(i)}) \left(\sum_{k=t}^{H-1} R(s_k^{(i)}, u_k^{(i)}) - b(s_t^{(i)}) \right) \end{aligned}$$

A choice for the baseline could be the average collected reward:

$$b = [R(\tau)] \approx \frac{1}{m} \sum_{i=1}^m R(\tau^{(i)})$$

A more interesting approach would be that of Actor-Critic methods, in which the baseline is learned through function approximators.

Monte Carlo estimate for the baseline

This consists of a standard supervised learning problem. First collect m trajectories, then define a loss function of the form:

$$L = \frac{1}{m} \sum_{i=1}^m \sum_{t=0}^{H-1} \left(V_\Phi^\pi(s_t^{(i)}) - \left(\sum_{k=t}^{H-1} R(s_k^{(i)}, u_k^{(i)}) \right) \right)^2 \quad (4.24)$$

where Φ are the parameters for the baseline function approximator.

Bootstrap estimate for the baseline

This consists of an other standard supervised learning problem based on the Bellman Equation for V^π :

$$V^\pi(s) = \sum_u \pi(u|s) \sum_{s'} P(s'|s, u) [R(s, u, s') + \gamma V^\pi(s')]$$

First collect data in a replay buffer of the form s, u, s', r , then define a loss function and an iterative optimization procedure:

$$\Phi_{i+1} \leftarrow \min_{\Phi_i} \sum_{(s, u, s', r)} (r + \gamma V_{\Phi_i}^\pi(s') - V_{\Phi_i}^\pi(s))^2 + \lambda \|\Phi - \Phi_i\|^2$$

Where Φ are the parameters for the baseline function approximator.

4.2.3 Variance reduction by Discounting

In the gradient approximation expression, the cumulative reward was simply obtained by summing all rewards along the trajectory.

$$\frac{1}{m} \sum_{i=1}^m \sum_{t=0}^H \nabla_\theta \log \pi_\theta(u_t^{(i)} | s_t^{(i)}) \left(\sum_{k=t}^{H-1} R(s_k^{(i)}, u_k^{(i)}) - b(s_t^{(i)}) \right)$$

As a matter of fact, by introducing discounting for the trajectory reward we can reduce variance, since present action have less influence far in the future.

Substituting the expression $\sum_{k=t}^{H-1} R(s_k^{(i)}, u_k^{(i)})$ with $Q^{\pi, \gamma, k}(s, u)$:

$$Q^{\pi, \gamma, k}(s, u) = [r_0 + \gamma r_1 + \cdots + \gamma^k V^\pi(s_k) | s = s_0, u = u_0] \quad (4.25)$$

4.2.4 Vanilla Policy Gradient algorithm - pseudocode

This pseudocode is taken from the article [23] which first introduced A3C (Asynchronous Advantage Actor Critic Methods).

Algorithm 1 Vanilla Policy Gradient

- 1: Initialize parameters for policy and Value function networks
 - 2: **for** $iteration = 1, 2, \dots, M$ **do**
 - 3: Collect a set of trajectories by executing the current policy
 - 4: At each timestep in each trajectory compute the return $R_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k$ and the advantage estimate $\hat{A}_t = R_t - b(s_t)$
 - 5: Re-fit the baseline, by minimizing $\|b(s_t) - R_t\|^2$, summed over all trajectories and timesteps
 - 6: Update the policy using the gradient estimate, sum of the terms $\nabla_\theta \log \pi(u_t | s_t, \theta) \hat{A}_t$
 - 7: **end for**
-

The class of algorithms introduced until now are based on a combination of two neural networks, one called the Policy Network since it decides which action to take from the current state and the other network called Critic Network, which assigns to every state configuration a certain value corresponding to the expected cumulative reward starting from such state. To summarize, this methods are called Actor-Critic Methods and can be represented in the following way:

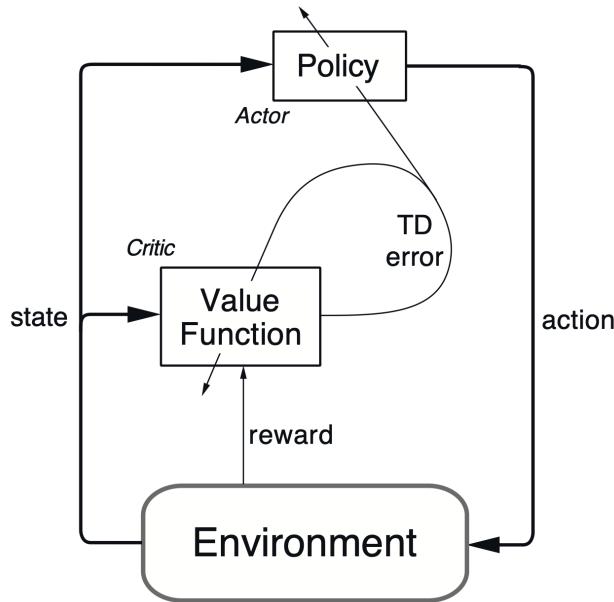


Figure 4.3. Actor Critic Methods [16]

4.3 TRPO and PPO

Deep Reinforcement Learning methods which use data to Regress some kind of Value Function are based on the hypothesis that data that the agent collects during the episodes is meaningful. This hypothesis can easily become not valid if the policy becomes corrupted in some way. An example could be that a Stochastic Gradient Descent step modifies too much the weights of a neural network, for instance bringing some activation functions in a saturation zone and hence make some units "die". This event can lead to the fact that the policy has become nonsensical and then during the next episode the actions taken do not allow the agent to explore the environment in a meaningful way.

In addition to that, Value Function based methods are optimizing a function that has some relationship actions but **is it not the actor itself**.

This two facts put together have brought great attention towards Policy Gradient methods, with the aim of defining some limits on how much each optimization step could change the policy, in order to be conservative and not allow some big numerical values for the gradient to corrupt the policy. A first very successful method developed by Schulman et al. in [5] is the Trust Region Policy Optimization.

4.3.1 Trust Region Policy Optimization (TRPO)

This method starts from optimizing a different objective function than the cumulative discounted reward. The interesting aspect is that the gradient of the two optimization methods is very similar. This Objective function contains the ratio between the probability distribution with some new parameters θ w.r.t. the old policy with parameters θ^{old} . This ratio is multiplied by the Advantage, which is defined as the difference between the actual discounted reward and the average forecast reward starting from the current state, and as we have seen previously is approximated with a Value Function neural network. The innovative aspect of this approach is the inequality constraint which has been introduced in the optimization procedure. This is called the trust region, since the ratio is not allowed to be too much distant from the scalar 1. The hyperparameter ϵ can be tuned in order to allow some exploration but not too much. The main disadvantage about this method is that it is a Second Order method, and it must be approximated in some way.

$$\max_{\theta} L(\theta) = \max_{\theta} \left[\mathbb{E}_{\pi^{old}} \left[\frac{\pi_{\theta}(a_t|s_t)}{\pi_{\theta}^{old}(a_t|s_t)} \hat{A}_t \right] \right] \quad (4.26)$$

$$\text{Subject to: } \mathbb{E}_{\pi^{old}} \left[KL(\pi_{\theta}^{old}(a_t|s_t), \pi_{\theta}(a_t|s_t)) \right] < \epsilon \quad (4.27)$$

An important aspect that must be noted is the fact that the constrained optimization problem is highly nonlinear and therefore can be very hard to solve. A much more easy problem to be solved is a non constrained one. Then the inequality constraint is brought directly inside the Objective function which needs to be maximized.

$$\max_{\theta} J(\theta) = \max_{\theta} \left[L(\theta) - \beta \mathbb{E}_{\pi^{old}} \left[KL(\pi_{\theta}^{old}(a_t|s_t), \pi_{\theta}(a_t|s_t)) \right] \right] \quad (4.28)$$

Recall that the Kullback-Lieber Divergence is a non symmetric measure of distance between Probability distributions.

$$KL(p|q)(x) = \sum_x p \log\left(\frac{q}{p}\right) \quad (4.29)$$

In order to solve such nonlinear problem we must start by taking its linear approximation:

$$J(\theta) \approx g(\theta - \theta^{old}) - \beta(\theta - \theta^{old})^T F(\theta - \theta^{old}) \quad (4.30)$$

$$g = \frac{\partial L}{\partial \theta} \Big|_{\theta^{old}} \quad (4.31)$$

$$F = \frac{\partial^2}{\partial \theta^2} \left[KL(\pi_\theta^{old}(a_t|s_t), \pi_\theta(a_t|s_t)) \Big|_{\theta^{old}} \right] \quad (4.32)$$

The quadratic part of $L(\theta)$ is negligible w.r.t. the Kullback-Lieber term hence can be avoided. One must notice that we are dealing with F a matrix of shape $\dim(\theta) \times \dim(\theta)$ and hence some technical methods must be used in order to approximate it in a numerically efficient way.

$$\theta - \theta^{old} = \frac{1}{\beta} F^{-1} g \quad (4.33)$$

Here $F(\theta^{old})$ is the Fisher Information Matrix and $g(\theta^{old})^T = \nabla \pi_\theta / \pi_\theta^{old}$ is the policy gradient.

This method is computationally expensive since it needs to compute the inverse of a huge matrix. This problem is solved by using Conjugate Gradient Method [5].

4.3.2 Proximal Policy Optimization (PPO)

The question to which the invention of PPO answered was: "Is it possible to develop a first order version of TRPO, in order to scale better with larger NN's and use existing Deep Learning frameworks?". The simple modification which has been done to TRPO was to force a clipping effect on the Loss function and has been developed in [6] by Schulman et al.

The un-clipped Loss function is defined as:

$$J(\theta) = L(\theta) - \beta E_{\pi_\theta^{old}} \left[KL(\pi_\theta^{old}(a_t|s_t), \pi_\theta(a_t|s_t)) \right] \quad (4.34)$$

By performing some SGD on such Loss function, the parameters θ could be optimized too much, hence obtaining a very different policy. In order to force some bounds on this possible update a clipping factor is introduced:

$$r(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_\theta^{old}(a_t|s_t)} \hat{A}_t \quad (4.35)$$

$$L^{CLIP}(\theta) = E \left[\min_\theta \left(r(\theta), \text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right] \quad (4.36)$$

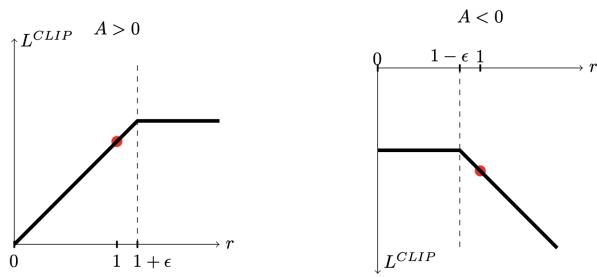


Figure 4.4. Example of the Clipped surrogate loss function [6]

The result of such formulation of the problem is that when some Advantage is very big in module, then if the Surrogate Loss exceeds its linear zone, then the gradient of the Loss w.r.t. parameters will be zero, hence that component of the loss will not contribute to change the policy, otherwise it could cause some damages.

Chapter 5

Proposed Environment Structures

5.1 Structure 1

As previously stated, a MDP is a tuple with structure $\langle S, A, P, R, \rangle$. In this case the system is deterministic since we do not suppose uncertainty in the state transition function.

5.1.1 State

The state of the system is a vector composed in this way:

$$S = (x, y, \theta, x_1, y_1, x_2, y_2, \dots, x_n, y_n)^T \quad (5.1)$$

Where x, y, θ are the three degrees of freedom that define the unicycle pose, x_i, y_i are the coordinates of the via points near which the robot must pass. In this first proposed structure for the environment the initial number of via points must be specified before training, since the dimension of the state cannot change once the network has been trained.

The original map is defined as a grid made up of squares with sides 1 by 1 meters, each of which could represent an **admissible square** for the robot to be or an **obstacle square**.

In addition to that, in order to let the policy be general with respect to various litter distributions, the squares which contain a reward signal are distributed with a uniform probability distribution.

The map is represented as a matrix, defined as follows:

$$\begin{cases} A_{ij} = 1 & \text{if position } i, j \text{ corresponds to nonzero reward} \\ A_{ij} = -2 & \text{if position } i, j \text{ corresponds to an obstacle} \\ A_{ij} = 0 & \text{else} \end{cases}$$

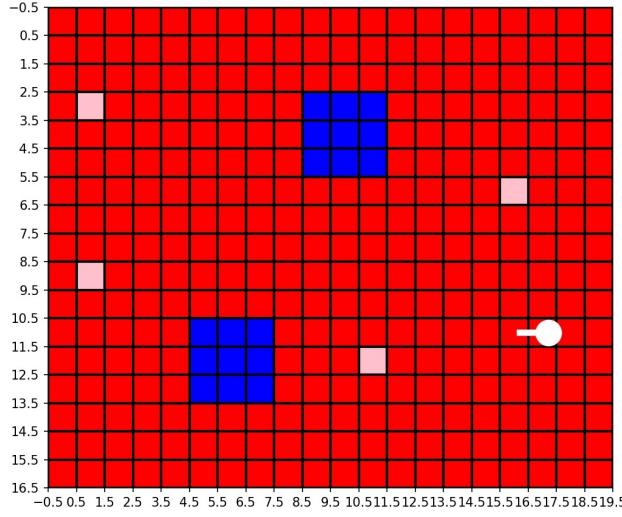


Figure 5.1. Example of a map
Red=empty square, Pink=Via point, Blue=Obstacle

5.1.2 State normalization

An important aspect in DRL is to normalize the state components or in general to not let them differ too much numerically. This could prevent numerical problems when computing the gradient and updating the weights.

In this case x and y components are normalized from $[0, size_x], [0, size_y]$ to $[0, 1]$ and the angle θ is normalized to $[0, 1]$ but in the following way, keeping into account the fact that this is a non euclidean space:

$$\begin{cases} \theta_{norm} = (\theta - 2\pi)/2\pi & \text{if } \theta > 2\pi \\ \theta_{norm} = (\theta + 2\pi)/2\pi & \text{if } \theta < 0 \\ \theta_{norm} = \theta/2\pi & \text{else} \end{cases} \quad (5.2)$$

5.1.3 Action

Since the model is a unicycle, in principle $a \in \mathcal{R}^2$. In this case, for simplicity reasons the linear velocity has been fixed while the angular velocity can either be $\{-\omega_{max}, 0, \omega_{max}\}$, therefore the robot moves according to the Dubin's Model [24].

5.1.4 State transition function

This transition function is a deterministic one, meaning that given the initial state and the chosen action, the next state can be univocally determined. Such transition function can be explained separately for the first 3 components of the state and for the remaining ones.

For the dynamic model of the unicycle the transition is given by the exact integration

method:

$$\begin{cases} x_{k+1} = x_k + \frac{v_k}{\omega_k} (\sin(\theta_{k+1}) - \sin(\theta_k)) \\ y_{k+1} = y_k - \frac{v_k}{\omega_k} (\cos(\theta_{k+1}) - \cos(\theta_k)) \\ \theta_{k+1} = \theta_k + \omega_k T \end{cases}$$

Actually, an admissibility check is performed on such new position. The robot position x_{t+1}, y_{t+1} is approximated to the nearest integer values $x_{t+1}^{int}, y_{t+1}^{int}$. If $A(x_{t+1}^{int}, y_{t+1}^{int}) = -2$ then it means that the robot has collided with an obstacle and therefore the new position is equal to the previous one at time t .

The other components of the state (i.e. the coordinates of the four via points) can be updated as follows: if the robot has collected the litter corresponding to the via point j then we replace the coordinates x_j, y_j with $(-1, -1)$.

Before taking into account the state transition due to the litter collection an important aspect related to the reward function must be clarified. Since the main routine is intended to represent an approximate suggestion for the robot, it is not required that the robot travels exactly on the square which stores a +1 in matrix A , but it is sufficient that the robot travels near to it. The chosen proximity region is chosen to be the 3 by 3 square centered around the actual robot position and, inside this region, every square which contains a +1 signal (i.e. a litter piece) is considered to be "collected".

The precedent figure has been colored by setting square color as blue if the corresponding value is less than zero, otherwise if the value is zero is set to red and finally if greater than zero is set to pink. In this figure the Blue squares which are **not** below the WMR are considered to be obstacles, and have value of -2, while the square on which the robot is located has value of -1.

5.1.5 Reward

The reward function is quite sparse since it is different from zero only when the robot collects some litter. This induces some problems related to exploration of the map as will be presented in the simulations section.

$$\begin{cases} r=0.5(1+a_{map}^2) & \text{if } A_{i,j} = 1 \\ 0 & \text{if } A_{i,j} = 0 \end{cases} \quad (5.3)$$

Where a_{map} corresponds to the percentage of goal points near which the robot has passed until the current timestep. This choice for r is meant to push the robot to look for rewards which correspond to a higher a_{map} .

One could think to use a negative reward when hitting obstacles, or even to terminate the game when such event occurs. As will be shown in the simulation section, this choice yields very poor learning since the robot learns to move in circles, afraid of hitting walls or obstacles. This problem is commonly referred to being stuck in a local minima.

Thanks to such results the choice of not using any negative reward at all, besides the one corresponding to the having reached the maximum number of moves, taken equal to -1.

5.2 Structure 2

5.2.1 State

The state of the system is a vector composed in this way:

$$S = (x, y, \theta, a_{map}, \text{Map}^T)^T$$

Where x, y, θ are the three degrees of freedom that define the unicycle pose, a_{map} is the ratio between the number of points that have been visited from the start of the episode and the total number of initial points. The last components of S i.e. **Map**, is a vectorized version of the original map, in order to use MLP's.

The original map is composed of the whole set of **admissible squares** and **obstacle squares** but, since the robot cannot occupy regions which coincide with obstacles, the only meaningful portions of the map to be placed inside the state of the DRL problem are the admissible ones.

Algorithm 2 Vectorize_map

```

1: Receives as input matrix  $A$  and initializes vector Map with an empty array
2: for  $row = 1, 2, \dots, M$  do
3:   for  $column = 1, 2, \dots, T$  do
4:     if  $A(row, column) > -2$  then
5:       Map.append(A(row, column))
6:     end if
7:   end for
8: end for

```

The algorithm considers the quantity -2 as a threshold since the map A is defined as having:

$$\begin{cases} A_{ij} = 1 & \text{if position } i, j \text{ corresponds to nonzero reward} \\ A_{ij} = -2 & \text{if position } i, j \text{ corresponds to an obstacle} \\ A_{ij} = -1 & \text{if position } i, j \text{ corresponds to the robot approximate location} \\ A_{ij} = 0 & \text{else} \end{cases}$$

In order to let the policy be general with respect to various litter distributions, the squares which contain a reward signal are distributed with a uniform probability distribution.

5.2.2 State normalization

An important aspect in DRL is to normalize the state components or in general to not let them differ too much numerically. This could prevent numerical problems

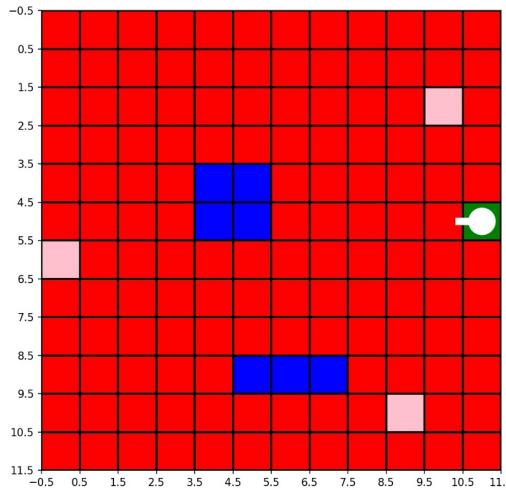


Figure 5.2. Example of a map Red=empty square, Pink=Via point, Blue=Obstacle,Green=Square occupied by the robot

when computing the gradient and updating the weights.

In this case both x and y are normalized to $[0, 1]$, the angle θ too is normalized in the same way as Structure 1:

$$\begin{cases} \theta_{norm} = (\theta - 2\pi)/2\pi & \text{if } \theta > 2\pi \\ \theta_{norm} = (\theta + 2\pi)/2\pi & \text{if } \theta < 0 \\ \theta_{norm} = \theta/2\pi & \text{else} \end{cases}$$

Then the Map portion of the state is defined as either 0 if the square is empty, 1 if the square contains litter, -1 if the square contains the robot.

5.2.3 Action

Since the model is a unicycle in principle $a \in \mathcal{R}^2$. In this case, for simplicity reasons the linear velocity has been fixed while the angular velocity can either be $\{-\omega_{max}, 0, \omega_{max}\}$, according to the Dubin's Model.

5.2.4 State transition function

This transition function is a deterministic one, meaning that given the initial state and the chosen action, the next state can be univocally determined.

For the dynamic model of the unicycle the transition is simply given by the exact integration method:

$$\begin{cases} x_{k+1} = x_k + \frac{v_k}{\omega_k}(\sin(\theta_{k+1}) - \sin(\theta_k)) \\ y_{k+1} = y_k - \frac{v_k}{\omega_k}(\cos(\theta_{k+1}) - \cos(\theta_k)) \\ \theta_{k+1} = \theta_k + \omega_k T \end{cases}$$

Actually, an admissibility check is performed on such new position. The robot position x_{t+1}, y_{t+1} is approximated to the nearest integer values $x_{t+1}^{int}, y_{t+1}^{int}$. If

$A(x_{t+1}^{int}, y_{t+1}^{int}) = -2$ then it means that that action was not allowed and therefore the cartesian position of the robot is not updated. In order to let the robot escape this kind of situation, the robot can change its orientation even when hitting obstacles or walls.

Before taking into account the state transition due to the litter collection an important aspect related to the reward function must be clarified. Since the main routine is intended to represent an approximate suggestion for the robot, it is not required that the robot travels exactly on the square which stores a +1 in matrix A , but it is sufficient that the robot travels near to it. The chosen proximity region is chosen to be the 3 by 3 square centered around the actual robot position and, inside this region, every square which contains a +1 signal (i.e. a litter piece) is considered to be "collected".

The precedent figure has been colored by setting square color as blue if the corresponding value is less than zero, otherwise if the value is zero is set to red and finally if greater than zero is set to pink. In this figure the Blue squares are considered to be obstacles.

5.2.5 Reward

The reward function is quite sparse since it is different from zero only when the robot collects some litter. This induces some problems related to exploration of the map as will be presented in the simulations section.

$$\begin{cases} r=0.5(1+a_{map}^2) & \text{if } A_{i,j} = 1 \\ 0 & \text{if } A_{i,j} = 0 \end{cases}$$

This choice for r is meant to push the robot to look for rewards which correspond to a higher a_{map} .

One could think to use a negative reward when hitting obstacles, or even to terminate the game when such event occurs. As will be shown in the simulation section, this choice yields very poor learning since the robot learns to move in circles, afraid of hitting walls or obstacles. This problem is commonly referred to being stuck in a local minima.

Thanks to such results the choice of not using any negative reward at all, besides the one corresponding to the having reached the maximum number of moves, taken equal to -1.

5.2.6 Voronoi diagram generalization

Even though the best possible approach for this class of problems should be that of using CNN's, one could still use this case for non-square maps. The only difference would be to scatter some points along the boundaries/non-square regions of the map and define a voronoi diagram around them. Then we can no longer model the map as a matrix but we need to define a dictionary, where each key, the scattered points coordinates, is associated to a value of -2: obstacle, 0:empty, +1:litter.

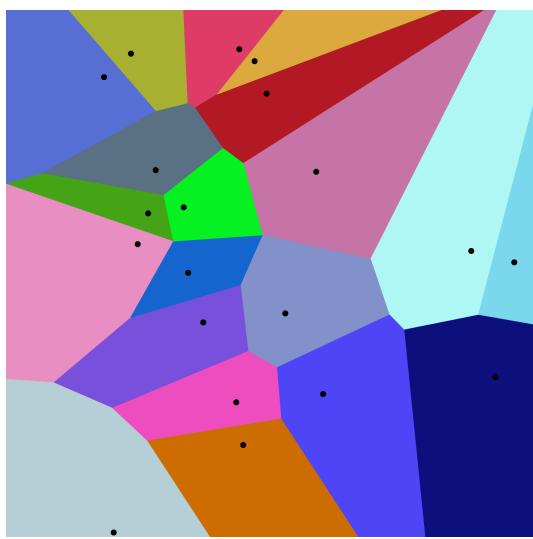


Figure 5.3. Example of Voronoi Tesselation

Chapter 6

Subroutine Control Law

The subroutine is switched on once the Computer vision system detects the presence of some litter pieces around itself.

The control law can be expressed as a trajectory tracking task exposed to the presence of artificial potential fields. The trajectory which needs to be tracked in this case is generated by interpolating couples of points, whose order is taken similarly to the Dubins Traveling Salesman Problem (DTSP) developed by [24].

Define $P = \{p_1, p_2, \dots, p_n\}$ a set of 2D points p_i which need to be connected with the shortest path.

Algorithm 3 DTSP Algorithm

```

1: Find  $\{a_1, a_2, \dots, a_n\}$ , the solution to the ETSP problem
2: Set  $\Phi_1$  = orientation of segment from  $a_1$  to  $a_2$ 
3: for  $i=2, \dots, n-1$  do
4:   if  $i$  is Even then
5:     Set  $\Phi_i = \Phi_{i-1}$ 
6:   else
7:     Set  $\Phi_i$  = orientation of segment from  $a_i$  to  $a_{i+1}$ 
8:   end if
9: end for
10: if  $n$  is Even then
11:   Set  $\Phi_n = \Phi_{n-1}$ 
12: else
13:   Set  $\Phi_n$  = orientation of segment from  $a_n$  to  $a_1$ 
14: end if
15: Return the set composed by  $\{(a_i, \Phi_i)_{i=1, \dots, n}\}$ 
```

Then given for example two points (a_1, Φ_1) and (a_2, Φ_2) a Cubic Spline can be used in order to connect such two configurations:

$$x(s) = -(s-1)^3 x_i + s^3 x_f + \alpha_x s^2 (s-1) + \beta_x s (s-1)^2 \quad (6.1)$$

$$y(s) = -(s-1)^3 y_i + s^3 y_f + \alpha_y s^2 (s-1) + \beta_y s (s-1)^2 \quad (6.2)$$

Then boundary conditions to the problem are:

$$\begin{aligned}x(0) &= x_i, \quad y(0) = y_i, \quad x(1) = x_f, \quad y(1) = y_f \\ \dot{x}(0) &= K\cos(\Phi^i), \quad \dot{y}(0) = K\sin(\Phi^i), \quad \dot{x}(1) = K\cos(\Phi^f), \quad \dot{y}(1) = K\sin(\Phi^f)\end{aligned}$$

This is achieved by choosing:

$$\begin{pmatrix} \alpha_x \\ \alpha_y \end{pmatrix} = \begin{pmatrix} K\cos(\Phi^f) - 3x_f \\ K\sin(\Phi^f) - 3y_f \end{pmatrix}, \quad \begin{pmatrix} \beta_x \\ \beta_y \end{pmatrix} = \begin{pmatrix} K\cos(\Phi^i) - 3x_i \\ K\sin(\Phi^i) - 3y_i \end{pmatrix} \quad (6.3)$$

Note that K has been used both at the beginning of motion and at the end of motion, in order to perform the overall trajectory in forward motion. This is required since the WMR developed by Pixies has its litter collection site located on the front.

An example is presented in the following part. The points where litter has been detected in the surroundings are:

$$P_1 = (0.0, 0.0), \quad P_2 = (0.38, 2.95), \quad P_3 = (-3.95, 0.14), \quad P_4 = (-2.33, 0.13)$$

The ETSP algorithm on such small set of points yields a visiting order of $\{0, 1, 3, 2\}$. Then such points are interpolated using $K = 5$:

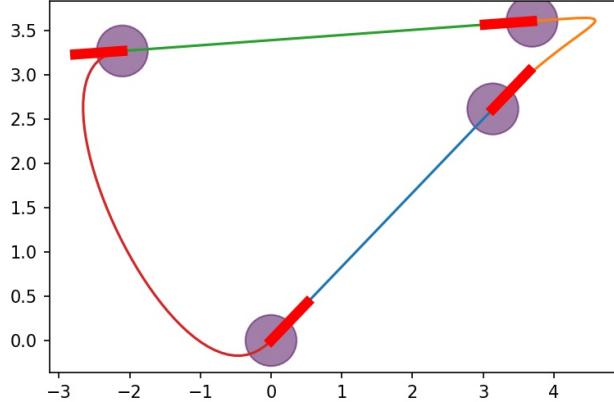


Figure 6.1. Interpolation

Since there are physical limitations on the torque generated by each motor (which translate in maximum allowed linear and angular velocities) some feasibility check must be performed, and in case of unfeasible trajectories some time scaling must be applied. This check can be done in the following way. Start by defining the timing law as $s(t) = t/T$. Then proceed to compute the values for the linear velocity and

angular velocity along the proposed trajectory:

$$\dot{x} = \frac{1}{T} \left[-3\left(\frac{t}{T} - 1\right)^2 x_i + 3\left(\frac{t}{T}\right)^2 x_f + \alpha_x \left(3\left(\frac{t}{T}\right)^2 - 2\frac{t}{T}\right) + \beta_x \left(3\left(\frac{t}{T}\right)^2 - 4\frac{t}{T} + 1\right) \right] \quad (6.4)$$

$$\dot{y} = \frac{1}{T} \left[-3\left(\frac{t}{T} - 1\right)^2 y_i + 3\left(\frac{t}{T}\right)^2 y_f + \alpha_y \left(3\left(\frac{t}{T}\right)^2 - 2\frac{t}{T}\right) + \beta_y \left(3\left(\frac{t}{T}\right)^2 - 4\frac{t}{T} + 1\right) \right] \quad (6.5)$$

$$\omega(t) = \frac{\ddot{y}\dot{x} - \ddot{x}\dot{y}}{\dot{x}^2 + \dot{y}^2} \quad (6.6)$$

Then if $v_{max} > v_{bound}$ or $|\omega|_{max} > \omega_{bound}$ a time scaling can be applied in order to render the trajectory feasible: if e.g. $v_{max}/v_{bound} = \gamma$ then $T \leftarrow T\gamma$.

Chapter 7

Simulation Results

7.1 Main Routine: Structure 1

This section presents some results about the first proposed structure for the environment. The state is then defined as:

$$S = (x, y, \theta, x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4)^T$$

As it can be immediately seen, this method is intrinsically limited since the number of goal states must be specified a-priori and cannot be modified. This limitation will motivate the proposition of structure 2, which processes the map as an image and hence can contain a variable number of goal states.

The advantage of this method is that, when the task is known to have a fixed number of goal states, the problem can be solved with a state which is significantly lower in dimension w.r.t. processing the whole picture. Define N as the number of goal states, then the left side of the following inequality represents the state dimension for Structure 1 and the right side the dimension for Structure 2: $3 + 2 * N << 3 + 1 + \text{size_x} \times \text{size_y}$, where 3 comes from the dimension of the state space for the unicycle. The additional +1 on the right side term of the inequality derives from the fact that, for the second structure, also the covered area percentage is given as input to the DRL algorithm.

map size x-axis	12
map size y-axis	12
linear speed	0.5 m/s
angular speed max	0.3 rad/s
δt	$2\pi/8\omega_{max}$
Maximum number of moves	500
Penalty when hitting obstacle	0

Table 7.1. Environment parameters - Structure 1

Remark: An important aspect must be noted: the time discretization. The angular velocity and the linear velocity must be coherent with each other. The

Network architecture	(200,200,200) MLP's
Learning rate schedule	Multistep
Gamma	0.99
Gae Lambda	0.95
Number of steps collected for each optimization	2000
Batch size	400
Epochs for each optimization	4
Clip range	0.2

Table 7.2. PPO parameters - Structure 1

ratio between the linear and angular velocities represents the radius of the circular trajectories described by the robot. This ratio cannot be too large because in that case the robot would not rotate, but on the contrary a very small ratio could make the robot steer too much. A choice of 1.66 meters for the radius has been done as a trade-off. In addition, a relation between δt and ω has been proposed: $\delta t = \frac{2\pi}{8\omega}$. Indeed, suppose that the robot performs always the action of going left, since the angular position will always vary of $\omega \times \delta t$ then if 8 movements are performed in this case, the robot performs a complete rotation of π radians. This is convenient since in this case we have also discretized the possible values that the angular position can assume, and are $[0, \pi/4, \pi/2, 3/4\pi, \pi, 5/4\pi, 3/2\pi, 7/4\pi]$.

7.1.1 PPO-DQN-A2C Comparison

Network architecture	(200,200,200) MLP's
Learning rate schedule	Multistep
Batch size	32
Buffer size	10^6
train freq	4
Batch size	400
exploration fraction	0.1
max grad norm	2
exploration final eps	0.05

Table 7.3. DQN parameters - Structure 1

Network architecture	(200,200,200) MLP's
Learning rate schedule	Multistep
rms_prop_eps	10^5
gamma	0.99
N_steps	400
max grad norm	0.5

Table 7.4. A2C parameters - Structure 1

This set of parameters has been slightly from the ones proposed in the Library Stable Baselines 3, developed by Raffin et al. [13].

In the following figures a comparison between DRL algorithms is shown. Not only PPO proves to be effective w.r.t. sample efficiency, but also w.r.t. computation time, which is not shown in the figures. In order to train for 2 million steps, using local CPU, DQN took 3 hours while PPO took only 45 minutes. By using current A2C parameters, A2C has not shown satisfactory behaviour as can be immediately seen by the following figures. Note that the variable "steps" on the x axis refers to single state transitions of the environment.

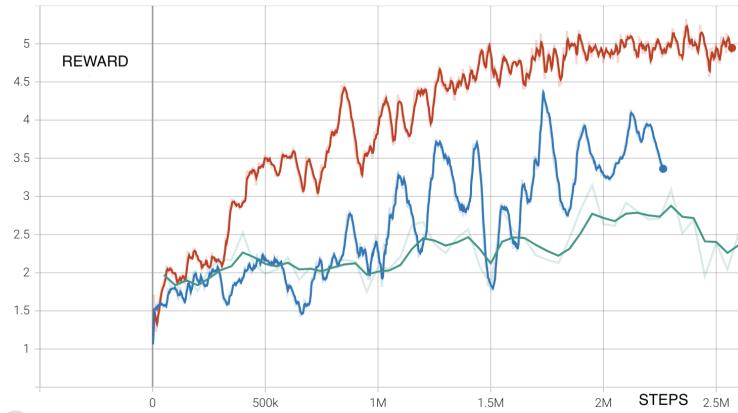


Figure 7.1. Episode Reward for DQN,PPO,A2C Comparison. Red: PPO, Blue: DQN, Green: A2C

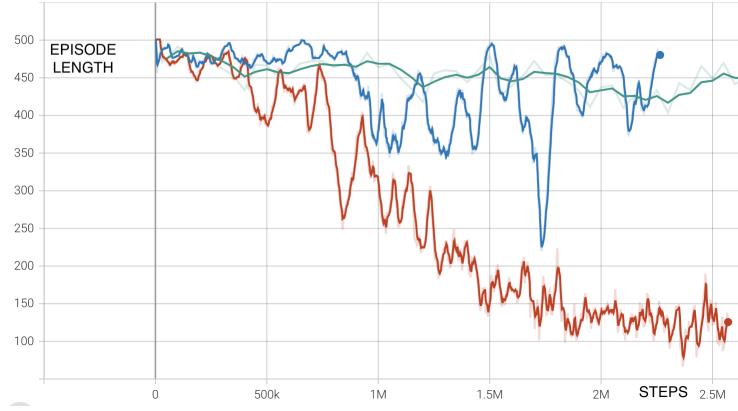


Figure 7.2. Episode length for DQN,PPO,A2C Comparison. Red: PPO, Blue: DQN, Green: A2C

7.1.2 Negative Reward Comparison

This section aims at investigating the effect of negative rewards when hitting obstacles or walls. The values are : $P \in \{-0.01, -0.1, -1\}$. This negative rewards affect the optimization process by introducing numerical problems.

Consider the following example, denoted with A: during an episode of 500 moves the robot hits the wall for 10 times and reaches one goal state, then with $P=-1$ then the episodic reward would be about $-10+1-0.5=-9.5$. In order to get a glimpse of what the algorithm perceives from a numerical point of view, consider that a perfect game has a score of about $N-0.5$, where N is the number of goal states (recall that the -0.5 comes from the penalty given at the end of the game). Then for example, if $N=5$, a perfect score would be 4.5, which is much less than the absolute value of the score of scenario A. If the same robot hits the wall for 100 times, scenario B then the episodic reward becomes -99.5 ! This fact does create huge problems.

The robot as a consequence develops a policy which aims at not hitting the wall, hence remaining stuck in a local minimum. Since the policy has been corrupted, the new distribution of state trajectories will be very poorly informative, hence deteriorating the learning process in a non-reversible way. With $P=-0.1$ the episodic reward for scenario A becomes -0.5 , but still in scenario B we get -9.5 . With $P=-0.01$, the corrupting effect becomes almost negligible.

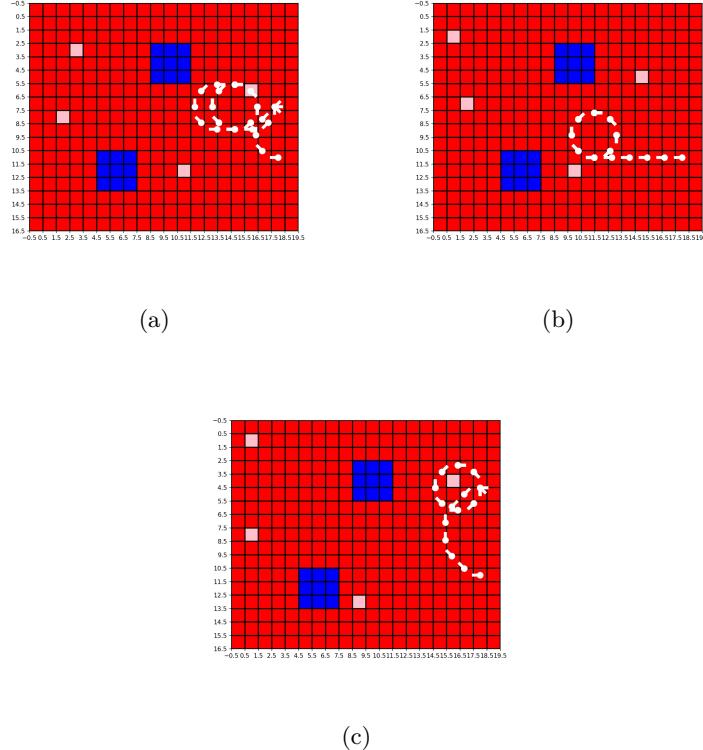


Figure 7.3. Trajectories obtained from a deteriorated policy.

The following figures show that the learning process deteriorates with negative

rewards. In particular, both Blue and Cyan curves, which correspond to $P=-1$, $P=-0.1$ get corrupted irreversibly after 200'000 steps. This result can be seen from the figure regarding average episodic length.

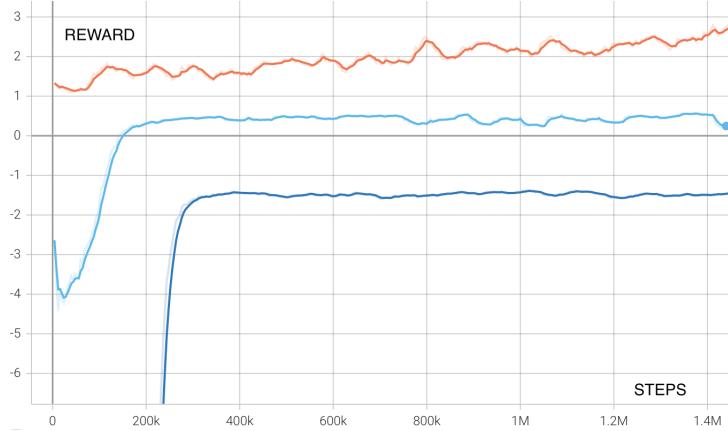


Figure 7.4. Learning deteriorating with negative rewards. Blue: $P=-1$, Cyan: $P=-0.1$, Orange: $P=-0.01$. (Reward)

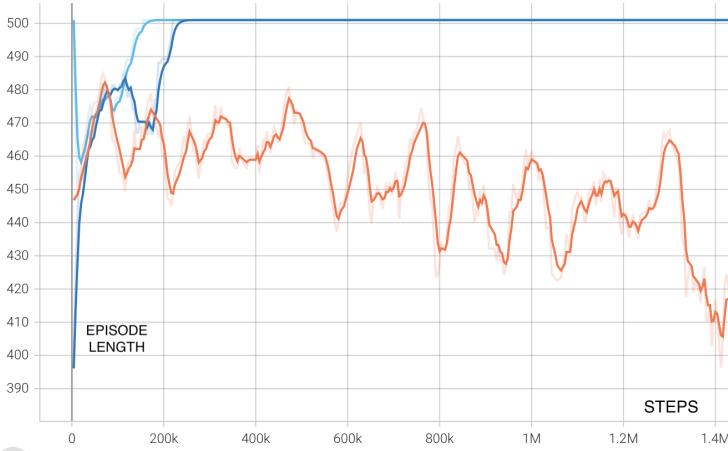


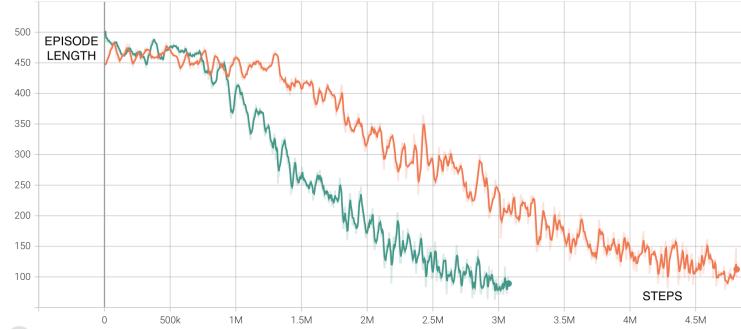
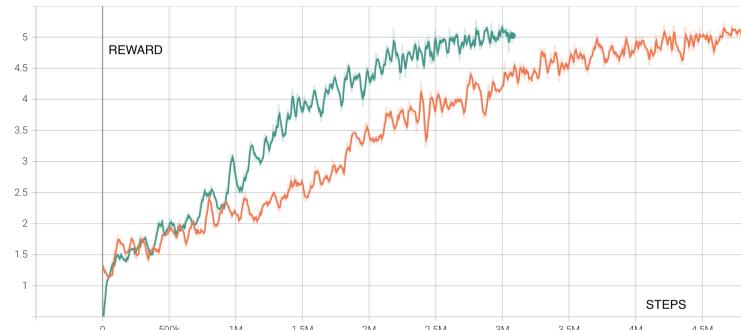
Figure 7.5. Learning deteriorating with negative rewards. Blue: $P=-1$, Cyan: $P=-0.1$, Orange: $P=-0.01$. (Episode length)

This results confirm that the choice of adopting non-penalizing environments is effective in the case of sparse rewards.

7.1.3 Learning rate schedule comparison

In this section the effect of the learning rate scheduler is investigated. The following simulation shows two different multi-step learning rate schedulers:

Steps	0	1M	2M	2.5M	3M	4M
Schedule 1	3×10^{-4}	2×10^{-4}	2×10^{-4}	1.5×10^{-4}	10^{-4}	0.9×10^{-4}
Schedule 2	1.5×10^{-4}	1.3×10^{-4}	1.1×10^{-4}	$\times 10^{-4}$	0.9×10^{-4}	0.6×10^{-4}

Table 7.5. Learning Rate Schedulers**Figure 7.6.** Learning rate comparison. Episode Length**Figure 7.7.** Learning rate comparison. Episode Length

7.2 Main Routine: Structure 2

As explained in section 5.2, the state of the system defined for this second structure is a vector composed in this way:

$$S = (x, y, \theta, a_{map}, \text{Map}^T)^T$$

Where x, y, θ are the three degrees of freedom that define the unicycle pose, a_{map} is the ratio between the number of goal points that have been visited and the total number of initial goal points. The last components of S i.e. **Map**, is a vectorized version of the original map, in order to use MLP's. The original map is defined as a grid made up of squares with sides 1 by 1 meters, each of which could represent an **admissible square** for the robot to be or an **obstacle square**. Starting from the upper left of the map and going row wise, each **admissible square** is stored inside the **Map** vector.

The main advantage possessed by this method w.r.t. the one explained in section 5.1 (Structure 1) is that in this case any number of goal states can be introduced, since the whole map is taken as input from the policy neural network.

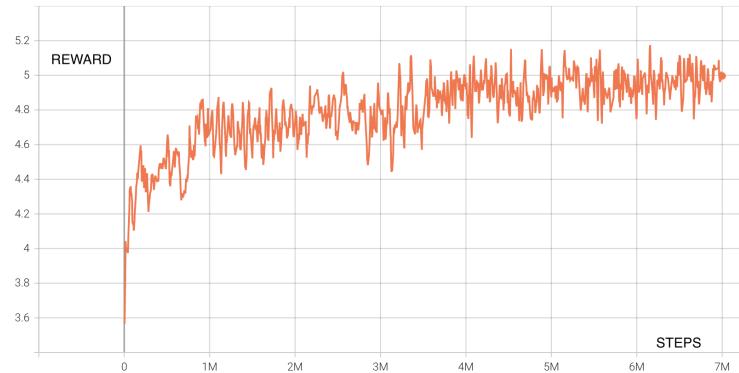
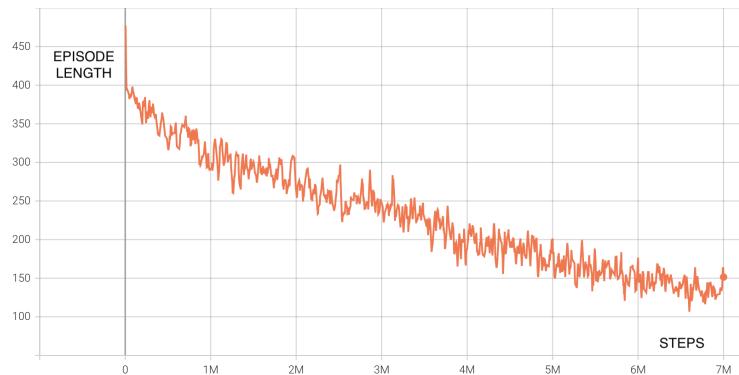
7.2.1 Variable number of Goal States (17×20 map)

This chapter focuses on investigating if this approach is able to handle variable number of goal states on the map. The following two tables contain the most important parameters used for such simulations, the first one contains parameters regarding the environment structure and how a step is performed and the second table shows parameters regarding the PPO algorithm. The number of parameters for an architecture of (500,300,300), equal for policy and Value networks is 403503. In addition, training time is 3.5 hours.

map size x-axis	17
map size y-axis	20
linear speed	0.5 m/s
angular speed max	0.3 rad/s
δt	$2\pi/8\omega_{max}$
Maximum number of moves	500
Penalty when hitting obstacle	0
Number of Goal States	4 ÷ 5

Table 7.6. Environment parameters - Structure 2

Network architecture	(500,300,300) MLP's
Learning rate schedule	Linear interpolation
Initial learning rate	$1.3e^{-4}$
Final learning rate	$1e^{-5}$
Gamma	0.99
Gae Lambda	0.95
Number of steps collected for each optimization	4000
Batch size	1000
Epochs for each optimization	4
Clip range	0.2

Table 7.7. PPO parameters - Structure 2**Figure 7.8.** Episode Reward on map of size 17×20**Figure 7.9.** Episode length on map of size 17×20

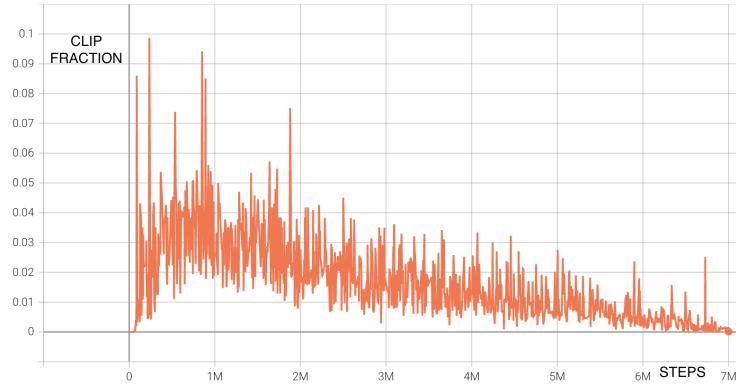


Figure 7.10. Clip fraction on map of size 17×20

Further training with learning rate lower than 8×10^{-6} is able to refine the policy.

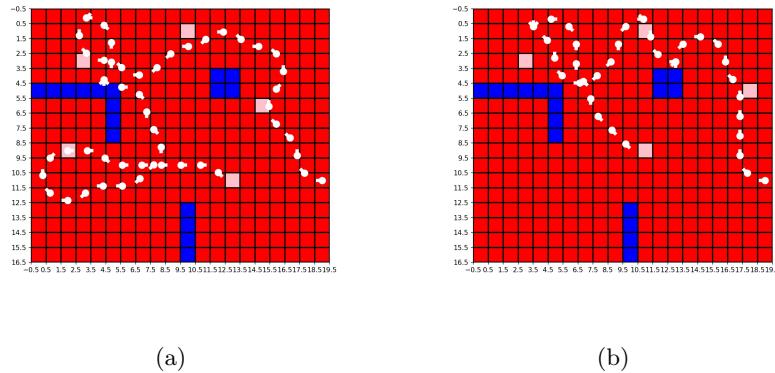


Figure 7.11. Two successful trajectories on map of size 17×20

After training with even smaller learning rates the agent found a flaw in the environment. The upper left obstacle in blue is not wide enough and the robot has found a way to overcome it.

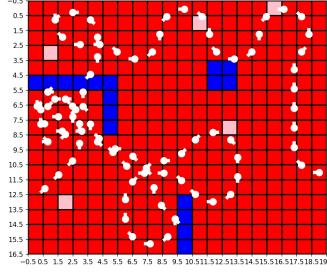


Figure 7.12. The agent found a flaw in the environment and exploited it to win the game

This flaw can be easily solved by hard coding that he cannot move in that direction, independently from the size of the step he takes.

Remark: An aspect which is intuitive but must be noticed is that the bigger the set of squares where Goal States can be distributed, the slower the training.

7.2.2 Neural Network Architecture Comparison (12×12 map)

In this section, the influence of the size of the Neural Network is investigated. The environment dimension used for this comparison is smaller w.r.t. the 17×20 map used for other simulations, a 12×12 map is used instead. This choice has been done just to speed up computations and does not represent a loss of generality issue. Three MLP's are proposed, which differ greatly in the number of parameters. The formula which is being used in order to compute the total parameters is the following:

$$\begin{aligned} i \times w_1 + w_1 + w_1 \times w_2 + w_2 + w_2 \times w_3 + w_3 + w_3 \times o + o &= \\ = i \times N + (N) + N \times N + (N) + N \times N + (N) + N \times 3 + (3) &= \\ = 2N^2 + (6 + i)N + 3 & \end{aligned}$$

Where $i = 11$ represents the size of the state of the system if considering four goal states. In addition $o = 3$ represents the number of possible actions that the network can take.

N	Parameters
50	5853
200	83403
300	185103

Table 7.8. Number of parameters of each network

The following figures consist of six KPI's regarding the learning process: Episode reward, Episode Length, Clip Fraction, Value Loss, Policy Loss, KL divergence. As it is evident, despite a great difference in number of network parameters the overall behaviour is similar. Even though the clip factor (0.2) is the same among

all simulations, it appears that for wider networks the learning is faster during the first phase, but the overall convergence to optimality is reached simultaneously. It appears that the less wide network is able to catch up with the others.

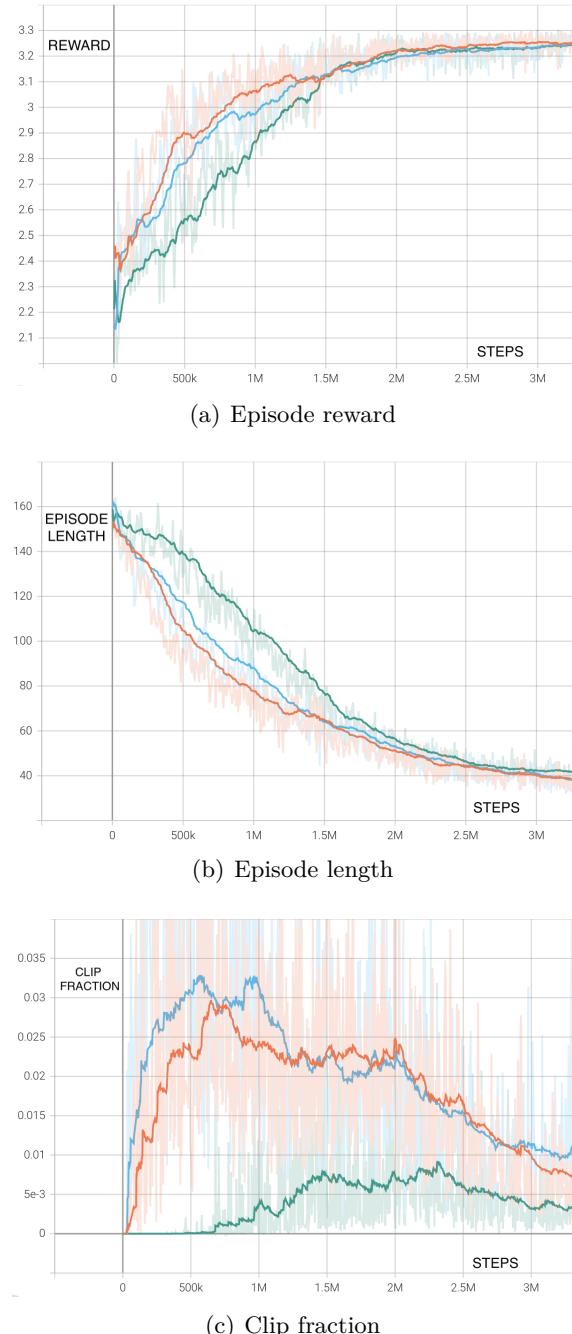
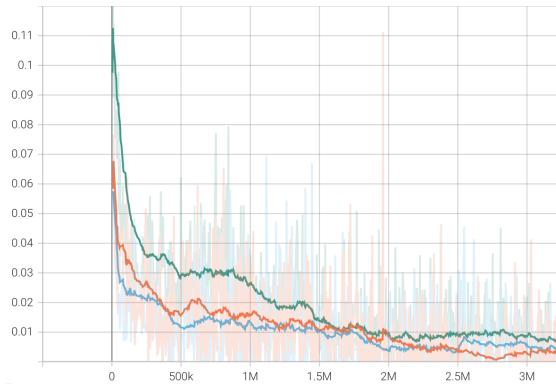
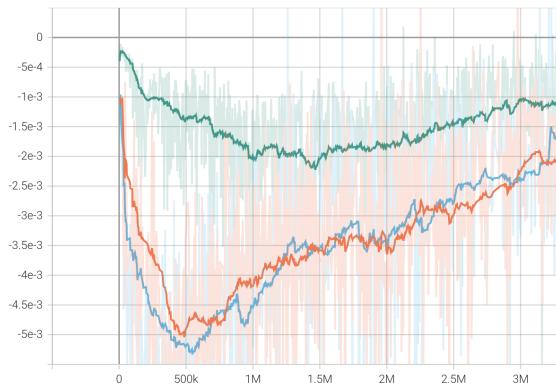


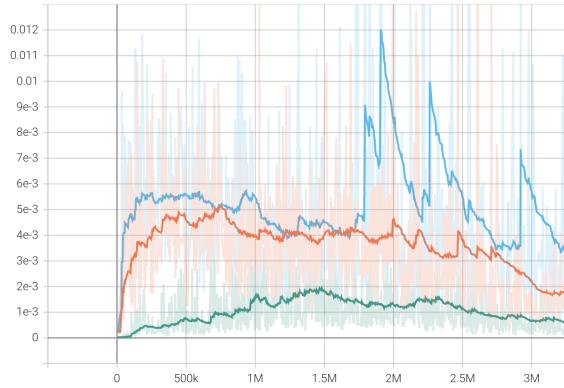
Figure 7.13. Neural Network Architecture Comparison, part 1 | Green: (50,50,50), Orange:(200,200,200),Cyan:(300,300,300)



(a) Value Loss



(b) Policy Loss



(c) Kullback-Liebler Approximate Divergence

Figure 7.14. Neural Network Architecture Comparison, part 2 | Green: (50,50,50), Orange:(200,200,200),Cyan:(300,300,300)

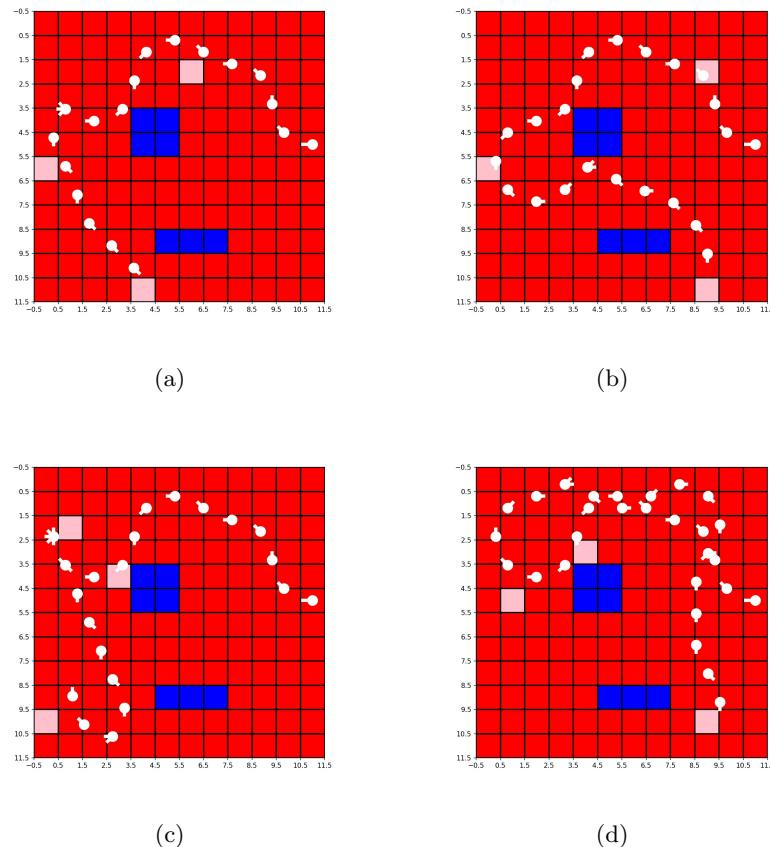


Figure 7.15. Some succesful trajectories, NN=(300,300,300)

7.2.3 ROS simulation

In addition, the tracking of the trajectory given by structure 2 on the small 12×12 map has been performed using the Simulation Software ROS (Robotic Operating System). Video available on <https://github.com/ArturoMaiani/Master-Thesis---Pixies/blob/main/README.md>.

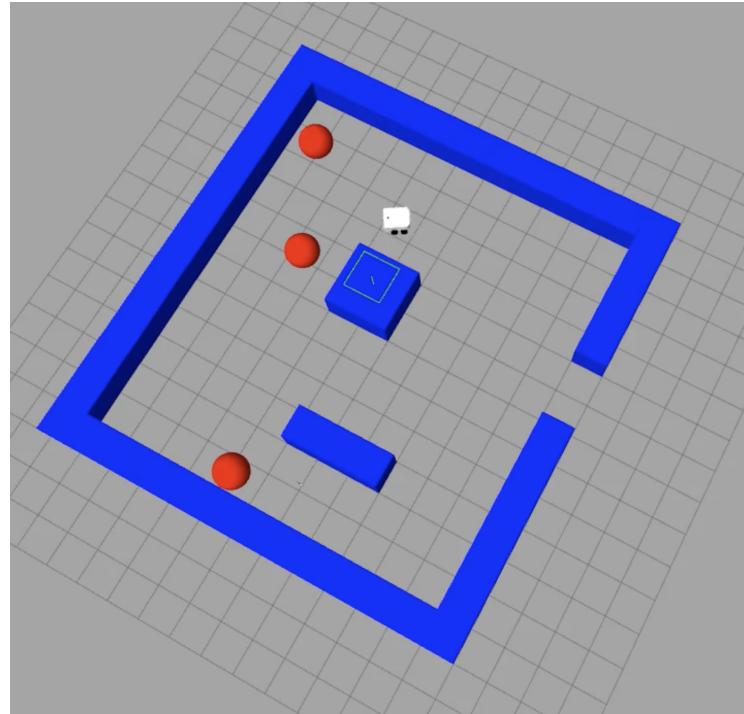


Figure 7.16. ROS environment

7.3 Simulations about Subroutine Control law

This section presents some results about the proposed subroutine. As stated in chapter 6, this function takes as input a list of points which must be visited. This list is provided by the Computer vision algorithm, which uses the bounding box technique to detect and locate litter pieces. The second step of this subroutine is that of defining a proper order for the points to be visited. This ordering is done through a standard Euclidean Traveling Salesman Problem, solving it in a brute force way since the number of points is very low (e.g. 4,5,..10).

The following step is that of interpolating each point in the sequence with cubic trajectories. Together with the path, a timing law is defined, for instance using a default $T=20$ as time duration for the single cubic. Then a Uniform Time Scaling check is done in order to let the motion be feasible, accordingly with the bounds of $\pm\omega_{max}$. The final step is that of using such reference trajectories in order to define a tracking problem. This tracking is done using the IO exact Linearization problem. As it can be seen from the third picture, an initial nonzero error is rapidly recovered, thanks to the proportional controller with $k_x, k_y = 0.1$ set as proportional coefficients. In addition, the initial and final linear velocity is set equal to 0.05 m/s. As it can be seen from this plots, with the following choice of random points to be interpolated and the choice of $\omega_{max} = 0.5, v_{max} = 0.5$ and bounds, yields no violation. One can notice that the red curve, corresponding to path 4 is critically near to the out-of-bound region for the angular velocity. The Pose of the robots which can be seen in the third figure is defined from the reference trajectory and refers to the standard unicycle model, not the one which considers as state the displaced point on the front of the robot. This can explain why for instance the yellow plot (path 2) doesn't show a peak in the angular velocity for the final part of the path. The modified robot doesn't need to perform some posture regulation but just to bring the displaced point along the reference.

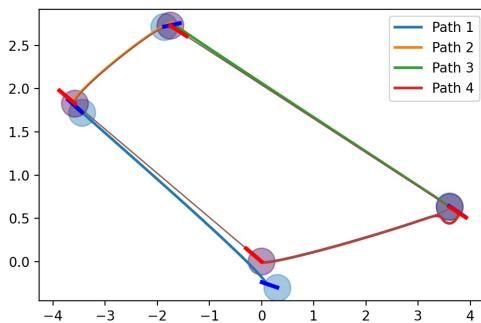


Figure 7.17. Subroutine test 1:Tracking

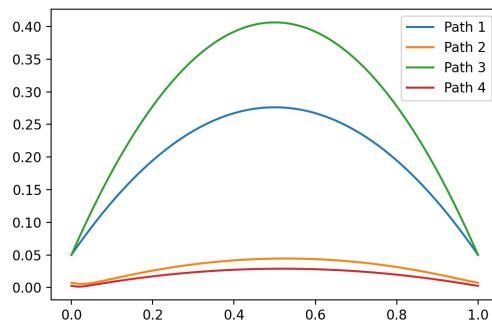


Figure 7.18. Subroutine test 1:linear velocity

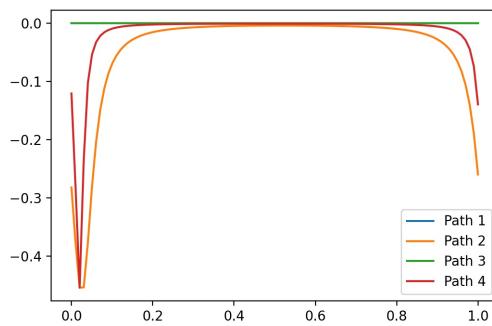


Figure 7.19. Subroutine test 1:angular velocity

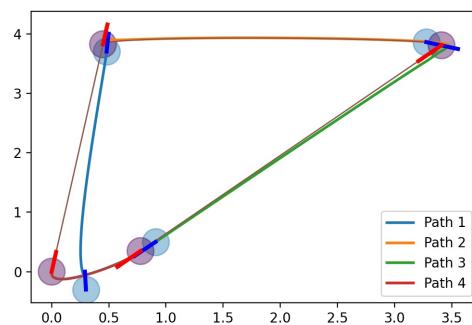


Figure 7.20. Subroutine test 2:Tracking

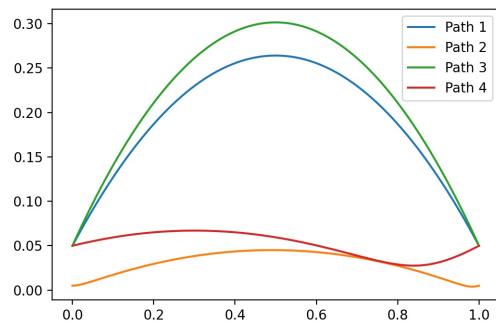


Figure 7.21. Subroutine test 2:linear velocity

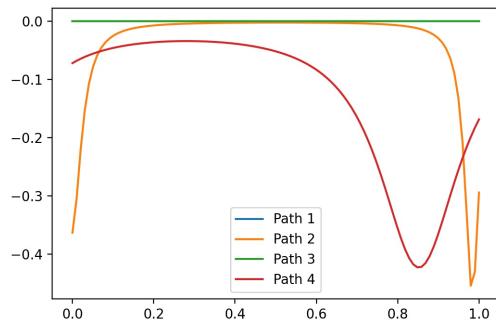


Figure 7.22. Subroutine test 2:angular velocity

Chapter 8

Conclusions and Future Works

8.1 Future works

Section 7 has shown that this approach is able to find a solution for the Multi-point path planning problem, but is it evident that there is still plenty of space for further improvements.

The limitations of structure 1 are evident, the number of goal states cannot be changed once the network has been trained. In Structure 2 an important information about the nature of our problem has not been used: the map is bi-dimensional and could be treated as an image instead of "vectorizing" the whole map. The natural way to use this prior would be that of using Convolutional Neural Networks and feed the whole image to the network. Then the huge amount of filters would understand under which circumstances is useful to steer or to go straight. An even finer method would be that of using CNN+LSTM's as in [9], a special class of Recurrent Neural Networks which take into account sequences of actions and has proven to be utterly effective in natural language processing.

Furthermore, a continuous action space could be investigated in order for the robot to perform more complex movements, for instance even going backwards like the Reed's and Shepp's WMR.

In order for DRL to become a valid alternative to DTSP problems also on an industrial level would be to raise the complexity of the problem. In the proposed approach the images captured by the camera of the robot have not been used. They represent an extremely important information since once the robot gets near to a goal state he could use the field of view of the robot to choose if it is worth to visit that area or if it does not contain any litter piece and therefore is not worth visiting. This problem is very complex since it needs to process mixed inputs both from a global view of the map as seen from above (equivalent to the one proposed by this work) but also the information from the sensors and from the camera.

8.2 Conclusions

This master thesis has operated in the field of path planning algorithms. This field has been greatly investigated by classical control or probabilistic methods and

in more recent years also with Deep Reinforcement Learning. In particular, the research direction in which this thesis has moved is that of multi point path planning, achieving in optimizing a policy network through the well known state-of-the-art algorithm Proximal Policy Optimization. The obtained trajectories confirm that this method could represent the starting point to develop a more advanced navigation system for inspection of some areas on a map. In addition a nice property which the proposed method has over its competitors is that the policy is stored into a neural network which, after having been trained, does not need further computational effort, unlike methods like Dubins Traveling Salesman Problem which need to compute a solution for each specific distribution of goal states.

Chapter 9

Code

In the following part, the Code is presented. This code solves the task for a 12×12 map using Structure 2.

9.1 Structure 2

```

1 !pip install stable-baselines3[extra]
2 import stable_baselines3
3
4 import gym
5 import numpy as np
6 from gym import Env
7 from gym.spaces import Discrete, Box
8 import random
9 import matplotlib.pyplot as plt
10 from matplotlib import colors
11 import time
12 import copy
13 import torch as th
14
15 from stable_baselines3 import PPO
16 from stable_baselines3.ppo import MlpPolicy
17
18 #----- Define a new cell in Google Colab
19
20 gain=1
21 penalty=0.0
22 size_x=12
23 size_y=12
24 mosse_max=200
25
26 flag_robot_su_griglia=-1
27 v_mid=0.5
28 omega_max=0.3
29 delta_t=(2*np.pi/(8*omega_max))
30 action_list=[ [v_mid,-omega_max], [v_mid,omega_max],[v_mid,0], ]
31

```

```

32 def init_reward_grid():
33     grid=np.zeros((17,20))
34     somma=0
35     c=0
36     d=0
37     e=0
38     f=0
39     g=0
40     if random.randint(0,1)==1:#random.randint(0,2)
41         while c+d+e+f+g<3:
42             c=random.randint(0,1)
43             d=random.randint(0,1)
44             e=random.randint(0,1)
45             f=random.randint(0,1)
46             g=random.randint(0,1)
47     else:
48         c=1
49         d=1
50         e=1
51         f=1
52         g=1
53
54     grid[random.randint(1,3),random.randint(1,5)]=c
55     grid[random.randint(0,6),random.randint(14,18)]=d
56     grid[random.randint(6,16),random.randint(0,3)]=e
57     grid[random.randint(7,16),random.randint(11,14)]=f
58     grid[1,random.randint(9,11)]=g
59     grid[13:,10]=-2
60     grid[4:6,12:14]=-2
61     grid[5,0:6]=-2
62     grid[6:9,5]=-2
63
64     grigliona_pixel=copy.copy(grid)
65     for i in range(17):
66         for j in range(20):
67             if grigliona_pixel[i,j]>0:
68                 somma+=grigliona_pixel[i,j]
69                 grigliona_pixel[i,j]=1
70     return grid,somma,grigliona_pixel
71
72 def ricompensa(lista):
73     reward=0
74     for i in range(len(lista)):
75         if lista[i]>0:
76             reward+=lista[i]
77     return reward
78
79 def resetta(x,y):
80     for i in range(-1,2):
81         for j in range(-1,2):
82             if x+i<size_x and x+i>=0 and y+j<size_y and y+j>=0:
83                 if env.reward_grid[x+i,y+j]>0:
84                     env.reward_grid[x+i,y+j]=0

```

```
85
86 def get_starting_location():
87     return 5, 11, 0.75
88
89 def vectorize_map(map):
90     vector = [i for i in np.reshape(map,-1) if i>-2]
91     return vector
92
93 size=len(vectorize_map(init_reward_grid()[0]))
94
95
96
97 #-----Define a new cell in Google Colab
98 class PiazzaDelPopolo(Env):
99     def __init__(self):
100         pos_x_init=0
101         pos_y_init=0
102         theta_in=0
103         self.reward_grid=[]
104         self.state=np.zeros(size+4)
105         self.state[0:4] = [pos_x_init/size_x,pos_y_init/size_y,theta_in,0]
106         self.state[4:size+4] = np.zeros(size)
107         self.action_space = Discrete(len(action_list))
108         highh=np.concatenate((np.array([1,1,1,1]),np.ones(size)))
109         loww= np.concatenate((np.array([0,0,0,0]),-np.ones(size)))
110         self.observation_space = Box(low=loww,high=highh)
111         self.stato_memoria=[]
112         self.mossa=0
113
114
115     def step(self, action):
116         self.mossa+=1
117         done=False
118         if self.mossa >mosse_max:
119             reward=-1
120             lista_aree.append(self.state[3])
121             numero_mosse_massime[0]+=1
122             done=True
123             info={}
124             return self.state, reward, done, info
125
126         # metto a zero l'elemento della pixel grid in cui stava prima il
127         # robot
128         self.reward_grid[int(np.round((size_x-1)*self.state[0])),int(np.
129             round((size_y-1)*self.state[1]))]=0
130
131         self.stato_memoria=[self.state[0],self.state[1]]
132
133         if self.state[0]<=1 and self.state[0]>=0 and self.state[1]<=1 and
134             self.state[1]>=0:
135             angolo=self.state[2]*np.pi*2
136             self.state[0]+=delta_t*np.cos(angolo)*action_list[action]/(
137                 size_x-1)
```

```

134         self.state[1]+=delta_t*np.sin(angolo)*action_list[action][0]/(size_y-1)
135         self.state[2]+=delta_t*action_list[action][1]/(2*np.pi)
136
137     if self.state[2]>1:
138         self.state[2]-=1
139     if self.state[2]<0:
140         self.state[2]+=1
141
142     if self.state[0]>=1:
143         reward=-penalty
144         self.state[0:2]=self.state_memoria
145         self.reward_grid[int(np.round(self.state[0]*(size_x-1))),int(np.
146 round(self.state[1]*(size_y-1)))] = flag_robot_su_griglia
147         self.state[4:size+4] = vectorize_map(self.reward_grid)
148
149     if self.state[0]<=0:
150         reward=-penalty
151         self.state[0:2]=self.state_memoria
152         self.reward_grid[int(np.round(self.state[0]*(size_x-1))),int(np.
153 round(self.state[1]*(size_y-1)))] = flag_robot_su_griglia
154         self.state[4:size+4] = vectorize_map(self.reward_grid)
155
156     if self.state[1]>=1:
157         reward=-penalty
158         self.state[0:2]=self.state_memoria
159         self.reward_grid[int(np.round(self.state[0]*(size_x-1))),int(np.
160 round(self.state[1]*(size_y-1)))] = flag_robot_su_griglia
161         self.state[4:size+4] = vectorize_map(self.reward_grid)
162         self.state[0:2]=self.state_memoria
163
164     if self.state[1]<=0:
165         reward=-penalty
166         self.state[0:2]=self.state_memoria
167         self.reward_grid[int(np.round(self.state[0]*(size_x-1))),int(np.
168 round(self.state[1]*(size_y-1)))] = flag_robot_su_griglia
169         self.state[4:size+4] = vectorize_map(self.reward_grid)
170
171     if self.reward_grid[int(np.round((size_x-1)*self.state[0])),int(np.
172 round((size_y-1)*self.state[1]))]<-1.6:
173         reward=-penalty
174         self.state[0:2]=self.state_memoria
175         self.reward_grid[int(np.round(self.state[0]*(size_x-1))),int(np.
176 round(self.state[1]*(size_y-1)))] = flag_robot_su_griglia
177         self.state[4:size+4] = vectorize_map(self.reward_grid)
178
179     pos_x=int(np.round(self.state[0]*(size_x-1)))
180     pos_y=int(np.round(self.state[1]*(size_y-1)))
181
182     reward=ricompensa(self.reward_grid[np.max([0,pos_x-1]):np.min([
183 pos_x+2,19]),np.max([pos_y-1,0]):np.min([pos_y+2,19])].reshape(-1))

```

```

179         self.reward_grid[pos_x,pos_y]=flag_robot_su_griglia
180
181     if reward>0:
182         resetta(pos_x,pos_y)
183         self.state[3]+=reward/3
184
185     self.state[4:size+4] = vectorize_map(self.reward_grid)
186
187     if self.mossa >mosse_max:
188         reward=-1
189         lista_aree.append(self.state[3])
190         numero_mosse_massime[0]+=1
191         done=True
192         info={}
193         return self.state, reward, done, info
194     else:
195         done=False
196
197     if self.state[3]>=0.99:
198         reward=1
199         lista_aree.append(self.state[3])
200         numero_partite_vinte[0]+=1
201         done=True
202         info={}
203         return self.state, reward, done, info
204
205     info = {}
206     reward=reward*0.5*(1+self.state[3])**2
207     return self.state, reward, done, info
208
209 def render(self):
210     pass
211
212 def reset(self):
213     self.state=np.zeros(size+4)
214     self.reward_grid=init_reward_grid()
215     pos_x_reset,pos_y_reset,theta_in=get_starting_location()
216     self.state[2]=theta_in
217     self.reward_grid[pos_x_reset,pos_y_reset]=flag_robot_su_griglia
218     self.state[0:2] = [pos_x_reset/(size_x-1),pos_y_reset/(size_y-1)]
219     self.state[4:size+4] = vectorize_map(self.reward_grid)
220     self.mossa=0
221     return self.state
222
223 env=PiazzaDelPopolo()
224
225 def plottagrid(matrice):
226
227     px=env.state[0]*(size_x-1)
228     py=env.state[1]*(size_y-1)
229     cmap = colors.ListedColormap(['blue', 'red','green','pink'])
230     bounds = [-2,-0.1,0.1,0.3,0.5]
231     norm = colors.BoundaryNorm(bounds, cmap.N)

```

```

232     theta=env.state[2]*2*np.pi
233     lungo=1
234
235     fig, ax = plt.subplots(figsize=(8, 8))
236
237     ax.imshow(matrice, cmap=cmap,norm=norm)
238     ax.grid(which='major', axis='both', linestyle='-', color='k', linewidth=2)
239     ax.set_xticks(np.arange(-0.5, 20, 1));
240
241     ax.set_yticks(np.arange(-0.5, 17, 1));
242
243     plt.plot([py,py+lungo*np.sin(theta)], [px,px+lungo*np.cos(theta)], linewidth=5,color='white')
244     plt.plot(py, px, marker="o", markersize=20, markeredgecolor="white",
245               markerfacecolor="white")
245 #-----Define a new cell to test the environment and see if it
246   performs as expected
246
247 episodes = 1
248 for episode in range(1, episodes+1):
249     env.reset()
250     done = False
251     score = 0
252     for i in range(10):
253         action = env.action_space.sample()
254         n_state, reward, done, info = env.step(action)
255         score+=reward
256         plothagrid(env.pixel_grid.reshape(size_x,size_y))
257         print('posizione',[env.state[0]*size_x,env.state[1]*size_y,env.
258 state[2]*np.pi*2,env.state[3]],'reward',reward,'azione',action_list[
259           action])
259 #-----New cell to define Schedulers
260
261 class Schedule(object):
262     def value(self, step):
263         raise NotImplementedError
264
265
266     def learning_rate_custom_PPO() -> Schedule:
267         def func(progress_remaining: float) -> float:
268             progress_remaining=1-progress_remaining
269             if progress_remaining < 0.2:
270                 return 0.0003
271             if progress_remaining < 0.4:
272                 return 0.0002
273             if progress_remaining < 0.5:
274                 return 0.00015
275             if progress_remaining < 0.6:
276                 return 0.0001
277             if progress_remaining < 0.8:
278                 return 0.0008

```

```
279     if progress_remaining < 0.9:
280         return 0.00003
281     else:
282         return 0.00008
283
284
285     return func
286
287 def learning_rate_custom_DQN() -> Schedule:
288
289     def func(progress_remaining: float) -> float:
290         progress_remaining=1-progress_remaining
291         if progress_remaining < 0.1:
292             return 0.00005
293         if progress_remaining < 0.2:
294             return 0.00005
295         if progress_remaining < 0.3:
296             return 0.00004
297         if progress_remaining < 0.4:
298             return 0.00004
299         if progress_remaining < 0.5:
300             return 0.00004
301         if progress_remaining < 0.6:
302             return 0.00003
303         if progress_remaining < 0.7:
304             return 0.00003
305         if progress_remaining < 0.8:
306             return 0.00002
307         if progress_remaining < 0.9:
308             return 0.00001
309         else:
310             return 0.00001
311
312     return func
313 def learning_rate_custom_A2C() -> Schedule:
314     def func(progress_remaining: float) -> float:
315         progress_remaining=1-progress_remaining
316         if progress_remaining < 0.1:
317             return 0.0007
318         if progress_remaining < 0.2:
319             return 0.0006
320         if progress_remaining < 0.3:
321             return 0.0005
322         if progress_remaining < 0.4:
323             return 0.0004
324         if progress_remaining < 0.5:
325             return 0.0003
326         if progress_remaining < 0.6:
327             return 0.0003
328         if progress_remaining < 0.7:
329             return 0.0002
330         if progress_remaining < 0.8:
331             return 0.0002
```

```

332         if progress_remaining < 0.9:
333             return 0.0001
334         else:
335             return 0.0001
336
337     return func
338
339 #-----Define a new cell to set the model
340
341
342 policy_kwargs = dict(activation_fn=th.nn.Tanh,net_arch=[dict(pi=[200,
343                 200,200], vf=[200, 200,200])])
344
345 policy_kwargs_DQN = dict(activation_fn=th.nn.Tanh,
346                           net_arch=[200, 200,200])
347 Model_type='DQN' #DQN,PPO,A2C
348
349 if Model_type=='DQN':
350     model = stable_baselines3.dqn.DQN("MlpPolicy", env,
351                                         learning_rate=
352                                         learning_rate_custom_DQN(),policy_kwargs=policy_kwargs_DQN,
353                                         batch_size=1600, verbose=0,
354                                         exploration_initial_eps=1,
355                                         exploration_fraction=0.1,train_freq
356                                         =(1, "episode"),max_grad_norm =2,gradient_steps=2,
357                                         tensorboard_log="../pixies_tensorboard
358                                         /")
359 if Model_type=='PPO':
360     model = PPO(MlpPolicy, env,policy_kwargs=policy_kwargs, ent_coef=0.0,
361     gamma=0.995,
362                                         learning_rate=
363                                         learning_rate_custom_PPO(),n_epochs=4,
364                                         batch_size=400,verbose=0,n_steps
365                                         =2000,
366                                         tensorboard_log="../pixies_tensorboard
367                                         /")
368
369 if Model_type=='A2C':
370     model = stable_baselines3.a2c.A2C("MlpPolicy",env, learning_rate
371     =0.0007,
372     n_steps=500, gamma=0.99, gae_lambda=1.0, ent_coef=0.0, vf_coef=0.5,
373     max_grad_norm=0.5, rms_prop_eps=1e-05,policy_kwargs=policy_kwargs,
374     tensorboard_log="../pixies_tensorboard/")
375
376
377 # PPO documentation on https://stable-baselines3.readthedocs.io/en/master/
378 #modules/ppo.html
379
380
381 #-----Cell to load the tensordboard
382 %load_ext tensorboard

```

```

373 %tensorboard --logdir ./pixies_tensorboard/
374
375 #-----Cell to train the model
376 model.learn(total_timesteps=5000000)
377
378 #----- Print some successful trajectories
379 episodes = 5
380
381 for episode in range(1, episodes+1):
382     done = False
383     obs = env.reset()
384     score = 0
385     plottagrid(env.reward_grid)
386     for i in range(100):
387
388         if done==True:
389             break
390         action, _states = model.predict(obs)
391         n_state, reward, done, info = env.step(action)
392
393         plt.plot([env.state[1]*(size_y-1),env.state[1]*(size_y-1)+0.3*np.sin(env.state[2]*2*np.pi)], [env.state[0]*(size_x-1),env.state[0]*(size_x-1)+0.3*np.cos(env.state[2]*2*np.pi)], linewidth=5,color='white')
394         plt.plot(env.state[1]*(size_y-1), env.state[0]*(size_x-1), marker="o", markersize=15, markeredgewidth="white", markerfacecolor="white")

```

Listing 9.1. OpenAI Gym

9.2 Subroutine Code

```

1 N=4
2
3 graph=np.zeros((N,N))
4 X=np.random.uniform(-4,4,N-1)
5 X=np.concatenate((np.zeros(1),X))
6 Y=np.random.uniform(0,4,N-1)
7 Y=np.concatenate((np.zeros(1),Y))
8 colors = [0 for i in range(N)]
9 area = (30 * 1)**2 # 0 to 15 point radii
10 fig, ax = plt.subplots()
11 plt.scatter(X, Y, s=area, c=colors, alpha=0.5)
12 plt.xlim([-6, 6])
13 plt.ylim([-2, 6])
14 n = range(N)
15 for i, txt in enumerate(n):
16     ax.annotate(txt, (X[i], Y[i]))
17
18 for i in range(1,N):
19     for j in range(0,i):
20         graph[i][j]=(((X[i])-X[j])**2+(Y[i]-Y[j])**2)**(0.5)
21 dist=graph+graph.T

```

```

22 # Helper function to calculate path length
23 def path_len(path):
24     return sum(dist[i][j] for i, j in zip(path, path[1:]))
25
26 # Set of all nodes to visit
27 to_visit = set(range(len(dist)))
28
29 # Current state {(node, visited_nodes): shortest_path}
30 state = {(i, frozenset([0, i])): [0, i] for i in range(1, len(dist[0]))}
31
32 for _ in range(len(dist) - 2):
33     next_state = {}
34     for position, path in state.items():
35         current_node, visited = position
36
37         # Check all nodes that haven't been visited so far
38         for node in to_visit - visited:
39             new_path = path + [node]
40             new_pos = (node, frozenset(new_path))
41
42             # Update if (current node, visited) is not in next state or we
43             # found shorter path
44             if new_pos not in next_state or path_len(new_path) < path_len(
45                 next_state[new_pos]):
46                 next_state[new_pos] = new_path
47
48     state = next_state
49
50 # Find the shortest path from possible candidates
51 shortest = min((path + [0] for path in state.values()), key=path_len)
52 print('path: {}', length: {}'.format(shortest, path_len(shortest)))
53 theta_= [0 for i in range(len(shortest))]
54
55 for i in range(len(shortest)-1):
56     if i%2==0:
57         vecio=shortest[i]
58         nuovo=shortest[i+1]
59         norm=((X[nuovo]-X[vecio])**2+(Y[nuovo]-Y[vecio])**2)**(0.5)
60         theta_[i:i+1]=[math.atan2(Y[nuovo]-Y[vecio],X[nuovo]-X[vecio]),math.
61         atan2(Y[nuovo]-Y[vecio],X[nuovo]-X[vecio])]
62         plt.plot([X[vecio], X[nuovo]], [Y[vecio], Y[nuovo]])
63         plt.plot([X[vecio], X[vecio]+(X[nuovo]-X[vecio])/norm], [Y[vecio],
64         Y[vecio]+(Y[nuovo]-Y[vecio])/norm], linewidth=2,color='red')
65         plt.plot([X[nuovo], X[nuovo]+(X[nuovo]-X[vecio])/norm], [Y[nuovo],
66         Y[nuovo]+(Y[nuovo]-Y[vecio])/norm], linewidth=2,color='red')
67
68 theta_=theta_[0:6]
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
737
738
739
739
740
741
742
743
744
745
745
746
747
747
748
749
749
750
751
752
753
754
755
755
756
757
757
758
759
759
760
761
762
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
1563
1564
1564
1565
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1570
1571
1571
1572
1572
1573
1573
1574
1574
1575
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1580
1581
1581
1582
1582
1583
1583
1584
1584
1585
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1590
1591
1591
1592
1592
1593
1593
1594
1594
1595
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1600
1601
1601
1602
1602
1603
1603
1604
1604
1605
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1610
1611
1611
1612
1612
1613
1613
1614
1614
1615
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1620
1621
1621
1622
1622
1623
1623
1624
1624
1625
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1630
1631
1631
1632
1632
1633
1633
1634
1634
1635
1635
1636
1636
1637
1637
1638
1638
1639
1639
1640
1640
1641
1641
1642
1642
1643
1643
1644
1644
1645
1645
1646
1646
1647
1647
1648
1648
1649
1649
1650
1650
1651
1651
1652
1652
1653
1653
1654
1654
1655
1655
1656
1656
1657
1657
1658
1658
1659
1659
1660
1660
1661
1661
1662
1662
1663
1663
1664
1664
1665
1665
1666
1666
1667
1667
1668
1668
1669
1669
1670
1670
1671
1671
1672
1672
1673
1673
1674
1674
1675
1675
1676
1676
1677
1677
1678
1678
1679
1679
1680
1680
1681
1681
1682
1682
1683
1683
1684
1684
1685
1685
1686
1686
1687
1687
1688
1688
1689
1689
1690
1690
1691
1691
1692
1692
1693
1693
1694
1694
1695
1695
1696
1696
1
```

```

70 fig0,ax0=plt.subplots()
71 fig1,ax1=plt.subplots()
72 fig2,ax2=plt.subplots()
73 def interpolate(x_i,y_i,tet_i,x_f,y_f,tet_f,T,discretizations,jjj):
74     factor=1
75     s=np.linspace(0,1,discretizations)
76     i=0
77     K=1
78     alpha_x=K*np.cos(tet_f)-3*x_f
79     alpha_y=K*np.sin(tet_f)-3*y_f
80
81     beta_x=K*np.cos(tet_i)+3*x_i
82     beta_y=K*np.sin(tet_i)+3*y_i
83
84     x=-(s-1)**3*x_i+s**3*x_f+alpha_x*s**2*(s-1)+beta_x*s*(s-1)**2
85     y=-(s-1)**3*y_i+s**3*y_f+alpha_y*s**2*(s-1)+beta_y*s*(s-1)**2
86     xdot=(-3*(s-1)**2*x_i+3*s**2*x_f+  alpha_x*(3*s**2-2*s)+beta_x*(3*s
87     **2-4*s+1))/T
88     ydot=(-3*(s-1)**2*y_i+3*s**2*y_f+  alpha_y*(3*s**2-2*s)+beta_y*(3*s
89     **2-4*s+1))/T
90
91     xdotdot=(-6*(s-1)*x_i+6*s*x_f+  alpha_x*(6*s-2)+beta_x*(6*s-4))/T**2
92
93     ydotdot=(-6*(s-1)*y_i+6*s*y_f+  alpha_y*(6*s-2)+beta_y*(6*s-4))/T**2
94
95     velocity=(xdot**2+ydot**2)**(0.5)
96     omega=(ydotdot*xdot-xdotdot*ydot)/(xdot**2+ydot**2)
97
98     omega=np.nan_to_num(omega)
99     #print(omega)
100    print('test',np.max(velocity),np.max(omega),np.min(omega))
101
102    while np.max(velocity)>0.51 or np.max(abs(omega))>0.51:
103        i+=1
104
105        factor=np.max([np.max(velocity),np.max(abs(omega))])*(1.1)/0.5
106        T=factor*T
107        print(factor)
108        s=np.linspace(0,1,discretizations)
109        i=0
110        K=0.5
111        alpha_x=K*np.cos(tet_f)-3*x_f
112        alpha_y=K*np.sin(tet_f)-3*y_f
113
114        beta_x=K*np.cos(tet_i)+3*x_i
115        beta_y=K*np.sin(tet_i)+3*y_i
116
117        x=-(s-1)**3*x_i+s**3*x_f+alpha_x*s**2*(s-1)+beta_x*s*(s-1)**2
118        y=-(s-1)**3*y_i+s**3*y_f+alpha_y*s**2*(s-1)+beta_y*s*(s-1)**2
119        xdot=(-3*(s-1)**2*x_i+3*s**2*x_f+  alpha_x*(3*s**2-2*s)+beta_x*(3*s
120        **2-4*s+1))/T
121        ydot=(-3*(s-1)**2*y_i+3*s**2*y_f+  alpha_y*(3*s**2-2*s)+beta_y*(3*s
122        **2-4*s+1))/T

```

```

119
120     xdotdot=(-6*(s-1)*x_i+6*s*x_f+ alpha_x*(6*s-2)+beta_x*(6*s-4))/T
121     **2
122
123     ydotdot=(-6*(s-1)*y_i+6*s*y_f+ alpha_y*(6*s-2)+beta_y*(6*s-4))/T
124     **2
125     velocity=(xdot**2+ydot**2)**(0.5)
126     omega=(ydotdot*xdot-xdotdot*ydot)/(xdot**2+ydot**2)
127
128     omega=np.nan_to_num(omega)
129
130
131     ax0.plot(s,velocity,label='Velocity_'+str(jjj))
132     ax2.plot(s,omega,label='Omega_'+str(jjj))
133     return [list(x),list(y),list(xdot),list(ydot)],T*factor
134
135 xxx=[]
136 yyy=[]
137
138 xxx_dot=[]
139 yyy_dot=[]
140
141
142
143
144 T=20
145 discretizations=100
146
147 theta_[len(shortest)-1]=theta_[0]
148 delta_ttt=np.zeros(10)
149 for i in range(len(shortest)-1):
150
151     uscita,ttt_=interpolate(X[shortest[i]],Y[shortest[i]],theta_[i],X[
152         shortest[i+1]],Y[shortest[i+1]],theta_[i+1],T,discretizations,i)
153     delta_ttt[i]=ttt_/discretizations
154     x_=uscita[0]
155     y_=uscita[1]
156     x_dot_=uscita[2]
157     y_dot_=uscita[3]
158     xxx.append(x_)
159
160     yyy.append(y_)
161     xxx_dot.append(x_dot_)
162     yyy_dot.append(y_dot_)
163
164 xxx=np.reshape(xxx,-1)
165 yyy=np.reshape(yyy,-1)
166
167 xxx_dot=np.reshape(xxx_dot,-1)
168 yyy_dot=np.reshape(yyy_dot,-1)

```

```

169
170
171 b=0.2
172 track_x=[0.3]
173 track_y=[-0.3]
174 track_th=[theta_0*1.1]
175 gamma=0.1
176
177
178 print(len(xxx_dot),len(track_x),len(track_th),len(xxx))
179 for j in range(4):
180     for i in range(discretizations):
181         if i==0 and j==0:
182             continue
183         i=discretizations*j+i
184         delta_t=delta_ttt[j]
185         desired=np.array([[xxx_dot[i-1]-gamma*(track_x[i-1]+b*np.cos(
186             track_th[i-1])-xxx[i-1]),
187             [yyy_dot[i-1]-gamma*(track_y[i-1]+b*np.sin(track_th[i
188             -1])-yyy[i-1])]])
189         A=np.linalg.inv(np.array([[np.cos(track_th[i-1]),-b*np.sin(track_th
190             [i-1]),[np.sin(track_th[i-1]),b*np.cos(track_th[i-1])]])))
191         vec=A.dot(desired)
192         v=np.clip(vec[0],0,0.5)
193         omega=np.clip(vec[1],-0.5,0.5)
194         track_x=np.append(track_x,track_x[i-1]+delta_t*np.cos(track_th[i
195             -1])*v)
196         track_y=np.append(track_y,track_y[i-1]+delta_t*np.sin(track_th[i
197             -1])*v)
198         track_th=np.append(track_th,track_th[i-1]+delta_t*omega)
199
200         track_x=np.reshape(track_x,-1)
201         track_y=np.reshape(track_y,-1)
202         track_th=np.reshape(track_th,-1)
203
204 ax1.scatter(track_x[0], track_y[0],c='#1f77b4', s=area, alpha=0.4)
205 ax1.scatter(track_x[discretizations], track_y[discretizations],c='#1f77b4',
206             s=area, alpha=0.4)
207 ax1.scatter(track_x[2*discretizations], track_y[2*discretizations], c='#1
208             f77b4',s=area, alpha=0.4)
209 ax1.scatter(track_x[3*discretizations], track_y[3*discretizations], c='#1
210             f77b4',s=area, alpha=0.4)
211
212 ax1.plot(np.reshape(track_x+b*np.cos(track_th),-1)[0:discretizations],np.
213             reshape(track_y+b*np.sin(track_th),-1)[0:discretizations],linewidth=2,
214             color='C0')
215 ax1.plot(np.reshape(track_x+b*np.cos(track_th),-1)[discretizations:2*
216             discretizations],np.reshape(track_y+b*np.sin(track_th),-1)[
217                 discretizations:2*discretizations],linewidth=2,color='C1')
218 ax1.plot(np.reshape(track_x+b*np.cos(track_th),-1)[2*discretizations:3*
219             discretizations],np.reshape(track_y+b*np.sin(track_th),-1)[2*
220                 discretizations:3*discretizations],linewidth=2,color='C2')

```

```

207 ax1.plot(np.reshape(track_x+b*np.cos(track_th),-1)[3*discretizations:],np.
208   reshape(track_y+b*np.sin(track_th),-1)[3*discretizations:],linewidth=2,
209   color='C3')
210
211 area = (20 * 1)**2
212 b=b*1.5
213 ax1.plot([track_x[0], track_x[0]+b*np.cos(track_th[0])], [track_y[0],
214   track_y[0]+b*np.sin(track_th[0])], linewidth=3,color='blue')
215 ax1.plot([track_x[discretizations], track_x[discretizations]+b*np.cos(
216   track_th[discretizations])], [track_y[discretizations], track_y[
217   discretizations]+b*np.sin(track_th[discretizations])], linewidth=3,
218   color='blue')
219 ax1.plot([track_x[2*discretizations], track_x[2*discretizations]+b*np.cos(
220   track_th[2*discretizations])], [track_y[2*discretizations], track_y[2*
221   discretizations]+b*np.sin(track_th[2*discretizations])], linewidth=3,
222   color='blue')
223 ax1.plot([track_x[3*discretizations], track_x[3*discretizations]+b*np.cos(
224   track_th[3*discretizations])], [track_y[3*discretizations], track_y[3*-
225   discretizations]+b*np.sin(track_th[3*discretizations])], linewidth=3,
226   color='blue')
227
228
229 ax1.scatter(X, Y, s=area, c=colors, alpha=0.4)
230
231
232 ax0.legend(['Path 1', 'Path 2','Path 3','Path 4'])
233 ax1.legend(['Path 1', 'Path 2','Path 3','Path 4'])
234 ax2.legend(['Path 1', 'Path 2','Path 3','Path 4'])
235
236 fig0.savefig('ciao0.jpg',dpi=200)
237 fig1.savefig('ciao1.jpg',dpi=200)
238 fig2.savefig('ciao2.jpg',dpi=200)

```

Listing 9.2. Subroutine Code

9.3 ROS code

```
1 #!/usr/bin/env python3
2
3 from __future__ import print_function
4
5
6
7 import rospy
8
9 from geometry_msgs.msg import Twist
10 from geometry_msgs.msg import TwistStamped
11
12 from geometry_msgs.msg import Pose
13
14 from gazebo_msgs.msg import ModelStates
15 from tf.transformations import euler_from_quaternion, quaternion_from_euler
16
17
18 import sys
19 from select import select
20 import time
21 import timeit
22
23 import numpy as np
24
25
26 lista_x=[0.41016769, 0.3030303 , 0.19589291, 0.15151515, 0.10713739
27     ,0.06275963,
28     0.10713739, 0.21427478, 0.25141217, 0.36578993, 0.32141217, 0.21427478,
29     0.10713739, 0.21427478,
30     0.32141217, 0.42854956, 0.53568696, 0.64282435, 0.74996174 ,0.85709913,
31     0.97609885,1.1]
32
33 lista_y=[0.89286261 ,0.84848485, 0.80410709 ,0.6969697 , 0.58983231,
34     0.48269491,
35     0.37555752, 0.33117976, 0.206802 , 0.17966461 ,0.07252722, 0.02814946,
36     0.07252722, 0.02814946,
37     0.07252722, 0.11690498 ,0.16128274 ,0.2056605 , 0.25003826, 0.29441602,
38     0.29441602 ,0.29441602]
39
40 lista_theta=[0.625 , 0.5 , 0.625 , 0.75 , 0.625 ,
41     0.75,
42     0.875 , 1. , 0.875 , 0.75 , 0.625 , 0.5,
43     0.375 , 1.,
44     0.125 , 0. , 0.125 , 0. , 0.125, 0,0,0]
45
46
47
48
49 def interpolate(x_i,y_i,tet_i,x_f,y_f,tet_f,T,discretizations):
50     factor=1
```

```

51     s=np.linspace(0,1,discretizations)
52     i=0
53     K=1
54     alpha_x=K*np.cos(tet_f)-3*x_f
55     alpha_y=K*np.sin(tet_f)-3*y_f
56
57     beta_x=K*np.cos(tet_i)+3*x_i
58     beta_y=K*np.sin(tet_i)+3*y_i
59
60     x=-(s-1)**3*x_i+s**3*x_f+alpha_x*s**2*(s-1)+beta_x*s*(s-1)**2
61     y=-(s-1)**3*y_i+s**3*y_f+alpha_y*s**2*(s-1)+beta_y*s*(s-1)**2
62     xdot=(-3*(s-1)**2*x_i+3*s**2*x_f+  alpha_x*(3*s**2-2*s)+beta_x*(3*s
63     **2-4*s+1))/T
64     ydot=(-3*(s-1)**2*y_i+3*s**2*y_f+  alpha_y*(3*s**2-2*s)+beta_y*(3*s
65     **2-4*s+1))/T
66
66     return [list(x),list(y),list(xdot),list(ydot)]
67
67 xxx=[]
68 yyy=[]
69
70 xxx_dot=[]
71 yyy_dot=[]
72
73
74
75 # Generate the trajectory
76 T=3
77 discretizations=20
78
79
80 delta_ttt=T/discretizations
81 for i in range(len(lista_x)-1):
82     uscita=interpolate(lista_x[i],lista_y[i],lista_theta[i],lista_x[i+1],
83     lista_y[i+1],lista_theta[i+1],T,discretizations)
84
84     x_=uscita[0]
85     y_=uscita[1]
86     x_dot_=uscita[2]
87     y_dot_=uscita[3]
88     xxx.append(x_)
89
90     yyy.append(y_)
91     xxx_dot.append(x_dot_)
92     yyy_dot.append(y_dot_)
93
94 xxx=np.reshape(xxx,-1)
95 yyy=np.reshape(yyy,-1)
96
97 xxx_dot=np.reshape(xxx_dot,-1)
98 yyy_dot=np.reshape(yyy_dot,-1)
99
100

```

```
101 track_x=[0]
102 track_y=[0]
103 track_th=[0]
104
105
106 def get_rotation (msg):
107     global roll, pitch, yaw
108     orientation_q = msg.pose.pose.orientation
109     orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z,
110                         orientation_q.w]
111     (roll, pitch, yaw) = euler_from_quaternion (orientation_list)
112     return yaw
113
114
115
116 b=0.17
117 gamma=0.1
118 if __name__=="__main__":
119     rospy.init_node('teleop_twist_keyboard')
120     publisher = rospy.Publisher('cmd_vel', Twist, queue_size = 1)
121
122     for i in range(1,len(xxx)-1):
123
124         command_velocity=Twist()
125
126         data = rospy.wait_for_message('/gazebo/model_states', ModelStates,
127                                     timeout=5)
127         data=data.pose[5]
128         orientation_list = [data.orientation.x, data.orientation.y, data.
129                             orientation.z, data.orientation.w]
130         (roll, pitch, yaw) = euler_from_quaternion(orientation_list)
131
132         print('pose:',data.position.x,data.position.y,yaw)
133         print('pose displaced:',data.position.x+b*np.cos(yaw),data.position.
134             y,yaw+b*np.sin(yaw))
135         desired=np.array([[xxx_dot[i-1]-gamma*(data.position.x+b*np.cos(yaw
136             )-xxx[i-1])],[yyy_dot[i-1]-gamma*(data.position.y+b*np.sin(yaw)-yyy[i
137             -1])]])
138         A=np.linalg.inv(np.array([[np.cos(yaw),-b*np.sin(yaw)],[np.sin(yaw)
139             ,b*np.cos(yaw)])))
140         vec=A.dot(desired)
141         v=np.clip(vec[0],0,1)
142         omega=np.clip(vec[1],-1,1)
143
144         command_velocity.linear.x = v
145         command_velocity.linear.y = 0
146         command_velocity.linear.z = 0
147         command_velocity.angular.x = 0
```

```
147     command_velocity.angular.y = 0
148     command_velocity.angular.z = omega
149
150     publisher.publish(command_velocity)
152
153     time.sleep(delta_ttt)
154
155
156     command_velocity.linear.x = 0
157     command_velocity.linear.y = 0
158     command_velocity.linear.z = 0
159     command_velocity.angular.x = 0
160     command_velocity.angular.y = 0
161     command_velocity.angular.z = 0
162     print('velocity published')
163     publisher.publish(command_velocity)
```

Listing 9.3. ROS Python code

List of Figures

1.1	Performance of PPO versus other algorithms. [6]	3
1.2	“Learning to navigate in complex environments” [7]	3
1.3	“Dynamic Path Planning of Unknown Environment Based on Deep Reinforcement Learning” [8]	4
1.4	DTSPN succesful trajectories	5
2.1	Pixies Smart Bench	7
2.2	High level Scheme	8
2.3	Harware elements	9
2.4	Bottle detection and localization.	10
3.1	Fixed and Steerable wheels.	11
3.2	Caster wheel.	12
3.3	Definition of polar coordinates for Unicycle [14]	19
3.4	Posture regulation simulation [14]	20
4.1	Example of a MDP in the case of a Chess Game. Taken from [16]	24
4.2	Deep Neural Network	26
4.3	Actor Critic Methods [16]	31
4.4	Example of the Clipped surrogate loss function [6]	34
5.1	Example of a map Red=empty square, Pink=Via point, Blue=Obstacle	36
5.2	Example of a map Red=empty square, Pink=Via point, Blue=Obstacle,Green=Square occupied by the robot	39
5.3	Example of Voronoi Tesselation	41
6.1	Interpolation	44
7.1	Episode Reward for DQN,PPO,A2C Comparison. Red: PPO, Blue: DQN, Green: A2C	49
7.2	Episode length for DQN,PPO,A2C Comparison. Red: PPO, Blue: DQN, Green: A2C	49
7.3	Trajectories obtained from a deteriorated policy.	50
7.4	Learning deteriorating with negative rewards. Blue: P=-1, Cyan: P=-0.1, Orange: P=-0.01. (Reward)	51
7.5	Learning deteriorating with negative rewards. Blue: P=-1, Cyan: P=-0.1, Orange: P=-0.01. (Episode length)	51
7.6	Learning rate comparison. Episode Length	52

7.7	Learning rate comparison. Episode Length	52
7.8	Episode Reward on map of size 17×20	54
7.9	Episode length on map of size 17×20	54
7.10	Clip fraction on map of size 17×20	55
7.11	Two successful trajectories on map of size 17×20	55
7.12	The agent found a flaw in the environment and exploited it to win the game	56
7.13	Neural Network Architecture Comparison, part 1 Green: (50,50,50), Orange:(200,200,200),Cyan:(300,300,300)	57
7.14	Neural Network Architecture Comparison, part 2 Green: (50,50,50), Orange:(200,200,200),Cyan:(300,300,300)	58
7.15	Some succesful trajectories, NN=(300,300,300)	59
7.16	ROS environment	60
7.17	Subroutine test 1:Tracking	61
7.18	Subroutine test 1:linear velocity	62
7.19	Subroutine test 1:angular velocity	62
7.20	Subroutine test 2:Tracking	62
7.21	Subroutine test 2:linear velocity	63
7.22	Subroutine test 2:angular velocity	63

List of Tables

7.1	Environment parameters - Structure 1	47
7.2	PPO parameters - Structure 1	48
7.3	DQN parameters - Structure 1	48
7.4	A2C parameters - Structure 1	48
7.5	Learning Rate Schedulers	52
7.6	Environment parameters - Structure 2	53
7.7	PPO parameters - Structure 2	54
7.8	Number of parameters of each network	56

Bibliography

- [1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” 2013.
- [2] H. van Hasselt, A. Guez, and D. Silver, “Deep reinforcement learning with double q-learning,” *CoRR*, vol. abs/1509.06461, 2015.
- [3] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Mach. Learn.*, vol. 8, p. 229–256, may 1992.
- [4] R. Sutton, D. Mcallester, S. Singh, and Y. Mansour, “Policy gradient methods for reinforcement learning with function approximation,” *Adv. Neural Inf. Process. Syst.*, vol. 12, 02 2000.
- [5] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust region policy optimization,” *CoRR*, vol. abs/1502.05477, 2015.
- [6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal policy optimization algorithms,” *CoRR*, vol. abs/1707.06347, 2017.
- [7] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu, D. Kumaran, and R. Hadsell, “Learning to navigate in complex environments,” 2016.
- [8] X. Lei, Z. Zhang, and P. Dong, “Dynamic path planning of unknown environment based on deep reinforcement learning,” *Journal of Robotics*, vol. 2018, pp. 1–10, 09 2018.
- [9] P. Lou, K. Xu, X. Jiang, Z. Xiao, and J. Yan, “Path planning in an unknown environment based on deep reinforcement learning with prior knowledge,” *Journal of Intelligent Fuzzy Systems*, vol. 41, pp. 1–17, 10 2021.
- [10] L. Butyrev, T. Edelhäuser, and C. Mutschler, “Deep reinforcement learning for motion planning of mobile robots,” *CoRR*, vol. abs/1912.09260, 2019.
- [11] K. Savla, E. Frazzoli, and F. Bullo, “Traveling salesperson problems for the dubins vehicle,” *IEEE Transactions on Automatic Control*, vol. 53, no. 6, pp. 1378–1391, 2008.
- [12] P. Váňa and J. Faigl, “On the dubins traveling salesman problem with neighborhoods,” in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 4029–4034, 2015.

- [13] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, “Stable-baselines3: Reliable reinforcement learning implementations,” *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.
- [14] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2010.
- [15] A. Isidori, M. Thoma, E. D. Sontag, B. W. Dickinson, A. Fettweis, J. L. Massey, and J. W. Modestino, *Nonlinear Control Systems*. Berlin, Heidelberg: Springer-Verlag, 3rd ed., 1995.
- [16] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
- [17] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, 1989.
- [18] N. Qian, “On the momentum term in gradient descent learning algorithms,” *Neural Networks*, vol. 12, no. 1, pp. 145–151, 1999.
- [19] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” 2014.
- [20] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” *CoRR*, vol. abs/1602.01783, 2016.
- [21] J. Peters and S. Schaal, “Policy gradient methods for robotics,” in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2219–2225, 2006.
- [22] R. Tedrake, T. Zhang, and H. Seung, “Stochastic policy gradient reinforcement learning on a simple 3d biped,” in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 3, pp. 2849–2854 vol.3, 2004.
- [23] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, “Asynchronous methods for deep reinforcement learning,” *CoRR*, vol. abs/1602.01783, 2016.
- [24] L. E. Dubins, “On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents,” *American Journal of Mathematics*, vol. 79, no. 3, pp. 497–516, 1957.