

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO



Reinforcement Learning for Robotic Navigation in Obstacle Scattered Environments

Francisco Soares Pinto da Silva Neves

DISSERTATION

MASTER THESIS

Advisor: António Pedro de Aguiar

Co-advisor: Matheus Reis

Co-advisor: Gustavo Andrade

12/03/2021

Reinforcement Learning for Robotic Navigation in Obstacle Scattered Environments

Francisco Soares Pinto da Silva Neves

MASTER THESIS

Abstract

Reinforcement Learning is more and more a key area of research in Artificial Intelligence, since it offers the possibility of creating agents with the capability of autonomously learn complex tasks while interacting with simulated or real environments. Robotic tasks are traditionally addressed by involving advanced mathematics and high-skilled human intervention. Hence, Reinforcement Learning comes as a modern, promising and simpler solution to approach robotics problems, by offering intelligent systems the capability of learning the behavior by themselves, relieving the amount of human intervention. Additionally, Reinforcement Learning is a tool to solve complex tasks typically not solved with traditional methods and human expertise, particularly in tasks such as object detection from images and game-playing.

Scaling Reinforcement Learning principles to robotic tasks is an huge and active area of research with lots of adversities ahead. One of the main challenges of transferring Reinforcement Learning to Robotics is the natural high-dimensionality of the state and action spaces. If the programmer wants to solve a robotic task using tabular Reinforcement Learning, and if the state representation is not chosen properly, the problem rapidly becomes unfeasible to be solved in a reasonable time frame. Or, since the tabular representation may become incredibly huge, it may not converge at all.

Motivated by the above, in this thesis, we propose a successful tabular Q-learning architecture capable of solving a robotic navigation and obstacle avoidance task. We have successfully built a tabular Q-learning agent with an effective state representation that aims at simplifying the problem by creating a simple MDP to represent the task model. Not only we achieved the desired simulated behavior, but also a successful transferability of the learned virtual behavior to the real robotic hardware. Conceptually, our proposed Q-learning architecture showed a clear and interesting capability and adaptability for the task at hand. Clearly, that was only possible by considerably reducing the dimensionality of the problem, which led to a fast and full convergence. The proposed results are illustrated through extensive computer simulations and experimental results.

Resumo

Reinforcement Learning é cada vez mais uma área de relevo em Inteligência Artificial, visto que oferece a possibilidade de criar agentes com a capacidade de, autonomamente, aprender tarefas complexas enquanto interage com ambientes simulados ou reais. As tarefas Robóticas são tradicionalmente abordadas usando matemática avançada e intervenção humana altamente especializada. Consequentemente, Reinforcement Learning aparece como uma moderna, promissora e simples solução para resolver problemas robóticos. Oferece a sistemas inteligentes a capacidade de aprender por eles próprios, aliviando a quantidade de intervenção humana necessária. Adicionalmente, Reinforcement Learning aparece como uma ferramenta para resolver tarefas complexas, não tipicamente resolvidas com métodos tradicionais e experiência humana, particularmente tarefas relacionadas com detecção de objetos a partir de imagens ou jogos virtuais.

Escalar os princípios de Reinforcement Learning para a Robótica é uma gigante e ativa área de investigação com grandes obstáculos à sua frente. Um dos maiores desafios de transferir Reinforcement Learning para a Robótica é a sua natural alta dimensionalidade dos espaços de estado e ação. Se o programador pretende resolver uma tarefa robótica usando Reinforcement Learning tabular, e se a representação de estado não é devidamente escolhida, o problema rapidamente se torna inviável para ser resolvido em razoáveis quantidades de tempo. Ou, se a representação tabular se tornar incrivelmente grande, o problema nem sequer convergir.

Motivados pelo que até aqui foi descrito, nesta tese, nós propomos uma arquitetura de Q-learning bem sucedida, capaz de resolver uma tarefa robótica de navegação e de desvio de obstáculos. Com sucesso, construímos um agente de Q-learning tabular com uma eficaz e bem sucedida representação de estado com o objetivo de simplificar o problema através da criação de um simples Processo de Decisão de Markov capaz de representar o modelo da tarefa. Não apenas conseguimos atingir o comportamento desejado, como também conseguimos transferir esse mesmo comportamento aprendido virtualmente para o hardware robótico real. Conceptualmente, a nossa arquitetura de Q-learning mostrou uma clara e interessante capacidade e adaptabilidade para a tarefa em questão. Claramente, apenas foi possível atingir o resultado pretendido porque se reduziu consideravelmente a dimensionalidade do problema, que, consequentemente, levou a uma rápida e total convergência. Os resultados propostos são ilustrados com recurso a extensivas simulações computacionais e resultados experimentais.

Acknowledgments

First and foremost, I would like to thank my supervisor António Pedro Aguiar for his support throughout this thesis and for the opportunity to work in this amazing field of Reinforcement Learning. Professor António's attentive guidance and knowledge provided the propitious environment for the development of the dissertation.

To Matheus Reis and Gustavo Andrade who have been amazingly helpful throughout the whole work, not only because of their knowledge but because of their friendship and consideration.

I would also like to thank the amazing people at the C2SR-Lab, who always brought and provided kindness and a comfortable working environment.

To my family I would like to thank for everything: for my education, privileges, love, advices, and support.

Francisco Soares Pinto da Silva Neves

“What we want is a machine that can learn from experience”

Alan Turing, 1947

Contents

1	Introduction	1
1.1	Context and Motivation	2
1.2	Objectives and Contributions	4
1.2.1	Task	5
1.2.2	Method	6
1.2.3	Metrics	6
1.2.4	Methodology	8
1.3	Organization of the Dissertation	10
2	Literature review	11
2.1	RL Methods	11
2.1.1	Model-Free	12
2.1.2	Model-Based	23
2.2	RL and Robotics	26
2.2.1	Curse of dimensionality	26
2.2.2	Function-approximation techniques	27
2.2.3	Sim2Real	27
2.2.4	Sample-efficiency	28
2.2.5	Safety	28
2.2.6	R3L	29
3	Reinforcement Learning	31
3.1	RL Framework	31
3.1.1	Agent-Environment Interaction	31
3.1.2	Rewards and Returns	32
3.1.3	Finite Markov Decision Process	33
3.1.4	Policies and Value Functions	35
3.2	Q-learning	42
3.2.1	Model-Free	43
3.2.2	Generalized Policy Iteration	43
3.2.3	Off-Policy Temporal-Difference Control	45
3.2.4	Convergence	50
3.3	DQN	51
3.3.1	Architecture	52
3.3.2	Overcoming the Deadly Triad	53

4	Q-learning for Robotic Navigation	56
4.1	Implementation	56
4.1.1	Architecture	56
4.1.2	Q-table: The "brain"	58
4.1.3	Reward function	67
4.1.4	Algorithm	71
4.2	Results	74
4.2.1	Training	74
4.2.2	Testing	79
5	Conclusions	86
	References	87

List of Figures

1.1	Reinforcement Learning loop. Available at: http://www.incompleteideas.net/book/ebook/node28.html	2
1.2	Illustration of the task under study. The robot needs to learn how to steer in the direction of the goal while avoiding collisions along the way.	5
1.3	The Turtlebot3 Burger	8
1.4	A screenshot from a Gazebo simulated environment where the robot learns the task	9
2.1	Useful taxonomy of the RL methods. Model-Free methods learn a behavior without understanding the environment (no model knowledge), by direct experience. Model-Based methods, could be given a model and learn a behavior inside of it, or they could learn a model first and then learn a behavior inside of its own learned model.	11
2.2	Model-Free types of algorithms. Policy-Based methods estimate actors, also named as policies π . Value-Based Methods estimate critics, also named as action-value functions Q . Actor-Critic methods combine both the former approaches and they estimate actors and critics.	13
2.3	A typical Actor-critic architecture. Available at: http://incompleteideas.net/book/first/ebook/node66.html	20
2.4	Sample-efficiency spectrum in RL; Available at: https://medium.com/@jonathan_hui/rl-model-based-reinforcement-learning-3c2b6f0aa323	23
2.5	Model-Free RL, also known as direct RL, directly uses experience to optimize value/policy functions. Model-Based RL uses experience for model-learning, and indirectly, by planning, they optimize value/policy functions. Figure taken from chapter 8 of Sutton-Barto's Book [1].	24
3.1	Illustrative view of the sequence of trajectories generated by the agent-environment interaction. For convenience, we represented a finite interaction, in other words, an episode, finishing in state s_T	32
3.2	An example illustration of a Finite Markov Decision Process. The circles in black represent the states of the FMDP, that is: $S = \{s_a, s_b, s_c, s_d, s_e, s_f\}$. In green , there are represented the transition probabilities for each state-action pair: $p(s' s, a) = Pr\{S_{t+1} = s' S_t = s, A_t = a\}$ (e.g., $p(s_b s_a, a_{b_1}) = 0.7$), since the environment is stochastic. In blue we have the actions available for all states $a \in \mathbb{A}(s)$ (e.g., $A(s_a) = \{a_{b_1}, a_c\}$). For the sake of simplicity we assume an implicit equiprobable policy for the actions in each state, that is, $\pi(a s) = \frac{1}{ A(s) }, \forall s \in \mathbb{S}$. In red we have the reward signals received for each transition, that is, $r = R(s, a)$ (e.g., $R(s_c, a_{a_2}) = -3$).	35

3.3	Relationship between v_π and q_π . It is an weighted sum over the possible action-values in s weighted by the policy. Figure taken from [2].	37
3.4	Relationship between q_π and v_π . It is a sum between the immediate reward and an weighted sum over the state-values weighted by the transition dynamics of the environment. Figure taken from [2].	38
3.5	The state-value function backup diagram. For each s' in the bottom of the tree, there is some different weighting applied. The first level of the tree is weighted by $\pi(a s)$. The second level of the tree is weighted by the transition dynamics of the environment $\mathbf{P}_{s,s'}^a$. Figure taken from [2].	38
3.6	The action-value function backup diagram. For each a' in the bottom of the tree, there is different weighting applied. The first level of the tree is weighted by the transition dynamics of the environment $\mathbf{P}_{s,s'}^a$. The second level of the tree is weighted by $\pi(a s)$. Figure taken from [2].	39
3.7	Relationship between $v_\star$ and $q_\star$. It is a maximum operation over all the possible optimal successor action-values. Figure taken from David Silver's slides at: https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf	40
3.8	Relationship between $q_\star$ and $v_\star$. It is equal to the immediate reward plus a sum over all the possible optimal successor state-values weighted by the transition dynamics of the environment. Figure taken from David Silver's slides at: https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf	41
3.9	The optimal state-value function backup diagram. A maximum operator chooses the best sum across all actions, between the immediate reward and the discounted sum for the sucessor state-values weighted by each transition probability ($\mathbf{P}_{s,s'}^a$). Figure taken from [3].	41
3.10	The optimal action-value function backup diagram. A maximum operator chooses the best action-value for each of the successor states. This values are then summed up and weighted by each transition probability ($\mathbf{P}_{s,s'}^a$). Figure taken from [3].	42
3.11	The Generalized Policy Iteration diagram. The value function is approximated repeatedly to be closer to the true value of the current policy, and, meanwhile, the policy is improved repeatedly to approach optimality. The process is iterative and ultimately converges to the optimal value-function $v_\star$ and, consequently, to the optimal policy $\pi^\star$. Figure taken from [4].	44
3.12	A sample backup over one step from the successor state-value $V(S_{t+1})$ in the bottom, to the original state-value $V(S_t)$ in the top. Figure taken from [5].	45
3.13	The Sarsa's [6] policy iteration diagram. The Q-learning's diagram is similar, differing only in the policy evaluation step (instead of approximating the true value-function ($Q = q_\pi$), it approximates the optimal value-function ($Q = q_\star$). The Figure is taken from Lecture 5 of David Silver's lectures, available at: https://www.davidsilver.uk/wp-content/uploads/2020/03/control.pdf	47
3.14	The <i>Q-learning</i> diagram. On top, an action-value is updated according to the maximum subsequent action-value in the bottom. The arc on the bottom represents the max operator performed over all the successor action-values. Figure taken from [7].	49

3.15	The most common DQN architecture. In this example it receives 4 image frames as input and outputs 7 action nodes. The convolutional layers learn to extract the most relevant features of the raw image frames. The compacted version of the information produced by the convolutional layers is then passed to the dense layers which are responsible for outputting the Q-value for each action. Available at: https://paperswithcode.com/method/dqn	52
4.1	An UML-based illustration of the implemented RL-ROS integrated system. In the figure, <i>nodes</i> , <i>topics</i> and auxiliary <i>modules</i> are in blue, yellow and pink, respectively. Solid arrows ($\longrightarrow$) represent subscribe-publish relations between <i>nodes</i> and <i>topics</i> and dashed arrows ($--\rightarrow$) represent interchanging of data between a <i>node</i> and its auxiliary <i>module</i> . A <i>module</i> is an object outside the main ROS network that performs isolated computation.	57
4.2	Q-table illustration. N_s state rows and N_a action columns. The complexity of the Q-table is $N_s \times N_a$. N_s is the dimension of the state-space. N_a is the dimension of the action-space.	59
4.3	Nearest Neighbor example. Given $Z = \{z_0, \dots, z_8\}$, $NN(x, Z) = z_1$	60
4.4	Illustration of the complete state representation. In this example, $r = 3, \phi = 180^\circ, b = 4$. The angle to the goal G is somewhere around 40 degrees ($\theta_g \approx 40^\circ$). The <i>los</i> is defined as follows: it can not perceive the obstacle O_1 since it is out of the sight. O_2 is occupying the third bound in the second distance level. The second bound is free since the distance to O_5 exceeds h_r . The first bound is occupied by the obstacle O_4 in both first and second distance levels, and by the obstacle O_3 in the second level. Prioritization is given to obstacle O_4 since it is the closest obstacle and it is causing the average of the laser distances to be contained in the first distance level.	61
4.5	A ramp filter symbolizing a degree of importance as a function of distance. The lower the distance the higher is the importance we need to give since closer obstacles (closer distances) are more urgent to take into account.	65
4.6	Shapeness described by always having a reward signal for every x . Shapeness is also described by having gradient.	67
4.7	Sharpness described by not always having a reward signal for specific values of x (in this example, there is no signal between 1 and 9, that is, $x = 0$ if $1 \leq x \leq 9$). Sharpness is also described by not having any gradient or being zero for different values of x (in this example, $x(4) = x(6)$).	68
4.8	A snapshot from our implemented simulator. In blue, we have a circle representing the robot (the circle diameter is equal to the real robot diameter). In green we have the obstacles with different diameters so as to offer variability and facilitate the exploration. In red we have the goal. The red arrow on the top of the robot is its current orientation.	75
4.9	The average TD error by episode, according to Metric 1. In the y axis, there is the absolute value of the average of the TD-error in the episode. In the x axis, there is the number of episodes.	77
4.10	The cumulative reward by episode, according to Metric 2. In the y axis, there is the value of the accumulated reward in the episode. In the x axis, there is the number of episodes.	77
4.11	The number of decisions/actions taken by episode, according to Metric 3. In the y axis, there is the number of decisions in the episode. In the x axis, there is the number of episodes.	78

4.12	The reasons for episode ending by episode, according to Metric 4. In the y axis, there is the binary value representing the ending reason for the episode, that is, a value of 1 if the goal was reached without collisions and a value of 0 if the episode ended because of collision with an obstacle. In the x axis, there is the number of episodes.	78
4.13	The first visit exploration percentage, according to Metric 5. In the y axis, there is the percentage value of the first visit exploration. In the x axis, there is the number of episodes.	79
4.14	The testing results for Episode 3	79
4.15	The testing results for Episode 5	80
4.16	The testing results for Episode 10	80
4.17	The testing results for Episode 20	80
4.18	The testing results for Episode 29	81
4.19	The testing results for Episode 34	81
4.20	The testing results for Episode 37	81
4.21	The testing results for Episode 39	82
4.22	The two real testing environments.	83
4.23	The results for the environment O_1 , robot position R_1 and goal position G_1	83
4.24	The results for the environment O_1 , robot position R_2 and goal position G_2	84
4.25	The results for the environment O_2 , robot position R_1 and goal position G_1	84
4.26	The results for the environment O_2 , robot position R_2 and goal position G_2	85

List of Tables

4.1	The chosen hyperparameters for training	76
-----	---	----

Abbreviations

RL	Reinforcement Learning
DQN	Deep Q-Networks
DRL	Deep Reinforcement Learning
SL	Supervised Learning
TD	Temporal-Difference
FMDP	Finite Markov Decision Process
MDP	Markov Decision Process
MB	Model-Based
MF	Model-Free
PB	Policy-Based
VB	Value-Based
MCTS	Monte Carlo Tree Search

Chapter 1

Introduction

In the field of Artificial Intelligence, the Reinforcement Learning (RL) paradigm is showing promising results as a generic purpose framework for solving, e.g. optimization, control and decision-making problems). More than a subfield of Machine Learning, RL stands out as a formalism for automated decision-making [8].

Reinforcement Learning is behind the science of intelligence and is being inspired by the way living organisms think and act in the real-world¹. This inspiration drew a paradigm based on the belief that a living organism, let us call it an agent, learns and improves on its own by trial-and-error and through interactions with a surrounding environment.

The RL process can be described informally as following:

1. The agent encounters itself in a certain state;
2. The agent reasons (using its current knowledge, acquired through past experiences) about the next action to be taken;
3. The agent performs an action causing an effect on the environment;
4. As a causal process, the environment responds with:
 - (a) A scalar reward signal allowing the agent to understand the consequence of the action;
 - (b) An observation with the new state for the agent.
5. The agent learns from the process and goes back to the first step.

Such an interaction consists of a closed-loop system as shown in the Figure 1.1, and remarkably one can also pose the described framework in a control system context. To conclude, we can easily establish a link between the robotics field, the paradigm of RL and the topic of this thesis: instead of explicitly programming a robot to follow instructions in an imperative way, the RL framework allows the robot to accomplish a given task by means of trial-and-error learning.

¹deepmind.com/blog/article/Dopamine-and-temporal-difference-learning-A-fruitful-relationship-between-neuroscience-and-AI

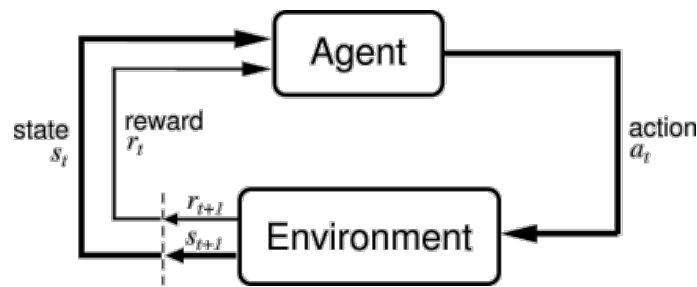


Figure 1.1: Reinforcement Learning loop. Available at: <http://www.incompleteideas.net/book/ebook/node28.html>

1.1 Context and Motivation

In order to give the necessary context and motivation to the topic of this thesis and how reinforcement learning came to the scene in the field of robotics, it is necessary to make a passage through the different levels and meanings of robotic intelligence and its evolution.

The evolution of robotic intelligence is supported by a balance between two aspects: computational complexity and the effort required by the programmer[9]. In the early days, direct programming was the usual approach: complete control of the robot at a low level and manually specifying (programming) any possible decision to be made by the robot. Although still widely used in industrial contexts due to its structured environment, such an approach has the great disadvantage of consuming a lot of “programming” time, not being effective in unstructured environments, being very prone to errors and not being subject to reuse and scalability. At that time, there was little or no autonomy on the robot side, a great effort for the programmer and low computational complexity.

A desire to expand the robot’s autonomy capabilities subsequently culminated in the appearance of imitation approaches in which the concept of autonomy was finally appearing, without yet taking effort from the programmer’s side because it was still necessary to formalize the execution of the task. The imitation process consisted of the manual control of the robot and the memorization of these movements made to be later revisited and executed. The memorization of the experience allowed the robot to start performing tasks in a somewhat autonomous manner. Other imitation approaches such as observational learning in which the robot watches humans performing a task using computer vision methods and subsequently executing the learned task were also explored. In that sense, we may say that the concept of autonomy started to emerge but in a timid way.

Then, another domain of intelligence began to appear trying to fill the gaps left by its predecessors. Questions like the ones that follows were considered:

- What if we want the robot to perform tasks that cannot be programmed or demonstrated because they are too complex? (e.g., lifting weights, jumping three meters high, running at high speed, etc.)

- What if the optimal solution to the problem is unknown due to the impossibility and/or complexity of the analytical formulation of the problem?
- What if we want the learned task to be generalized to problems of a similar nature and for the robot to be able to adapt to new, unexpected situations?
- What if unexpected events happen on the hardware itself and we want the robot to dynamically adapt to those changes? (e.g., changing body parts, mechanical wear, heating, etc.)

Reinforcement Learning appears as the domain that would endow the robot, the ability of learning to perform a given task in an autonomous way, under a trial and error paradigm. By taking samples from a designed reward function while it operates inside an environment, the robot is able to learn the actions that would lead to an increase on the amount of collected reward on the long run. In this context, the reward function must be designed to guide the robot's actions towards a desired behaviour. We thought of an example of the basic thinking behind RL, as follows:

Consider that we are trying to teach a dog to catch a ball, which clearly cannot be explicitly explained to the dog. Instead, we do the following: we throw the ball, and whenever the dog catches the ball, we give the dog a cracker and the dog would be happy. If the dog doesn't catch the ball, we don't give him a cookie and the dog gets sad. The dog tries and tries again and again until he finds a generalized behaviour for different circumstances. Under a reward or punishment policy, the goal of learning to catch a ball in different ways is achieved. The learning process becomes a process whose objective is to maximize the chance of being successful.

The direction in which the evolution of intelligence is heading is fascinating. We are no longer talking about "scripted" and explicitly programmed commands, but autonomous, possibly non-deterministic, and more natural behaviour. In other words, we are talking about algorithms that learn optimal behaviors by themselves.

On the programmer's side we started to see also a radical change: Instead of explicitly implementing and designing the desired behavior, the programmer's goal is to decide which reward signals (representing what is good and what is bad) are necessary to guide the robot to learn a behavior by itself.

As the computational power of modern computers increase, so does the levels of complexity of learning and control algorithms. This allows the robotic behaviour to be very different than the explicit programming of tasks by a human, resulting in higher levels of robot autonomy and adaptability.

After this contextualization, we can rediscover the main theme: Reinforcement Learning and Robotics. At present, robotics is immersed in the investigation of this new paradigm. Many methods are being studied and analysed at the present time and they all have their unique characteristics.

The aim of this thesis is to implement a well known RL method called Q-learning [10; 11] (explained in detail in the section 3.2) and evaluate and discuss about its performance to solve a

specific robotic task. In the following section it will be exposed a more detailed explanation about the main objectives of this thesis and its contributions.

1.2 Objectives and Contributions

The final goal of this thesis is to solve a robotic navigation task in obstacle scattered environments (the task is exposed in 1.2.1) using Q-learning [10; 11]. The solving of the task will be evaluated and discussed by analysing intuitively chosen metrics that seemed suitable to measure the quality of the execution of the task (presented in 1.2.3), both for the training and the testing phases.

Regardless of the main decision one needs to have about the most suitable RL method to solve any problem at hand, the main challenges of an RL problem lay mostly on a proper **design of a reward function** and in a reasonable **definition of the state** for the agent. These former mentioned aspects are the foundation of the problem: with a well designed reward function the robot will be well guided to what we want it to achieve [12]. Additionally, with an efficient state engineering, the agent has sufficient awareness about its situation. Without this foundations there is no advantage in implementing RL. RL brings several and unique advantages, but in the present time it still needs to be well supported by these considerations.

In subsection 1.2.2, we present the reasons behind the choice of the RL methods, taking into account their suitability for the context of the problem and also the timing constraints of this dissertation.

Additionally, in subsection 1.2.4, we will present the methodology for the accomplishment of the proposed work for this dissertation.

With our work we seek to reproduce that Model-Free and Value-Based RL methods sustained by Q-learning principles could be applied to obstacle avoidance tasks in a robotic setup. Not only we hope to prove its applicability by not requiring a model of the environment, but also, by proving that the robot can learn an obstacle avoidance task only by using 2D LiDAR perception and Q-learning-based RL.

Essentially, Q-Learning was chosen for the sake of its simplicity and popularity. Q-learning is a tabular method that works well in small state-spaces. The Q-learning approach consists of the use of a table that stores the learned *quality* of all of the state-action pairs available for the agent (the Q-table is exposed in subsection 4.1.2).

Generally, tabular RL algorithms have limitations for high-dimensional tasks like robotics and real-world tasks. Huge RL tables require a lot of memory usage to store vast state-action spaces, and the convergence conditions for those tables are hardly respected. To converge it is essential to explore the Q-table in its entirety and frequently revisit every single state-action pair in a reasonable amount of time (the convergence conditions for Q-learning are presented in subsection 3.2.4). However, if the problem could be modeled by a relatively small and simple MDP (MDPs are exposed in subsection 3.1.3), resulting in a low-dimensional Q-table, it becomes feasible to use Q-learning to solve complex tasks, as we can demonstrate by the results of this thesis where we successfully solved a navigational and obstacle avoidance type of task.

Despite the former mentioned constraints and limitations, with this work, we aim to propose a simple and effective approach to solve a robotic navigation obstacle avoidance type of task using Q-learning. We contribute by presenting a successful state representation and an effective reward function design to fit the needs of the task and overcome the dimensionality problem (the curse of dimensionality is presented in subsection 2.2.1). We present, not only satisfying results when testing the learned behavior in the simulated environment, but also in the real world with real hardware, ultimately proving that it is possible to apply Reinforcement Learning into Robotics.

1.2.1 Task

It was decided that a navigational type of task with obstacle avoidance and waypoint-goal reaching, more generally an obstacle avoidance type of task [13], could be learned by the robot. This task choice was motivated by its simplicity, convenience and popularity, easing our needs for a robotic task to work with.

The problem to be solved consists in learning a policy that steers the robot in the direction of a waypoint-goal while avoiding obstacles during the trajectory. The linear velocity of the robot will remain constant, positive and non-zero and the available control actions are discrete steering actions (angular velocities). During training, the obstacles and the goal do not exhibit dynamics, hence they are static. The essence of the task is illustrated in the Figure 1.2.

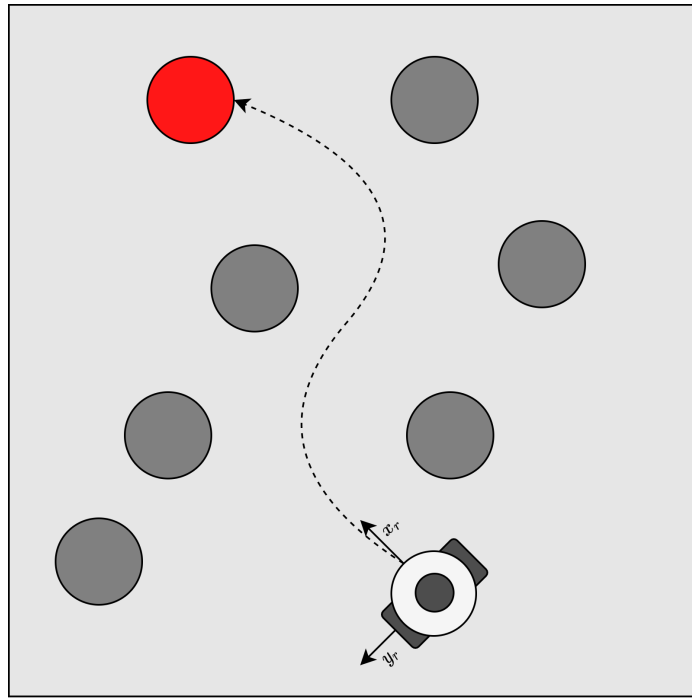


Figure 1.2: Illustration of the task under study. The robot needs to learn how to steer in the direction of the goal while avoiding collisions along the way.

1.2.2 Method

Q-learning belongs to the family of the **Model-Free** (Model-Free methods are exposed in subsection 2.1.1) and **Temporal-Difference (TD) Learning** [5] methods. **TD** methods are appropriate for non-episodic tasks because they can update their behaviour at each time step. This is ideal for robotic tasks, hence not requiring episodic settings like Monte Carlo Methods [14]. For the sake of simplicity, a **Model-Free** setting is also convenient, so that we do not need to estimate and use a model of the environment. In the highest-level, the field of RL is divided into Model-Free methods and Model-Based methods. Model-Free methods have the following advantages over Model-Based methods:

- Easier to implement, more popular and more intuitive.
- Although sample inefficient they are more adaptable, because they are guided by value only, not requiring a model that could change or be poorly estimated. There are, of course, advantages of using Model-Based methods because they are sample efficient, in other words, with access to a model, the agent can learn inside its inner model, not requiring too much interaction with the real environment. However, the process of estimating a model is still, currently, a demanding and complex process [15]. For the reasons explained above, in this work, we have focused on the Model-Free approach.

1.2.3 Metrics

Like any machine learning problem, there is a training phase where a model is learning by adjusting its parameters while trying to achieve some kind of behavior or task. There is also an evaluation phase, where a model is tested to check if the training was useful and gave space for generalization. For both phases, we need to create metrics that will measure its quality and efficiency.

Training performance and metrics To monitor the convergence of an algorithm during training we need to check the evolution of specific variables which translate the distance between what are the right values a model should predict (targets) and the actual values it predicts (predictions). In the context of Value-Based RL, specifically Temporal-Difference Methods (like Q-learning), these variables are called TD-errors.

The cumulative reward by episode is another type of metric that enables us to understand the evolution of the efficiency of the agent's behavior during episodes: if it accumulates more positive reward, or less negative reward (dependent on the reward function design), it means that it is improving based on the main goal, that is maximizing the cumulative reward. We must be aware that this metric could also represent hacking and undesired agent's behavior: instead of reaching the desired behavior it could be exploiting loop-holes in bad designed reward functions, a phenomenon known as reward tampering [16].

The total number of steps or decisions by episode is another useful metric of efficiency. Taking less decisions could mean that the agent is becoming more efficient (e.g., arriving quicker to a goal), however it could also mean that it is doing risky behavior. On the other hand, a higher amount of steps could represent longer "surviving" time, while being more prudent along the way (e.g., avoiding some type of obstacles along the way).

Fusing all of these kind of metrics, we have a more robust interpretation about the quality of the behavior being learned, since what it is ambiguous or not conclusive by some metrics, could be clearly interpreted by others.

For the training phase, we considered the following metrics:

- M1** For convergence, in the case of Q-learning we follow the average TD-errors by episode.
- M2** The cumulative reward by episode.
- M3** The number of actions by episode.
- M4** The reason for episode ending by episode: for each episode, the algorithm outputs a boolean *True* if the episode ended because of collision with an obstacle. It outputs *False* if the episode ended because of a successful goal reaching. If we observe more goal reachings during training, it means that the algorithm is improving.
- M5** We thought of plotting the exploratory first visit percentage during training. Each new first visit for a state-action pair is counted. The obtained percentage is the number of counted first visits (v) for each s, a divided by the total amount of state-action pairs, in other words, the complexity of the Q-table. Let L be the total size of the Q-table. This metric is defined as follows:

$$M_5(\%) = 100 \frac{v}{L} \quad (1.1)$$

Testing performance and metrics To test the trained behavior, we must analyse the agent's performed trajectory. Essentially, the performed trajectory is what will describe the success of the task: if it reaches the goal and avoids obstacles along the way, it is achieving the desired behavior.

The testing phase will consist of a batch of episodes each with different positions for the robot and the goal. With different robot and goal positions we can effectively test the generalizing capabilities of the learned behavior.

For all testing metrics, consider a time step $i \in [0, D - 1] \subset \mathbb{N}$ for each decision taken in a total of D decisions for each testing scenario. We considered the following testing metrics:

- M6** The distance error between the goal position and the robot position during the evaluation episode. Let e_i be the difference between the robot position R_i and the goal position G_i in time step i . The error to be plotted is computed as follows:

$$M_6^i = e^i = |R^i - G^i|, \quad (1.2)$$

where

$$M_6 = \{M_6^0, \dots, M_6^{D-1}\}. \quad (1.3)$$

M7 The successful episodes s (without collisions) in a total of E episodes, as follows:

$$M_7(\%) = 100 \frac{s}{E}. \quad (1.4)$$

M8 Plotting the robot position, the goal position and the obstacles positions for each testing scenario evaluation, so as to visualize the performed trajectory. Let i be the time step by decision taken, the robot position in time step i be $R^i = (R_x^i, R_y^i)$, the goal position in time step i be $G^i = (G_x^i, G_y^i)$ and an obstacle o^j position in time step i be $o^{j,i} = (o_x^{j,i}, o_y^{j,i})$, in a set of k obstacles, that is, $O^i = \{o^{1,i}, o^{2,i}, \dots, o^{k,i}\}$. This metric, for each scenario, will be defined by:

$$M_8 = \{R^i, G^i, O^i, \dots, R^{D-1}, G^{D-1}, O^{D-1}\} \quad (1.5)$$

1.2.4 Methodology

To accomplish our work, the following elements were employed:

Mobile Robot: A two-wheeled robot named TurtleBot3 Burger²



Figure 1.3: The Turtlebot3 Burger

²<http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/>

Control/Communication: To communicate with the robot hardware, it is necessary to use a ROS framework in order to take advantage of all its features and capabilities compatible with the robot model. The Robot Operating System³ (ROS) is an open-source set of software libraries and tools specifically designed for robot applications and for an easy integration of several software parts needed for robot operation, from drivers to state-of-the-art algorithms.

Simulated environment: The RL algorithms will be initially trained and tested in a simulated environment. Part of the objective consists in evaluating the performance of the RL algorithm in a similar real-world environment, a process known as transfer learning. In a first stage, we chose Gazebo⁴ as our training simulator. However, we questioned ourselves if there was another simulator alternative, since we were dealing with huge training time durations. We realized that it was not necessary to sacrifice lots of computational resources with powered physical 3D features and high-quality graphics. We intuitively assumed that we could achieve the same results by an hypothetical simulator with 2D physics and LiDAR perception. Hence, we took the initiative to implement our own simulator⁵ from scratch with a lighter and simpler 2D setting. Successfully, we reduced considerably the amount of time for our simulations without compromising the learning process and the goal of achieving the desired behavior. Additionally, to note that, due to the change of simulator, the architecture presented in 4.1.1 will suffer a subtle change in a specific node, specifically the *gazebo* node. This node turns out to be our new simulator, behaving the same way as gazebo, however being much less computationally demanding and therefore drastically decreasing total training time for the Q-learning.

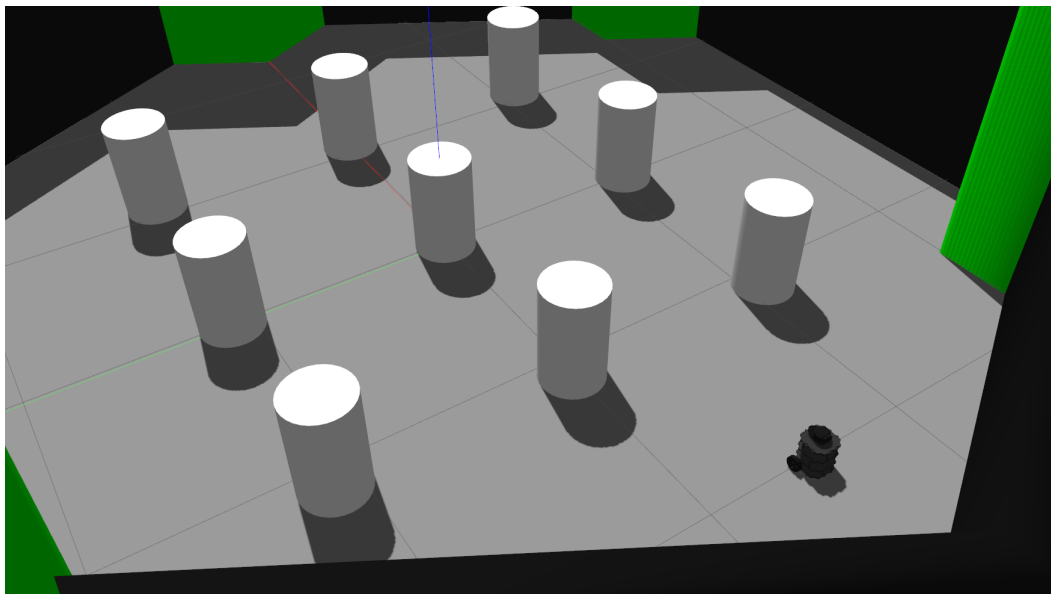


Figure 1.4: A screenshot from a Gazebo simulated environment where the robot learns the task

³ROS's official website: <https://www.ros.org/>

⁴http://gazebo.org/tutorials?tut=ros_overview

⁵The source code is available at: https://github.com/Fsn9/2D_World_with_obstacles_for_RL_testing

Physical environment: A preparation of a physical environment where the RL algorithms will be tested using the mobile robots. The physical environment should have similar characteristics to the simulated scenario, such that the RL algorithm will be able to apply the simulated experience to the real-world task.

Algorithms: A Reinforcement Learning algorithm named Q-learning.

1.3 Organization of the Dissertation

This dissertation is organized as follows:

Chapter 1 introduces the dissertation, providing the context and the motivation for the work proposed. It also presents the objectives, the proposed problem and the contributions of the work. Chapter 2 reviews the main state-of-the-art RL methods and the main challenges when combining RL and Robotics. Chapter 3 presents the theory behind the chosen RL method: Q-learning. Chapter 4 dives into the practical side of the thesis exposing the developed implementation for the Q-learning method. It also exposes the achieved results. Chapter 5 exposes conclusions about the work and leaves future work perspectives.

Chapter 2

Literature review

This chapter exposes the state-of-the-art and an overview analysis of the current literature with respect to Reinforcement Learning applied to Robotics. Section 2.1 introduces the main modern available RL methods in the literature. Section 2.2 introduces key challenges on modern Reinforcement Learning when applied to Robotics as well as state of the art examples.

2.1 RL Methods

This section presents an overview over the current main state-of-the-art methods available in modern RL ¹. Although several variants of these methods and more advanced RL techniques exist, we focus on making a concise exposition describing the most important ideas behind the main methods. Hence, the goal of this section is to present the most popular design choices when we want to apply RL to more demanding and complex problems, like Robotics. To note that these methods are naturally fused with neural networks, so we will dive inside the field of Deep RL. We expose the ideas in this section following the tree structure represented in the Figure 2.1, which gives a glimpse over the methods, their relations and their categories.

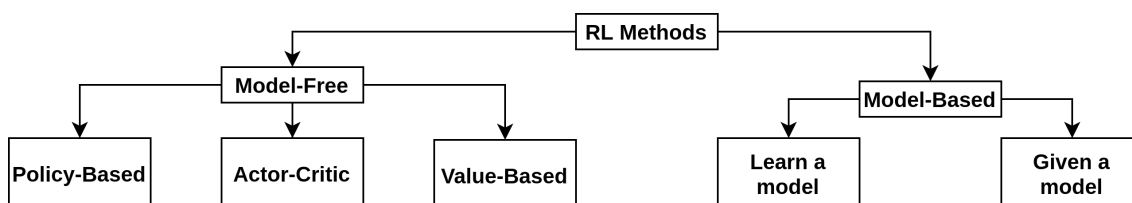


Figure 2.1: Useful taxonomy of the RL methods. Model-Free methods learn a behavior without understanding the environment (no model knowledge), by direct experience. Model-Based methods, could be given a model and learn a behavior inside of it, or they could learn a model first and then learn a behavior inside of its own learned model.

In brief, RL is divided into two main branches of methods: Model-Based (MB) methods and Model-Free (MF) methods. The central question behind the branching choice is whether the

¹https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

agent has access to a model of the environment, whatever this model is given (known) or updated in an online fashion from the data (learned). **Model-Free** methods have the unique feature of not requiring a model of the environment's dynamics. The algorithms based on this approach learn from experience (by trial-and-error). **Model-Based** methods, as the name suggests, have the knowledge about (or the ability to learn) a model of the environment and have the unique capacity to plan ahead because they have access to the transition dynamics of the environment.

As stated by Sergey Levine ² in a podcast ³, Reinforcement Learning methods could be described in a very fundamental way like the following: "It's about learning models that could answer 'what if' questions". Based on this, we may say that: (1) Model-based methods try to answer the question "**What** new state would I get **if** I take a certain action in some state"; (2) Value-based methods try to answer the question "**What** value would I get **if** I am in some state"; (3) Policy-based methods try to answer the question "**What** is the best action **if** I am in some state".

To conclude, let's add that, from the perspective of other fields like Psychology and Neuroscience, MB and MF are interestingly connected to different aspects of intelligence [17]. Model-Free RL is more related with reactive decision-making and trial-and-error learning, while Model-Based RL is more related with cognition and deliberative planning. Huge debates concerning the advantages and disadvantages between both approaches have been occurring throughout the years. MF methods are simpler, direct and do not suffer from the complex estimation of a world model, but they cannot plan ahead. MB methods can make a lot better use of limited amount of experience and are not blind to the environment they are interacting with, thus they are aware of, can imagine and plan ahead in their own estimated world model, however they can suffer from poor model estimation, a process known as model-bias, compromising the learning process.

2.1.1 Model-Free

Model-Free methods are a family of algorithms that directly learn a behaviour (in the form of a policy, an action-value function or both) by trial-and-error and by direct interaction with an environment. They are MF in the sense that they do not have access to and do not seek to understand a model of the environment that is, there is no knowledge about the transition dynamics of the environment $T(s', a, s)$. Hence, they purely learn and seek the optimal policy π^* from "experience".

These methods are more popular, and, to the current date, more intensively explored and tested than MB methods [15]. Since they do not require the estimation of a model they are less demanding and easily implemented.

MF methods do not plan ahead, they instead could:

- Directly learn a policy function $\pi_\theta(a|s)$ (Policy-Based methods in subsection 2.1.1.1)
- Learn an action-value function $Q_w(s, a)$ (Value-Based methods in subsection 2.1.1.2)

²<https://scholar.google.com/citations?user=8R35rCwAAAAJ&hl=en>

³https://youtu.be/kxi-TT_-Nc?t=3018

- Learn both an action-value function $Q_w(s, a)$ and a policy function $\pi_\theta(a|s)$ (Actor-Critic methods in subsection 2.1.1.3)

Each one of the former methods are usually implemented in the literature by means of function approximators such as Neural Networks, hence their parameterization with weight vectors (e.g., $\theta, w, \dots$). In the following subsections we will expose each one of the former methods.

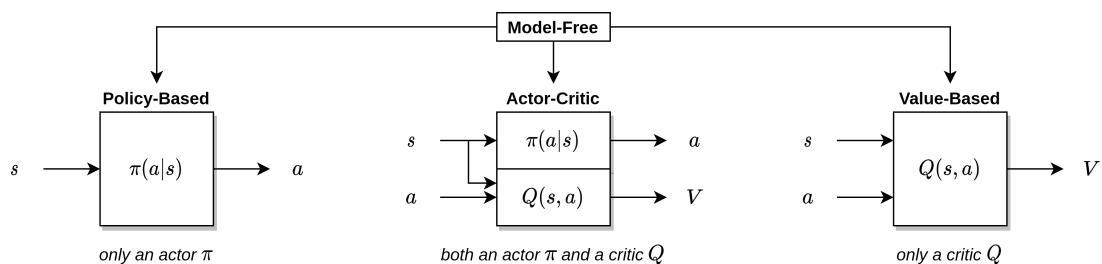


Figure 2.2: Model-Free types of algorithms. Policy-Based methods estimate actors, also named as policies π . Value-Based Methods estimate critics, also named as action-value functions Q . Actor-Critic methods combine both the former approaches and they estimate actors and critics.

2.1.1.1 Policy-Based

Policy-Based (PB) is a class of methods that directly learn a policy π_θ [18]. Contrary to Value-Based methods, PB methods directly parameterize the policy, hence the behaviour of the agent can be adjusted by tweaking the parameters θ . With a parameterized policy, they can select actions without consulting value-functions. These parameters are learned iteratively through the learning process so as to maximize the agent's performance.

As stated by David Silver^{4,5} in one of his lectures:

“How can I see some experience and from that experience figure out how to change my policy in the direction that makes my policy better?”

PB methods are often classified as direct RL⁶ methods. Direct in the sense of pragmatically finding and optimizing what we are looking for: a policy for the agent, in other words, a mapping from states to actions. If the policy we are trying to learn is stochastic then we represent it as a probability distribution over actions $a \in \mathbb{A}(s)$ given a state $s \in \mathbb{S}$: $a = \pi(a|s, \theta) = \text{Pr}(A_t = a | S_t = s, \theta_t = \theta)$. If the policy we are trying to learn is deterministic it is a direct mapping from states to actions $a = \mu(s, \theta)$. The final goal is to find the optimal policy $\pi^*(s)$ that achieves the maximum expected cumulative reward for all states ($\forall s \in \mathbb{S}$).

⁴<https://www.davidsilver.uk/teaching/>

⁵<https://youtu.be/KHZVXao4qXs?t=126>

⁶PB methods inside the Model-Free family are considered direct because they directly optimize the policy, if we compare them with Value-Based methods that don't directly learn a policy. Direct RL is also used in another context: it is often associated with Model-Free methods if we compare them with Model-Based methods. Model-Free directly use the experience to optimize value/policy functions, while Model-Based methods indirectly use the experience to optimize policy/value functions (they use the experience for model-learning, and consequently use the learned model for policy/value function optimization).

These methods seek to maximize performance, and, contrary to most methods where we want to minimize an error function performing gradient descent (minimize the error between targets and predictions), policy gradient methods perform gradient ascent instead, maximizing a score function $J(\theta)$. Each parameter update $\Delta\theta$ (see in equation 2.1) is performed by multiplying a learning rate $\alpha \in [0, 1]$ (how fast we want to learn, in other words, how strong is each parameter update) by the gradient of the score function $\nabla_{\theta}J(\theta_t)$. This gradient points to the direction one needs to update the parameters so the performance is maximized. Hence, optimizing π_{θ} , is seen as finding the best parameters θ so as to maximize a score function $J(\theta)$.

$$\begin{aligned}\Delta\theta &= \theta_{t+1} - \theta_t \\ &= \alpha \nabla_{\theta}J(\theta_t)\end{aligned}\tag{2.1}$$

Generally, in an episodic setting ($T \rightarrow \mathbb{N}$), this score function is formalized as the expected cumulative reward from an initial state s_0 to a terminal time step T while following policy π_{θ} . This expected cumulative reward is defined as:

$$\begin{aligned}J(\theta) &\doteq v_{\pi_{\theta}}(s_0) \\ &= \mathbb{E}_{\pi_{\theta}} \left[\sum_{t=0}^T \gamma^t r_t \right].\end{aligned}\tag{2.2}$$

To update the parameters, one needs to differentiate the score function (proof in [19]), arriving at the final expression, where $R(\tau)$ is the episodic return, defined as:

$$\begin{aligned}\nabla_{\theta}J(\theta) &= \mathbf{E}_{\pi_{\theta}} \left[\sum_{t=0}^T \frac{\nabla_{\theta}\pi_{\theta}(a_t|s_t)}{\pi_{\theta}(a_t|s_t)} R(\tau) \right] \\ &= \mathbf{E}_{\pi_{\theta}} \left[\sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t|s_t) R(\tau) \right].\end{aligned}\tag{2.3}$$

The expression from equation 2.3 describes the following intuition: the agent updates the parameters θ in the direction of the gradient ∇_{θ} . This direction points to a place in the parameter space that most increases the probability of taking a_t on future situations where one visits s_t again. In the gradient's direction one updates the parameters proportionally to the return score $R(\tau)$, turning actions that yield higher returns to be more likely, and actions that yield lower returns to be less likely. The parameter update is also inversely proportional to the probability of taking a_t in s_t , $\pi_{\theta}(a_t|s_t)$. If there is no division by $\pi_{\theta}(a_t|s_t)$, more probable actions are at an advantage. This division assures that actions that are less likely, but promising (yielding higher returns), they have a chance to uplift their probabilities strongly⁷. The $\log \pi_{\theta}(a_t|s_t)$ term is derived by a mathematical trick, simplifying the expression, since:

$$\nabla \log f(x) = \frac{\nabla f(x)}{f(x)}.\tag{2.4}$$

⁷<https://towardsdatascience.com/an-intuitive-explanation-of-policy-gradient-part-1-reinforce-aa4392cbfd3c>

A well known and fundamental example of a PB method is the REINFORCE algorithm [20], where the agent updates its parameters θ using the Monte Carlo return G_t . The REINFORCE agent updates its parameters according to:

$$\Delta\theta = \alpha \nabla_{\theta} \log \pi_{\theta}(a_t|s_t) G_t \quad (2.5)$$

Policy gradient methods have good convergence properties, but typically suffer from high variance caused by noisy estimations⁸ of the value of the policy being followed. It came the need to improve the evaluation of the policy, and so, a more general form of REINFORCE was created, the REINFORCE with Baseline [21]. This method subtracts a term from G_t in order to reduce the variance and speed up the convergence, as follows:

$$\Delta\theta = \alpha \nabla_{\theta} \log \pi_{\theta}(a_t|s_t) (G_t - b(s_t)) \quad (2.6)$$

The baseline $b(s_t)$ improves performance and could be any value, but a more natural approach is to turn it into a state value function $\hat{v}(s, \mathbf{w})$. If the agent learns a state-value function it could make better evaluations, thus improving its learning process.

Lastly, one concludes that Policy-Based Methods are direct and have assured convergence properties, but they suffer from poor evaluations with no credit assignment for each action in the episode. Even with $b(s) \equiv \hat{v}(s, \mathbf{w})$, this value is only estimated for each state transition, thus all rewards $r_k \in \{r_1, r_T\}$ from the episode are equally subtracted from this state-value function. RL created a new branch where we have a dedicated evaluator, more known as a critic, that performs evaluations for the cumulative reward by each state-action transition, not needing to wait and sample the rewards until the end of an episode. This new branch of RL is called Actor-Critic, where the agent learns a separate action-value function $Q_w(s, a)$ or a state-value function $V_w(s)$. This separate value function is called a critic, and it has the ability of estimating value functions at each time step, hence not needing to work on episodic regimes⁹. Hence, the actor $\pi_{\theta}(a|s)$ is evaluated and supported by a critic, improving its learning due to better scoring, estimation and less variance and with an improved ability of assigning credit for each action taken. Further, in subsection 2.1.1.3 we expose Actor-Critic methods.

2.1.1.2 Value-Based

Value-Based (VB) is a class of methods that estimate and optimise value functions. VB methods only estimate critics, however implicit actors are extracted from those value functions, that is, an implicit policy is also improved subsequently if we act greedily with respect to those value

⁸These noisy estimations, in vanilla Policy Gradient methods are typically computed by Monte Carlo returns G_t . G_t accumulates rewards during the whole episode and so, we have a lot of variance, thus it is hard to assign credit to each action taken during the episode. The policy is then poorly evaluated and the convergence takes longer.

⁹Actor-Critic methods quit Monte-Carlo episodic estimates and adopt Temporal-Difference estimations instead. With TD, Actor-Critic makes n-step predictions instead of sampling, proven to be superior in terms of reducing variance. It also enables online and continuous learning, because the estimates are at each step, not needing to wait until the end of the episode.

functions. Value-learning focuses in the problem of estimating the value of being in a state, in other words, the expected cumulative reward of following policy π from state s onwards:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t | S_t = s] \\ v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \end{aligned} \quad (2.7)$$

The second and third line from equation 2.7 are just two ways of expressing the same thing. It is useful, however, getting the third line expression to better explain the next equation. Equation 2.8 exposes us the general form of a state-value function. The value of being in state s is the weighted sum of the probability of taking action a in s while behaving like π multiplied by the probability of transiting from s to s' and receiving reward r after acting with a multiplied ultimately with the immediate return plus a discounted value of being in the next state s' .

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_\pi(s')]. \quad (2.8)$$

In an iterative fashion, according to the Dynamic Programming framework [22], we can iteratively update the values of states according to the expression in equation 2.9:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')]. \quad (2.9)$$

From the former equations one can conclude the intuition behind the usage of a model to choose actions based on state-values: knowing the model's dynamics, the agent looks ahead one step and chooses the action that takes him to the state with the highest value. However, since we are working in a Model-Free setting, and we don't have access to the model dynamics $p(s',r|s,a)$, we cannot implement that. From the former thinking, one arrives at the necessity of action-value learning [23] in a Model-Free setting: it is useful to learn the values for each action we can take, so we can arrive at trial-and-error behavior (completely blind to the world's dynamics).

In Model-Free settings, action-value learning¹⁰ is the principle that sustains Value-Based methods. It consists of the learning of a parameterized action-value function Q_θ , a critic, meaning that, through time, the action-values get better and more accurate. An implicit actor is trained indirectly¹¹, since a VB agent acts optimally if acting greedily with respect to the action-value function Q . If the agent converges to the optimal action-value function Q^* , the optimal policy for each s will choose the action a that maximizes the expected return, that is,

$$\pi^*(s) = a^*(s) = \arg \max_{a \in \mathbb{A}} Q^*(s, a) \quad \forall s \in \mathbb{S}. \quad (2.10)$$

¹⁰When no model is available, if we want to act based on value functions, one needs to act with respect to action-value functions.

¹¹Since Q-value estimates are getting better, if we act greedily according to those estimates, we have an optimal policy. Hence, we have an implicit actor, that, without being parameterized, is indirectly improving.

By definition, $Q^*(s, a)$ is the expected cumulative reward after taking action a in s and acting according to the optimal policy forever after: 2.11:

$$Q^*(s, a) = \max_{\pi} \mathbb{E}[G_t | S_t = s, A_t = a]. \quad (2.11)$$

Finally, we can establish a final connection between action-value and state-value functions¹². By definition, the optimal state-value function is the maximum expected cumulative reward for any given state s and following the optimal policy thereafter, thus,

$$V^*(s) = \max_{\pi} \mathbb{E}[G_t | S_t = s]. \quad (2.12)$$

Since the optimal policy for an action-value-learning setting is Q^* then we can assume that the best state-value is equal to the best action-value if we choose the best action. The optimal Q^* can then be linked to the optimal V^* [3] as follows:

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{a \sim \pi}[Q^{\pi}(s, a)] \\ V^*(s) &= \max_a Q^*(s, a). \end{aligned} \quad (2.13)$$

$$\begin{aligned} Q^*(s, a) &= \mathbb{E} \left[R_{t+1} + \gamma \max_{\pi} \mathbb{E}[G_{t+1} | s_{t+1}] | S_t = s, A_t = a \right] \\ Q^*(s, a) &= \mathbb{E} [R_{t+1} + \gamma V^*(s_{t+1}) | S_t = s, A_t = a]. \end{aligned} \quad (2.14)$$

There are various successful state-of-the-art deep RL VB methods in the literature. We will show some examples of more advanced algorithms and a mandatory exposition of the classic Q-learning algorithm.

Q-learning [11] stands out as one of the most popular and successful algorithms from the VB family. It is an easy to implement and quite effective algorithm for the majority of problems where an agent seeks to find the best actions to solve a relatively small and deterministic environment. It is a tabular method, since it stores its experience in a table, specifically a Q-value for each state-action pair $Q(s, a)$ in the agent's state-action space. It is also an off-policy algorithm in the sense that it could learn a policy different from the behavior policy being executed. Briefly, being off-policy means that it could follow another greedy behavior, thus converging in its direction independently of its own policy's chosen actions. If total exploration and a reasonable amount of visits for all state-action pairs (if all pairs continue to be updated through learning), we have correct convergence to the optimal policy Q^* . Until convergence or termination, Q-values are

¹²In other words, a connection between optimal policies derived by optimal action-value functions and optimal state-value functions

updated according to¹³:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a_{t+1}} Q^-(s_{t+1}, a_{t+1}) - Q(s_t, a_t)). \quad (2.15)$$

Proceeding to deep RL methods, DQN [24] is another successful and popular choice when one wants to fuse Q-learning with deep neural networks, hence its name Deep Q-Networks. Q-learning suffers from not being feasible¹⁴ to more high-dimensional and complex problems involving huge state-spaces, hence the necessity of extending its capabilities to work with function approximation and the creation of more powerful variants like DQN. DQN is a classic¹⁵, and it is an end-to-end algorithm hugely applied to dynamically complex domains using only raw high-dimensional data as input. It is most widely known to solve and reach super-human and near-human performance in a variety of Atari Games, with different levels of complexity, but it also showed impressive capabilities of being applicable to other types of tasks¹⁶ (e.g., Decision Making, Multi-Agent RL, Continuous Control, etc.). The common architecture consists of a deep convolutional neural network that performs, in a first phase, 2D convolutions across the input image, extracting the most relevant features¹⁷ from the input. In a second and final-phase, those relevant features are fed directly to feed-forward layers, where, ultimately, in the output layer, there are all the available Q-values for each action available for the agent¹⁸. DQN was created in 2013, but it remains strongly used and mentioned in research¹⁹.

The progress and the majority of the research in Value-Based RL throughout the years, consisted mainly of improvements over the DQN method (overcoming its frailties²⁰). DQN highly suffers from a phenomenon called Maximization Bias[25]: it is an overestimation of action-values caused by the max operator in the target estimate y^{DQN} produced by the target network $Q(s, a; \theta^-)$, as shown in the following expression:

$$y_i^{DQN} = r + \gamma \max_{a'} Q(s', a'; \theta_i^-). \quad (2.16)$$

Double DQN [26] proposes a new target estimate y^{DDQN} , using the target network $Q(s, a; \theta^-)$ for action evaluation, and the policy/behavioral network $Q(s, a; \theta)$ for action selection, as follows:

¹³The "-" superscript on top of the Q^- was used only to reinforce the idea of off-policy learning: the target prediction for the next state, could be taken from a different policy other than Q .

¹⁴It is not feasible in terms of time and memory consumption to construct a table for high-dimensional state-spaces. It is when we want to apply Q-learning to more complex problems, that one chooses, instead, to approximate/estimate a Q-function using powerful function approximation techniques like neural networks.

¹⁵DQN could be easily considered as the most well known deep RL algorithm.

¹⁶<https://paperswithcode.com/method/dqn>

¹⁷Deep Convolutional Neural Networks perform feature extraction autonomously. Hence, there is no need to perform manual feature extraction.

¹⁸The DQN neural network is typically fed with the agent's state as input, outputting the corresponding Q-values for each action. It could also be fed with the agent's state and performed action as input, outputting a single Q-value for the corresponding state-action pair in the input.

¹⁹<https://paperswithcode.com/method/dqn>

²⁰DQN's frailties are specifically: Maximization Bias and limited Policy Evaluation when facing similar action-values in specific situations (when it is better to simply not choose any action, in other words, when all actions are equally reasonable).

$$y_i^{DoubleDQN} = r + \gamma Q(s', \arg \max_{a'} Q(s', a'; \theta_i); \theta^-). \quad (2.17)$$

Double DQN considerably surpassed the performance of vanilla DQN, resulting in a faster convergence with prudent and stable target estimations (reducing the overestimation problem).

Dueling Networks [27] (also called Dueling Deep Q-Networks or simply DDQN), is another improved method on top of DQN principles. The DDQN shares the same convolutional layers architecture from DQN, but has a different fully connected part. The fully connected part is divided into two independent fully connected streams, one to estimate a state-value $V(s)$, and another to estimate the action-advantages $A(s, a)$. Before the output layer, there is also an aggregation module that combines the outputs from each stream. Theoretically, a Q-value can be formalized as the sum of the value of being in a state, with the advantage of taking an action in that same state, that is,

$$Q(s, a) = V(s) + A(s, a). \quad (2.18)$$

It would be logical, this former sum to be executed in the aggregation layer, since we want to obtain $Q(s, a)$. This doesn't happen because of an identifiability issue²¹. In the study, this identifiability was solved with two approaches (defined by both equations 2.21 and 2.22). If we apply a max operator in $A(s, a'; \theta, \alpha)$ as exposed in equation 2.21, the optimal action will be defined as:

$$a^*(s) = \arg \max_{a' \in \mathbb{A}} Q(s, a'; \theta, \alpha, \beta) = \arg \max_{a' \in \mathbb{A}} A(s, a'; \theta, \alpha). \quad (2.19)$$

Therefore, the Q-value for the optimal action will be exactly equal to the state-value, as expressed in equation 2.20. With this first approach we then have $V(s, a, \theta, \beta)$ identified for the optimal action and the $A(s, a, \theta, \alpha)$ for the other actions.

$$Q(s, a^*; \theta, \alpha, \beta) = V(s; \theta, \beta). \quad (2.20)$$

The second approach applied an average across the advantage estimates. This approach removes the identifying semantics for V and A (because now they are subtracted by a scalar, and they are not really V and A), but, in practice, it showed a more stable optimization performance²².

With a state-value function decoupled from an advantage function, DDQN has the knowledge about an estimated state-value, and the sensibility to recognize how much valuable is to perform actions in a given state, in other words, the advantages. There are situations in which it is not worth to act, therefore, this network can sense those situations, having an interesting expressiveness²³. To

²¹If we sum both values to get Q , we cannot recover V and A , uniquely. Empirically, this resulted in poor practical performance. If there is no identification for V and A , the backpropagation process will not differentiate both estimates.

²²In practice, the advantages change as fast as the mean, hence they don't need to follow the optimal action's advantage that happens in 2.19.

²³This network is capable of estimating the real value of a state, without having to calculate the Q-values for each action in that state. With this decoupling of the Q value between V and A , we get more reliable and independent estimates for what an agent really wants to know in a state: the value of a state and the advantage of performing actions in that state separately.

conclude, let $V(s; \theta, \beta)$ be the state-value stream function, $A(s, a; \theta, \alpha)$ be the advantage-values stream function and $Q(s, a; \theta, \alpha, \beta)$ (α and β are the independent parameter vectors from the advantage-values function and the state-value function, respectively and θ is the parameter vector from the convolutional layers) be the complete parameterized DDQN Q-function. Lastly, the outputted Q-values from DDQN are given by:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \max_{a' \in |A|} A(s, a'; \theta, \alpha)), \quad (2.21)$$

and,

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \frac{1}{|A|} \sum_{a'} A(s, a'; \theta, \alpha)). \quad (2.22)$$

DDQN improves its Q-value estimation only by proposing a change in its architecture, sharing the same algorithm. DDQN surpassed the performance of DQN and even Double DQN across the same benchmark array of Atari Games.

2.1.1.3 Actor-Critic

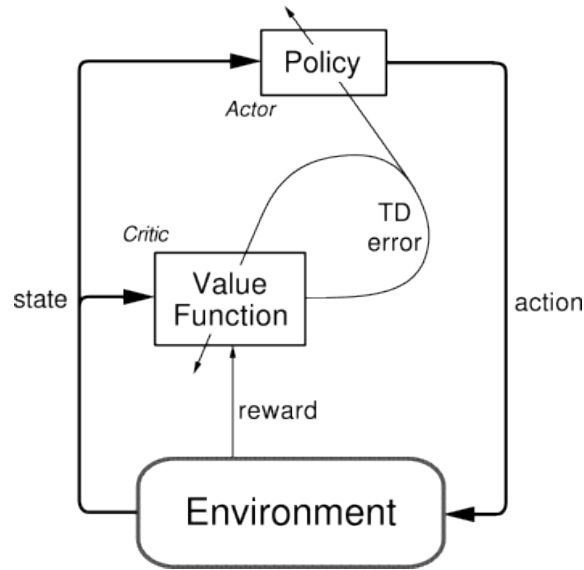


Figure 2.3: A typical Actor-critic architecture. Available at: <http://incompleteideas.net/book/first/ebook/node66.html>

Actor-Critic (AC) methods, as introduced in subsection 2.1.1.1, have the property of learning both a value function and a policy function. The actor is used to select actions, and the critic is used to "criticize" the actor's choices improving the evaluations performed by solitary actors in PB methods. It combines properties from the former PB and VB methods (described in subsubsections 2.1.1.1 and 2.1.1.2, respectively), thus it is considered as an hybrid method: it learns a **critic** in the form of a parameterized state-value function $V(s, w)$ or an action-value function $Q(s, a, w)$ (depending on the algorithm) with parameters w . It also learns an **actor** in the form of a parameterized policy function $\pi(a|s, \theta)$ with parameters θ .

AC methods seek to maximize the performance of the policy, using a learnt critic estimate to evaluate a transition. Commonly, this critic's estimate²⁴ appears in the form of a Q function $Q_w(s, a)$, an Advantage function $A_w(s, a)$ or a TD-error δ . The equation 2.25 exposes an AC actor's parameter vector update, using an advantage critic evaluation (we consider an independent learning rate α^θ for the actor's update). This update is performed using gradient ascent because we want to maximize the actor's objective function $J(\theta)$. More precisely,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(a_t | s_t) A^w(s_t, a_t)]. \quad (2.23)$$

The advantage function estimate²⁵, can be derived as follows:

$$\begin{aligned} Q(s_t, a_t) &\approx r_{t+1} + \gamma V(s_{t+1}) \\ A(s_t, a_t) &= Q_\phi(s_t, a_t) - V_w(s_t) \\ A(s_t, a_t) &\approx \mathbb{E}_{\pi_\theta} [r_{t+1} + \gamma V_w(s_{t+1} | s_t, a_t)] - V_w(s_t) \\ A^w(s_t, a_t) &= r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t). \end{aligned} \quad (2.24)$$

The intuition to use advantages is to update the gradients in the direction of the advantage function. This direction tells us what is the advantage of taking a_t in s_t compared to the the value of being in s_t . If the advantage is positive, then we want to increase the probability of taking a_t in the future, and vice-versa,

$$\begin{aligned} \Delta \theta &= \theta_{t+1} - \theta_t \\ &= \alpha^\theta \nabla_\theta J(\theta). \end{aligned} \quad (2.25)$$

Additionally, advantage AC methods seek to minimize the error of the predictions of the critic, thus they perform an update of the critic's parameters using an approximated TD-error²⁶ as shown in equation 2.26, that is,

$$\begin{aligned} \mathbb{E}_{\pi_\theta} [\delta_t | s_t, a_t] &= A^{\pi_\theta}(s_t, a_t) \approx r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t) \\ \delta_w &= r_{t+1} + \gamma V_w(s_{t+1}) - V_w(s_t) = A^w(s_t, a_t). \end{aligned} \quad (2.26)$$

This update is performed with gradient descent because we want to minimize the critic's objective function $J(w)$,

$$\nabla_w J(w) \approx \nabla_w V_w(s_t) \delta_w. \quad (2.27)$$

²⁴<https://www.davidsilver.uk/wp-content/uploads/2020/03/pg.pdf> - on the last slide there are all the possible critic estimates.

²⁵It may seem that one needs two critics with parameters ϕ and w , but if one learns a critic state-value function, one only needs one critic

²⁶While following policy π_θ , the TD-error is approximately equal to the advantage function if one performs a_t in s_t . Proof in the David Silver's teaching slides at: <https://www.davidsilver.uk/wp-content/uploads/2020/03/pg.pdf>

The critic's parameter vector update, using an independent learning rate α^w is given by:

$$\begin{aligned}\Delta w &= w_{t+1} - w_t \\ &= -\alpha^w \nabla_w J(w).\end{aligned}\tag{2.28}$$

There are various different types of AC algorithm in the literature showing impressive results in a widely amount of applications (e.g., Atari games, simulated physics tasks and robotics tasks). They evolved while overcoming some common problems from vanilla AC algorithms specifically: correlation between samples during online learning, slow learning and huge sample inefficiency. Following, we briefly describe some of the most well known Actor Critic state-of-the-art methods:

DDPG [28] is one of the most well known off-policy²⁷ Deep RL AC algorithms. It is inspired by DQN [29] ideas²⁸, but with the new capability of working in the continuous action domain. It is capable of robustly solving a huge amount of simulated physics tasks (e.g., legged locomotion, dexterous manipulation, cartpole swing-up and car driving) and Atari games, maintaining the same network architecture, hyperparameters and learning algorithm. It is capable of learning end-to-end policies, meaning that it can work with high-dimensional input raw data. It surpasses the performance of planning algorithms, with model dynamics knowledge, and DQN in terms of training time for the same performance.

PPO [30] is another well known Deep RL AC algorithm that has the stability and reliability of trust-region methods [31] but it is much simpler to implement, has better performance and better sample efficiency. To date, it is the best state-of-the-art algorithm in continuous control tasks outperforming other AC algorithms on Atari Games and MuJoCo²⁹ tasks. Due to its ease of use, excellent performance and easy tuning, PPO has become the default RL algorithm at OpenAI³⁰.

SAC [32] is a successful and well known Deep RL AC algorithm and the first off-policy actor-critic method in the maximum entropy reinforcement learning framework. Standard RL has the objective of maximizing the expected sum of rewards. Maximum entropy RL considers a more general objective: not only an agent needs to maximize the expected sum of rewards but also has to maximize its policy's entropy. The idea behind this framework is to make the agent to act as randomly as possible but still collecting high rewards. The former idea brings huge advantages, that are applied, not only to simulated environments, but also to real-world behaviors. Policies trained with maximum entropy will be more robust, because, due to the fact that they can tolerate highly random behavior during training, they can also handle unexpected and random perturbations during test time³¹. This robustness aspect is essential if we want to apply learned policies to the real-world randomness. SAC can not only learn off-policy (being more sample efficient than other on-policy AC methods) but it can also overcome the convergence frailties from

²⁷Off-policy algorithms have the ability of learning from past experiences, thus they spend less samples to learn because they reuse them. Hence, Off-policy Actor-Critic algorithms tend to be naturally more sample efficient.

²⁸Like DQN, DDPG also uses stable learning techniques (e.g., frozen target networks and replay memory) to overcome divergence and correlations that happen when one combines RL with Deep Neural Networks.

²⁹<http://www.mujoco.org/>

³⁰<https://openai.com/blog/openai-baselines-ppo/>

³¹<https://ai.googleblog.com/2019/01/soft-actor-critic-deep-reinforcement.html>

other off-policy AC methods [33]. Not only has surpassed other AC algorithms in simulated tasks but has also been successfully applied to real-world robotics tasks and its capable of learning the policy from real-world samples[34] in modest learning durations.

2.1.2 Model-Based

Model-Based (MB) is a class of RL methods concerned with planning and model-learning. MB methods differ from MF methods mainly due to the agent ability to predict what is going to happen in the environment in response to its actions. Model-Based methods have the uniqueness of having the access to a model (learned or given) and the ability of using it to learn an optimal policy.

As they have access to a model, these methods are much more sample-efficient³² because they can learn, not only by interacting with the real-world but also by simulating experience inside the inner model, therefore not needing to spend that much environment samples. The Figure 2.4 gives insight about the strong advantage of model-based methods on sample-efficiency compared to other RL methods and gradient-free methods.



Figure 2.4: Sample-efficiency spectrum in RL; Available at: https://medium.com/@jonathan_hui/rl-model-based-reinforcement-learning-3c2b6f0aa323

Since we already introduced MF methods, it is already appropriate to meditate about one fundamental aspect in which Model-Free and Model-based essentially differ, that is the way how real experience is approached by each of them. MF methods use real experience to improve directly value functions, policy functions, or both, while MB methods use it for model-learning, thus by making the inner model more approximately close to the real model. MB methods are also classified as indirect RL, in the sense that they don't improve their value/policy functions by direct interaction with the environment but instead interacting with simulated experience from its own inner learned model. Figure 2.5 summarizes the former explanation.

³²A concise article about Model-Based RL: https://medium.com/@jonathan_hui/rl-model-based-reinforcement-learning-3c2b6f0aa323

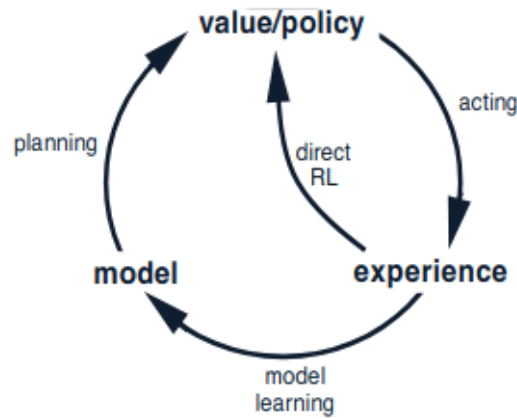


Figure 2.5: Model-Free RL, also known as direct RL, directly uses experience to optimize value/policy functions. Model-Based RL uses experience for model-learning, and indirectly, by planning, they optimize value/policy functions. Figure taken from chapter 8 of Sutton-Barto’s Book [1].

Model-Based agents are essentially divided into two categories. They could:

- Given a model (with access to a known model), use planning to learn a global value and/or policy function.
- Learn a model, focusing on both learning a model and a global value-function and/or policy function. This class of MB could be considered the most complete in terms of capabilities an agent could have.

In the following subsections, we present some examples of well known methods in Model-Based RL. We give a brief overview over state-of-the-art Model-Based RL, not entering into much detail since the main focus of this dissertation revolves around Model-Free methods.

2.1.2.1 Given a model

Model-Based RL could be also applied without the intermediate step of learning the model of the environment they interact with. Additionally, it could learn policies or value functions by acting on a given model, thus by applying tree-search techniques [35] to plan ahead and learn the best policies in a given/known environment.

DeepMind worked on several MB models that, by having access to the rules of the game³³, successfully learned board games from scratch achieving superhuman performances.

Around 2015, DeepMind released a program named AlphaGo that combines DRL with tree search techniques learning from data collected from human games. It was capable of defeating a world-class player of Go by 4 games to 1, named Lee Sedol³⁴, in a match that was watched by an

³³What differs this subsection from the following subsection 2.1.2.2 is the willing that an RL agent has to learn the dynamics of the environment, to build an inner model and learning policies and value functions inside of it. In this subsection, the model is already estimated and given to the agent.

³⁴https://en.wikipedia.org/wiki/Lee_Sedol

estimated 200 million viewers. Lee Sedol announced his retirement from his professional career due to the increasing dominance of AI, referring AI as "an entity that cannot be defeated".

In 2017, AlphaGo Zero[36] was released as an improved version of the 2015 program AlphaGo with the distinction of being able to learn by playing against itself, a skill named Self-Play³⁵. Curiously it beat the previous AlphaGo program by 100 games to 0.

Following AlphaGo Zero, AlphaZero[37] arrived in 2017 as a more generalized variant of AlphaGo Zero, with the goal of mastering the game of Go as well as other table games like Chess and Shogi. It learns also by playing against itself and from scratch like its previous versions. It achieved within 24 hours a superhuman level of play in the games of Chess, Shogi and Go and convincingly defeated a world-champion program in each game.

2.1.2.2 Learn a model

Learning a model consists essentially in observing the unknown real environment and learn its dynamics using function approximation techniques³⁶. Naturally, the environment to learn is irreversible, and the learned model of the environment is reversible, hence it could be used for planning. In other words, a model of the environment could be seen as a memory representation of the most interesting aspects of the dynamics of the real environment. Following we will expose some classic and state-of-the-art Model-Based, that learn a model, methods:

Referring Dyna-Q [38] is a gentle and classic introduction, since it is a conceptual Model-Based algorithm that both learns how to act and at the same time learns a model of the environment. Dyna-Q is a classic RL algorithm capable of integrating model-learning with planning and value/policy function learning. Specifically, tabular Dyna-Q, integrates Q-learning with model-learning assuming deterministic environments. The process of Dyna-Q is straight-forward and consist of: acting on the environment and collecting experience, updating its Q estimates, based on the observed rewards r_{t+1} and the observed next states s_{t+1} . Both the rewards and the next states are associated with the states s_t and actions a_t that caused them. Hence, each dynamic transition is saved on the tabular model ($Model(S = s_t, A = a_t) \leftarrow (r_{t+1}, s_{t+1})$). An n hyperparameter controls the number of iterations of the simulated experience the Dyna-Q agent will perform in its own inner learned model. The bigger the n the more learning iterations the agent has in the learned model. Dyna-Q showed huge improvements in learning speed and sample efficiency compared to normal Q-learning (with no planning), proving the improved capabilities of integrating learning and planning in the same algorithm.

Additionally, another interesting method to refer is World Models (WM) [39]. WM is a Model-Based RL algorithm that seeks to build an agent that is capable of learning a model and learn a policy inside of it. Inspired by our own cognitive system a WM agent integrates three components: A vision system capable of capturing visual information (images) and encoding it into a small

³⁵<https://openai.com/blog/competitive-self-play/>

³⁶These techniques could be parametric or non-parametric. In the context of deep RL the common chosen approach is parametric using neural networks. We are not considering hugely memory demanding techniques like tabular methods.

compressed representation. A memory component capable of predicting future observations based on historical (past) compressed information. A controller component capable of integrating the information built by the memory and the vision systems and decide which action to take according to it. The WM method showed interesting results on a car racing task and in the game VizDoom while learning in its own inner model, or, as said in the study, in its own dream/hallucination. The learned behavior showed considerable results when tested on the original virtual environments confirming the robustness of the model-learning of WM.

Lastly, MuZero [40] is another state-of-the-art model-based RL method that learns a model that it is worth the mention. This model applies end-to-end learning³⁷ and achieves similar or better results compared to its predecessor model-based algorithm AlphaZero [37] (as presented in the previous subsection 2.1.2.1). Contrary to AlphaZero, MuZero did not have access to the rules of the games it was tested in, meaning that it learned the dynamics of the game from scratch. It consists of a deep RL model that incorporates planning, acting and training inside of a learned model. It applies Monte Carlo Tree Search (MCTS) [35] techniques to inner simulate the learned game (planning), ultimately discovering the best actions to take in the real environment (acting). The learning process consists of remembering past experience stored in a memory buffer (training).

2.2 RL and Robotics

The current and recurring challenges of applying RL in robotics as well as recent studies and the state of the art will be discussed in this section.

2.2.1 Curse of dimensionality

The most evident fact when we speak of applying RL to robotics is the natural high-dimensionality of robotics problems. Robotics settings usually have many degrees of freedom (DoF), continuous states and actions and measurement uncertainty (we have to assume that the true state is not completely observable, hence the learning system will not be able to know precisely in which state it is and even vastly different states might look very similar) [9; 41]. Traditional RL approaches essentially consider grid-based representations and discretized state and action spaces which have problems scaling up to the continuous world of robotics. To better picture the state-space's exponential growth of dimensionality, we will provide the following short example: "Suppose we have F features(dimensions) representing the state-space of the robot, and each feature is equally discretized into R levels. Hence, we have R^F states". Hence, it is clear to conclude that in a really high-dimensional and high-resolution environment the representation of a state-space is enormous and thus, impossible to explore effectively and to represent without a huge computational cost.

³⁷End-to-end is an approach where high-dimensional raw inputs are directly learned and mapped to outputs by a single model in a single pipeline, without consulting any third-party. It's end-to-end in the sense that, from one end to the other, autonomously and directly learns to map raw input data to desired outputs.

2.2.2 Function-approximation techniques

As questioned in the article “What to do when state and action spaces explode...literally?”³⁸, we have to find new techniques to overcome the limited tabular and discretized way of representing the problem[42], which consists essentially in the memorisation of each state which is only feasible in terms of memory consumption and computational cost in relatively small state and action spaces.

Function approximation [43] is a family of mathematical and statistical techniques used to represent a function of interest when it is computationally or information-theoretically intractable to represent the function exactly or explicitly (e.g., in tabular form)[41].

RL problems use function approximation techniques on sampled data acquired during the interaction of the agent with the environment. Due to the impossibility of visiting and representing every single state, function approximation techniques are used in this settings as a means to essentially generalize the representation of neighboring states and actions. The aim is to provide enough generalization in the representation of states that have similar features. To summarize, function approximation can be employed to represent policies, value functions and models of the environment, providing to RL the necessary mathematical tools to scale up to the complexity of robotics applications and real-world problems.

To give more insight into the function approximation domain, broadly speaking, there are two kinds of function approximation techniques: **parametric** and **non-parametric**. **Parametric** function approximators (e.g., neural networks, linear basis functions,...) use a finite set of parameters with the goal of finding the best ones to fit the agent’s experienced data. **Non-parametric** function approximators expand representational power and, hence, are not limited by the representational capacity of a chosen finite parametrization [44]. Gaussian process regression [45] is a good example of a common non-parametric method used in RL.

2.2.3 Sim2Real

The wish to train RL agents in simulations with synthetic data is highly motivated by lowering costs and engineering effort [46]. The usual approach is to learn the behaviour in simulation and subsequently transfer it to the real robot. Creating sufficient accurate models of the real-world is, however, pretty challenging because it requires huge physics knowledge and a lot of assumptions. Manual simulated environments used by narrow and low-dimensional tasks are sufficient as we can see in this study [47], but if we are speaking of real complex tasks with high-dimensional features like vision, the gap between simulated and real-world behaviours becomes evident. Other complex mechanical interactions are also obviously hardly modelled accurately (e.g., friction, restitution, dampening factors and air drag). This under-modelling results in discrepancies and errors during the learning process, thus the accumulated discrepancies quickly diverge the simulated behaviour from the real/ideal behaviour. Recent studies showed interesting solutions to reduce this Sim2Real gap by creating useful and creative models [48; 49].

³⁸<https://towardsdatascience.com/function-approximation-in-reinforcement-learning-85a4864d566>

2.2.4 Sample-efficiency

The process of collecting samples from the real-world is one of the major challenges when it comes to robotics. Robot hardware is usually expensive and requires regular and careful maintenance so it is highly desirable to improve sampling efficiency to minimize real-world interaction. As already explained in Section 2.2.3 the wish to train in simulated environments is naturally convenient to improve sample efficiency but it is easily disqualified by its limitations.

A most promising path through sample efficiency is to apply Model-Based RL. Model-Based RL provides the agent the ability of learning a model of the environment thus dramatically reducing the number of interactions it needs with the real-world. This approach, however, can suffer from critical failures due to errors in the learned model. Although model-based RL showed great success in problems where the dynamics of the environment is relatively linear and low-dimensional, in more complex environments model-based approaches are still currently imperfect. Typically, when the learned model is not accurate, during learning and planning, the policies tend to exploit those failures, a process known as model-bias.

To effectively avoid the model-bias problem, a recent paper proposed a solution named STEVE [50] (Sample-Efficient Reinforcement Learning with Stochastic Ensemble Value Expansion) where an agent learns several different models of the environment, and during planning and learning, uniformly samples from these model collection favoring those whose estimates have lower uncertainty, and thus lower error. Another study proposed a different approach named MBPO [51] (Model-Based Policy Optimization) where the agent limits the use of the predictive model to a small-length horizon avoiding the progressive accumulation of uncertainty and error caused by long-horizon prediction. Another approach based on a widely known recent Actor-Critic-based algorithm named SAC [33] (Soft-Actor-Critic) is presented in this paper as a strong candidate to solve real-world applications with sample-efficient learning. In summary, we can conclude that sample efficient algorithms that are able to learn from a small number of trials are essential.

2.2.5 Safety

Robot policies must be safe during both training and deployment in the real world in order to ensure the integrity of the robot, to guarantee the safety of people in the environment and to avoid the violation of critical constraints.

A recent paper [52] proposed a safety layer built on top of the agent's policy to ensure that the robot will not violate the constraints of the environment. This is an alternative to reward shaping: the usual method is to assign huge negative rewards near the workspace's boundaries so the robot will learn that it should not overpass them. This has clear drawbacks according to the study. The results showed that reward shaping always violated the zero-constraint-violations goal (it was impossible to avoid constraint violations during training), the second is the difficulty of choosing the correct parameter of the best M (the distance margin to the boundaries), which depends strongly on the task so it is not generalizable. The experiments showed that the safety layer worked perfectly in all type of different tasks of different complexity.

Another studies suggested mathematical approaches within the control theory domain: (1) A study [53] proposed an agent capable of building Lyapunov Functions³⁹ itself under the formalism of CMDPs⁴⁰ (Constrained Markov Decision Processes) showing better performance than existing baselines in balancing constraint satisfaction and performance; (2) Another study [54] based on CBFs [55](Control Barrier Functions) suggested the use of a CBF-based controller with model-free RL named RL-CBF that showed guarantees in safety with high probability during the learning process, regardless of the RL algorithm used, and demonstrated greater policy exploration efficiency.

2.2.6 R3L

A recent and promising study [56] addresses the three final main challenges to be presented in this section: (1) Reward engineering; (2) Working with Raw Data; (3) Manual Reset. The study proposes a real-world RL system named R3L, a novel system with the ability to learn in the absence of resets, to learn directly from the robot's own sensor information and to learn from the reward inferred by the robot itself. R3L believes in novel systems that require minimal instrumentation and human-effort.

Reset-Free RL typically learns under an episodic paradigm, hence manual reset techniques are needed during simulated or real-world training. Behind successful examples of application of robotics in the real-world, when we are speaking of real-world training, manual-resetting the robot is a recurring need [34] and therefore further studies address the problem by engineering additional layers in the architecture of the agent to overcome the need to manually reset the robot within the training's workspace boundaries[57].

In order to enable scalable and autonomous real-world learning we need systems that do not require an episodic formulation of the learning problem. Reset-Free RL is challenging because when the agent is training it usually gets stuck in certain states and when reaches the goal it stays there, hence reset techniques are needed to free the agent from those situations. R3L proposes the implementation of a randomized perturbation controller which is trained to intermittently introduce perturbations into the system through a policy that is trained to properly explore the environment allowing the agent to attempt the task from a variety of configurations. While the main policy of the robot is trained following VICE-based rewards[58; 59] for reaching the desired goals, this special randomized perturbation controller is trained with an intrinsic motivation objective that encourages visiting under-explored states. This is promising because it removes the need for reset monitoring and provides to agents an ability to autonomously and intelligently explore the state-space and the capacity of freeing themselves from "stuck-situations".

Raw sensory input Typically all of the information necessary for learning must be obtained from the robot's own sensors which require huge and tedious instrumentation and complex state

³⁹<https://mathworld.wolfram.com/LyapunovFunction.html>

⁴⁰https://en.wikipedia.org/wiki/Markov_decision_process#Constrained_Markov_decision_processes

estimation. A real world system typically does not have access to the underlying state information and instead needs to learn from high dimensional sensor inputs.

To address this fact, R3L aims to train Variational Auto Encoders (VAE)[60] in order to compact the information contained in the high-dimensional-input-space to be further used to train the policy in a low-dimensional latent space⁴¹. This is promising in robotics because it opens the door to scalable robotic systems, that work directly with raw sensory information, and to reduce the need of demanding state estimation along, feature extractors⁴² and real-world instrumentation.

Reward inference The design of a proper reward function to guide the algorithm to success is, in vast majority of applications in robotics, manually programmed by the user, a process called reward engineering[61; 62]. While this works in simulated scenarios where we have access to all ground truth information, in the real world it is harder to implement because it requires a lot of instrumentation to keep track of the information required by the reward function. In the majority of high-dimensional problems manually inferring the right reward function is highly complex and based on a lot of assumptions: reward function design needs to be sufficiently encapsulated so as to avoid unwanted behaviour (e.g., negative side effects, reward hacking, ...)[63; 16].

R3L suggests the implementation of a goal classifier, in other words, a data-driven reward specification framework called variational inverse control with events (VICE)[58; 59] that is capable of distinguishing successes from failures by mixing a small initial dataset of goal examples provided by the user with on-policy samples (successes and failures) obtained during training. This model classifier receives raw camera images, representing robot current observations, as inputs, and learns to evaluate them outputting a scalar value which represents the likelihood of success as reward. If robots gain the ability to infer the rewards themselves, the tedious and inefficient hand-designed reward functions will eventually be abandoned.

⁴¹<https://towardsdatascience.com/understanding-latent-space-in-machine-learning-de5a7c687d8d>

⁴²<https://robotacademy.net.au/masterclass/feature-extraction/>

Chapter 3

Reinforcement Learning

In this chapter we formally present the necessary background and fundamental concepts from the bottom to the top, to support the chosen method Q-learning.

3.1 RL Framework

A description of the RL Framework will be made in this section. It is made a more conceptual and fundamental exposition of basic concepts related to RL to establish the needed foundations.

3.1.1 Agent-Environment Interaction

In a high-level view, RL is described as the learning process of a behavior through interaction to achieve a goal [64]. The one who learns is an agent. In each time step t , the agent interprets its situation by observing its current state (S_t). With respect to S_t , the agent chooses an action (A_t) according to its actual¹ learned behavior (π_t), that is, $A_t = \pi_t(A_t|S_t)$. After each performed action, the environment responds with a feedback scalar reward signal (R_{t+1}), representing the immediate value of the action performed². Lastly, the agent learns from the interaction and it continues the process, starting with the next observation of the next state (S_{t+1}). The former description is supported by the illustration in the Figure 3.1. To conclude, this ongoing interaction is what describes the learning process of RL.

¹The subscript of t in the policy π_t represents the agent's behavior at timestep t . The agent chooses an action according to its current learned policy, meaning that it could change over time.

²As a convention we assume that the reward signal is received in the subsequent timestep, that is, $t + 1$.

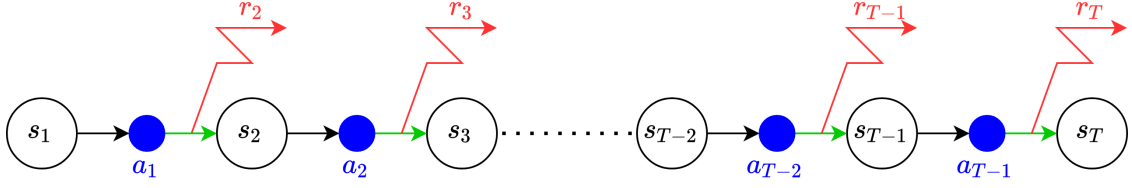


Figure 3.1: Illustrative view of the sequence of trajectories generated by the agent-environment interaction. For convenience, we represented a finite interaction, in other words, an episode, finishing in state s_T .

3.1.2 Rewards and Returns

The standard RL framework seeks to formalize the goal of the agent using reward signals. A reward is, simply, a scalar number, that is $R_t \in \mathfrak{R}$. The agent's main objective consists of the maximization of the total amount of reward it receives. This maximization is described not in respect to the immediate rewards, but the cumulative reward in the long run (taking the best decisions through time). Hence, the goal of RL is defined by finding a policy that maximizes the expected cumulative reward until a final timestep T , as follows:

$$\max_{\pi} \mathbf{E} [R_{t+1} + R_{t+2} + \dots + R_{T-1} + R_T]. \quad (3.1)$$

The cumulative reward, is commonly named the maximum return G_t , that is,

$$G_t \doteq R_{t+1} + R_{t+2} + \dots + R_{T-1} + R_T. \quad (3.2)$$

The final timestep (T) convention, only makes sense in applications where we can clearly define a temporal limit, in other words, when the task at hand could be broken in episodes. The tasks that are defined by episodes are called episodic tasks. This timestep T is logically variable, because each episode has its own duration. The other category of tasks are called continuing tasks, and they are not naturally identifiable by episodes. Mathematically, a continuing task is defined by an infinite final time-step ($T \rightarrow \infty$), hence the mathematical formulation of the return needs to protect the cumulative reward to not sum up until infinity. A mathematical trick called *discounting* is done to assure convergence, and to generalize the expected return to both the former described tasks. The discounting trick is done using a parameter named *discount rate*, that is $\gamma \in [0, 1]$. Hence, the general³ return formulation is bounded, as long as the return sequence $\{r_k\}$ is bounded, and is defined by:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \quad (3.3)$$

Conceptually, the *discounting* is not only a mathematical trick, but it also describes the uncertainty we have about the reward in the distant future, and also, it is inspired by the biological decision

³It generalizes the return expectation, assuring convergence for continuing tasks, but without affecting the return expectation for episodic tasks (with $\gamma = 1$ it goes back to the naive and not general approach).

making and cognitive models found in animals and humans when it comes to the preference for proximal rewards⁴.

We can realize the importance of this parameter in the learning process. When one chooses short discount factors ($\gamma \rightarrow 0$), the agent turns to be more "myopic", being concerned with maximizing expected immediate rewards. On the other side, when one chooses large discount factors ($\gamma \rightarrow 1$), the agent turns to be more farsighted, giving importance about future rewards more strongly. The choosing of this γ value is dependent on the performance one wants to achieve in the task at hand and it is generally tuned by trial-and-error.

To conclude, to note that, traditionally, the reward function $R(s, a)$ is hand-designed by the programmer⁵, hence the exercise of designing a proper reward function to help the agent achieve the task is extremelly challenging requiring a lot of domain knowledge and intuition. In theory, good reward function design is normally characterized by reward functions that communicate the agent *what* they need to achieve rather than *how* they need to achieve a goal [65]. Bad and careless design could misdirect the agent's learning process, causing undesired or limited behavior.

3.1.3 Finite Markov Decision Process

If the problem we want to solve can be described as a Markov Decision Process (MDP), then RL can be used to solve it. MDPs are used when we want to model a decision-making problem. The final goal of an RL agent is to learn how to achieve a goal through interactions with the environment, in the form of decisions. In chapter 1 and in subsection 3.1.1 we briefly introduce the recurrent interaction, but in this section, let us take the chance to explain it with more detail.

As already described in subsection 3.1.1 and, as already illustrated in the Figure 1.1, the MDP structure of the RL problem is sustained by an agent-environment interaction, where an agent is continuously making decisions and the environment is continuously giving feedback. The agent seeks to maximize these rewards over time by, through learning, finding the best decisions (choosing the best actions) for the situations it faces.

The agent's interactions are sequentially described over a discrete time sequence $t = 0, 1, 2, 3, \dots$. At each time step t , the agent is provided with some representation of the environment $S_t \in \mathbb{S}$. The agent reasons about the current situation S_t and chooses an action $A_t \in \mathbb{A}(s)$. The action A_t produces an effect in the environment making it respond with a new representation S_{t+1} and a reward signal $R_{t+1} \in R(s, a) \subset \mathfrak{R}$. This interaction repeats continuously generating trajectories, as follows:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots \quad (3.4)$$

An MDP, when finite, in other words a Finite Markov Decision Process (FMDP), has a state-space S , an action-space A and a reward set R with a finite number of elements. With this finitude, the model of a reliable FMDP consists of a well defined discrete probability distribution of transiting

⁴As explained by David Silver in one of his lectures, here: <https://youtu.be/lfHX2hHRMVQ?t=1116>. This information can be also found in slide 13 of Markov Decision Process David Silver's lectures, here: <https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf>

⁵Designing, implementing and tuning a reward function is a process called reward engineering.

to a new state S_{t+1} and receiving a reward R_{t+1} after executing action A_t while in a state S_t for each time step t (meaning that this probabilities could change over time) for all $s', s \in \mathbb{S}$ and $a \in \mathbb{A}(s)$. The dynamics of a finite MDP is a probability distribution and it is formally represented as follows:

$$p(s', r|s, a) \doteq \Pr\{S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a\}. \quad (3.5)$$

The dynamics function p is a four argument deterministic function $p : S \times R \times S \times A \rightarrow [0, 1]$ that returns a probability distribution of transiting to a new state s' and receiving the reward r for each state s and action a . Formally:

$$\sum_{s' \in \mathbb{S}} \sum_{r \in R} p(s', r|s, a) = 1, \forall s \in \mathbb{S} \text{ and } a \in \mathbb{A}(s) \quad (3.6)$$

We can conclude from equation 3.6 that the probability of the transition dynamics for S_{t+1} and R_{t+1} is fully characterized by the information given by the only the previous state s and action a and not by an history of previous states and actions. It is a restriction that the state encodes the information about everything that happened in past interactions in which it could make a difference for the future. This information aspect from the state is called the *Markov property*⁶, and we assume it for the sake of this section's exposal and through the entire chapter. To note that there are another types of MDPs⁷, but it is convenient to expose FMDPs since they present all the needed aspects for the description of the RL Framework.

Knowing p , we can derive the expected immediate reward for any state-action pair (s, a) as follows:

$$r(s, a) \doteq \mathbf{E}[R_{t+1} | S_t = s, A_t = a] = \sum_{r \in R} r \sum_{s' \in \mathbb{S}} p(s', r|s, a). \quad (3.7)$$

We can also derive the expected state-transition probability function, that defines the probability of transiting to a new state s' given s and a , defined by:

$$p(s'|s, a) \doteq \Pr\{S_{t+1} = s' | S_t = s, A_t = a\} = \sum_{r \in R} p(s', r|s, a). \quad (3.8)$$

The following Figure 3.2 illustrates every aspect described previously.

⁶A definition available at: <http://incompleteideas.net/book/first/ebook/node32.html> . Another brief description is in Sutton Barto's book, in [66]

⁷Other type of MDPs are exposed briefly in the following slides (49-57): <https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf> . However, the common approach to describe RL is with the use of Finite Markov Decision Processes.

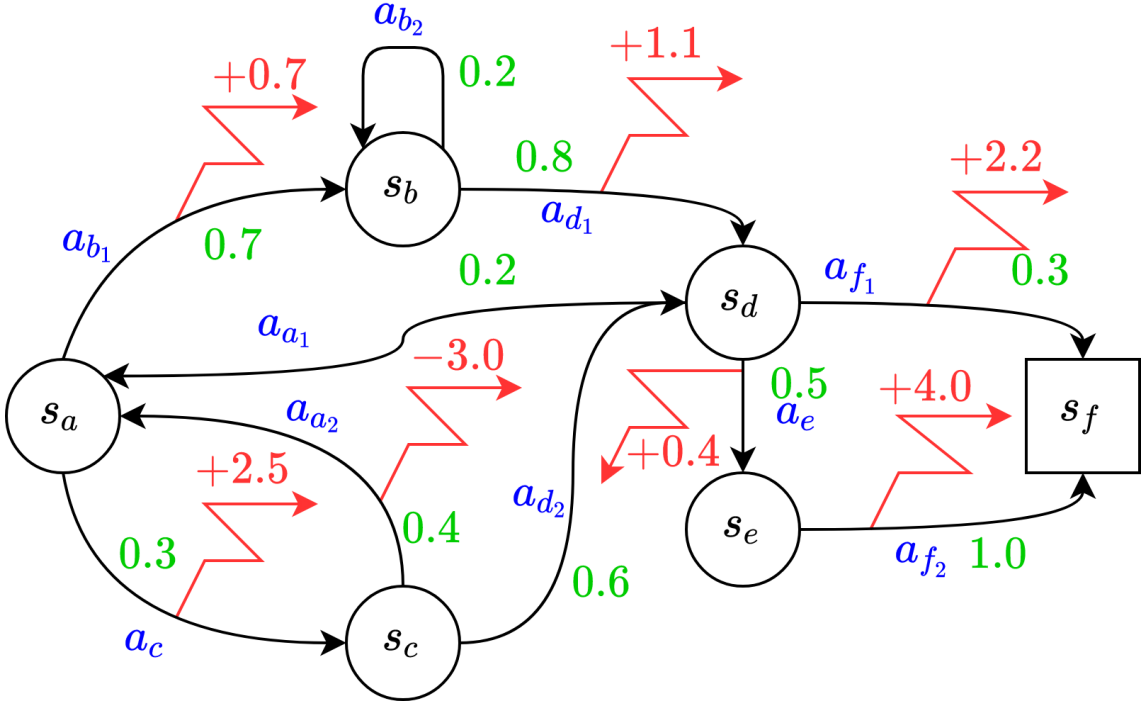


Figure 3.2: An example illustration of a Finite Markov Decision Process. The circles in **black** represent the states of the FMDP, that is: $S = \{s_a, s_b, s_c, s_d, s_e, s_f\}$. In **green**, there are represented the transition probabilities for each state-action pair: $p(s'|s, a) = \Pr\{S_{t+1} = s' | S_t = s, A_t = a\}$ (e.g., $p(s_b|s_a, a_{b1}) = 0.7$), since the environment is stochastic. In **blue** we have the actions available for all states $a \in \mathbb{A}(s)$ (e.g., $A(s_a) = \{a_{b1}, a_c\}$). For the sake of simplicity we assume an implicit equiprobable policy for the actions in each state, that is, $\pi(a|s) = \frac{1}{|A(s)|}, \forall s \in \mathbb{S}$. In **red** we have the reward signals received for each transition, that is, $r = R(s, a)$ (e.g., $R(s_c, a_{a2}) = -3$).

In the next subsection we will focus on how to effectively solve an MDP by defining the agent's behavior, also known as policy, and how the agent measures the quality of its behavior using value functions.

3.1.4 Policies and Value Functions

Until now, the term *behavior* was just conceptually and briefly exposed. Technically, the behavior of the agent is fully defined by a probability distribution⁸ over actions given states, that is,

$$\pi(a|s) = \mathbf{P}[A_t = a | S_t = s]. \quad (3.9)$$

Assuming that a state describes everything that the agent needs to know about its current situation⁹, an action is chosen according to its likelihood probability given the state.

⁸Stochastic policies are more commonly exposed, because they are more generally applied (taking into account the uncertainties of the environment), in contrast to deterministic policies where the output is a direct mapping to an action instead of a probability distribution over actions, that is, $\mu(s) = a$.

⁹As already mentioned previously, if the state has the Markov Property.

It is worth to mention that, the effective probability transition from one state to another, is a combination between something the agent controls (its decisions) and something the agent can not control (the environment). In other words, the probability of transiting from a state s to the next state s' is defined by a sum of products between the environment's transition dynamics probability function ($\mathbf{P}_{ss'}^a = p(s'|s, a)$) and the policy function for each possible action, as follows:

$$\mathbf{P}_{s,s'}^\pi = \sum_{a \in \mathbb{A}} \pi(a|s) \mathbf{P}_{s,s'}^a \quad (3.10)$$

Similarly, the effective reward received while following a policy is a sum of products between the reward's transition dynamics function and the agent's policy function, as follows:

$$\mathbf{R}_s^\pi = \sum_{a \in \mathbb{A}} \pi(a|s) \mathbf{R}_s^a \quad (3.11)$$

Common to all RL methods, in order to learn and improve a policy, the agent needs to know how good is its policy in any given time. The way the policy changes with respect to the experience is what distinguishes each RL method. The value of a policy is characterized by a value function, meaning that, a policy is as good as the amount of expected future reward it can get. The rewards the agent gets depend on the choice of its actions and so, it is the policy that will define the cumulative reward received. The value function of a policy (v_π) is generally defined as the expected return the agent could get, given a state and following the policy π forever after, named state-value function, that is,

$$v_\pi(s) \doteq \mathbf{E}_\pi [G_t | S_t = s] = \mathbf{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \forall s \in \mathbb{S}. \quad (3.12)$$

Besides the state-value function, another special type of value-function is the action-value function q_π , used in Model-Free settings (exposed in subsection 2.1.1), defined by the expected return for taking an action a from a specific state s and following policy q_π forever after:

$$q_\pi(s, a) \doteq \mathbf{E}_\pi [G_t | S_t = s, A_t = a] = \mathbf{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]. \quad (3.13)$$

The rule of an RL agent is to solve the MDP it is in, hence there is one fundamental equation in RL that allows it to do it, that is the **Bellman Equation**. This equation suggests an intuitive and useful idea that will help us improve our policy and value functions. It basically tells us that the value of a state is composed by two parts:

1. An immediate reward: R_{t+1}
2. A discounted value of a successor state: $\gamma v_\pi(S_{t+1})$

First, we will focus on a specific type of Bellman Equation, that is the Bellman Expectation Equation, because we are expecting some return subjected to some policy. Using what we already

presented from equations 3.12 and 3.13 the Bellman Expectation Equation is defined by breaking the term G_t into the two parts described previously, that is,

$$\begin{aligned}
 v_\pi(s) &\doteq \mathbf{E}_\pi [G_t | S_t = s] \\
 &= \mathbf{E}_\pi [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s] \\
 &= \mathbf{E}_\pi [R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) | S_t = s] \\
 &= \mathbf{E}_\pi [R_{t+1} + \gamma G_{t+1} | S_t = s] \\
 &= \mathbf{E}_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s].
 \end{aligned} \tag{3.14}$$

Similarly, we can define a Bellman Expectation Equation for the action-value function, as follows:

$$q_\pi(s, a) = \mathbf{E}_\pi [R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]. \tag{3.15}$$

Once we have Bellman Equations we can use them to solve the MDP and to discover the action and state values by doing backups while looking at the future¹⁰. Value functions follow recursive relationships, hence the operation of finding current values is performed using a backup process using the known dynamics of the environment, the policy, and the discounted value of the successor state. Before speaking about the backup process, it is useful to expose the relationship between the state-value function and the action-value since they are directly connected. A state-value is equal to the weighted sum over all action-values, that is,

$$v_\pi(s) = \sum_{a \in \mathbb{A}} \pi(a|s) q_\pi(s, a). \tag{3.16}$$

The former defined relationship from v_π to q_π is supported by the Figure 3.3, as follows:

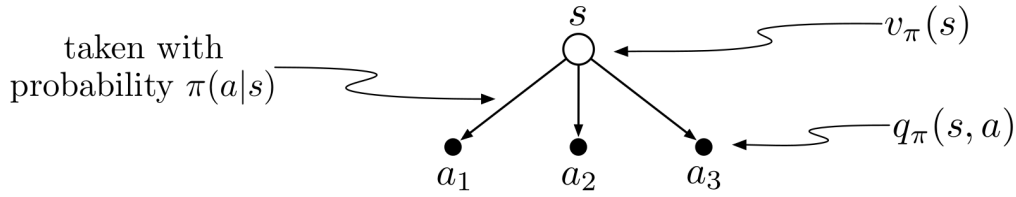


Figure 3.3: Relationship between v_π and q_π . It is an weighted sum over the possible action-values in s weighted by the policy. Figure taken from [2].

On the other hand, an action-value function is equal to the immediate reward plus a discounted weighted sum over all successor state-values, defined by:

$$q_\pi(s, a) = \mathbf{R}_s^a + \gamma \sum_{s' \in \mathbb{S}} \mathbf{P}_{s,s'}^a v_\pi(s'). \tag{3.17}$$

¹⁰Several techniques estimate current value functions using different types of lookaheads in the future (having or not access to a model), namely Monte-Carlo, Temporal-Difference Learning, Dynamic Programming and Exhaustive Tree Search.

The former defined relationship from q_π to v_π is supported by the Figure 3.3, as follows:

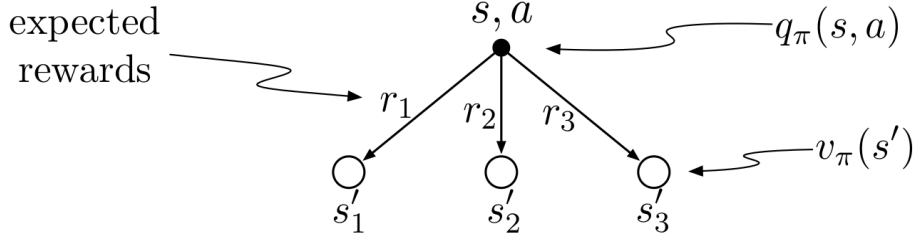


Figure 3.4: Relationship between q_π and v_π . It is a sum between the immediate reward and an weighted sum over the state-values weighted by the transition dynamics of the environment. Figure taken from [2].

Putting it all together, and merging the former exposed relationships, to compute a state-value, depending on successor state-values (as a recursive relationship), we backup the possible successor state-values weighting them with both the policy and the transition dynamics of the environment. There is some weighting of choosing action a while in s , that is $\pi(a|s)$. Next, there is another weighting of transiting from s to any s' after taking some a , that is $\mathbf{P}_{s,s'}^a$. Finally those weighted values are all summed and backed up, computing $v_\pi(s)$, as follows:

$$v_\pi(s) = \sum_{a \in \mathbb{A}} \pi(a|s) \left(\mathbf{R}_s^a + \gamma \sum_{s' \in \mathbb{S}} \mathbf{P}_{s,s'}^a v_\pi(s') \right). \quad (3.18)$$

To support this explanation, we present the complete v_π backup diagram, as shown in the Figure 3.5, for the state-value's computation $v_\pi(s)$, as follows:

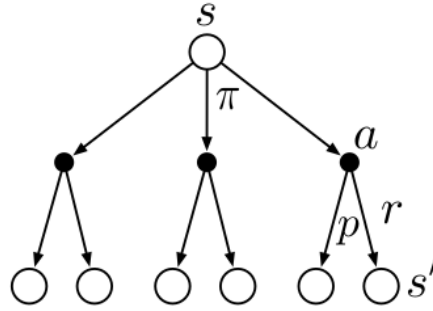


Figure 3.5: The state-value function backup diagram. For each s' in the bottom of the tree, there is some different weighting applied. The first level of the tree is weighted by $\pi(a|s)$. The second level of the tree is weighted by the transition dynamics of the environment $\mathbf{P}_{s,s'}^a$. Figure taken from [2].

Again, we apply the same thinking to action-values only with the weighting order reversed¹¹.

¹¹For the action-value, the first weighting is applied with respect to the transition dynamics of the environment. The second weighting is applied with respect to the policy.

The equation that computes $q_\pi(s, a)$ is defined by:

$$q_\pi(s, a) = \mathbf{R}_s^a + \gamma \sum_{s' \in \mathbb{S}} \mathbf{P}_{s, s'}^a \sum_{a' \in \mathbb{A}} \pi(a' | s') q_\pi(s', a'). \quad (3.19)$$

The complete q_π backup diagram for the action-value function is shown in the Figure 3.6, as follows:

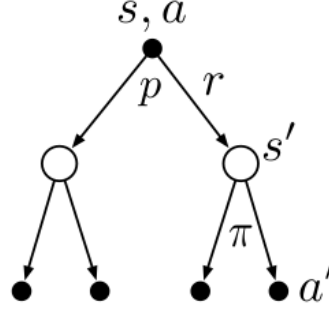


Figure 3.6: The action-value function backup diagram. For each a' in the bottom of the tree, there is different weighting applied. The first level of the tree is weighted by the transition dynamics of the environment $\mathbf{P}_{s, s'}^a$. The second level of the tree is weighted by $\pi(a' | s)$. Figure taken from [2].

To conclude this section, it is time to expose the real desire of an agent: the **optimal** way of behaving in an MDP. Hence, the real goal is to find optimal policies (π_*) and optimal value functions (v_* and q_*) [3].

By definition, the optimal state-value function is the maximum state-value function over all policies, that is,

$$v_*(s) \doteq \max_{\pi} v_{\pi}(s). \quad (3.20)$$

The optimal action-value function is the maximum action-value function over all policies, that is,

$$q_*(s, a) \doteq \max_{\pi} q_{\pi}(s, a). \quad (3.21)$$

The optimal value-function specifies the best performance an agent can extract in an MDP. In an MDP there is at least one policy that is better or equal to all other possible policies ($\pi \geq \pi', \forall \pi$) if its expected return is greater or equal to any other possible returns of all other possible policies for all states. There could be many optimal policies, and they all share the same value function, specifically the optimal value function (v_* or q_*). Some policy (π) is better than other possible policy (π'), as expressed as follows:

$$\pi \geq \pi' \text{ if } v_{\pi} \geq v_{\pi'}, \forall s \in \mathbb{S}. \quad (3.22)$$

All optimal policies achieve the optimal state-value function, as follows:

$$v_{\pi_*}(s) = v_*(s). \quad (3.23)$$

All optimal policies also achieve the optimal action-value function, as defined by:

$$q_{\pi_*}(s, a) = q_*(s, a). \quad (3.24)$$

Finding an optimal policy is, therefore, acting optimally with respect to the action-value function. If we maximize over q_* , picking the action that gives us the greater expected return, we find the optimal policy, as follows:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg \max_{a \in \mathbb{A}} q_*(s, a) \\ 0 & \text{otherwise} \end{cases} \quad (3.25)$$

Finally, it is time to arrive at another type of Bellman Equation called the Bellman Optimality Equation. This equation describes the expected return while acting optimally (π_*). Let us first describe the relationships between the optimal state and action-value functions. The optimal state-value function is the optimal action-value function over all actions, that is,

$$v_*(s) = \max_{a \in \mathbb{A}} q_*(s, a). \quad (3.26)$$

The former described relationship from v_* to q_* is supported by the Figure 3.7, as follows:

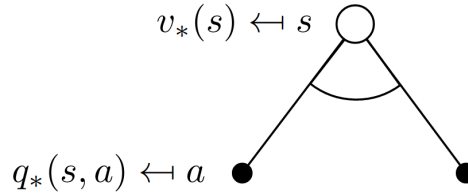


Figure 3.7: Relationship between v_* and q_* . It is a maximum operation over all the possible optimal successor action-values. Figure taken from David Silver's slides at: <https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf>.

On the other hand, the optimal action-value function is the immediate reward plus the discounted sum weighted by the transition dynamics of the environment over all the optimal possible successor state-values, that is,

$$q_*(s, a) = \mathbf{R}_s^a + \gamma \sum_{s' \in \mathbb{S}} \mathbf{P}_{s,s'}^a v_*(s'). \quad (3.27)$$

The former described relationship from q_* to v_* is supported by the Figure 3.8, as follows:

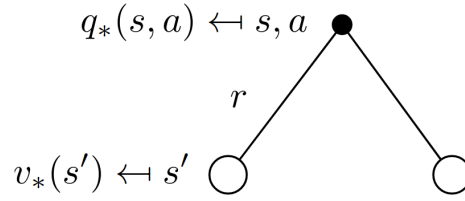


Figure 3.8: Relationship between q_* and v_* . It is equal to the immediate reward plus a sum over all the possible optimal successor state-values weighted by the transition dynamics of the environment. Figure taken from David Silver's slides at: <https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf>.

Putting it all together we arrive at the final complete relationships to compute optimal state and action-values (as a recursive relationship), using a backup process. The Bellman Optimality equation for v_* is defined by:

$$v_*(s) = \max_{a \in \mathbb{A}} R_s^a + \gamma \sum_{s' \in \mathbb{S}} \mathbf{P}_{s,s'}^a v_*(s'). \quad (3.28)$$

To support the former equation, we expose the v_* backup diagram, as follows:

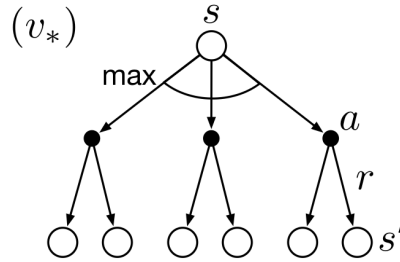


Figure 3.9: The optimal state-value function backup diagram. A maximum operator chooses the best sum across all actions, between the immediate reward and the discounted sum for the successor state-values weighted by each transition probability ($\mathbf{P}_{s,s'}^a$). Figure taken from [3].

Similarly, we have a Bellman Optimality Equation for q_* , defined by:

$$q_*(s, a) = R_s^a + \gamma \sum_{s' \in \mathbb{S}} \mathbf{P}_{s,s'}^a \max_{a' \in \mathbb{A}} q_*(s', a'). \quad (3.29)$$

To support the former equation, we expose the q_* backup diagram, as follows:

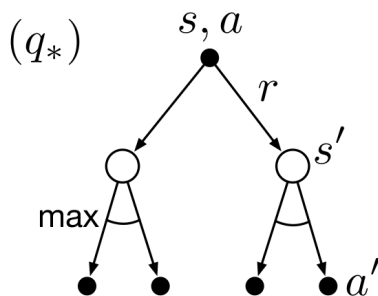


Figure 3.10: The optimal action-value function backup diagram. A maximum operator chooses the best action-value for each of the successor states. These values are then summed up and weighted by each transition probability ($\mathbf{P}_{s,s'}^a$). Figure taken from [3].

Solving the Bellman Optimality Equation for a large MDP is computationally complex and demands a lot of memory. Additionally, it is a non-linear equation and has no closed form solution, generally. To overcome this problem, many iterative solutions methods are applied in the RL literature: as an example, Dynamic Programming techniques like Value Iteration and Policy Iteration, Temporal-Difference techniques like Q-learning and Sarsa, Monte-Carlo techniques and Exhaustive Tree Search techniques. These methods differ, essentially, in the deepness of the search and the width of the search and the access to a model of the environment to estimate and to iteratively update value functions. To note that, until now, we described the problem of RL accompanied by a knowledge of the environment's dynamics ($\mathbf{P}_{s,s'}^a$), however that knowledge is not always accessible, because, or it is not estimated or it is too complex to estimate. Dynamic Programming and Exhaustive Tree Search methods (Model-Based techniques) have the access and use a model of the environment to update their value functions. On the other hand Temporal-Difference Learning and Monte Carlo methods (Model-Free techniques), don't have knowledge about the environment, hence, by sampling from it, and, through experience, they can update their own value functions. In the following section, we will focus on one of the former mentioned approaches, named Temporal-Difference Learning, which describes the way that a Q-learning (the chosen method to solve the problem of this dissertation) agent learns.

3.2 Q-learning

In the early days of Reinforcement Learning a big breakout was the development of a successful Model-Free Off-policy Temporal-Difference Control algorithm, named *Q-learning* [10][11]. Still popular to date because of its simplicity and effectiveness, this algorithm gathers several important features that makes it one of the most popular choices for solving RL problems. In the following subsections we will explain Q-learning by exposing those features, namely: being Model-Free, using Temporal-Difference Control for evaluating and improving its policy and being Off-Policy.

3.2.1 Model-Free

In the previous subsection 3.1.4 we described the problem of RL as solving an MDP using Bellman Equations. There was one assumption, that was the use of a model, in other words, the use of the knowledge about the transition dynamics, to discover and to update true value-functions (v_π and q_π) or optimal value-functions (v_* and q_*). Implicit optimal policies (π_*) are consequently extracted if we act greedily with respect to those value-functions.

Dynamic programming [67] techniques are applied when we have access to the transition dynamics $\mathbf{P}_{s,s'}^a$, however, in this section, we present a specific Model-Free RL method, named Q-learning algorithm, that, despite not having access to $\mathbf{P}_{s,s'}^a$, it can learn and seek optimal behavior likewise Model-Based methods.

In a Model-Free setting, the MDP is solved, not by planning in the known dynamics, but by **sampling** experience from it. Thus, the agent learns by trial-and-error and direct interaction, since that is the only way of interacting with the environment while not knowing its dynamics. Hence, sample backups are performed instead, meaning that, the agent will perform actions and empirically backup the sampled experience to learn value-functions. In the next subsection we will describe the process in which, generally, all RL methods (including Q-learning) rely on to find optimal policies and value-functions, that is the Generalized Policy Iteration.

3.2.2 Generalized Policy Iteration

The general principle of finding optimal policies in RL is sustained by the Generalized Policy Iteration (GPI) principle [4]. Specifically, all Model-Free methods, are well described by GPI. The GPI principle consists of two distinct and interactive processes: one focuses on evaluating a policy, a process known as **Policy Evaluation** [68]. The other focuses on improving the policy, a process known as **Policy Improvement** [69].

As exposed in the Figure 3.11, the up arrows correspond to the Policy Evaluation process, consisting of an iterative update of the state-value function (v_π) for an arbitrary policy π according to the Bellman Expectation Equation, as follows:

$$\begin{aligned} v_{k+1} &\doteq \mathbf{E}_\pi [R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')]. \end{aligned} \quad (3.30)$$

On the other hand, the down arrows correspond to the Policy Improvement process, consisting of improving the current policy to at least an equal or better policy. A policy improves by acting greedily with respect to its value function. Considering a deterministic policy $a = \pi(s)$, and acting greedily with respect to it, let the new improved policy (π') be defined by:

$$\pi'(s) = \arg \max_{a \in \mathbb{A}} q_\pi(s, a). \quad (3.31)$$

This greedy choice, improves the value we are going to get at least over one step from any state s and following policy π afterwards. We get at least as much reward over one step and following policy π forever after, as the policy we started with, in other words, at least as much reward than any particular action that we could have chosen before, under the old policy, as defined by:

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathbb{A}} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s). \quad (3.32)$$

Hence, by iterating this process, as proven in [70], what is better for one step, it is better for the whole executed trajectory, that is, the value of policy π' is at least better or equal than the value of the old policy π , as defined by:

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) = \mathbf{E}_{\pi'} [R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\ &\leq \mathbf{E}_{\pi'} [R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\ &\leq \mathbf{E}_{\pi'} [R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) | S_t = s] \\ &\leq \mathbf{E}_{\pi'} [R_{t+1} + \gamma R_{t+2} + \dots | S_t = s] = v_{\pi'}(s). \end{aligned} \quad (3.33)$$

If this iterative process ultimately stops, the Bellman Optimality Equation is satisfied, that is,

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathbb{A}} q_\pi(s, a) = q_\pi(s, \pi(s)) = v_\pi(s), \quad (3.34)$$

and finally:

$$v_\pi(s) = \max_{a \in \mathbb{A}} q_\pi(s, a). \quad (3.35)$$

Hence, if it is true for one state, it is true for all states $s \in \mathbb{S}$, as defined by:

$$v_\pi(s) = v_*(s), \forall s \in \mathbb{S}. \quad (3.36)$$

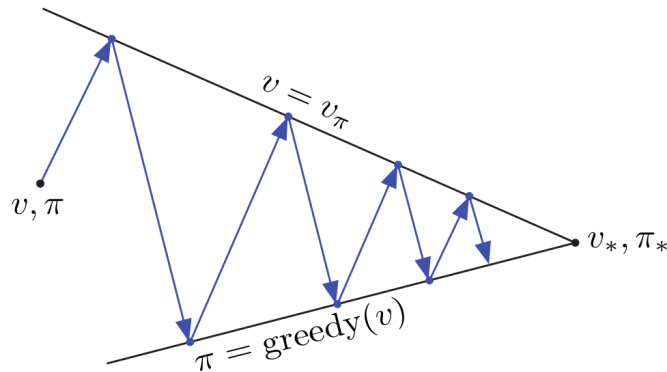


Figure 3.11: The Generalized Policy Iteration diagram. The value function is approximated repeatedly to be closer to the true value of the current policy, and, meanwhile, the policy is improved repeatedly to approach optimality. The process is iterative and ultimately converges to the optimal value-function v_* and, consequently, to the optimal policy π^* . Figure taken from [4].

3.2.3 Off-Policy Temporal-Difference Control

Let us finish this section by explaining the remaining pieces of the Q-learning method: the Temporal-Difference Control and the Off-Policy aspects.

Temporal-Difference: The evaluation process of Q-learning applies Temporal-Difference (TD) learning [5] principles. TD is a Model-Free prediction method that estimates a state-value according to experience sampled from the environment and applying **bootstrapping**. Bootstrapping is a technique where a value estimate is updated according to another value estimate¹², instead of relying on complete returns, like in Monte-Carlo methods [14]. Specifically, TD(0) is a special case of TD Learning, where we update a value estimate over only one time-step¹³. TD methods have the uniqueness of not depending on episodic regimes, since they do not depend on complete returns like Monte-Carlo methods, thus they can learn by incomplete sequences. Hence, they can perform predictions at every time-step, only needing to wait until the next time-step, in the extreme case, to perform evaluation.

As already defined by the Bellman Equation in 3.14, the expected return from a given state s could be given as a sum of the immediate reward plus the value of the successor state. In TD, the value of the successor state is an estimate $V(S_{t+1})$, because we have not access to the true value¹⁴ of the successor state $v_\pi(S_{t+1})$. According to TD(0), for a specific visited state s we update its value, as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)). \quad (3.37)$$

Evaluating any policy according to the TD(0) method, is backing up and updating any given $V(S_t)$ in the direction of a TD-target, that is, $R_{t+1} + \gamma V(S_{t+1})$, over one step into the future, as illustrated in the Figure 3.12, as follows:

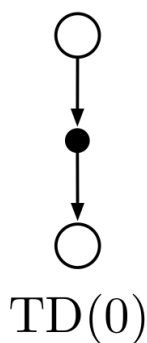


Figure 3.12: A sample backup over one step from the successor state-value $V(S_{t+1})$ in the bottom, to the original state-value $V(S_t)$ in the top. Figure taken from [5].

¹²In other words, *bootstrapping* is updating a guess from another guess.

¹³The general case of TD, specifically n-step TD, defines the number of sampling steps that we want to perform in the environment in order to define the expected return for the state-value. TD(0) is a special case that Q-learning follows, since it samples over only one time step.

¹⁴As already, shown previously in 3.1.4, for that, we need the access to a model.

Control: While the Temporal-Difference aspect is related to the evaluation step, the Control aspect in the Temporal-Difference Control feature, is related to the Policy Improvement step of Q-learning's policy iteration process. If Temporal-Difference Learning is used with state-value functions, it is not possible to apply policy improvement, in other words, acting greedily, without having access to a model, as follows:

$$\pi'(s) = \arg \max_{a \in \mathbb{A}} \mathbf{R}_s^a + \mathbf{P}_{s,s'}^a V(S_{t+1}). \quad (3.38)$$

However, if we choose to work with action-value functions, as in the case of Q-learning, we can improve the policy by acting greedily with respect to it, without needing a model, as follows:

$$\pi'(s) = \arg \max_{a \in \mathbb{A}} Q(s, a). \quad (3.39)$$

Action-value functions enable us to do Control in a Model-Free setting, thus, by caching the values of the actions we can take, we reduce the burden of having to use a model. The information about the model interaction experience is somewhat already present in the structure of the action-value function itself. Hence, applying Temporal-Difference to action-value functions is classified as Temporal-Difference Control.

Q-learning's Policy Iteration: Let us focus now on explaining the Q-learning's policy iteration process. We present the Figure 3.13 taken from David Silver's Lectures¹⁵ to illustrate the process. To note that, due to not finding the representative policy iteration diagram of Q-learning, we present instead the Sarsa [6] diagram, an algorithm similar to Q-learning but instead of being Off-Policy, it is an On-Policy method. In the diagram representation, the main difference is that Sarsa evaluates the followed policy with respect to the true value-function q_π (as represented in the figure: $Q = q_\pi$), while Q-learning evaluates the followed policy with respect to the optimal value-function q_* ($Q = q_*$). The difference between On-Policy and Off-Policy is explained further.

¹⁵David Silver's lectures are available at: <https://www.davidsilver.uk/teaching/>

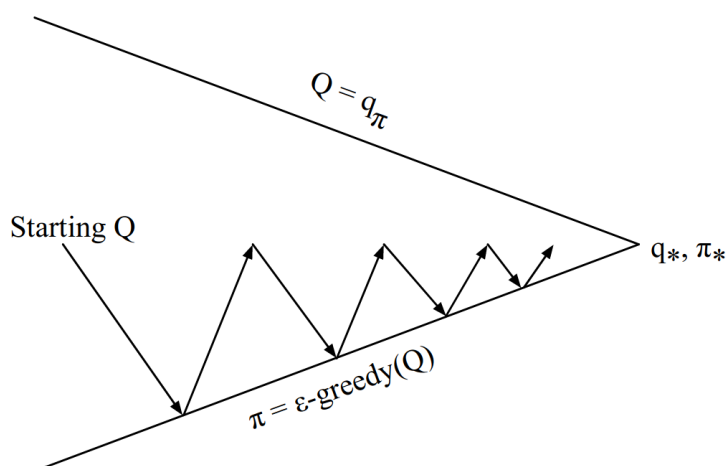


Figure 3.13: The Sarsa's [6] policy iteration diagram. The Q-learning's diagram is similar, differing only in the policy evaluation step (instead of approximating the true value-function ($Q = q_\pi$), it approximates the optimal value-function ($Q = q_*$). The Figure is taken from Lecture 5 of David Silver's lectures, available at: <https://www.davidsilver.uk/wp-content/uploads/2020/03/control.pdf>.

Firstly, we can observe that the Policy Evaluation step is not completely moving closer to the top (meeting the true q_π). Learning while sampling from an environment means that the value estimates are updated according to the value estimates of the successor states we encounter according to the current policy. We saw previously that, by having the access to a model, the evaluation step moves the value of the policy exactly to its true value (v_π), because, by making a lookahead into the future, we backup the true state-value (as the complete weighted sum of the values of the successor states according to the environment's dynamics, as already exposed in subsection 3.1.4). Due to Q-learning's sampling experience, its evaluation process, represented by the up arrows in the Figure 3.13, does not go straight to the top, because the evaluation does not converge directly to the exact true/optimal value-function, being rather an approximation or estimate of it. However, this limited aspect does not prevent Q-learning to converge to the optimal value-function q_* .

Secondly, the Policy Improvement step, while acting greedily, is a caveat in the case of Model-Free methods. Model-Free methods learn from the environment by interacting with it, therefore if they do not explore the environment properly they will not arrive at the optimal and true value-functions, because they may suffer from not extracting sufficient knowledge from the entire state-action values available in the environment, hence not getting reasonable estimates of the value function and arriving at sub-optimal behavior. Therefore, by not benefiting from performing full sweeps in the known dynamics of the environment, if they only act greedily, they will get stuck in certain areas of the state-space, hence not exploring properly other areas of interest in the state-action space. To overcome that problem, several exploration strategies exist in the literature, but a common approach is the ϵ -greedy strategy (as we can observe from the Policy Improvement's down arrows endpoint in the Figure 3.13), consisting of balancing greedy action selection (exploitation) with random action selection (exploration). With probability $1 - \epsilon$ a greedy

action is selected, with probability ε a random action is selected, as defined by:

$$\pi(a|s) = \begin{cases} \frac{\varepsilon}{|A|} + 1 - \varepsilon & \text{if } a^* = \arg \max_{a \in A} Q(s, a) \\ \frac{\varepsilon}{|A|} & \text{otherwise} \end{cases} \quad (3.40)$$

Off-Policy: There are two types of learning procedures in the literature, each with its unique approach. Briefly, the learning process consists of following some behavior and then one could optimise the behavior in the direction of another behavior, or one could optimise the behavior in the direction of its own behavior. This particular aspect, related to the source of the optimisation, is what distinguishes On-Policy Learning from Off-Policy Learning.

On-Policy Learning, as stated by David Silver in one of his lectures¹⁶ is described intuitively as "Learn on the job", meaning that, some policy π is being followed and collecting experience and, at the same time, the agent learns according to its online behavior π . This is the intuition behind the Sarsa [6] algorithm, that is, some action is sampled, according to the ε -greedy strategy, from the behavior policy, as follows:

$$A_t \sim \pi(\cdot|S_t). \quad (3.41)$$

Next, the behavior policy is sampled again according to the same ε -greedy strategy, to produce the action for the next time step, that is,

$$A_{t+1} \sim \pi(\cdot|S_{t+1}). \quad (3.42)$$

Lastly, the current action-value ($Q(S_t, A_t)$) is optimised in the direction of the bootstrapping estimate for the next state, produced by the same behavior policy, as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)). \quad (3.43)$$

To conclude, the On-Policy nature is represented by both the sampling aspects mentioned above: sampling from π to explore and sampling from π to produce a target for optimization.

Off-Policy Learning, on the other hand, is described¹⁷ intuitively as learning by a "Look over someone's shoulder", meaning that, some policy μ is being followed and collecting experience, while learning from another target policy π . Now the target policy is disconnected from the behavior policy, hence the agent is taking inspiration and optimising its policy in the direction of another policy. This is the learning procedure of Q-learning, that is, some action is sampled according to the behavior policy μ , following some ε -greedy strategy, as follows:

$$A_t \sim \mu(\cdot|S_t). \quad (3.44)$$

¹⁶These statements are available at: <https://www.davidsilver.uk/wp-content/uploads/2020/03/control.pdf>.

¹⁷Also stated by David Silver in one of his lectures. Lecture 5 available at: <https://www.davidsilver.uk/wp-content/uploads/2020/03/control.pdf>.

Afterwards, the target policy π greedily produces the action for the next time step, as defined by:

$$A_{t+1} = \pi(S_{t+1}) = \arg \max_{a \in \mathbb{A}} Q(S_{t+1}, a). \quad (3.45)$$

Therefore, the Q-learning target is defined by:

$$\begin{aligned} & R_{t+1} + \gamma Q(S_{t+1}, \pi(S_{t+1})) \\ &= R_{t+1} + \gamma Q(S_{t+1}, \arg \max_{a \in \mathbb{A}} Q(S_{t+1}, a)). \end{aligned} \quad (3.46)$$

Lastly, the current action-value is optimised in the direction of the bootstrapping estimate for the next state, produced by the target policy, as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (R_{t+1} + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)). \quad (3.47)$$

Off-Policy comes naturally as one interesting aspect that brings a lot of advantages¹⁸, namely: (1) Learning about the optimal behavior while following exploratory behavior (as described above, a policy μ is in charge of producing exploratory behavior while other policy π is producing optimal behavior by being greedy); (2) The ability to learn from observing other agents or humans (since the target policy is disconnected from the agent's self-behavior, the agent could optimise itself in the direction of other behaviors); (3) Both μ and π improve, that is, both are performing iterative steps in the direction of the optimal behavior. Finally, the diagram of the Q-learning algorithm is exposed in the Figure 3.14, as follows:

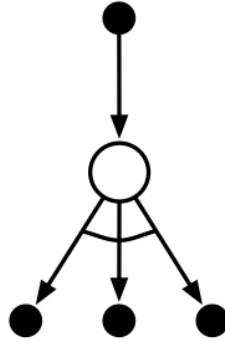


Figure 3.14: The *Q-learning* diagram. On top, an action-value is updated according to the maximum subsequent action-value in the bottom. The arc on the bottom represents the max operator performed over all the successor action-values. Figure taken from [7].

¹⁸The advantages are exposed in: <https://www.programmingsought.com/article/63464655397/> and in David Silver's Lecture 5: <https://www.davidsilver.uk/wp-content/uploads/2020/03/control.pdf>.

3.2.4 Convergence

In theory, Q-learning directly approximates the optimal value-function q_* , due to the *max* operation applied in its iterative update process independent of the policy being followed (due to the Off-Policy aspect). This iterative process (learning) has correct convergence properties only if it meets the following conditions:

1. The *Greedy in the Limit with Infinite Exploration* (GLIE) condition¹⁹. This condition is met if all state-action pairs are infinitely visited, as defined by:

$$\lim_{k \rightarrow \infty} n_k(s, a) = \infty. \quad (3.48)$$

Also, if, in the limit, the policy converges to the greedy policy, as follows:

$$\lim_{k \rightarrow \infty} \pi_k(a|s) = \mathbf{1}(a = \arg \max_{a' \in \mathbb{A}} Q_k(s, a')). \quad (3.49)$$

In principle it needs to exist a non-zero probability of visiting every $a \in \mathbb{A}(s)$ for all $s \in \mathbb{S}$ at any time step t during learning, while following policy Q , that is,

$$\Pr_t(a, s|Q) \geq c > 0, \forall s, a. \quad (3.50)$$

In practice, the GLIE condition is satisfied by following an ε -greedy policy strategy, with $0 < \varepsilon \leq 1$, that ultimately converges to the greedy policy ($\varepsilon \rightarrow 0$)²⁰.

2. The *Robbins-Monro* [71] sequence of step-sizes α_t condition that guarantees pointwise convergence ($Q_t \rightarrow Q$). The learning-rate α controls the influence of the new information on the learning update rule for each Q . For convergence, the strength of the assimilation of new information needs to decay and stabilize in order to prevent oscillations, ensuring that the learned Q-values eventually settle into a final value. The condition proposes a learning rate scheduling, depending on one of the following time-signatures: (1) Global-clocks: a global decaying learning rate proportional to some global time step t ; (2) Local-clocks: A local decaying learning rate for each state-action pair (s, a) proportional to the number of visits in each state-action pair $n_t(s, a)$.

Formally, to satisfy this condition the learning rate needs to be sufficiently high to ensure that a Q-value can arrive to its desired final value, that is,

$$\sum_{t=1}^{\infty} \alpha_t = \infty. \quad (3.51)$$

¹⁹Description presented in page 2 of the following document: https://webee.technion.ac.il/shimkin/LCS11/ch7_exploration.pdf. It is also available at slide 15 of David Silver's lecture about Model-Free Control, available at: <https://www.davidsilver.uk/wp-content/uploads/2020/03/control.pdf>.

²⁰This particular aspect of decaying to the greedy policy is achieved by, per example, decaying the epsilon parameter according to the number of the time steps t during learning, that is, $\varepsilon = 1/t$, as stated in [6].

It also needs to be sufficiently small to ensure that the changes in the Q-values eventually vanish, as defined by:

$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty. \quad (3.52)$$

To conclude, we can say that convergence is guaranteed as easily as the less complex and smaller it is the state-action space. Exploring vast state-action spaces is unreasonable in terms of computational resources and time. Also, to guarantee that the Q-values eventually converge to a final value, they need to be continually updated, therefore, when exploring is hard, these Q-values are less likely visited or continually updated, hence convergence naturally suffers.

To introduce the following section, to note that, Q-learning belongs to the class of the RL tabular methods [42], meaning every action-value is stored in a table, therefore only when the state-action space is relatively small, it is feasible to visit and update every single state-action pair in the Q-table [4.1.2]. In more complex problems that require a lot of states and actions to be described, due to their huge dimensionality [2.2.1], applying Q-learning is not feasible and will not converge, since it will not be possible to respect the condition 1.

To escalate simple RL algorithms like Q-learning to more complex and high-dimensional problems, instead of using tabular memorized experience, function-approximation techniques are used to estimate the Q-function directly. Commonly, parameterised function-approximation techniques like Neural Networks are used with the goal of estimating and approximating the optimal Q-function directly (q_*). If one uses a function with parameters θ , the approach consists of approximating directly the optimal value-function, that is,

$$Q(s, a; \theta) \approx q_*(s, a). \quad (3.53)$$

In the following section, we will present the approximate solution method that extends Q-learning to more real and complex problems: it is called Deep Q-Network and it combines Q-learning with Deep Neural Networks.

3.3 DQN

As already exposed previously, instead of learning by remembering the experience for each state-action pair like Q-learning does (unfeasible when the number of states and actions becomes too large), DQN follows a different approach: it generalizes and approximates the optimal Q-value function directly using neural network techniques. The intuition is, essentially, instead of learning every single action-value, function approximation seeks to generalise action-values across all states.

DQN (Deep Q-Networks) [24; 29], presented by DeepMind²¹ in 2013, is the the first Deep Learning model to learn successful control policies directly from high-dimensional raw sensory input using Reinforcement Learning principles. It is widely known for surpassing human-level

²¹DeepMind's official website: <https://deepmind.com/>

performance across a set of 49 Atari 2600 games. In this remarkable achievement, the DQN model kept the same architecture and hyperparameters proving its incredible versatility and adaptability.

3.3.1 Architecture

The most common DQN architecture consists of a deep convolutional neural network [72] which receives several image frames of the environment as input and outputs the Q-values for all possible actions in the discrete action space of the agent, as shown in the Figure 3.15. The input consists of N frames (in the example shown in the Figure 3.15, the input consists of 4 frames) correspondent to the last N observations of the agent²². The images are fed into the network where, through the convolutional layers, filtering and convolutional methods are applied, extracting the most useful features of the input frames. After passing the convolutional layers, only the most important pixels of the input are fed to a final dense layer that learns a mapping to each action-value ($Q(s, a), \forall a \in \mathbb{A}$). This specific described architecture is designed for high-dimensional problems when the inputs are 2D images, like Atari Games, however the architecture could be adapted depending on the problem at hand.

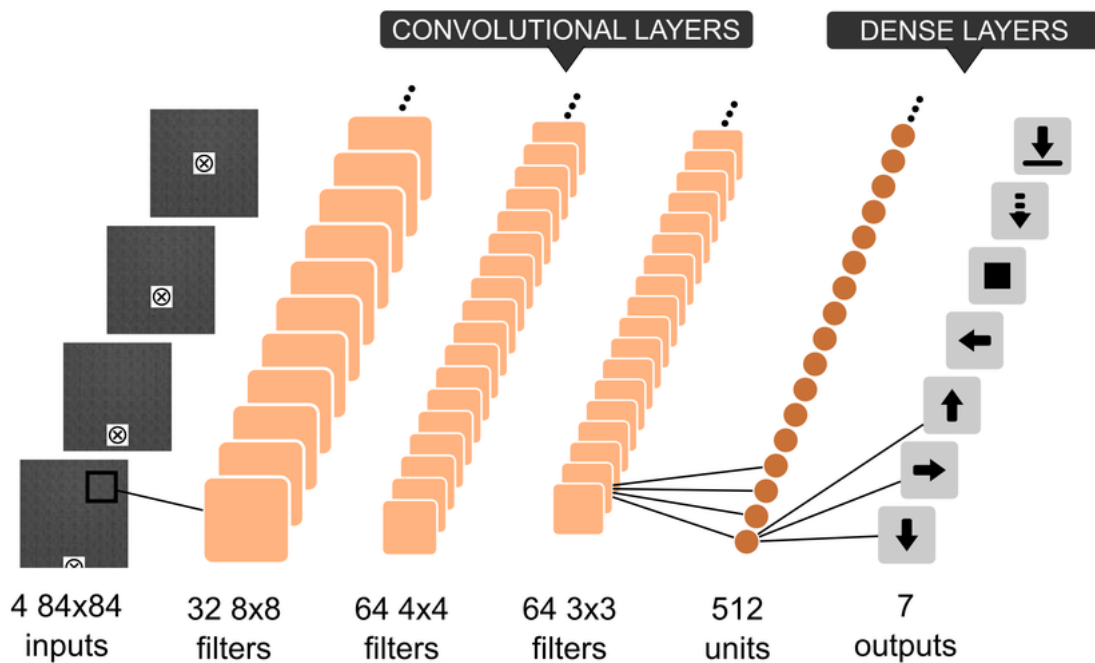


Figure 3.15: The most common DQN architecture. In this example it receives 4 image frames as input and outputs 7 action nodes. The convolutional layers learn to extract the most relevant features of the raw image frames. The compacted version of the information produced by the convolutional layers is then passed to the dense layers which are responsible for outputting the Q-value for each action. Available at: <https://paperswithcode.com/method/dqn>

²²This input is based on frames, because with frames it is possible to perceive dynamics in the input.

3.3.2 Overcoming the Deadly Triad

Temporal-Difference Learning and bootstrapping is very flexible, convenient and efficient for the majority of problems[73]. Off-Policy Learning is equally useful if we want to extract knowledge and get inspiration from other behaviors. Applying function approximation techniques is also a necessity when we want to escalate Reinforcement Learning to real-world and complex problems. However, when Bootstrapping, Function Approximation and Off-Policy Learning are combined in a single Reinforcement Learning algorithm, divergence and instability issues appear naturally. Combining the three former mentioned aspects results in the so called **Deadly Triad** by Sutton and Barto [74].

Applying function approximation techniques introduces generalization across states, meaning that, if two states are similar, and the value of one of them is updated, the value of the other one is update accordingly. Applying bootstrapping with function approximation implies updating state-action values according to subsequent states, hence, if these states are similar, they will be inadvertently updated. Applying both the former techniques is not a problem if the agent updates the bootstrapped values as often as they are used, something that happens with On-Policy Learning. However if the agent learns Off-Policy it might not update these bootstrapped values sufficiently often to prevent them from spiralling out of control. As stated by Sutton and Barto [75], "The potential for off-policy learning remains tantalizing, the best way to achieve it still a mystery.", one can not give up from off-policy training that easily because it is extremely beneficial to learn from other behaviors in parallel.

According to a DeepMind study [76], combining Reinforcement Learning with non-linear function approximation techniques like Neural Networks could easily go wrong if we do not take into account several aspects. The study concluded that the overestimation of the Q-values during learning is extremely common and a highly critical aspect that compromises both training and further performance of the agent. This problem can be reduced by merging double Q-learning with the use of target networks, hence reducing the overestimation bias issue. When an agent is learning off-policy, it samples past experience from its replay memory, however, if this sampling is more uniformly random and less prioritized, it improves a lot the stability of the algorithms, the study concluded. Another interesting conclusion is that smaller neural networks are more stable and diverge less during training, not overestimating that much, however they perform poorly when compared to larger networks. Lastly, an interesting strategy to reduce the bootstrapping inadvertent updates is to increase the number of steps of the bootstrap method, hence reducing bias and preventing inadvertent updates naturally caused by small-step bootstrapping.

Although, DQN is capable of working with the Deadly Triad: DQN uses Deep Neural Networks²³, learns Off-Policy and uses one-step Q-learning for bootstrapping, and, surprisingly, despite combining all of the deadly triad elements, it still works successfully [29; 76].

²³Deep Neural Networks are non-linear function approximation techniques.

Remarkably, DQN's creators arrived at the conclusion that, if we take inspiration from and apply Supervised Learning principles, the problems that arise from the online experience created by a Reinforcement Learning agent could be solved. In Supervised Learning (SL), the data that it is being learned is uniformly distributed and it is static. Contrary, Reinforcement Learning's datasets used and created by Off-Policy agents are dynamically changing because the experience that is being stored, it is highly dependent from the policy that the agent is following that is also dynamically changing. These former mentioned problems associated with learning from dynamic data, derives in the following issues:

- P1** The strong correlations present in the sequence of observations when learning online.
- P2** The correlations between the action-values from the approximated Q-function and the target action values produced by the same approximated Q-function.
- P3** The short-term oscillations produced by moving targets. During the learning process, the Q-function is changing (the parameters change), hence the targets being produced by the same Q-function are also changing. Hence, the learning process suffers from chasing non-stationary targets.

In SL, a model learns by sampling from a dataset where the input is independent (samples being independent of each other in the same batch) and identically distributed (samples being randomized among batches and hence each batch having a similar data distribution)²⁴. DQN achieves the former behaviour because it has a mechanism known as *experience replay*: it consists of a buffer that stores the agent experience over time. Online experience is temporally correlated, so it is advisable, according to statistical learning and optimization theory, to break the temporal correlations from the stored experience. Randomly sampling from the replay memory breaks these correlations, feeding the network with independent and identically distributed data, solving problem **P1**.

Additionally, a SL setting has labels or targets for each input that do not change over time. This data stability is another feature that allows supervised learning to perform well. DQN is able to learn with a similar stability by having two networks: a frozen target network alongside the policy network. This target network is periodically updated, remaining frozen the rest of the time, allowing the policy network to bootstrap fixed targets produced by the stationary target network. Hence, DQN also solves the problems **P2,P3**: with frozen/stationary targets, the chasing is highly reduced and the correlations between the action-values and the target action-values are removed because we produce the target bootstrapped estimate from a separate network that remains frozen for a fixed duration²⁵.

²⁴Concise exposition of the DQN's process: <https://jonathan-hui.medium.com/rl-dqn-deep-q-network-e207751f7ae4>. Also an interesting analysis, available at: [77]

²⁵The target network frozen duration is controlled via an hyperparameter. This hyperparameter dictates the period in which the target network is synchronized with the main behavior network. The rest of the time it remains stationary.

The goal of DQN is to approximate the optimal Q-function defined by:

$$\begin{aligned}
 q_*(s, a) &= \max_{\pi} \mathbf{E} [R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \dots | S_t = s, A_t = a] \\
 &= \max_{\pi} \mathbf{E} [G_t | S_t = s, S_t = a] \\
 &= \mathbf{E} \left[R_{t+1} + \gamma \max_{A_{t+1}} q_*(S_{t+1}, A_{t+1}) \right]
 \end{aligned} \tag{3.54}$$

The problem of approximating $q_*(s, a)$ is solved by minimizing a loss function. Like a SL setting, minimizing a loss function consists of reducing the difference between a target function and a prediction function.

In the DQN framework, the loss function $\mathcal{L}$ is defined by the squared difference between the optimal Q-function q_* , represented by the target network Q^- with parameters θ_i^- , and the policy Q-function q_π , represented by the policy network Q with parameters θ_i .

As mentioned before, at each time step, the learning process will sample a tuple $e_t = (S_t, A_t, R_{t+1}, S_{t+1})$ containing a state, an action, a reward, and a successor state from the experience replay buffer $D_t = \{e_1, \dots, e_t\}$ uniformly at random $(S_t, A_t, R_{t+1}, S_{t+1}) \sim U(D)$. The parameters θ of the policy network will be updated in the direction of the minimization of the following loss function, that changes iteratively:

$$\begin{aligned}
 \mathcal{L}_i(\theta_i) &= \mathbf{E}_{e_t \sim U(D_t)} [(Q^-(S_t, A_t; \theta_i^-) - Q(S_t, A_t; \theta_i))^2] \\
 &= \mathbf{E}_{e_t \sim U(D_t)} \left[\left(R_{t+1} + \gamma \max_{A_{t+1}} Q^-(S_{t+1}, A_{t+1}; \theta_i^-) - Q(S_t, A_t; \theta_i) \right)^2 \right]
 \end{aligned} \tag{3.55}$$

Chapter 4

Q-learning for Robotic Navigation

This chapter is organized into two parts. The first part presents a detailed description about the implementation of Q-learning. The second part describes the simulation results and the experiments with the mobile Turtlebot.

4.1 Implementation

In this section we will first give an overview over the global architecture of the Q-learning system (in subsection 4.1.1). Secondly, we will expose the Q-table (in subsection 4.1.2) and the intrinsic needs for its construction: the discretization method and the definition and construction of the state space and the action space. Next, in subsection 4.1.3 we will expose and explain the algorithm of the reward function. Lastly, in subsection 4.1.4, we will expose and explain the Q-learning algorithm.

4.1.1 Architecture

The UML-based illustration shown in Figure 4.1 represents the implemented architecture sustained by ROS-based concepts¹. In our diagram we represent the different elements of our architecture (ROS-nodes, ROS-topics and auxiliar modules called by the nodes) and the flow of the interactions between them.

Briefly, ROS-nodes² are programs that are continually running and performing computations while interchanging or not data between them following a *publish-subscribe*³ messaging pattern. ROS-topics⁴ are communication channels containing ROS-messages⁵ published (written) by nodes. ROS-nodes can not only be publishers but also subscribers if they want to have access (to read) to topics containing messages of interest. In the diagram, an arrow pointing from a node

¹<http://wiki.ros.org/ROS/Concepts>

²<http://wiki.ros.org/Nodes>

³https://en.wikipedia.org/wiki/Publisher-subscribe_pattern

⁴<http://wiki.ros.org/Topics>

⁵<http://wiki.ros.org/Messages>

to a topic means that the node publishes in the topic. The arrow pointing from a topic to a node means that the node is subscribed to the topic.

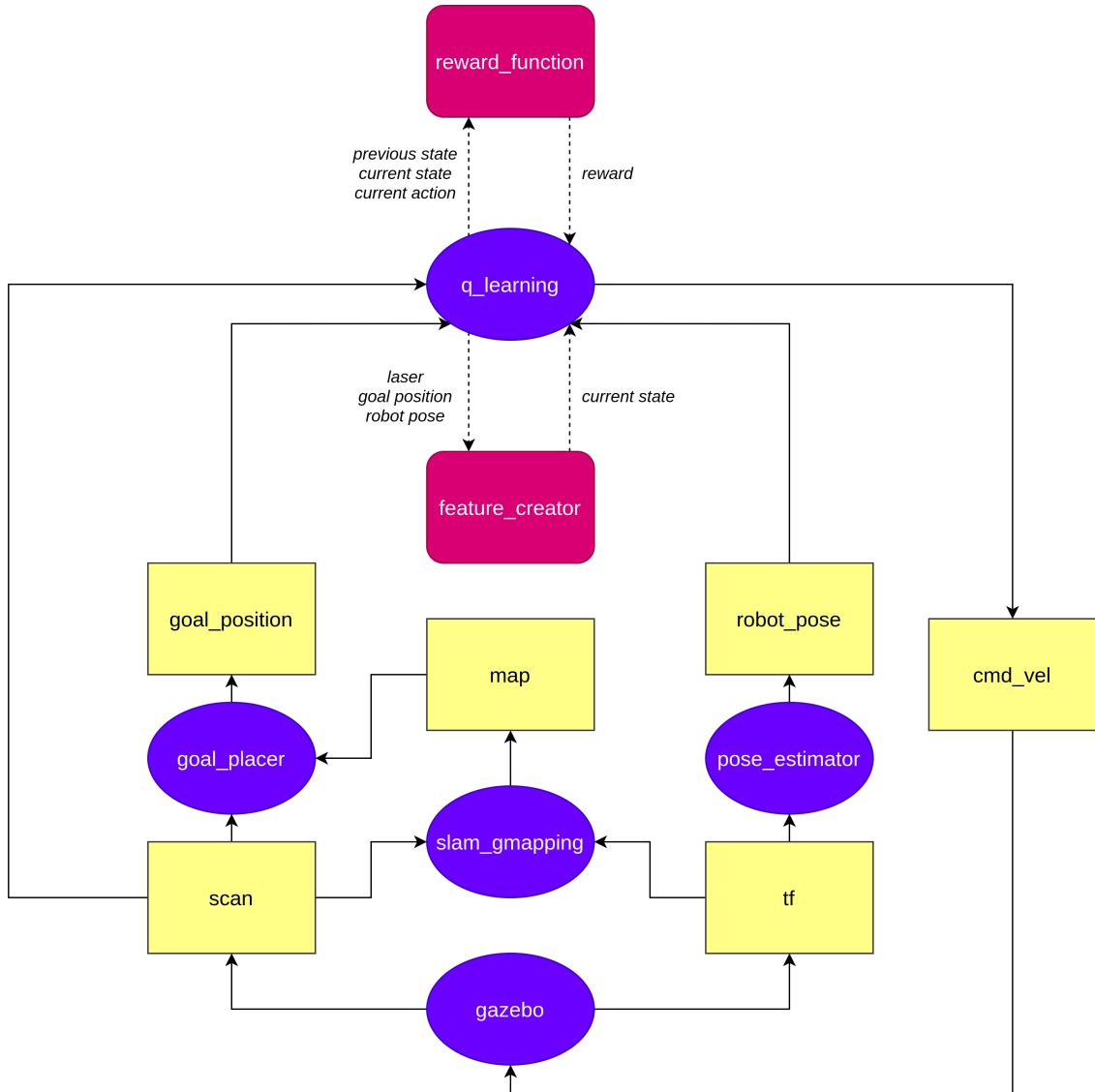


Figure 4.1: An UML-based illustration of the implemented RL-ROS integrated system. In the figure, *nodes*, *topics* and auxiliary *modules* are in blue, yellow and pink, respectively. Solid arrows ($\longrightarrow$) represent subscribe-publish relations between *nodes* and *topics* and dashed arrows ($\dashrightarrow$) represent interchanging of data between a *node* and its auxiliary *module*. A *module* is an object outside the main ROS network that performs isolated computation.

Focusing now on describing the elements of the system, firstly, in blue and yellow, there are represented the **nodes** and **topics**, respectively. The *gazebo* node is responsible for simulating the physics of the environment, including the model of the robot (its hardware, its physical body and its kinematics). It publishes the necessary real-time laser scan measurements in the topic *scan* and the necessary transforms related to mathematical frames of different objects in the map, useful for geometrical computation in the *tf* topic. The *slam_gmapping* node subscribes the *scan* topic to

extract the necessary information from the laser measurements to build and publish the occupancy grid representation (a compact representation of the map information in discretized cells) in the topic *map*. It performs laser-based SLAM⁶ (Simultaneous Localization and Mapping) to create the map. The *pose_estimator* node extracts the necessary frame-related information from the subscribed topic *tf* and publishes the robot's pose in the *robot_pose* topic. The *goal_placer* node is responsible for choosing new random and episodic goal objective positions to be published in the *goal_position* topic. The *q_learning* node implements and performs the agent's behavior for the Q-learning algorithm. It reads the current state and the reward produced by the neighboring *feature_creator* and *reward_function* modules, respectively, and publishes the decided action as velocity steering commands in the topic *cmd_vel* to be finally processed and drawn by the *gazebo* node in its physical environment simulation.

Lastly, in pink, there are represented the **modules**: Modules are auxiliary scripts outside the main ROS network, with the purpose of isolating programming code and to give modularity to the system. The *feature_creator* module extracts the robot's pose, the goal position and the laser scan information in order to build the required data structure representing the current agent's state (the state's composition is shown in 4.1.2.2). The *reward_function* module takes the agent's current state built by the *feature_creator* module, the previous state stored in memory from the previous step and the current chosen action to compute the scalar reward representing the immediate value of the transition. This reward scalar signal together with the current state are the necessary ingredients for the agent to learn about its past decision and to choose its next action.

4.1.2 Q-table: The "brain"

The Q-learning algorithm stores its learning experience in a table, hence its labelling as a tabular method. The Q-table, as represented in Figure 4.2, stores a Q-value for every possible state-action pair. Each Q-value ($Q(s, a)$) is an entry, representing, as already described in previous sections, the value of performing an action $a \in \mathbb{A}(s)$ in a state $s \in \mathbb{S}$ and following policy q_π forever after. One can conclude that, the greater the resolution of the state or action space, the greater the size of the Q-table. The curse of dimensionality (exposed in Section 2.2.1) plays its role in this size aspect because when the state space is highly dimensional, typical from robotics domains and real-world problems, the table starts to become dramatically big, compromising the convergence properties. A big Q-table requires an huge amount of time to be entirely explored, therefore, making it hard or impossible for Q-learning to converge. Not only Q-learning needs to explore entirely its Q-table, but also continually visit again and again each Q-value to guarantee individual convergence. In small state-spaces, although, it is clear that it is feasible and applicable to learn Q-tables.

For this dissertation work, the challenge of applying Q-learning to a robotics problem, demands as much low-dimensionality as possible for the state-space. Not to forget that, since the action-space is the second dimension of the Q-table, it also needs to be as much low-dimensional

⁶Slideshow presentation about SLAM: <http://ais.informatik.uni-freiburg.de/teaching/ss12/robotics/slides/12-slam.pdf>

as possible without compromising the agent's control capabilities⁷. The Q-table needs to be as small as possible, hence, essentially, the challenge will consist of choosing the best state representation for the robot without compromising the awareness of the situation⁸. In the following subsections, it will be introduced the discretization technique applied to the state-space, the state-space representation and the action-space representation.

Q Table	a_1	a_2		a_{N_a-1}	a_{N_a}
s_1	$Q_{s_1 a_1}$	$Q_{s_1 a_2}$			$Q_{s_1 a_{N_a}}$
s_2	$Q_{s_2 a_1}$				
$\vdots$	$\vdots$				
s_{N_s-1}					
s_{N_s}	$Q_{s_{N_s} a_1}$				$Q_{s_{N_s} a_{N_a}}$

Figure 4.2: Q-table illustration. N_s state rows and N_a action columns. The complexity of the Q-table is $N_s \times N_a$. N_s is the dimension of the state-space. N_a is the dimension of the action-space.

4.1.2.1 Discretization

To store the Q value for every state-action pair in the table a proper discretization technique is needed. Nearest Neighbor⁹ is a simple and fast technique where a number is transformed into its nearest discrete neighbor from a finite set of numbers. Formally, for every number x and for each discrete number z_i belonging to the discrete set Z with length N_z ,

$$Z = \{z_0, \dots, z_{N_z-1}\},$$

the Nearest Neighbor function $NN(x, Z)$ takes the value

$$NN(x, Z) = \underset{z \in Z}{\operatorname{argmin}} \|x - z\|. \quad (4.1)$$

An example of the applied technique is shown in the Figure 4.3.

⁷The agent needs to have access to reasonable action-spaces of proper length, to be able to solve the task.

⁸The state needs to represent everything that the agent needs to know to be aware of its situation, therefore, it requires a lot of intuition and a challenging feature selection for the programmer.

⁹Nearest Neighbor technique is exposed at: <https://people.eecs.berkeley.edu/~pabbeel/cs287-fa12/slides/discretization.pdf>

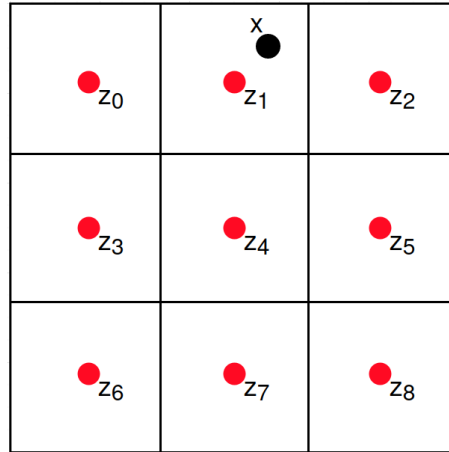


Figure 4.3: Nearest Neighbor example. Given $Z = \{z_0, \dots, z_8\}$, $NN(x, Z) = z_1$.

4.1.2.2 State Space

The state of the agent needs to be representative of the context of the problem, meaning that it needs to encode the right amount of information to represent the situation of the agent in the environment. The process of seeking for the right state features is known as *feature engineering*¹⁰. This process requires domain knowledge, a lot of intuition and a lot of calibration, because finding the right features for the problem is more like art than science. The state space complexity (S_c) is the first dimension of the Q-table and so the state's complexity directly influences the complexity of the Q-table. According to our intuition, the chosen features for the context of this problem need to take into account:

1. The direction to the goal.
2. How close the agent is from an obstacle.
3. The direction to an obstacle.

The enumerated reasons above consider two important aspects for the problem: In the first place, the agent must be able to reach the goal so it needs to be aware of the angle (or direction) to the goal position. Secondly, the agent must be able to avoid collisions and so it needs to sense in some way the distance and the direction to obstacles in the neighborhood. For that, using 2D LiDAR perception, the lasers are processed to build an occupancy line of sight representation to represent the occupancy of the neighborhood. Further, in more detail, this processing step will be presented.

The angle to the goal (θ_g), and the line of sight (*los*) are the chosen features for the state of the agent. Therefore, the state of the agent s is described as a tuple, as follows:

$$s = (\theta_g, los).$$

¹⁰Useful considerations about the process of feature engineering, available at: <https://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/>

Figure 4.4 illustrates the complete state representation whose rational and variables are presented in sequel.

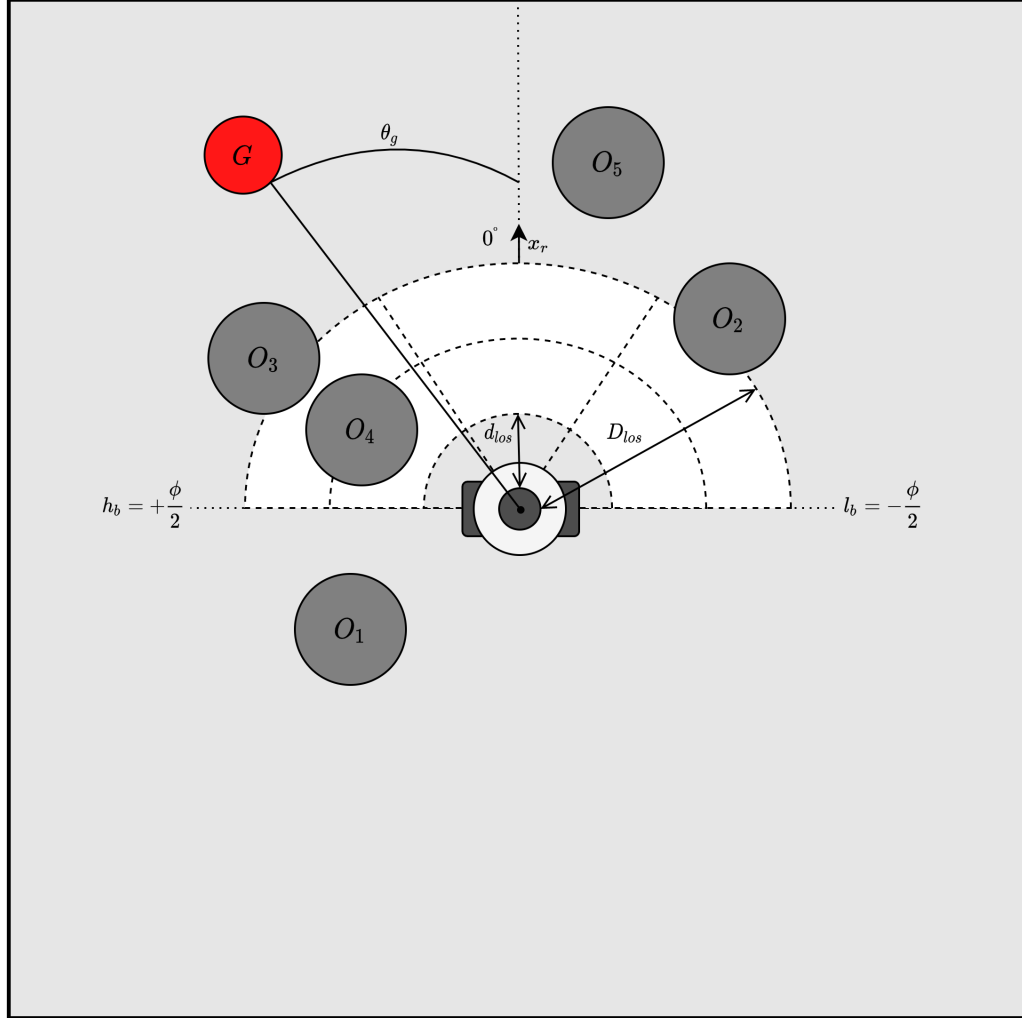


Figure 4.4: Illustration of the complete state representation. In this example, $r = 3$, $\phi = 180^\circ$, $b = 4$. The angle to the goal G is somewhere around 40 degrees ($\theta_g \approx 40^\circ$). The los is defined as follows: it can not perceive the obstacle O_1 since it is out of the sight. O_2 is occupying the third bound in the second distance level. The second bound is free since the distance to O_5 exceeds h_r . The first bound is occupied by the obstacle O_4 in both first and second distance levels, and by the obstacle O_3 in the second level. Prioritization is given to obstacle O_4 since it is the closest obstacle and it is causing the average of the laser distances to be contained in the first distance level.

Dimensionality and Discrete sets The dimensionality of the θ_g is the number of angular bins, that is, N_{θ_g} . N_{θ_g} needs to be chosen so as to be centered in 0° , therefore it needs to be an odd number. It is also bounded by a fixed maximum angle of $h_{\theta_g} = 180^\circ$ and a fixed minimum angle of $l_{\theta_g} = -180^\circ$.

The dimensionality of the los is dependent on its range (r) and angular (b) resolution. Let $r \in [1, \infty[\subset \mathbb{N}$ be the number of discrete distance bins perceived for each bound, and $b \in [1, \infty[\subset \mathbb{N}$ the

number of discrete angular bins inside a symmetric angular window space of length $\phi \in]0^\circ, 360^\circ]$ between a left and right bound of $-\frac{\phi}{2}$ and $\frac{\phi}{2}$, respectively. Specifically, an *los* consists of a tuple of length b . Each element of the tuple is a discrete distance value for a specific angular region. At most, the agent perceives a maximum distance defined by the maximum laser range capabilities, that is, L_{max} (upper bound). It is also limited by the minimum distance perceived by the laser, that is, L_{min} (lower bound). Let $\sigma \in [\frac{L_{min}}{L_{max}}, 1]$ be the parameter that controls the fraction of the maximum distance perceived by the agent. Therefore, D_{los} , the effective maximum distance perceived, is defined by:

$$D_{los} = \sigma L_{max}. \quad (4.2)$$

The same applies for the effective minimum distance d_{los} perceived by the agent. It needs to be defined, as follows:

$$L_{min} < d_{los} < D_{los}. \quad (4.3)$$

Finally, the dimensionality¹¹ of the *los* is defined by:

$$N_{los} = r^{b-1}.$$

Therefore, the state space complexity S_c (the first dimension length of the Q-table) is defined by:

$$S_c = N_{\theta_g} \times r^{b-1} = N_{\theta_g} \times N_{los}. \quad (4.4)$$

Each feature is discretized following the former mentioned resolutions. The discrete space for each feature is bounded as follows:

- Angle to the goal (θ_g):
 - Lowest angle to goal: l_{θ_g} , bounded between -180° and h_{θ_g} , that is, $l_{\theta_g} \in [-180^\circ, h_{\theta_g}[$
 - Highest angle to goal: h_{θ_g} , bounded between l_{θ_g} and 180° , that is, $h_{\theta_g} \in]l_{\theta_g}, 180^\circ]$
- Line of sight (*los*):
 - Distance of *los*:
 - * Minimum distance: (d_{los}), bounded between the laser's minimum range capabilities (L_{min}) and maximum effective distance of the *los* (D_{los}), that is, $d_{los} \in [L_{min}, D_{los}[$
 - * Maximum distance: (D_{los}), bounded between the minimum range for the *los* (d_{los}) and the laser's maximum range capabilities (L_{max}), that is, $D_{los} \in]d_{los}, L_{max}]$.
 - Angular window for the *los*. The bounds are limited between the following angles:
 - * Lowest angle: $l_b = -\frac{\phi}{2}$

¹¹ b is subtracted by 1, because we are interested in a *los* configuration where we represent angular regions instead of angles. Hence, we have $b - 1$ angular regions for b angular bins. It is just a representation detail.

* Highest angle: $h_b = \frac{\phi}{2}$

We then build the discrete set for each feature as in Algorithm 1.

Algorithm 1 Building the discrete sets for θ_g and los

Input: Resolution of θ_g (N_{θ_g}), Lowest angle to goal (l_{θ_g}), Highest angle to goal (h_{θ_g}), Number of distance bins for the los (r), Number of angular bins for the los (b), Angular window (ϕ), Effective maximum distance perceived by the los (D_{los}), Effective minimum distance perceived by the los (d_{los}).

```

1: Initialize an empty array  $B$  of length  $b$ , an empty array  $R$  of length  $r$  representing the discrete
   sets for the  $los$ . Initialize an empty array  $\Theta$  of length  $N_{\theta_g}$  representing the discrete set for the
    $\theta_g$ .
2:  $l_b \leftarrow -\frac{\phi}{2}$ 
3:  $h_b \leftarrow \frac{\phi}{2}$ 
4: for  $i = 0; i < b; i += 1$  do
5:    $B[i] \leftarrow \frac{i(h_b - l_b)}{b-1} + l_b$ 
6: end for
7: for  $i = 0; i < r; i += 1$  do
8:    $R[i] \leftarrow \frac{i(D_{los} - d_{los})}{r-1} + d_{los}$ 
9: end for
10: for  $i = 0; i < N_{\theta_g}; i += 1$  do
11:    $\Theta[i] \leftarrow \frac{i(h_{\theta_g} - l_{\theta_g})}{N_{\theta_g} - 1} + l_{\theta_g}$ 
12: end for

```

To support the exposition of the algorithm 1, as an example, let us define a discrete set B of $b = 4$, $h_b = 90^\circ$, $l_b = -90^\circ$. Additionally let us define a discrete set R of $r = 3$ distance bins, $d_{los} = 0.11$, $\sigma = 0.1$, and therefore $D_{los} = 0.35$. Lastly, a $N_{\theta_g} = 13$. The resultant discrete sets are defined as follows:

$$B = \{-90, -30, 30, 90\},$$

$$R = \{0.11, 0.23, 0.35\},$$

and

$$\Theta = \{-180, -150, -120, -90, -60, -30, \\ 0, 30, 60, 90, 120, 150, 180\}.$$

Estimation We now describe the estimation steps for each feature.

To obtain θ_g , we need to take into account the robot pose (r_x, r_y, r_θ) and the goal position (g_x, g_y) . First, we need to get the goal position in the robot frame and then apply the trigonometric

arctangent2 function, as follows:

$$\begin{bmatrix} g'_x \\ g'_y \end{bmatrix} = \begin{bmatrix} \cos(r_\theta) & \sin(r_\theta) \\ -\sin(r_\theta) & \cos(r_\theta) \end{bmatrix} \begin{bmatrix} g_x - r_x \\ g_y - r_y \end{bmatrix}, \quad (4.5)$$

and

$$\theta_g = \arctan2(g'_y, g'_x). \quad (4.6)$$

Lastly, we perform the discretization step using the discrete set Θ , as follows:

$$\theta_g \leftarrow NN(\theta_g, \Theta) \quad (4.7)$$

To obtain the *los*, we need to process the laser scan to obtain the desired information distances measured by the LDS-01 sensor placed in the top of the robot¹². Thanks to the turtlebot3's native ROS-*nodes*¹³ we have access to the *scan* topic that makes available an array containing the distances measured by the LIDAR device. With an angular resolution of 1° and an angular range of 360° we have access to the distance by each of the 360 degrees with a refresh rate of approximately 5Hz. The distance range of the laser is bounded between $L_{min} = 120mm$ and $L_{max} = 3500mm$. We decided that the robot does not need to have knowledge about the complete laser scan information. We intuitively assumed that with a restricted angular window, the robot would have a compact and sufficient amount of information to interpret its situation properly. The intuition is inspired by the biological field of view, and by the way humans perceive the environment. In practice, we transferred that intuition by applying angular windows smaller than 180 degrees ($\phi \leq 180^\circ$).

Let us define a laser set array as $L = \{l_0, \dots, l_{359}\}$. The distances will be bounded by a minimum and a maximum value of 0.12 and 3.5 (the distance range mentioned above), respectively, in meters. We will give more importance to closer rays, by applying a ramp filter function $w(d)$ (as illustrated in Figure 4.5) with the purpose of properly weighting the distance for each ray. This weighting ranges from 0 to 1, assuming $w(0.12) = 1$ and $w(3.5) = 0$.

¹²Turtlebot3's Laser Distance Sensor device information: https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/

¹³The Turtlebot3's package available at: <https://github.com/ROBOTIS-GIT/turtlebot3>

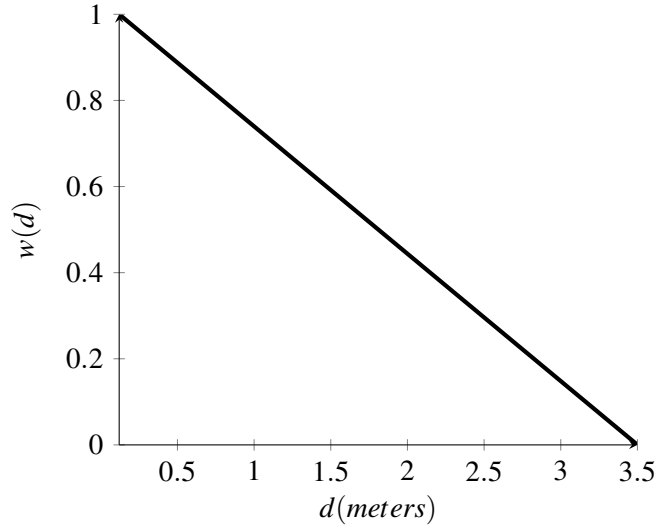


Figure 4.5: A ramp filter symbolizing a degree of importance as a function of distance. The lower the distance the higher is the importance we need to give since closer obstacles (closer distances) are more urgent to take into account.

After filtering all the rays in L , we obtain W , the weight vector, where $w_i \in W$ is the corresponding weight for each laser ray distance $l_i \in L$.

Let's also define B as the set of angular bins, for each angle $\theta_i \in \Theta$ as follows:

$$B = \bigcup_{i=0}^{b-1} \{\theta_i, \theta_{i+1}\}. \quad (4.8)$$

For each region $b_i \in B$, we apply the filter to obtain isolated subsets $W' \subset W$ of weights.

We finally compute a weighted average using the correspondent W' for all l_i in a specific angular region. This weighted average is the estimated distance for a specific angular region in the los , let us call it d_o , as follows:

$$d_o = \frac{\sum_{i=1}^N w_i l_i}{\sum_{i=1}^N w_i}. \quad (4.9)$$

Additionally, for each d_o , we perform the discretization step, using the discrete set R , that is,

$$d_o \leftarrow NN(d_o, R). \quad (4.10)$$

Finally, each d_o correspondent to each angular region b_i is inserted into the los array, as we expose in more detail in the Algorithm 2.

Algorithm 2 Estimating the *los*

Input: Robot position (r_x, r_y, r_θ) , Goal position (g_x, g_y) , Laser scan array (L), Minimum effective distance perceived (d_{los}), Maximum effective distance perceived (D_{los}), Angular discrete set of the *los* (B), Distance discrete set of the *los* (R), Number of bounds of the *los* (b).

```

1: function RAMP FILTER( $\theta_0, \theta_1$ )
2:   Initialize empty array  $W$  of weights
3:    $s \leftarrow 0$  ▷ Sum of the weights variable
4:    $d \leftarrow 0$  ▷ Distance variable
5:    $\theta \leftarrow 0$  ▷ Angle variable
6:    $m \leftarrow \frac{1}{d_{los} - D_{los}}$ 
7:    $b \leftarrow -md_M$ 
8:   for  $i = \theta_0$  to  $\theta_1$  do ▷ Build weight array
9:      $W[i] \leftarrow m \times L[i] + b$ 
10:     $s \leftarrow s + W[i]$ 
11:  end for
12:  for  $i = \theta_0$  to  $\theta_1$  do ▷ Estimate distance to obstacle in the bound  $(\theta_0, \theta_1)$ 
13:     $d \leftarrow d + \frac{W[i] \times L[i]}{s}$ 
14:  end for
15:  return  $d$ 
16: end function
17: function ESTIMATE los
18:   Initialize empty array los
19:    $B \leftarrow -B$  ▷ Reverse  $B$  so as to build the los from higher angles to lower angles.
20:   for  $i = 0; i < (b - 1); i += 1$  do
21:      $\theta_0 \leftarrow B[i]$  ▷ First angle bound in degrees
22:      $\theta_1 \leftarrow B[i + 1]$  ▷ Second angle bound in degrees
23:      $d_o \leftarrow RampFilter(\theta_0, \theta_1)$ 
24:      $d_o \leftarrow NN(d_o, R)$  ▷ Discretize the real estimated distance by applying the Nearest Neighbor technique
25:      $los[i] \leftarrow d_o$  ▷ Append to los
26:   end for
27:   return los
28: end function

```

4.1.2.3 Action Space

The action space $\mathbb{A}$ for our Q-learning architecture consists of a set of discrete angular velocities, thus we need to choose the proper amount N_a of actions since A will be the second dimension of our Q-table. The action set available for the agent consists in an odd quantity N_a of angular velocities $\omega_i \in \mathbb{A}$ representing the steering actions available for the navigation. Let's assume a

maximum value of angular velocity a_M and define a function $\Omega(x)$ that generates angular velocities, as follows:

$$\Omega(x) = x \frac{a_M}{\lfloor \frac{N_a}{2} \rfloor}. \quad (4.11)$$

Finally, the action set A is defined as follows:

$$A = \left\{ \Omega(x) \mid x \in \mathbb{N}_0 \wedge x < \left\lfloor \frac{N_a}{2} \right\rfloor \right\} \cup \left\{ -\Omega(x) \mid x \in \mathbb{N} \wedge x < \left\lfloor \frac{N_a}{2} \right\rfloor \right\}. \quad (4.12)$$

Finally, combining the state space complexity (S_c) and action space complexity ($A_c = N_a = |A|$), we have a total complexity of T_c , that is,

$$\begin{aligned} T_c &= S_c \times |A| \\ &= S_c \times A_c. \end{aligned} \quad (4.13)$$

4.1.3 Reward function

In RL, the purpose or goal of the agent is formalized in terms of a special reward signal passing from the environment to the agent, as we already described in subsection 3.1.2. Designing a reward function consists of the exercise of mathematically representing the main objective of a task. Conceptually, not only the reward needs to describe the goal, but it also needs to be relatively shaped to speed up and improve the convergence of the algorithm. To give some intuition, an example of a shaped reward function, in other words, some reward signal that is given at any time step, could be represented by:

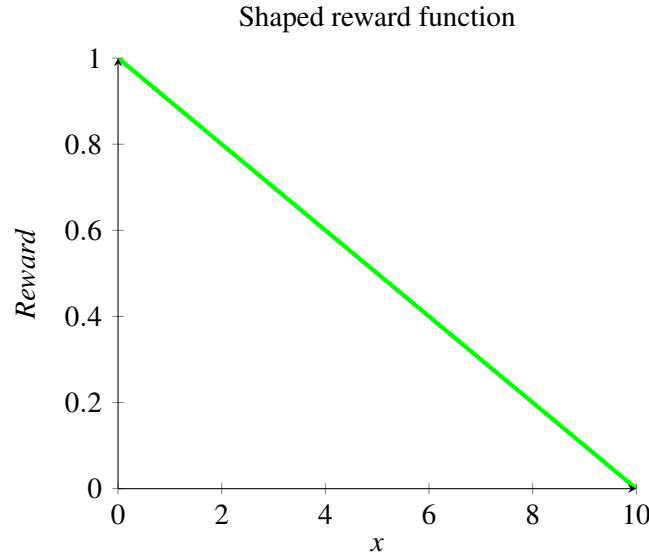


Figure 4.6: Shapeness described by always having a reward signal for every x . Shapeness is also described by having gradient.

A desirable reward design, with less effort for the programmer, would be to give only sparse rewards. If the agent achieves the goal, the programmer could give a reward of $+1$, and if failure

a reward of -1 . However in the majority of the problems, using pure Reinforcement Learning approaches, the sparseness of the reward inhibits convergence, not providing enough feedback for learning value functions. Intuitively, a sparse reward function can be represented as follows:

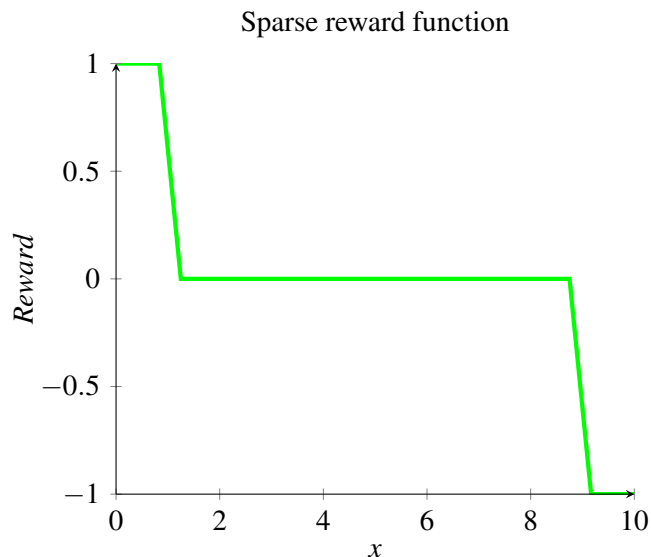


Figure 4.7: Sharpness described by not always having a reward signal for specific values of x (in this example, there is no signal between 1 and 9, that is, $x = 0$ if $1 \leq x \leq 9$). Sharpness is also described by not having any gradient or being zero for different values of x (in this example, $x(4) = x(6)$).

Reward design¹⁴ is a really hard exercise for more complicated tasks (e.g., robotic grasping, robotic lifting, robotic walking...), where it is really hard to describe a proper signal for such a long and complex task, that requires a lot of steps and a lot of minor task achievements inside of the bigger main task. If the programmer wants to classically approach the rewarding problem, well engineered reward shaping is vital, requiring a lot of human knowledge, calibration and trial-and-error, as achieved by this study [78] from researchers at Berkeley. However, more advanced, natural, less tiring and modern approaches are currently being developed. Integrating reward shaping with human demonstrations is shown to speed up training as shown by this study [79]. Using modeled reward predictors, a process known as *reward modelling*¹⁵, is another promising approach where the reward function is an independent and parameterised model that learns by itself what should be the reward function given human feedback on the executed behavior, as developed by this study [80; 81]. It is also possible to train an agent only with sparse rewards, as achieved successfully by this study [82], where a really sample-efficient and off-policy technique called Hindsight Experience Replay is applied.

¹⁴A concise and intuitive description of reward designing is given by the following article, available at: <https://medium.com/@BonsaiAI/deep-reinforcement-learning-models-tips-tricks-for-writing-reward-functions-a84fe525e8e0>.

¹⁵Reward modelling is concisely described in this article: <https://medium.com/@deepmindsafetyresearch/scalable-agent-alignment-via-reward-modeling-bf4ab06dfd84>

For our own reward design, we chose a classic reward shaping path given the simplicity of the task. To achieve the desired behaviour for our robotic obstacle avoidance navigation task, we consider the following:

1. The agent needs to feel the orientation to the goal.
2. The agent needs to be strongly awarded when reaching the goal.
3. The agent needs to know that it should avoid obstacles if it is near them.
4. The agent needs to be strongly penalized when colliding with obstacles.

Taking these considerations into account we thought of breaking the reward into four components ($R_{gr}, R_{prize}, R_{oa}, R_{collision}$), each to fulfill each consideration.

Consideration 1 describes the problem of goal reaching (R_{gr}), and it is responsible for rewarding positively the agent for actions that makes it head more towards the goal, and penalizing it if it deviates from heading towards the goal. To achieve 1, we thought of an error between the angle to the goal in the current state θ_g and the angle to the goal in the state afterwards θ'_g . After a transition, if this error is positive, in other words, if the angle to the goal is reduced ($|\theta'_g| < |\theta_g|$), then the agent is heading more towards the goal. This error value is the first component of the reward and it fulfills the consideration 1, because it takes care of the goal reaching problem when there are no obstacles in sight. However, if there are any obstacles in sight, we switch to the R_{oa} component, as we describe further. The error is defined by the difference of the absolute values of the angles, that is,

$$R_{gr}(s, s') = |\theta_g| - |\theta'_g|. \quad (4.14)$$

Consideration 2 describes the prize for successful goal reaching, in other words, if the agent reaches 0° of orientation towards the goal, it should be successfully rewarded (R_G). To achieve 2, we defined a relatively strong and positive scalar G . When reaching the goal, the agent strongly feels the achievement, as follows:

$$R_{prize}(s, a) = G. \quad (4.15)$$

Consideration 3 describes the problem of obstacle avoidance, being responsible for defining what are the right actions when the agent is facing a specific line of sight (los) and angle to the goal (θ_g) configuration. Actions that lead the agent closer to obstacles are more penalized, and vice-versa. Obstacle avoidance should also take into account the direction to the goal, that is, actions that lead the agent away from obstacles and towards the right orientation to the goal, should be more premiated than actions that lead the agent away from obstacles and deviate from the right orientation to the goal. To respect the former conditions, we thought of weighting ($k \in [0, 1]$) this former considerations into one unique reward component, that is, R_{oa} . The stronger is k the more degree of importance for the obstacle avoidance component. This weighted reward is described as follows:

$$R_{oa}(s, a, s') = k r_{oa}(s, a) + (1 - k) R_{gr}(s, s'). \quad (4.16)$$

Consideration 4 is defined by a relatively strong and negative scalar. When a collision happens, the agent strongly feels the "pain". When in a state s , if the action a taken lead the agent to a terminal state s_T , then there is a reward component, that is $R_{collision}$, defined by:

$$R_{collision}(s, a) = C. \quad (4.17)$$

Finally, the proposed reward function is represented by the following piecewise function:

$$R(s, a, s') = \begin{cases} R_{gr} & \text{if no obstacles in sight} \\ R_{prize} & \text{if } \theta'_g == 0^\circ \\ R_{oa} & \text{if obstacles in sight} \\ R_{collision} & \text{if collided} \end{cases} \quad (4.18)$$

In more detail, the algorithm for the reward function¹⁶ is exposed, as follows:

Algorithm 3 Reward function: $R(s, a, s')$

Input: Goal big prize (G), Collision penalty (C), Gain (k)

```

1: function REWARD( $s, a, s'$ )           ▷ Receives the agent's current state, action and next state
2:   if goal was reached then
3:     return  $G$                            ▷ The  $R_{prize}$  component
4:   else if collision happened then
5:     return  $C$                            ▷ The  $R_{collision}$  component
6:   else if no obstacles in sight in  $s$  or  $s'$  then           ▷ If the line of sight ( $los$ ) is not detecting
    obstacles neither in  $s$  or  $s'$ 
7:     return  $|\theta_g| - |\theta'_g|$            ▷ The  $R_{gr}$  component
8:   else                                   ▷ If obstacles in sight
9:      $a_{deg} \leftarrow a \frac{180}{\pi}$            ▷ Convert action in  $rad/s$  to  $degrees/s$ 
10:     $idx \leftarrow digitize(a_{deg}, B)$    ▷ Identifies the position of  $a_{deg}$  with respect to the intervals
    defined by set  $B$ 
11:     $idx \leftarrow clip(idx, 1, b - 1) - 1$            ▷ Saturates  $idx$  between 1 and  $b - 1$ 
12:     $los_d \leftarrow digitize(los[idx], R) - 1$    ▷ Identifies the position of the current  $los[idx]$  with
    respect to the intervals defined by set  $R$ 
13:     $r_{oa} \leftarrow \frac{los_d - (r - 1)}{r - 1}$ 
14:    return  $kr_{oa} + (1 - k)(|\theta_g| - |\theta'_g|)$ 
15:  end if
16: end function

```

Lastly, for the choosing of the reward values, we were motivated by both a relative aspect and

¹⁶We use two external function from the *numpy* library in our reward function, called *digitize* and *clip* available at: <https://numpy.org/doc/stable/reference/generated/numpy.clip.html> and <https://numpy.org/doc/stable/reference/generated/numpy.digitize.html>

an absolute aspect. The relative aspect is related to the proportionality between different reward components, meaning that, a reward component that is capable, at maximum, of returning the double of the value as another reward component, is twice more "important" than the other. The absolute aspect is related to the way we choose the values for the reward signals, and how we bound them.

We chose a $R_{prize} = 1.0$ and a $R_{collision} = -2.0$. Hence, we give more significance to the fact that colliding is twice more critical than reaching a goal, therefore, speeding up the behavior towards the obstacle avoidance component relative to the goal reaching component during training. The other components (R_{gr}, R_{oa}) were bounded by $[R_{collision}, R_{prize}]$ so as to conceptually associate the stronger absolute reward value bounds to the terminal states¹⁷. The goal reaching component R_{gr} is bounded between -1 and 0 , that is, $R_{gr} \in [-1, 0]$. Actions that make the agent pull away from the right orientation to the goal, at maximum, are penalized with a -1 reward. On the other hand, actions that make the agent closer to the desired orientation to the goal (0°) are rewarded, at maximum, with a reward of 0 . Finally, we bound our obstacle avoidance component between -1.5 and 0 , that is, $R_{oa} \in [-1.5, 0]$. Actions that make the agent pull away from obstacles and more towards the right orientation to the goal, at maximum are rewarded with a reward of 0 . The worst case is for actions that lead the agent closer to obstacles and away from the right orientation to the goal, being rewarded, at maximum with a reward of -1.5 .

4.1.4 Algorithm

In this subsection, we give an explanation over important aspects of some remaining key elements of our implementation, the hyperparameters (α, γ, ϵ and E), the exploration policy strategy and the training metrics that we will use to evaluate the trained performance.

Number of episodes The training experience is generally sustained under an episodic setting, a finite amount of maximum steps or some other stopping criteria. We implemented our training under an episodic setting where an episode ends when the robot collides with an obstacle (on failure) or when the robot reaches the goal (on success). The amount of episodes needed is manually tuned, because it is not possible to deduce from the start how many trials are needed order to assure convergence (the convergence is explained in subsection 3.2.4). A low amount of episodes could not guarantee convergence. On the other hand, a huge amount of episodes could be just a waste of time and computational resources. Hence, this E hyperparameter needs to be calibrated until we achieve the desired performance indicated by the training and testing metrics (the metrics are exposed in subsection 1.2.3).

Learning-rate The α parameter controls the speed of training. As we saw earlier, the Q-learning update consists in updating a specific $Q(s, a)$ by summing an error term to its previous value. Let

¹⁷Terminal states, for our state configuration, are those associated with $\theta_g = 0^\circ$ and the configurations of the *los* that encounter collision situations.

δ , known as the TD-error be:

$$\delta \doteq r + \gamma \max_{a'} Q(s', a') - Q(s, a). \quad (4.19)$$

The error term is propagated depending on $\alpha \in [0, 1]$ as:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta. \quad (4.20)$$

We can observe clearly the effect of α since, low values of alpha will make training slower and more cautious. On the other hand, higher learning rates will propagate the learned behavior quicker. Generally, smaller learning rates ($\alpha \in [0.001 - 0.1]$) are preferable since a smoother and prudent training is more robust against the training variance caused by the interchanging between exploration and exploitation strategies. Some scheduling on the learning rate parameter, as already exposed in subsection 3.2.4, could be applied to avoid oscillations when the Q-values are near the final value. We applied the local-clock learning rate decaying schedule, to fulfill the convergence condition 2.

Discount-factor The γ parameter controls the amount of consideration the agent gives to long-term rewards. As we saw already, theoretically, $Q(s, a)$ is the expected long term reward after executing a in s and following policy Q forever after, as defined in equation 3.13. As we observe from equation 4.19, δ is conditioned by γ . Let's consider a more isolated and directly influenced by γ term called TD-target (η), as follows:

$$\begin{aligned} \eta &= \delta - Q(s, a) \\ \eta &= r + \gamma \max_{a'} Q(s', a') - Q(s, a). \end{aligned} \quad (4.21)$$

η is the value to which our Q-values are converging to. In other terms, the TD-target could also be defined by:

$$\eta_t = R_{t+1} + \gamma \max_{a'} \mathbb{E}_Q [G_{t+1} \mid S_{t+1} = s', A_{t+1} = a'], \quad (4.22)$$

that is, the higher γ , the higher the consideration we have for the expectation for the long-term rewards while following the policy Q . The lower the γ , the higher consideration we have for the short-term rewards. Generally higher values of $\gamma \in [0, 1]$ closer to 1 are chosen by researchers, and most of the interesting results in the field are achieved when the agent is more far-sighted. To note that we can consider $\gamma = 1$ since our implementation works on an episodic setting¹⁸, hence it $\exists M \in \mathfrak{R} : |G_t| \leq M, t \in \mathfrak{R}$.

Exploration-Exploitation As already exposed in subsection 4.1.4 and 3.2.4, convergence and training efficiency is conditioned by the chosen policy strategy that balances between exploration

¹⁸As already explained in subsection 3.1.2, for episodic settings, the cumulative reward never diverges, even for $\gamma = 1$.

and exploitation. To achieve optimal behavior, the agent needs to totally explore the state-space so it knows how to behave for every state and action pair.

We chose the ϵ -greedy strategy with decaying. We decayed the ϵ parameter from an initial epsilon value ϵ_i until a final epsilon value ϵ_f proportionally to the number of completed episodes e , that is,

$$\epsilon \leftarrow \frac{\epsilon_i}{\epsilon_f - E}e + \epsilon_i, \quad (4.23)$$

to fulfill the convergence condition 1.

To conclude, exploration consists of choosing a random action, pressuring the agent to enter in new regions of the state-action space. The exploitation consists of choosing greedy actions based on the learned behavior. The balance between exploration and exploitation is controlled by the parameter ϵ , as defined by:

$$\pi(s) = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg \max_{a \in \mathbb{A}} Q(s, a) & \text{with probability } 1 - \epsilon \end{cases}. \quad (4.24)$$

Finally, the pseudo-code for Q-learning is shown in Algorithm 4.

Algorithm 4 *Q-learning*: for estimating the $Q \sim q_*$

Input: Build a state space: $S = \{s_0, \dots, s_{N_s-1}\}$, Build an action space: $\mathbb{A} = \{a_0, \dots, a_{N_a-1}\}$, Design a reward function: $R : S \times A \times S \rightarrow \mathfrak{R}$, Choose hyperparameters: Learning rate $\alpha \in [0, 1]$, Discount factor $\gamma \in [0, 1]$, *Exploration probability* $\varepsilon \in [0, 1]$, Choose a number of episodes E , Choose an exploration policy strategy (e.g., ε -greedy, decaying ε -greedy,...)

```

1: procedure Q-LEARNING
2:   Initialize table  $Q : S \times A \rightarrow \mathfrak{R}$  arbitrarily
3:   for  $e = 1$  to  $E$  do
4:     Start in a state  $s \in \mathbb{S}$ 
5:     while  $s$  is not terminal do
6:        $n \leftarrow \text{random number} \in [0, 1]$ 
7:       if  $n \geq \varepsilon$  then
8:          $a \leftarrow \arg \max_{a \in \mathbb{A}} Q(s, a)$ 
9:       else
10:         $a \leftarrow \text{Choose random action } a$ 
11:      end if
12:      Execute  $a$  and observe new state  $s'$ 
13:       $r \leftarrow R(s, s')$ 
14:      if  $s'$  is terminal then
15:         $Q(s, a) \leftarrow Q(s, a) + \alpha(r - Q(s, a))$ 
16:      else
17:         $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a' \in \mathbb{A}} Q(s', a') - Q(s, a))$ 
18:      end if
19:       $s \leftarrow s'$ 
20:    end while
21:  end for
22: end procedure

```

4.2 Results

In this section we present the concrete results of the solved task by the implemented Q-learning algorithm. We start by exposing the training results for an intentionally designed complex scenario in subsection 4.2.1. Next, in subsection 4.2.2, we evaluate the performance. The performance is tested in the simulated environment as exposed in subsubsection 4.2.2.1 and in the real-world, as exposed in subsubsection 4.2.2.2.

4.2.1 Training

In the Figure 4.8 we expose a snapshot from our training environment taken from the visualizer of our implemented simulator (as we introduced in subsection 1.2.4). We chose a really diverse

and complex environment, with obstacles of different diameters in random positions, so as to guarantee that the agent encounters all possible state configurations. A really diverse environment is the ideal type of environment because it is a substrate for proper exploration, gathering all the conditions for a full convergence.

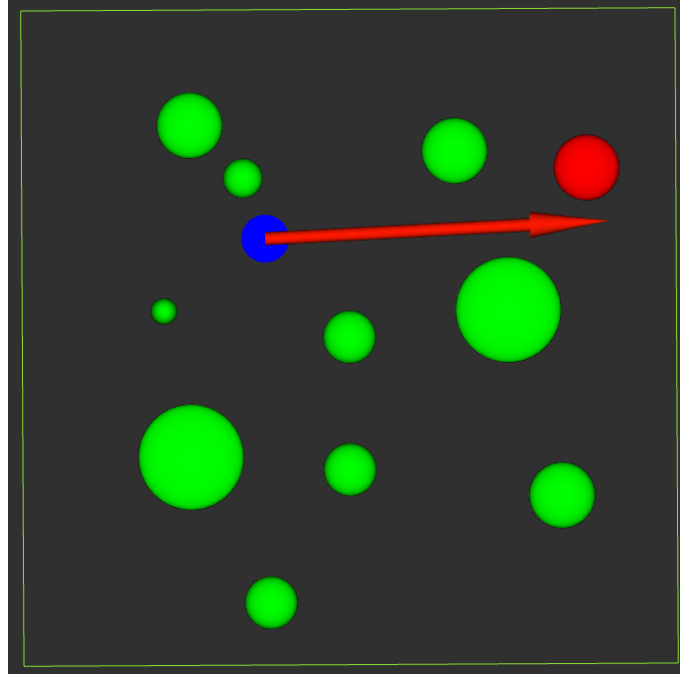


Figure 4.8: A snapshot from our implemented simulator. In blue, we have a circle representing the robot (the circle diameter is equal to the real robot diameter). In green we have the obstacles with different diameters so as to offer variability and facilitate the exploration. In red we have the goal. The red arrow on the top of the robot is its current orientation.

Following, we expose the chosen hyperparameters for the training procedure, in the Table [4.1](#).

Hyperparameter	Value
Episodes	$E = 8000$
Initial Epsilon	$\epsilon_i = 1.0$
Final Epsilon	$\epsilon_f = 0.01$
Linear Velocity (m/s)	$v = 0.2$
Maximum angular velocity (rad/s)	$a_M = 1.4$
Discount Factor	$\gamma = 0.95$
Learning rate	$\alpha = 0.1$
Line of sight distance resolution	$r = 3$
Line of sight angular resolution	$b = 4$
Angle to goal resolution	$N_{\theta_g} = 13$
Number of actions	$A_c = 5$
Goal Prize	$G = 1.0$
Collision Penalty	$C = -2.0$
Angle window	$\phi = 180$
Fraction of maximum distance perceived	$\sigma = 0.1$

Table 4.1: The chosen hyperparameters for training

Let us now expose the results from the training procedure. For each exposed result we have two plots: in the left a raw version and in the right a smoothed version so as to easily visualize the tendency with no noise. Naturally, each exposed result is associated with each training metric (as exposed in subsection 1.2.3).

In Figure 4.9 we present the average TD-error by episode. We can verify a clear reduction of the TD-error during training. It is not only visible a clear reduction in the error between predictions and targets, but we can also verify a clear increase in performance. In the beginning, from $0k$ to $4k$ episodes, collisions were more frequent as we can extract from the amount of values towards $R_{collision} = -2.0$. On the other hand, from the middle to the end, from $4k$ to $8k$ episodes, it is clear to verify the decrease in collisions and the increase in the amount of successful goal prizes received $R_{prize} = 1.0$.

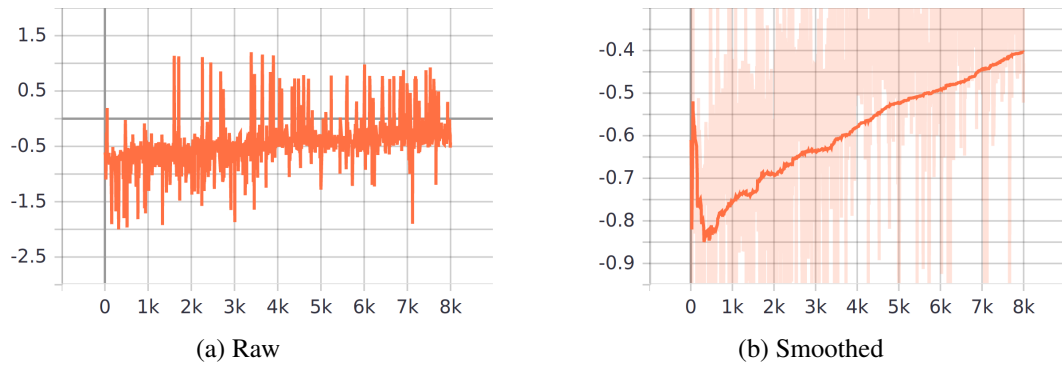


Figure 4.9: The average TD error by episode, according to Metric 1. In the y axis, there is the absolute value of the average of the TD-error in the episode. In the x axis, there is the number of episodes.

In Figure 4.10 we present the cumulative reward by episode. We can verify a clear increase in the amount of collected reward during training. This result could mislead to the fact that the agent is performing worse, because it is collecting more and more negative reward. Although, what it really means is that it is surviving longer, thus accumulating more reward, performing longer paths while avoidance collisions, which is a good aspect. The less amount of cumulative reward in the beginning of training (e.g., 0k to 4k episodes) is due to the fact that it is colliding more often, hence accumulating less reward and ending the episodes earlier.

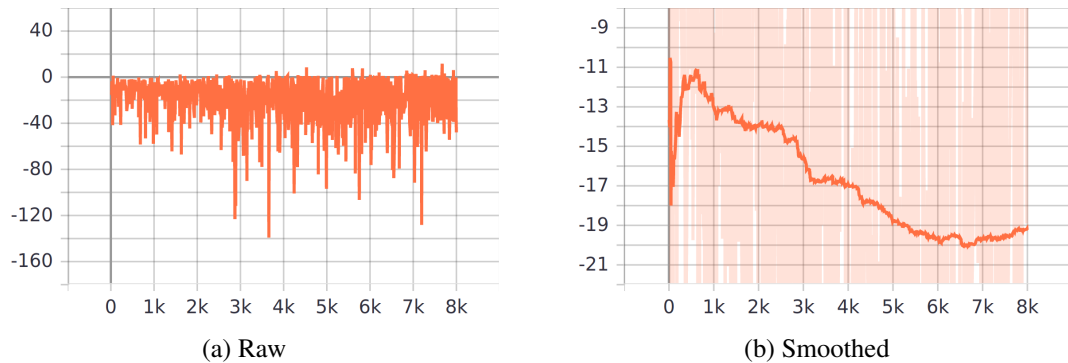


Figure 4.10: The cumulative reward by episode, according to Metric 2. In the y axis, there is the value of the accumulated reward in the episode. In the x axis, there is the number of episodes.

In Figure 4.11 we expose the number of decisions by episode. We can clearly verify an initial small amount of decisions taken in the beginning of training. Also, this result could mislead to the fact that the agent is taking less decisions while being optimal. Although, what is really happening is that the agent is colliding earlier, thus taking less decisions, that were, in fact, worse. Lastly, what the results really tell us is the increase in the surviving time of the agent, and thus taking more and better decisions by episode, while avoiding obstacles along the way.

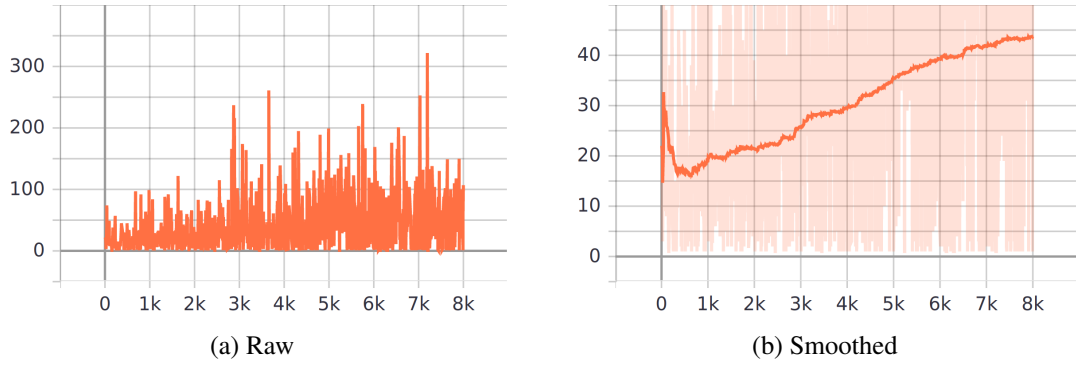


Figure 4.11: The number of decisions/actions taken by episode, according to Metric 3. In the y axis, there is the number of decisions in the episode. In the x axis, there is the number of episodes.

In Figure 4.12, we present the reasons for episode ending by episode. This result confirms and supports every previous exposed result. This plot has binary values: a value of +1 for episode ending on goal reaching, and a value of 0 for episode ending on collision. It is clear an increase in achieved goals as the training goes on supporting the previous presented results by confirming a growing improvement in performance.

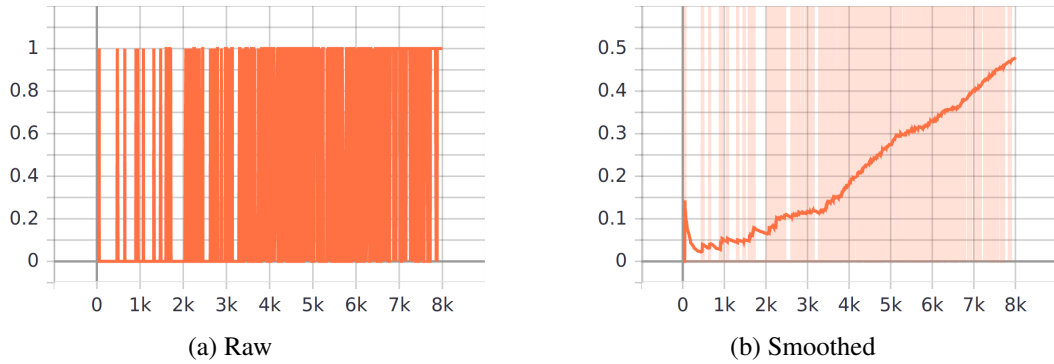


Figure 4.12: The reasons for episode ending by episode, according to Metric 4. In the y axis, there is the binary value representing the ending reason for the episode, that is, a value of 1 if the goal was reached without collisions and a value of 0 if the episode ended because of collision with an obstacle. In the x axis, there is the number of episodes.

In Figure 4.13 we present the first visit exploration percentage by episode. The total complexity of our state-action space allowed a complete first visit exploration. According to equation 4.13, for $r = 3, b = 4, N_{\theta_g} = 13$ and $A_c = 5$ (as extracted from table 4.1), we have a total dimensionality that is,

$$T_c = r^{b-1} \times N_{\theta_g} \times A_c = 1755.$$

For this small complexity, the amount of episodes we selected was sufficient to ensure that every state-action pair is visited at least once. Additionally, we can conclude that, if we had increased the number of episodes, we could guarantee even more visits, improving the convergence.

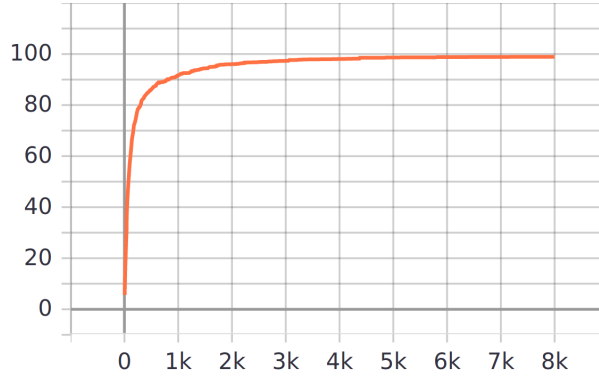


Figure 4.13: The first visit exploration percentage, according to Metric 5. In the y axis, there is the percentage value of the first visit exploration. In the x axis, there is the number of episodes.

4.2.2 Testing

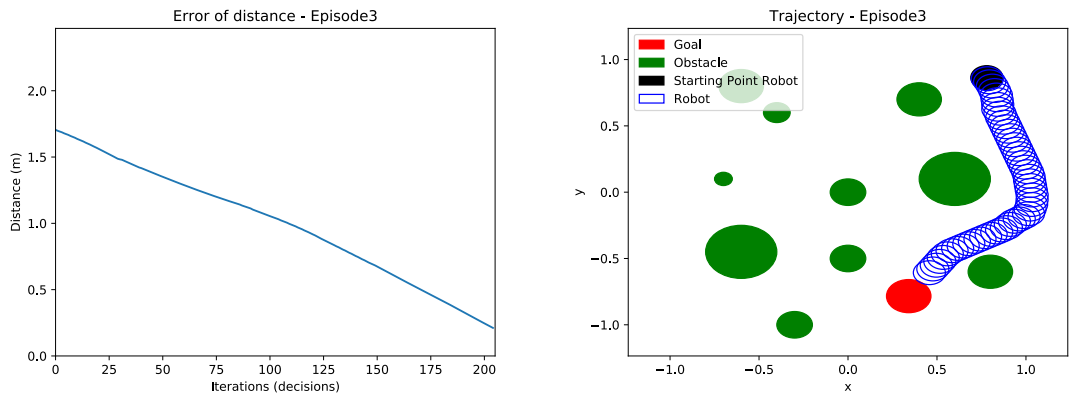
We then tested the learned behavior in the simulated environment and in the real-world. In the following subsections, we will expose and discuss the achieved results. Similarly, the architecture (as described in subsection 4.1.1) will suffer a subtle change in the *gazebo* node, however, this time, for testing, this node will be directly the robotic hardware.

4.2.2.1 Simulation

In simulation, we achieved satisfying results with a successful ratio of 80% in a total of 50 episodes (according to Metric 7 as described in subsection 1.2.3) for the same environment used in training.

Let us now present the results for our testing in simulation. We will present the results according to Metrics 6 and 8, as described in subsection 1.2.3.

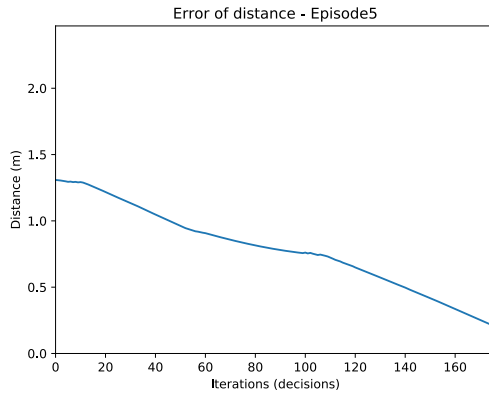
In the following Figures, we expose eight successful episodes. For each result we present the error of distance by decision taken and the developed trajectory.



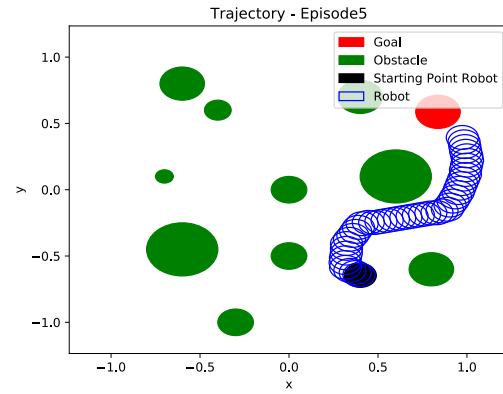
(a) Error of distance according to Metric 6.

(b) The developed trajectory according to Metric 8.

Figure 4.14: The testing results for Episode 3

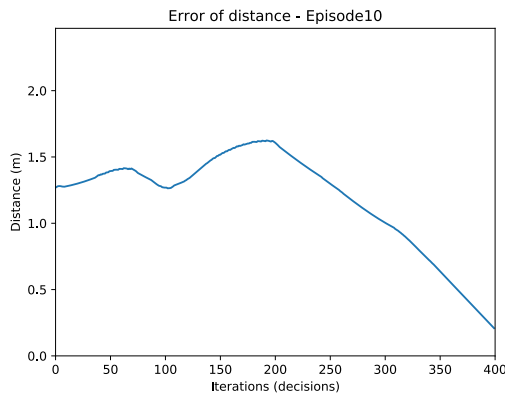


(a) Error of distance according to Metric 6.

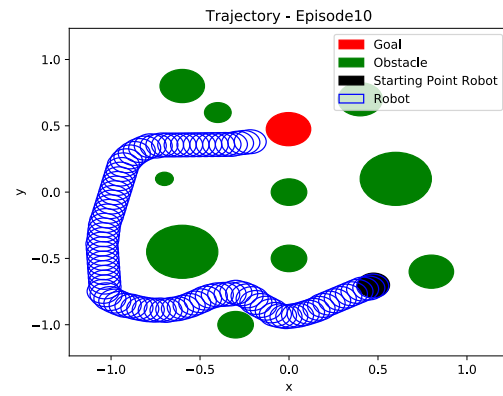


(b) The developed trajectory according to Metric 8.

Figure 4.15: The testing results for Episode 5

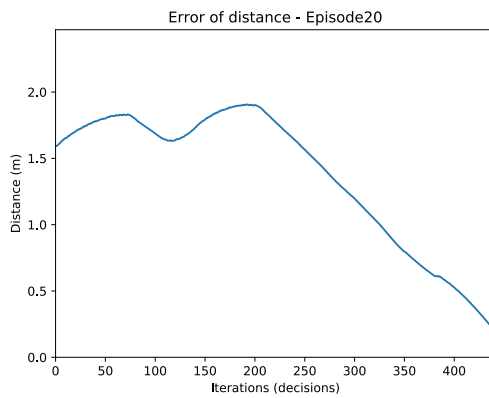


(a) Error of distance according to Metric 6.

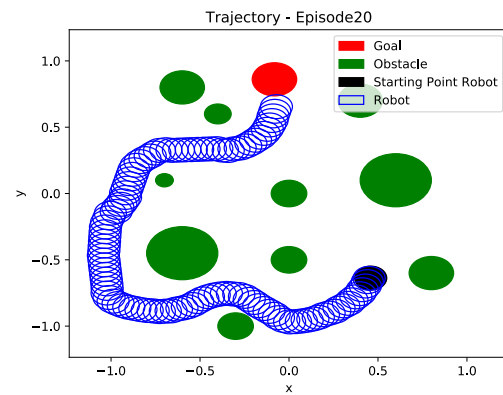


(b) The developed trajectory according to Metric 8.

Figure 4.16: The testing results for Episode 10



(a) Error of distance according to Metric 6.



(b) The developed trajectory according to Metric 8.

Figure 4.17: The testing results for Episode 20

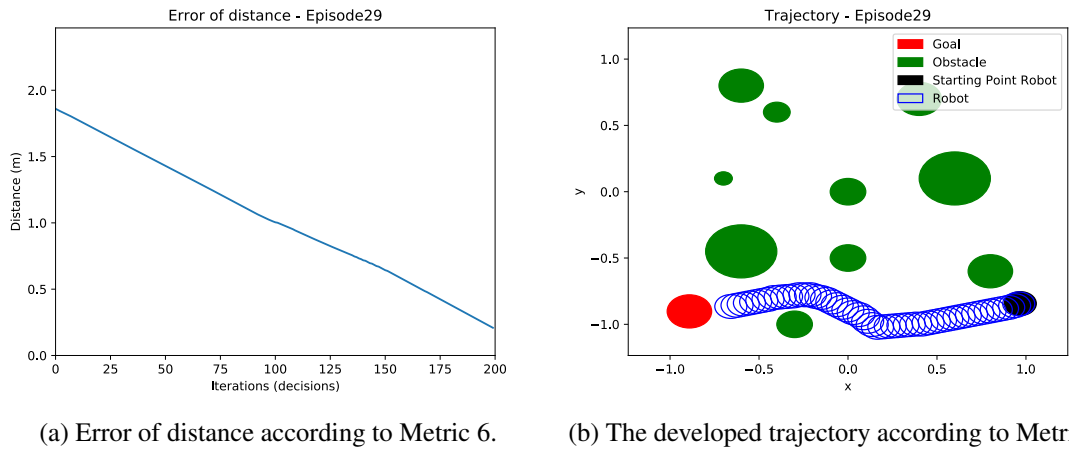


Figure 4.18: The testing results for Episode 29

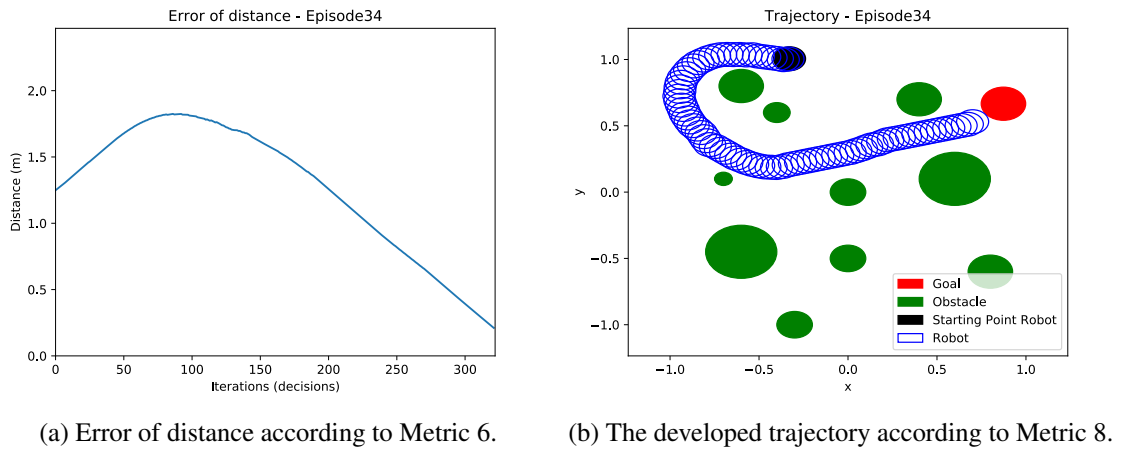


Figure 4.19: The testing results for Episode 34

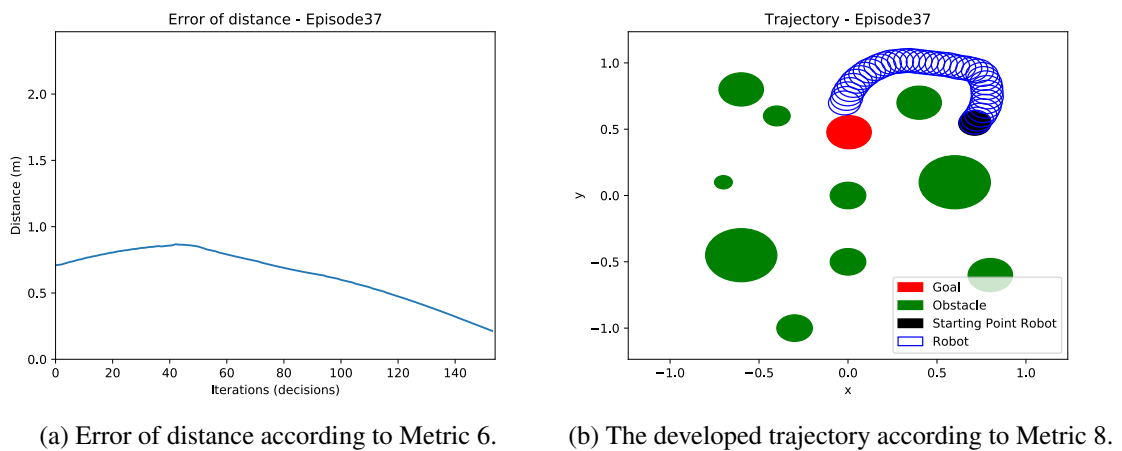


Figure 4.20: The testing results for Episode 37

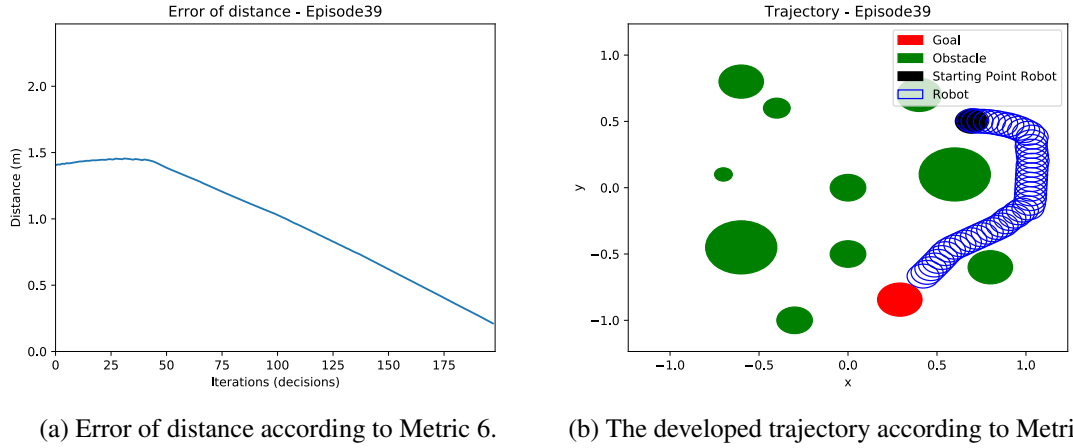


Figure 4.21: The testing results for Episode 39

4.2.2.2 Real-World

In the real-world we built two environments with two different configurations of obstacle positions, robot initial positions and goal positions.

Following, in the Figure 4.22, we expose the two real testing setups we built. For each environment the robot will start in two different positions. As exposed in the Figure 4.22, the robot's starting positions are in the top left corner and in the bottom right corner of the environment, as represented by both Figures 4.22a and 4.22b, respectively. For the robot's top left initial position there is a correspondent goal final position in the bottom left corner of the Figure 4.22a, and for the robot's bottom initial right position there is a correspondent goal final position in the top-left corner of the Figure 4.22b (we can visualize the goal positions by the red dots painted on the floor in both Figures).

To facilitate the further discussion let us define the robot's initial positions in the top-left and bottom-right corners as R_1 and R_2 , respectively. Additionally, let us define the goal positions in top-left and bottom-right corners as G_1 and G_2 , respectively. Lastly, let us define the first environment represented in the Figure 4.22a as O_1 and the environment from the Figure 4.22b as O_2 .

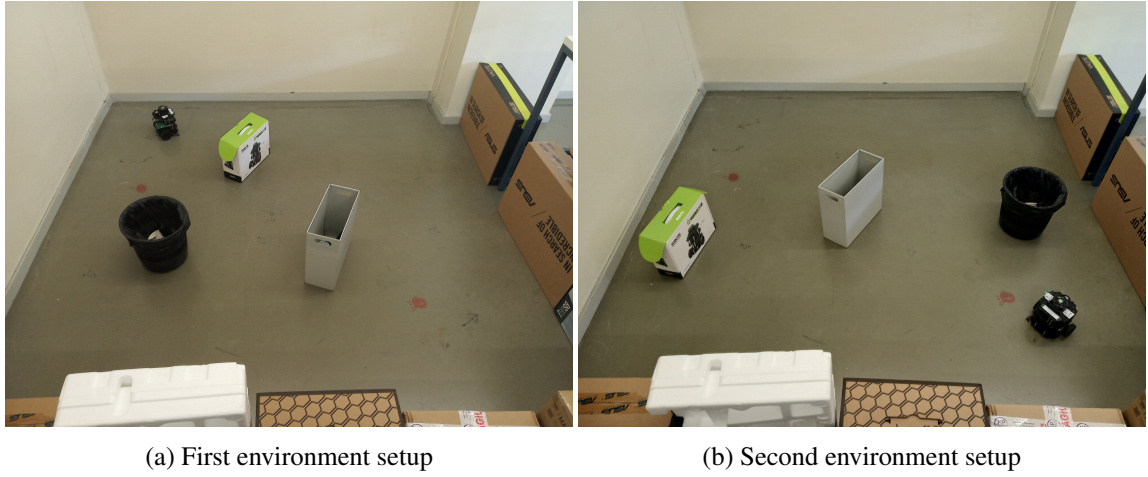
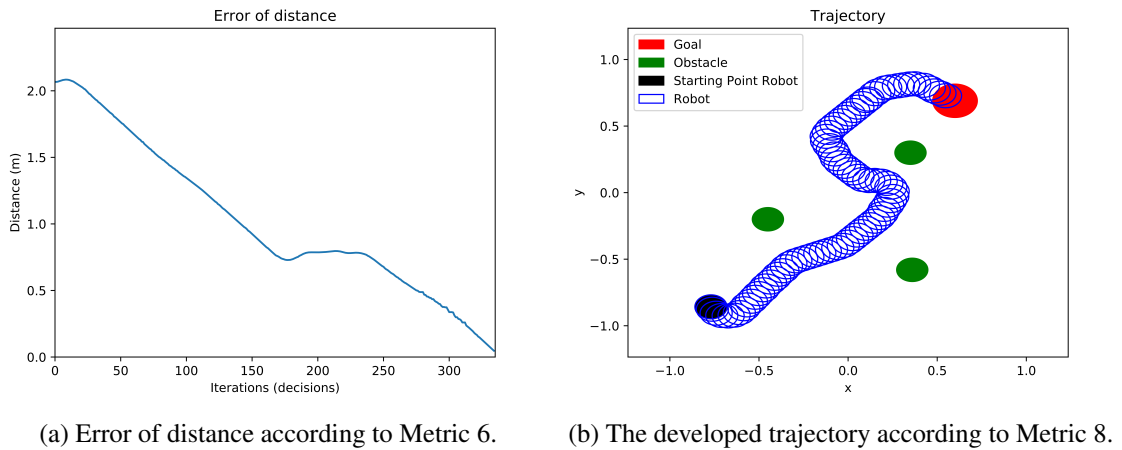


Figure 4.22: The two real testing environments.

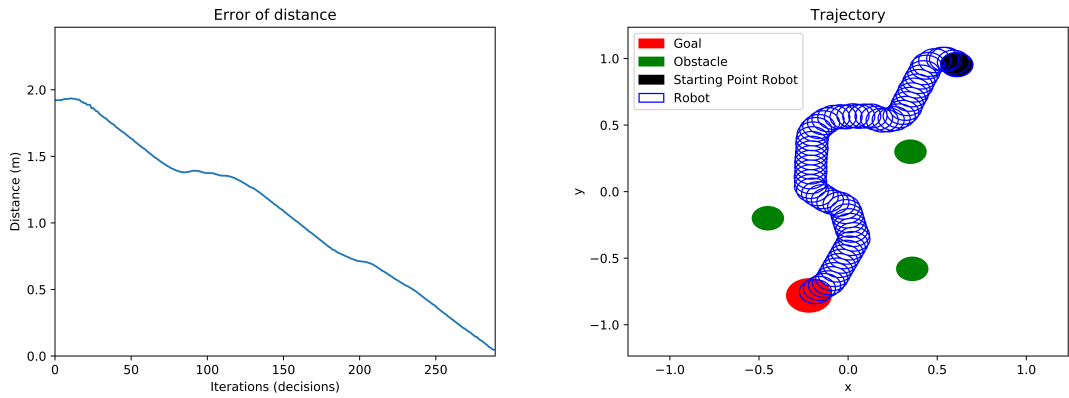
Let us now present the results of our real-world testing. For each environment, robot and goal configurations, we will present the results according to Metrics 6 and 8. We did not evaluate the real-world behavior with Metric 7 because it is expensive to develop huge amount of testing episodes in the real-world.

Firstly, we below represent the two developed trajectories, for the environment O_1 .

In the Figure 4.23, we expose the error of distance by decision taken and the developed trajectory for the environment O_1 , robot position R_1 and goal position G_1 .

Figure 4.23: The results for the environment O_1 , robot position R_1 and goal position G_1

In the Figure 4.24, we expose the error of distance by decision taken and the developed trajectory for the environment O_1 , robot position R_2 and goal position G_2 .

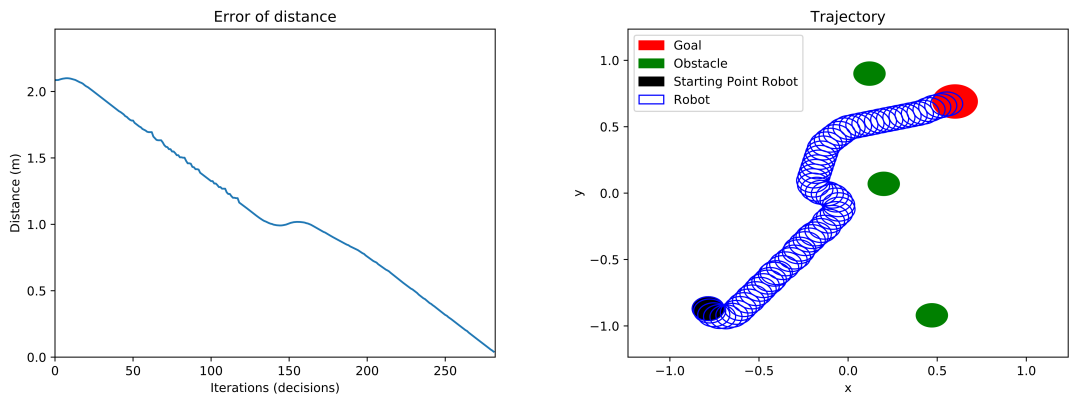


(a) Error of distance according to Metric 6. (b) The developed trajectory according to Metric 8.

Figure 4.24: The results for the environment O_1 , robot position R_2 and goal position G_2

Lastly, we below represent the two developed trajectories for the environment O_2 .

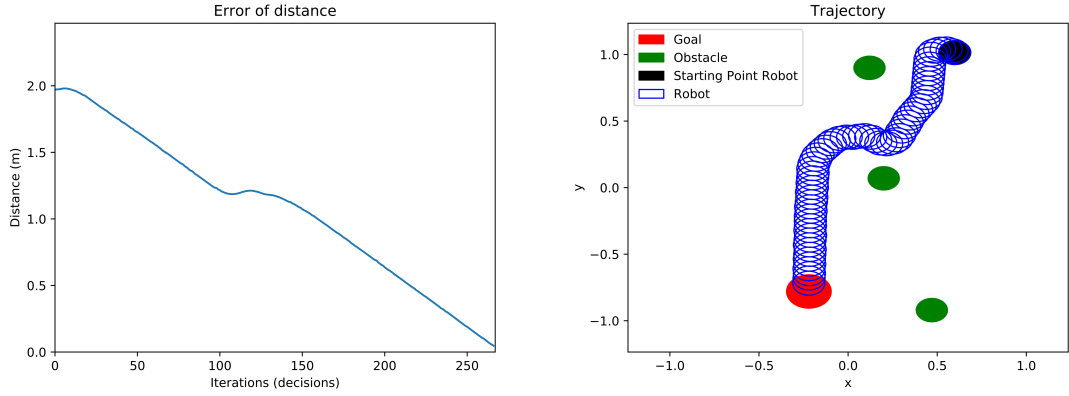
In the Figure 4.25, we expose the error of distance by decision taken and the developed trajectory for the environment O_2 , robot position R_1 and goal position G_1 .



(a) Error of distance according to Metric 6. (b) The developed trajectory according to Metric 8.

Figure 4.25: The results for the environment O_2 , robot position R_1 and goal position G_1

In the Figure 4.26, we expose the error of distance by decision taken and the developed trajectory for the environment O_2 , robot position R_2 and goal position G_2 .



(a) Error of distance according to Metric 6.

(b) The developed trajectory according to Metric 8.

Figure 4.26: The results for the environment O_2 , robot position R_2 and goal position G_2

To conclude, we were impressed by the robust transferability from the simulated behavior to the real robotic hardware. The behavior was not affected by natural physical events (e.g., measurement noise, inertia, slipping, drifting,...), hence the robustness of the trajectories. The developed trajectories were interestingly similar to the virtual ones. The behavior of the robot was also really close to the simulated behavior. Hence, the gap from the simulated environment to the real environment (the Sim2Real gap as described in section 2.2.3) was not even felt, and thus we can conclude that our Q-learning architecture successfully overcame that challenge.

Chapter 5

Conclusions

Solving robotics tasks with tabular reinforcement learning methods requires a concise state representation to avoid the curse of dimensionality that compromises or totally prevents convergence. Taking that into account, we have assertively built a Q-learning agent with an effective and successful state representation and reward design. The *los* feature gave sufficient awareness to perceive obstacles along the trajectory, and the reward function architecture successfully described the task, breaking the main problem into sub-problems of goal reaching and obstacle avoidance, conducting the agent to the desired behavior.

The executed trajectory was totally based on automated decisions in real-time, dependent on the obstacles it encountered along the way. Not only we achieved a successful simulated behavior, but also a successful transferability of the learned virtual behavior to the real robotic hardware. Conceptually, the proposed Q-learning architecture showed a clear and interesting capability and adaptability for the task at hand. Clearly, that was only possible by considerably reducing the dimensionality of the problem, which leads to fast and full convergences.

Future enhancements on our architecture include: (a) improving Q-learning to a Double Q-learning architecture so as to avoid the overestimation bias [25], the biggest issue of Value-Based RL, and thus speeding up the learning process, improving the estimation of the Q-values and consequently, more correct and fast convergence; (b) a more attentive choice of hyperparameters and a subtle increase in the state resolution, thus naturally improving performance, however with the cost of requiring a lot more exploration time, since the state representation quickly scales up in dimensionality; (c) additionally, to address both high-dimensional state-space (e.g., vision) and action-space complexities, we naturally suggest using Deep Q-Networks (DQN) or, ultimately, Double DQN to completely solve the overestimation issue of Q-learning.

References

- [1] Richard S. Sutton and Andrew G. Barto. *Planning and Learning with Tabular Methods*, page 159. A Bradford Book, 2018.
- [2] Richard S. Sutton and Andrew G. Barto. *Policies and Value Functions*, page 59. A Bradford Book, 2018.
- [3] Richard S. Sutton and Andrew G. Barto. *Optimal Policies and Optimal Value Functions*, page 63. A Bradford Book, 2018.
- [4] Richard S. Sutton and Andrew G. Barto. *Generalized Policy Iteration*, page 86. A Bradford Book, 2018.
- [5] Richard S. Sutton and Andrew G. Barto. *Temporal-Difference Learning*, page 119. A Bradford Book, 2018.
- [6] Richard S. Sutton and Andrew G. Barto. *Sarsa: On-policy TD Control*, page 131. A Bradford Book, 2018.
- [7] Richard S. Sutton and Andrew G. Barto. *Q-learning: Off-policy TD Control (Diagram)*, page 134. A Bradford Book, 2018.
- [8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.
- [9] Petar Kormushev, Sylvain Calinon, and Darwin Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2:122–148, 09 2013. doi: [10.3390/robotics2030122](https://doi.org/10.3390/robotics2030122).
- [10] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.
- [11] Richard S. Sutton and Andrew G. Barto. *Q-learning: Off-policy TD Control*, page 131. A Bradford Book, 2018.
- [12] Richard S. Sutton and Andrew G. Barto. *Goals and Rewards*, pages 53–54. A Bradford Book, 2018.
- [13] Bruno Siciliano and Oussama Khatib. *Definition of obstacle avoidance*. Springer-Verlag, Berlin, Heidelberg, 2007.
- [14] Richard S. Sutton and Andrew G. Barto. *Monte-Carlo Methods*, page 91. A Bradford Book, 2018.

- [15] Thomas M Moerland, Joost Broekens, and Catholijn M Jonker. Model-based reinforcement learning: A survey. *arXiv preprint arXiv:2006.16712*, 2020.
- [16] *Designing agent incentives to avoid reward tampering*. URL: <https://medium.com/@deepmindsafetyresearch/designing-agent-incentives-to-avoid-reward-tampering-4380c1bb6cd>.
- [17] Richard S. Sutton and Andrew G. Barto. *Dyna: Integrated Planning, Acting, and Learning*, page 162. A Bradford Book, 2018.
- [18] Richard S. Sutton and Andrew G. Barto. *Policy Gradient Methods*, page 321. A Bradford Book, 2018.
- [19] Richard S. Sutton and Andrew G. Barto. *Policy Gradient Theorem*, page 325. A Bradford Book, 2018.
- [20] Richard S. Sutton and Andrew G. Barto. *REINFORCE*, page 326. A Bradford Book, 2018.
- [21] Richard S. Sutton and Andrew G. Barto. *REINFORCE with Baseline*, page 326. A Bradford Book, 2018.
- [22] Richard S. Sutton and Andrew G. Barto. *Dynamic Programming*, page 73. A Bradford Book, 2018.
- [23] Richard S. Sutton and Andrew G. Barto. *Monte Carlo Estimation of Action Values*, page 73. A Bradford Book, 2018.
- [24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [25] Richard S. Sutton and Andrew G. Barto. *Maximization Bias and Double Learning*, page 134. A Bradford Book, 2018.
- [26] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.
- [27] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.
- [28] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.
- [29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.
- [30] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

- [31] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.
- [32] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL: <http://proceedings.mlr.press/v80/haarnoja18b.html>.
- [33] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.
- [34] Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker, and Sergey Levine. Learning to walk via deep reinforcement learning. *arXiv preprint arXiv:1812.11103*, 2018.
- [35] Richard S. Sutton and Andrew G. Barto. *Monte Carlo Tree Search*, page 185. A Bradford Book, 2018.
- [36] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.
- [37] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.
- [38] Richard S. Sutton and Andrew G. Barto. *Planning and Learning with Tabular Methods: Tabular Dyna-Q*, page 164. A Bradford Book, 2018.
- [39] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.
- [40] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.
- [41] Jens Kober, J. Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32:1238–1274, 09 2013. doi:10.1177/0278364913495721.
- [42] Richard S. Sutton and Andrew G. Barto. *Part I: Tabular Solution Methods*, page 23. A Bradford Book, 2018.
- [43] T.J. Rivlin. *An Introduction to the Approximation of Functions*. Blaisdell book in numerical analysis and computer science. Blaisdell Publishing Company, 1969. URL: <https://books.google.pt/books?id=dTnLGb590bIC>.

- [44] C.M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer New York, 2016. URL: <https://books.google.pt/books?id=kOXDtAEACAAJ>.
- [45] C.E. Rasmussen and C.K.I. Williams. *Gaussian Processes for Machine Learning*. Adaptive computation and machine learning series. University Press Group Limited, 2006. URL: <https://books.google.pt/books?id=vWtwQgAACAAJ>.
- [46] Błażej Osioński, Adam Jakubowski, Piotr Miłoś, Paweł Zięcina, Christopher Galias, and Henryk Michalewski. Simulation-based reinforcement learning for real-world autonomous driving. *arXiv preprint arXiv:1911.12905*, 2019.
- [47] Joonho Lee, Jemin Hwangbo, and Marco Hutter. Robust recovery controller for a quadrupedal robot using deep reinforcement learning. *arXiv preprint arXiv:1901.07517*, 2019.
- [48] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30. IEEE, 2017.
- [49] Stephen James, Paul Wohlhart, Mrinal Kalakrishnan, Dmitry Kalashnikov, Alex Irpan, Julian Ibarz, Sergey Levine, Raia Hadsell, and Konstantinos Bousmalis. Sim-to-real via sim-to-sim: Data-efficient robotic grasping via randomized-to-canonical adaptation networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 12627–12637, 2019.
- [50] Jacob Buckman, Danijar Hafner, George Tucker, Eugene Brevdo, and Honglak Lee. Sample-efficient reinforcement learning with stochastic ensemble value expansion. In *Advances in Neural Information Processing Systems*, pages 8224–8234, 2018.
- [51] Michael Janner, Justin Fu, Marvin Zhang, and Sergey Levine. When to trust your model: Model-based policy optimization. In *Advances in Neural Information Processing Systems*, pages 12519–12530, 2019.
- [52] Gal Dalal, Krishnamurthy Dvijotham, Matej Vecerík, Todd Hester, Cosmin Paduraru, and Yuval Tassa. Safe exploration in continuous action spaces. *ArXiv*, abs/1801.08757, 2018.
- [53] Yinlam Chow, Ofir Nachum, Edgar Duenez-Guzman, and Mohammad Ghavamzadeh. A lyapunov-based approach to safe reinforcement learning. In *Advances in neural information processing systems*, pages 8092–8101, 2018.
- [54] Richard Cheng, Gábor Orosz, Richard M Murray, and Joel W Burdick. End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3387–3395, 2019.
- [55] Aaron D Ames, Samuel Coogan, Magnus Egerstedt, Gennaro Notomista, Koushil Sreenath, and Paulo Tabuada. Control barrier functions: Theory and applications. In *2019 18th European Control Conference (ECC)*, pages 3420–3431. IEEE, 2019.
- [56] Henry Zhu, Justin Yu, Abhishek Gupta, Dhruv Shah, Kristian Hartikainen, Avi Singh, Vikash Kumar, and Sergey Levine. The ingredients of real-world robotic reinforcement learning. *arXiv preprint arXiv:2004.12570*, 2020.

- [57] Sehoon Ha, Peng Xu, Zhenyu Tan, Sergey Levine, and Jie Tan. Learning to walk in the real world with minimal human effort. *arXiv preprint arXiv:2002.08550*, 2020.
- [58] Justin Fu, Avi Singh, Dibya Ghosh, Larry Yang, and Sergey Levine. Variational inverse control with events a general framework for data-driven reward definition. In *NeurIPS*, 2018.
- [59] Avi Singh, Larry Yang, Kristian Hartikainen, Chelsea Finn, and Sergey Levine. End-to-end robotic reinforcement learning without reward engineering. 04 2019.
- [60] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *ICLR*, 2014. URL: <http://dblp.uni-trier.de/db/conf/iclr/iclr2014.html#KingmaW13>.
- [61] Daniel Dewey. Reinforcement learning and the reward engineering principle. In *2014 AAAI Spring Symposium Series*, 2014.
- [62] *The reward engineering problem*. URL: <https://ai-alignment.com/the-reward-engineering-problem-30285c779450>.
- [63] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.
- [64] Richard S. Sutton and Andrew G. Barto. *The Agent-Environment Interface*, page 47. A Bradford Book, 2018.
- [65] Richard S. Sutton and Andrew G. Barto. *Rewards and Episodes*, page 54. A Bradford Book, 2018.
- [66] Richard S. Sutton and Andrew G. Barto. *The Agent-Environment Interface - Markov Property*, page 49. A Bradford Book, 2018.
- [67] Richard S. Sutton and Andrew G. Barto. *Dynamic Programming*, page 73. A Bradford Book, 2018.
- [68] Richard S. Sutton and Andrew G. Barto. *Policy Evaluation (Prediction)*, page 74. A Bradford Book, 2018.
- [69] Richard S. Sutton and Andrew G. Barto. *Policy Improvement*, page 76. A Bradford Book, 2018.
- [70] Richard S. Sutton and Andrew G. Barto. *Policy Improvement (proof)*, page 76. A Bradford Book, 2018.
- [71] Dmitry B. Rokhlin. Robbins–monro conditions for persistent exploration learning strategies. *arXiv preprint arXiv:1808.00245*, 2018.
- [72] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. IEEE, 2017.
- [73] Richard S. Sutton and Andrew G. Barto. *Advantages of TD Prediction Methods*, page 124. A Bradford Book, 2018.
- [74] Richard S. Sutton and Andrew G. Barto. *The Deadly Triad*, page 264. A Bradford Book, 2018.

- [75] Richard S. Sutton and Andrew G. Barto. *Off-policy Methods with Approximation: Summary*, page 285. A Bradford Book, 2018.
- [76] Hado Van Hasselt, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. Deep reinforcement learning and the deadly triad. *arXiv preprint arXiv:1812.02648*, 2018.
- [77] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. A theoretical analysis of deep q-learning. In *Learning for Dynamics and Control*, pages 486–489. PMLR, 2020.
- [78] Aditya Gudimella, Ross Story, Matineh Shaker, Ruofan Kong, Matthew Brown, Victor Shnayder, and Marcos Campos. Deep reinforcement learning for dexterous manipulation with concept networks. *arXiv preprint arXiv:1709.06977*, 2017.
- [79] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. 06 2018. doi:10.15607/RSS.2018.XIV.049.
- [80] Paul Christiano, Jan Leike, Tom B Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *arXiv preprint arXiv:1706.03741*, 2017.
- [81] OpenAI. Learning from Human Preferences. <https://openai.com/blog/deep-reinforcement-learning-from-human-preferences/>, 2017. [Online; accessed 10-March-2021].
- [82] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017.