,

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

This thesis has been typeset by LaTeX and the Sapthesis class.

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

“Un uomo tanto può quanto sa.
Sir Francis Bacon

Ringraziamenti

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 based method aimed at finding a trajectory for the unicycle model in order for the robot to visit some regions of interest on the map in an optimal way and in presence of obstacles. This regions are given a-priori and represent places where presence of litter has been statistically detected. Instead of planning a trajectory which connects two points at a time, for instance using 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 could also be considered as to find an empirical solution for the Traveling Salesman Problem (TSP) in the case of Unicycle model and presence of obstacles.
For simplicity reasons, the unicycle model moves according to Dubin's curves, i.e. either straight left or right with fixed linear speed and angular velocity equal to $\pm\omega_{max}$.

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. 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 classifies 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 di una mappa che contiene alcuni ostacoli. Questi punti sono dati a priori e rappresentano in questo caso delle regioni su cui è stata statisticamente determinata presenza di rifiuti. Invece di pianificare una traiettoria che connetta coppie di punti, come ad esempio mediante gli "Artificial Potential Fields", in questo caso 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 può essere considerato come un metodo empirico per risolvere il Traveling Salesman Problem in presenza di ostacoli e per il modello uniciclo.
Per mantenere una certa semplicità nell'approccio si è scelto di utilizzare come modello per l'uniciclo quello di Dubins, ovvero tale per cui il robot abbia velocità lineare costante e velocità angolare pari a $\pm\omega_{max}$.
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 se 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

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 behiours.

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 pioneristic 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 evolven 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. This 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.

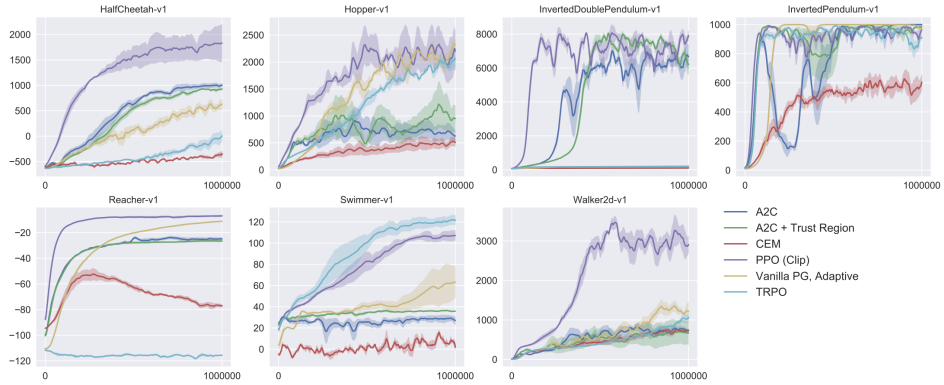


Figure 1.1. Performance of PPO versus other algorithms. [6]

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.

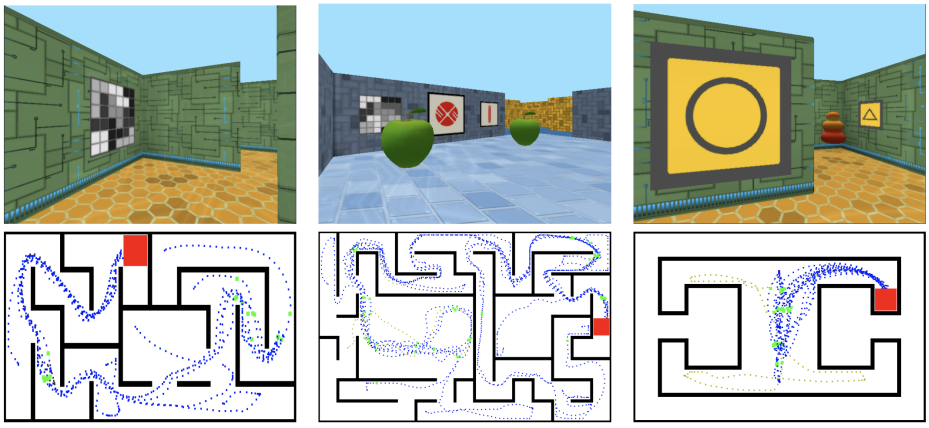


Figure 1.2. "Learning to navigate in complex environments" [7]

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.

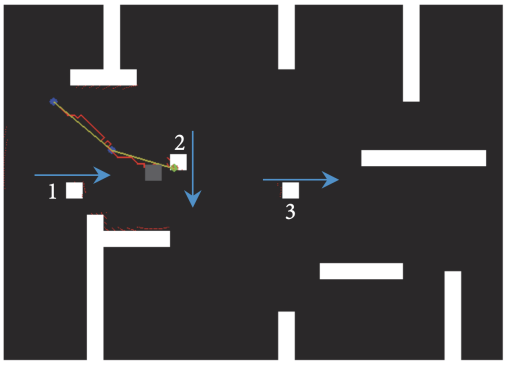


Figure 1.3. "Dynamic Path Planning of Unknown Environment Based on Deep Reinforcement Learning" [8]

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).

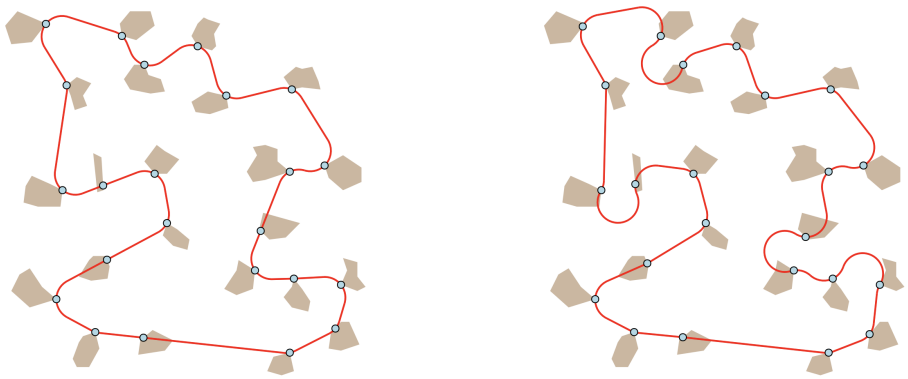


Figure 1.4. DTSPN succesful trajectories

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 reminder 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 litium-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

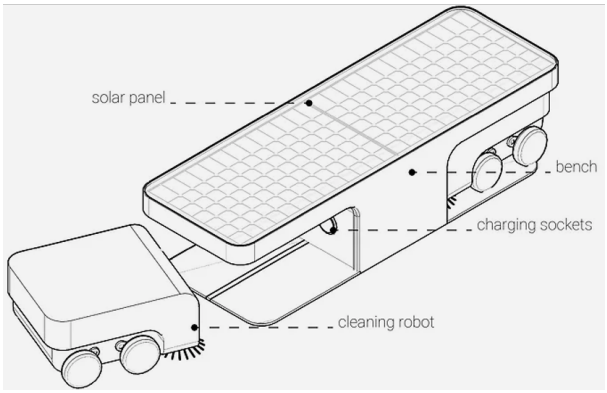


Figure 2.1. Pixies Smart Bench

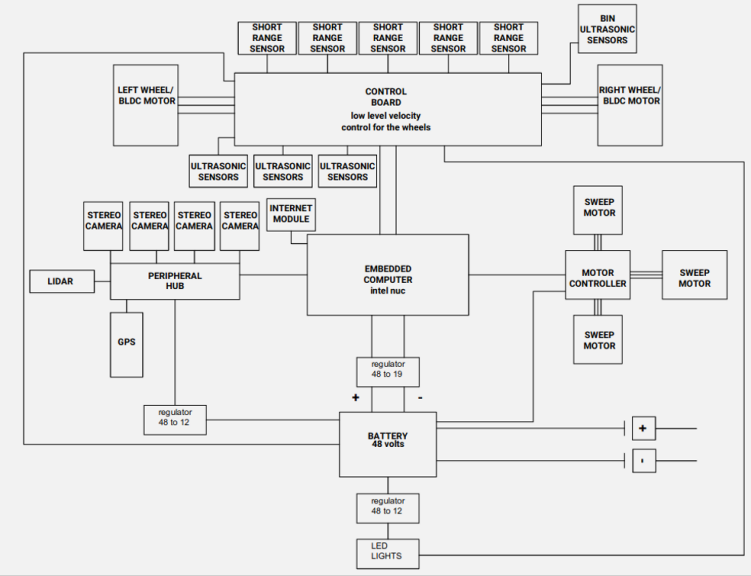


Figure 2.2. High level Scheme

Figure 2.3. ZLTECH motor

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 foundamental 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.

(a) Lidar

(b) Videocamera

(c) ZLTECH motor

Figure 2.4. 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 2.5. Bottle detection and localization.

Chapter 3

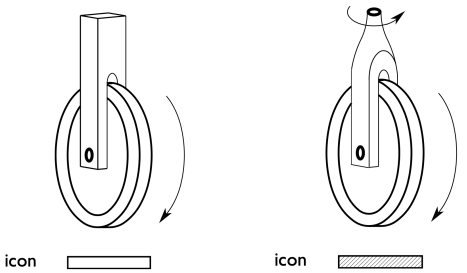
Wheeled mobile robots modeling and control

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. This 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.



(a) Fixed wheel (b) Steerable wheel

Figure 3.1. Fixed and Steerable wheels.

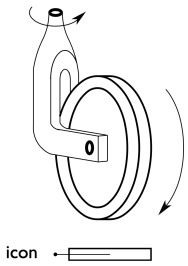


Figure 3.2. Caster wheel.

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:

$$v = v_r + v_l$$
$$\omega = b(v_r - v_l)$$

- **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 enter 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 differrential equations that describe the kinematics of the robot, as follows.

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

$$h_i(q) = 0 \quad i = 0, 1, \ldots, n \tag{3.1}$$

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

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

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 $span(Ker(A^T))$ 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

Frobwnius 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 $Ker(A(q)^T) = \Delta(q) = span(v_1, \ldots, v_d)$, then in order for $\Delta(q)$ to be integrable it mus be involutive, which means that:

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

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]:

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

And since $L_{vj}\lambda = 0$,$L_{vi}\lambda = 0$ 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:

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

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:

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

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

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

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.

$$\begin{cases} x_{k+1} = x_k + cos(\theta_k)v_kT \\ y_{k+1} = y_k + sin(\theta_k)v_kT \\ \theta_{k+1} = \theta_k + \omega_kT \end{cases} \tag{3.5}$$

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 in 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.

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

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.

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

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

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

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:

$$x = x(y, \dot{y}, \ldots, y^{(r)}) \tag{3.9}$$

$$u = u(y, \dot{y}, \ldots, y^{(r)}) \tag{3.10}$$

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

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

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

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

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 is 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:

$$\begin{cases} \dot{z}_1 = v_1 \\ \dot{z}_2 = v_2 \\ \dot{z}_3 = z_2 v_1 \\ \ldots \\ \dot{z}_n = z_{n-1} v_1 \end{cases} \tag{3.14}$$

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:

$$z_1 = \theta \tag{3.15}$$

$$z_2 = xcos(\theta) + ysin(\theta) \tag{3.16}$$

$$z_3 = xcos(\theta) - ysin(\theta) \tag{3.17}$$

$$v_1 = \omega \tag{3.18}$$

$$v_2 = v - z_3 v_1 \tag{3.19}$$

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.

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

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

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

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.

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

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 achieved this, a change of coordinates is performed:

$$x_b = x + bcos(\theta) \tag{3.23}$$
$$y_b = y + bsin(\theta) \tag{3.24}$$

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

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

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

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

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

$$\dot{x}_b - v_x^d = k_x(x_b - x^d) \tag{3.25}$$
$$\dot{y}_b - v_y^d = k_y(y_b - y^d) \tag{3.26}$$
$$\tag{3.27}$$

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:

$$\rho = \sqrt{x^2 + y^2} \tag{3.28}$$

$$\gamma = Atan2(y, x) - \theta + \pi \tag{3.29}$$

$$\delta = \gamma + \theta \tag{3.30}$$

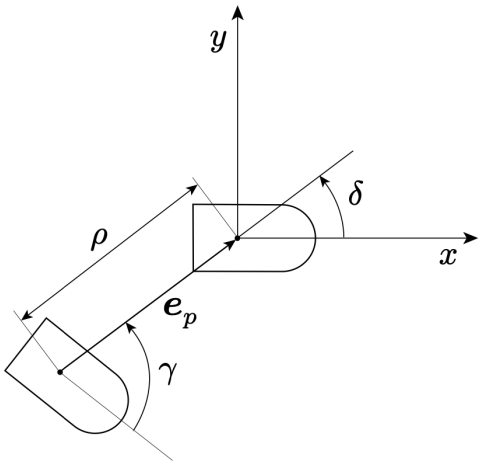


Figure 3.3. Definition of polar coordinates for Unicycle [14]

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

$$\dot{\rho} = -vcos\gamma \tag{3.31}$$

$$\dot{\gamma} = sin\gamma v - \omega \tag{3.32}$$

$$\dot{\delta} = \frac{sin\gamma}{\rho} v \tag{3.33}$$

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

$$v = k_1 \rho cos\gamma \tag{3.34}$$

$$\omega = k_2\gamma + k_1 \frac{sin\gamma cos\gamma}{\gamma}(\gamma + k_3\delta) \tag{3.35}$$

$k_1, k_2 > 0$ 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.

$$V = \frac{1}{2}(\rho^2 + \gamma^2 + k_3\delta^2) \tag{3.36}$$

$$\dot{V} = -k_1 cos^2(\gamma)\rho^2 - k_2\gamma^2 \tag{3.37}$$

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 $\dot{V}$ in uniformly continuous.

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

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].

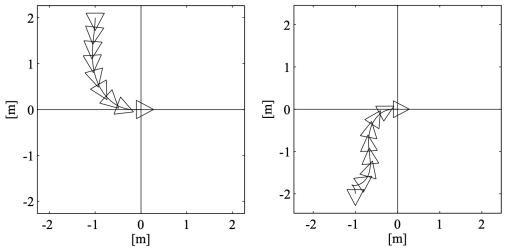


Figure 3.4. Posture regulation simulation [14]

Chapter 4

Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) stems from [**?**], 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 awaken 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 contonuos 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:

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

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 roves to be sufficient to let the agent explore unknown portions of the state space. This procedure take the name of "$\epsilon - greedy$ 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 (<S,A,P,R,γ>). 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". A Markov process is a tuple <S,P> where S is the state and P is the state transition function:

$$P_{ss'} = P(S_{t+1} = s' | S_t = s) \tag{4.2}$$

In order to obtain a Markov reward process <S,P,R,γ> the reward function must be defined as follows:

$$R = E[R_{t+1} | S_t = s] \qquad \text{(reward function)} \tag{4.3}$$

$$\gamma \in [0, 1] \qquad \text{(discount factor)} \tag{4.4}$$

If a set of actions A is introduced, then we obtain a Markov Decision Process (<S,A,P,R,γ>).

$$S = \text{finite set of states} \tag{4.5}$$

$$A = \text{finite set of actions} \tag{4.6}$$

$$P^a_{ss'} = P(S_{t+1} = s' | S_t = s, A_t = a) \tag{4.7}$$

$$R^a_s = E[R_{t+1} | S_t = s, A_t = a] \tag{4.8}$$

$$\gamma \in [0, 1] \tag{4.9}$$

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

$$\pi(a|s) = P(A_t = a|S_t = s) \tag{4.10}$$

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:

$$V_\pi(s) = E_\pi[F_t|S_t = s] \tag{4.11}$$
$$Q_\pi(s) = E_\pi[F_t|S_t = s, A_t = a] \tag{4.12}$$

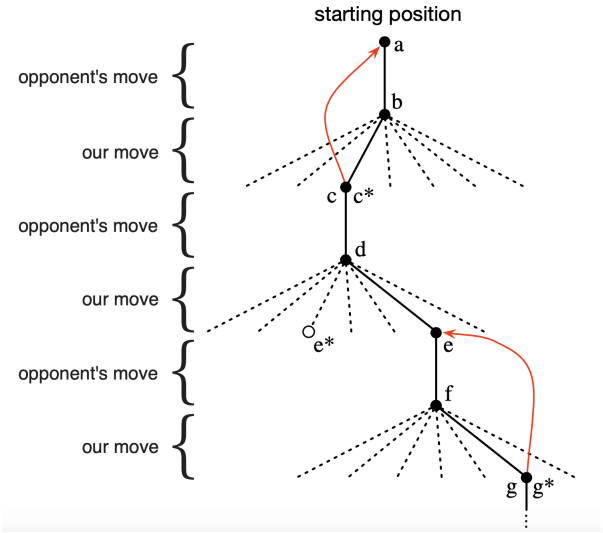


Figure 4.1. Example of a MDP in the case of a Chess Game. Taken from [16]

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

$$F_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4.13}$$

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:

$$V_\pi(s) = \pi(a|s) \sum_{a \in A} (R_s^a + \sum_{s' \in S} P_{ss'}^a v_\pi(s)) \tag{4.14}$$
$$Q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a \in A} \pi(s'|a') q_\pi(s', a') \tag{4.15}$$

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:

$$V_*(s) = \max_{a \in A} Q_*(s, a) \tag{4.16}$$

$$Q_*(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V^*(s) \tag{4.17}$$

$$a^* = \arg\max_{a \in A} Q(s, a) \tag{4.18}$$

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 anc 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.

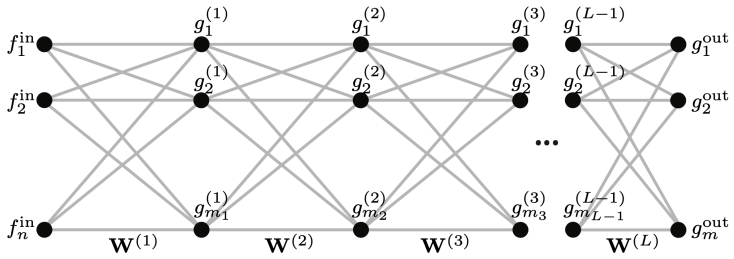


Figure 4.2. Deep Neural Network

$$L(\theta, \mathbf{x}) = \sum_{i=0}^{n} (y_i - f(x_i, \theta)^2 + \lambda\theta^2 \tag{4.19}$$

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\mathbf{x}, \theta) \tag{4.20}$$

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 the refine the policy once it has been generally defined. Some common schedules could be linear and exponential:

$$\eta^t = \Big(1 - \frac{t}{r}\Big)\eta^0 + \frac{t}{r}\eta^{final}$$

$$\eta^t = \frac{1}{1 + rt}\eta^0$$

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:

$$\theta \leftarrow \theta - \eta \nabla_\theta \Big(\sum_{i=0}^{B} (y_i - f(x_i, \theta)^2 + \lambda\theta^2 \Big) \tag{4.21}$$

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

$$x^{(1)} = x^{(0)} - \alpha \nabla f(x^{(0)})$$

$$x^{(2)} = x^{(1)} - \alpha \nabla f(x^{(1)}) = x^{(0)} - \alpha \nabla f(x^{(0)}) - \alpha \nabla f(x^{(1)})$$

$$\dots$$

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

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.

$$v^{(t+1)} = \lambda v^{(t)} - \alpha\nabla f(x^{(t)}) \tag{4.22}$$

$$x^{(t+1)} = x^{t0)} + v^{(t+1)} \tag{4.23}$$

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 algorith, is better than DQN in Beamrider, Breakout, Pong, Q*bert, Space invaders. This 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 $s_0, u_0, \ldots, s_H, u_H$. 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:

$$U(\theta) = \mathrm{E}\Big[\sum_{t=0}^{H} R(s_t, u_t)|\pi_\theta\Big] \tag{4.24}$$

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:

$$\nabla_\theta U(\theta) = \mathrm{E}\Big[\nabla_\theta \log P(\tau,\theta)R(\tau)\Big] \tag{4.25}$$

and can be approximated by the empirical estimate for m sample paths under policy π_θ:

$$\nabla_\theta U(\theta) \approx \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)},\theta)R(\tau^{(i)}) \tag{4.26}$$

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.

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

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].

$$\nabla_\theta U(\theta) \approx \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta \log P(\tau^{(i)}, \theta)(R(\tau^{(i)}) - b) \tag{4.27}$$

Indeed the contribution to the expected value is:

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

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:

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

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:

$$L = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H-1}\left(V_{\Phi}^{\pi}(s_t^{(i)}) - (\sum_{k=t}^{H-1} R(s_k^{(i)}, u_k^{(i)}))\right)^2 \tag{4.28}$$

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:

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

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 CITATION, 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 $Q^{\pi,\gamma,k}(s,u)$:

$$Q^{\pi,\gamma,k}(s,u) = [r_0 + \gamma r_1 + \cdots + \gamma^k V^{\pi}(s_k)|s = s_0, u = u_0] \tag{4.29}$$

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** $iteration = 1, 2, \ldots, M$ **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**

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 is 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 con 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 succesful 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 wuth 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 colled 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 his method is that it is a Second Order method, and it must approximated in some way.

$$\max_{\theta} L(\theta) = \max_{\theta} \Big[\mathrm{E}_{\pi^{old}} \Big[\frac{\pi_\theta(a_t|s_t)}{\pi_\theta^{old}(a_t|s_t)} \hat{A}_t \Big] \Big] \tag{4.30}$$

$$\text{Subject to:} \quad \mathrm{E}_{\pi^{old}} \Big[KL(\pi_\theta^{old}(a_t|s_t), \pi_\theta(a_t|s_t)) \Big] < \epsilon \tag{4.31}$$

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 brough directly inside the Objective function which needs to be maximized.

$$\max_{\theta} J(\theta) = \max_{\theta} \Bigg[L(\theta) - \beta \mathrm{E}_{\pi^{old}} \Big[KL(\pi_\theta^{old}(a_t|s_t), \pi_\theta(a_t|s_t)) \Big] \Bigg] \tag{4.32}$$

$$\tag{4.33}$$

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

$$KL(p|q)(x) = \sum_x p \; log(\frac{q}{p}) \tag{4.34}$$

$$\tag{4.35}$$

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

$$J(\theta) \approx g(\theta - \theta^{old}) - \beta(\theta - \theta^{old})^T F(\theta - \theta^{old}) \tag{4.36}$$

$$g = \frac{\partial L}{\partial \theta}|_{\theta^{old}} \tag{4.37}$$

$$F = \frac{\partial^2}{\partial \theta^2}\left[KL(\pi_\theta^{old}(a_t|s_t), \pi_\theta(a_t|s_t))|_{\theta_{old}}\right] \tag{4.38}$$

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.

$$\theta - \theta^{old} = \frac{1}{\beta} F^{-1} g \tag{4.39}$$

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:

$$J(\theta) = L(\theta) - \beta \mathrm{E}_{\pi^{old}}\Big[KL(\pi_\theta^{old}(a_t|s_t), \pi_\theta(a_t|s_t))\Big] \tag{4.40}$$

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:

$$r(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_\theta^{old}(a_t|s_t)} \hat{A}_t \tag{4.41}$$

$$L^{CLIP}(\theta) = \mathrm{E}\left[\min_\theta \left(r(\theta), \mathrm{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{4.42}$$

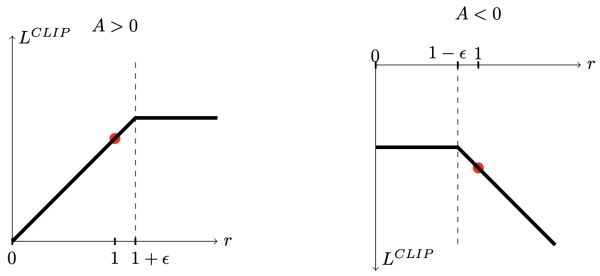


Figure 4.3. Example of the Clipped surrogate loss function [6]

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,$\gamma\rangle$. 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:

$$S = (x, y, \theta, x_1, y_1, x_2, y_2, .., x_4, y_4)^T \tag{5.1}$$

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:

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

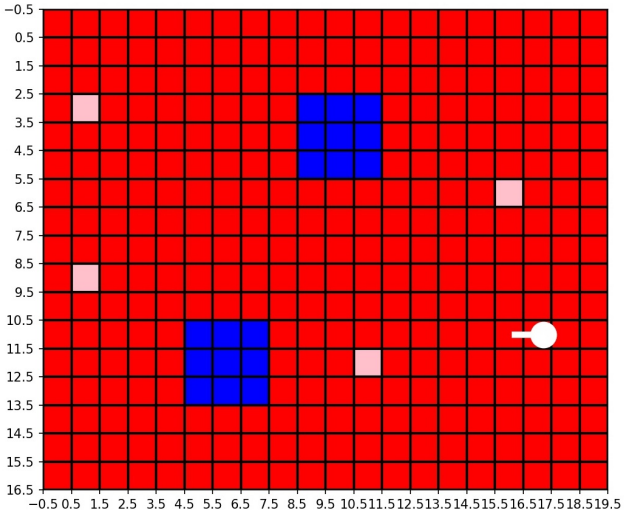


Figure 5.1. Example of a map
Red=empty square, Pink=Via point, Blue=Obstacle

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:

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

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:

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

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.

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

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:

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

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 still need to be visited 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`

```
Receives as input matrix A and initializes vector Map with an empty array
for row = 1, 2, ..., M do
    for column = 1, 2, ..., T do
        if A(row, column) > −2 then
            Map.append(A(row,column))
        end if
    end for
end for
```

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

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

Note that in the previous picture, the square underneath the robot has become blue w.r.t. the same map example regarding structure 1. This is justified by the fact that with this approach the corresponding element inside the matrix map A is equal to -1.

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.

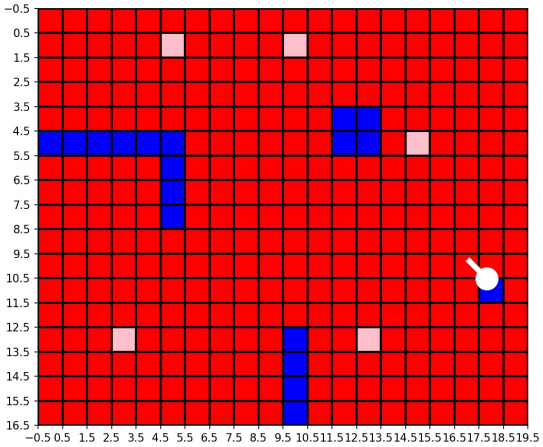


Figure 5.2. Example of a map Red=empty square, Pink=Via point, Blue=Obstacle or Square
occupied by the robot

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 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:

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

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.

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

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 -0.5.

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.

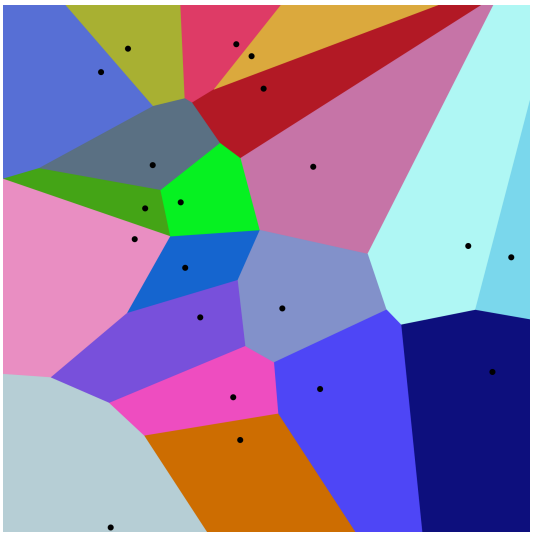


Figure 5.3. Example of Voronoi Tesselation

Chapter 6

Subroutine Control Law

6.1 Local Trajectory generation

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

Algorithm 3 DTSP Algorithm

Find $\{a_1, a_2, \ldots, a_n\}$, the solution to the ETSP problem
Set Φ_1 = orientation of segment from $a_1$to a_2
for i=2,. . . ,n-1 **do**
    if i is Even **then**
        Set $\Phi_i = \Phi_{i-1}$
    else
        Set Φ_i = orientation of segment from a_i to a_{i+1}
    end if
end for
if n is Even **then**
    Set $\Phi_n = \Phi_{n-1}$
else
    Set Φ_n = orientation of segment from a_n to a_1
end if
Return the set composed by $\{(a_i, \Phi_i)_{i=1,..,n}\}$

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:

$$x(s) = -(s-1)^3 x_i + s^3 x_f + \alpha_x s^2(s-1) + \beta_x s(s-1)^2 \tag{6.1}$$
$$y(s) = -(s-1)^3 y_i + s^3 y_f + \alpha_y s^2(s-1) + \beta_y s(s-1)^2 \tag{6.2}$$
$$\tag{6.3}$$

Then boundary conditions to the problem are:

$$x(0) = x_i \ , \ y(0) = y_i \ , \ x(1) = x_f \ \ y(1) = y_f$$
$$\dot{x}(0) = Kcos(\Phi^i) \ , \ \dot{y}(0) = Ksin(\Phi^i) \ , \ \dot{x}(1) = Kcos(\Phi^f) \ , \ \dot{y}(1) = Ksin(\Phi^f)$$

This is achieved by choosing:

$$\begin{pmatrix} \alpha_x \\ \alpha_y \end{pmatrix} = \begin{pmatrix} Kcos(\Phi^f) - 3x_f \\ ksin(\Phi^f) - 3y_f \end{pmatrix} \ , \ \begin{pmatrix} \beta_x \\ \beta_y \end{pmatrix} = \begin{pmatrix} Kcos(\Phi^i) - 3x_i \\ ksin(\Phi^i) - 3y_i \end{pmatrix} \tag{6.4}$$

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:

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

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

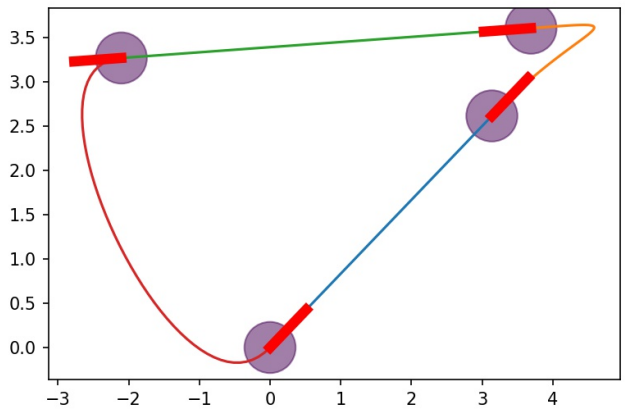


Figure 6.1. Interpolation

Then a timing law must be chosen in order for the bounds on linear and angular velocities to not be violated...
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:

$$\dot{x} = \frac{1}{T}[-3\Big(\frac{t}{T}-1\Big)^2 x_i + 3\Big(\frac{t}{T}\Big)^2 x_f + \alpha_x(3\Big(\frac{t}{T}\Big)^2 - 2\frac{t}{T}) + \beta_x(3\Big(\frac{t}{T}\Big)^2 - 4\frac{t}{T}+1)]$$
$$\dot{y} = \frac{1}{T}[-3\Big(\frac{t}{T}-1\Big)^2 y_i + 3\Big(\frac{t}{T}\Big)^2 y_f + \alpha_x(3\Big(\frac{t}{T}\Big)^2 - 2\frac{t}{T}) + \beta_y(3\Big(\frac{t}{T}\Big)^2 - 4\frac{t}{T}+1)]$$
$$\omega(t) = \frac{\ddot{y}\dot{x} - \ddot{x}\dot{y}}{\dot{x}^2 + \dot{y}^2}$$

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 $T \leftarrow T\gamma$.
Another strategy could be to amplify the istantaneous linear or angular velocity whenever there is still room for reaching admissibility bounds. This is done in order to exploit full power at any time:

$$\begin{cases} \omega(t) \leftarrow \omega(t)\gamma & \text{if } \omega(t) \text{ doesn't violate bounds} \\ v(t) \leftarrow v(t)\gamma & \text{if } \omega(t) \text{ doesn't violate bounds} \end{cases}$$

where $\gamma = \min(\frac{\omega_{bound}}{\omega(t)}, \frac{v_{bound}}{v(t)})$ and by definition $\gamma > 1$

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 ide of the following inequality represents the state dimension for Structure 1 and the right side the dimension for Structure 2: $3 + 2 * N << 3 + 1 + \texttt{size\_x} \times \texttt{size\_y}$, 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.

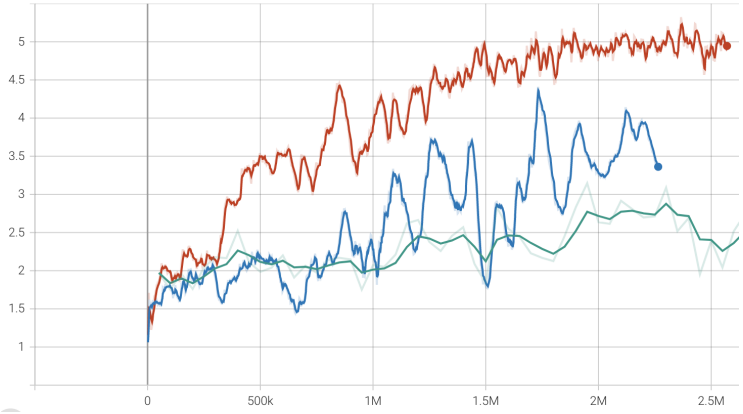


Figure 7.1. Episode Reward for DQN,PPO,A2C Comparison. Red: PPO, Blue: DQN, Green: A2C

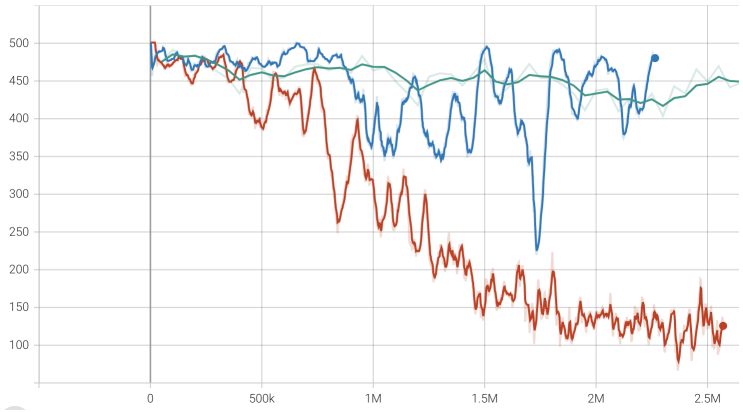
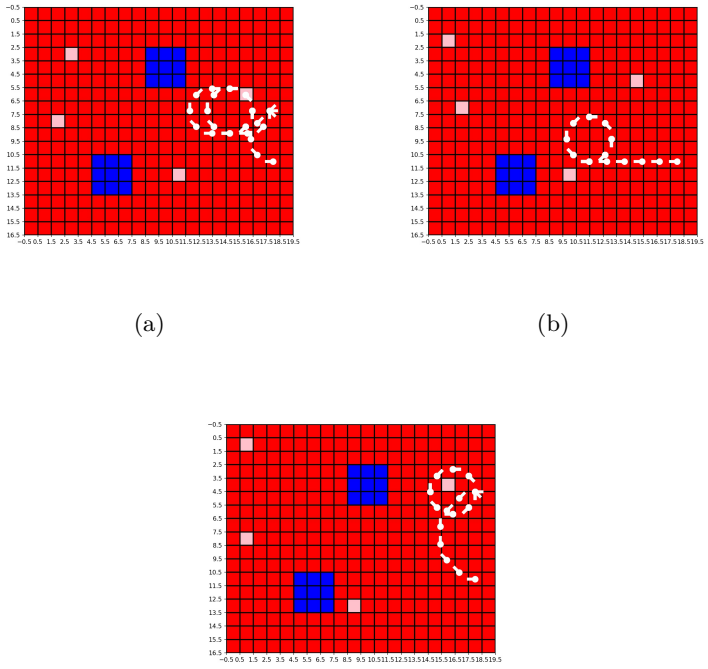


Figure 7.2. Episode length for DQN,PPO,A2C Comparison. Red: PPO, Blue: DQN, Green: A2C

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.



(a) (b)

(c)

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.

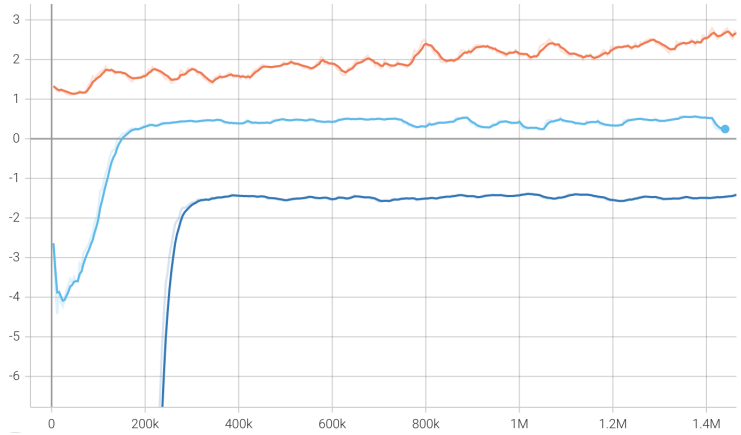


Figure 7.4. Learning deteriorating with negative rewards. Blue: P=-1, Cyan: P=-0.1, Orange: P=-0.01. (Reward)

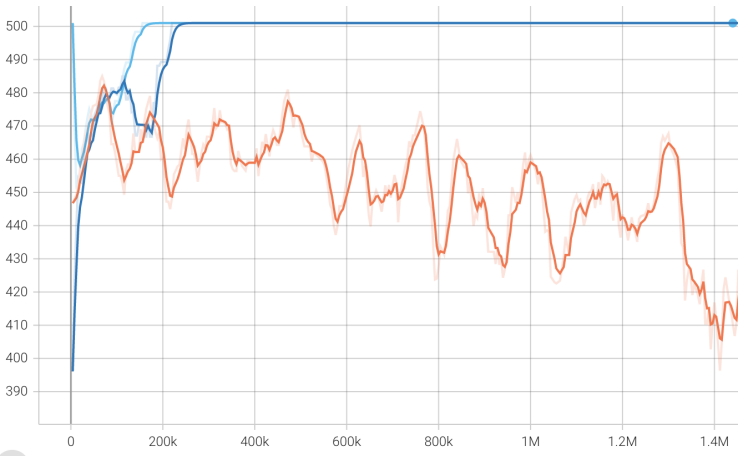


Figure 7.5. Learning deteriorating with negative rewards. Blue: P=-1, Cyan: P=-0.1, Orange: P=-0.01. (Episode length)

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

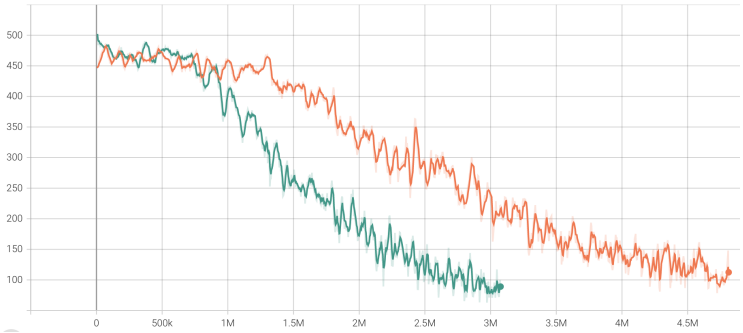


Figure 7.6. Learning rate comparison. Episode Length

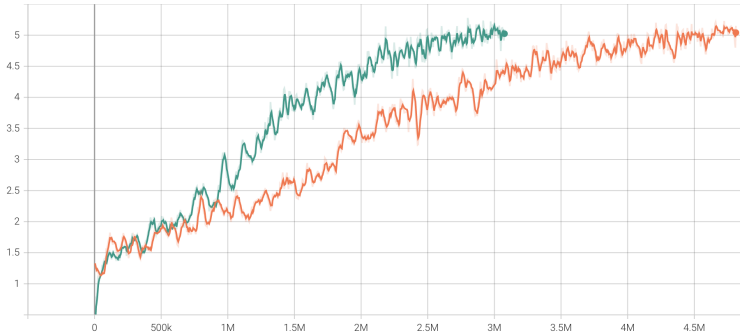


Figure 7.7. Learning rate comparison. Episode Length

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:

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

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 still need to be visited 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 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`	$4 \div 5$

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

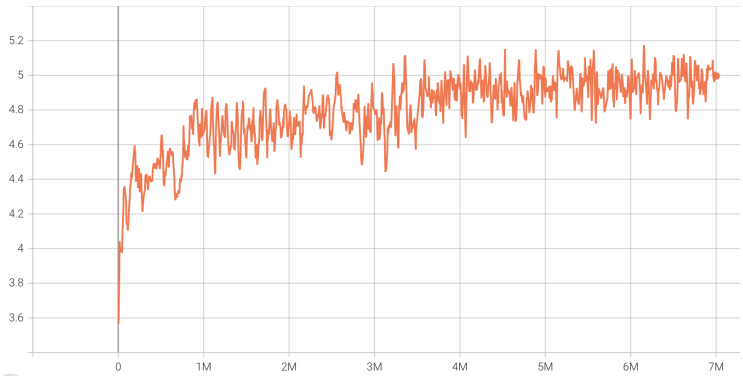


Figure 7.8. Episode Reward on map of size 17×20

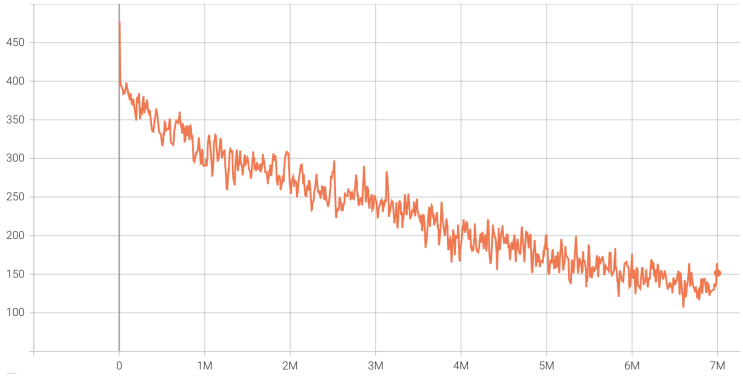


Figure 7.9. Episode length on map of size 17×20

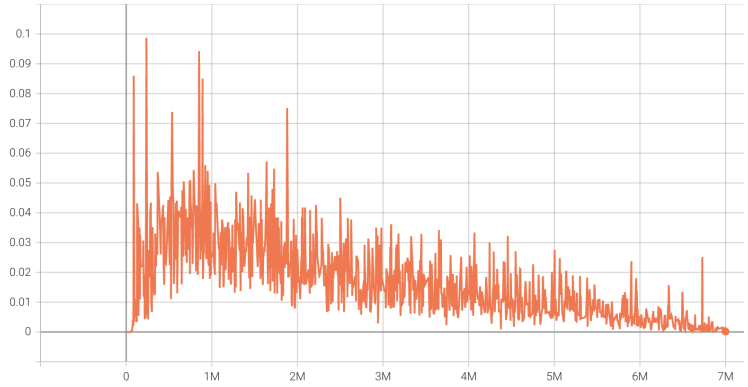
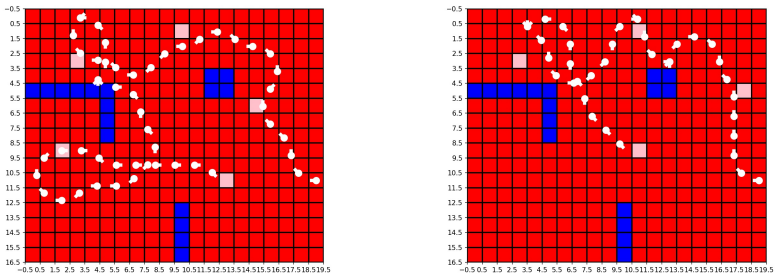


Figure 7.10. Clip fraction on map of size 17×20

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



(a) (b)

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.

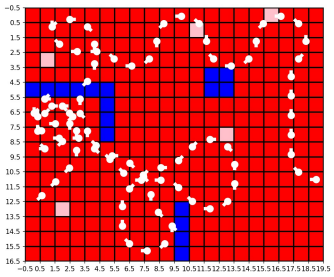


Figure 7.12. The agent found a flaw in the environment and exploited it to win the game

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 bust 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:

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

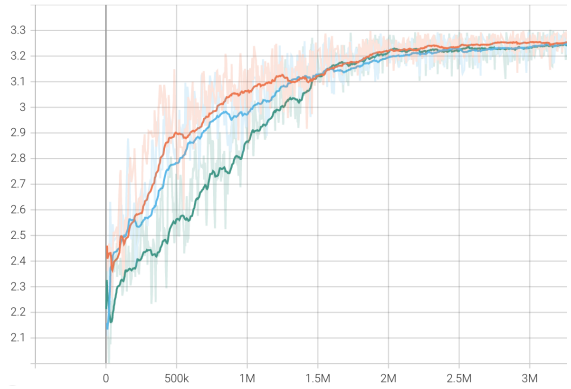
Where $m = 11$ represents the size of the state of the system if considering four goal states. In addition $o = 3$ represents the number of possible actions that the network can take.

N	Parameters
50	5853
200	83403
300	185103

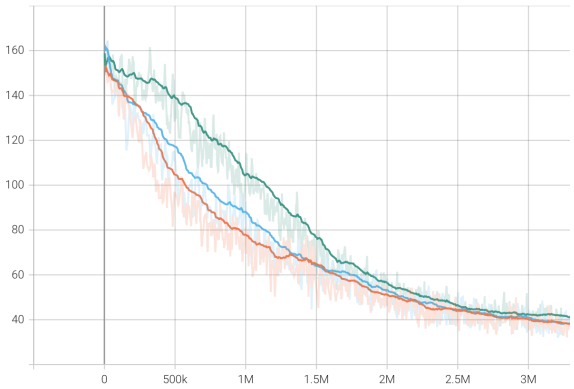
Table 7.8. Number of parameters of each network

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

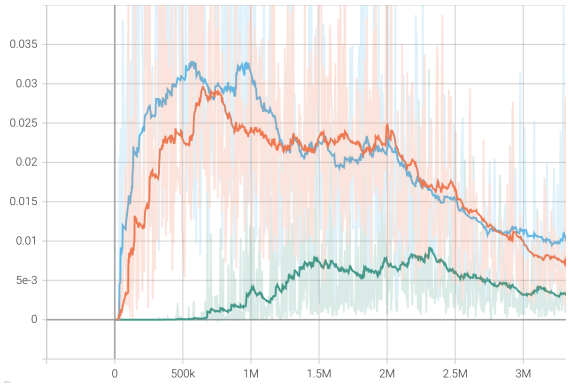
all simulations, it appears that for wider networks the learning is faster during the first phase, but the overall convergence to optimality is reached simultaneously. It appears that the less wide network is able to catch up with the others.



(a) Episode reward

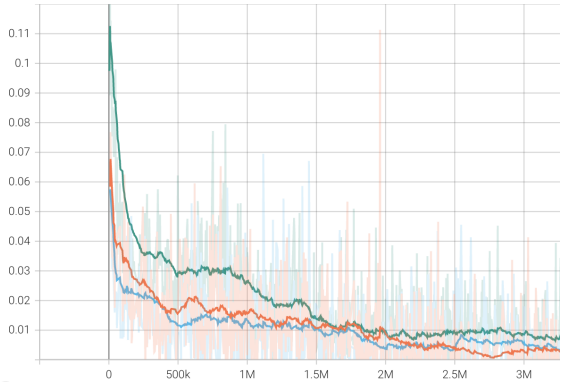


(b) Episode length

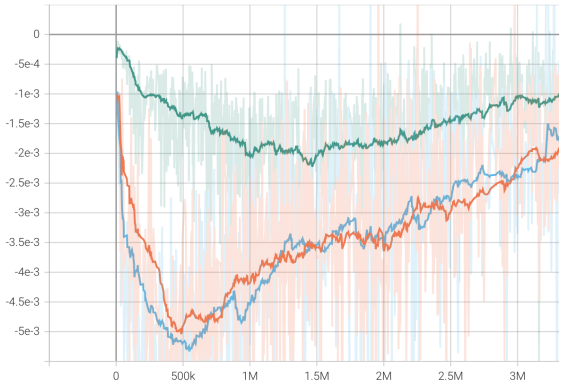


(c) Clip fraction

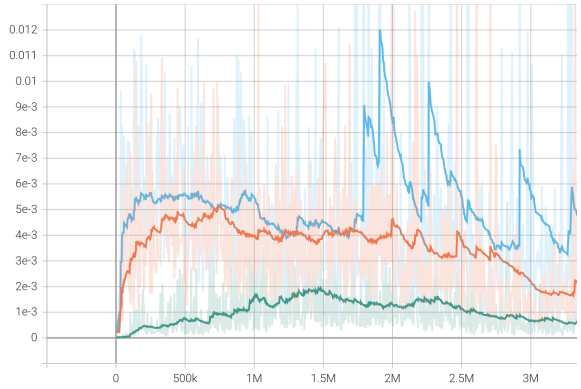
Figure 7.13. Neural Network Architecture Comparison, part 1 | Green: (50,50,50), Orange:(200,200,200),Cyan:(300,300,300)



(a) Value Loss

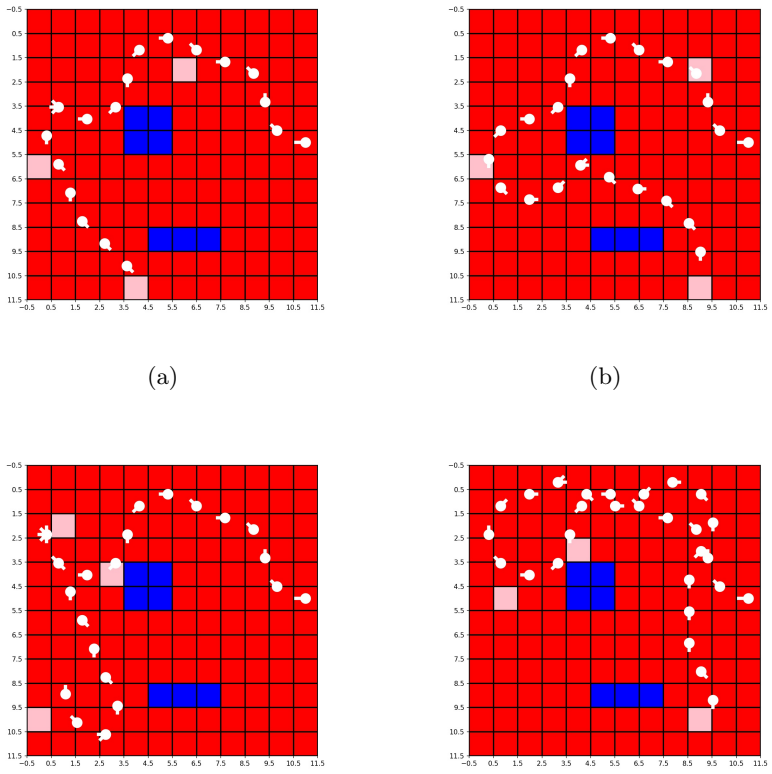


(b) Policy Loss



(c) Kullback-Lieber Approximate Divergence

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



(a) (b)

(c) (d)

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

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.

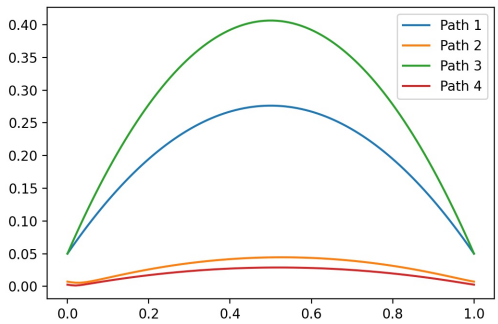


Figure 7.16. Subroutine test 1:linear velocity

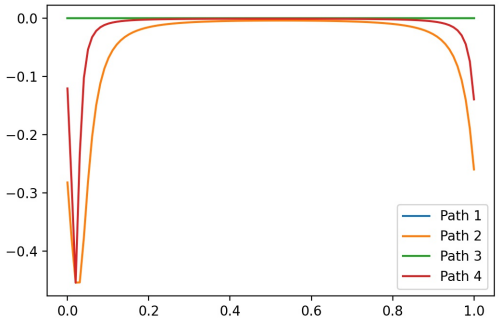


Figure 7.17. Subroutine test 1:angular velocity

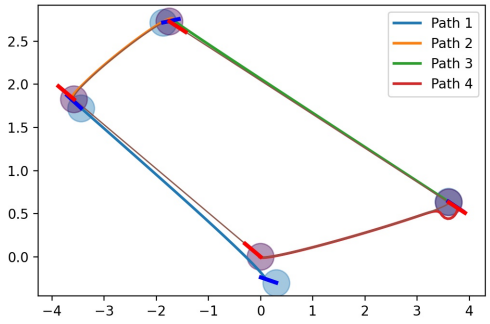


Figure 7.18. Subroutine test 1:Tracking

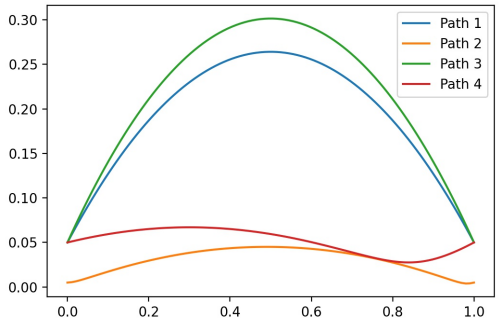


Figure 7.19. Subroutine test 2:linear velocity

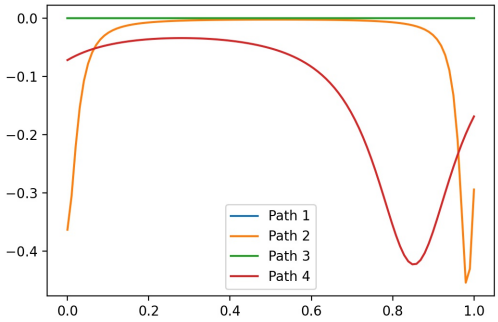


Figure 7.20. Subroutine test 2:angular velocity

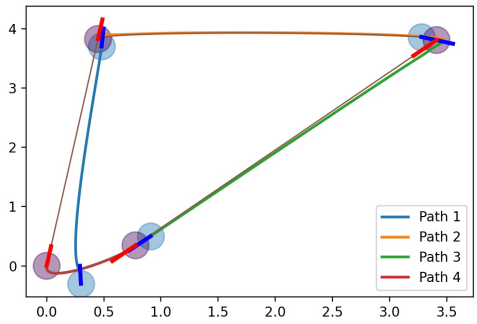


Figure 7.21. Subroutine test 2:Tracking

Chapter 8

Conclusions and Future Works

8.1 Future works

Section 7 has shown that this approach is able to effective find a solution for the Multi-point path planning problem, but is it evident that there is still plenty of space for further improvements.
First and foremost, 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 or 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.
Finally, the DRL environment could get extremely accurate and include also local information about litter around the map.

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 a valid alternative to many classical path planning algorithms. 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

```
!pip install stable-baselines3[extra]
import stable_baselines3

import gym
import numpy as np
from gym import Env
from gym.spaces import Discrete, Box
import random
import matplotlib.pyplot as plt
from matplotlib import colors
import time
import copy
import torch as th

from stable_baselines3 import PPO
from stable_baselines3.ppo import MlpPolicy

#---------------- Define a new cell in Google Colab

gain=1
penalty=0.0
size_x=12
size_y=12
mosse_max=200

flag_robot_su_griglia=-1
v_mid=0.5
omega_max=0.3
delta_t=(2*np.pi/(8*omega_max))
action_list=[ [v_mid,-omega_max], [v_mid,omega_max],[v_mid,0], ]

```

```
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
    robot
127         self.reward_grid[int(np.round((size_x-1)*self.state[0])),int(np.
    round((size_y-1)*self.state[1]))]=0
128
129         self.stato_memoria=[self.state[0],self.state[1]]
130
131         if self.state[0]<=1 and self.state[0]>=0 and self.state[1]<=1 and
    self.state[1]>=0:
132             angolo=self.state[2]*np.pi*2
133             self.state[0]+=delta_t*np.cos(angolo)*action_list[action][0]/(
    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.stato_memoria
145             self.reward_grid[int(np.round(self.state[0]*(size_x-1))),int(np.
    round(self.state[1]*(size_y-1)))]=flag_robot_su_griglia
146             self.state[4:size+4] = vectorize_map(self.reward_grid)
147 
148         if self.state[0]<=0:
149             reward=-penalty
150             self.state[0:2]=self.stato_memoria
151             self.reward_grid[int(np.round(self.state[0]*(size_x-1))),int(np.
    round(self.state[1]*(size_y-1)))]=flag_robot_su_griglia
152             self.state[4:size+4] = vectorize_map(self.reward_grid)
153 
154         if self.state[1]>=1:
155             reward=-penalty
156             self.state[0:2]=self.stato_memoria
157             self.reward_grid[int(np.round(self.state[0]*(size_x-1))),int(np.
    round(self.state[1]*(size_y-1)))]=flag_robot_su_griglia
158             self.state[4:size+4] = vectorize_map(self.reward_grid)
159             self.state[0:2]=self.stato_memoria
160 
161         if self.state[1]<=0:
162             reward=-penalty
163             self.state[0:2]=self.stato_memoria
164             self.reward_grid[int(np.round(self.state[0]*(size_x-1))),int(np.
    round(self.state[1]*(size_y-1)))]=flag_robot_su_griglia
165             self.state[4:size+4] = vectorize_map(self.reward_grid)
166 
167 
168         if self.reward_grid[int(np.round((size_x-1)*self.state[0])),int(np.
    round((size_y-1)*self.state[1]))]<-1.6:
169             reward=-penalty
170             self.state[0:2]=self.stato_memoria
171             self.reward_grid[int(np.round(self.state[0]*(size_x-1))),int(np.
    round(self.state[1]*(size_y-1)))]=flag_robot_su_griglia
172             self.state[4:size+4] = vectorize_map(self.reward_grid)
173 
174         pos_x=int(np.round(self.state[0]*(size_x-1)))
175         pos_y=int(np.round(self.state[1]*(size_y-1)))
176 
177         reward=ricompensa(self.reward_grid[np.max([0,pos_x-1]):np.min([
    pos_x+2,19]),np.max([pos_y-1,0]):np.min([pos_y+2,19])].reshape(-1))
178 
```

```
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",
        markerfacecolor="white")
245 #-----------------Define a new cell to test the environment and see if it
        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         plottagrid(env.pixel_grid.reshape(size_x,size_y))
257         print('posizione',[env.state[0]*size_x,env.state[1]*size_y,env.
        state[2]*np.pi*2,env.state[3]],'reward',reward,'azione',action_list[
        action])
258 
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,
    200,200], vf=[200, 200,200])])
343 
344 policy_kwargs_DQN = dict(activation_fn=th.nn.Tanh,
345                          net_arch=[200, 200,200])
346 Model_type='DQN' #DQN,PPO,A2C
347 
348 
349 if Model_type=='DQN':
350     model = stable_baselines3.dqn.DQN("MlpPolicy", env,
351                                       learning_rate=
    learning_rate_custom_DQN(),policy_kwargs=policy_kwargs_DQN,
352                                       batch_size=1600, verbose=0,
    exploration_initial_eps=1,
353                                       exploration_fraction=0.1,train_freq
    =(1, "episode"),max_grad_norm =2,gradient_steps=2,
354                                       tensorboard_log="./pixies_tensorboard
    /")
355 if Model_type=='PPO':
356     model = PPO(MlpPolicy, env,policy_kwargs=policy_kwargs, ent_coef=0.0,
    gamma=0.995,
357                                       learning_rate=
    learning_rate_custom_PPO(),n_epochs=4,
358                                       batch_size=400,verbose=0,n_steps
    =2000,
359                                       tensorboard_log="./pixies_tensorboard
    /")
360 
361 if Model_type=='A2C':
362     model = stable_baselines3.a2c.A2C("MlpPolicy",env, learning_rate
    =0.0007,
363  n_steps=500, gamma=0.99, gae_lambda=1.0, ent_coef=0.0, vf_coef=0.5,
364   max_grad_norm=0.5, rms_prop_eps=1e-05,policy_kwargs=policy_kwargs,
      tensorboard_log="./pixies_tensorboard/")
365 
366 
367 # PPO documentation on https://stable-baselines3.readthedocs.io/en/master/
    modules/ppo.html
368 
369 
370 
371 #-------------Cell to load the tensordboard
372 %load_ext tensorboard
```

```
%tensorboard --logdir ./pixies_tensorboard/

#-------------Cell to train the model
model.learn(total_timesteps=5000000)

#------------- Print some successful trajectories
episodes = 5

for episode in range(1, episodes+1):
    done = False
    obs = env.reset()
    score = 0
    plottagrid(env.reward_grid)
    for i in range(100):

        if done==True:
          break
        action, _states = model.predict(obs)
        n_state, reward, done, info = env.step(action)

        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')
        plt.plot(env.state[1]*(size_y-1), env.state[0]*(size_x-1), marker="o", markersize=15, markeredgecolor="white", markerfacecolor="white")
```

Listing 9.1. Python example

9.2 Subroutine Code

```
N=4

graph=np.zeros((N,N))
X=np.random.uniform(-4,4,N-1)
X=np.concatenate((np.zeros(1),X))
Y=np.random.uniform(0,4,N-1)
Y=np.concatenate((np.zeros(1),Y))
colors = [0 for i in range(N)]
area = (30 * 1)**2  # 0 to 15 point radii
fig, ax = plt.subplots()
plt.scatter(X, Y, s=area, c=colors, alpha=0.5)
plt.xlim([-6, 6])
plt.ylim([-2, 6])
n = range(N)
for i, txt in enumerate(n):
    ax.annotate(txt, (X[i], Y[i]))

for i in range(1,N):
    for j in range(0,i):
            graph[i][j]=(((X[i])-X[j])**2+(Y[i]-Y[j])**2)**(0.5)
dist=graph+graph.T
```

```
# Helper function to calculate path length
def path_len(path):
    return sum(dist[i][j] for i, j in zip(path, path[1:]))

# Set of all nodes to visit
to_visit = set(range(len(dist)))

# Current state {(node, visited_nodes): shortest_path}
state = {(i, frozenset([0, i])): [0, i] for i in range(1, len(dist[0]))}

for _ in range(len(dist) - 2):
    next_state = {}
    for position, path in state.items():
        current_node, visited = position

        # Check all nodes that haven't been visited so far
        for node in to_visit - visited:
            new_path = path + [node]
            new_pos = (node, frozenset(new_path))

            # Update if (current node, visited) is not in next state or we found shorter path
            if new_pos not in next_state or path_len(new_path) < path_len(next_state[new_pos]):
                next_state[new_pos] = new_path

    state = next_state

# Find the shortest path from possible candidates
shortest = min((path + [0] for path in state.values()), key=path_len)
print('path: {0}, length: {1}'.format(shortest, path_len(shortest)))
theta_=[0 for i in range(len(shortest))]

for i in range(len(shortest)-1):
    if i%2==0:
        vecio=shortest[i]
        nuovo=shortest[i+1]
        norm=((X[nuovo]-X[vecio])**2+(Y[nuovo]-Y[vecio])**2)**(0.5)
        theta_[i:i+1]=[math.atan2(Y[nuovo]-Y[vecio],X[nuovo]-X[vecio]),math.atan2(Y[nuovo]-Y[vecio],X[nuovo]-X[vecio])]
        plt.plot([X[vecio], X[nuovo]],[Y[vecio], Y[nuovo]])
        plt.plot([X[vecio], X[vecio]+(X[nuovo]-X[vecio])/norm], [Y[vecio], Y[vecio]+(Y[nuovo]-Y[vecio])/norm], linewidth=2,color='red')
        plt.plot([X[nuovo], X[nuovo]+(X[nuovo]-X[vecio])/norm], [Y[nuovo], Y[nuovo]+(Y[nuovo]-Y[vecio])/norm], linewidth=2,color='red')
theta_=theta_[0:6]
```

```
fig0,ax0=plt.subplots()
fig1,ax1=plt.subplots()
fig2,ax2=plt.subplots()
def interpolate(x_i,y_i,tet_i,x_f,y_f,tet_f,T,discretizations,jjj):
    factor=1
    s=np.linspace(0,1,discretizations)# sarebbe il rapporto tra t/T
    i=0
    K=1
    alpha_x=K*np.cos(tet_f)-3*x_f
    alpha_y=K*np.sin(tet_f)-3*y_f

    beta_x=K*np.cos(tet_i)+3*x_i
    beta_y=K*np.sin(tet_i)+3*y_i

    x=-(s-1)**3*x_i+s**3*x_f+alpha_x*s**2*(s-1)+beta_x*s*(s-1)**2
    y=-(s-1)**3*y_i+s**3*y_f+alpha_y*s**2*(s-1)+beta_y*s*(s-1)**2
    xdot=(-3*(s-1)**2*x_i+3*s**2*x_f+  alpha_x*(3*s**2-2*s)+beta_x*(3*s
    **2-4*s+1))/T
    ydot=(-3*(s-1)**2*y_i+3*s**2*y_f+  alpha_y*(3*s**2-2*s)+beta_y*(3*s
    **2-4*s+1))/T

    xdotdot=(-6*(s-1)*x_i+6*s*x_f+  alpha_x*(6*s-2)+beta_x*(6*s-4))/T**2

    ydotdot=(-6*(s-1)*y_i+6*s*y_f+  alpha_y*(6*s-2)+beta_y*(6*s-4))/T**2

    velocity=(xdot**2+ydot**2)**(0.5)
    omega=(ydotdot*xdot-xdotdot*ydot)/(xdot**2+ydot**2)

    omega=np.nan_to_num(omega)
    #print(omega)
    print('test',np.max(velocity),np.max(omega),np.min(omega))

    while  np.max(velocity)>0.51 or np.max(abs(omega))>0.51:
        i+=1

        factor=np.max([np.max(velocity),np.max(abs(omega))])*(1.1)/0.5
        T=factor*T
        print(factor)
        s=np.linspace(0,1,discretizations)# sarebbe il rapporto tra t/T
        i=0
        K=0.5
        alpha_x=K*np.cos(tet_f)-3*x_f
        alpha_y=K*np.sin(tet_f)-3*y_f

        beta_x=K*np.cos(tet_i)+3*x_i
        beta_y=K*np.sin(tet_i)+3*y_i

        x=-(s-1)**3*x_i+s**3*x_f+alpha_x*s**2*(s-1)+beta_x*s*(s-1)**2
        y=-(s-1)**3*y_i+s**3*y_f+alpha_y*s**2*(s-1)+beta_y*s*(s-1)**2
        xdot=(-3*(s-1)**2*x_i+3*s**2*x_f+  alpha_x*(3*s**2-2*s)+beta_x*(3*s
    **2-4*s+1))/T
        ydot=(-3*(s-1)**2*y_i+3*s**2*y_f+  alpha_y*(3*s**2-2*s)+beta_y*(3*s
    **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
    **2
121
122         ydotdot=(-6*(s-1)*y_i+6*s*y_f+  alpha_y*(6*s-2)+beta_y*(6*s-4))/T
    **2
123         velocity=(xdot**2+ydot**2)**(0.5)
124         omega=(ydotdot*xdot-xdotdot*ydot)/(xdot**2+ydot**2)
125         #ax0.plot(s,velocity,label='Velocity_'+str(jjj))
126         #ax2.plot(s,omega,label='Omega_'+str(jjj))
127         omega=np.nan_to_num(omega)
128         #return [list(x),list(y),list(xdot),list(ydot)],T*factor
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
136 xxx=[]#lista completa della traiettoria
137 yyy=[]#lista completa della traiettoria
138
139 xxx_dot=[]
140 yyy_dot=[]
141
142
143
144
145 T=20
146 discretizations=100
147
148 theta_[len(shortest)-1]=theta_[0]
149 delta_ttt=np.zeros(10)
150 for i in range(len(shortest)-1):
151     #print(i,'theta',theta_[i+1])
152     uscita,ttt_=interpolate(X[shortest[i]],Y[shortest[i]],theta_[i],X[
    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
165 xxx=np.reshape(xxx,-1)
166 yyy=np.reshape(yyy,-1)
167
168 xxx_dot=np.reshape(xxx_dot,-1)
```

```
yyy_dot=np.reshape(yyy_dot,-1)


b=0.2
track_x=[0.3]
track_y=[-0.3]
track_th=[theta_[0]*1.1]
gamma=0.1


print(len(xxx_dot),len(track_x),len(track_th),len(xxx))
for j in range(4):
    for i in range(discretizations):
        if i==0 and j==0:
            continue
        i=discretizations*j+i
        #print(i)
        #print(len(xxx_dot),len(track_x),len(track_th),len(xxx))
        delta_t=delta_ttt[j]
        desired=np.array([[xxx_dot[i-1]-gamma*(track_x[i-1]+b*np.cos(
    track_th[i-1])-xxx[i-1])],
                          [yyy_dot[i-1]-gamma*(track_y[i-1]+b*np.sin(track_th[i
    -1])-yyy[i-1])]])
        A=np.linalg.inv(np.array([[np.cos(track_th[i-1]),-b*np.sin(track_th
    [i-1])],[np.sin(track_th[i-1]),b*np.cos(track_th[i-1])]]))
        vec=A.dot(desired)
        v=np.clip(vec[0],0,0.5)
        omega=np.clip(vec[1],-0.5,0.5)
        track_x=np.append(track_x,track_x[i-1]+delta_t*np.cos(track_th[i
    -1])*v)
        track_y=np.append(track_y,track_y[i-1]+delta_t*np.sin(track_th[i
    -1])*v)
        track_th=np.append(track_th,track_th[i-1]+delta_t*omega)

        track_x=np.reshape(track_x,-1)
        track_y=np.reshape(track_y,-1)
        track_th=np.reshape(track_th,-1)

ax1.scatter(track_x[0], track_y[0],c='#1f77b4', s=area, alpha=0.4)
ax1.scatter(track_x[discretizations], track_y[discretizations],c='#1f77b4',
     s=area, alpha=0.4)
ax1.scatter(track_x[2*discretizations], track_y[2*discretizations], c='#1
    f77b4',s=area, alpha=0.4)
ax1.scatter(track_x[3*discretizations], track_y[3*discretizations], c='#1
    f77b4',s=area, alpha=0.4)

ax1.plot(np.reshape(track_x+b*np.cos(track_th),-1)[0:discretizations],np.
    reshape(track_y+b*np.sin(track_th),-1)[0:discretizations],linewidth=2,
    color='C0')
ax1.plot(np.reshape(track_x+b*np.cos(track_th),-1)[discretizations:2*
    discretizations],np.reshape(track_y+b*np.sin(track_th),-1)[
    discretizations:2*discretizations],linewidth=2,color='C1')
```

```
209 ax1.plot(np.reshape(track_x+b*np.cos(track_th),-1)[2*discretizations:3*
        discretizations],np.reshape(track_y+b*np.sin(track_th),-1)[2*
        discretizations:3*discretizations],linewidth=2,color='C2')
210 ax1.plot(np.reshape(track_x+b*np.cos(track_th),-1)[3*discretizations:],np.
        reshape(track_y+b*np.sin(track_th),-1)[3*discretizations:],linewidth=2,
        color='C3')
211 ax1.plot(xxx,yyy,linewidth=1,color='C5')
212
213
214 area = (20 * 1)**2
215 b=b*1.5
216 ax1.plot([track_x[0], track_x[0]+b*np.cos(track_th[0])], [track_y[0],
        track_y[0]+b*np.sin(track_th[0])], linewidth=3,color='blue')
217 ax1.plot([track_x[discretizations], track_x[discretizations]+b*np.cos(
        track_th[discretizations])], [track_y[discretizations], track_y[
        discretizations]+b*np.sin(track_th[discretizations])], linewidth=3,
        color='blue')
218 ax1.plot([track_x[2*discretizations], track_x[2*discretizations]+b*np.cos(
        track_th[2*discretizations])], [track_y[2*discretizations], track_y[2*
        discretizations]+b*np.sin(track_th[2*discretizations])], linewidth=3,
        color='blue')
219 ax1.plot([track_x[3*discretizations], track_x[3*discretizations]+b*np.cos(
        track_th[3*discretizations])], [track_y[3*discretizations], track_y[3*
        discretizations]+b*np.sin(track_th[3*discretizations])], linewidth=3,
        color='blue')
220
221 for i in range(len(shortest)-1):
222     if i%2==0:
223         vecio=shortest[i]
224         nuovo=shortest[i+1]
225         norm=((X[nuovo]-X[vecio])**2+(Y[nuovo]-Y[vecio])**2)**(0.5)*3
226         ax1.plot([X[vecio], X[vecio]+(X[nuovo]-X[vecio])/norm], [Y[vecio],
    Y[vecio]+(Y[nuovo]-Y[vecio])/norm], linewidth=3,color='red')
227         ax1.plot([X[nuovo], X[nuovo]+(X[nuovo]-X[vecio])/norm], [Y[nuovo],
    Y[nuovo]+(Y[nuovo]-Y[vecio])/norm], linewidth=3,color='red')
228
229
230
231
232 ax1.scatter(X, Y, s=area, c=colors, alpha=0.4)
233
234
235 ax0.legend(['Path 1', 'Path 2','Path 3','Path 4'])
236 ax1.legend(['Path 1', 'Path 2','Path 3','Path 4'])
237 ax2.legend(['Path 1', 'Path 2','Path 3','Path 4'])
238
239 fig0.savefig('ciao0.jpg',dpi=200)
240 fig1.savefig('ciao1.jpg',dpi=200)
241 fig2.savefig('ciao2.jpg',dpi=200)
```

Listing 9.2. Python example

List of Figures

List of Tables

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. Edelhäußer, 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.