POLYTECHNIC OF TURIN

Master's Degree in Mathematical engineering

Master's Degree Thesis

Review and application of Offline Reinforcement Learning methods for mobile robots.

Supervisors

Prof. Marcello CHIABERGE

PhD. Mauro MARTINI

PhD. Andrea EIRALE

Candidate

Filippo BUFFA

November 2023

Summary

In the age of advanced technology, the deployment of mobile robots has emerged as a vital solution, transforming various industries by automating tasks, improving efficiency, and shaping the future of autonomous systems. One significant advancement in the past decade is Reinforcement Learning, made possible by the evolution of deep learning, allowing for task generalization and impressive results. However, modern Reinforcement Learning algorithms still rely on a trial-and-error approach, leading to high costs in terms of time and money. This thesis explores an alternative strategy by employing Offline Reinforcement Learning algorithms in these tasks. It demonstrates instances where this new approach yields positive outcomes and enables purely offline training, saving both time and costs. On the other hand, this thesis will explore the challenges that this approach present, particularly in the areas of generalizing tasks and handling diverse ones. To validate these methods, simulation experiments were conducted, highlighting their pros and cons. Indeed, while some tasks were successfully solved by all algorithms tested, others proved to be unsolvable for each of them. Looking ahead, as new algorithms and tools continue to be invented and discovered, this approach has the potential to revolutionize how we address the mobile robot problem. It offers a more viable solution for companies entering this realm, be it as users or producers.

Table of Contents

List of Tables

List of Figures

Acronyms

AI	Artificial Intelligence
RL	Reinforcement Learning
ORL	Offline Reinforcement Learning
MDP	Markov Decision Process
TD	Temporal Difference
SAC	Soft Actor Critic
IQL	Implicit Q-Learning
BCQ	Batch-Constrained Q-learning
CQL	Conservatice Q-Learning
EDAC	Ensamble-Diversified Actor-Critic

Chapter 1

Introduction

1.1 Motivation

Imagine having to play card a game without knowing the rules and without have ever seen a match. How many times would it take to you to understand the underlying structure of the game when only the only feedback from the game are if you have made a good choice in the short term (without taking into account the whole duration of the game) and if an action is possible or not? How many times would you loose? And what if you have to pay to play each game, how much would you spend before getting good at it?
Reinforcement learning algorithms have shown remarkable achievements across diverse domains, including robotics and game-solving. However such an approach would be very similar to the one described.
Would not you prefer having some information before start playing?
Offline reinforcement learning address this problem in the simplest way possible, by excluding every interaction with the system (you can not lose money in this way). The aim of this approach is to learn everything a priori. Having recorded data of the setting from the past this kind of algorithms try to understand the best way to deal with each problem or task. In the other hand as will be shown in this thesis such approach introduces new challenges which, in some cases, can lead to sub-optimal solutions or even or even not being able to solve the task at all.

1.2 Problem statement

The focus of this thesis will be the application of Offline Reinforcement Learning on a specific task, the mobile robot task. In this setting the goal of the learning is to understand which action have to be taken in order to make the robot able to reach a specific position. Such task have already been widely study in engineering

and state-of-the-art algorithms show very good result, however this approach relies on engineered algorithms which in general are less able to deal with different environment compare to machine learning methods. In the other hand, even if machine learning algorithms have shown a reliable alternative in some cases, most of the best algorithms are still based on old methods. The aim of this thesis is to state the actual state Reinforcement Learning methods and Offline Reinforcement Learning methods, with more focus on the latter.
The mobile robots show a wide variety of problems, the one address here will be the one about the motion, without entering in other fields such as energy consumption or human robots interaction. Here are some of the main problems this thesis will try to face

- **Navigation and Path Planning**: designing algorithms that enable robots to navigate through complex, dynamic environments, avoiding obstacles and finding optimal paths to their destinations.
- **Obstacle Avoidance**: developing algorithms that allow robots to detect and avoid obstacles in real-time, ensuring safe navigation, especially in crowded or dynamic spaces.
- **Adaptability and Learning**: creating robots that can adapt to changing environments and learn from their experiences.
- **Safety**: ensuring that mobile robots operate safely, especially in environments shared with humans.
- **Cost**: designing learning methods that reduce the overall cost of the learning, in terms of money and time consumption.

1.3 Thesis outline

Chapter 2 provides brief introduction to the state-of-the-art, however more in depth details of the algorithms will be shown in Chapter 5. In Chapter 3 a theoretical background of Reinforcement Learning is stated and in Chapter 4 the same is done for Offline Reinforcement Learning. Chapter 5 explains in details all the Offline Reinforcement Learning used in this study, meanwhile in Chapter 6 one can find all the information about the environment where such algorithms have been applied. In conclusion in Chapter 7 the results of all the experiments are given and discussed.

Chapter 2

State-of-the-Art

In this chapter there will be an introduction to the state-of-the-art of three components: robots mobile problem, Reinforcement Learning for mobile robots and Offline Reinforcement learning applied to mobile robots.

2.1 Mobile robot algorithm

The mobile robot problem is as old as first robots were invented, being able to make it self driving over tasks has been one of the main goal in robot research. Indeed such problem has been highly engineered ì in the past decades, however thanks to the development of neural networks and more complex artificial intelligences methods that both take into account old methods and new tools have taken places. In the other hand it is very important to understand and know such types of algorithms which untill now have shown the best results over tasks that are not possible to handle in other ways. Here there is a list of the best approach to mobile robots.

- **DWA**: Fox et al. 1997 has introduced this new approach which is still one of the best today. It is primarily employed for real-time obstacle avoidance and path planning for mobile robots operating in dynamic environments. DWA focuses on generating safe and feasible robot trajectories by considering the robot's dynamics and the surrounding environment.
- **SLAM**: this family of algorithms enable a mobile robot to create a map of its environment while simultaneously localizing itself within that map. Popular SLAM algorithms include FastSLAM, Graph SLAM, and visual SLAM methods like ORB-SLAM.
- **MPC**: is a control strategy that predicts the system's behavior over a short time horizon and optimizes control inputs to achieve specific objectives. MPC

can be used for trajectory tracking, obstacle avoidance, and other tasks requiring predictive control.

- **Path Planning Algorithms**: help robots find the optimal path from their current location to a target location while avoiding obstacles. Common path planning algorithms include A* (A star), Dijkstra's algorithm, and Rapidly-exploring Random Trees (RRT).

2.2 Reinforcement Learning for mobile robots

Reinforcement learning algorithms are rapidly evolving over the past years, due to the increasing interest in this field. It is of high interest noting how such an approach does not depends on the field of application thanks to its generality. Indeed, unless one take the specific problem into account, the best RL algorithms for mobile robots application are the same as the one for the general one. It is also worth pointing out how the performance of such approaches depends on the application, which make it very difficult to decide which is the best algorithm overall. Some of them shine for their simplicity and great performance. Here a list of the bests is given, some of them will be further discus in chapter 3.

- **SAC**: one the most used algorithm due to its high applicability over various field. It is designed to handle continuous action spaces. It incorporates entropy regularization, which encourages exploration in the learning process. It was first introduced by Haarnoja et al. 2018.

- **TD3**: it is an improvement over the original DDPG introduced by Fujimoto, Hoof, et al. 2018. It is an actor-critic algorithm that is well-suited for continuous action spaces. It has been applied to various robotic tasks, including mobile robot control.

- **A3C** It is an asynchronous variant of the actor-critic algorithm proposed by Mnih, Badia, et al. 2016, designed to handle parallelization more efficiently. It has been used in various robotic applications, including mobile robot navigation.

- **MPC**: it is a control strategy that predicts the system's behavior over a finite time horizon and optimizes the control inputs. In the past few years researchers have tried to combined it with reinforcement learning, it have been shown that it can lead to effective control policies for mobile robots.

2.3 Offline Reinforcement Learning for mobile robots

Being it difficult for RL algorithms to address the mobile robots problem in an online setting, as will be shown in this thesis, it is very hard to draw up a list of the best offline methods due to the lack of literature. Indeed such standings is one of the aim of this thesis, here a list of the best Offline algorithms based on benchmarks such as D4RL introduced in Fu et al. 2021 which has been widely used to test ORL algorithms. In this work the focus will be on model-free algorithms due to their higher applicability and generality. Here there is a list of the best algorithms, most of them will be discussed in further details in chapter 5.

- **BCQ**: introduced in Fujimoto, Meger, et al. 2019, it was one of the first attempt to deal with Offline reinforcement learning with a specific algorithm. However how due to its complexity and low capability to handle complex and various environment it fails to deal with robots mobile applications.
- **CQL**: based on the idea of penalize the Q-values over their "distance" from the dataset distribution it has shown good results over the past years handling different problems, it was first proposed in Kumar et al. 2020.
- **IQL**:maybe the simplest yet powerful ORL algorithm, the idea of deal with the overestimation of the Q-values as a "risk" problem using the expectile regression has shown great results in various fields. Since it introduction in Kostrikov et al. 2021 it is the state of the art for ORL in general.
- **EDAC**: based on SAC it has shown how ORL problems can be seen as RL problems with some adjustments. Proposed by An et al. 2021 has shown how handle the request for a very high number of neural network.
- **MOPO**: the only model-based method of the one enlisted here, introduced in Yu et al. 2020 is the most used and famous model-based Offline Reinforcement learning algorithm.

Chapter 3

Reinforcement Learning

Reinforcement Learning (RL) is a subfield of Machine Learning, where the main goal is to find a policy to follow for sequential decision-making problem. In this chapter an introduction to this field will be made based on the book Sutton and Barto 2018 which has been the baseline for such problem in the past decades.

3.1 Elements of RL

The main components for an RL problem are the agent, the environment and the reward.

Agent The agent is a learning actor which interacts with the environment to reach a specified goal. The interaction is given by an action chosen by the agent based on the information from the environment called states. The codomain of the policy is called the action-space and can be either continous or discrete depending on the problem. The actions are the only way the agent can interact with the environment.

Environment The environment is where the agent can act. The way the agent knows about the environment is through to the observations. Those together with the rewards are given to the agent every time it takes an action. The observations can be seen as an encoding of the environment or part of it into a space called state-space.

Reward The reward is a scalar value passed to the agent every time it takes an action. It is worth nothing that this value is a function of the action taken and the specific state in which it is taken. It can be either stochastic or deterministic, but in this study it will be mostly be treated as a deterministic value.

Other important concepts have to be introduced to help better understand the following work.

States In the literature often the concept of state and observation are mixed. As we will see in the next section the mathematical tool used to describe RL problems is the MDP, which needs that every state has the Markovian property:

$$\mathbb{P}[s_{t+1}|s_1, s_2, ..., s_t] = \mathbb{P}[s_{t+1}|s_t].$$

In most of the real settings the observation given by the environment does not have this property. Thus to be able to use this mathematical tool it is necessary to distinguish the concept of observation and state. Indeed, we define the state as a function of the *history* of the episode, (i.e. function of all observations, actions and rewards) which have all the necessary information to makes the Markovian property true.

Trajectories The set of temporal sequences of *state* and *actions* pairs of a single episode it is called trajectory. As an example an episode where n different actions have been taken can be seen as follows

$$\{(s_1, a_1), (s_2, a_2), ..., (s_n, a_n)\}.$$

An episode can finish for various reasons, for examples: the task is completed, the number of action taken is over a certain chosen limit.

Return The aim of RL is to maximise the sum of the rewards over a trajectory, this value is called *return*,

$$G_t = \sum_{i=0}^{K} r_{t+i+1},$$

where K can be either finite or infinite depending on the system settings. In most cases it is useful to refer to the *discounted return*, instead. This new value is:

$$G_t = \sum_{i=0}^{K} \gamma^i r_{t+i+1} \qquad \gamma \in [0,1).$$

where γ is the discount factor.

Policy The policy is a map from *states* to *actions*. It can be either stochastic or deterministic. In the former case it gives a probability distribution over the actions given the state. From a *policy* point of view, the aim of RL is to get the optimal policy (i.e. , the one that gives the highest return or cumulative reward). It is true that such a policy does not always exist or is findable.

Value function This function is based on a policy and gives the expected return of trajectory from a state s following the policy π,

$$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}\left[G_t | s_t = s, a_t = a\right].$$

It is worth noting that the expected value is not only on the policy, but over all the stochastic components (i.e. policy, reward, evolution of the states). Being able to compute such function or at least approximate them is very important for a lot of RL algorithms, since it gives an idea of which policy is better.

Q-value Most of the algorithms nowadays are not based on *value functions*, but rather on the *Q-value*. This function is again based on a policy, but it maps both *state* and *actions* into the future expected *return* following the policy.

$$Q^{\pi}(s,a) = r_t(s,a) + \mathbb{E}_{a\sim\pi}\left[G_{t+1} | s_{t+1} = s'\right]$$

where s' is the new observation after the agent took the action a in the state s.

3.2 Markov decision process

It has been seen how Markov decision processes (MDP) are the natural mathematical tool to describe RL problems. A MDP is a discrete time decision control process that gives the possibility to model discrete decision making where the reward is partially random and partially subject to the decision taken in every state by the decision-maker. They are define by the tuple $(\mathbf{S}, \mathbf{A}, \mathbb{P}, d_0, R)$:

- $\mathbf{S}$ is the set of possible states (i.e. the *state-space*).
- $\mathbf{A}$ is the set of possible actions (i.e. the *action-space*)
- $\mathbb{P}[s_{t+1} = s' | s_t = s, a_t = a]$ which is the probability to be in the new state s' at time $t+1$ being in the state s at time t and taking the action a.
- d_0 is the initial distribution over the states
- $R(s, s', a)$ is the immediate reward or the expected value of the immediate reward given by passing from state s to state s' taking the action a.

It is worth noting how, in this type of description, in the states embed all the information about the environment and the history of the process. Indeed, the assumption is that the probability of a specific next state depends only on the current state and the action taken. It is also important to stress that not all the actions can be taken in every state (i.e. $\mathbf{A}(s) \subseteq \mathbf{A}$ where $\mathbf{A}(s)$ is the set of possible actions in the state s). To be more coherent with the notation used before we can also add to this description the discount factor $\gamma \in [0,1)$. In most of this work the MDP formalism will be used, but to be complete an extension have to be introduced.

Partially observable MDP The partially observable Markov decision process is defined as a tuple $(\mathbf{S}, \mathbf{A}, \mathbf{O}, \mathbb{P}, d_0, R, \mathbb{E})$, where $\mathbf{S}$, $\mathbf{A}$, $\mathbb{P}$, d_0, R are defined as before. $\mathbf{O}$ is a set of observations, where each observation is given by $o \in \mathbf{O}$, and $\mathbb{E}$ is an emission function, which defines the distribution $\mathbb{E}(o_t|s_t)$. This kind of MDP are used to describe partially observable environments.

3.3 Bellman equation

As discussed in 3.1 and 3.1 *value functions* and *q-values* are very important to exploit a optimal policy. In this section those two will be studied to reach a more explicit and useful formulation for the RL problem. Rewriting the value function we can exploit a recurrence into it.

$$\begin{aligned} V^\pi(s) =& \mathbb{E}_{a_i \sim \pi,\, s_i \sim \mathbb{P},\, r_i \sim R}[G_t | s_t = s] \\ =& \mathbb{E}_{a_i \sim \pi,\, s_i \sim \mathbb{P},\, r_i \sim R}[r_t(s_t, a_t) + \gamma r_{t+1}(s_{t+1}, a_{t+1}) + ... + \gamma^K r_{t+K}(s_{t+K}, a_{t+K})] \end{aligned}$$

It is to see that collecting γ where possible and rewriting the expected value as a sum, the following is obtained

$$\begin{aligned} &\sum_{a \in \mathbf{A}} \pi(a|s) \sum_{s' \in \mathbf{S}} P(s'|s,a)(R(s,s',a) + \gamma \mathbb{E}_{r_i \sim R}(r_{t+1}(s_{t+1}, a_{t+1}) \\ &+ ... + \gamma^{K-1} r_{t+K}(s_{t+K}, a_{t+K})) \\ =& \sum \pi(a|s) \sum P(s'|s,a)\,(R(s,s',a) + \gamma(V^\pi(s')) \\ =& \mathbb{E}_{a_i \sim \pi,\, s_i \sim \mathbb{P},\, r_i \sim R}\,[R(s,s',a) + \gamma(V^\pi(s')] \end{aligned}$$

Using the same argument, it is to obtain the similar result for the *Q-values.*

$$\begin{aligned} Q^\pi(s,a) =& \mathbb{E}_{s' \sim \mathbb{P}}[R(s,s',a)] + \mathbb{E}_{a_i \sim \pi,\, s_i \sim \mathbb{P},\, r_i \sim R}[G_{t+1} | s_{t+1} = s'] \\ =& \mathbb{E}_{s' \sim \mathbb{P}}[R(s,s',a)] + \sum_{a \in \mathbf{A}} \pi(a'|s') \sum_{s' \in \mathbf{S}} P(s''|s',a')(R(s',s'',a') + \\ & \gamma \mathbb{E}_{r_i \sim R}(r_{t+1}(s_{t+1}, a_{t+1}) + ... + \gamma^{K-1} r_{t+K}(s_{t+K}, a_{t+K})) \\ =& \mathbb{E}_{s' \sim \mathbb{P}}[R(s,s',a)] + \mathbb{E}_{a_i \sim \pi,\, s_i \sim \mathbb{P},\, r_i \sim R}[\gamma Q^\pi(s',a')] \end{aligned}$$

Having this new formulations it is possible to apply the principle of optimality: *"An optimal policy has the property that whatever the initial state and initial decision* *are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision" Bellman, 1957, Chap. III.3.* Following this principle it is possible to break the problem of finding the optimal policy into smaller sub-problems, where only one step is consider. Indeed, at each step a maximization problem has to be solved. The formulations of this steps become

$$V^{\pi^*}(s) = \max_{a \in \mathbf{A}} \mathbb{E}_{s_i \sim \mathbb{P}} \left[R(s, s', a) + \gamma V^{\pi^*}(s') \right]$$
$$Q^{\pi^*}(s, a) = \mathbb{E}_{s' \sim \mathbb{P}}[R(s, s', a)] + \gamma \max_{a' \in \mathbf{A}} \mathbb{E}_{s_i \sim \mathbb{P}, R_i \sim R}[Q^{\pi^*}(s', a')].$$

Many methods have been found to solve this kind of problem, in the next sections some of them will be discussed. As it has been discussed finding the optimal policy for an MDP is equivalent to finding the optimal policy for an RL setting on the same MDP, thus since the Bellman equation are a formulation for the solution of the MDP problem those are also solutions for the RL one. More complex algorithm of RL and Offline RL are based on solution to the Bellman equation. It is possible to rewrite the value function in matrix form obtaining the following equation

$$V^\pi = R + \gamma \mathbf{P} V^\pi, \tag{3.1}$$

where $\mathbf{P}$ is the transition probability matrix. It is trivial to find a solution for such system

$$V^\pi = R + \gamma \mathbf{P} V^\pi$$
$$(\mathbf{I} - \gamma \mathbf{P}) V^\pi = R$$
$$V^\pi = (\mathbf{I} - \gamma \mathbf{P})^{-1} R.$$

Such solution it is also the solution of the Bellman equation. Now the Bellman expectation backup operator will be introduced. This will be used to better understand theorem in the following section.

Definition 1. *The Bellman expectation backup operator is define as follow*

$$F(V) = R^\pi + \gamma \mathbf{P}^\pi V. \tag{3.2}$$

Definition 2. *A contraction in a metric-space (M, d) is a function $f : M \to M$ and $\exists\ \gamma \in$ [0,1) such that*

$$d(f(x), f(y)) \leq \gamma d(x, y). \tag{3.3}$$

Theorem 1. *The Bellman expectation backup operator is a contraction with the distance induced by the ℓ^∞ norm.*

Proof. By its definition the Bellman expectation backup operator is a function from the matrix space of the V dimension to itself.

$$\begin{aligned} ||F(V) - F(U)||_\infty =& ||R^\pi + \gamma \mathbf{P}^\pi V - R^\pi - \gamma \mathbf{P}^\pi U||_\infty && (3.4) \\ =& ||\gamma \mathbf{P}^\pi (V - U)||_\infty && (3.5) \\ \leq& \gamma ||\mathbf{P}^\pi \mathbb{1}||_\infty ||(V - U)||_\infty && (3.6) \\ =& \gamma ||V - U||_\infty . && (3.7) \end{aligned}$$

□

3.4 Problems and approaches to RL

3.4.1 Differences from other type of learning

Reinforcement learning The key differences between reinforcement learning and other kind of learning such as supervised or unsupervised are easy to understand. First of all in RL, the feedback is in the form of rewards or penalties, which are often delayed and sparse. In contrast in other settings such as supervised learning, the feedback is direct and immediate, in the form of labeled data. Due to its setting it is important to incorporate a trade-off between exploring new actions to discover their rewards and exploiting known actions to maximize immediate rewards. This exploration-exploitation dilemma is unique to RL. RL deals with sequential decision-making problems where the agent's actions affect future states and subsequent rewards. Supervised learning and most unsupervised learning tasks do not involve sequential decision making. RL agents interact with an environment, which might be real (like a robot) or simulated. This interaction is a fundamental aspect of RL and is not present in most supervised or unsupervised learning scenarios.

3.4.2 Exploration vs Exploitation tradeoff

In reinforcement learning, the exploration-exploitation trade-off refers to the balance between exploring new actions and states in the environment, denoted by a_t, and exploiting the current knowledge about the environment to maximize reward, denoted by r_t. This trade-off can be formalized as a decision problem, where the agent must choose between exploration and exploitation at each timestep t.

One way to model this decision problem is using a multi-armed bandit framework, where the agent must choose between different actions, each with a different expected reward. The expected reward for each action can be represented by a mean value μ_i and a variance σ_i^2. The agent can then use a strategy, such as epsilon-greedy exploration or Thompson sampling, to balance exploration and exploitation.

For example, the epsilon-greedy exploration strategy involves choosing the action with the highest expected reward with probability $1-\epsilon$, and choosing a random action with probability ϵ. This can be formalized as:

$$a_t = \begin{cases} \arg\max_{a_i} \mu_i & \text{with probability } 1-\epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

Alternatively, the Thompson sampling strategy involves sampling from the posterior distribution of the expected reward for each action, and choosing the action with the highest sample. This can be formalized as:

$$a_t = \arg\max_{a_i} \mathcal{N}(\mu_i, \sigma_i^2)$$

where $\mathcal{N}(\mu_i, \sigma_i^2)$ represents a normal distribution with mean μ_i and variance σ_i^2. Finding the right balance between exploration and exploitation is a key challenge in reinforcement learning. If the agent explores too little, it may not learn about the full potential of the environment and may become stuck in a suboptimal policy. If it explores too much, it may waste time and resources on actions that do not lead to high reward. There are various strategies that can be used to address this trade-off, including epsilon-greedy exploration, Thompson sampling, and UCB algorithms.

3.5 Dynamic programming

Dynamic programming is a well know set of algorithms used to learn optimal behaviour over finite and known MDP. This kind of methods are indeed at the basis of most of the new algorithm to solve RL problems. Due to its computational cost and the fact that in RL problems the MDP is not known it is not a good way to address this kind of problem by itself, but it the right manipulation of the algorithms and using approximation functions it one of the main approach to RL. The key idea of DP is to use Bellman equations to update the policy, the way to use it depends on the algorithms. In this section some of the main ideas of DP and some algorithms will be discussed and explained.

3.5.1 Policy evaluation

Let π be a policy over a known MDP, the goal is to exploit the value function of such policy. To do so it is enough to use a recurrent algorithm based on the Bellman equation, which at every step do the following process

$$V_{k+1}^{\pi}(s) = \mathbb{E}\left[r(s,a) + \gamma V_k^{\pi}(s')\right]. \tag{3.8}$$

Using 1 we know that the operator associated to such equation is a contraction, thus if exists the value function V^π is the fixed point of such iteration.

3.5.2 Policy improvement

Once one is able to evaluate policy over an MDP it is necessary to be able to improve it, thus finding a new policy π' such that $V(s)^\pi < V(s)^{\pi'}$. To this end policy improvement theorem comes in help.

Theorem 2. *Let be π and π' two policies over the same MDP, such that*

$$Q_\pi(s, \pi'(s)) \geq V(s)^\pi \qquad \forall s \in \mathbf{S}. \tag{3.9}$$

It follows directly that π' is as good as π or "better". Where "better" means that its value function has a greater value.

Proof. Consider a succession over the policy, where

$$\begin{aligned} \pi_0 = \pi \quad \forall s \in \mathbf{S} = \mathbf{S}_0 \\ \pi_1 = \pi_0 \quad \forall s \in \mathbf{S}_0 \backslash \{s_0\} = \mathbf{S}_1 \quad \pi_1(s_0) = \pi'(s_0), \quad s_0 \in \mathbf{S}_0 \\ \vdots \\ \pi_{k+1} = \pi_k \quad \forall s \in \mathbf{S}_k \backslash \{s_k\} = \mathbf{S}_{k+1} \quad \pi_{k+1}(s_k) = \pi'(s_k), \quad s_k \in \mathbf{S}_k, \end{aligned}$$

this succession converges to π'. By hypothesis we know that

$$Q_\pi(s, \pi(s)) = Q_{\pi_0}(s, \pi_0(s)) = \mathbb{E}\left[R(s, \pi_0(s)) + \gamma V(s')^{\pi_0}\right] \tag{3.10}$$

and it also that

$$Q_{\pi_k}(s_{k+1}, \pi_{k+1}(s_{k+1})) \geq V(s_k)^{\pi_k} \tag{3.11}$$

by definition of the policies. Without loosing generality it can be assumed that the succession over the policy is made such that s_{k+1} is the state visited after s_k following the policy π'. Now it is possible to prove that

$$\begin{aligned} V(s_0)^\pi \leq & Q_{\pi_0}(s_0, \pi_1(s)) = \mathbb{E}\left[R(s_0, \pi_1(s_0)) + \gamma V(s_1)^{\pi_0}\right] \\ \leq & \mathbb{E}\left[R(s_0, \pi_1(s_0)) + \gamma\left(R(s_1, \pi_2(s_1) + V(s_1)^{\pi_0}\right)\right] \\ & \vdots \\ \leq & \mathbb{E}\left[\sum_{i=0}^{K} \gamma^i R(s_i, \pi_{i+1}(s_i)\right] = \mathbb{E}\left[\sum_{i=0}^{K} \gamma^i R(s_i, \pi'(s_i)\right] = V(s_0)^{\pi'}. \end{aligned}$$

This is true for all s_0, thus the theorem is proved, having equal or greater value function for all the states. □

The building of a policy which satisfies the theorem hypothesis is very simple, let be π a policy that has been evaluated by policy evaluation. One can create a new policy as

$$\pi'(s) = \underset{a}{argmax}\, Q^{\pi}(s,a), \tag{3.12}$$

this new policy which take the action that gives the better result in each state assuming to follow the previous policy π afterward, satisfies the hypothesis of the previous theorem, thus it will be "better" than π. Such policy is known as *greedy policy*. Suppose that the new policy π' is equally good to π but not better, one would obtain the following

$$V^{\pi'}(s) = \max_{a \in \mathbf{A}} \mathbb{E}\left[R(s,a) + \gamma V^{\pi'}(s')\right], \tag{3.13}$$

but this is the Bellman equation, it follows that if this is satisfied π' is an optimal policy and π is too, since it is as good as the other.

3.5.3 Policy iteration

Once one knows how to evaluate a policy and how to improve it, it is easy to find an algorithm which find the best policy. This algorithm ,which is shown in 1, is based on three main steps, the first one is to decide the policy, the second is to evaluate such policy and the third aim to improve it thanks to policy improvement. Reiterating this steps at every time a better policy is exploited. Being it monotonically increasing it will find the optimal policy.

3.5.4 Value iteration

One problem of the policy iteration is that at every step it have to evaluate the new value function V^{π} which by the policy evaluation algorithm can take multiple steps, since the convergence is given to the limit. For this reason it would be helpful in terms of time spending to be able to truncate the steps necessary to get the value function. In 2 is shown the new approach, at each loop one step of policy evaluation and one step of policy improvement are taken. It has been shown that such approach can have faster convergence. From the algorithm it is clear how this is an approximation of the best policy due to θ.

3.6 Monte Carlo methods

In this section Monte Carlo methods for RL will be introduced briefly, due to their incompatibility with the offline version of RL which is the main purpose of this work. Monte Carlo methods do not need any previous knowledge of the

Algorithm 1 Policy iteration

```
Require: $V(s) \in \mathbf{R}$ and $\pi(s) \in \mathcal{A}\ \forall s \in S$
  while $stop! = True$ do
    while $\Delta > \theta$ do
      $\Delta \leftarrow 0$
      for $s \in S$ do
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s)) \left[r + \gamma V(s')\right]$
        $\Delta \leftarrow \max\left(\Delta, |v - V(s)|\right)$
      end for
    end while
    $stop \leftarrow True$
    for $s \in S$ do
      $a \leftarrow \pi(s)$
      $\pi(s) \leftarrow arg \max \sum_{s',r} p(s', r|s, \pi(s)) \left[r + \gamma V(s')\right]$
      if $a! = \pi(s)$ then
        $stop \leftarrow False$
      end if
    end for
  end while
```

Algorithm 2 Value iteration

```
Require: $V(s) \in \mathbf{R}\ \forall s \in S$ except $V(terminal) = 0$
  while $\Delta > \theta$ do
    $\Delta \leftarrow 0$
    for $s \in S$ do
      $v \leftarrow V(s)$
      $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s)) \left[r + \gamma V(s')\right]$
      $\Delta \leftarrow \max\left(\Delta, |v - V(s)|\right)$
    end for
  end while
  for $s \in S$ do
    $\pi(s) \leftarrow arg \max \sum_{s',r} p(s', r|s, \pi(s)) \left[r + \gamma V(s')\right]$
  end for
```

environment, but it is necessary to be able to simulate the transitions or be able to make experiment in the real application. In the other hand many off-policy methods have been proposed following the Monte Carlo logic, those are the basis of some offline RL methods, but they will be discussed in the next chapter 4. Having the wanted trajectories one is able to compute the expected value of the value function and determine the optimal policy trough it. As in DP some of the main important step will be introduced to be able to better understand all the algorithms of RL and offline RL.

3.6.1 Monte Carlo prediction and evaluation

Suppose to be able to sample trajectories following a certain policy, than it would be easy to make estimations of the value function of every state under the policy π. Indeed it would be the average of the return of each trajecotory which pass by the state s. It is easy to verified that such estimation converges to the real value $V^{\pi}(s)$ as the number of trajectories goes to infinity. In Monte Carlo methods is usually preferred to estimate Q-values instead of the value function, due to the fact that usually the model of the environment is not available. The way to estimate such value is the same as for the value function, by averaging the return of a trajectory form the state s where the specific action a is taken. It is important to notice how it is not sure that all the possible action are explored in every state, for example if one has a deterministic policy only one action per state will be exploited, thus using this kind of methods is very important to remember the exploration.

3.6.2 Monte Carlo control

Same ideas of policy improvement as in DP can be used in Monte Carlo methods, indeed theorem 3 is not referred specifically to DP settings. One must take more care with Monte Carlo methods due to its slow convergence that can lead a very dangerous errors. As in DP one can take the *greedy-policy*, evaluate it and repeat the process until the greedy policy does not update anymore.

3.7 Temporal Difference Learning

3.7.1 SARSA

SARSA methods land on on-policy algorithm, it is based on TD prediction method. In SARSA the aim is to learn Q-values, thus there is a small change on the temporal difference setting, but the idea is kept indeed the new update rule is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R(s, a) + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]. \tag{3.14}$$

Where $Q(s,a)$ is set to zero for final state. It is trivial to find a control algorithm to exploit an optimal policy from this update. Its convergence is based on the idea of visiting each state-action pair infinite times. Indeed with an ϵ-greedy approach which scales over time, but ensure the visit of all state-actions infinitely many times it converges to the optimal policy with probability 1. In 3 the pseudo-code for an update over a single episode is presented.

Algorithm 3 SARSA

```
Require: $Q(s,a) \in \mathbf{R}\ \forall s \in S$ and $\forall a \in \mathcal{A}$ except $Q(terminal,\cdot) = 0$
  $\alpha \in (0,1]$ $\epsilon \geq 0$
  while $s! = terminal$ do
      $a \sim \epsilon$-greedy policy
      take action $a$ and get $R$, $s'$
      $a' \sim \epsilon$-greedy policy
  end while
```

3.7.2 Q-learning

This method is the off-policy counterpart to SARSA, the main change of this methods is how the update is made

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left[R(s,a) + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\right].$$

By imposing that in the new state the action is taken as the one which gives the maximum Q-value, thus the only part affected by the policy is which state-actions pairs are visited and update. This is crucial because as it will be explained better in the offline reinforcement learning part the exploitation from the Q-learning of some policy has still a big effect on how the algorithm works and learn the best policy. Thus one should not think that off-policy algorithm are not based on policies and they can work directly in an offline setting as they are.

It has been shown that this approach suffer from the so-called *maximization bias*. Indeed, fixing a state, being the Q-value an approximation , one can have an error which overestimate the value of some action a. This error can easily propagate and lead to superior errors. To mitigate this fact many solutions has been proposed, the most known is *Double Q-learning* form Hasselt et al. 2015, an update version has been proposed in Fujimoto, Hoof, et al. 2018 which is called *Clipped Double Q-learning*, but is based on Deep RL which will be discussed in 3.11.3. The main idea to mitigate the overestimation of some action is to use another approximator to estimate the Q-value of the new state. Indeed in Double

Q-learning two approximation are used, leading to a bigger memory consumption, but better results. The new update rule are

$$Q_1(s_t, a_t) \leftarrow Q_1(s_t, a_t) + \alpha \left[R(s, a) + \gamma Q_2 \left(s_{t+1}, \underset{a}{argmax}\, Q_1(s_{t+1}, a)\right) - Q_1(s_t, a_t)\right]$$
$$Q_2(s_t, a_t) \leftarrow Q_2(s_t, a_t) + \alpha \left[R(s, a) + \gamma Q_1 \left(s_{t+1}, \underset{a}{argmax}\, Q_2(s_{t+1}, a)\right) - Q_2(s_t, a_t)\right]$$

As one can see in this new update the other approximator is used to estimate what the one updated think is the best action. This way leads to better results, but is still too strict to use the one not updated, in some cases it is possible that in general one approximator is biased based on the state and it overestimate the Q-values, this would lead to an approximation error which can keep going over steps.

Algorithm 4 Q-learning

```
Require: $Q(s, a) \in \mathbf{R}\ \forall s \in S$ and $\forall a \in \mathcal{A}$ except $Q(terminal, \cdot) = 0$
  $\alpha \in (0,1]\ \epsilon \geq 0$
  while $s! = terminal$ do
      $a \sim \epsilon$-greedy policy
      take action $a$ and get $R$, $s'$
      $a' \sim arg\max_{a \in \mathcal{A}} Q(s', a)$
      $Q(s, a) \leftarrow Q(s, a) + \alpha[R(s, a) + \gamma Q(s', a') - Q(s, a)]$
  end while
```

3.8 Policy gradient

Policy gradient methods are a type of reinforcement learning algorithm that are used to learn a policy for an agent to follow in an environment. These algorithms learn a policy by estimating the gradient of the expected reward with respect to the policy parameters, and using this gradient to update the policy in a way that increases the expected reward.

Policy gradient methods are particularly useful for learning continuous or high-dimensional policies, such as policies that involve selecting actions in a continuous action space. They can also be used to learn policies in environments where the reward signal is sparse or noisy, as they are able to make use of the entire episode of interactions with the environment to estimate the gradient of the expected reward.

One of the key advantages of policy gradient methods is their ability to learn directly from the environment, without the need to estimate a value function. This makes them particularly well-suited to learn tasks where the value function is difficult to estimate, or where it is not possible to represent the value function explicitly.

However, policy gradient methods also have several limitations, including the need for large amounts of data and computation, and the difficulty of ensuring that the learned policy is stable and converges to a satisfactory solution. Despite these limitations, policy gradient methods have been successfully applied to a wide range of reinforcement learning tasks and have contributed to significant progress in the field.

3.8.1 The log-derivative trick

The log-derivative trick is a result in probability theory and machine learning that allows us to rewrite the gradient of an expected value as an expectation of the product of the function and the gradient of the log probability.

To prove the log-derivative trick, one start by expressing the expected value of a function $f(x)$ as an integral over the distribution $p(x, \theta)$ where θ is a certain parameter:

$$\mathbb{E}_{p(x,\theta)}[f(x)] = \int f(x)p(x,\theta)dx$$

Applying the gradient over θ and using the chain rule to express the gradient of the expected value with respect to the distribution $p(x, \theta)$ as:

$$\nabla_\theta \mathbb{E}_{p(x,\theta)}[f(x)] = \nabla_\theta \int f(x)p(x,\theta)dx$$

$$= \int \nabla_\theta [f(x)p(x,\theta)]dx$$

$$= \int f(x)\nabla_\theta [p(x,\theta)]dx + \int \nabla_\theta [f(x)]p(x,\theta)dx$$

Taking $f(x)$ independent from θ, it follows that the gradient is equal to 0, thus the second term disappear. For what concern the first one, one can multiply and divide by $p(x, \theta)$ and remembering that

$$\nabla_\theta \log f(\theta) = \frac{\nabla_\theta f(\theta)}{f(\theta)}$$

, one get

$$\nabla_\theta \mathbb{E}_{p(x,\theta)}[f(x)] = \int f(x)\frac{\nabla_\theta p(x,\theta)}{p(x,\theta)}p(x,\theta)dx$$

$$= \int f(x)\nabla_\theta \log(p(x,\theta))dx = \mathbb{E}_{p(x,\theta}\left[f(x)\nabla_\theta) \log(p(x,\theta))\right]$$

This completes the proof of the log-derivative trick.

The log-derivative trick is a useful result in probability theory and machine learning that allows us to simplify calculations involving gradients and expectations. It is often used in optimization and learning, particularly in the context of reinforcement learning, where it is used to derive the policy gradient theorem.

3.8.2 Policy gradient theorem

The policy gradient theorem is a fundamental result in reinforcement learning that provides a way to estimate the gradient of the expected return with respect to the policy parameters. It is based on the idea of using the gradient of the log likelihood of the actions taken under the policy to estimate the gradient of the expected return.

To prove the policy gradient theorem, we start by expressing the expected return as an expectation over the distribution of states and actions:

$$\begin{aligned} J(\theta) = \mathbb{E}_{\pi(\theta)}[G_t] &= \mathbb{E}_{\pi(\theta)}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\right] \\ &= \sum_{s_t,a_t} p(s_t,a_t)\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\right] \\ &= \sum_{s_t,a_t} p(s_t,a_t)\left[\sum_{k=0}^{\infty} \gamma^k \mathbb{E}_{\pi(\theta)}[r_{t+k+1}|s_t,a_t]\right] \end{aligned}$$

where $p(s_t, a_t) = \mathbb{P}(s_t, a_t) = \mathbb{P}(s_t)\pi_\theta(a_t|s_t)$ is the joint probability of taking action a_t in state s_t.

Next, we can use the log-derivative trick to rewrite the gradient of the expected return as:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \sum_{s_t,a_t} p(s_t,a_t)\left[\sum_{k=0}^{\infty} \gamma^k \mathbb{E}_{\pi(\theta)}[r_{t+k+1}|s_t,a_t]\right] \\ &= \sum_{s_t,a_t} p(s_t,a_t)\nabla_\theta \left[\sum_{k=0}^{\infty} \gamma^k \mathbb{E}_{\pi(\theta)}[r_{t+k+1}|s_t,a_t]\right] \\ &= \sum_{s_t,a_t} p(s_t,a_t)\sum_{k=0}^{\infty} \gamma^k \nabla_\theta \mathbb{E}_{\pi(\theta)}[r_{t+k+1}|s_t,a_t] \end{aligned}$$

Now applying the log derivative trick one can obtain

$$\sum_{s_t,a_t} p(s_t,a_t)\sum_{k=0}^{\infty} \gamma^k \mathbb{E}_{\pi(\theta)}[r_{t+k+1}|s_t,a_t]\nabla_\theta \log \pi_\theta(a_t|s_t)$$

$$= \mathbb{E}_{\pi(\theta)}[\nabla_\theta \log \pi_\theta(a_t|s_t) Q^\pi(s_t, a_t)]$$

which is the desired result. This completes the proof of the policy gradient theorem.

It is worth noting that the policy gradient theorem assumes that the policy is differentiable with respect to the parameters, which may not always be the case. In practice, it may be necessary to use approximation techniques, such as Monte Carlo sampling, to estimate the gradient.

3.8.3 Use of policy gradient

Policy gradient methods are a type of reinforcement learning algorithm that are used to learn a policy for an agent to follow in an environment. These algorithms learn a policy by estimating the gradient of the expected reward with respect to the policy parameters, and using this gradient to update the policy in a way that increases the expected reward. Suppose that we want to learn a policy $\pi_\theta(a|s)$ that takes as input a state s and outputs a distribution over actions a. The expected return for a given state s under the policy π_θ can be represented by the expected value of the return over all possible actions and future states, given by:

$$J(\theta) = \mathbb{E}_{\pi_\theta}[G_t] = \mathbb{E}_{\pi_\theta}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\right] \tag{3.15}$$

where G_t is the return at timestep t, r_{t+k+1} is the reward at timestep $t+k+1$, and $\gamma \in [0,1]$ is the discount factor.

To learn the policy π_θ, it is possible to use a policy gradient algorithm to estimate the gradient of the expected return with respect to the policy parameters θ:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) Q^\pi(s_t, a_t)] \tag{3.16}$$

where $Q^\pi(s_t$ To estimate the gradient $\nabla_\theta J(\theta)$, there are a lot of algorithms such as the reinforce algorithm, which involves sampling actions from the policy and using the sampled returns to estimate the gradient. The reinforce algorithm can be expressed as:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{3.17}$$

where α is the learning rate. Alternatively, it is possible to use a baseline to reduce the variance of the gradient estimate. A common choice of baseline is the value function $V^\pi(s)$, which estimates the expected return for a given state s under the current policy π. The gradient estimate with a value function baseline can be expressed as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)(Q^\pi(s_t, a_t) - V^\pi(s_t))] \tag{3.18}$$

This form of the gradient estimate is known as the advantage function and is often used in policy gradient methods. There are many variants of the policy gradient algorithm, each with its own advantages and limitations. Some of the key factors to consider when choosing a policy gradient algorithm include the type of environment, the complexity of the policy, the amount of data available, and the computational resources available.

3.9 Actor-critic methods

Many times it is not possible to straightforwardly apply Q-learning to continuous action spaces, because in continuous spaces finding the greedy policy requires an optimization of a_t at every step. This optimization is too slow to be practical with large, unconstrained function approximators and nontrivial action spaces. To address this problem Actor-Critic have been introduced, it will be shown how this methods have take a great place into reinforcement learning methods, mostly thanks to deep learning. This approach combines aspects of both value-based and policy-based reinforcement learning algorithms. First let us introduce two of the main elements of this kind of algorithms.

The actor

The actor is responsible for determining the policy, which defines the agent's behavior in the environment. It selects actions based on the current state to maximize the expected cumulative reward. In the context of neural networks, the actor can be represented as a function approximator (like a deep neural network) that takes the environment state as input and outputs a probability distribution over possible actions. (policy-based)

The critic

The critic evaluates the actions chosen by the actor. It estimates the value or the expected cumulative reward of being in a certain state and following a certain policy. The critic helps the actor by providing feedback on the quality of its actions. Similar to the actor, the critic can also be implemented using a function approximator. (value-based)

Actor-Critic flow

Here is how this kind of model usually learn from the environment:

1. The agent interacts with the environment, receives a reward, and transitions to a new state. The agent's actions are taken from the actor distribution, in some cases some exploration is forced into this.

2. The critic's parameters are updated to minimize the difference between the predicted value of the current state and the actual received reward plus the estimated value of the next state. This helps the critic to better approximate the value function.

3. The actor's parameters are updated to maximize the expected reward, taking the critic's evaluation into account. This is done by performing policy gradient ascent using the advantage function, which represents how much better or worse an action is compared to the average action.

The actor-critic method combines the benefits of both value-based methods and policy-based methods, allowing for more stable and efficient learning in complex environments. By continuously updating both the policy and the value function, the actor-critic algorithm can find optimal or near-optimal policies in online reinforcement learning scenarios.

3.10 Model-based approach

Model-based reinforcement learning is a type of machine learning algorithms that are used to learn a policy for an agent to follow in an environment. In model-based reinforcement learning, the agent has access to a model of the environment, which allows it to make predictions about the consequences of its actions and plan its behavior accordingly. One advantage of model-based reinforcement learning is that it allows the agent to learn more efficiently, as it can use the model of the environment to explore and learn about the consequences of different actions without needing to actually take those actions in the environment. This can be particularly useful in situations where the cost of interacting with the environment is high. However, model-based reinforcement learning is not always the best approach, particularly in the context of offline reinforcement learning. In offline reinforcement learning, the agent must learn from a fixed dataset of past interactions with the environment, rather than learning online in real-time. In this setting, it is often difficult or impossible to build an accurate model of the environment, as the data may be incomplete or outdated. As a result, model-based reinforcement learning may not be able to effectively learn a policy in this setting. Instead, model-free reinforcement learning algorithms, which do not rely on a model of the environment, may be a more effective approach to offline reinforcement learning. These algorithms typically involve learning a value function, which estimates the expected reward for a given state or state-action pair, and using the value function to guide the agent's actions.

3.11 Deep-RL

Deep reinforcement learning is a subfield of machine learning that combines the concepts of reinforcement learning and deep learning. It involves using artificial neural networks, which are a type of machine learning algorithm that is inspired by the structure and function of the brain, to learn policies for decision-making and control tasks. Deep reinforcement learning has been applied to a wide range of problems, including control tasks in robotics, game playing, and natural language processing. It has achieved impressive results on many of these tasks, surpassing human-level performance in some cases. One of the key advantages of deep reinforcement learning is its ability to learn from high-dimensional sensory input, such as images or audio, and to learn policies that are able to adapt to changing environments. This makes it a powerful tool for learning complex tasks that require the ability to perceive and interpret the environment. However, deep reinforcement learning also poses several challenges, including the need for large amounts of data and computation, and the difficulty of interpreting the learned policies. Despite these challenges, deep reinforcement learning has the potential to revolutionize many areas of artificial intelligence and has attracted significant attention from researchers and industry alike. There are several reasons why deep reinforcement learning may be superior to traditional reinforcement learning in some cases:

- Ability to learn from high-dimensional sensory input: Deep reinforcement learning algorithms are able to learn from high-dimensional sensory input, such as images or audio, and can learn policies that are able to adapt to changing environments. This makes them particularly well-suited to learn complex tasks that require the ability to perceive and interpret the environment.

- Improved performance: In some cases, deep reinforcement learning algorithms have been able to achieve superior performance to traditional reinforcement learning algorithms on a wide range of tasks, including control tasks in robotics, game playing, and natural language processing.

- Ability to learn from unstructured data: Deep reinforcement learning algorithms are able to learn from unstructured data, such as images or audio, which can be difficult to process using traditional machine learning methods. This allows them to learn from data that may not be easily represented in a structured form.

- Improved generalization: Deep reinforcement learning algorithms are able to learn complex, non-linear relationships in the data and can generalize to unseen situations. This allows them to learn policies that are able to adapt to changing environments and perform well on tasks that may not have been seen during training.

However, it is important to note that deep reinforcement learning is not always the best approach, and traditional reinforcement learning algorithms may still be superior in some cases. The choice of algorithm will depend on the specific task.

3.11.1 Deep learning

The structure

Deep learning (DL) is a technique of learning that utilizes a function $f : X \to Y$, which is parameterized with $w \in \mathbb{R}^n$, such that the output y is determined by $f(x; w)$. The foundation of this field of research is the artificial neuron, modeled after the biological neurons found in the brains of animals and humans. A neuron is composed of multiple inputs, known as dendrites, that come from preceding neurons. The neuron processes these inputs and, if the value reaches a certain threshold, it sends a signal through its single output, called an axon. The inputs are processed by taking the weighted sum, adding a bias term b, and applying an activation function f, according to the relationship $y = f(\sum_{i=1}^{n} w_i x_i + b)$. In order to achieve optimal performance, the parameter set w must be adjusted through a process known as learning. A deep neural network (NN) is a structure that utilizes a collection of artificial neurons arranged in a series of processing layers. These layers perform non-linear transformations that guide the learning process. More complex structure can be created using artificial neurons. One very important structure which will be at the basis of most NN used in this thesis it a fully-connected hidden layer. The hidden layers have a value given by the first operation

$$h_j = f(\sum_{i=1}^{n} w_i x_i + b)$$

The output is given by reiterating the same operation with different weights and a different function

$$o_k = g(\sum_{j=1}^{m} a_j h_h + d)$$

Activation function The class of function f plays a key role in the NN structure. Their purpose is to introduce non-linearity into the output of the neuron, allowing the neural network to learn a broader range of functions and to make more complex decision boundaries.

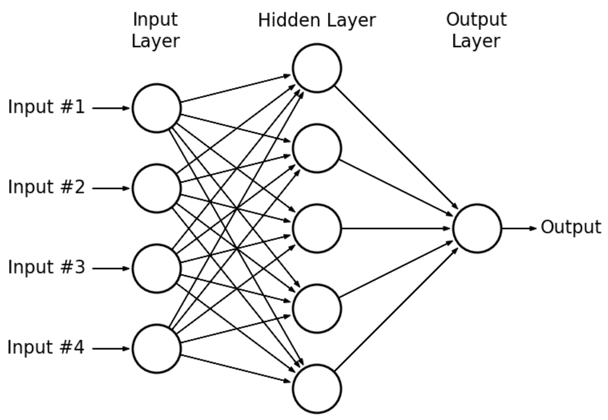

Figure 3.1: Representation of a fully connected hidden layer

Activation functions are typically applied element-wise to the output of a linear combination of the inputs and weights. The most commonly used activation functions in neural networks are:

- Sigmoid: which produces output values between 0 and 1 and is commonly used in the output layer of a binary classification problem.

$$f(x) = \frac{1}{1 + e^{-x}}$$

- ReLU (Rectified Linear Unit): which produces output values between 0 and x and is commonly used in the hidden layers of neural networks.

$$f(x) = max(0, x)$$

- Tanh (Hyperbolic Tangent): which produces output values between -1 and 1 and is commonly used in the hidden layers of neural networks.

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Softmax: which produces output values between 0 and 1 and is commonly used in the output layer of a multi-class classification problem. It normalizes the output values of the neuron to a probability distribution.

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Activation functions play a crucial role in the training and performance of a neural network. Each activation function has its own unique characteristics and is suited for different types of tasks. For example, the sigmoid activation function is commonly used in the output layer of a binary classification problem because it produces output values between 0 and 1, which can be interpreted as a probability. On the other hand, the ReLU activation function is commonly used in the hidden layers of neural networks because it allows for faster training and a reduction in the vanishing gradient problem.

The learning

The learning process of a neural network involves adjusting the parameters, such as the weights and biases, of the network in order to optimize its performance on a given task. This process is typically done through a method known as backpropagation. The training process starts by providing the neural network with a set of input-output pairs, called training data. The network uses these inputs to make predictions, and the error between the predictions and the actual outputs is calculated. This error is then propagated backwards through the network, adjusting the weights and biases of each layer in a way that minimizes the error. The process of adjusting the weights and biases is done using an optimization algorithm, such as stochastic gradient descent, which iteratively updates the parameters in the direction that minimizes the error. The training process continues until the error reaches an acceptable level or the performance of the network on a validation set stops improving. Once the network has been trained, it can be used for making predictions on new, unseen data. It's important to note that the learning process in NN's can be computationally expensive and time-consuming, especially for large datasets and complex architectures. In the next paragraphs the learning process will be summarize schematically.

Forward step Several inputs X are given to the NN and the outputs Y are archived.

Loss function A loss function, also known as a cost function, is a mathematical function that quantifies the difference between the network's predictions and the actual outputs. The goal of the training process is to minimize the value of the loss function. The choice of loss function depends on the task that the neural network is being trained to perform. For example, in a regression task, the mean squared error (MSE) is often used as the loss function, while in a classification task, the cross-entropy loss is commonly used. The most common loss functions used in neural networks are the mean squared error (MSE) and cross-entropy loss. MSE is typically used in regression tasks and measures the average squared difference between the predicted and actual values. Cross-entropy loss, on the other hand, is commonly used in classification tasks and measures the difference between the predicted probability distribution and the true distribution. The loss function plays an important role in the training of neural networks, as it provides a measure of the network's performance and guides the optimization process. The loss function is an essential component of the training process in a neural network, it quantifies the difference between the network's predictions and the actual outputs and guides the optimization process to minimize the error. In many cases using a pure loss function leads to overfitting problems. Overfitting occurs when a model becomes too complex and memorizes the training data instead of generalizing to new data. This can lead to poor performance on unseen data and a lack of generalization ability. Regularization is a technique used to combat overfitting by adding a penalty term to the loss function that discourages large values of the parameters, such as weights and biases. This helps to keep the model from becoming too complex by constraining the size of the parameters. The regularization term helps to keep the weights and biases small, which reduces the model's capacity to memorize the training data and forces it to learn more general features of the data. There are several types of regularization methods that can be used to regularize a loss in a neural network training. The most common ones are:

- L1 Regularization (also called Lasso regularization) which adds the absolute value of the weights to the loss function.

$$L_{L1} = L + \lambda \sum_i |w_i|$$

- L2 Regularization (also called Ridge regularization) which adds the square of the weights to the loss function.

$$L_{L2} = L + \lambda \sum_i w_i^2$$

- Dropout: which drop some neurons during the training process with a certain probability.

Where L is the un-regularized loss function and λ is the regularization strength, which is a hyperparameter that determines the amount of regularization applied to the model. The choice of regularization method depends on the specific problem and the characteristics of the data. L1 regularization is useful for sparse models, L2 regularization for models with small weights, and Dropout is useful for preventing overfitting. Regularization can be applied to any type of neural network and is a crucial step in training a neural network in order to prevent overfitting and improve the generalization performance of the model.

Backpropagation Backpropagation is an algorithm used to train neural networks by adjusting the weights and biases of the network in order to minimize the error between the network's predictions and the actual outputs. The algorithm uses the gradients of the loss function with respect to the network's parameters to update the weights and biases. The backpropagation algorithm starts by providing the network with an input and forwarding it through the layers to calculate the output. Then the error between the predicted output and the actual output is calculated using the loss function. The error is then propagated backwards through the network, starting from the output layer and working backwards through the hidden layers. The gradients of the loss function with respect to the network's parameters are calculated using the chain rule of calculus. The weights and biases are then updated using the gradients and a chosen optimization algorithm, such as stochastic gradient descent. In mathematical terms it is based on the chain rule which computes derivatives of composed function by multiplying local derivatives, in the case of a loss function one have

$$\frac{\partial L(f(x;w))}{\partial w_{i,j}} = \frac{\partial L(f(x;w))}{\partial f(x;w)} \frac{\partial f(x,w)}{\partial w_{i,j}}.$$

Therefore, the chain rule is used to pass the calculated global gradient of loss $\frac{\partial L(w)}{\partial w}$ backwards through the network, in the opposite direction of the forward pass. The process computes the local derivatives during the forward pass and estimates the local gradient of loss during backpropagation by determining the product of the local derivative and the local gradient of the loss of the connected neuron of the next layer. If the neuron has multiple connections to other neurons, the algorithm sums up all of the gradients.

Update of the parameters The last step is the update of the parameters such as weights and biases, those are updated during the training process in order to optimize the performance of the network on a given task. The update process can be represented mathematically by the following formula:

$$w_{i,j} = w_{i,j} + \Delta w_{i,j}$$

Where:

- $w_{i,j}$ is the weight between the i-th input unit and the j-th output unit
- $\Delta w_{i,j}$ is the change of weight

$\Delta w_{i,j}$ is calculated using the gradient of the loss function with respect to the weight, the learning rate and an optimization algorithm. For example, in the case of using stochastic gradient descent the update can be represented by the following:

$$\Delta w_{i,j} = -\eta \frac{\partial L}{\partial w_{i,j}}$$

Where:

- L is the loss function
- η is the learning rate

The learning rate is a hyperparameter that determines the step size of the updates. It controls how much the weights and biases are updated in each iteration. A small learning rate will make the optimization process slower but more precise. It's important to note that the update process is done for all the parameters, meaning that the weights and biases are updated after each iteration. In summary, the update of the parameters in a neural network is a crucial step in the training process that allows the network to learn and improve its performance on a given task. The update is typically done using an optimization algorithm such as stochastic gradient descent and the change of weight is calculated using the gradient of the loss function with respect to the parameters, the learning rate and the optimization algorithm.

3.11.2 Value-based methods

Value-based methods for reinforcement learning (RL) are a class of algorithms that estimate the value of a given state or action in order to make optimal decisions. These methods use a value function, represented as a neural network, to estimate the expected future reward for each state or action. The value function is trained using supervised learning techniques, with the goal of predicting the expected return for a given state or action. Deep neural networks (DNN) are commonly used to represent the value function in value-based methods for RL. DNNs have the ability to approximate complex, non-linear functions, making them well-suited for approximating the value function in RL. The DNN is trained using experience replay and a variant of stochastic gradient descent called Q-learning. In this section some of the most important and potent methods will be discussed.

3.11.3 DQN

In 2013 Mnih et al. Mnih, Kavukcuoglu, et al. 2013 have introduced the Deep Q-Network (DQN algorithm. DQN is a value-based method for reinforcement learning (RL) that uses a deep neural network to approximate the Q-function. The Q-function represents the expected return for a given state-action pair, and the goal of the DQN algorithm is to learn the optimal Q-function in order to make the best decisions. DQN uses experience replay to store past experiences and a variant of stochastic gradient descent called Q-learning to update the parameters of the neural network. This allows DQN to learn from raw sensory input, such as image or video, and to handle high-dimensional and continuous state spaces. One of the key innovations of DQN is the use of a technique called fixed Q-targets, which helps to stabilize the training process and improve the performance of the algorithm. This technique decouples the target Q-values from the current Q-values, allowing the network to learn from more stable targets. DQN has been able to achieve human-level performance on a wide range of Atari games and it's considered one of the most successful RL methods. It has also been used to control robots and other physical systems, and it's also a base for other more advanced RL methods like Double DQN, Dueling DQN, and Rainbow.

DQN algorithm

As it has been discussed in section 3.11.1 DNN can be used to approximate functions. Such approach can be very useful. The main way to define such an algorithm is trough its loss- function. In the case of DQN it is of the following form

$$L_Q(\theta) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta)^2\right]$$

where

$$y_i = \mathbb{E}_{s'\sim T(s)}\left[r + \gamma \max_{a'} Q(s',a';\theta)|s,a\right]$$

and ρ is a probability distribution over the action given the state. After reaching the convergence the policy obtained is given by

$$a = \max_{a\in\mathcal{A}} Q(s,a;\theta)$$

The DQN algorithm also incorporates an experience memory replay buffer, which is crucial to its success. This replay buffer is used to store past experiences and allows the algorithm to learn from them. Without the replay buffer, the algorithm would be prone to overfitting because the data is not independent and identically distributed. The replay buffer stores a limited number of experiences, replacing old experiences with new ones as they are collected. The experiences are collected in the form of tuples (s_t, a_t, r_t, s_{t+1}) using an epsilon-greedy policy. During the learning phase, a subset of these experiences, called a mini-batch, is used to update the parameters of the neural network. This improves the performance of the algorithm by reducing variance in the updates.

Prioritised Experience Replay In 2016 Schaul et al. Schaul et al. 2015 have introduced a new formulation of the experience memory replay buffer which is called Prioritized Experience Replay buffer (PER). The idea behind prioritized experience replay is that some experiences may be more valuable for learning than others. For example, experiences where the agent made a big mistake or achieved a high reward may be more informative for learning than other experiences. By prioritizing these experiences, the agent can learn more efficiently. PER can be implemented by assigning a priority value to each experience, based on a measure of its importance. For example, the temporal difference error (TD-error) or the absolute error between the predicted Q-value and the target Q-value can be used as a measure of importance. Experiences with higher priority values are more likely to be sampled during the learning phase. PER can provide a significant improvement in the performance of RL algorithms. It has been shown to speed up the convergence and stability of learning, and to improve the final performance of the agent. It is particularly useful in problems with sparse rewards and non-stationary environments.

3.12 Reinforcement Learning algorithms

In this section two reinforcement learning algorithms will be explained, the choice of this two particular approach it is based on the fact that they are the one with the best results in the experiments done in this study. Thus a brief introduction to their main attributes and functionality will be made.

3.12.1 TD3

First introduced by Fujimoto, Hoof, et al. 2018, it is a actor-critic method based on the idea of using a deep policy gradient, this approach had already been tried by Lillicrap et al. 2019 with an algorithm called Deep Deterministic Policy Gradient (DDPG) which have shown great results. Here first DDPG and than TD3 will be introduced.

DDPG

DDPG (Deep Deterministic Policy Gradient) is an actor-critic algorithm used in reinforcement learning for continuous action spaces. It maintains two neural networks: an actor network $\pi(s|\phi)$ and a critic network $Q(s, a|\theta)$, where s is the state, a is the action, ϕ and θ are the respective parameters.

Critic Update: DDPG uses a target Q-network to stabilize learning. The target Q-network parameters θ' are slowly updated to the current Q-network parameters:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta' \tag{3.19}$$

The critic loss function is defined as the mean squared error between the predicted Q-values and the target Q-values:

$$L(\theta) = \mathbb{E}\left[\left(Q(s,a|\theta) - (r + \gamma Q(s', \pi(s'|\phi)|\theta^{Q'})\right)^2\right] \tag{3.20}$$

Where r is the reward, s' is the next state, $\pi(s'|\phi)$ is the target actor's output for the next state s', and γ is the discount factor. The critic parameters θ are updated by minimizing $L(\theta)$ using gradient descent.

Actor Update: The actor is updated to maximize the expected return as estimated by the critic. The actor loss is defined as:

$$L(\phi) = \mathbb{E}\left[Q(s, \pi(s|\phi)|\theta)\right] \tag{3.21}$$

The actor parameters ϕ are updated by performing gradient ascent on $J(\phi)$. The gradient of the actor loss with respect to its parameters is given by:

$$\nabla_\phi L(\phi) = \mathbb{E}\left[\nabla_a Q(s,a|\theta|_{a=\pi(s)} \nabla_\phi \pi(s|\phi)\right] \tag{3.22}$$

The actor parameters ϕ are updated using this gradient.

Exploration: DDPG incorporates noise in action selection to encourage exploration:

$$a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t. \tag{3.23}$$

Where $\mathcal{N}_t$ is the noise added to the action.
These equations represent the core of the DDPG algorithm, balancing exploration and exploitation in a continuous action space by updating the actor and critic networks through policy and value gradients.

TD3

TD3 (Twin Delayed Deep Deterministic Policy Gradient) is an extension of DDPG designed to improve stability and performance. Similar to DDPG, it maintains two actor networks $\pi(s|\phi)$ and two critic networks $Q_1(s,a|\theta_1)$ and $Q_2(s,a|\theta_2)$.

Critic Update: The critic loss is similar to DDPG, with the addition of target value smoothing and target noise:

$$L(\theta_1) = \mathbb{E}\left[(Q_1(s,a|\theta_1) - (r + \gamma \min(Q_1'(s', \tilde{a}'), Q_2'(s', \tilde{a}')))^2\right] \tag{3.24}$$

$$L(\theta_2) = \mathbb{E}\left[(Q_2(s,a|\theta_2) - (r + \gamma \min(Q_1'(s', \tilde{a}'), Q_2'(s', \tilde{a}')))^2\right] \tag{3.25}$$

Where $Q'_1(s', \tilde{a}')$ and $Q'_2(s', \tilde{a}')$ are target critic values of the next state with added noise. Where

$$\tilde{a}' = \pi(s'|\phi) + \mathcal{N}_t, \tag{3.26}$$

$\mathcal{N}_t$ is the target policy smoothing noise. This is applied to reduce overestimation bias.

Actor Update: TD3 employs a trick called "target policy smoothing" to improve stability. The actor loss is defined as:

$$L(\phi) = \mathbb{E}\left[Q_1\left(s, \pi(s|\phi) + \mathcal{N}_t|\theta_1\right)\right] \tag{3.27}$$

The actor parameters ϕ are updated by performing gradient ascent on $L(\phi)$. The gradient of the actor loss with respect to its parameters is given by:

$$\nabla_\phi L(\phi) = \mathbb{E}\left[\nabla_a Q_1\left(s, a|\theta_1\right)|_{a=\pi(s)} \nabla_\phi \pi(s|\phi)\right] \tag{3.28}$$

The actor parameters ϕ are updated using this gradient.

TD3 achieves more stable and efficient learning in continuous action spaces by incorporating these modifications into the traditional DDPG algorithm.

3.12.2 Soft Actor Critic

Soft Actor-Critic (SAC) is a deep reinforcement learning algorithm that maximizes the entropy of the policy in addition to expected cumulative reward. It was introduced by Haarnoja et al. 2018. SAC maintains a stochastic policy $\pi(a|s, \phi)$ and two value functions: $Q_1(s, a|\theta_1)$ and $Q_2(s, a|\theta_2)$.

Actor Update: The actor is trained to maximize both expected reward and entropy, encouraging exploration. The objective function for the actor is:

$$L(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}}\left[\mathbb{E}_{a_t \sim \pi(\cdot|s_t)}\left[\alpha \log(\pi(a_t|s_t, \phi)) - \min_{i=1,2} Q_i(s_t, a_t|\theta_i)\right]\right] \tag{3.29}$$

Where $\mathcal{D}$ is the replay buffer and α is the temperature parameter.

Critic Update: The critic networks are updated to minimize the Bellman error with the entropy-regularized reward:

$$\begin{aligned} L(\theta_1) =& \mathbb{E}_{(s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}}\Big[\frac{1}{2}(Q_1(s_t, a_t|\theta_1) - (r_t + \\ &\gamma \mathbb{E}_{a_{t+1} \sim \pi(\cdot|s_{t+1})}\left[\min(Q_1(s_{t+1}, a_{t+1}|\theta_1), Q_2(s_{t+1}, a_{t+1}|\theta_2))\right]))^2\Big] \end{aligned} \tag{3.30}$$

Algorithm 5 Twin Delayed Deep Deterministic Policy (TD3)

```
1: Initialize actor network $\pi(s)$ and two Q-networks $Q_1(s,a)$, $Q_2(s,a)$ with random weights
2: Initialize target networks $Q'_1$, $Q'_2$, and $\pi'$ with same weights as their respective online networks
3: Initialize target policy smoothing noise $\epsilon \sim \text{clip}(\mathcal{N}(0,\sigma), -c, c)$
4: Initialize target policy smoothing coefficient $\sigma$, noise clip $c$, target update frequency $d$, and discount factor $\gamma$
5: Initialize replay buffer $\mathcal{D}$
6: for each episode do
7:     Initialize state $s$
8:     for each step in the episode do
9:         Select action $a$ from the current policy with noise: $a = \pi(s) + \epsilon$
10:        Execute action $a$, observe reward $r$ and next state $s'$
11:        Store $(s, a, r, s')$ in $\mathcal{D}$
12:        if step mod $d = 0$ then
13:            Sample a minibatch of transitions $(s_i, a_i, r_i, s'_i)$ from $\mathcal{D}$
14:            Compute target Q-values:
15:                $y_i = r_i + \gamma \min(Q'_1(s'_i, \pi'(s'_i) + \epsilon'), Q'_2(s'_i, \pi'(s'_i) + \epsilon'))$
16:            Update Q-networks:
17:                $L = \frac{1}{N} \sum_i (y_i - Q_1(s_i, a_i))^2 + (y_i - Q_2(s_i, a_i))^2$
18:        end if
19:        if step mod $d = 0$ then
20:            Update the actor network using the deterministic policy gradient:
21:                $\nabla_\theta J(\phi_\pi) = \frac{1}{N} \sum_i \nabla_a Q_1(s_i, a)|_{a=\pi(s_i)} \nabla_\theta \pi(s_i)$
22:        end if
23:        Update target networks with soft updates:
24:            $\theta_{Q'_1} = \tau\theta_{Q_1} + (1-\tau)\theta_{Q'_1}$
25:            $\theta_{Q'_2} = \tau\theta_{Q_2} + (1-\tau)\theta_{Q'_2}$
26:            $\theta_{\pi'} = \tau\theta_\pi + (1-\tau)\theta_{\pi'}$
27:    end for
28: end for
```

$$L(\theta_2) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{D}}\Big[\frac{1}{2}(Q_2(s_t,a_t|\theta_2) - (r_t + \gamma\mathbb{E}_{a_{t+1}\sim\pi(\cdot|s_{t+1})}\left[\min(Q_1(s_{t+1},a_{t+1}|\theta_1), Q_2(s_{t+1},a_{t+1}|\theta_2))\right]))^2\Big] \tag{3.31}$$

Algorithm 6 Soft Actor-Critic (SAC)

```
1: Initialize critic networks $Q_1(s,a)$, $Q_2(s,a)$, and actor network $\pi(s)$ with random weights
2: Initialize target networks $Q'_1$, $Q'_2$, and $\pi'$ with same weights as their respective online networks
3: Initialize replay buffer $\mathcal{D}$ and temperature parameter $\alpha$
4: for each episode do
5:     Initialize state $s$
6:     for each step in the episode do
7:         Select action $a$ from the current policy: $a = \pi(s) + \epsilon$, where $\epsilon \sim \mathcal{N}(0,1)$
8:         Execute action $a$, observe reward $r$ and next state $s'$
9:         Store $(s,a,r,s')$ in $\mathcal{D}$
10:        Sample a minibatch of transitions $\mathcal{A}(s_i,a_i,r_i,s'_i)$ from $\mathcal{D}$
11:        Compute target Q-values:
12:            $y_i = r_i + \gamma \min_{j=1,2} Q'_j(s'_i, \pi'(s'_i)) - \alpha \log(\pi(s'_i, \theta_\pi))$
13:        Update critic networks by minimizing the mean squared loss:
14:            $L = \frac{1}{N}\sum_i (y_i - Q_1(s_i,a_i))^2 + (y_i - Q_2(s_i,a_i))^2$
15:        Update the actor network:
16:            $\nabla_\phi \frac{1}{N}\sum_{s\in\mathcal{A}} \left(\min_{j=1,2} Q_{\theta_j}(s_i,a)|_{a=\pi(s_i)} - \alpha \log \pi_\phi(a|s_i)\right)$
17:        Update target networks with soft updates:
               $\theta_{Q'_1} = \tau\theta_{Q_1} + (1-\tau)\theta_{Q'_1}$
               $\theta_{Q'_2} = \tau\theta_{Q_2} + (1-\tau)\theta_{Q'_2}$
               $\theta_{\pi'} = \tau\theta_\pi + (1-\tau)\theta_{\pi'}$
18:    end for
19: end for
```

Chapter 4

Offline Reinforcement Learning

As it has been shown in the previous section that RL is an *online* learning. This kind of learning is both the strength and the weakness of RL. Meanwhile the online setting allows the algorithm to find the optimal policy (or at least a policy which is very close in term of performance to the optimal), on the other hand in some settings it is not possible or it is not convenient to do this kind of experiment. Applying RL algorithms to real scenarios can be very expensive or dangerous, for example letting a robot have his own experience to learn the optimal behaviour can lead to its destruction or someone being harmed. Nevertheless the power of RL have to be used. In the past years data-driven methods have taken a major role in ML, thanks to their ability of scaling with data, from visual recognition, natural language processing. The aim of incorporate such behaviour in RL algorithm has become more and more urgent. On the other hand such an approach has shown other types of challenges, those will be discussed in 4.2.
Offline Reinforcement Learning (ORL), also called Batch Reinforcement Learning, algorithm have been introduced as a proposal to handle this challenge. The main difference from RL is that there is no more interacting with the environment, thus the algorithm is not able to explore new states and policies. ORL approach can be effectively seen as a data-driven formulation of the RL problem. In this chapter an overview of the main methods to address this challenge will be introduced and discussed, for what concern specific algorithms those will be discussed in the next chapter 5.

4.1 Setting of Offline Reinforcement Learning

As it has been discussed in this new setting the algorithm is not able to interact with the environment to obtain new observations and rewards, instead the batch of *"experience"* is fixed.

$$\mathcal{D} = \left\{ (s_t^i, a_t^i, s_{t+1}^i, r_t^i) \right\} \tag{4.1}$$

are our set of data, where every element is composed by *state*, *action*, *next-state* and *immediate reward.* It can be composed by trajectories or random sample of action in a random state. The dataset is used to learn the optimal policy, it can be seen as the *training set* of a normal supervised learning problem. From this collection the algorithm have to exploit sufficient understanding of the underlying MDP and obtain the best policy π^*. From the dataset it is possible to exploit the distribution of the sample states $d^{\pi_\beta}(s)$ and the behaviour policy π_β. This two assets can be both given by a previous algorithm or be approximate from the dataset.
To address this kind of problem many methods have been tried, most of the methods based on Q-learning are easily adapted to such a scenario. In the other hand, as discussed when this kind of methods have been introduced, most of them rely on acquiring new data following a specific policy, it is trivial to understand that this is not possible, therefore modification and additions have been proposed to mitigate the inaccuracy caused by this.

4.2 Difficulties of Offline Reinforcement Learning

As one can expect this new setting poses new and different problems compared to RL, some easily addressable other more difficult to understand and propose a solution. In this section some of the most challenging and known difficulties will be discussed.

4.2.1 Distributional shift

Most of RL algorithms are based on making assumptions on which policy could give better results , after this consulting the environment such assumptions can be confirmed or not. It is clear that such thing is not possible in a static setting as ORL. This cause a major problem when such policy have to be evaluated or update, since π is different from π_β this can lead to out of distribution states where computing the values have an approximating error such that the algorithm could misinterpret the policies. Most of today's machine learning tools does not permit to train and evaluate on batch of data from different distributions. Never the less evaluating new ways to reach some goal is the base of RL and ORL, thus this problem has to be faced in some way and can not be avoid. Proposal to face this difficulty will be discussed in details where ORL algorithms will be introduced.

4.2.2 Exploration

It is intuitive that having a static set of data it is not possible to have an exploration, so if the dataset is not optimal and does not show some region with high reward it is possible to not be able to exploit an optimal policy. On the other hand such a problem is not the scope of ORL and there is no way to address this challenge. Since this is the case it is not productive to try to find solutions and in the following discussion the dataset **D** will be assumed to be adequate to find optimal solutions.

4.3 Importance sampling methods

In this section a group of methods that rely on direct estimation of the policy will be introduced, most of them use importance sampling as a tool to directly evaluate the return of a policy or compute the policy gradient.
Importance sampling is a variance reduction technique use in Monte Carlo methods, the main idea is that some sample are more important from the others to estimate some quantity. More importantly for this work, importance sampling can be used to estimate one distribution sampling from another, this can be widely used in ORL.

4.3.1 Basic theory

Suppose to wish to estimate the expected value of a function of some random variable

$$X : \Omega \to \mathbb{R}$$

in some probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Thus we have to solve

$$\mathbb{E}_p[f(X)] = \int_{\mathbf{D}} f(x)p(x)dx,$$

where p is the probability density function of X, $\mathbf{D} \subset \mathbb{R}^d$ and outside of it the probability is null. Let now q be a probability density function on $\mathbb{R}^d$, such that $q(x) > 0$ when $f(x)p(x) \neq 0$. Than the following holds

$$\mathbb{E}_p[f(X)] = \int_{\mathbf{D}} f(x)p(x)dx = \int_{\mathbf{D}} \frac{f(x)p(x)}{q(x)} q(x)dx = \mathbb{E}_q\left[\frac{f(x)p(x)}{q(x)}\right] \tag{4.2}$$

where the second expected value is computed for $X \sim q$. The factor $\frac{p(x)}{q(x)}$ is called *likelihood-rate*, the distribution p *nominal-distribution* and q *importance distribution.*

Multiplying the function for the *likelihood-rate* it is possible to compensate the fact that the samples are from a different distribution.
Let now $\mathbf{Q} = \left\{x \in \mathbb{R}^d | q(x) > 0\right\}$, than the following calculations follows directly

$$\begin{aligned}\mathbb{E}_q\left[\frac{f(x)p(x)}{q(x)}\right] &= \int_{\mathbf{Q}} \frac{f(x)p(x)}{q(x)} q(x)dx = \int_{\mathbf{Q}} f(x)p(x)dx \\ &= \int_{\mathbf{D}} f(x)p(x)dx + \int_{\mathbf{Q}\cap\mathbf{D}^c} f(x)p(x)dx - \int_{\mathbf{Q}^c\cap\mathbf{D}} f(x)p(x)dx \\ &= \int_{\mathbf{D}} f(x)p(x)dx, \end{aligned} \tag{4.3}$$

since when $x \in \mathbf{Q} \cap \mathbf{D}^c$ $p(x) = 0$ and if $x \in \mathbf{Q}^c \cap \mathbf{D}$ $f(x)p(x) = 0$. It is worth noting how by definition of $\mathbf{Q}$ it is impossible to have a point $x \in \mathbf{Q}$ such that $q(x) = 0$, thus the problem is well posed and there are no problem with the fraction. This method can be expanded to work with different function and same *importance distribution*, it is enough to ask that q satisfies the requirements for each function.
It is possible now applying the standard unbiased estimator for the mean value to the new formulation, obtaining

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} \frac{f(x_i)p(x_i)}{q(x_i)}. \tag{4.4}$$

Theorem 3. *Let $\hat{\mu}$ be the estimator of the expected value $\mathbb{E}_p[f(x)] = \mu$, p and q probability density as before, than $\mathbb{E}_q[\hat{\mu}] = \mu$ and $Var_q[\hat{\mu}] = \frac{\sigma_q^2}{n}$. Where*

$$\sigma_q^2 = \int_{\mathbf{Q}} \frac{(f(x)p(x))^2}{q(x)} dx - \mu^2 = \int_{\mathbf{Q}} \frac{(f(x)p(x) - \mu q(x))^2}{q(x)} dx. \tag{4.5}$$

Following the steps in the previous part $\mathbf{Q} = \{x|q(x) > 0\}$ can be substitute to $\mathbf{D} = \{x|p(x) > 0\}$.

Proof. Directly from 4.3 one can obtain that

$$\mathbb{E}_q\left[\frac{f(x)p(x)}{q(x)}\right] = \int_{\mathbf{D}} f(x)p(x)dx = \mu. \tag{4.6}$$

Applying the expected value to $\hat{\mu}$ the following holds

$$\mathbb{E}_q[\hat{\mu}] = \mathbb{E}_q\left[\frac{1}{n}\sum_{i=1}^{n} \frac{f(x_i)p(x_i)}{q(x_i)}\right] = \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}_q\left[\frac{f(x_i)p(x_i)}{q(x_i)}\right] = \frac{1}{n} n\mu = \mu, \tag{4.7}$$

showing that $\hat{\mu}$ is an unbiased estimator for the expected value of some function f under the probability density distribution p. To show the variance of the estimator it is enough to compute it explicitly remembering that samples x_i are i.i.d. and that

$$\begin{aligned} Var_q\left[\frac{f(x)p(x)}{q(x)}\right] =& \mathbb{E}_q\left[\left(\frac{f(x)p(x)}{q(x)}\right)^2\right] - \mu^2 \\ =& \int_Q \left(\frac{f(x)p(x)}{q(x)}\right)^2 q(x)dx - \mu^2 = \sigma_q^2, \end{aligned}$$

it follows directly that

$$\begin{aligned} Var_q[\hat{\mu}] =& Var_q\left[\frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)p(x_i)}{q(x_i)}\right] = \frac{1}{n^2}\sum_{i=1}^{n} Var_q\left[\frac{f(x_i)p(x_i)}{q(x_i)}\right] \\ =& \frac{1}{n^2}\sum_{i=1}^{n}\sigma_q^2 = \frac{\sigma_q^2}{n} \end{aligned} \tag{4.8}$$

□

It is important to notice how even if $Var[f(X)]$ is finite there is no guarantee that σ_q is bounded too.

4.3.2 Self-Normalized importance sampling

In many real cases it is possible that both the distribution p and q are known up to a normalizing constant (i.e. $p(x) = cp_0(x)$ and $q(x) = bq_0(x)$, where only $p_0(x)$ and $q_0(x)$ are known). To deal with this problem it is possible to compute the ratio $W(x) = \frac{p(x)}{q(x)}$ and consider the following estimation

$$\tilde{\mu}_q = \frac{\frac{1}{n}\sum_{i=1}^{n}\frac{f(x_i)p(x_i)}{q(x_i)}}{\frac{1}{n}\sum_{i=1}^{n}\frac{p(x_i)}{q(x_i)}} = \frac{\sum_{i=1}^{n} W(x_i)f(x_i)}{\sum_{i=1}^{n} W(x_i)} \tag{4.9}$$

is called self-normalized importance sampling estimate. It is clear how the assumption on p and q must be stronger, indeed we need that $q(x) > 0$ whenever $p(x) > 0$.

Theorem 4. *Under the assumption on p and q made before, the self-normalized importance sampling estimate converges in probability to the expected value $\mathbb{E}_p[f(x)]$ (i.e. $\mathbb{P}\left(\lim_{n\to\infty}\tilde{\mu}_q = \mu\right) = 1$)*

Proof. By applying the strong law of large number to the numerator and denominator separately we obtain that

$$\mathbb{P}\left(\frac{1}{n}\sum_{i=1}^{n} W(x_i)f(x_i) \underset{\infty}{\to} \mu\right) = 1$$
$$\mathbb{P}\left(\frac{1}{n}\sum_{i=1}^{n} W(x_i)f(x_i) \underset{\infty}{\to} 1\right) = 1.$$

Where in the second the expected value is done with $f(x) = 1$ for almost every $x \in \Omega$. $\square$

Theorem 5. *The estimator in equation 4.9 is biased.*

Proof. As we have seen in 3 the estimator

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} \frac{f(x_i)p(x_i)}{q(x_i)},$$

is unbiased. Now take into consideration the estimator of the ratio

$$\hat{\omega} = \frac{1}{n}\sum_{i=1}^{n} W(x_i),$$

this is an unbiased estimator of the normalizing constant. Indeed

$$\begin{aligned}\mathbb{E}_q[\frac{1}{n}\sum_{i=1}^{n} W(x_i)] =& \frac{1}{n}\sum_{i=1}^{n}\int_{\mathbf{Q}} W(x_i)q(x)dx \\ =& \int_{\mathbf{Q}} \frac{p(x)}{q(x)}q(x)dx = \int_{\mathbf{D}} p(x)dx = 1 = \mathbb{E}_q[W(x)]\end{aligned}$$

Let X be a strictly positive random variable random, than by Jensen inequality

$$\mathbb{E}[1/X]$$

$\square$

4.3.3 Off-policy evaluation via importance sampling

The value function of a given policy is an expected value, where actions are taken in the policy distribution. In offline reinforcement learning such policy can not be experimented to gain an effective value of such function, since the dataset follow a different distribution π_β, importance sampling is a natural way to estimate such value.

$$\begin{aligned}V^\pi(s) =& \mathbb{E}_{a\sim\pi}[G_t|s = s_0] = \mathbb{E}_{a\sim\pi}\left[\sum_{i=0}^{K}\gamma^i r(s_i, a_i)|s = s_0\right] \\ =& \mathbb{E}_{a\sim\pi_\beta}\left[\prod_{i=0}^{K}\frac{\pi(a_i|s_i)}{\pi_\beta(a_i|s_i)}\sum_{i=0}^{K}\gamma^i r(s_i, a_i)\middle| s = s_0\right] \\ \approx& \frac{1}{n}\sum_{j=1}^{n}\prod_{i=0}^{K}\frac{\pi(a_i^j|s_i^j)}{\pi_\beta(a_i^j|s_i^j)}\sum_{i=0}^{K}\gamma^i r(s_i^j, a_i^j) \\ =& \sum_{j=1}^{n} W_K^j \sum_{i=0}^{K}\gamma^i r(s_i^j, a_i^j)\end{aligned}$$

Such estimator can have a potentially unbounded variance, due to the product of importance weights. To minimize effect it is possible to use the self-normalized importance sampling by dividing for the weights sum, which is still a consistent estimator and can have much lower variance. Precup et al. 2000 suggest that it is possible to improve the estimator by studying the problem. In this case since $r(s_i, a_i)$ does not depend on future states and action it is possible to rewrite the expected value as follow

$$\begin{aligned} &\mathbb{E}_{a\sim\pi_\beta}\left[\prod_{i=0}^{K}\frac{\pi(a_i|s_i)}{\pi_\beta(a_i|s_i)}\sum_{i=0}^{K}\gamma^i r(s_i,a_i)\middle| s=s_0\right] \\ =&\mathbb{E}_{a\sim\pi_\beta}\left[\sum_{i=0}^{K}\prod_{t=0}^{t'}\frac{\pi(a_i|s_i)}{\pi_\beta(a_i|s_i)}\gamma^i r(s_i,a_i)\middle| s=s_0\right] \approx \frac{1}{n}\sum_{j=1}^{n}\sum_{i=0}^{K} w(x)_i^j \gamma^i r(s_i^j, a_i^j). \end{aligned} \tag{4.10}$$

Again this estimator can have very high variance, even weighing this in many application is still too unstable to be used.

4.3.4 Doubly Robust Estimator

Jiang and Li 2016 have introduced the Doubly Robust Estimator, which is a good estimator if π_β is known or the model is correct, but still remain unbiased if the former does not hold and if the latter does not hold than the error is proportionate to the model error.

$$\begin{aligned} DR(V^\pi(s)) =& \sum_{j=1}^{n}\sum_{i=0}^{K} w(x)_i^j \gamma^i r(s_i^j, a_i^j) \\ &- \sum_{j=1}^{n}\sum_{i=0}\gamma^i\left(w_i^j Q^\pi(s_t,a_t) - w_{i-1}^j \mathbb{E}_{a\sim\pi}\left[Q^\pi(s_t,a)\right)\right]. \end{aligned} \tag{4.11}$$

In P. S. Thomas and Brunskill 2016, this this method has been studied in details and a mathematical derivation of the infinite horizon formulation has been given. Even though this is an improvement of the importance sampling strategy more sophisticated estimator have been introduced by Y.-X. Wang et al. 2017 and Farajtabar et al. 2018.
Having a good estimator is usually not enough to address ORL problem, since the goal is to find a policy which has overall good performance the guarantee to have a good performance is very important. To this end the study of confidence interval is the key, some of them are based on distributional assumption and bootstrapping.P. Thomas et al. 2015 derived confidence bounds based on concentration inequalities specialized to deal with the high variance and potentially large range of the importance weighted estimator. Estimators and confidence interval together can be used to look for new policies having a lower bound on their performance. This can be used in policy improvement.

4.3.5 The off-policy gradient

Importance sampling methods can be used to directly estimate other useful quantity. As we have introduced in section 3.8 one way to improve a policy is thorough the policy gradient. In this section the way to use importance sampling to estimate such value will be studied and than use of it will be explained. First recall the formulation of the policy gradient

$$\nabla_\theta V^{\pi_\theta} = \mathbb{E}_{a\sim\pi_\theta}\left[\sum_{t=0}^{K}\gamma t \nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}(s_t,a_t)\right]. \tag{4.12}$$

As done in the previous section one can apply the importance sampling method directly to this expected value, obtaining

$$\begin{aligned}\nabla_\theta V^{\pi_\theta} =& \mathbb{E}_{a\sim\pi_\theta}\left[\prod_{i=0}^{K}\frac{\pi(a_i|s_i)}{\pi_\beta(a_i|s_i)}\sum_{t=0}^{K}\gamma^t \nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}(s_t,a_t)\right] \\ \approx& \sum_{j=1}^{n} W_K^j \sum_{t=0}^{K}\gamma^i \nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}(s_t,a_t).\end{aligned} \tag{4.13}$$

As before we can lower the variance by using the self-normalized importance sampling and again remembering the properties of the problem it is possible to rewrite 4.13 in a per-decision form.

$$\hat{A}(s_t,a_t) = \sum_{t'=t}^{K}\gamma^{t'-t} r(s_{t'},a_{t'}) - b(s_t^i) \tag{4.14}$$

the reward does not depend on future state and action, thus as before we can drop those dependencies leading to the form

$$\nabla_\theta V^{\pi_\theta} \approx \sum_{j=1}^{n}\sum_{t=0}^{K}\gamma^i W_t^j \nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{K}\left(\frac{W_{t'}^j}{W_t^j}(\gamma^{t'-t} r(s_{t'},a_{t'}) - b(s_t^i)\right). \tag{4.15}$$

Having this form it is possible to sample form π_β to estimate the gradient of another policy. All the criticism discussed for the estimation of the value function are still valid, indeed the high variance is one of the main problem in real applications. Methods to mitigate this problem have been introduced giving better result, but, even if they are mathematically correct, they need a huge amount of samples to be accurate. One example is the *soft max* regularization.By rewriting the estimator as

$$\sum_{j=1}^{n} W_K^j \sum_{t=0}^{K}\gamma^i \nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}(s_t,a_t) + \lambda \log\left(\sum_{i=1}^{n} W_K^i\right). \tag{4.16}$$

It is worth noting how this formulation is still consistent since

$$\lim_{n\to\infty} \sum_{i=1}^{n} W_K^i = 1.$$

There are plenty of possible regularization term, but Levine and Koltun 2013 has shown how this is the best, by trial and error.

4.4 Dynamic programming for Offline Reinforcement Learning

Dynamic programming methods as discussed in 3.5 are based on learn value functions or Q-values. Most of them do this through approximating the function by simple regressions or deep learning methods. In principle such way to address the problem should give good performance without any adjustment, indeed it should be enough to set a fixed buffer of *states*, *actions*, *next states* and *rewards*. On the other hand as discussed in 3.12.2 new method such as actor critic or SAC even if consider to be off-policy algorithms they are based on sampling from the actor distribution during the exploration, in ORL setting this is not possible leading to distributional shift between the behaviour policy π_β and the actor policy π. Many methods have been proposed but two main categories of solutions shine. One is to force the actor policy to be close to the behaviour one and the other is based on uncertainty, both will be discussed in this section trying to emphasize pro and cons of both of them. Before going deep into the algorithms it is important to stress and understand better how distributional shift can have impact on such algorithms.

4.4.1 Impact of distributional shift on dynamic programming methods

Distributional shift has two different impact on dynamic programming methods, one is the shift over the states and the other over the actions. The latter affect the training part and the former the test. This two problems have to be addressed separately and with different methods.

Action distributional shift Having a fixed dataset of states, actions and next-states does not permit an exploration and can lead to errors that can be accumulate over time. Take into consideration a standard Q-learning where in each step the value of a given (s_t, a_t) is update with the following

$$Q(s_t, a_t) = Q(s_t, a_t) + \lambda \left[r(s_t, a_t) + \gamma \max_{a_{t+1}\in \mathbf{A}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]. \tag{4.17}$$

As one can expect the action which maximize the Q-value can be different from the one from the dataset. Thus the approximation of $Q(s_{t+1}, a_{t+1})$ will be affected by an error. In most of the algorithms used today the maximum is changed, instead it is preferred to evaluate the Q-value along a policy π , even in this case the distributional shift between π and π_β gives an error. While this seems to have a minor effect on final result due to the approximation over a dataset which is consider to be dense and with enough data to cover the whole space this is not the case. To be better understood take into consideration an approximating error which always gives Q-value greater then the real one in some state (S, A). If this is the case actor-critic methods will try to enforce such state-action situation due to the previous great results. Having a positive approximation error such estimation can be over estimate at every step. This effect is due to the fact that the Q-value has to be evaluated in states where it will never be train in. Having such errors in every step it can lead to an arbitrary bad performance. In RL setting this is mitigate thanks to the possibility from the algorithm to try such a policy and learn how good or bad it really is.

State distributional shift This type of distributional shift mostly affect the algorithm at test time. It is natural that having a fixed dataset, in real application it is likely to find state that were not in the training set. This can be because initial state in the dataset or a policy which differs from the behaviour policy, while in the first case it is not possible to fix such a problem in the second constraining the policy to be close to the behaviour one is possible. One of the attempt made is to constrain the Kullback divergence between the two policies to be bounded. It has been studied and prove that such an attempt help the state distributional shift problem, in the other hand it can leads to sub-optimal solutions. In this case it is important to choose whether is better to achieve better performance or having a more safe algorithm that ensure to reach the goal in a longer and less efficient ways.

4.4.2 Policy constraints methods

The main idea behind policy constraints methods is to make the policy be as close as possible to the behaviour policy. This limits the evaluation out-of-distribution actions, leading to less approximating errors. If the policy π is equal to π_β one would not have accumulation of error and standard results of RL can be applied. During the training of an ORL algorithm the only states available are the one from the dataset, thus it is not necessary to impose a constrain also on $d^\pi(s)$. Such constraints can be applied in two different ways:

Policy penalty Policy penalty methods directly incorporate a penalty into the update of the algorithms approximator. As an example if on consider an actor-critic method it is possible to add a penalty into the update of both the actor and the Q-value. This penalty is made such that it increase its magnitude as π differs from π_β.

$$Q^{\pi}_{k+1} \leftarrow \arg\min_{Q} \Big\{ \mathbb{E}_{(s,a,s')\in D}\Big[\Big(Q(s,a) - \Big(r(s,a) + \gamma \mathbb{E}_{a'\sim\pi}\Big[\hat{Q}^{\pi}(s',a')\Big]\Big) - \alpha\gamma D(\pi_k(\cdot|s')|\pi_\beta(\cdot|s')\Big)^2\Big]\Big\} \tag{4.18}$$

$$\pi_{k+1} \leftarrow \arg\max_{\pi} \Big\{\mathbb{E}_{s\in D}\Big[\mathbb{E}_{a\sim\pi}\Big[\hat{Q}(s,a) - \alpha D(\pi_k(\cdot|s)|\pi_\beta(\cdot|s)\Big]\Big]\Big\} \tag{4.19}$$

It is also possible to incorporate the penalty modifying the reward substrating the penalty. Various formulation of the penalty has been proposed, the most well known is the Kullbabk divergence.

Policy constraints Another way to force the learned policy to be close to the behaviour one is to constrain it by force. Adding a verification where the two policy have to be in some way similar, this term as before can be various and different proposal have been made. Incorporating it into a actor-critic method the following formulation would come out

$$Q^{\pi}_{k+1} \leftarrow \arg\min_{Q} \Big\{ \mathbb{E}_{(s,a,s')\in D}\Big[\Big(Q(s,a) - \Big(r(s,a) + \gamma \mathbb{E}_{a'\sim\pi}\Big[\hat{Q}^{\pi}(s',a')\Big]\Big)\Big)^2\Big]\Big\} \tag{4.20}$$

$$\pi_{k+1} \leftarrow \arg\max_{\pi} \Big\{\mathbb{E}_{s\in D}\Big[\mathbb{E}_{a\sim\pi}\Big[\hat{Q}(s,a)|\pi_\beta(\cdot|s')\Big]\Big]\Big\} \qquad s.t.\ D(\pi_{k+1}|\pi_\beta) < \epsilon \tag{4.21}$$

for some ϵ chosen.

The way to define such a constraint or penalty differs from one method to the other but they can be summarized in three different categories.

Explicit f-divergence policy constraints

Definition 3. *Let* $\mathbf{P}$ *and* $\mathbf{Q}$ *be two different probability distribution over* Ω*, such that* $\mathbf{P} << \mathbf{Q}$ *(i.e.* $\mathbf{P}$ *is absolute continuous with respect to* $\mathbf{Q}$*, for every* $\mathbf{Q}$ *measurable set* A *such that* $\mathbf{Q}(A) = 0$ *than* $\mathbf{P}(A) = 0$*). Then, for a convex function*

$$f : [0,\infty) \rightarrow (-\infty,\infty] \qquad s.t.\ f(1) = 0,\ f(0) = \lim_{t^+\rightarrow 0} f(t)$$

the f-divergence from **P** *from* **Q** *is define as*

$$D_f(\mathbf{P}||\mathbf{Q}) = \int_\Omega f\left(\frac{d\mathbf{P}}{d\mathbf{Q}}d\mathbf{Q}\right) \tag{4.22}$$

Most known examples of f-divergence in this field are

KL-divergence

$$D_{KL}(\mathbf{P}||\mathbf{Q}) = \int_\Omega p(x)\log\left(\frac{p(x)}{q(x)}\right)dx \tag{4.23}$$

This is a distance between the two probability distribution. Its formulation is similar to the entropy one, it gives an idea of how much one will get different result using **Q** as a model when the real distribution is **P**.

χ^2-divergence This family of divergences includes two main subset:

$$D_{\chi^2_P}(\mathbf{P}||\mathbf{Q}) = \int_\Omega \frac{(q(x)-p(x))^2}{p(x)}dx \tag{4.24}$$

$$D_{\chi^2_N}(\mathbf{P}||\mathbf{Q}) = \int_\Omega \frac{(p(x)-q(x))^2}{q(x)}dx \tag{4.25}$$

Total-variation distance Let be **P** and **Q** two probability measures over $(\Omega, \mathcal{F})$, than the total variation distance between the two is define as

$$\delta(\mathbf{P},\mathbf{Q}) = \sup_{A\in\mathcal{F}} |\mathbf{P}(A) - \mathbf{Q}(A)| \tag{4.26}$$

thus it is the largest difference in probability of all the possible set $A \in \mathcal{F}$.

4.5 Offline model-based Reinforcement Learning

In Model-based Offline Reinforcement Learning (MB-ORL), the agent learns a model of the environment dynamics from a dataset of past interactions. The learned model can then be used to simulate future interactions, and the agent can use it to optimize its policy. Unlike online RL, where the agent interacts with the environment to collect data and updates its policy accordingly, MB-ORL utilizes a dataset of past interactions to learn a policy. This approach allows the agent to learn from past interactions without the need for further data collection, making it useful in scenarios where data collection is costly or difficult. Additionally, having a model of the environment can provide the agent with a deeper understanding of the problem and the ability to plan and reason about the effects of different actions. However, learning an accurate model of the environment can be challenging, particularly if the dataset is limited or the environment is complex. Additionally, the agent may need to balance the exploration-exploitation trade-off when learning the model and the policy. Overall, MB-ORL offers the ability to learn from past interactions without the need for further data collection and provides the ability to plan and reason about the environment. With the advancements in deep learning, it is possible to learn more accurate models and better policies using offline data. In Model-based Offline Reinforcement Learning, one important factor to consider is distributional shift, which refers to the difference between the distribution of states and actions encountered during training and during deployment. This can happen when the agent is deployed in a different environment, or when the environment changes over time. Distributional shift can affect the performance of MB-ORL models in several ways. One issue is that the model learned from the offline dataset may not accurately represent the dynamics of the new environment, resulting in poor performance. Additionally, the policy learned from the offline dataset may not be optimal for the new environment, leading to suboptimal performance. Another issue is that the offline dataset may not include samples from all possible states and actions, leading to a bias in the learned model and policy. This can cause the agent to perform poorly in states and actions that were not represented in the offline dataset. To address distributional shift, several methods have been proposed such as domain adaptation, domain randomization and meta-learning. Domain adaptation methods aim to adjust the learned model and policy to the new environment by fine-tuning on a small amount of online data. Domain randomization methods aim to make the learned model and policy more robust to different environments by training on a variety of simulated environments. Meta-learning methods aim to learn a policy that can adapt quickly to new environments by learning from a variety of tasks. Overall, distributional shift is an important consideration in MB-ORL and several methods have been proposed to address it. However, it is still an open research problem and more work is needed to develop robust and efficient methods for dealing with distributional shift in MB-ORL.

Chapter 5

Offline Reinforcement Learning algorithms

After having introduced ORL in its general settings, in this chapter some of the most used yet powerful algorithms to tackle this problem will be explained. The one described below are the ones that will be used in the experiments, which is the core of this thesis.

5.1 BCQ

In Fujimoto, Meger, et al. 2019 one of the first try to address the Offline Reinforcement Learning problem has been proposed Batch-Constrained deep Q-learning (BCQ), by using a variational autoencoder the algorithm is able to "create" new possible actions. Here a brief explanation of the algorithm will be done by focussing on pros and cons of this approach.

5.1.1 Algorithm overview

For a given state, BCQ generates plausible candidate actions with high similarity to the batch, and then selects the highest valued action through a learned Q-network. Than the algorithm bias this value estimate to penalize rare, or unseen, states through a modification to Clipped Double Q-learning.

VAE

To let the algorithm create new unseen state-actions pair the writers propose the use of a conditional variational autoencoder (VAE) which was first introduced in Kingma and Welling 2013. The way it works is through the modelling of the distribution by transforming an underlying latent space by using an encoder $E(s,a)$ and than it recovers new actions using a decoder $D(s)$. This two elements are trained one alongside the other through the following loss function:

$$\begin{aligned}&\hat{a} \sim D(s), \qquad \mu, \sigma \sim E(s,a),\\&L_{vae}(\omega) = \mathbb{E}_{a\sim\mathcal{D}, \hat{a}\sim vae(s)}((a-\hat{a})^2 + D_{\mathcal{KL}}(\mathcal{N}(\mu,\sigma)|\mathcal{N}(0,1)).\end{aligned} \tag{5.1}$$

Where in one side it is made it learning to decode actions similar to the one from the dataset and in the other it is forced to have a specific distribution in the latent space.

Q-values

As in many other cases the Q-values learning is done by four different neural networks, two are the network which are actually learning by "competing" one against the other and two are the targets that are slowly update step by step. The update of such neural network is done in the following way

$$\begin{aligned}&a_j \sim D(s)\\L_Q(\theta) =&\mathbb{E}_{(s,a)\sim\mathcal{D}}\Big[\Big(Q(s,a) - \max_{a_j}(r + \gamma \max(\lambda \min_{i=1,2} Q^i_{\hat{\theta}_i}(s,a_j)\\&+ (1-\lambda)\max_{i=1,2} Q^i_{\hat{\theta}_i}(s,a_j)\Big)\Big]\\&\hat{\theta}_i = \alpha\hat{\theta}_i + (1-\alpha)\theta_i\end{aligned} \tag{5.2}$$

Noise

The perturbation model it is used to increase the diversity of seen actions, allowing the algorithm to evaluate more possibilities when it has to decide which action should be taken. Such noise can be modelled as a normal distribution where the parameters are learned through the following loss

$$L_\Lambda(\xi) = \mathbb{E}_{\hat{a}\sim vae(s)}\left[-Q_\theta(s, \hat{a} + \Lambda_\xi(s,\hat{a})\right] \tag{5.3}$$

Policy

Here the policy is actually taken as a real argmax over the possible actions, indeed it is not modelled as a neural network, but instead each action is chosen as

$$\pi(s) = arg \max_{a_i\sim vae(s)}\left[Q_\theta(s, a_i + \Lambda_\xi(s,a_i)\right] \tag{5.4}$$

Algorithm 7 Batch-constrained deep Q-learning (BCQ)

```
1: Initialize Q-function parameters $\theta_1$, $\theta_2$, , target Q-function parameters $\hat{\theta}_1$,$th\hat{e}ta_2$,
   initialize the parameter $\omega$ of the VAE, the parameter $\xi$ of the perturbation
   model and offline data replay buffer $\mathcal{D}$
2: for each VAE preliminary update step do
3:     sample a batch $\mathcal{B} = \{(s, s', a, R)\} \sim \mathcal{D}$
4:     $\hat{a} \sim D_\omega(s)$, $\mu, \sigma \sim E_\omega(s, a)$
5:     compute the loss $L_{vae}(\omega)$
6:     update the parameters $\omega \leftarrow \omega + \lambda_\omega \nabla_\omega L(\omega)$
7: end for
8: for each gradient step do
9:     sample a batch $\mathcal{B} = \{(s, s', a, R)\} \sim \mathcal{D}$
10:    $\hat{a} \sim D_\omega(s)$, $\mu, \sigma \sim E_\omega(s, a)$
11:    compute the loss $L_{vae}(\omega)$
12:    update the parameters $\omega \leftarrow \omega + \lambda_\omega \nabla_\omega L(\omega)$
13:    sample n actions: $\{a_i \sim G_\omega(s')\}_{i=1}^n$
14:    perturb the actions $\{a_i = a_i + \Lambda_x i(s', a_i)\}_{i=1}^n$
15:    set $y = r + \gamma \max_{a_i}(\lambda \min_{j=1,2} Q^j_{\hat{\theta}_j}(s, a_i) + (1 - \lambda) \max_{j=1,2} Q^j_{\hat{\theta}_j}(s, a_i))$
16:    compute the loss $L_Q(\theta)$
17:    update the parameters $\theta \leftarrow \theta + \lambda_\theta \nabla_\theta L(\theta)$
18:    compute the loss $L_\Lambda(\xi)$
19:    update the parameters $\xi \leftarrow \xi + \lambda_\xi \nabla_\xi L(\xi)$
20:    soft update the targets parameters: $\hat{\theta}_i = \alpha\hat{\theta}_i + (1 - \alpha)\theta_i$
21: end for
```

5.2 IQL

In this section the state of the art in ORL will be discussed, it have been introduced by Kostrikov et al. 2021. This new approach to ORL is based on the idea that *in-distribution* constraint used before might not be sufficient to avoid extrapolation errors, instead they tried to learn the optimal policy by *in-sample* learning without querying out of distribution actions. They did so by approximating an upper expectile of the distribution over values with respect to the distriution of the dataset action. The main contribution of the article is to introduce *implicit Q-learning* (IQL), a new method able to perform multi-step dynamic programing while not querying for out of distribution actions.

5.2.1 Mathematical preliminaries

Expectile regression

One of the main point of this new method is to implement the expectile regression. Expectile regression has been widely used in statistic and econometric to estimate statistics of random variable.

Definition 4. *The* $\tau \in (0{,}1)$ *expectile of a random variable* $\mathbf{X}$ *the solution to the following problem*

$$arg \min_{m_\tau} \mathbb{E}^{\tau}_{x \sim X}[L^{\tau}_2(x - m_\tau)] \tag{5.5}$$

where

$$L^{\tau}_2(x - m_\tau) = |\tau - \mathbf{1}(x - m_\tau < 0)|(x - m_\tau)^2 \tag{5.6}$$

This is an asymmetric loss function, indeed with $\tau < 0.5$ it downgrade the value of x values greater than m_τ while weights more larger values. It is worth noting how this loss function can be optimize by gradient decend.

Value function with expectile regression

Expectile regression is a powerful tool to estimate statistics of a random variable. It is possible to incorporate it to learn and estimate an upper expectile of the temporal-difference target. While the squared form of temporal difference error is given by

$$L(\theta) = \mathbb{E}_{(s,a,s') \sim D}\left[\left(r(a,s) + \gamma \max_{a'} Q_{\hat{\theta}}(s', a') - Q_\theta(s,a)\right)^2\right], \tag{5.7}$$

implementing it as a an expectile regression gives

$$L(\theta) = \mathbb{E}_{(s,a,s',a') \sim D}\left[L^{\tau}_2(r(s,a) - \gamma Q_{\hat{\theta}}(s', a') - Q_\theta(s,a)\right]. \tag{5.8}$$

In the article they also notice how this formulation take a drawback with it self. In some ORL settings the new state s' is not always fixed and it is given by a probability distribution $s' \sim p(\cdot|s,a)$. Thus it is possible that some state-actions that give good values are just lucky and that is not incorporate into the state itself. To address this problem their solution is to train another value function which approximate the expecile whit respect only to the action distribution. The resultin loss is

$$L_V(\psi) = \mathbb{E}_{(s,a)\sim\mathbf{D}}\left[L_2^\tau(Q_{\hat{\theta}}(s,a) - V_\psi(s))\right]. \tag{5.9}$$

This target value function is than used to train the Q-value function with a normal MSE loss, having

$$L_Q(\theta) = \mathbb{E}_{(s,a,s')\sim\mathbf{D}}\left[\left(r(s,a) + \gamma V(s') - Q_\theta(s,a)\right)^2\right]. \tag{5.10}$$

Policy extrapolation

The process followed does not give a explicit formulation of the policy. Ispiered by Peters and Schaal 2007, Q. Wang et al. 2018 Peng et al. 2019,Nair et al. 2021 where they proposed to use advantage wheight regression.

$$L_\pi(\phi) = \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[exp\left(\beta Q_{\hat{\theta}}(s,a) - V_\psi(s)\right)\log(\pi_\phi(a|s))\right] \tag{5.11}$$

Algorithm 8 Implicit Q-Learning (IQL)

```
1: Initialize $\theta$, $\hat{\theta}$, $\psi$ and $\pi$
2: Initialize replay buffer $\mathcal{D}$, discount factor $\gamma$, learning rate $\alpha$ and $\lambda_\psi$,$\lambda_\theta$,$\lambda_\pi$
3: for each gradient step do
4:     sample a batch $\{s, s', a, R\} \sim \mathcal{D}$
5:     compute the value loss $L(\psi)$
6:     update the value weights $\psi \leftarrow \psi + \lambda_\psi \nabla_\psi L(\psi)$
7:     compute the Q-loss $L(\theta)$
8:     update the Q-value weights $\theta \leftarrow \theta + \lambda_\theta \nabla_\theta L(\theta)$
9:     soft-update the Q-value target weights $\hat{\theta} \leftarrow (1-\alpha)\hat{\theta} + \alpha\theta$
10: end for
11: for each policy-update do
12:     compute the policy loss $L(\pi)$
13:     update the policy weights $\pi \leftarrow \pi + \lambda_\pi \nabla_\pi L(\pi)$
14: end for
```

5.3 CQL

CQL stands for "Conservative Q-Learning. The "conservative" in the name refers to the way the algorithm updates the Q-values. In traditional Q-learning, the Q-values are updated based on the maximum expected future reward, which can lead to over-estimation of the Q-values. CQL addresses this issue by using a more conservative update rule that takes into account the uncertainty in the Q-value estimates. By being more cautious in its updates, CQL is less likely to over-estimate the Q-values and can improve the stability and performance of the algorithm. CQL algorithms refer to both Q-learning and actor-critic methods. In this section this approach introduced by Kumar et al. 2020 will be discussed in details, all the further analysis of the theorem and their proof can be found in the article cited.

5.3.1 Mathematical preliminaries

The empirical Bellman operator

Empirical Bellman operator is an estimation of the true Bellman operator based on the samples from the underlying process, it uses the sample transitions and rewards to estimate the expected value of the next state. For example, given a dataset of transitions $D = \{(s_i, a_i, r_i, s'_i)\}$ for i = 1...N, the empirical Bellman operator can be defined as:

$$\hat{B}^{\pi}(\hat{V})(s) = \frac{1}{N}\sum_{i=1}^{N}(r_i + \gamma * V(s'_i)) \quad \text{for} \quad (s_i, a_i, r_i, s'_i) \in D \quad \text{with} \quad s_i = s$$

Where $\hat{B}^{\pi}(\hat{V})$ is the estimation of the true Bellman operator $B^{\pi}V$ by using the sample transitions (s_i, a_i, r_i, s'_i) and the current value function V.

Algorithm 9 Conservative Q-Learning (CQL)

```
1: Initialize $\theta$
2: for each gradient step do
3:     sample a batch $\{s, s', a, R\} \sim \mathcal{D}$
4:     compute $CQL(\mathcal{R})$
5:     update the Q-value weights $\theta \leftarrow \theta + \lambda_\theta \nabla_\theta CQL(\mathcal{R})$
6: end for
```

5.3.2 Conservative off-policy evaluation

As in many offline RL algorithm this method is based on being able to estimate policy π which is different from the policy behaviour π_β. The aim of this approach is to not overestimate the Q-value on such policies, this problem is addressed by learning a lower bound of Q. This is done by minimize Q-values along side the Bellman error. Since this algorithm naturally queries out-of-distribution state-action the following assumption on such unseen state-action is that they still belongs to a marginal distribution $\mu(s,a) = d^{\pi}(s)\mu(a|s)$, this means that all the queried states belongs to the distribution of the dataset D (i.e. all the states where the update of the Q-values will be made belong to the dataset). Having this in mind the following update rule does exactly what has been discussed

$$\hat{Q}^{k+1} \leftarrow arg \min_{Q} \alpha \mathbf{E}_{s\sim D\, a\sim\mu(a|s)}\left[Q(s,a)\right] + \frac{1}{2}\mathbf{E}_{(s,a)\sim D}\left[\left(Q(s,a) - \hat{B}^{\pi}\hat{Q}(s,a)\right)^2\right].$$

In the paper they prove how this update lower-bounds the Q-values for each (s,a). This method can be too conservative in terms of Q-values, being our goal to estimate $V^{\pi}(s)$ one can argue that such approach can be misleading when all the information are given. Knowing this they propose the new less conservative update rule

$$\begin{aligned}\hat{Q}^{k+1} \leftarrow & arg \min_{Q} \alpha \mathbf{E}_{s\sim D\, a\sim\mu(a|s)}\left[Q(s,a) - \mathbf{E}_{s\sim D, a\sim\pi_{\beta}}[Q(s,a)]\right] \\ & + \frac{1}{2}\mathbf{E}_{(s,a)\sim D}\left[\left(Q(s,a) - \hat{B}^{\pi}\hat{Q}(s,a)\right)^2\right]. \qquad (5.12)\end{aligned}$$

Adding the new term they, in practical terms, make a maximization over the behavior policy π_{β}. This leads this new update to not be a point wise lower bound on the $V^{\pi}(s)$, but this does holds in terms of expected value (i.e. $\mathbf{E}_{\pi(a|s)}(\hat{Q}(s,a)) \leq V^{\pi}(s)$).Indeed it is easy to notice how action that follows the behavior policy can be over-estimate.

5.3.3 Conservative Q-learning for ORL

The approach described in the previous section it is a general way to estimate a lower bound for Q-values under some distribution μ. Assume to want to estimate the value function under some policy π, by just solving the equation 5.12 with $\mu = \pi$ one is able to estimate a lower bound for the real value function and than apply a simple policy improvement step. However this approach is too expensive in computational terms. Another approach is to take $\mu(a|s)$ as the approximation of the current best policy given by the Q-values. This is formally given by the following family of optimization problems:

$$\begin{aligned}CQL(\mathcal{R}) = & \min_{Q}\max_{\pi} \alpha \left(\mathbf{E}_{s\sim D,\, a\sim\mu(a|s)}\left[Q(s,a)\right] - \mathbf{E}_{s\sim D, a\sim\pi_{\beta}}[Q(s,a)]\right] \\ & + \frac{1}{2}\mathbf{E}_{(s,a)\sim D}\left[\left(Q(s,a) - \hat{B}^{\pi}\hat{Q}(s,a)\right)^2\right] + \mathcal{R}(\mu). \qquad (5.13)\end{aligned}$$

$\mathcal{R}(\mu)$ can be chosen accordingly to the problem for example

- $\mathcal{R}(\pi) = -D_{KL}(\pi, \rho)$ where ρ is a prior distribution that can be equal to the behaviour policy one.
- $\mathcal{R}(\pi) = \mathbf{E}_{s\sim D}\left[\mathcal{H}(\pi(\cdot|s))\right]$ where $\mathcal{H}(\cdot)$ stands for the entropy.

The following is the form of the loss function in terms of neural network parameters.

$$\begin{aligned} L_Q(\theta) =& \alpha(\mathbb{E}_{s\sim\mathcal{D},a\sim\pi}[Q_\theta(s,a)] - \mathbb{E}_{(s,a)\sim\mathcal{D}}[Q_\theta(s,a)]) \\ &+ \frac{1}{2}\mathbb{E}_{(s,a,r,a',s')\sim\mathcal{D}}[(Q_\theta(s,a) - r(s,a) + \gamma Q_{\hat{\theta}}(s',a'))^2] \end{aligned} \tag{5.14}$$

The way to extrapolate the policy from this Q-values proposed in Kumar et al. 2020 is the standard which takes as action the one the maximize the Q-values, however for implementation simplicity the one used is

$$L_\pi(\phi) = \mathbb{E}_{s\sim\mathcal{D},a\sim\pi}\left[\alpha\log(\pi(a|s)) - Q_{\hat{\theta}}(s,a)\right] \tag{5.15}$$

5.3.4 CQL conclusion

In conclusion, CQL is a valuable method for offline reinforcement learning. It utilizes a conservative update rule for the Q-values which results in improved sample efficiency compared to standard Q-learning. CQL has been successfully applied to a variety of tasks and has shown promising results in numerous studies. Its ability to learn from offline data makes it a useful tool for real-world applications where data collection is costly or difficult. Overall, CQL is a valuable addition to the offline RL methods

5.4 SAC-N

In An et al. 2021 two of the most powerful methods to address the offline reinforcement learning problem have been released. This two algorithm share the same approach, while the one described in this section is more naive the one that will be discussed in the next one will be more complex.

5.4.1 N-clipping Q-learning

As discussed in section 3.7.2 one of the methods to avoid overestimation of Q-values in the online setting is to have more than one approximators to evaluate this value. One of the main examples where this has been successfully implemented is the Soft-Actor-Critic algorithm, where two neural network approximators are used. The main idea of this new new algorithm is to take this and exaggerate it to have a better under estimation of the value, indeed how will be shown below this would be a good estimation of a lower bound of the value. Now let us take into account a normal SAC method as described in 3.12.2 where instead of having only two Q-value approximators N are taken. The algorithm just change in the critic and actor update in the following way:

Actor update : for what concern the actor the change is in the loss function as below

$$L(\phi) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[\mathbb{E}_{a_t \sim \pi(\cdot|s_t)} \left[\alpha \log(\pi(a_t|s_t, \phi)) - \min_{i=1,\ldots,N} Q_i(s_t, a_t|\theta_i) \right] \right] \tag{5.16}$$

Critic update : as one can expect the critic update change in the same fashion, indeed for each critic network the loss becomes

$$\begin{aligned} L(\theta_i) =& \mathbb{E}_{(s_t,a_t,r_t,s_{t+1}) \sim \mathcal{D}} \Big[\frac{1}{2} \Big(Q_i(s_t, a_t|\theta_i) - \Big(r_t + \\ & \gamma \mathbb{E}_{a_{t+1} \sim \pi(\cdot|s_{t+1})} \left[\min_{j=1,\ldots,N} (Q_j(s_{t+1}, a_{t+1}|\theta_j)) \right] \Big) \Big)^2 \Big] \end{aligned} \tag{5.17}$$

5.4.2 Mathematical justification

The clipped Q-learning algorithm, which chooses the worst-case Q-value instead to compute the pessimistic estimate, can also be interpreted as utilizing the Lower Confidence Bound of the Q-value predictions. Without loosing generality one can assume that the Q-value $Q(s, a)$ follows a Gaussian distribution with mean $m(s, a)$ and standard deviation $\sigma(s, a)$. Also, let $Q_i(s, a)_{i=1}^N$ be realizations of $Q(s, a)$. Then, we can approximate the expected minimum of samples as

$$\mathbb{E} \left[Q_i(s, a)_{i=1}^N \right] = m(s, a) - \Phi^{-1} \left(\frac{N - \frac{\pi}{8}}{N - \frac{\pi}{4} + 1} \right) \sigma(s, a) \tag{5.18}$$

where Φ^{-1} stands for the inverse of the cumulative distribution function of a normal gaussian.

5.5 Ensamble Diversified Actor-Critic

N-SAC can require a very high amount of networks to work properly, indeed for simple tasks with a very good dataset the order of approximators needed is 500. To address this problem in An et al. 2021 a study on what makes the difference between a high performance results and a low one has been made. Here the idea, the proof and the following algorithm will be discussed.

5.5.1 The gradient alignment problem

In the cited article the authors have empirically found how the performance of SAC-N is negatively correlated with the degree to which the input gradients of Q-functions $\nabla_a Q_{\theta_i}(s,a)$ are aligned, which increases with N. This observation imply that the performance of the learned policy degrades significantly when the Q-functions share a similar local structure. Here how they addressed and explained this problem mathematically will be shown. Let be $\nabla_a Q_{\theta_i}(s,a)$ the normalized gradient of the i-th Q function. Assuming that the gradients are aligned one can find a direction w such that the variance of the out of distribution actions along it is minimum. It is also possible to assume that for actions that are on distribution the value $Q_{\theta_i}(s,a)$ is actually equivalent for each i i.e. $Q_{\theta_i}(s,a) = Q(s,a)$. One can obtain the following variance by exploiting the first order Taylor approximation:

$$\begin{aligned}\mathbf{Var}(Q_{\theta_i}(s,a+kw)) &\approx \mathbf{Var}((Q_{\theta_i}(s,a) + k\langle w, \nabla_a Q_{\theta_i}(s,a)\rangle)\\ &= k^2\mathbf{Var}\langle w, \nabla_a Q_{\theta_i}(s,a)\rangle)\\ &= k^2 w^{\mathbf{T}}\mathbf{Var}(\nabla_a Q_{\theta_i}(s,a))w. \end{aligned} \tag{5.19}$$

Computing the total variance of the matrix $\mathbf{Var}(\nabla_a Q_{\theta_i}(s,a))$, (i.e. the trace of it, which is equal to the sum of the eigenvalues), one can obtain

$$\begin{aligned}\mathbf{Tr}\left(\mathbf{Var}(\nabla_a Q_{\theta_i}(s,a))\right) &= \mathbf{Tr}\left(\frac{1}{N}\sum_{i=1}^{N}(q_i-\bar{q})(q_i-\bar{q})^{\mathbf{T}}\right)\\ &= \frac{1}{N}\sum_{i=1}^{N}\mathbf{Tr}\left((q_i-\bar{q})(q_i-\bar{q})^{\mathbf{T}}\right)\\ &= \frac{1}{N}\sum_{i=1}^{N}\left((q_i-\bar{q})^{\mathbf{T}}(q_i-\bar{q})\right)\\ &= \frac{1}{N}\sum_{i=1}^{N}\left(q_i^{\mathbf{T}}q_i - 2q_i^{\mathbf{T}}\bar{q} + \bar{q}^{\mathbf{T}}\bar{q}\right), \end{aligned} \tag{5.20}$$

where $q_i = \nabla_a Q_{\theta_i}(s,a)$ Remembering that the gradients are normalized the following holds

$$\begin{aligned}\mathbf{Tr}\left(\mathbf{Var}(\nabla_a Q_{\theta_i}(s,a))\right) &= 1 - 2\left(\frac{1}{N}\sum_{i=1}^{N} q_i\right)^{\mathbf{T}}\bar{q} + ||\bar{q}||_2^2\\ &= 1 - ||\bar{q}||_2^2. \end{aligned} \tag{5.21}$$

It is possible to upper-bound the smallest eigenvalue λ_{min} by some constant ϵ. To do so first let us take into consideration the following

$$\begin{aligned}
||\bar{q}||_2^2 &= \frac{1}{N^2} \sum_{1 \leq i,j \geq N}^{N} \langle q_i, q_j \rangle \\
&= \frac{1}{N^2} \left(\sum_{i=1}^{N} \langle q_i, q_i \rangle + \sum_{1 \leq i \neq j \geq N} \langle q_i, q_j \rangle \right) \\
&\geq \frac{1}{N^2} \left(N + N(N-1)(1-\epsilon) \right) \\
&= 1 - \frac{N-1}{N} \epsilon.
\end{aligned} \tag{5.22}$$

Thus an upper-bound of the total variance has been found,

$$\mathbf{Tr}\left(\mathbf{Var}(\nabla_a Q_{\theta_i}(s,a))\right) \leq \frac{N-1}{N} \epsilon. \tag{5.23}$$

The trace of a matrix is also equal to the sum of the eigenvalues. It follows that

$$\begin{aligned}
\lambda_{min} &\leq \frac{1}{|\mathcal{A}|} \sum_{i=1}^{|\mathcal{A}|} \lambda_i \\
&=\leq \frac{1}{|\mathcal{A}|} \mathbf{Tr}\left(\mathbf{Var}(\nabla_a Q_{\theta_i}(s,a))\right) \\
&\leq \frac{1}{|\mathcal{A}|} \frac{N-1}{N} \epsilon,
\end{aligned} \tag{5.24}$$

where λ_i are the eigenvalue of the **Var** matrix, λ_{min} is the minimum of them and $|\mathcal{A}|$ is the cardinality of the actions. Now assume that $w = w_{min}$ is the eigenvector associated to λ_{min}, taking into account 5.19 one can obtain the following relation

$$Thiscanbeseenasalossfunctionandwithasmallmodificationthefinaland\mathbf{Var}(Q_{\theta_i}(s, a + kw_{min})$$

$$\tag{5.25}$$

Therefore, since the goal of this study is to maximize the variance the problem can be reformulate as a maximization over θ_i of the following

$$\mathbf{E}_{s,a\sim D}\left[\lambda_m in \mathbf{Var}(\nabla_a Q_{\theta_i}(s,a))\right]. \tag{5.26}$$

However the computational cost of this would be very high, being it iterative. Thanks to 5.21 this can be reformulate as the minimization of

$$\mathbf{E}_{s,a\sim D}\left[\left\langle \frac{1}{N}\sum_{i=1}^{N}\nabla_a Q_{\theta_i}(s,a), \frac{1}{N}\sum_{j=1}^{N}\nabla_a Q_{\theta_j}(s,a)\right\rangle\right]. \tag{5.27}$$

This can be seen as a loss function and with a small modification the final and used form is

$$L_{Q_i}(\theta_i) = \mathbb{E}_{s,a\sim D}\left[(Q_{\theta_i}(s,a) - y(r,s'))^2 + \frac{\eta}{N-1}\sum_{1\leq i,j\leq N}\left\langle \nabla_a Q_{\theta_i}(s,a), \nabla_a Q_{\theta_j}(s,a)\right\rangle\right] \tag{5.28}$$

In the end this can be interpreted as measuring the pairwise alignment of the gradients using cosine similarity, which they denote as the Ensemble Similarity (ES) metric. To update the policy the standard method from SAC is implemented

$$L_\pi(\phi) = \mathbb{E}_{s\sim\mathcal{D},a\sim\pi}\left[\alpha\log\pi(a|s) - \min_{i=1,\ldots,N} Q_{\theta_i}(s,a)\right] \tag{5.29}$$

$$L_Q(\theta) = \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\Big(Q(s,a) - \max_{a_j}(r + \gamma(\lambda\min_{i=1,2} Q^i_{\hat{\theta}}(s,a_j) + (1-\lambda)\max_{i=1,2} Q^i_{\hat{\theta}}(s,a_j)\Big)\right] \tag{5.30}$$

$$L_{vae}(\omega) = \mathbb{E}_{a\sim\mathcal{D},\hat{a}\sim vae(s)}((a-\hat{a})^2 + D_{\mathcal{KL}}(\mathcal{N}(\mu,\sigma)|\mathcal{N}(0,1)) \tag{5.31}$$

$$\pi : \mathcal{S} \to \mathcal{A} \tag{5.32}$$

Algorithm 10 Ensamble Diversified Actor-Critic (EDAC)

```
 1: Initialize policy parameters $\theta$, Q-function parameters $\{\phi\}_i^N$, target Q-function
    parameters $\{\phi'\}_i^N$ and offline data replay buffer $\mathcal{D}$
 2: for each gradient step do
 3:     sample a batch $\mathcal{B} = \{(s, s', a, R)\} \sim \mathcal{D}$
 4:     Compute target Q-values:
 5:
        $$y(r, s') = r + \gamma(\min_{i=1,\dots,N} Q_{\{\phi'\}_i}(s', a') - \beta \log \pi_\theta(a'|s')), \quad a' \sim \pi_\theta(\cdot|s')$$
 6:     Update each Q-function with gradient descent using
 7:
        $$\nabla_{\phi_i} \frac{1}{|\mathcal{B}|} \sum_{(s,s',a,R)\in\mathcal{B}} \left( (Q_{\phi_i}(s, a) - y(r, s'))^2 + \frac{\eta}{N-1} \sum_{1\leq i\neq j\leq N} ES_{\phi_i,\phi_j}(s, a) \right)$$
 8:     Update the policy using:
 9:
        $$\nabla_\theta \frac{1}{|\mathcal{B}|} \sum_{s\in\mathcal{B}} \min_{i=1,\dots,N} Q_{\phi_i}(s, \hat{a}_\theta) - \beta \log \pi_\theta(\hat{a}\theta|s)$$
    where
        $$\hat{a}_\theta \sim \pi_\theta(s)$$
10:     Soft-update the Q-value weights: $\phi'_i \leftarrow (1 - \alpha)\phi' + \alpha\phi$
11: end for
```

Chapter 6

Experiment setup

After online and offline reinforcement learning methods has been discussed in the previous chapters is now time to introduce the environment where those will be applied. A more in detail report of the environment can be found Martini et al. 2023.

6.1 The software

In this section the software utilized in the experiments will be explain in all its details.

6.1.1 Gazebo environment

As highlighted before real experiment can be very expensive in both terms of time and money, thus it is worth having a way to being able to simulate real scenarios as good as possible. When robots are involved into the research GAZEBO is one of the more suitable and utilized tool. GAZEBO is an open-source 3D robot simulator, that allows the user to create complex real scenarios. It is a physics based environment which can model and simulate complex robot dynamics, incorporate sensor data, and emulate real-world challenges. Its realism is very well suited for both RL and ORL algorithms and it is an efficient way to learn policies which than can be incorporate and trained into a real robot.
Thanks to the incorporation of python and ROS (Robot Operating System) it is possible to interact with the GAZEBO environment and being able to train and test RL and ORL algorithms.

6.1.2 Robot Operating System

The Robot Operating System (ROS) is a powerful and flexible framework that has become one of the most used tools in robotics research and development. It enables the user to create and manipulate robots, being it modular it makes very simple to project complex machines. The core of this framework, which it relies on, are nodes and topics.

- **Nodes**: Nodes are fundamental computational units in ROS. They are individual software modules or processes that perform specific tasks within a robotic system. Nodes can be thought of as small programs that can be written in languages like Python or C++. Nodes are designed to be very specific and modular. Those are typically responsible for tasks like sensor data processing, motor control, localization and mapping.

- **Topics**: Topics are channels which allows the communication between ROS nodes, those channels can be used by the nodes to receive and send signal or information. Data sent on a topic is structured as messages. ROS provides a wide range of message types for various data types, such as images, sensor readings, commands, and custom data structures.

- **Node-Topic interaction**: Nodes interact with each other by publishing and subscribing to topics. A node that publishes data to a topic is called a "publisher," while a node that receives data from a topic is called a "subscriber." Multiple nodes can publish and subscribe to the same topic, allowing for distributed and parallel processing of data. This publish-subscribe model promotes modularity and flexibility, as nodes can be added, removed, or replaced without affecting the entire system's functionality.

In the environment used in this thesis all the nodes are written in Python.

6.1.3 TensorFlow

TensorFlow is a versatile and widely used machine learning framework developed by Google. It empowers developers and researchers to build and train machine learning models for various tasks, such as image recognition, natural language processing and as in this case reinforcement learning or offline reinforcement learning. Tensors are the fundamental data structures in TensorFlow. They are multi-dimensional arrays that can hold numerical data of various types (e.g., floats, integers). Tensors can be constants (immutable) or variables (mutable), and they serve as the primary data carriers within TensorFlow. TensorFlow operates using a directed graph called a computational graph. In this graph, nodes represent operations, and edges represent the flow of data (tensors) between operations. The computational graph is created when you define operations in TensorFlow, and it is executed when you run a TensorFlow session. It differs from other framework such as PyTorch in the way the computational graph is created. While in TensorFlow it is necessary to specify when and where create the graph in PyTorch this is done automatically. The choice of one over the other is a matter of preferences, in this case TensorFlow has been chosen due to previous work by the Pic4Ser department.

6.2 Reinforcement learning elements

In this section the environment utilized for this study will be explained. The analysis of it will be made based on RL elements as described in 3.1.

6.2.1 The environment

The environment studied is an in-door setting, where a robot can move and reach goals. Being it in-door allows it to be easier to study, it is not affected by weather or night-day cycle which can interfere with the sensor utilized. In the other hand this setting makes it difficult for the sensors to detect the objects due to their size. In this case the setting is based on a fictional house however due to the generality of the algorithm used it is easily changeable to other kind of in-door structure such as factories or laboratories.

6.2.2 The agent

The agent as written above is a square robot with four wheels. The size of the machine is 0.508 m of length and 0.430 m for what concern the width. Such type of robot can be utilized in various fields such as production, work assistance and in general can be seen as a self-driving car.

6.2.3 The goal

As described above the aim of this learning process is to find a better way to teach to an AI installed on a mobile robot how to reach a specific position. To do so the goal in each iteration is set to be a point of the environment. This is passed to the algorithm in the form of a relative position to the robot in terms of distance and angle,

$$G = (d, \alpha). \tag{6.1}$$

In the other hand it would be too restrictive asking to have the center of the robot be exactly aligned with the goal, thus for the algorithm the goal would be reached if the machine reaches position near to it, i.e. the robot "collides" with a circumference of radius 0.4 m centered in the goal.

6.2.4 The states

In this setting the states are taken as tensor of the form [38,1], while the first two input are computed as the distance and the angle (in odometry terms) between the robot and the goal, the last 36 are taken by lidars. Here both odometry and lidar will be explained.

Odometry

Odometry is crucial in the navigation of autonomous robots, including autonomous cars, drones, and mobile robots used in logistics, agriculture, and healthcare. It is the science of precisely measuring the movement of a robot as it traverses its environment. Odometry serves as a fundamental technique for estimating a robot's position in a known or unknown environment. By continuously tracking wheel or sensor movements, it accumulates data to determine the robot's relative position with respect to its starting point. Accurate odometry data is essential for motion control, enabling robots to execute tasks such as path following, obstacle avoidance, and precise maneuvers. It is worth noting how odometry is subjected by to error accumulation over time, mall inaccuracies in measurements can lead to significant positioning errors. Odometry can be sensitive to changes in the environment, such as uneven terrain or wheel-surface interactions. In the case of differential drive robots, where wheel movements are tracked, odometry equations can be expressed as follows:

d_l and d_r : Distance traveled by the left and right wheels, respectively.
Δs : Linear distance traveled by the robot's center between time steps.
$\Delta\theta$: Change in orientation.

These quantities are related by:

$$\Delta s = \frac{d_l + d_r}{2}, \qquad \Delta\theta = \frac{d_r - d_l}{L}$$

where L represents the distance between the robot's two wheels.
Using the incremental motion model, the robot's new pose (x', y', θ') can be updated from the previous pose (x, y, θ) as follows:

$$x' = x + \Delta s \cdot \cos\left(\theta + \frac{\Delta\theta}{2}\right)$$
$$y' = y + \Delta s \cdot \sin\left(\theta + \frac{\Delta\theta}{2}\right)$$
$$\theta' = \theta + \Delta\theta$$

LiDAR

LiDAR, which stands for Light Detection and Ranging, is a remote sensing technology that utilizes laser light to measure distances and create detailed 3D maps of objects and environments. LiDAR operates on the principle of time-of-flight, where it measures the time it takes for a laser pulse to travel to an object and back.

$$t = \frac{2d}{c} \tag{6.2}$$

where d is the distance to the object and c is the speed of light. Using time-of-flight measurements, the distance (d) to an object can be calculated as:

$$d = \frac{ct}{2} \tag{6.3}$$

LiDAR systems emit laser pulses and measure the time it takes for these pulses to return after bouncing off objects. LiDAR technology finds applications in various fields, including:

1. **Topographic Mapping:** LiDAR is used to create high-resolution elevation models for applications in geology, forestry, and land-use planning.
2. **Autonomous Vehicles:** LiDAR is a crucial sensor for self-driving cars, helping them perceive their surroundings and navigate safely.
3. **Robotics:** LiDAR is integrated into robotic systems for mapping and navigation, making it possible for robots to navigate in complex environments.
4. **Environmental Monitoring:** LiDAR is used to monitor changes in forests, coastlines, and ecosystems.
5. **Archaeology and Cultural Heritage:** LiDAR helps archaeologists discover and map ancient ruins and cultural heritage sites hidden beneath vegetation.

LiDAR is a powerful remote sensing technology that relies on the measurement of laser pulse travel times to calculate distances to objects.

6.2.5 The reward

The reward which drives the whole learning process is very important in this kind of task. In principle the aim of the AI is to reach the goal, indeed a reward of 1000 is given every time the robot "collides" with it. In the other hand one of the main problem that this work is focus on is to reduce the coast of the learning, thus to drive the algorithm to avoid any collision a cost of 600 is subtracted each time the machine bumps into some obstacle. It has been observed that, even if in principle this should be enough, adding a reward to push the robot to reduce the distance from the goal leads to better performances. In the end the final reward can be written as

$$R = 30(d - d') - \mathbf{1}(\mathit{if\ collision})600 + \mathbf{1}(\mathit{if\ goal})1000. \tag{6.4}$$

Chapter 7

Experiments and results

In this chapter the core of this thesis will be discussed. After all the elements have been discussed it is now time to put all together. In particular the focus will be on the difficulties and results of applying offline reinforcement learning methods in real cases and how this could still be a powerful approach as a pre-trainer of online RL, indeed how can this be utilized to minimize both the learning step of an online RL method and the "cost" will be part of this study.

7.1 Dataset

As one can expect one of the main difficulties of the offline reinforcement learning is to get usable dataset, in the real world such thing can be done by hand (i.e. a human spend his time on getting the dataset), using less refine AI or by using old data stored over time. Such dataset can be very different, for example using an "expert" leads to have a very good data, in such case an imitation learning approach can be the way to go. In the other hand one of the main task offline reinforcement learning have to address is to extrapolate very good policy from every type of data. To do so different dataset have been stored using different approach. Here how they were taken and their characteristics will be explained.

7.1.1 Dataset from online reinforcement learning

This kind of dataset has been taken by saving all the transition taken by an online reinforcement learning method. One would expect this kind of dataset would be very well suited for the offline reinforcement learning, since the online algorithm has already learned from it. How we will see, even if in theory this should lead to a similar understanding of the task and the environment, this is not completely the case. How we have discussed in the chapter 3 leaving the algorithm explore what it thinks is the best approach is one of the key of a successful online reinforcement learning training. This dataset are composed by all the transitions a RL algorithm have taken to completely learn the task. To have more "good" and various results the training was not stopped when the algorithm reached convergence, but instead a fixed number of transitions was pre-set. It is important to notice how even RL algorithms have difficult times learning this kind of task, indeed even at convergence they were not able to perform all task within the same parameters, leading to the idea that such NN were not general enough for such job.

SAC

The most used as RL algorithm to take the dataset in this study was Soft Actor Critic which was already explained in 3.12.2. To have dataset that shows good and bad behaviours all the transitions where stored in the replay buffer, meaning that each try the algorithm has made is saved and can be used in the offline settings. To show and learn how different dataset leads to different results four different buffers were stored with transition from this algorithm each having different size and parameters. This will be shown in table REFERENZA TABELLA, where all the dataset are described. To have more idea of how also data from bad behaviour can be useful an expert dataset was taken. In this case only episodes where the goal was reached were stored. This will be called the "Expert dataset".

TD3

To avoid our results be too correlated to the soft-actor critic algorithm two more have been taken from TD3 algorithm, explained in 3.12.1.

Dynamic Window Approach (DWA)

This a state-of-the-art algorithm for mobile robot was first introduced in Fox et al. 1997, it will be further explain in A.1. Since this is not a learning algorithm it would not be possible to directly store all the transition, since most of them would follow an optimal policy. To avoid this the dataset has been taken introducing an "exploration" of action, following an $\epsilon-$greedy approach.

Name	Size	Method
Expert	3000000	SAC
SAC_500k	5000000	SAC
SAC_300k	3000000	SAC
TD3_300k	3000000	TD3
DWA_100k	1000000	DWA

Table 7.1: Description of different configurations.

7.2 Experiment setup

Here the evaluation and the comparison between offline methods will be made. All the algorithms where written in Python by using the Tensorflow package. They all share some components such as the Q-value network or the policy one. Here the specificity of the shared components will be explained.

Q-value network

The Q-value network has been made as a standard multilayer perceptron neural network, with two sets of hidden layers with a size of 256 each. After each step ReLu activation function was applied. It takes as input the concatenation over axis 0 of the state and the action and gives the extimated Q-value output.

Actor network

The actor network has been modeled as a Gaussian actor, again with a multilayer perceptron neural network with two sets of hidden layers with 256 neuron each, except for the last outputs a ReLu activation function was always applied at each step. It takes as input the state and gives as result a tensor of mean and one of the log of the standard deviation, which combined gives a Gaussian distribution over the actions, it is worth noting how in this setup each dimension of the action is taken as independent from the others. This is used as the distribution over the actions by clipping the value over the eligible ones.

Offline trainer

Each algorithm share the same training structure. That will be shown in the following algorithm.

7.2.1 Parameters

Each algorithm have its own parameters however some of them are shared

Algorithm 11 Offline trainer

```
 1: get $\epsilon$ small
 2: get $max\_steps$
 3: $not\_convergence \leftarrow True$
 4: while $not\_convergence$ & $steps < max\_steps$ do
 5:     sample a batch from the dataset of size $d$, $\{(s, s', a, R)\} \sim \mathcal{D}$
 6:     Do one step of training with the sample batch
 7:     $conv \leftarrow Convergence\_measure(algorithm)$
 8:     if $conv < \epsilon$ then
 9:         $not\_convergence \leftarrow False$
10:     end if
11:     $steps \leftarrow steps + 1$
12: end while
```

7.3 Results

In this section the obtained results will be discussed. First the performances over the different dataset of each offline algorithm will be shown dataset and than a comparison between how they perform over some tasks will be discussed and in the end some thoughts about the differences in performance between online and offline approaches will be analysed. All the results obtained below are achieved by trying every single combination of initial points and goal positions.

7.3.1 BCQ

Due to the use of a Variational Auto Encoder this algorithm is the most dataset dependent, indeed the fact that the possible actions taken by the agent are very similar to the one from the dataset lead to high variance in terms of performances. How one can see in 7.1 the best results are obtained with the *DWA* dataset. In the other cases the results show how this method struggles with this kind of tasks and it would not be suggested in the case of low quality dataset, where other kind of approaches are instead capable to find better policies.

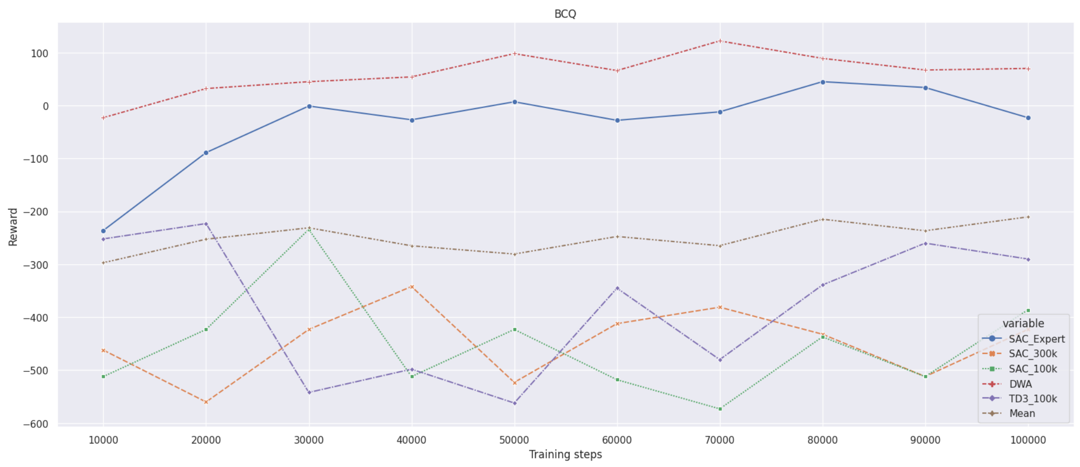

Figure 7.1: Reward during the training of BCQ algorithm

7.3.2 IQL

In figure 7.4 the performance of IQL algorithm over the different dataset are shown. It is very important to notice how the results are have differences over the different dataset, having the best result for *DWA*, however the trend of the training over such dataset suggest to reach very early an overfitting, such problem seems to scale with the dataset size, on the other hand other kind of training do not suffer from this problem making it difficult to analyse properly this behaviour. Overall this seems to be the best offline approach for this kind of task having both a very fast learning and overall results that are comparable to the online approach.

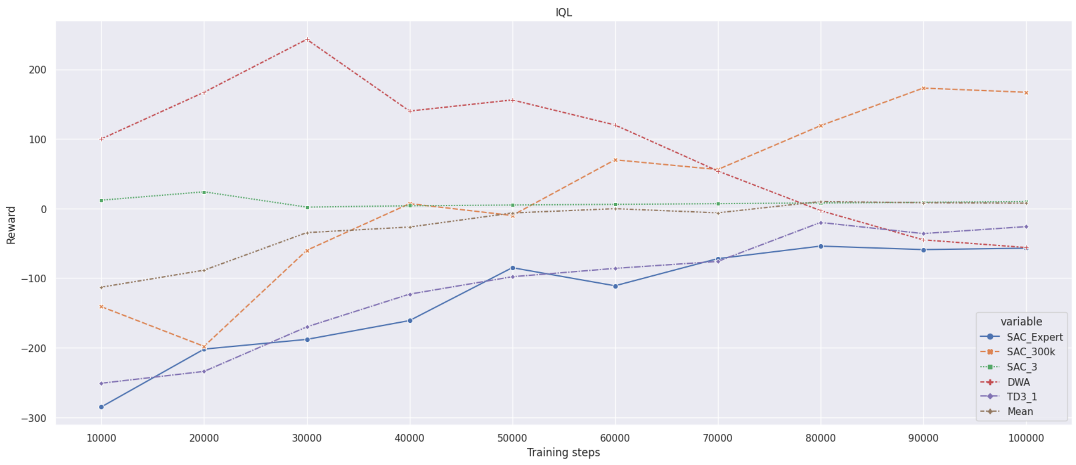

Figure 7.2: Reward during the training of IQL algorithm

7.3.3 CQL

Conservative Q-Learning shows the worst results over all the methods, even if it does not reach very bad results learning from some specific dataset. This behaviour is given by the fact that it learns to avoid collisions, but it is not able to perform the requested task in most of the cases. Indeed few test were successfully completed by this method, in most of the cases it stopped near some obstacle not being able to follow a good trajectory.

7.3.4 EDAC

This methods have overall the good performances, the training which took a the most noticeable result is the one made on the *DWA* dataset even if it seems to suffer from an high variance. This results suggest that further investigations are possible. Indeed due to its simplicity and easy transfer from an online model, it would be a very easy approach to handle offline learning. However it is very noticeable from the results how in most of the setting the algorithm start to overfitting the data giving lower results from around 70000 learning steps. This behaviour does seems to scale with the dataset size.

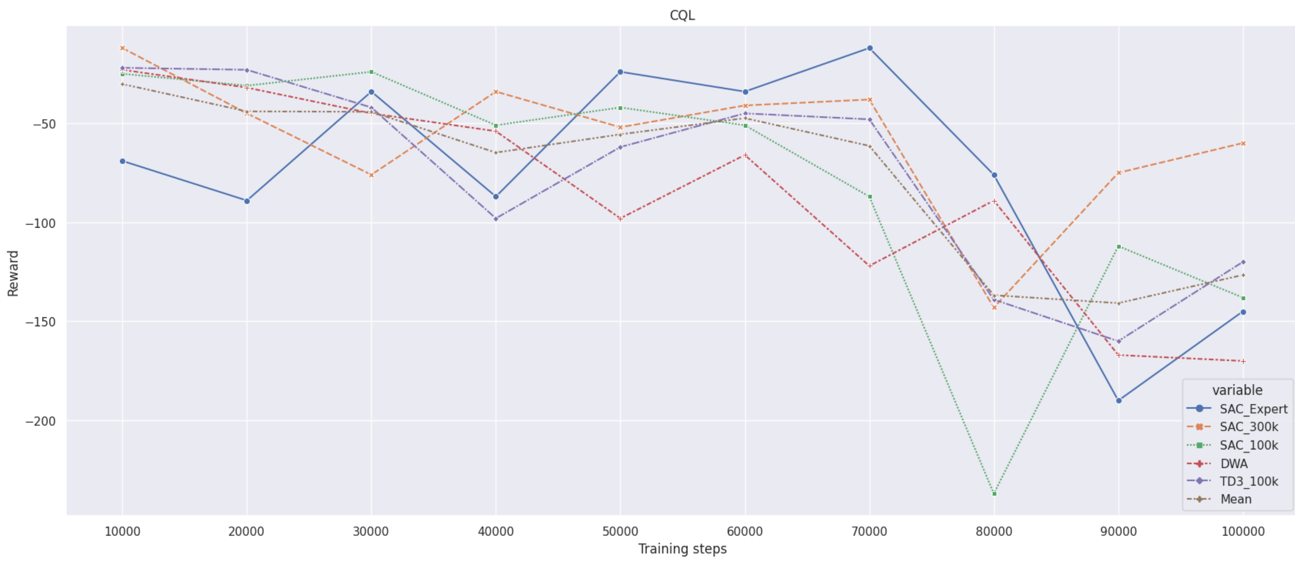

Figure 7.3: Reward during the training of CQL algorithm

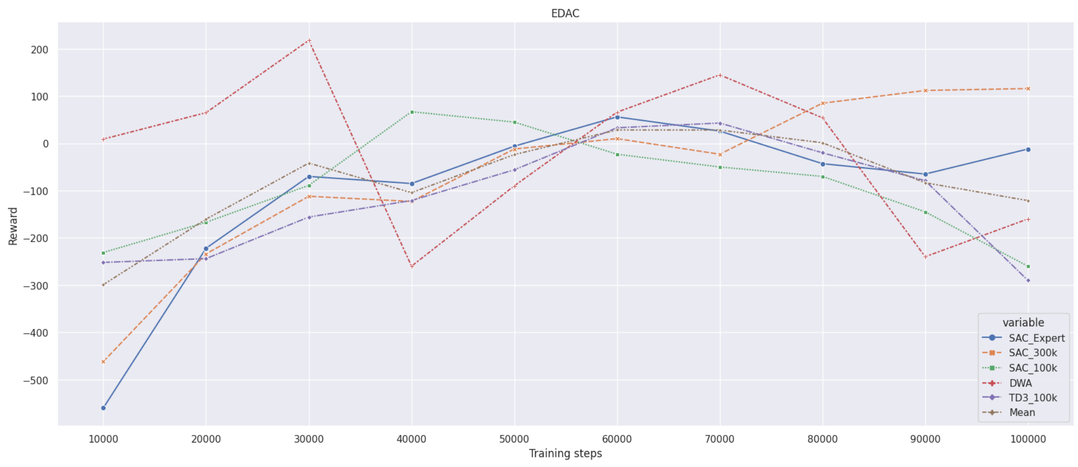

Figure 7.4: Reward during the training of IQL algorithm

7.3.5 Examples of cases

Here three examples of task and results are analysed with more focus on the trajectories.

Pose:[1.5 ,7.5, -1.57], Goal: [-4,4] As one can see from 7.5, this task has been completed from all the algorithms following similar trajectories, it is worth notice how even if this task does not seems an easy one form a human prospective the algorithm are able to complete it without trouble. One of the reasons why this is the case is because of the wall that "guides" the trajectory.

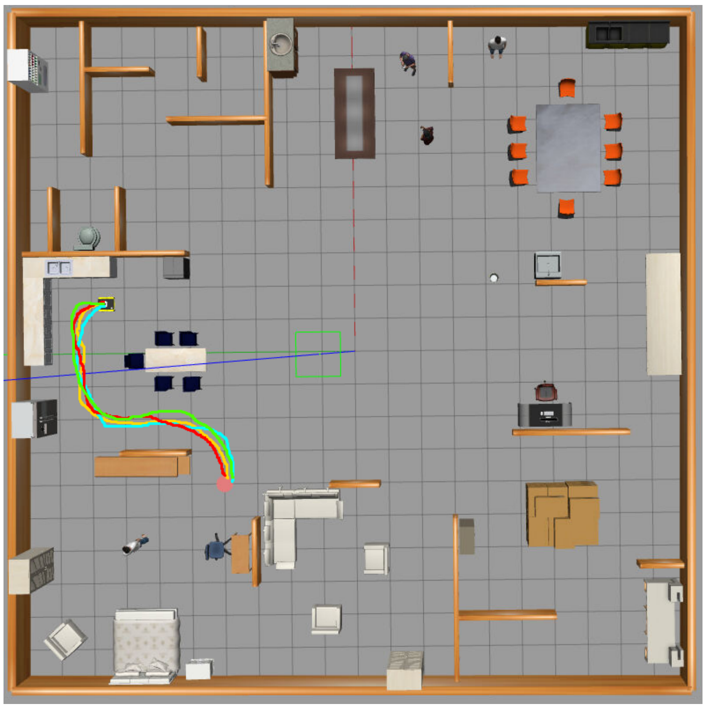

Figure 7.5: Trajectories from:[1.5 ,7.5, -1.57] to [-4,4]. Yellow: IQL, red: EDAC, light blue: BCQ, green: CQL

Pose:[8 ,-9, 3.14], Goal: [3.5,-3.5] From the results analysed before it is Here is shown how some tasks were successfully completed only by two of the four algorithms, indeed in figure 7.6 one can notice how they have found a trajectory to solve the task, however it was not the fastest one. As seen before CQL continue to stop into some hard decision states.

Pose:[7 ,9, 3.14], Goal: [-7,7] In figure 7.7 one of the most difficult task is shown. As one can notice IQL, EDAC and BCQ collided against some objects, while following the behaviour seen before CQL stopped before colliding, however it did not completed the task as well.

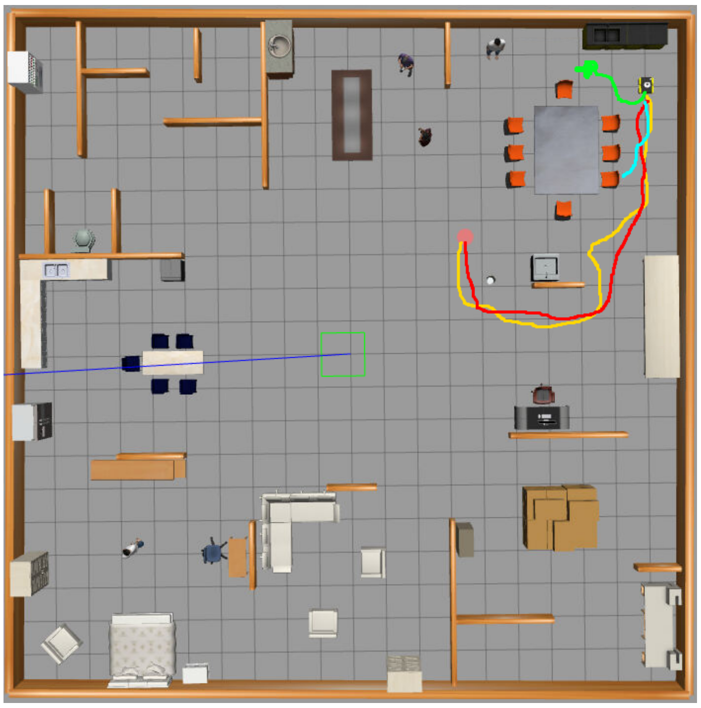

Figure 7.6: Trajectories from:[8 ,-9, 3.14] to [3.5,-3.5]. Yellow: IQL, red: EDAC, light blue: BCQ, green: CQL

7.4 Comparison between online and offline approach

After having evaluate how good are the performance of the offline approaches one can ask how are they in comparison to online methods. Here some considerations will be made in terms of results, time-consuming and difficulties.

7.4.1 Results

One can expect the results obtained by an online training to be way more optimal in terms of performances, however here something unexpected will be shown. Indeed if one take into account the results over all the tasks obtained by online RL methods it would encounter lower results due to the fact that this kind of approaches on one hand learn by still taking into consideration the exploration and in the other are capable to deal with some specific task at time. Even if all the tasks were completed over the training the algorithms were able to complete few of them with the same set of parameters. This can be caused by different reasons, one in on-policy training by the fact that the algorithm focus the learning on just one task, however this

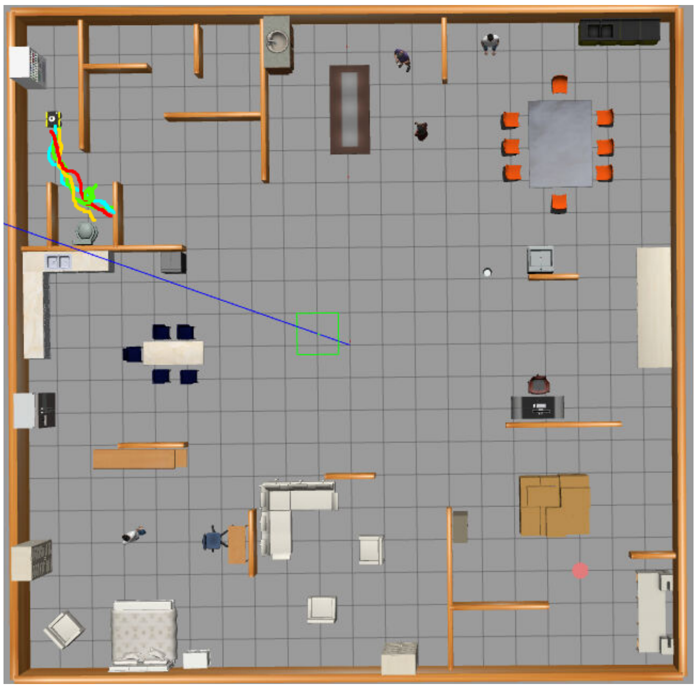

Figure 7.7: Trajectories from:[7 ,9, 3.14] to [-7,7]. Yellow: IQL, red: EDAC, light blue: BCQ, green: CQL

should not be the case for what concern off-policy methods due to their learning structure. For off-policy methods some factors can lead to this problem, one is that in most cases, such as SAC, being the exploration incorporated with the Q-values leads to lower performances, this would be easily fixed by lowering the temperature in the last steps of learning, another can be the fact that being the space various and with differences that are not easy to interpret not having a control over the Q-values can lead to some unexpected results. However this kind of problems of the online learning can be mitigate by taking more trial, leading to better performances over this setting. In the other hand one of the main reason to use reinforcement learning approaches is to avoid the study of the single problem to find the optimal strategy.

7.4.2 Time consuming

One of the most important aspect of finding new learning methods is the time used to make the algorithm understand the task and make it able to complete it. Here one factor is the discriminant on what approach is better to deal with the mobile robot task, the presence of the dataset. Indeed ,if one, due to various reasons, already have a structure dataset with the records of an agent computing the requested task it is very convenient in terms of time (and results) applying offline reinforcement learning methods, indeed the most time consuming part of the learning are the simulations, by a lot. In the other hand if this is not the case and the application of a non-expert agent does not lead to severe lost in financial terms applying an online reinforcement learning algorithm can be better.

7.4.3 Difficulties

Online and offline reinforcement learning pose both a lot of difficulties, some of them are similar some are specific of a method. First of all both are very time consuming due to the large number of hyperparameters and the high dependence of the results form them. Some of them in an online setting are "trainable" during the learning, such as the "temperature" in SAC, making it easier for the user to find the optimal ones. On the other hand being not possible to interact and getting feedback about the performances in an offline setting make this approach impossible for it. One of the main problem of RL in general is the study of the problem being fundamental for such learning approach, as an example the choice of the perfect reward function is one of the things which can deeply change an algorithm behaviour. As mentioned in the section before the main discriminants in terms of difficulties and time consuming are both having already a dataset and the ease of doing new simulations, both virtual or real. In many cases one can find himself in a situation where both this condition are in the worst possible case scenario, not having a dataset and being it very hard to simulate an online setting. In this case many options are possible if one still want to use a RL approach, one is to make the dataset by hand, for example by moving the robot through a human pilot, another which in this case I would suggest is to use other kind of algorithm, which are safer in terms of outcomes, such as DWA to take a dataset and than use it to make an offline algorithm learn an optimal or sub-optimal policy.

Chapter 8

Conclusions

This thesis aimed to assess the current state of Offline Reinforcement Learning (ORL) methods applied to mobile robots within a GAZEBO simulation of a real mobile robot in an indoor setting. The primary goal was to guide the robot to a specific position using only the relative coordinates of the goal and 36 LiDAR distance data.

Initially, an introduction and review of Online Reinforcement Learning (RL) methods were conducted, with a specific focus on TD3 and SAC. These methods were employed to obtain datasets for the offline segment of the project. Subsequently, an exploration of key methods addressing the challenges of Offline Reinforcement Learning was undertaken. This analysis highlighted the difficulties associated with a fixed dataset, including issues during training and the application of such algorithms in real environments, such as distributional shift and the inability to explore new states and actions. This analysis serves as the foundation for most ORL methods.

Following a general evaluation of ORL methods, four specific algorithms—Batch-constrained Q-learning, Implicit Q-learning, Conservative Q-learning, and Ensemble Diversified Actor-Critic—were selected based on their superior performance over the benchmark D4RL. Each algorithm was meticulously examined, elucidating the underlying thought processes, providing mathematical proofs, and justifying their selection.

Before delving into the experiments, details of the experimental environment were presented, including information on LiDAR, odometry, and each reinforcement learning component. A specific focus was placed on one of the key aspects of offline reinforcement learning—the acquisition of various datasets.

The results of all experiments were scrutinized, revealing that Implicit Q-learning (IQL) outperformed the other algorithms overall, although Ensemble Diversified Actor-Critic (EDAC) also exhibited commendable results, albeit with a greater dependence on dataset.

To conclude this thesis, it emphasizes that, in general, Offline Reinforcement Learning methods are not yet ready for practical utilization in mobile robot tasks. The challenges lie in the difficulty of finding optimal hyperparameters and learning steps, which, while manageable in a simulated setting, loses practicality in real-world scenarios. Moreover, many ORL methods are surpassed by conventional approaches. Despite these problems, the field has witnessed significant growth in recent years, offering optimism for future approaches. Finally, it is noted that small neural networks, as employed in this thesis, still struggle to master complex tasks in detail.

Appendix A

A.1

Here Dynamic Window Approach (DWA) will be briefly explain. DWA is a local motion planning algorithm commonly used in mobile robotics. It enables a robot to navigate in real-time by evaluating different velocity commands and selecting the one that leads to a safe and optimal.

A.1.1 Motion equations

Let $x(t)$ and $y(t)$ denote the robot's coordinate at time t in some global coordinate system, and let the robot's orientation (heading direction) be $\theta(t)$. The motion of a synchro-drive robot is constrained in a way such that the translational velo city v always leads in the steering direction θ of the robot. Let be $x(t_i)$ and $y(t_i)$ denote the coordinates of the robot at time t_i . Let $v(t)$ denote the translational velocity of the robot at time t, and $\omega(t)$ its rotational velocity. Then the evolution on time of the coordinates x and y can be expressed as:

$$x(t_i) = x(t_0) + \int_{t_0}^{t_i} v(t)\cos\theta(t)dt \tag{A.1}$$

$$y(t_i) = y(t_0) + \int_{t_0}^{t_i} v(t)\sin\theta(t)dt \tag{A.2}$$

In a real setting it is not possible to continuously control the robot, thus let be t_0 and t_n the initial and final time respectively and n the number of possible control over this period. Than the acceleration $\dot{v}(t)$ and the first derivative over time of the angular velocity $\dot{\omega}(t)$ will remain constant piece-wise in every small interval $t_{i+1} - t_i$. Knowing this one can rewrite the former as

$$\begin{aligned} x(t_i) =& x(t_0) + \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} (v(t_i) + \dot{v}(t_i)(t - t_i) \\ & \cdot \cos\left(\theta(t_i) + \omega(t_i)(t - t_i) + \frac{1}{2}\dot{\omega}(t_i)(t - t_i)^2\right) dt \end{aligned} \tag{A.3}$$

$$\begin{aligned} y(t_i) =& y(t_0) + \sum_{i=0}^{n-1} \int_{t_i}^{t_{i+1}} (v(t_i) + \dot{v}(t_i)(t - t_i) \\ & \cdot \sin\left(\theta(t_i) + \omega(t_i)(t - t_i) + \frac{1}{2}\dot{\omega}(t_i)(t - t_i)^2\right) dt. \end{aligned} \tag{A.4}$$

As shown in Fox et al. 1997 by solving the integral one can get the following form which is more suitable for in a practical setting:

$$x(t_n) = x(t_0) + \sum_{i=1}^{n-1} F_x^i(t_{i+1}) \tag{A.5}$$

$$y(t_n) = y(t_0) + \sum_{i=1}^{n-1} F_y^i(t_{i+1}) \tag{A.6}$$

where

$$F_x^i(t) = \begin{cases} \frac{v_i}{\omega_i}\left(\sin(\omega(t_i) - \sin\left(\theta(t_i) + \omega_i(t - t_i)\right)\right), & \omega_i \neq 0 \\ v_i \cos(\omega(t_i)t, & \omega_i = 0 \end{cases} \tag{A.7}$$

$$F_y^i(t) = \begin{cases} -\frac{v_i}{\omega_i}\left(\cos(\omega(t_i) - \cos\left(\theta(t_i) + \omega_i(t - t_i)\right)\right), & \omega_i \neq 0 \\ v_i \sin(\omega(t_i)t, & \omega_i = 0 \end{cases} \tag{A.8}$$

It is worth noting how if $\omega_i = 0$ than the robot follow a straight line, in the other hand if not the following holds

$$\left(F_x^i(t_i) - \left(-\frac{v_i}{\omega_i}\sin(\theta(t_i))\right)\right)^2 + \left(F_y^i - \frac{v_i}{\omega_i}\cos(\theta(t_i))\right)^2 = \left(\frac{v_i}{\omega_i}\right)^2 \tag{A.9}$$

This shows how the trajectories are circles of radius $\frac{v_i}{\omega_i}$.

A.1.2 Overview of the algorithm

The following scheme is taken directly from Fox et al. 1997:

Search space The s e arch space of the possible velocities is reduced in three steps:

1. **Circular trajectories**: The dynamic window approach considers only circular trajectories (curvatures) uniquely determined by pairs (v, ω) of translational and rotational velocities. This results in a two-dimensional velocity search space.

2. **Admissible velocities**: The restriction to admissible velocities ensures that only safe trajectories are considered. A pair (v, ω) is considered admissible, if the robot is able to stop before it reaches the closest obstacle on the corresponding curvature.

3. **Dynamic window**: The dynamic window re stricts the admissible velocities to those that can be reached within a short time interval given the limited accelerations of the robot.

Optimization The objective function

$$G(v, \omega) = \sigma\left(\alpha \cdot heading + \beta \cdot dist + \gamma \cdot vel\right) \tag{A.10}$$

is maximized. With respect to the current position and orientation of the robot this function trades off the following aspects:

1. **Target heading**:*heading* is a measure of progress towards the goal location. It is maximal if the robot moves directly towards the target

2. **Clearance**:*dist* is the distance to the closest obstacle on the trajectory. The smaller the distance to an obstacle the higher is the robot's desire to move around it.

3. **Velocity**: *vel* is the forward velocity of the robot and supports fast movements.

The function σ smoothes the weighted sum of the three components and results in more side-clearance from obstacles.

Bibliography

Fox, D., W. Burgard, and S. Thrun (1997). «The dynamic window approach to collision avoidance». In: *IEEE Robotics & Automation Magazine* 4.1, pp. 23–33. DOI: 10.1109/100.580977 (cit. on pp. 3, 70, 84).

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine (2018). *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.* arXiv: 1801.01290 [cs.LG] (cit. on pp. 4, 34).

Fujimoto, Scott, Herke van Hoof, and David Meger (2018). *Addressing Function Approximation Error in Actor-Critic Methods.* arXiv: 1802.09477 [cs.AI] (cit. on pp. 4, 17, 32).

Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu (2016). *Asynchronous Methods for Deep Reinforcement Learning.* arXiv: 1602.01783 [cs.LG] (cit. on p. 4).

Fu, Justin, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine (2021). *D4RL: Datasets for Deep Data-Driven Reinforcement Learning.* arXiv: 2004.07219 [cs.LG] (cit. on p. 5).

Fujimoto, Scott, David Meger, and Doina Precup (2019). *Off-Policy Deep Reinforcement Learning without Exploration.* arXiv: 1812.02900 [cs.LG] (cit. on pp. 5, 50).

Kumar, Aviral, Aurick Zhou, George Tucker, and Sergey Levine (2020). «Conservative Q-Learning for Offline Reinforcement Learning». In: *CoRR* abs/2006.04779. arXiv: 2006.04779. URL: https://arxiv.org/abs/2006.04779 (cit. on pp. 5, 55, 57).

Kostrikov, Ilya, Ashvin Nair, and Sergey Levine (2021). *Offline Reinforcement Learning with Implicit Q-Learning.* DOI: 10.48550/ARXIV.2110.06169. URL: https://arxiv.org/abs/2110.06169 (cit. on pp. 5, 53).

An, Gaon, Seungyong Moon, Jang-Hyun Kim, and Hyun Oh Song (2021). *Uncertainty-Based Offline Reinforcement Learning with Diversified Q-Ensemble.* arXiv: 2110.01548 [cs.LG] (cit. on pp. 5, 57, 58).

Yu, Tianhe, Garrett Thomas, Lantao Yu, Stefano Ermon, James Zou, Sergey Levine, Chelsea Finn, and Tengyu Ma (2020). *MOPO: Model-based Offline Policy Optimization.* arXiv: `2005.13239 [cs.LG]` (cit. on p. 5).

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction.* Second. The MIT Press. URL: `http://incompleteideas.net/book/the-book-2nd.html` (cit. on p. 6).

Hasselt, Hado van, Arthur Guez, and David Silver (2015). *Deep Reinforcement Learning with Double Q-learning.* arXiv: `1509.06461 [cs.LG]` (cit. on p. 17).

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller (2013). *Playing Atari with Deep Reinforcement Learning.* DOI: `10.48550/ARXIV.1312.5602`. URL: `https://arxiv.org/abs/1312.5602` (cit. on p. 30).

Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver (2015). *Prioritized Experience Replay.* DOI: `10.48550/ARXIV.1511.05952`. URL: `https://arxiv.org/abs/1511.05952` (cit. on p. 31).

Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra (2019). *Continuous control with deep reinforcement learning.* arXiv: `1509.02971 [cs.LG]` (cit. on p. 32).

Precup, Doina, Richard Sutton, and Satinder Singh (June 2000). «Eligibility Traces for Off-Policy Policy Evaluation». In: *Computer Science Department Faculty Publication Series* (cit. on p. 43).

Jiang, Nan and Lihong Li (2016). *Doubly Robust Off-policy Value Evaluation for Reinforcement Learning.* arXiv: `1511.03722 [cs.LG]` (cit. on p. 43).

Thomas, Philip S. and Emma Brunskill (2016). *Data-Efficient Off-Policy Policy Evaluation for Reinforcement Learning.* arXiv: `1604.00923 [cs.LG]` (cit. on p. 43).

Wang, Yu-Xiang, Alekh Agarwal, and Miroslav Dudik (2017). *Optimal and Adaptive Off-policy Evaluation in Contextual Bandits.* arXiv: `1612.01205 [stat.ML]` (cit. on p. 43).

Farajtabar, Mehrdad, Yinlam Chow, and Mohammad Ghavamzadeh (2018). *More Robust Doubly Robust Off-policy Evaluation.* arXiv: `1802.03493 [cs.AI]` (cit. on p. 43).

Thomas, Philip, Georgios Theocharous, and Mohammad Ghavamzadeh (Feb. 2015). «High-Confidence Off-Policy Evaluation». In: *Proceedings of the AAAI Conference on Artificial Intelligence* 29.1. DOI: `10.1609/aaai.v29i1.9541`. URL: `https://ojs.aaai.org/index.php/AAAI/article/view/9541` (cit. on p. 43).

Levine, Sergey and Vladlen Koltun (17–19 Jun 2013). «Guided Policy Search». In: *Proceedings of the 30th International Conference on Machine Learning.* Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research 3. Atlanta, Georgia, USA: PMLR, pp. 1–9. URL: `https://proceedings.mlr.press/v28/levine13.html` (cit. on p. 45).

Kingma, Diederik P and Max Welling (2013). *Auto-Encoding Variational Bayes.* arXiv: `1312.6114 [stat.ML]` (cit. on p. 50).

Peters, Jan and Stefan Schaal (2007). «Reinforcement Learning by Reward-Weighted Regression for Operational Space Control». In: *Proceedings of the 24th International Conference on Machine Learning.* ICML '07. Corvalis, Oregon, USA: Association for Computing Machinery, pp. 745–750. ISBN: 9781595937933. DOI: `10.1145/1273496.1273590`. URL: `https://doi.org/10.1145/1273496.1273590` (cit. on p. 54).

Wang, Qing, Jiechao Xiong, Lei Han, peng sun peng, Han Liu, and Tong Zhang (2018). «Exponentially Weighted Imitation Learning for Batched Historical Data». In: *Advances in Neural Information Processing Systems.* Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Vol. 31. Curran Associates, Inc. URL: `https://proceedings.neurips.cc/paper_files/paper/2018/file/4aec1b3435c52abbdf8334ea0e7141e0-Paper.pdf` (cit. on p. 54).

Peng, Xue Bin, Aviral Kumar, Grace Zhang, and Sergey Levine (2019). *Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning.* arXiv: `1910.00177 [cs.LG]` (cit. on p. 54).

Nair, Ashvin, Abhishek Gupta, Murtaza Dalal, and Sergey Levine (2021). *AWAC: Accelerating Online Reinforcement Learning with Offline Datasets.* arXiv: `2006.09359 [cs.LG]` (cit. on p. 54).

Martini, Mauro, Andrea Eirale, Simone Cerrato, and Marcello Chiaberge (2023). «PIC4rl-gym: a ROS2 Modular Framework for Robots Autonomous Navigation with Deep Reinforcement Learning». In: *2023 3rd International Conference on Computer, Control and Robotics (ICCCR)*, pp. 198–202. DOI: `10.1109/ICCCR56747.2023.10193996` (cit. on p. 63).