POLITECNICO DI TORINO

MASTER's Degree in Ingegneria Matematica

MASTER's Degree Thesis

Offline Reinforcement Learning for hybrid vehicles energy consumption optimization

Supervisors

Prof. Francesco VACCARINO

Dott. Luca SORRENTINO

Dott. Rosalia TATANO

Candidate

Federico GAMBASSI

March 2022

Sator
Arepo
Tenet
Opera
Rotas [Sator square]

Summary

In the last decade, we have seen many practical applications of Reinforcement Learning (RL) to different practical tasks, that obtained great success. The fields of applications are various, ranging from robotics and autonomous driving, to AI for video games or strategic games like chess. In particular, a key element for the success of these methods is without doubt the integration of RL with Deep Learning. Indeed, thanks to the advances in terms of computational power, today we have the possibility of training neural deep approximators to learn patterns from unstructured data like images or text.
However, very recently a new paradigm has emerged from traditional RL, called Offline Reinforcement Learning (Offline RL), in which every interaction between the agent and the environment is prohibited, so that the agent to be trained can only learn from previously collected datasets. The necessity for this new branch of RL stems from the fact that, in many practical applications, learning from scratch in the real environment can be unfeasible, or even dangerous for the agent and the surroundings.
The purpose of this work is to experiment with two offline RL algorithms, namely CQL and COMBO, on a problem concerning the energy consumption of a hybrid vehicle along a fixed trajectory. In particular, in order to assess the goodness of these methods we also chose a suitable baseline, which is basically a plain adaptation of classical online DQN algorithm to offline RL.
The training and testing of the three agents (including the baseline) was performed on a simulated environment developed by PoliTo and AddFor S.p.a. written in Python/Matlab. Regarding the training, datasets have been gathered using different online RL agents trained on the same framework. In this way, we were able to gather up to 5 different datasets.
CQL is a model-free algorithm, meaning that the data is used just to learn how good are certain state-action couples; on the other hand, COMBO is a model-based algorithm, meaning that it first try to learn a representation of the outside world using the data available.
The experiments performed on CQL and COMBO were differentiated based upon the dataset used for training an agent. At the end of each training, a testing

episode was performed; as a metric to compare the results, we chose the return obtained by the agent on the testing episode. What we found is that, while CQL is not able to outperform the baseline, COMBO is able to obtain good results in every dataset-based scenario; moreover, COMBO obtains the best results overall in 4 out of 5 scenarios.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The traditional approach to **Reinforcement Learning** (or RL, for short) has always been to tackle problems within an *online* setting, meaning that the agent trained enacts a *real-time* interaction with the environment, which in turns output a numerical signal (the **reward**). This online paradigm has proven to be succesful in many scenarios, especially when coupled with **Deep Learning**, which let us exploit powerful approximation architectures in order to correctly learn from high-dimensional and *unstructured* data, like images and text.
Within the set of RL algorithms we can characterize two different families, called **model-free** and **model-based** RL: methods from the first family codify the information about the environment directly within the agent, which in turn assigns good rewards to promising actions and bad rewards to nonoptimal actions; on the other hand, a model-based algorithm attempts to learn a model representation of the transition dynamics of the environment: in this way, starting from the current state, the agent can look ahead to search for promising states.

There are situations in which training an online RL agent from scratch could be expensive, or even *unsafe*: for example, it could be that we have to control a robotical arm operating on an industrial line, or we could be in the situation where an RL agent must assign the proper treatment to a human patient based on some data concerning his health. To this end, we must prevent the agent from exploring too much the set of solutions, because the risk-reward ratio is simply not worth it. At the same time, the agent must be able to explore a large portion of the state-action space in order to optimally solve the task.

In order to go around this *impasse*, in the last years the researchers have tried to follow a different path, which led research to a new RL paradigm called **Offline Reinforcement Learning**. In Offline RL, the interaction between agent and environment is not allowed; instead, the agent can solely learn from data which has been previously collected by other domain experts (for example, they can be online RL agents or even human demonstrations). In this way, we prevent the agent to take risky actions in the environment, avoiding economical and physical damages.

The lack of exploration and real-time feedback in Offline RL is the cause of an inherent structural problem called *distributional shift*: this is the change in distribution of states and actions examined and chosen by the agent in relation to the ones from the training data: this shift occurs naturally as the agent learns and modifies its behavior; the fact that it no longer receives rewards from the environment has the consequence that the behaviors that it thinks can be rewarding might actually be overly optimistic, leading to poor results at test time.

In recent works, an adopted solution is to design agents that take a *conservative* approach at training time, meaning that actions that are too much distant in distribution from the ones in the training dataset are seen with diffidence.

The goal of this work thesis is to investigate algorithms with such approach on a practical task: we want to train an agent in order to optimally control the energy consumption of a hybrid vehicle along a fixed trajectory. In particular, we want to make a comparison between the offline RL model-free and the offline RL model-based approaches and assess their power against a suitable baseline.

The Offline RL research field is rather young, although many algorithms have already been theorized and experimented: the most known is probably an extension to the offline scenario of the well-known **DQN** algorithm, called **CQL** (**C**onservative **QL**earning): in this extension, the data used for training is not sampled from the interaction with the environment; instead, the agent is provided with a fixed training dataset that has already been gathered; then, at each training step, a batch from this dataset is sampled and a conservative training step is performed. Our goal is to apply CQL along with a more sophisticated offline model-based algorithm called **COMBO** (**C**onservative **O**ffline **M**odel-**B**ased Policy **O**ptimization), which is basically an upgrade of CQL to the model-based world, meaning that the agent is coupled with a model (also trained on the offline dataset) which accurately approximates the dynamics of the environment. In our thesis, these two methods are compared to a chosen baseline, which consists of a plain adaptation of DQN to the offline scenario.
To achieve the desired goals, experiments have been performed on a practical driving mission simulator developed by **Addfor Indsutriale** and **Politecnico di Torino**. In the experiments, we tested CQL and COMBO on this simulated environment, along with the chosen baseline, after training them on data collected by online agents from this environment. Basically, we used three different agents: the first is a random one (**AgentRandom**), meaning that at each step it takes an action uniformly sampled from the set of feasible action given the current state; the second one is based on the well-known **QLearning** algorithm (therefore called **AgentQLearning**), after being fully trained. Finally, we employed the online DQN algorithm to obtain two different agents, one coming from the partially

trained DQN, the other coming from the fully trained one (both these ones are called **AgentDDQN**). As we show in Chapter 4, we collect up to 5 different datasets (one coming from **AgentRandom**, one from **AgentQLearning**, three from **AgentDDQN**). Then, each offline algorithm is trained on all of the datasets; in this way, we obtain 5 different agents per algorithm. What we found is that COMBO performs better than CQL and the baseline; In particular, we implement two different instances of COMBO, which differ in the choice of the model: one version, called Single COMBO, assumes that the transition dynamics of the environment follow a normal distribution; on the other hand, the second version, called Ensemble COMBO, employs an ensemble of models, each of them representing a normal distribution: this ensures that variance is reduced at inference time. From the experiments performed we evince that, on every dataset, COMBO is better than CQL and the baseline.
This thesis is structured as follows: in Chapter 2, we provide an introduction to Reinforcement Learning, with details on the agent-environment interface and a rather rigorously mathematical treatment concerning the goal of RL, p1olicies and value functions; moreover, some basic algorithms like **Policy Iteration** are described, along with some properties concerning optimality convergence; lastly, we provide an introduction to **Deep Reinforcement Learning**. Chapter 3 deals with Offline Reinforcement Learning, discussing its inherent problems and the solutions employed by many promising algorithms; we present CQL and COMBO, along with their theoretical properties. The performed experiments are detailed in Chapter 4, which provides also a brief description of the simulated environment and its related task; we discuss how data has been collected for training, and we present the results obtained by our algorithms against the baseline. Finally, in Chapter 5 we draw our conclusions.

Chapter 2

Reinforcement Learning

When we think about the nature of practical learning, we usually have in mind the idea of acquiring a *task* through an interaction with some sort of environment. Indeed, we can think about a child who's learning to ride a bicycle, or a bird pet who wants to fly for the first time: in each of these situations, the environment responds to what we do, and we seek to influence its changing through our behavior. More generally, learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence: in this work we deal with **Reinforcement Learning**, or shortly, RL, a mathematical framework that deals with a type of learning process.
RL is a class of problems and mathematical methods that deals with what *to do* in a certain scenario, which means what actions to perform in it in order to maximize a numerical signal; so, it's a particular kind of optimization problem.
More formally, we have an **agent** that interacts with an **environment** by taking some **actions**. The environment is represented by an internal **state** which changes accordingly to the present state and the action performed by the agent.
Reinforcement Learning is different from *supervised learning* (also referred to as SL), the most known field of *machine learning*. In SL, we deal with a training set of labeled examples provided by an external domain expert. The objective here is to generalize, i.e. to extrapolate a function in order to answer to previously unseen data. On the other hand, in RL we do not have a label that tells us whether the instance is "right"/"wrong" (as in the case of binary classification). Instead, the agent receives a **reward**, which is a numerical *intensity* regarding the goodness of the taken action. Moreover, in interactive problems one does not have already an experience dataset at hand (unless we are dealing with Offline RL, which will be presented in subsequent chapters). This means that if we want to acquire significative learning, we have to *explore* the world surrounding us, *acting* in it and collecting essential information.
RL has its roots in the field of **Dynamic Programming** (DP), a general and

flexible principle that is used to design optimization algorithms, based upon the fundamental principle of decomposing a multistage (stochastic or not) decision problem into a sequence of simpler, single-stage problems. However, as we will see, the remarkable feature of RL, thanks to which it is becoming a pervasive technology, lies in the fact that one does not need to know how to represent the environment. This ability, mixed with the powerful approximation capacity of neural networks, is responsible for RL's success in many fields, ranging from robotics to management science.
In this chapter we present the framework under which many concepts of classic RL are developed.

2.1 Agent-Environment Interface

The learner or decision maker is called the *agent*. The world with which the agent interacts and in which it takes actions is the *environment*, which in response to said actions updates its internal *state* and provides the agent with a *reward*: we can imagine a closed-loop feedback like in Fig.2.1. In this work, the mathematical

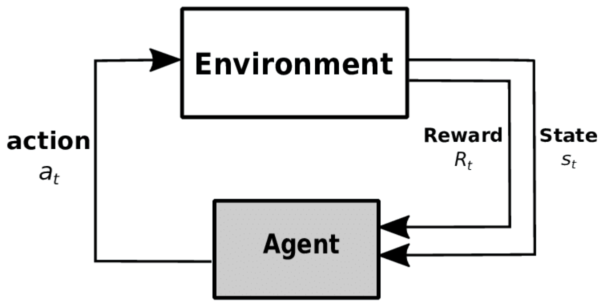


Figure 2.1: RL closed-loop feedback diagram: the agent receives a state representation and a reward associated with it from the environment; in turn, the agent selects an action which is transmitted to the environment, which updates its previous state representation and reward signal.

structure modeling the system is the (finite) **Markov Decision Process**, or shortly MDP. Basically, the interaction between the agent and the environment can be represented as a discrete-time stochastic process on the state representation space of the environment. more formally:

Definition 2.1.1 (**M**arkov **D**ecision **P**rocess)**.** *A Markov decision process is a 4-tuple* (S, A, P, R) *where*

- S *is a set of states called the* ***state space***
- A *is a set of actions called the* ***action space***

- $P : S \times A \times S \to [0,1]$ *is the function underlying the transition dynamics of the environment; that is, we define*

$$P(s', a, s) \coloneqq Pr(S_{t+1} = s' | S_t = s, A_t = a) \tag{2.1}$$

- $R : S \times A \times S \to \mathbb{R}^+$ *is the reward function: this models the feedback communication between agent and environment, that is*

$$R(s', a, s) \coloneqq r, \tag{2.2}$$

  where r is the immediate reward received after performing action a in state s, with the environment transitioning to the next state s'.

Every (finite) MDP has a (finite) **state space $\mathcal{S}$** which represent the possible configurations that the environment can assume. On the other hand, the set of possible actions that the agent can take at a given time generally depends on the present state; so, when the environment is in state $S_t \in \mathcal{S}$, the agent can take an action in the **feasible action space $\mathcal{A}(S_t)$**. At each discrete time step $t \in \mathbb{N}$, the environment is in a state S_t; when the agent takes an action $A_t \in \mathcal{A}(S_t)$; then, the environment updates the state to S_{t+1} and outputs a reward $R_{t+1} \in \mathbb{R}$.
In this case, both state and reward are discrete random variables and their probability distributions depend only on the previous state and action, that is

$$p(s', r | s, a) \coloneqq \mathbb{P}[S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a], \tag{2.3}$$

for all $s, s' \in \mathcal{S}, a \in \mathcal{A}(s)$. p is usually called the **transition dynamics function** and, under the Markov hypothesis, it completely characterizes the environment's dynamics. This means that the probability of the environment transitioning to state s and outputting a reward r simply depends on the state of the environment and the action taken by the agent at the previous time step, and not on the past history: in a sense, we can say that *the environment codifies the entire previous history in the current state.*

2.2 The goal of RL

We have seen that the only way the environment communicates with the agent is through the output reward. In fact this signal, which is a real number, suffices to enclose all the information the agent needs to learn. This is because the agent's goal is to maximize the total amount of reward it receives, which can be rephrased by the *reward hypothesis* (taken from [1]):

> " That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward). "

This means that the objective is not to simply maximize immediate reward, but the cumulative reward in the long run.
At this point it should be clear to the reader that the way the reward function is shaped influences the goal we want the agent to accomplish. For example, let's consider an agent which must learn to play chess: the goal is clearly accomplished if an agent can win the game against his opponent. Now: what reward function should we define?
This is the kind of situation in which we want to be careful: indeed, if we assign positive reward when the agent captures some pieces (so that we associate positive reward with a subgoal), the agent could find a way of losing the game but after capturing the maximum number of pieces possible (or the pieces that lead to greatest reward), pursuing a **greedy** strategy. Actually, after some thinking, we recognize the right reward function in the one that assigns reward +1 when the game is won, -1 for losing and 0 whenever that's a draw and for all nonterminal positions.

2.3 Formal definition of return

Now let's state the goal introduced in the previous section in a formal mathematical way. What does it mean to "maximize the cumulative reward in the long run" ?
If we are at time t, we have ahead of us a sequence of rewards $R_{t+1}, R_{t+2}, \dots, R_T$ (with T that can also be infinite). The sum of all these rewards is called the **return** G_t, defined as

$$G_t \coloneqq \sum_{\tau=t}^{T} R_\tau \tag{2.4}$$

and, since it's a random variable, what we seek to maximize is the *expected return* $\mathbb{E}[G_t]$.
This definition makes sense when T is finite; this is the case of the so-called *episodic tasks*. In this case, a sequence of tuples $\{(s_t, a_t, r_t, s_{t+1})\}$, with $t = 1, \dots, T$, is called an **episode**: every episode finishes with the agent in the same special state called the **terminal state**; after that, the agent is reset to any of the possible initial states and a new episode begins. The time of termination is in general a random variable.
On the other hand, let's suppose a task where we have a robot that must solve some kind of balancing problem: the objective is to remain in an equilibrium phase the longest possible time. We understand that this kind of problem doesn't suit well an episodic framework; tasks like this one are called **continuing tasks** and, since it's $T = \infty$, the return definition that we have given it's problematic, because this random variable could be infinite with nonzero probability. In order to solve this problem we must add the concept of **discounting**.

The idea is to introduce a **discounting factor** $0 < \gamma < 1$ and, instead of the simple expected return, to compute the expected value of the **discounted return**

$$G_t \coloneqq R_{t+1} + \sum_{k=1}^{\infty} \gamma^k R_{t+k+1}. \tag{2.5}$$

Now this sum is finite in expectation, as long as the rewards are a.s. bounded. For example, if the rewards are +1 constantly, then the expected return is $\sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}$.
Discounting surely appears as a mere trick to make the return converge. While this is certainly its main *raison d'être*, discounting has also other meanings, even related to the nature of the problem: for example, discounting is common in financial applications, where we have a stream of future random cashflows which must be converted to their present value (see [2] for some examples). More in general, the discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it was received immediately. If $\gamma = 0$, the agent acts in a short-sighted manner: since all rewards are seen as zero except for R_{t+1}, it tries only to maximize the next immediate reward by correctly choosing A_t.
This would heavily simplify the original RL goal, because now the multistage decision problem is decomposed in a sequence of single-step optimization problems, so that such a "myopic" agent could maximize the total return by separately maximizing each immediate reward. However, in practice this approach does not work, since these problems are not independent, i.e., a decision (action) taken at a certain timestep influences the behavior of the environment at future timesteps.

2.4 Policies and Value Functions

What we really want in the end is that our agent behaves in a way such that, at every state, the best action is taken by it. This strategy to pursue can be formally represented by a **policy** π, i.e. a distribution π over actions, conditioned on the present state; that is,

Definition 2.4.1 (Policy)**.** *Given a Markov decision process, a* ***policy*** $\pi : S \times A \to [0,1]$ *is a function of the form*

$$\pi(a|s) = \mathbb{P}[a_t = a | s_t = s] \tag{2.6}$$

.

With this definition at hand, now we can state the **RL Objective**: an RL agent aims to learn a policy that maximizes the expected discounted return, obtained by following the very same policy:

Definition 2.4.2 (RL Objective)**.** *The goal of an RL agent is to learn a policy π that solves the following optimization problem:*

$$\max_{\pi} \mathbb{E}_{\pi}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)|s_0 \sim p_0, a_t \sim \pi(\cdot|s_t), s_{t+1} \sim P(\cdot|s_t, a_t)] \tag{2.7}$$

where p_0 is the distribution over the set of initial states.

Now we have a way of select actions; what about states? Indeed, some states should be preferable to others, in the sense that a wise agent should try to get to *good* states instead of *bad* ones. A formal definition of goodness of a state is the **state value function**:

Definition 2.4.3 (State value function)**.** *Given a policy π, the value function $V : S \to \mathbb{R}$ is defined as*

$$V_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s]. \tag{2.8}$$

where the expected value is computed provided that the agent at every time step follows the policy π (under this definition the value of a terminal state is zero).

Since we deal also with actions, sometimes it's more useful to talk about the **action-value function** (or just **q-function**), which measures the goodness of a state-action tuple:

Definition 2.4.4 (State-action value function)**.** *Given a policy π, the state-action value function (or q-function) is a function $Q : S \times A \to \mathbb{R}$ defined as*

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a]. \tag{2.9}$$

This is the expected return starting from s, taking the action a (even if not in the support of the policy) and thereafter following the policy π.
These two functions can also be defined by a recursive equation after some algebraic manipulation; for example, we can write

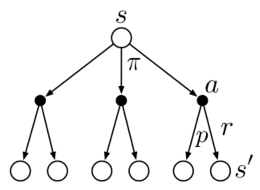
$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[G_t|s_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|s_t = s] && (2.10)\\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|s_{t+1} = s']] && (2.11)\\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V_\pi(s')] && (2.12) \end{aligned}$$

This is the **Bellman Equation** for V_π, which relates the value of a state to the value of its successor states, meaning that the value function under the policy can be decomposed into two parts:

- the immediate expected reward r starting from state s

- the discounted value of any successor state, weighted by the transition probability.

An analogous equation exists for the action-value function.
As we can see in Fig. 2.2, we can think of starting from state s and looking a step ahead; the agent can take some actions from $\mathcal{A}(s)$ with a nonzero probability, based on the current policy. From each of these state-action pairs, the environment transitions to some possible states s', with probabilities defined by its transition dynamics, yielding an immediate reward r. Each lookahead contribute in the Equation is used for computing the average outcome, with weights each given by its probability of occurring. What we have said holds for every policy; but, in the



Backup diagram for v_π

Figure 2.2: backup diagram for v_π: starting from state s as the root of the tree, we choose the action a according to the policy distribution $\pi(a|s)$; then, the environment updates its state to s' according to the transition probability $p(s'|s,a)$.

end, we are interested with the ones which yield highest rewards along time. Still, it turns out that value functions define a partial ordering over the set of feasible policies, that is

$$\pi \geq \pi' \iff V_\pi(s) \geq V_{\pi'}(s) \quad \forall s \in \mathcal{S}. \tag{2.13}$$

Now let

$$V^*(i) := \sup_\pi V_\pi(i). \tag{2.14}$$

We have the following:

Definition 2.4.5 (Policy optimality)**.** *A policy π^* is said to be γ-**optimal** if*

$$V_{\pi^*}(s) = V^*(s) \quad \forall s \in \mathcal{S}. \tag{2.15}$$

Hence, a policy is γ-**optimal** if its expected γ-discounted return is maximal for every initial state.
What's interesting about the optimal value function defined in (2.14), is that said function has their own special Bellman Equation, as stated by the following (proof is taken from [3])

Theorem 1 (Optimality Equation)**.**

$$V^*(s) = \max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V^*(s')] \tag{2.16}$$

Proof. Let π be any arbitrary policy that chooses action $a \in \mathcal{A}$ at time $t = 0$ with probability p_a. Then,

$$V_\pi(s) = \sum_a p_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma W_\pi(s')], \tag{2.17}$$

where $W_\pi(s')$ represents the the discounted return from $t = 1$ onward obtained by following policy π and starting from state s'. We clearly have $W_\pi(s') \leq \gamma V^*(s')$, thus

$$\begin{aligned} V_\pi(s) &\leq \sum_a p_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V^*(s')] && (2.18)\\ &\leq \sum_a p_a \max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V^*(s')] && (2.19)\\ &= \max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V^*(s')] && (2.20) \end{aligned}$$

Because π is arbitrary, we have

$$V^*(s) \leq \max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V^*(s')]. \tag{2.21}$$

Now, let a_0 be s.t. it maximizes the right member of (2.16) and let π be the policy which chooses a_0 in $t = 0$ and, if the next state is s', it follows a policy $\pi_{s'}$ s.t. $V_{\pi_{s'}}(s') \geq V^*(s') - \epsilon$. Hence,

$$\begin{aligned} V_\pi(s) &= \sum_{s'} \sum_r p(s'.r|s, a_0)[r + \gamma V_{\pi_{s'}}(s')] && (2.22)\\ &\geq \sum_{s'} \sum_r p(s'.r|s, a_0)[r + \gamma V^*(s')] - \gamma\epsilon, && (2.23) \end{aligned}$$

which, because $V^*(s) \geq V_\pi(s)$, implies that

$$V^*(s) \geq \sum_{s'} \sum_r p(s'.r|s, a_0)[r + \gamma V^*(s')] - \gamma\epsilon. \tag{2.24}$$

Hence, by the particular choice of a_0, we have

$$V^*(s) \geq \max_a \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V^*(s')] - \gamma\epsilon. \tag{2.25}$$

The proof is now completed given that ϵ is arbitrarily positive. □

Starting from this Equation, three important theorems can be proved (see [3]); we just state them without proofs:

Theorem 2 (The policy determined by the optimality equation is optimal)**.** *Let f be he stationary policy that, when the agent observes state s, it selects a maximizing action. Then $V_f(s) = V^*(s)$, and hence f is γ-optimal.*

Theorem 3. *V^* is the unique bounded solution of* (2.16)

Theorem 4. *For any policy π, V_π is the unique solution of*

$$V_\pi(s) = \sum_{s'} \sum_{r} p(s', r|s, \pi(s))[r + \gamma V_\pi(s')] \tag{2.26}$$

In the same way we can define an optimal action-value function Q^* that satisfies its own Bellman Optimality Equation.
For finite MDPs, the value function is just a vector and so to solve the Equation we must solve a system of piecewise linear equations, which can be solved by rather simple iterative methods. While this looks quite the trivial task, in practice it's actually not; for example, it can be that the state space is very large or we really don't know the transition probabilities (i.e., when we don't have a **model** of the environment). However, some difficulties are mitigated if we consider the Q-function: indeed, we can observe that

$$Q^*(s, a) = \sum_{s'} \sum_{r} p(s', r|s, a)[r + \gamma V^*(s')] \tag{2.27}$$

and

$$V^*(s) = \max_a Q^*(s, a). \tag{2.28}$$

Hence, we can rewrite the optimality equation as

$$Q^*(s, a) = \sum_{s'} \sum_{r} p(s', r|s, a)[r + \gamma \max_{\tilde{a}} Q^*(s', \tilde{a})] \tag{2.29}$$

If we compare this last result to Eq. (2.16), we have good and bad sides:

- since the Q-function depends on two variables instead of one like the value function, it seems that we have augmented the dimensionality, which may be not convenient.

- we have swapped expectation and optimization, which is good: it is easier to solve many simple (deterministic) optimization problems than a large (stochastic) one.

When V^* is known, it's easy to get an optimal policy: for each state s, there will be one or more actions which maximizes the Bellman Optimality Equation; so, every policy which assigns a nonzero probability only to those actions is optimal. Such policies are called **greedy** with respect to those actions, meaning that they select actions based only on short-term consequences. Since V^* already takes in account all the possible future reward behaviors, we have that just a one-step lookahead suffices to give optimal policies in the long term, since, thanks to V^*, the optimal expected return is turned into a quantity that is locally available in all states.
In this sense, with Q^* our life is even easier, since the agent does not require to look one step further in the future: given state s, it suffices to select the action $\tilde{a} = \arg\max_a Q^*(s, a)$; however, this comes at the cost of representing a function of state-action pairs instead of just states. The good side is that now the agent doesn't have to know anything about the environment's dynamics.
So, the "q-values" can be learned by statistical sampling, which is useful when the transition dynamics is not known: this leads to **model-free Reinforcement Learning**. Lastly, when dimensionality makes finding estimates for all state-action couples prohibitive, we can resort to a compact representation of the Q-function based on deep function approximators like **Deep Neural Networks**.

2.5 Generalized Policy Iteration

Generalized Policy Iteration (or **GPI** for short) refers to an algorithmic framework to find an optimal policy. Almost all RL methods can be classified as GPI instances.
GPI is the combination of two interacting processes: **Policy Evaluation** and **Policy Improvement**. The first process makes the value function consistent with the current policy; the second improves the policy in a greedy way, with respect to the current value function. In this way, at each step the current policy gets properly assessed in term of goodness of returns that can be obtained; then, we can refine such policy.
When the value function becomes consistent with the current policy, the evaluation process doesn't produce changes anymore; on the other hand, the improvement step halts when the policy becomes greedy with respect to the current value function. So, when both processes stabilize, the value function and policy must be optimal.
What's interesting to observe is that the two processes are adversarial, meaning that they pull in opposite directions: for example, when the policy changes, the value function is not consistent anymore. In the long run, however, they work to a joint solution, as we can see in Figures 2.3-2.4.

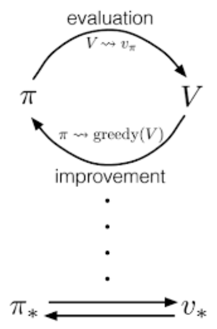

evaluation
$V \rightsquigarrow v_\pi$
π
V
$\pi \rightsquigarrow \text{greedy}(V)$
improvement
π_*
v_*

Figure 2.3: GPI algorithmic flow

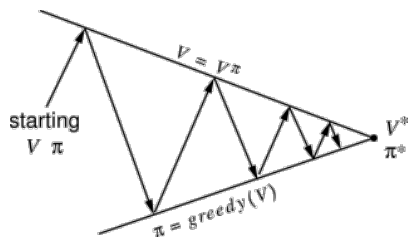


Figure 2.4: GPI iteration: the two processes converge to a common solution

2.6 QLearning

QLearning is a model-free algorithm (meaning that we don't have an approximation of the environment's dynamics) used to solve finite MDPs. The reality is that, in many applications, we actually don't know the transition probabilities. Hence, we have to learn by sampling from the environment. How to do it? Let's give a constructive explanation of the algorithm: supposing we are at iteration k, the agent has to choose action a_k and we have estimates $Q_{k-1}(s,a)$ which has been computed at the previous step. Supposing we are at state $s_k = i$, the action is selected in a greedy way:

$$a_k = \arg\max_a Q_{k-1}(i,a) \tag{2.30}$$

After applying this action, the environment yields:

- a next state $s_{k+1} = j$
- a reward r_{k+1}

Now, we can update the estimate $Q(s_k, a_k)$. Indeed, we have sampled an observation of a random variable that combines the short-term contribution (the reward) with the long-term behavior:

$$\tilde{q} = r + \gamma \arg\max_a \tilde{Q}_{k-1}(s_{k+1}, a) \tag{2.31}$$

Then, we can update the q-value for the pair (s_k, a_k) in this way:

$$\tilde{Q}_k(s_k, a_k) = \alpha\tilde{q} + (1-\alpha)\tilde{Q}_{k-1}(s_k, a_k) \tag{2.32}$$

This kind of estimation is called **exponential smoothing**, since we can roll it out recursively as:

$$\begin{aligned}
\hat{Q}_k &= \alpha\tilde{q}_k + (1-\alpha)\hat{Q}_{k-1} && (2.33)\\
&= \alpha\tilde{q}_k + \alpha(1-\alpha)\tilde{q}_{k-1} + (1-\alpha)^2\hat{Q}_{k-2} && (2.34)\\
&= \sum_{t=0}^{k-1} \alpha(1-\alpha)^k \tilde{q}_{k-t} + (1-\alpha)^k\hat{Q}_0 && (2.35)
\end{aligned}$$

We see that, unlike the standard sample mean, the weights of older observations have an exponential decreasing, controlled by a parameter α, which is called **learning rate**. We can express this concept in another way: we define a factor named **temporal difference**

$$\Delta_k = [r + \gamma \max_a \hat{Q}_{k-1}(j, a)] - \hat{Q}_{k-1}(i, a_k) \tag{2.36}$$

Then, the update is

$$\hat{Q}_k(s,a) = \begin{cases} \hat{Q}_{k-1}(s,a) + \alpha\Delta_k & \text{if } s = s_k \text{ and } a = a_k \\ \hat{Q}_{k-1}(s,a) & \text{otherwise} \end{cases} \tag{2.37}$$

From a GPI perspective, in QLearning we constantly changes the policy, which is implicitly defined by the q-values: therefore, we are so optimistic about the assessment of the current policy that we only do a policy improvement step (so no evaluation steps at all). Moreover, we can say that QLearning is an **off-policy** algorithm, since we apply one policy (the greedy one at step $k-1$) to learn about another one (the improved policy).
Finally, we can spend a few more words on the update: the temporal difference is also called **bootstrapping error**. If the q-function was the optimal one, then, in expectation with respect to the reward and next state, the term Δ_k should be 0, since the two terms of the difference should be equal by the Bellman Eq.2.29. Actually this doesn't happen, both because we take a sample instead of computing the full expectation and because the current q-function may not be optimal (otherwise the method would have already converged). So, in practice we have $\Delta_k \neq 0$ and, thanks to that, we can update our current estimates. The term "bootstrapping"

is used since the q-values are updated based only on the reward and the previous q-values.
QLearning was introduced by [4] in his PhD thesis in 1989; here a first proof on the convergence of this one-step QLearning approach was given.
The idea is that Q^* is the fixed point of the contraction operator $\mathbf{H}$, defined as

$$(\mathbf{H}q)(s,a) = \sum_{s'}\sum_{r} p(s',r|s,a)[r + \gamma \max_{a'} q(s',a')], \tag{2.38}$$

for a generic function

$$q : S \times A \longrightarrow \mathbb{R} \tag{2.39}$$

. This operator is also called the **Bellman Operator** $\mathcal{B}$; it is a contraction in the sup-norm, that is

$$\|\mathbf{H}q_1 - \mathbf{H}q_2\|_\infty \leq \gamma \|q_1 - q_2\|_\infty \tag{2.40}$$

The following lemma holds:

Lemma 1. *The random process $\{\Delta_t\}$ taking values in $\mathbb{R}^n$ and defined as*

$$\Delta_{t+1}(x) = (1-\alpha_t(x))\Delta_t(x) + \alpha_t(x)F_t(x) \tag{2.41}$$

converges to zero under the following assumptions:

- $0 \leq \alpha_t \leq 1$, $\sum_t \alpha_t(x) = \infty$ *and* $\sum_t \alpha_t^2(x) < \infty$
- $\|\mathbb{E}[F_t(x)|\mathcal{F}_t]\|_W \leq \gamma\|\Delta_t\|_W$, *with* $\gamma < 1$
- $Var[F_t(x)|\mathcal{F}_t] \leq C(1 + \|\Delta_t\|_W^2)$, *for* $C > 0$

Now we can state the Theorem concerning the convergence of the algorithm:

Theorem 5. *Given a finite MDP $(\mathcal{S}, \mathcal{A}, P, r)$, the QLearning algorithm given by the update rule*

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha_t(s_t,a_t)[r + \gamma \max_b Q_t(s_{t+1},b) - Q_t(s_t,a_t)] \tag{2.42}$$

converges almost surely to the optimal q-function as long as

$$\sum_t \alpha_t(s,a) = \infty \quad \textit{and} \quad \sum_t \alpha_t^2(s,a) < \infty, \tag{2.43}$$

for all $(s,a) \in \mathcal{S} \times \mathcal{A}$. Notice that, since $0 \leq \alpha < 1$, last two equations require all state-action pairs be visited infinitely often.

Proof. We start by rewriting Eq. (2.42) as

$$Q_{t+1}(s_t, a_t) = (1 - \alpha_t(s_t, a_t))Q_t(s_t, a_t) + \alpha_t(s_t, a_t)[r_t + \gamma \max_b Q_t(s_{t+1}, b)] \quad (2.44)$$

Subtracting from both sides the term $Q^*(s_t, a_t)$ and letting

$$\Delta_t(s_t, a_t) = Q_t(s_t, a_t) - Q^*(s_t, a_t) \quad (2.45)$$

yields

$$\Delta_t = (1 - \alpha_t(s_t, a_t))\Delta_t(s_t, a_t) + \alpha_t(s_t, a_t)[r_t + \gamma \max_b Q_{t+1}(s_t, b) - Q^*(s_t, a_t)] \quad (2.46)$$

If we write

$$F_t(s, a) = r(s, a, X(s, a)) + \gamma \max_b Q(s', b) - Q^*(s, a), \quad (2.47)$$

where $X(s, a)$ is a random sample state obtained from the Markov chain $(\mathcal{S}, P_a)$, we have

$$\begin{aligned} \mathbb{E}[F_t(s, a)|\mathcal{F}_t] &= \sum_{s'} p(s', r|s, a) && (2.48) \\ &= [r + \gamma \max_b Q_t(s', b) - Q^*(s, a)] && (2.49) \\ &= (\mathbf{H}Q_t)(s, a) - Q^*(s, a) && (2.50) \end{aligned}$$

Using the fact that $\mathbf{H}Q^* = Q^*$,

$$\mathbb{E}[F_t(s, a)|\mathcal{F}_t] = (\mathbf{H}Q_t)(s, a) - (\mathbf{H}Q^*)(s, a). \quad (2.51)$$

It is now immediate from the contraction property that

$$\|\mathbb{E}[F_t(s, a)|\mathcal{F}_t]\|_\infty \le \gamma\|Q_t - Q^*\|_\infty = \gamma\|\Delta_t\|_\infty. \quad (2.52)$$

Finally,

$$\begin{aligned} \mathbf{var}[F_t(s, a)|\mathcal{F}_t] &= && (2.53) \\ &= \mathbb{E}[(r + \gamma \max_b Q_t(s', b) - Q^*(s, a) - (\mathbf{H}Q)(s, a) + Q^*(s, a))^2] && (2.54) \\ &= \mathbb{E}[(r + \gamma \max_b Q_t(s', b) - (\mathbf{H}Q)(s, a))^2] && (2.55) \\ &= \mathbf{var}[r + \gamma \max_b Q_t(s', b)|\mathcal{F}_\sqcup] && (2.56) \end{aligned}$$

which, due to the fact that r is bounded, clearly verifies

$$\mathbf{var}[F_t(s)|\mathcal{F}_t] \le C(1 + \|\Delta_t\|_W^2) \quad (2.57)$$

for some constant C. Then, previous Lemma, Δ_t converges a.s. to *zero*, i.e., Q_t converges to Q^* a.s. □

QLearning is a building block of many complex algorithms in Deep RL, as we shall see in the next chapters.

2.6.1 State vs Observation

In this section we make an important distinction concerning the **observability** of the environment.
Sometimes, in certain applications, the agent isn't able to directly access the full state representation of the environment, which acts as a "black-box"; instead, the environment just yields a compact state representation called **observation**, which does not need to be in a one-to-one correspondence with the internal state, meaning that different hidden states can be mapped into the same observation. For example, this often happens when working with image-based tasks, as represented in Fig. 2.5. This means that we must modify the notion of MDP to take in account this

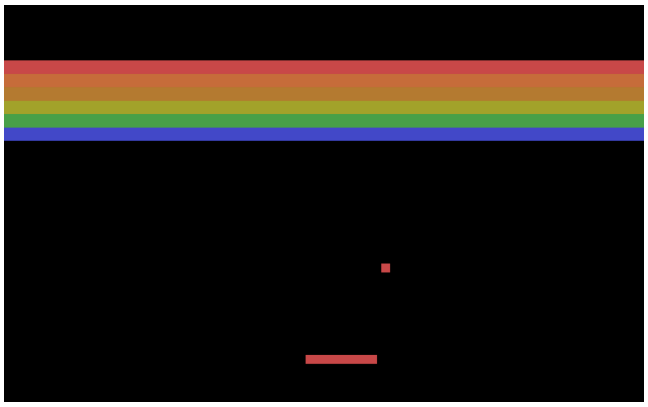


Figure 2.5: Atari Breakout: this system has partial observability because, if the agent is only supplied with the image, it is not able to predict the direction and the verse of the ball's velocity

obsrvability problem. Formally, we have the following definition:

Definition 2.6.1 (Partially Observable MDP)**.** *A* ***POMDP*** *(Partially Observable MDP) is a tuple* $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \mathcal{Z}, \mathcal{O}, \gamma >$*, where* $\mathcal{O}$ *is the set of observations and* $\mathcal{Z}$ *is the observation conditional distribution, that is*

$$\mathcal{Z}(o, s, a) := \mathbb{P}[O_{t+1} = o | S_{t+1} = s, A_t = a]. \tag{2.58}$$

So, partial observability blurs the idea of a current state, with the consequence that the rule of selecting actions based on the current state is no longer valid. While the hidden transition dynamics is still Markovian, since we don't have access to the internal state of the system we should keeping track of the entire history of the process, meaning that the Markov Property is no longer valid for the agent-observer. Equivalently, instead of representing the whole history explicitly, we can define a probability distribution over the state space conditioned on the current history and, when we perform an action and make an observation, we update the distribution. This distribution is called the **belief state**; formally,

$$b(s;h) \coloneqq \mathbb{P}[S_t = s|h], \tag{2.59}$$

where $h \coloneqq (o_1, o_2, \ldots, o_t)$ is the history up to time $t+1$. Luckily, we can find that the process defined over the belief state space is Markovian, which is equivalent to say that the next belief state depends only on the current belief state (and the current action and observation). This means that we can convert a discrete POMDP problem into a continuous MDP one and we can apply the methods previously illustrated.

2.6.2 Maximization Bias in QLearning

As we have seen, QLearning involves a maximization operation for computing the target $\tilde{q}$ in the update step. That is, a maximum of the q-values over all possible actions given a certain next state s_{k+1} is computed, and used implicitly to estimate the state-action value function in the tuple (s_k, a_k). This gives rise to a problem, because this operation can lead to positive bias.

For example, in Fig. 2.6, we have depicted an MDP with two nonterminal states (example is taken from [1]). Episodes always start in A with two possible action choices: $left$ or $right$. When the agent chooses $right$, the environment updates the state to a terminal one with zero reward and return. On the other hand, the $left$ action also yields a reward of zero, but now from the state B many different action choices are possible, all with the same reward distribution, which is normal with mean -0.1 and variance 1. This means that, when in A, the *right* action is superior on average to the $left$ action in terms of total return. However, due to sampling error and maximization bias, the $left$ action might appear more promising. As a matter of fact, in the Figure we can also observe that QLearning with an $\epsilon-$greedy strategy (which means that the action is uniformly random selected with probability ϵ and chosen in the usual way with probability $1-\epsilon$) initially learns to strongly favor the $left$ action in this example. One possible solution to the maximization bias is the **Double Learning**. Since using the maximum of the estimates to obtain the estimate of the maximum of the true values can lead to positive bias, one way to view the problem is to see that we use the same samples both to determine the best action and its estimated value; so, we can try to disentangle these two processes, meaning that we divide the samples into two sets for two different Q-value estimates Q_1 and Q_2. Then, one estimate is used to determine the maximizing action, while the other used to estimate its value.
When we couple this idea with the QLearning algorithm we obtain **Double QLearning**: at each timestep one of two estimates is randomly selected (suppose Q_1) and updated; after that, the maximizing action is extrapolated from that

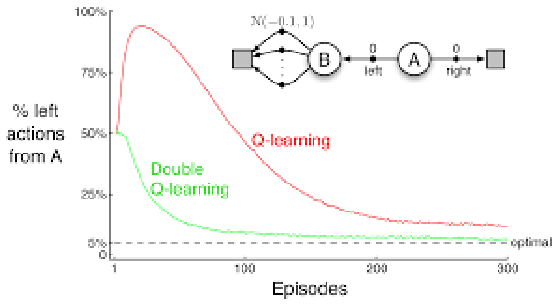


Figure 2.6: MDP example: in this Figure the MDP is represented with its transition and reward functions; moreover, in the graph is shown how, as a consequence of maximization bias, QLearning highly prefers the *left* action during the first part of training; what's more, we can observe how it takes time for the algorithm to abandon this incorrect belief. On the other hand, when Double QLearning is employed, the training proceeds smoothly, as the *left* action is chosen with low probability already after a few episodes

estimate (Q_1 again), while the value of the action is estimated by the other q-function estimate (suppose Q_2). In formulas, we have

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha[r(s,a) + \gamma Q_2(s', \arg\max_{a'} Q_1(s',a')) - Q_1(s,a)] \tag{2.60}$$

The estimator obtained in this way is unbiased, in the sense that

$$\mathbb{E}[Q_1(\cdot, a^*)] = q(\cdot, a^*) = \mathbb{E}[Q_2(\cdot, a^*)]. \tag{2.61}$$

Double QLearning was introduced for the first time in [5].
As we said, the q-function update in QLearning can give us some problems since we try to estimate the maximum expected value over actions using the maximum action value. The idea of the paper is to use a *double estimator*, that is, trying to decouple the action selection from the estimate of the q-function on the action.
Generally speaking, we can think of the problem of estimating the maximum of the expected values of N variables $X_1, X_2 \dots, X_N$. If we use a *single estimator* approach in the style of QLearning, we define N estimators $\hat{X}_1, \hat{X}_2, \dots, \hat{X}_N$ with means μ_i, $i = 1,2,\dots,N$. The *maximal estimator* $\mu := \max_i \mu_i$ is obviously an unbiased estimate for $\mathbb{E}[\mu]$.
Given the PDFs and CDFs for the estimators μ_i by, respectively, the notation f^i_μ, F^i_μ, we can recover the CDF for μ as

$$F_\mu(x) = \mathbb{P}(\max_i \mu_i \leq x) = \prod_{i=1}^N \mathbb{P}(\mu_i \leq x) \prod_{i=1}^N F^i_\mu(x). \tag{2.62}$$

Then, we can express $\mathbb{E}[\mu]$ as

$$\mathbb{E}[\mu] = \int_{-\infty}^{\infty} x f_\mu(x) dx \tag{2.63}$$

$$= \int_{-\infty}^{\infty} x \frac{d}{dx} \prod_i F_\mu^i(x) dx \tag{2.64}$$

$$= \sum_j \int_{-\infty}^{\infty} x f_\mu^j(x) \prod_{i \neq j} F_\mu^i(x) dx. \tag{2.65}$$

So, this estimator is unbiased for $\mathbb{E}[\max_i \mu_i]$, but positively biased for $\max_i \mathbb{E}[\hat{X}_i]$; that's where the overestimation comes from.
The alternative method is to use two different estimators for each variable, which we group into two sets $\mu^A = \{\mu_1^A, \dots, \mu_N^a\}$ and $\mu^B = \{\mu_1^B, \dots, \mu_N^B\}$.
Now, if a^* is an index that maximizes μ^A, i.e., $\mu_{a^*} = \max_i \mu_i^A$, then we can use this index to choose in the set μ^B the estimate $\mu_{a^*}^B$ for $\max_i \mathbb{E}[X_i] = \max_i \mu_i^B$. Moreover, we can recover the expected value of this estimator, since

$$\sum_j \mathbb{P}(j = a^*) \mathbb{E}[\mu_j^B] = \sum_j \mathbb{E}[\mu_j^B] \int_{-\infty}^{\infty} f_j^A(x) \prod_{j \neq i} F_i^A(x) dx. \tag{2.66}$$

Eqs. 2.66 and 2.63 express the means of the two estimators. We can see that the second uses $\mathbb{E}[\mu_j^B]$ in lie of x; so it's an underestimation, since the probabilities $\mathbb{P}(j = a^*)$ represent the coefficients of a convex linear combination of expected values which are lower or equal to the maximum one which is being estimated. On the other hand, the single estimator tends to overestimate, since x is positively correlated with the monotonically increasing product of CDFs inside the integral.
Also Double QLearning converges in the limit to the optimal policy in a finite MDP: (the following theorems can be found in [5])

Lemma 2. *Consider a stochastic process $(\zeta_t, \Delta_t, F_t), t \geq 0$ where $\zeta_t, \Delta_t, F_t : X \to \mathbb{R}$ satisfy the equations:*

$$\Delta_{t+1}(x_t) = (1 - \zeta_t(x_t))\Delta_t(x_t) + \zeta_t(x_t) F_t(x_t), \tag{2.67}$$

where $x_t \in X$. Let P_t be a sequence of increasing $\sigma - fields$ s.t. Δ_0 and ζ_0 are $P_0 - measurable$ and $\zeta_t, \Delta_t, F_{t-1}$ are $P_t - measurable$. Assume that the following hold:

- *X is finite;*
- *$\zeta_t(x_t) \in [0,1], \sum_t \zeta_t(x_t) = \infty, \sum_t \zeta_t(x_t)^2 < \infty w.p.1$ and $\forall x_t : \zeta_t(x) = 0$*
- *$||\mathbb{E}[F_t|P_t]|| \leq \kappa ||\Delta_t|| + c_t$, where $\kappa \in [0,1)$ and c_t converges to zero w.p.1.*

- $Var[F_t(x_t)|P_t] \leq K(1 + \kappa||\Delta_t||)^2$, *for some constant* K. *Here* $||\cdot||$ *denotes a maximum norm. Then,* Δ_t *converges to zero almost surely.*

Having this Lemma, the next Theorem follows:

Theorem 6. *In an ergodic MDP s.t.*

- $\gamma \in [0,1)$;
- $Q_1 and Q_2$ *receive an infinite number of updates;*
- $\alpha_t(s_t, a_t) \in [0,1], \sum_t \alpha_t(s_t, a_t) = \infty, \sum_t \alpha_t(s_t, a_t)^2 < \infty$ *w.p.1 and* $\forall(s,a) \neq (s_t, a_t) : \alpha_t(s,a) = 0$;
- $\forall s, a, s' : Var[\mathbb{E}[R(s', a, s)]] < \infty$;

both Q_1 *and* Q_2 *will converge to the optimal q-function* Q^* *almost surely if an infinite number of experiences in the form of rewards and state transitions for each state-action pair are given by a proper learning policy. (e.g., a random policy)*

2.7 Model-Based RL

Up to now, we have seen RL methods belonging to the **model-free** family: this kind of methods use the information leveraged from the interaction with the environment only to update their critic, which can be an exact tabular representation of the value function, or just some deep approximator. However, if on one hand solutions like QLearning are created to overcome the problem of not knowing the real transition dynamics probabilities, another idea could be starting with an approximated representation of these distributions, and try to refine them online as more information comes from the environment. This is the main feature of **model-based** RL algorithms.
We can think of methods belonging to these families as two possible approaches to the problem of visiting a foreign city: since we don't know well the city, one way to go around without getting lost (let's say from point A to point B) could be to use some reference points (like a big square with a fountain) and remember what route to take from there. This is equivalent to employ a state (or state-action) function, assigning a certain value to certain positions.
Another method could be to use a map: we could buy one (really accurate but expensive solution, maybe unfeasible) or drawing one: if we go with the latter choice, at first our map will be really approximative; however, as more information comes in, we can refine it and use it to simulate the real world. Meaning, if we get stuck in a point, we can trace back our steps or even look ahead to find the right trajectory.

In practice, what we actually want to represent is how the environment changes its state in response to the agent's action. Given a state and an action, a model produces a prediction of the resultant next state (or a distribution over the possible next states, and possibly the reward: the part dealing with the states is the *transition function*, while the second is the *reward function*.
Why a model of the environment should be useful? For starting, a model can be used to simulate *experience*. Given a starting state and action, a model generates all possible transitions weighted by their probabilities of occurring. Given a starting state and a policy, a model could generate all possible episodes and their probabilities. So, in the classic model-based learning cycle the interaction with the real world is used to learn the model; in turn, the model is used to plan through it generating a value function/policy which is used to interact in the real world, gaining new experience, so that we can learn a better model.
An example which conveys the idea of model-based usefulness is the game of chess: in this case, the state space has a size of approximately 10^{48}! Moreover, moving one piece from its square to an adjacent one can completely change the position from winning to losing; this means that the optimal value function is really sharp, so learning that (or directly the policy) is really hard. On the contrary, the dynamics is quite easy to represent, since chess rules are simple and transitions are deterministic. So, if I can use the model to look ahead, I'm able to estimate the value function by clever tree search strategies (planning). So, an advantage of model-based RL is that model can be a more useful representation of the information of the environment than value function. One another advantage is that model can be learned by supervised learning; moreover, the model is also useful if you want to reason about *model uncertainty*, which is what you don't know about the environment and what *you don't know you don't know*; in other words, you want to understand the world better. With the model, you can choose actions to reach regions of the space you don't know well.
One disadvantage is that we use an approximate model to learn an approximate value function, so now there are two sources of error.

Model-free RL:

- No model
- Learn value function (and/or policy) from experience

Model-based RL:

- Learn a model from experience
- Plan (lookahead using the model) value function (and/or policy) from the model

In the end, what we want to learn is a transition function $\hat{P}_\Psi[\cdot|s,a]$ which approximates the one of the real environment, which is $P[\cdot|s,a]$. In this way we have two ways of generating experience:

- **real experience**: sampled from the environment (true MDP) $s' \sim P[\cdot|s,a]$
- **simulated experience**: sampled from the model (approximate MDP) $s' \sim \hat{P}_\Psi[\cdot|s,a]$

2.7.1 DYNA: model-free + model-based

Dyna is an algorithmic framework developed by Richard Sutton (see [6]) which is structured in this way:

- learn model from real experience
- learn and plan value function (and/or policy) from real **and** simulated experience

Real experience can be exploited by a planning agent in two different ways: it can be used to update and improve the model, or to directly improve value function/ policy via model-free approaches (*direct* RL). This is summarized in Fig. 2.7: each arrow shows a relationship of influence and presumed improvement. The algorithm can be summarized as following (see Fig.2.8): once the q-function and transition model are initialized, we enter the main loop, in which, given a state, an action is selected by an **ϵ-greedy** policy. Subsequently, the action is executed and the environment transitions to the next state s' with a reward r; we end this process by updating our q-function. So far, this looks like plain QLearning. However, at this point we enter an inner loop which exploits the learned model: at every inner step, given state s, we choose a random action from those that were previously taken when encountered state s and, after that, we output a next state s' and a reward r using the learned model. Then, we update the q-function and the step ends. Model-free and model-based approaches both have advantages and disadvantages. While the former are much simpler in their design and are not affected by biases due to model learning, model-based methods usually make fuller use of a limited amount of experience and thus achieve a better policy with fewer interactions with the environment, as we can see in Fig. 2.9: as the agent is allowed more "thinking time" (the number N of planning steps), we observe that the number of steps required to reach the end of an episode sharply decreases and reaches its minimum after few training steps, contrarily to the case where the agent is not allowed to "think" at all (pure model-free method).

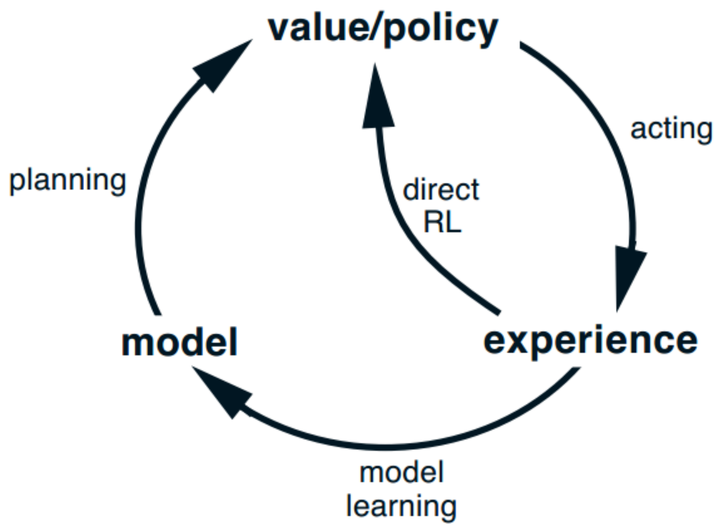
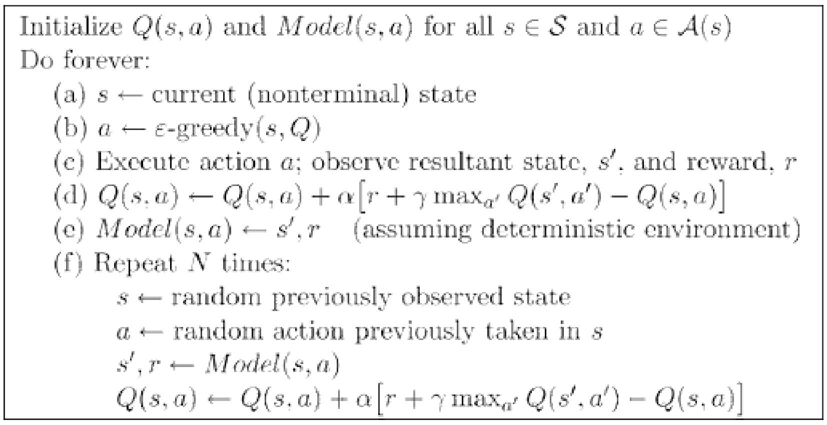


Figure 8.1: Relationships among learning, planning, and acting.

Figure 2.7: Algorithm flow of Dyna



```
Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
    (a) $s \leftarrow$ current (nonterminal) state
    (b) $a \leftarrow \varepsilon$-greedy$(s,Q)$
    (c) Execute action $a$; observe resultant state, $s'$, and reward, $r$
    (d) $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
    (e) $Model(s,a) \leftarrow s', r$    (assuming deterministic environment)
    (f) Repeat $N$ times:
            $s \leftarrow$ random previously observed state
            $a \leftarrow$ random action previously taken in $s$
            $s', r \leftarrow Model(s,a)$
            $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
```

Figure 2.8: Dyna algorithm

2.8 Deep Learning

2.8.1 Deep Function Approximators

The way we presented QLearning gives us a ready-to-use recipe to compute the state-action value function when the state-action space $\mathcal{S} \times \mathcal{A}$ can be represented

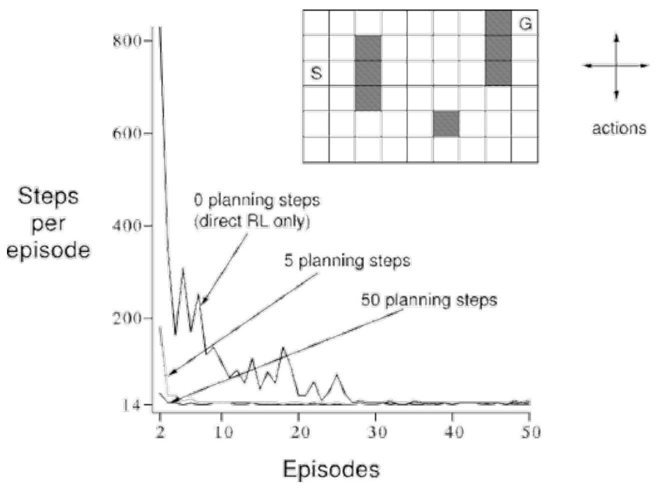


Figure 2.9: Sample efficiency of Dyna: as we can see, leveraging the predictive power of an accurate model really really contributes to the decrease of the number of steps per episode.

in a tabular way; in order for this to be possible, this space must be discrete and sufficiently small. However, in many practical scenarios, we have the very opposite situation: for example, think about a task which requires our agent to work with images of 200×200 pixels. If an element of the space (i.e. an image) is **RBG** encoded, this means that the cardinality of just the state space is $|\mathcal{S}| = (256^3)^{200\times200}$! This is a pretty huge number: in fact, it's bigger than the number of atoms in the universe.
As a second example, even if we could theoretically deal with this memory problem, there's nothing we can do when the state (and/or action) space is continuous.
A solution is therefore to resort to **approximators**, i.e. interpolation functions: for example, we can discretize the problem in some way and apply an algorithm of choice to the discretized problem; then, we can interpolate with these functions (for example we can use cubic splines) and compute the final policy in the original continuous space. At present days, the common choice is to resort to **deep approximators**.
A major breakthrough of the last decade in AI has been the "re-discovery" of **Artificial Neural Networks**: the evolution of hardware capabilities and dedicated software libraries has made possible to succesfully solve tasks in many fields like text/speech recognition, computer vision and so on. A rigorous presentation of neural networks can be found in [7]; basically, we can define a neural network in the following way: we start by defining its components and then we give the final definition of artificial neural network.

Definition 2.8.1 (Topology)**.** *A topology for a neural network is an ordered pair*

(F, I), where

- *F is the framework, an ordered pair $(L, \{N_1, \ldots, N_L\})$, where $L \in \mathbb{N}$ is the number of layers and $\{N_1, \ldots, N_L\}$ is the sequence of number of neurons per layer;*
- *I is the **interconnection scheme**, which is a set of relations between sets of source neurons $\{\omega_{l,i}\}$ and terminal neurons $\{n_{m,j}\}$.*

Definition 2.8.2 (Constraints)**.** *The **constraints** on a neural net form a tuple (C_W, C_Θ, C_A), where $C_W \subset \mathbb{R}$ defines the value ranges for the neurons' weights, while $C_\Theta \subset \mathbb{R}$ and $C_A \subset \mathbb{R}$ represent respectively the value ranges for the layer biases and for the activation functions.*

Definition 2.8.3 (Initial State)**.** *The **initialization state** for a neural net is a triple $(W(0), \Theta(0), A(0))$, where $W(0)$ is the initial weights state, while $\Theta(0) = \{\theta_{l,i} \in C_Theta : 1 \leq l \leq L, 1 \leq i \leq W_l\}$ and $(0) = \{a_{1,i} \in C_A : 1 \leq i \leq N_1\}$.*

Definition 2.8.4 (Transition Function)**.** *A **transition function** of a neural net is a 4-tuple (nf, lr, cf, of) where:*

- *nf is the neuron function, which specify the output of a neuron given its inputs;*
- *lr is the learning rule, which defines how weights get updated;*
- *cf is the clamping function, which defines how the output signal of each neuron gets clamped;*
- *af is the ontogenic function, which specifies any change in the topology of the network;*

Definition 2.8.5 (Artificial Neural Network)**.** *An **artificial neural network** is a 4-tuple containing its topology, its constraints, its initial state and its transition function.*

In practice, if we want to give an informal idea of what a neural network is, we can say that a neural network is a function $f_\theta : \mathcal{X} \to \mathcal{Y}$ (or better, a composition of functions), depending on a vector of parameters $\theta \in \Theta$, where Θ is the **parameter space**.
A basic example of neural network is the **MultiLayer Perceptron** (**MLP** for short), which is depicted in Fig. 2.10: it consists of multiple **layers** of **neurons** each **fully connected** to those in the subsequent and precedent layers; in the case depicted the net has three inputs, two outputs and one hidden layer containing four hidden units. Fully connected means that the input vector goes as input to

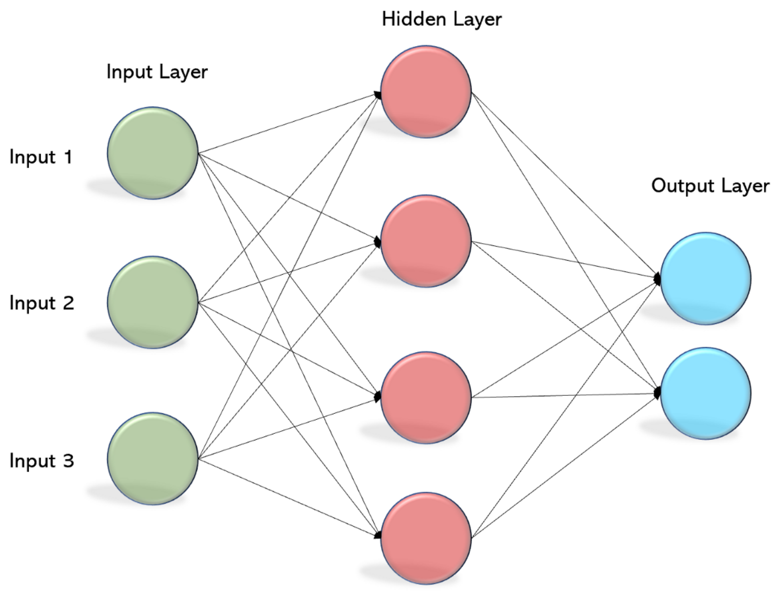


Figure 2.10: Scheme of MLP: each neuron of the Input Layer (in green), receives the input sample, which passes through a pipeline containing an affine transformation and a nonlinear activation function (similar to a filter in signal analysis); subsequently, the output serves as input for the Hidden Layer(s) (in red), and finally the Output Layer (in cyan) produces the result.

each hidden unit, that is: given the input vector $\mathbf{x} = (x_1, x_2, \ldots, x_d)$ (where $d = 3$ is the dimension of $\mathbf{x}$); given the matrix of hidden-layer weights $\mathbf{W}^1 \in \mathbb{R}^{d \times h}$ (where $h = 4$ is the number of hidden units), the *bias* vector $\mathbf{b}^1 \in \mathbb{R}^h$, we have that the output of the hidden layer is

$$\mathbf{h} = \sigma(\mathbf{W}^1\mathbf{x} + \mathbf{b}^1)\ . \tag{2.68}$$

In order to realize the potential of multilayer architectures, we add another element, which is the nonlinear **activation function** $\sigma : \mathbb{R} \to \mathbb{R}$, which is applied element-wise to each hidden unit following the affine transformation. Finally, the result of the output layer is

$$\mathbf{o} = \mathbf{W}^2\mathbf{h} + \mathbf{b}^2, \tag{2.69}$$

where $\mathbf{W}^2 \in \mathbb{R}^{h \times q}$ and $\mathbf{b}^2 \in \mathbb{R}^q$ are the output-layer weights and biases (here we have $q = 2$). More general MLPs can be designed by stacking many hidden layers (each with its own activation function) one on top of the other.
Although the MLP proposed seems somewhat naive, it is known that it works as a **universal approximator**: even with a single hidden-layer network, given enough nodes (possibly absurdly many), and the proper set of weights, we can model any

function, though actually learning that function is the hard part.
There are mainly three ways in which we can employ this technology in designing succesful **Deep RL** algorithms (see Fig. 2.11):

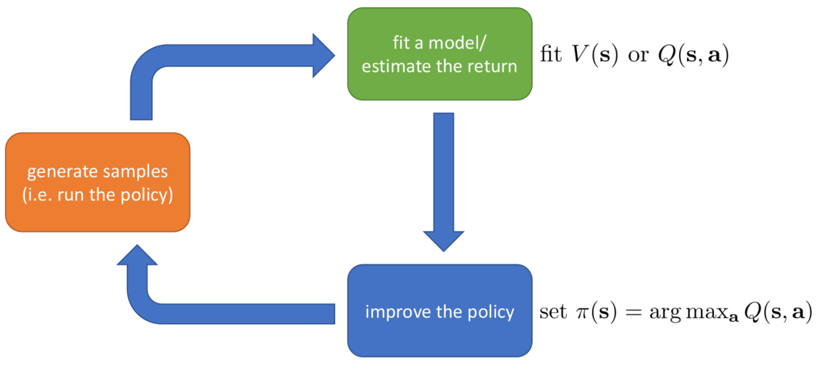

fit a model/
estimate the return
fit $V(\mathbf{s})$ or $Q(\mathbf{s},\mathbf{a})$
generate samples
(i.e. run the policy)
improve the policy
set $\pi(\mathbf{s}) = \arg\max_{\mathbf{a}} Q(\mathbf{s},\mathbf{a})$

Figure 2.11: Scheme 1: pipeline of an abstract value-based algorithm.

- **value-based** algorithms: a neural network is used to approximate either the state value function **V** or the state-action value function **Q**; the policy then is defined and updated implicitly as $\pi(s) = \arg\max_a Q(s, a)$. Indeed, by definition, the value function estimates the expected return obtained by following the policy π.
  Along the fashion of QLearning algorithm, we can turn Eq.2.29 into a learning algorithm, by minimizing the difference between the left-hand side and right-hand side of this equation with respect to the parameters θ of a parametric Q-function estimator $Q_\theta(s, a)$, for example by taking gradient steps on the Bellman error objective, as we will show in the next subsection.

- **policy gradient** algorithms: this is a class of intuitive methods to optimize the RL objective in Eq.2.5 by directly estimating its gradient. The method is to parametrize the policy $\pi_\theta(\cdot|s)$ by means of a neural network that outputs a distribution for the actions conditioned on the state s. In this way, the

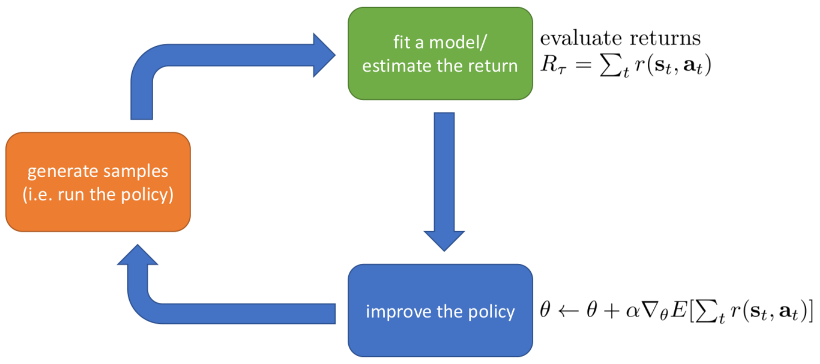

fit a model/
estimate the return
evaluate returns
$R_\tau = \sum_t r(\mathbf{s}_t, \mathbf{a}_t)$
generate samples
(i.e. run the policy)
improve the policy
$\theta \leftarrow \theta + \alpha \nabla_\theta E[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)]$

Figure 2.12: Scheme 2: pipeline of an abstract policy gradient algorithm.

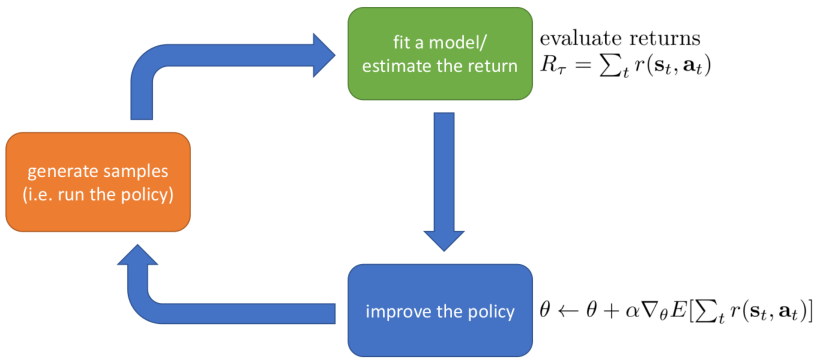

fit a model/
estimate the return
evaluate returns
$R_\tau = \sum_t r(\mathbf{s}_t, \mathbf{a}_t)$
generate samples
(i.e. run the policy)
improve the policy
$\theta \leftarrow \theta + \alpha \nabla_\theta E[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)]$

Figure 2.13: Scheme 3: pipeline of an abstract actor-critic algorithm.

gradient of the objective w.r.t. θ is

$$\nabla_\theta G(\pi_\theta) = \mathbb{E}_{\tau \sim p_{\pi_\theta}(\tau)}\left[\Sigma_{t=0}^{T}\gamma^t \nabla_\theta \log \pi_\theta(a_t|s_t) \underbrace{\left(\Sigma_{t'=t}^{T}\gamma^{t'-t} r(s_{t'}, a_{t'}) - b(s_t)\right)}_{\text{return estimate } \hat{A}(s_t, a_t)}\right], \tag{2.70}$$

where $\hat{A}$ can be learned either as a separate network or estimated by Monte Carlo sampling by generating trajectories from $p_{\pi_\theta(\tau)}$, while b is a suitable *baseline* (for example, we can set $b := V$): the usefulness of the baseline is to reduce the variance.
Indeed, this is because policy gradient suffers from high variance in term of the samples. One way of reduce variance is to introduce the principle of *causality*, which states that the action in the present doesn't affect the rewards obtained in the past: this is why in the return estimate we only consider time steps that come after t. Another trick is to use the baseline: in this way "bad" returns are negative and "good" rewards are positive; this doesn't change the value of the expectation since it can be proved that $\mathbb{E}[\nabla_\theta \log p_\theta(\tau) b] = 0$, but the variance gets reduced.

- **actor-critic** algorithms: an actor-critic method combines the ideas of the two ones shown before, by approximating with neural networks both the policy and the value function: in particular, the latter is used to try to compute the return estimate $\hat{A}$ in the policy gradient.
  While in QLearning we use samples to directly update our estimates of the optimal Q-function, in actor-critic methods we work on the Q-function defined by the policy π, that is the one defined in Eq.2.9.
  Actor-critic follows the philosophy of Policy Iteration: the policy evaluation phase consists of computing the q-function for the current policy π, for example via gradient updates. Then, in the policy improvement phase the next iterate of the policy is computed as the one which maximizes the q-function at each state (greedy update), or by using a gradient update procedure.

.

2.8.2 DQN

DQN (**D**eep **Q-N**etwork) is the most common value-based online algorithm for Deep RL and it has been introduced in 2015 by **DeepMind** research group [8].
It was the first algorithm employing neural networks in Reinforcement Learning to be stable.
In this work, a neural network is used to model the q-function and trained by adjusting its parameters θ to reduce the mean-squared error in the Bellman Equation.

In fact, since we don't know the transition dynamics, we can't work with the exact Bellman Operator $\mathcal{B}$, operatively defined as

$$(\mathcal{B}Q)(s,a) \coloneqq r(s,a) + \gamma \cdot \mathbb{E}_{s' \sim P(\cdot|s,a)}[\max_{a'} Q(s',a')], \tag{2.71}$$

we must work with an empirical version of $\mathcal{B}$ obtained by sampling data from the environment: this approximation is the **empirical Bellman operator $\hat{\mathcal{B}}$**, which is basically the same approach taken in simple QLearning in Section 2.6.
The network used to model the q-function is called **Q-Network**; it takes as input a state and it yields as output the q-values $Q_\theta(s,\cdot)$, for every action $a \in \mathcal{A}$. At each training step, assuming we have a sample batch of n transitions $\{(s_i, a_i, r_i, s'_i)\}_{i=1}^n$ (here we use s'_i as notation for the next state seen after s_i to not confuse it with the state s_{i+1} of the next transition in the sample batch), we compute the target for each sample as

$$y_i = r_i + \gamma \cdot \max_a Q_\theta(s'_i, a); \tag{2.72}$$

then, the loss is simply the MSE over the targets:

$$L(\theta) \coloneqq \frac{1}{n} \sum_{i=1}^{n} [y_i - Q_\theta(s_i, a_i)]^2. \tag{2.73}$$

It must be observed that in Eq. 2.72, the operator implicitly used is $\hat{\mathcal{B}}$. Unfortunately, under these conditions the algorithm doesn't work properly; this is because:

- in Supervised Learning, we work under **i.i.d. hypothesis**, meaning that we assume the samples to be drawn independently from the same distribution. This assumption doesn't hold in RL, since transitions from the same trajectory are obviously not independent.

- in Supervised Learning, the gradient descent is performed w.r.t the gradient of the target; however, if we consider the gradient update of a basic value-based method, that is

$$\theta \leftarrow \theta - \alpha \frac{\partial Q}{\partial \theta} \Big(Q_\theta(s,a) - [r(s,a) + \gamma \max_{a'} Q_\theta(s',a')] \Big), \tag{2.74}$$

  we are not taking the gradient also w.r.t. the target value in squared parentheses $[\cdot]$. Moreover, the target *moves itself* during training as the q-function gets updated, with the risk of causing divergence during the gradient descent.

To solve the first problem, the idea was to use a **data buffer** (see Fig.2.14) , i.e., to collect the transitions observed during the interaction with the environment and store them in this buffer, which is called **Experience Replay**. In this way,

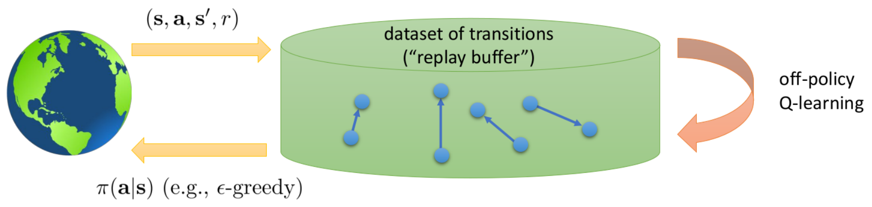
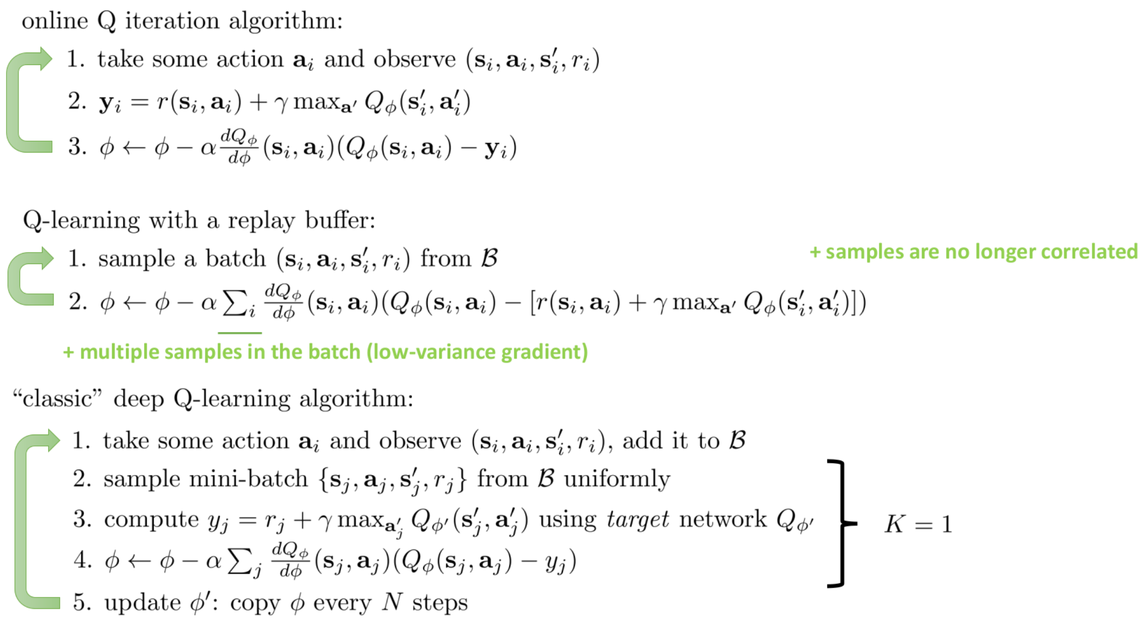

$(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$
dataset of transitions
("replay buffer")
off-policy
Q-learning
$\pi(\mathbf{a}|\mathbf{s})$ (e.g., ϵ-greedy)

Figure 2.14: DQN Buffer Replay: this data structure interacts with the environment, which at each step supplies the replay with the most recent transition; in turn, the replay adds the transition to its internal stack, so that it can be used for future batch samplings.

every time we have to update the q-function, we perform gradient descent by sampling a batch of transitions from the buffer and computing the target. Every transition is represented in the form (s, a, r, s'), where s' is the state to which the environment transitions to after the agent sees state s and takes action a. If we employ a sufficiently large stack (size is up to 10^6 in the paper), then sampling from experience is very close to i.i.d. sampling from a distribution over the state-action space. This contributes to decrease the variance of $L(\theta)$, ensuring algorithmic stability.
The second problem was addressed with the introduction of a second network called **target network**: since the moving target causes instability in training, the idea is to use *lagging targets* for a fixed number of training steps: this means that we define a second network which initially has the same parameters' values as the Q-Network; at training time, its parameters stay fixed for a number C of consecutive training steps, while the Q-Network gets updated; then, after C steps we copy the Q-Network's parameters into the target network and the process restarts. In Fig. 2.15, the three-steps evolution of this Q-iteration are shown: the first is the basic implementation, without buffer replay neither target network; in the second we add the buffer replay but still no target network, while the third one



online Q iteration algorithm:

```
1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$
```

Q-learning with a replay buffer:

```
1. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
2. $\phi \leftarrow \phi - \alpha \underline{\sum_i} \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$
```

+ samples are no longer correlated
+ multiple samples in the batch (low-variance gradient)

"classic" deep Q-learning algorithm:

```
1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. update $\phi'$: copy $\phi$ every $N$ steps
```

Figure 2.15: Evolution of the dqn algorithm, starting from a simple "Online Q-Iteration", up to the classic DQN implementation (Buffer Replay + Target Network), which has proven to work in many different tasks.

is the classic DQN implementation, which has actually been proved to work.
Since this is a QLearning-derived algorithm, it suffers from the maximization bias problem; the solution that has been taken in this direction is, even in this case, to apply Double QLearning: this means that, besides the Q-network, we employ another network (for easiness we can use the target network) to act as critic. So, the Q-network is used to evaluate the action and the target network is used to evaluate the value; in formulas, the target is computed as

$$y = r + \gamma Q_{\theta'}(s', \arg\max_{a'} Q_\theta(s', a')), \tag{2.75}$$

where θ' are the weights of the target network. This enhanced version of DQN is called **Double Deep Q-Network** (or **DDQN**); in this thesis work, the offline RL algorithms presented are built on top of this one.
Finally, a rigorous theoretical explanation of the success of DQN can be found in [9]. In the article, a theoretical foundation for DQN is provided: in particular, the mathematical motivations for the use of experience replay and target network are showed.

Chapter 3

Offline RL

As we have seen, standard RL provides an *online* learning paradigm, which involves the act of iteratively collecting experience by interacting with the environment. However, in practical applications, this approach can be unfeasible, either because data collection is expensive (e.g., industrial robotics) or dangerous (e.g., healthcare). Moreover, there are domains for which collected datasets already exist, so it would be more efficient to use those to leverage knowledge.
Actually, the main successes in the field of machine learning have come with the advent of scalable *data-driven* methods, which become better and better at their tasks when trained with more and more data (without incurring in *overfitting*); this has been possible also thanks to the advances in deep learning theory, which has produced numerous complex deep architectures, each one suitable for its own task.
In recent years, the question arose whether it's possible to develop completely *offline* RL algorithms, offline meaning that they are trained on previously collected datasets, without additional interaction with the environment; a high-level scheme of such methods is depicted in Fig. 3.1 .

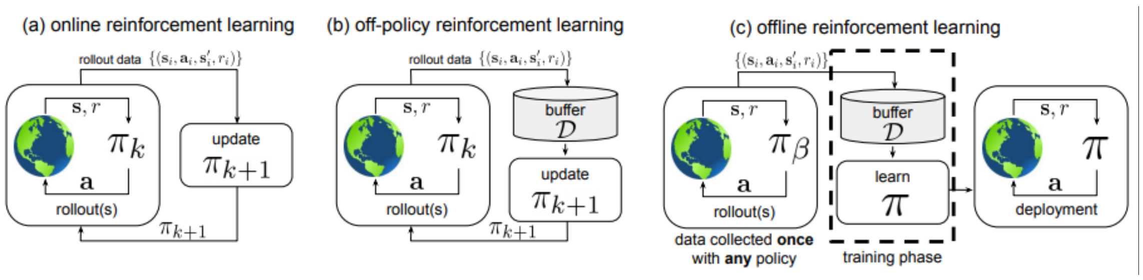

(a) online reinforcement learning
rollout data $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$
$\mathbf{s}, r$
π_k
$\mathbf{a}$
rollout(s)
update
π_{k+1}
π_{k+1}
(b) off-policy reinforcement learning
rollout data $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$
buffer
$\mathcal{D}$
update
π_{k+1}
(c) offline reinforcement learning
$\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$
π_β
data collected once with any policy
learn
π
training phase
deployment

Figure 3.1: Offline RL workflow: since no interaction with the environment is allowed, we use data collected from the world to train the agent, which is then deployed in the real environment

Of course, this change of paradigm doesn't come for free: unfortunately, such data-driven *offline* RL methods also pose major algorithmic challenges.
Formally, the end goal is still to optimize the objective in Eq.(2.7); however, the agent no longer has the ability to interact with the environment and collect additional transitions using any policy. Instead, the learning algorithm is provided with a *static* dataset of transitions $\mathcal{D} = \{(s_t^i, a_t^i, s_{t+1}^i, r_t^i)\}$, and must learn the best policy it can using the dataset. This is similar to what happens in supervised learning, therefore we can think of $\mathcal{D}$ as the **training** dataset.
Clearly, the training set has an inherent probability distribution over states and actions which we can call the **behavioral distribution** (or **behavior policy**) π_β; in this sense, states are sampled from $s \sim d^{\pi_\beta}(s)$ and $a \sim \pi_\beta(a|s)$.

3.1 Difficulties of Offline RL

The main practical challenge with offline reinforcement learning is **distributional shift**: if we want to train a policy which behavior at the end of training is more rewarding than the one of the policy (policies) used to collect data, our trained agent must learn new strategies and action trajectories, therefore distant from the ones that can be sampled from the dataset. This goes against the theoretical assumptions of classical machine learning, i.e. the **i.i.d. hypothesis**, which assume that data is independent and identically distributed; so, while in standard supervised learning we want our model to perform well (for example, with great accuracy) on data coming from the same distribution of the training set, this is quite the opposite of what we want to accomplish in offline RL: that is, we want our agent to learn a policy that does better (and therefore it's *different*) from the behavior seen in the dataset.
In a technical way, we say that the agent must make and answer to **counterfactual queries**: the agent must formulate hypotheses about what *might* happen if it were to choose actions different from the ones seen in the data. The problem is that with counterfactual queries, distributional shift arises: while our function approximator (for example the one representing the value function) is trained under one distribution, it will be evaluated on a different distribution, due both to the change in visited states by the new policy and by the act of maximizing the expected return (for example, when using algorithms in the style of QLearning).
How harmful can distributional shift be when trying to learn a policy?
In [10] we are provided with a theoretical example: suppose that at every state $s \in \mathcal{D}$ we are provided with optimal actions labels a^*. Under this assumption, we can try to bound the number of mistakes made by the learned policy $\pi(a|s)$ based

on this labeled dataset, as

$$l(\pi) = \mathbb{E}_{p_\pi(\tau)}[\sum_{t=0}^{H} \delta(a_t \neq a_t^*)] \tag{3.1}$$

If we train $\pi(a|s)$ with supervised learning (i.e. standard empirical risk minimization) on this labeled dataset, we have the following result:

Theorem 7 (Behavioral cloning error bound)**.** *If $\pi(a|s)$ is trained via empirical risk minimization on $s \sim d^{\pi_\beta}(s)$ and optimal action labels a^*, and attains generalization error ϵ on $s \sim d^{\pi_\beta}(s)$, then $l(\pi) \leq C + H^2\epsilon$ is the best possible bound on the expected error of the learned policy.*

So, even if this optimality assumption should hold, we get an error bound that is at best quadratic in the time horizon H. We can give an intuitive explanation of why this happens: at evaluation time, the learned policy $\pi(a|s)$, may enter into states that are far outside of its training distribution, since $d^\pi(s)$ may be very different from $d^{\pi_\beta}(s)$. In these out-of-distribution states, the generalization error bound ϵ no longer holds, since standard empirical risk minimization makes no guarantees about error when encountering out-of-distribution inputs that were not seen during training. Once the policy enters one of these out-of-distribution states, it will keep making mistakes and may remain out-of-distribution for the remainder of the trial, accumulating $O(H)$ error. Since there is a non-trivial chance of entering an out-of-distribution state at every one of the H time steps, the overall error therefore scales as $O(H^2)$.

3.2 Model-Free Offline RL - Conservative QLearning

As we have seen, value-based methods such as QLearning algorithms are often an appealing choice for RL tasks, due to both the simplicity of design and their effectiveness: this is because their main objective is to learn a state (or state-action) value function and then use it to recover the optimal policy. However, standard algorithms designed for the online setting tend to produce unsatisfying performances in the offline scenario.
As we know, learning in the offline setting revolves around counterfactual predictions (i.e., answering *what-if* queries). The problem is, counterfactual predictions for decisions that deviate too much from the behavior in the dataset cannot be made reliably, because we don't know the underlying data distribution.
So, as a consequence of maximization bias, algorithms like QLearning, which perform their update steps by querying the state-action value at out of distribution

actions for computing the bootstrapping target in training, tend to overestimate the outcome of such state-action pairs which are unknown. As we can see in Fig. 3.2, the policy learned tends to deviate away from the behavior distribution looking

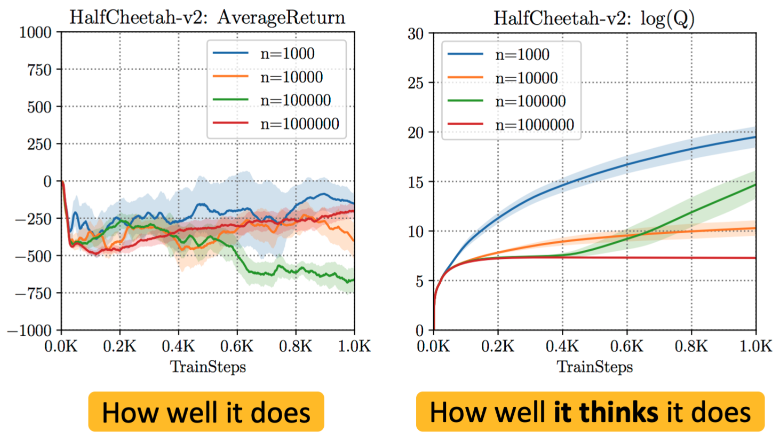


Figure 3.2: Overestimation of unseen, out-of-distribution outcomes when value-based algorithms like QLearning are trained on offline datasets.

for a promising outcome, which actually is not optimal.
Recently, in order to avoid the learned policy to adopt reckless behaviors, in many different papers researchers have presented their own *safe* strategies in which policies and value functions are learned in a *conservative* way, which simply means that we try to estimate the value of out of distribution state-action pairs conservatively, or, in other words, we try to assign them a low value; in this way, the policy will not take advantage of this overly-optimistic biases; in this way, the agent can adopt safe behaviors.
A simply designed but powerful algorithm is **C**onservative **QL**earning, or shortly **CQL** (reference paper can be found at [11]).
As we have seen, directly adapting QLearning-type algorithms to the offline scenario results in poor performances, due to issues with bootstrapping from out-of-distribution actions, which has as its consequence the erroneously optimistic estimation of the value function. In contrast, the core idea of CQL framework offers an alternative: the real value function is estimated in a *conservatve* way, which means that we can extract a lower bound on the true one. This is accomplished by minimizing the q-values $Q(s, a)$ under a suitably chosen distribution over state-action tuples, and then tighten more this bound by adding a maximization term over the dataset distribution.
Being possible to implement CQL both as a value-based (like DQN) and actor-critic algorithm, we can think the algorithm to be made up of two main steps, which are illustrated in the next subsections: one is the evaluation (i.e., the assessment) of

the current policy iterate; the second is a policy improvement step.

3.2.1 Conservative Off-Policy Evaluation

When we talk about off-policy evaluation we mean that the objective is to estimate the value function V^π of a certain target policy π computed by using a datset $\mathcal{D}$ collected by a behavior policy π_β.
Since we want to take a conservative approach, we learn a lower-bound q-function by minimizing the standard Bellman Error objective along with the minimization of q-values: in more detail, we minimize the expected q-value under a particular distribution of state-action pairs $\mu(s,a) = d^{\pi_\beta}(s)\mu(a|s)$, where $d^{\pi_\beta}(s)$ is the state marginal distribution of the dataset: we leave this part unaltered since the problem with maximization bias is only on unseen actions and not on unseen states. Moreover, the added penalization term is controlled by a tradeoff parameter α; in this way the iterative update results in

$$\hat{Q}^{k+1} \leftarrow \arg\min_Q \alpha(\mathbb{E}_{s\sim\mathcal{D},a\sim\mu(a|s)}[Q(s,a)]+ \\ \frac{1}{2}\mathbb{E}_{s,a\sim\mathcal{D}}[(Q(s,a)-\hat{\mathcal{B}}^\pi\hat{Q}^k(s,a))^2], \quad (3.2)$$

where $\hat{\mathcal{B}}^\pi$ is the **empirical Bellman operator**, which is the empirical approximation under the dataset collected and the target policy π of the contraction operator **H** defined in Chapter 1.
In the paper it has been shown that the limit of this sequence of functions, for $k \to \infty$, is a point-wise lower bound for the true Q^π; however, this can be a too strict bound if we just want to obtain a lower-bound on expectation of Q^π (which is equivalent to obtain a lower-bound estimation of V^π). Consequently, in order to refine this bound we can add a q-value maximization term, where the implicit distribution is the one of the dataset; in mathematical formalism (in explicit, the added term is the one preceded by a minus sign), we have

$$\hat{Q}^{k+1} \leftarrow \arg\min_Q \alpha(\mathbb{E}_{s\sim\mathcal{D},a\sim\mu(a|s)}[Q(s,a)] - \mathbb{E}_{s,a\sim\mathcal{D}}[Q(s,a)]+ \\ \frac{1}{2}\mathbb{E}_{s,a,s'\sim\mathcal{D}}[(Q(s,a)-\hat{\mathcal{B}}^\pi\hat{Q}^k(s,a))^2]. \quad (3.3)$$

The refined bound now becomes

$$\mathbb{E}_{\pi(a|s)}[\hat{Q}(s,a)] \leq V^\pi(s) \quad (3.4)$$

when $\mu(a|s) = \pi(a|s)$. The intuitive explanation of Eq.(3.3) is that, since q-values under the behavior policy $\hat{\pi}_\beta$ are maximized, q-values for actions that are likely

under such distribution might be overestimated, with the consequence that $\hat{Q}^{\pi}$ might not be a pointwise lower-bound of Q^{π}.
It can be shown that, for a suitable choice of α, both bounds hold under sampling error (related to the size of the dataset) and function approximation. Moreover, as the dataset gets larger, the theoretical value of α that is needed to obtain such bounds decreases; so, in the limit of infinite data, we can use infinitely small values for the parameter.

3.2.2 CQL for Off-Policy Learning

So far we have seen how we can conservatively estimate the value of a chosen policy π if we set $\mu = \pi$. How can we use this for learning an optimal policy offline?
Since at time k the policy $\hat{\pi}^k$ is implicitly derived from the q-function (for example, as an $\epsilon-$greedy policy), we can choose $\mu(a|s)$ to approximate the policy that would maximize the current q-function iterate; in this way, we obtain a family of optimization problems over $\mu(a|s)$. An instance of this family is $CQL(\mathcal{R})$, which is characterized by a particular choice of the regularizer $\mathcal{R}(\mu)$:

$$\min_{Q} \max_{\mu} \alpha(\mathbb{E}_{s\sim\mathcal{D},a\sim\mu(a|s)}[Q(s,a)] - \mathbb{E}_{s,a\sim\mathcal{D}}[Q(s,a)] + \\ \frac{1}{2}\mathbb{E}_{s,a,s'\sim\mathcal{D}}[(Q(s,a) - \hat{\mathcal{B}}^{\pi}\hat{Q}^{k}(s,a))^2] + \mathcal{R}(\mu). \quad (3.5)$$

Following the work proposed in the paper, we choose $\mathcal{R}(\mu)$ to be the **KL-divergence** against the prior distribution $\rho(a|s) = \mathbf{Unif}(a)$, where $Unif(a)$ is the uniform distribution on the action space.
We recall that, given two probability distributions P, Q over a probability space with sample space Ω, we can define the **Kullback-Leibler divergence** as

$$D_{KL}(P,Q) := \int_{-\infty}^{\infty} p(x)\log\left(\frac{p(x)}{q(x)}\right)dx, \quad (3.6)$$

where p and q denote the probability densities of P and Q respectively.
Now, if we define $\mathcal{R}(\mu) := -D_{KL}(\mu,\rho)$, it can be shown that $\mu(a|s) \propto \rho(a|s) \cdot \exp Q(s,a)$.
So, with this choice the first term in Eq.3.5 corresponds to a soft-maximum of the q-values at any state s; the optimization problem obtained, called $CQL(\mathcal{H})$, is

$$\min_{Q} \alpha\mathbb{E}_{s\sim\mathcal{D}}[\log\sum_{a}\exp Q(s,a) - \mathbb{E}_{a\sim\hat{\pi}_{\beta}(a|s)}[Q(s,a)]] + \\ \frac{1}{2}\mathbb{E}_{s,a,s'\sim\mathcal{D}}[(Q(s,a) - \hat{\mathcal{B}}^{\pi}\hat{Q}^{k}(s,a))^2]. \quad (3.7)$$

The derivation of CQL($\mathcal{H}$) is immediate once we substitute $\mathcal{R} = \mathcal{H}(\mu)$, where $\mathcal{H}(\mu)$ is the **entropy** of μ; then, we solve the optimization over μ in closed form for a given q-function. Indeed, if we consider the optimization problem

$$\max_{\mu} \mathbb{E}_{x\sim\mu(x)}[f(x)] + \mathcal{H}(\mu) \quad \text{s.t.} \ \sum_x \mu(x) = 1, \mu(x) \geq 0 \ \forall x, \tag{3.8}$$

the solution is $\mu^*(x) = \frac{1}{Z}\exp f(x)$, where Z is a normalizing factor.
Moreover, if we use as our regularizer the KL-divergence instead of the entropy $\mathcal{H}$, we can restate the problem as

$$\max_{\mu} \mathbb{E}_{x\sim\mu(x)}[f(x)] + D_{KL}(\mu||\rho) \quad \text{s.t.} \ \sum_x \mu(x) = 1, \ \mu(x) \geq 0 \ \forall x. \tag{3.9}$$

In this case the optimal solution is $\mu^*(x) = \frac{1}{Z}\rho(x)\exp f(x)$. Plugging this result back in Eq. 3.5 we obtain

$$\min_Q \alpha \mathbb{E}_{s\sim d^{\pi_\beta}(s)}\left[\mathbb{E}_{a\sim\rho(a|s)}\left[Q(s,a)\frac{\exp Q(s,a)}{Z}\right] - \mathbb{E}_{a\sim\pi_\beta(a|s)}[Q(s,a)]\right] + \frac{1}{2}\mathbb{E}_{s,a,s'\sim\mathcal{D}}\left[\left(Q - \mathcal{B}^{\pi_k}\hat{Q}^k\right)^2\right] \tag{3.10}$$

In this work, we implemented a QLearning variant of CQL, meaning that we only parametrize the q-function, while the policy is implicitly defined as a greedy one over said q-function. The Algorithm is illustrated in Fig.1:

Algorithm 1 CQL: Conservative QLearning

```
1: Initialize Q-function, $Q_\theta$ and target network $Q_{\theta'}$.
2: for step $t$ in $i$ = 1,2,3, ... do
3:     Train the Q-function using $G_Q$ gradient steps on objective from Equation 3.7
4: $\theta_t := \theta_{t-1} - \eta_Q \nabla_\theta$ CQL($\mathcal{R}$)($\theta$)
5:     Update target network every $K$ steps
6: end for
```

3.2.3 Theoretical Analysis of CQL

In this Section we present results that show the conservatism of the policy updates, i.e, each successive policy iterate is optimized against a lower bound on its value. The next Theorem states that, at each step, the q-value estimates that are learned lower-bound the actual q-function under the action-distribution defined by the current policy if this is updated slowly. Hereon, we use the notation $D_{TV}(\cdot,\cdot)$ to represent the **total variation distance** between two probability measure (for its definition, see [12]).

Theorem 8 (CQL learns lower-bounded values)**.** *Let $\pi_{\hat{Q}^k}(a|s) \propto \exp \hat{Q}^k(s,a)$ and assume that $D_{TV}(\hat{\pi}^{k+1}, \pi_{\hat{Q}^k}) \leq \epsilon$ (i.e., $\hat{\pi}^{k+1}$ changes slowly w.r.t $\hat{Q}^k$). Then, the policy value under $\hat{Q}^k$ lower-bounds the actual policy value, $\hat{V}^{k+1}(s) \leq V^{k+1}(s)$ $\forall s$, if*

$$\mathbb{E}_{\pi_{\hat{Q}^k}(a|s)}\left[\frac{\pi_{\hat{Q}^k}(a|s)}{\hat{\pi}_\beta(a|s)} - 1\right] \geq \max_{a \text{ s.t. } \hat{\pi}_\beta(a|s)>0}\left(\frac{\pi_{\hat{Q}^k}(a|s)}{\hat{\pi}_\beta(a|s)}\right)\cdot \epsilon. \tag{3.11}$$

The term on the left corresponds to the actual amount of conservatism induced in the value function by choosing a soft-max policy over the q-function, i.e., $\hat{\pi}^{k+1} = \pi_{\hat{Q}^k}$. However, since the actual policy $\hat{\pi}^{k+1}$ may be different, the term on the right is the maximal amount of potential overestimation due to this difference. Since we want a lower-bound on the value function, we require the amount of underestimation to be higher: this can be obtained if the policy changes slowly, i.e. for small values of ϵ.
Moreover, we can observe that the q-function receives updates that are "gap-expanding", in the sense that the difference between q-values for in-distribution actions (i.e., actions represented in the dataset) and over-optimistically Q-values for out-of-distribution actions is larger than the one obtained computing the true Q-function on the same actions. This has the logical consequence that the policy $\pi^k(a|s) \propto \exp \hat{Q}^k(s,a)$ is constrained to be closer to the dataset distribution $\hat{\pi}_\beta(a|s)$. So, CQL helps in preventing the dangerous effects of over-estimation and distributional shift.

Theorem 9 (CQL is gap-expanding.)**.** *At any iteration k, CQL expands the difference in expected q-values under the behavior policy $\pi_\beta(a|s)$ and μ_k, such that for large enough values of α_k, we have that*

$$\forall s, \quad \mathbb{E}_{\pi_\beta(a|s)}[\hat{Q}^k(s,a)] - \mathbb{E}_{\mu_k(a|s)}[\hat{Q}^k(s,a)] \geq \mathbb{E}_{\pi_\beta(a|s)}[Q^k(s,a)] - \mathbb{E}_{\mu_k(a|s)}[Q^k(s,a)]. \tag{3.12}$$

In conclusion, with the proper values of α, the algorithm learns a conservative estimate of the actual q-function, with the consequence that the CQL-learned policy performs *at least* at the same level of the actual one. Moreover, we showed that the q-function updates are *gap-expanding*: that is, only the difference between in-distribution and out-of-distribution actions is overestimated, so that maximization bias can be prevented.

3.2.4 Safe Policy Improvement Guarantees

In this section we show that CQL optimizes the objective defined by the **empirical return**, providing a safe policy improvement.
Given a policy π, we define its empirical return $J(\pi, \hat{M})$ as the discounted return of π in the *empirical* MDP $\hat{M}$, induced by the transitions present in the dataset, i.e., $\hat{M} = \{(s, a, r, s') \in \mathcal{D}\}$.

Theorem 10. *Let $\hat{Q}^\pi$ be the fixed point of Eq. 3.3; then*

$$\pi^*(a|s) := \arg\max_\pi \mathbb{E}_{s\sim\rho(s)}[\hat{V}^\pi(s)] \tag{3.13}$$

is equivalently obtained by solving:

$$\pi^*(a|s) \leftarrow \arg\max_\pi J(\pi, \hat{M}) - \alpha\frac{1}{1-\gamma}\mathbb{E}_{s\sim d^\pi_{\hat{M}}(s)}[D_{CQL}(\pi, \hat{\pi}_\beta)(s)], \tag{3.14}$$

where $D_{CQL}(\pi, \hat{\pi}_\beta)(s) := \sum_a \pi(a|s) \cdot \left(\frac{\pi(a|s)}{\hat{\pi}_\beta(a|s)} - 1\right)$.

The Theorem states that CQL performs an optimization of a policy's performance in the empirical MDP $\hat{M}$ while at the same time it succeeds in keeping the learned policy not too far from the behavior policy $\hat{\pi}_\beta$; what's more, the policy obtained is actually a $\zeta-$safe improvement over the behavior policy:

Theorem 11. *Let $\pi^*(a|s)$ be the the policy obtained by optimizing Eq.3.14; then, the policy is a $\zeta-$safe improvement over $\hat{\pi}_\beta$ in the actual MDP M, i.e., $J(\pi^*, M) \geq J(\hat{\pi}_\beta, M) - \zeta$ with probability $1-\delta$, where ζ is given by:*

$$\zeta = 2\Big(\frac{C_{r,\delta}}{1-\gamma} + \frac{\gamma R_{max} C_{T,\delta}}{(1-\gamma)^2}\Big)\mathbb{E}_{s\sim d^{\pi^*}_{\hat{M}}(s)}\Big[\frac{\sqrt{|\mathcal{A}|}}{|\mathcal{D}(s)|}\sqrt{D_{CQL}(\pi^*, \hat{\pi}_\beta)(s) + 1}\Big] - \underbrace{\Big(J(\pi^*, \hat{M}) - J(\hat{\pi}_\beta, \hat{M})\Big)}_{\geq \alpha\frac{1}{1-\gamma}\mathbb{E}_{s\sim d^*_{\hat{M}}(s)}[D_{CQL}(\pi^*, \hat{\pi}_\beta)(s)]}, \tag{3.15}$$

where $C_{r,\delta}$ and $C_{T,\delta}$ are constants depending on δ, the reward function r and the true transition function T.

We can see how ζ is made up of two terms: the first term represents the policy decrease in performance in $\hat{M}$, since this one was actually trained on $\hat{M}$; we refer to this contribution as **sampling error**. The second term represents how the policy performance increase due to the conservatism induced by CQL in the empirical MDP $\hat{M}$.
The policy π^* that we obtain improves over the behavior policy $\hat{\pi}_\beta$ for suitable choices of α; this values can be chosen as small as sampling error decreases (i.e., as the dataset size increases).

3.3 Model-Based Offline RL

The use of predictive models can be a powerful tool for enabling effective offline reinforcement learning. Since basically the core principle of model-based reinforcement learning algorithms is to leverage the learned transition dynamics $T(s_{t+1}|s_t, a_t)$, which is approximated by a parametrized model $T_\Psi(s_{t+1}|s_t, a_t)$, their use is surely motivated when we have powerful supervised learning methods for fitting the model, as well as access to large and differentiated data sources. However, also model-based methods are susceptible to distributional shift.
Indeed, since the policy is optimized to obtain the highest possible expected return under the current model, this optimization process can lead to the policy *exploiting* the model, intentionally producing o.o.d (out-of-distribution) states and actions at which the model $T_\Psi(s_{t+1}|s_t, a_t)$ erroneously predicts successor states s_{t+1} that lead to higher returns than the actual successor states that would be obtained in the real MDP. This *model exploitation* problem can lead to policies that produce substantially worse performance in the real MDP than what was predicted under the model. Regarding optimality guarantees using a model-based approach, theoretical analysis of model-based policy learning can provide bounds on the error incurred from the distributional shift due to the divergence between the learned policy $\pi(a|s)$ and the behavior policy $\pi_\beta(a|s)$, similar to what we have done before, except that now both the policy and transition probabilities experience distributional shift. In [10] some results are shown concerning the possibility of these methods in terms of final return.
In particular, if we assume that the **total variational distance**[1] (**TVD**) between the learned model T_Ψ and the true dynamics T is bounded by an error

$$\epsilon_m = \max_t \mathbb{E}_{d_t^\pi} D_{TV}(T_\Psi(s_{t+1}|s_t, a_t) \| T(s_{t+1}|s_t, a_t)), \tag{3.16}$$

and the TVD between π and π_β is likewise bounded by ϵ_π, then the true policy value $J(\pi)$ is related to the policy estimate under the model, $J_\Psi(\pi)$, according to

$$J(\pi) \geq J_\Psi(\pi) - \left[\frac{2\gamma r_{max}(\epsilon_m + 2\epsilon_\pi)}{(1-\gamma)^2} + \frac{4 r_{max}\epsilon_\pi}{1-\gamma}\right] \tag{3.17}$$

Intuitively, the second term within squared parentheses represents the error due to the distributional shift, while the other one is consequence of the shift in distribution of the approximated model. The first error also includes a dependence on ϵ_π, because as the policy goes farther from π_β, new states encountered will be consequently not present in the data distribution; this means that, since the model

[1]Recall that, given two probability measures P, Q defined on a sigma-algebra $\mathcal{F}$, their total variation distance can be computed as $D_{TV}(P, Q) \coloneqq \sup_{A\in\mathcal{F}}|P(A) - Q(A)|$.

is trained on the very offline dataset, it will be less confident with predictions conditioned on these more recent states.
Although model-based reinforcement learning appears to be a natural fit for the offline paradigm, many of the current methods rely on explicit uncertainty estimation for the model to detect and quantify distributional shift, for example by using a bootstrap model ensemble. For instance, recently two concurrent methods, MOReL ([13]) and MOPO ([14]) have proposed offline model-based methods that aim to utilize conservative value estimates to provide analytic bounds on performance. This is accomplished not immediately on the critic's training; instead, both these methods work on a *conservative* modification of the original MDP, in order to directly induce conservative behavior in the simulated transition dynamics. Basically, the reasoning behind these precautions is to induce the policy to assign low values to states on which we think the model is probably not able to make safe predictions. In this way, as the learned policy is constrained to perform actions belonging to regions where the model is accurate, then the model-based estimate of the policy's value will be accurate too.
Although these methods has consistently proven to be succesful, their weak link lies in the fact that, in order to work, they must address in some manner the uncertainty caused by the model approximation. For the sake of theory, we can just assume the existence of an *error oracle* $u(s, a)$ that provides a consistent estimate of the accuracy of the model at state-action tuple (s, a); for example, the oracle could satisfy the property that $D(T_\Psi(s_{t+1}|s_t, a_t) \| T(s_{t+1}|s_t, a_t)) \leq u(s_t, a_t)$ for some divergence measure D. Then, conservative behavior can be induced either by modifying the reward function to obtain a conservative reward of the form $\tilde{r}(s, a) = r(s, a) - \lambda u(s, a)$, as in MOPO, or by modifying the MDP under learned model so that the agent enters an absorbing state with a low reward value when $u(s, a)$ is below some threshold, as in MOReL. In both cases, we have that the estimated policy's performance under the modified reward function or MDP structure is a lower bound for the policy's true performance in the real MDP; this means that if we stay safe we avoid the risk of overestimating the final result. However, this is easier said than done, since such ways of proceeding still need a faithful estimation of the error oracle $u(s, a)$.

3.4 COMBO

We now present **COMBO** (short for **C**onservative **O**ffline **M**odel-**B**ased Policy **O**ptimization), the main algorithm used in this work (reference paper can be found at [15]).
As we have seen before, practical variants of model-based algorithms rely on explicit uncertainty quantification for incorporating pessimism. The innovative idea of

COMBO instead is to regularize the value function on out-of-support state-action tuples generated via rollouts under the learned model. In this way, we avoid being overly-optimistic when it comes to assess the value of state-action tuples that are out-of-support (i.e., not present in the training dataset), without the necessity to explicitly represent the world's uncertainty (which is what *we don't know we don't know*). Concerning the general algorithmic framework of COMBO, since it's a model-based approach, we must first use the data available to learn an approximation of the transition dynamics; subsequently, we can either train an actor-critic agent or a value-based one (similarly to DQN); however, differently from CQL, since now we have a model representation of the environment, we can use it to augment our dataset by generating synthetic transitions. This is similar to what we have seen with Dyna, but, differently from it, the critic function in COMBO stays conservative by trying to penalize the value function in state-action tuples that, as we said before, are out of the distribution for the dataset: such tuples are therefore only the ones synthetically generated by the model.
Regarding the theoretical results, it can be proven that, same as CQL, the policy learned by this algorithm obtains a lower-bound performance on the one from the real optimal policy.
We can observe that COMBO is an extension of CQL adapted to the model-based setting; so, as in the model-free algorithm, at each iteration we first try to assess the goodness of the current policy in a conservative way; secondly, we try to improve the policy, as follows:

- **conservative policy evaluation**: at iteration k, we will have a suboptimal policy π_k; what we want to do is trying to assess the value of such policy by evaluating its q-function Q^{π_k} in a conservative way. Similarly to CQL, this is accomplished by penalizing Q^{π_k} on tuples drawn from a particular state-action distribution that is more likely to be out-of-support while pushing up the q-values on state-action pairs that are reliable. In short, we solve the following optimization problem:

$$\hat{Q}^{k+1} \leftarrow \arg\min_{Q} \beta(\mathbb{E}_{s,a\sim\rho(s,a)}[Q(s,a)] - \mathbb{E}_{s,a\sim\mathcal{D}}[Q(s,a)] + \frac{1}{2}\mathbb{E}_{s,a,s'\sim d_f}[(Q(s,a) - \hat{\mathcal{B}}^{\pi}\hat{Q}^{k}(s,a))^2], \quad (3.18)$$

  where $\hat{\mathcal{M}}$ is the approximated MDP implicitly defined by the learned model; $\rho(s,a)$ and d_f are sampling distributions that we can choose: for $\rho(s,a)$, we make the following choice:

$$\rho(s,a) = d^{\pi}_{\hat{\mathcal{M}}}(s)\pi(a|s), \quad (3.19)$$

  where $d^{\pi}_{\hat{\mathcal{M}}}$ is the discounted marginal state distribution when executing π in the learned model $\hat{\mathcal{M}}$. Samples from such distribution can be drawn by rolling out π in the learned model. Similarly, d_f is an $f-$interpolation between the offline dataset and synthetic rollouts from the model:

$$d_f^\mu = fd(s,a) + (1-f)d^\mu_{\hat{\mathcal{M}}}(s,a), \tag{3.20}$$

where $f \in [0,1]$ is the ratio of the datapoints drawn from the offline dataset and $\mu(\cdot|s)$ is the rollout distribution used with the model, which can be modelled as π or a uniform distribution. For simplicity, hereon we denote $d_f := d_f^\mu$. Under such choices of ρ and d_f, we push down (or conservatively estimate) q-values on the state-action tuples from model rollouts and push up q-values on the real state-action pairs from the offline dataset. When updating q-values with the Bellman backup, we make use of both the model-generated data and the real data, similar to Dyna. Note that in comparison to CQL and other model-free algorithms, COMBO learns the q-function over a richer set of states beyond the states in the offline dataset. This is made possible by performing rollouts under the learned dynamics model, denoted by $d^\mu_{\hat{\mathcal{M}}}$.

- **Policy Improvement Using a Conservative Critic:** after learning a conservative critic Q^π , we improve the policy as:

$$\pi' \leftarrow \arg\max_\pi \mathbb{E}_{s\sim\rho, a\sim\pi(\cdot|s)}[\hat{Q}^\pi(s,a)] \tag{3.21}$$

  where $\rho(s)$ is the state marginal of $\rho(s,a)$.

When policies are parameterized with neural networks, we approximate the arg max with a few steps of gradient descent. The algorithm is represented in Fig. 2.

Algorithm 2 COMBO: Conservative Model Based Offline Policy Optimization

```
Require: Offline dataset $\mathcal{D}$, rollout distribution $\mu(\cdot|s)$, learned dynamics model
    $\hat{T}_\theta$, initialized critic $Q_\Psi$.
    Train the probabilistic dynamics model $\hat{T}_\theta(s', r|s,a)$ on $\mathcal{D}$.
 2: Initialize the replay buffer $\mathcal{D}_{model} \leftarrow \emptyset$.
    for $i = 1,2,3,\dots$ do
 4:     Perform model rollouts by drawing samples from $\mu$ and $\hat{T}_\theta$ starting from
    states in $\mathcal{D}$. Add model rollouts to $\mathcal{D}_{model}$.
        Conservatively evaluate current policy by repeatedly solving (3.18) to obtain
    $Q^i_\Psi$ using data sampled from $\mathcal{D} \bigcup \mathcal{D}_{model}$.
 6:     Improve policy under state marginal of $d_f$ by solving (3.21) to obtain $\pi^i$.
    end for
```

3.4.1 Theoretical Properties

This Section shows the results obtained by COMBO's authors in [15].
Assuming that state and action spaces are both finite, the q-function found at step k can be obtained by differentiating 3.18 with respect to Q^k:

$$\hat{Q}^{k+1}(s,a) = (\hat{\mathcal{B}}^{\pi} Q^k)(s,a) - \beta \frac{\rho(s,a) - d(s,a)}{d_f(a,)}. \tag{3.22}$$

We can see that the three distributions ρ, d, d_f constitutes a penalty weighted by β for the q-function; such penalty can be shown to be positive in expectation, which is useful since we want to prevent overestimation. Moreover, we have the following result:

Lemma 3 (Interpolation Lemma)**.** *For any $f \in [0,1]$, and any given $\rho(s,a) \in \Delta^{|\mathcal{S}||\mathcal{A}|}$, let d_f be an $f-$interpolation of ρ and $\mathcal{D}$, i.e., $d_f := f d(s,a) + (1-f)\rho(s,a)$. For a given iteration of Eq.3.22, define the expected penalty under $\rho(s,a)$ as:*

$$\nu(\rho, f) := \mathbb{E}_{\rho}\Big[\frac{\rho(s,a) - d(s,a)}{d_f(s,a)}\Big]. \tag{3.23}$$

Then, $\nu(\rho, f)$ satisfies:

- $\nu(\rho, f) \geq 0 \quad \forall \rho, f$
- *$\nu(\rho, f)$ is monotonically increasing in f for a fixed ρ*
- *$\nu(\rho, f) = 0$ iff $\forall (s,a)$, $\rho(s,a) = d(s,a)$ or $f = 0$*

Since in the COMBO implementation we set $\rho(s) = d^{\pi}_{\hat{\mathcal{M}}}(s)$ and $\rho(a|s) = \pi(a|s)$, each update step of 3.22 penalizes the q-function, making it more conservative; such penalization is controlled by f, which in practice controls the balance between real and synthetically generated data.
COMBO optimizes a lower bound on the expected return of the learned policy, which is close to the performance of the actual policy, if its state-action distribution is in support of the one of the behavior policy; otherwise, it's nevertheless a conservative estimation of it.
It can be shown that, given any policy π, the asymptotic q-function learned by COMBO is a lower-bound of the actual q-function of the policy with high probability, when β is large enough. Indeed, we can think of the Bellman backup in Eq.(3.18) as an $f-$interpolation of two Bellman backups, namely $\mathcal{B}^{\pi}_{\bar{M}}$ and $\mathcal{B}^{\pi}_{\hat{M}}$, where $\bar{M}$ is the empirical MDP obtained by taking raw data counts of the elements in the dataset, while $\hat{M}$ is the learned model.
Now we can state the Theorem:

Theorem 12. *[Asymptotic lower-bound] Let P^π denote the Hadamard product of the dynamics P and a given policy π in the actual MDP and let $S^\pi := (I-\gamma P^\pi)^{-1} c_S$, where c_S is a suitably chosen positive constant. Let D denote the total variation divergence between two distributions. For any $\pi(a|s)$, the q-function obtained by recursively applying Eq.(3.22) with $\hat{\mathcal{B}}^\pi = f\mathcal{B}^\pi_{\overline{M}} + (1-f)\mathcal{B}^\pi_{\hat{M}}$, with probability at least $1-\delta$, results in $\hat{Q}^\pi$ that satisfies:*

$$\forall s, a, \hat{Q}^\pi(s,a) \leq Q^\pi(s,a) - \beta \cdot \epsilon_c + f\epsilon_s + (1-f)\epsilon_m,$$

where $\epsilon_c, \epsilon_s, \epsilon_m$ are given by:

$$\begin{aligned} \epsilon_c(s,a) &:= \left[\frac{1}{c_S} S^\pi \left[\frac{\rho - d}{d_f}\right]\right](s,a), \\ \epsilon_m(s,a) &:= \left[S^\pi\left[|R - R_{\hat{M}}| + \frac{2\gamma R_{max}}{1-\gamma} D(P, P_{\hat{M}})\right]\right](s,a), \\ \epsilon_s(s,a) &:= \left[S^\pi\left[\frac{C_{r,T,\delta} R_{max}}{(1-\gamma\sqrt{|\mathcal{D}|})}\right]\right](s,a). \end{aligned}$$

Having stated these results, we can formulate a straightforward corollary:

Corollary 1. *For a sufficiently large β, we have that*

$$\mathbb{E}_{s\sim\mu_0(s), a\sim\pi(\cdot|s)}[\hat{Q}^\pi(s,a)] \leq \mathbb{E}_{s\sim\mu_0(s), a\sim\pi(\cdot|s)}[Q^\pi(s,a)],$$

where $\mu_0(s)$ is the initial state distribution. Furthermore, when ϵ_s is small, such as in the large sample regime; or when the model bias ϵ_m is small, a small β is sufficient along with an appropriate choice of f.

Theorem 12 states that large values of β emphasize conservatism, which leads to "smaller" q-functions, which happens also with CQL. However, COMBO improves over the latter, since we can leverage the learned model to handle states that are not represented in the dataset; moreover, COMBO does not produce a value function which underestimates the real one at every state, but only lower-bounds the expected value function under the initial state distribution: this results in a tighter lower-bound.

3.4.2 Safe Policy-Improvement Guarantees

Finally, we show that the policy learned by COMBO is actually an improvement over the behavior policy:

Theorem 13. *$\zeta-$safe policy improvement Let $\hat{\pi}_{out}$ be the policy learned by COMBO. The policy is a $\zeta-$safe policy improvement over π_β in the actual MDP $\mathcal{M}$, i.e.,*

$J(\hat{\pi_{out}}, \mathcal{M}) \geq J(\hat{\pi_\beta}, \mathcal{M}) - \zeta$, with probability at least $1-\delta$, where ζ is given by:

$$\zeta = \mathcal{O}\Big(\frac{\gamma f}{(1-\gamma)^2}\Big) \underbrace{\Big[\mathbb{E}_{s \sim d_{\mathcal{M}}^{\hat{\pi}_{out}}}\Big[\sqrt{\frac{|\mathcal{A}|}{|\mathcal{D}(s)|} D_{CQL}(\hat{\pi}_{out}, \pi_\beta)}\Big]\Big]}_{:=(1)} + \tag{3.24}$$

$$\mathcal{O}\Big(\frac{\gamma(1-f)}{(1-\gamma)^2} \underbrace{D_{TV}(\bar{\mathcal{M}}, \hat{\mathcal{M}})}_{:=(2)} - \underbrace{\beta \frac{\nu(\rho^\pi, f) - \nu(\rho^\beta, f)}{(1-\gamma)}\Big)}_{:=(3)}. \tag{3.25}$$

Here, D_{CQL} is a probabilistic distance between policies defined as

$$D_{CQL}(\pi, \pi_\beta)(s) := \sum_a \pi(a|s) \cdot \Big(\frac{\pi(a|s)}{\pi_\beta(a|s)} - 1\Big) \tag{3.26}$$

So, we see that ζ is made up of three contributions: term (1) accounts for the performance decrease due to the limited size of the dataset and gets negligible with huge amount of data available; term (2) accounts for the approximation error induced by using a learned model; lastly, it can be shown that term (3) is equivalent to the improvement in policy performance if we run COMBO first in the empirical and then in the model MDPs. This is because the learned model has been trained on the dataset, so that the marginal distribution $\rho(s,a)$ will be closer to the dataset distribution $d(s,a)$ than its counterpart for the learned policy, which is ρ^π. So, term (3) tends to be positive in practice, and help in reducing the penalties introduced by the other two terms, refining the performance gain.
It's important to notice that sampling error (term (1)) is significantly reduced w.r.t. the one of CQL, if a near-accurate learned model is employed to generate data. Moreover, differently from other state-of-art model-based methods, even if the model is suboptimally accurate, we can bias the updates towards the empirical MDP by controlling the parameter f.

Chapter 4

Environment Setup and Experiments

In this chapter we present the environment on which the algorithms have been applied as well as the results obtained.
We first start with describing the problem we have tried to solve as well as the framework developed to formalize such problem.

4.1 CARS Framework

At present time, hybrid electric vehicles (**HEV** for short) seems like one of the best ways to go among the many technologies available for greenhouse-gases and pollutant emissions on-road reduction. In this thesis we want to address the problem of balancing energy consumption in a hybrid electric vehicle: in particular, fixed a certain trajectory, we want the vehicle to use fuel as little as possible, making sure at the same time that battery charge always remains within a certain charge interval. The simulation of the physical environment and its experiments are realized by working on a framework developed by **PoliTo-AddFor CARS** team in [16].

In HEVs, energy is generated by different sources: the conventional internal combustion engine **ICE** integrated with one or more motor-generators (**MGs**) and batteries. This energy-power complex system must be managed by an internal controller: this is the role of the energy management system (**EMS**), which is used to define the sequence of operating modes to be realized during a specific driving mission (DM for short).
As said before, our goal is EMS optimal control: however, we want to address this problem from an Offline RL perspective.

As in every RL task, the general formulation of the problem accounts for an objective function J to be optimized under a control policy π and a set of boundary conditions. More specifically, here we want to minimize the fuel consumption (FC) variable while achieving a charge-sustaining (CS) of the battery. In formulas, we have

$$\pi^* = \arg\min_{\pi \in f} J = \arg\min_{\pi \in f} \int_0^T \dot{m}_f \tag{4.1}$$

$$\text{with} \begin{cases} SoC^T = SoC^0 \\ V_v^t = V_r^t \end{cases}, \tag{4.2}$$

for $t = 1,2,\dots,T$, where $\dot{m}_f$ is the actual FC, T is the final timestep of the DM, SoC^0 and SoC^T are the initial and final battery *states of charge* (SoCs)and V_v^t, V_r^t are respectively the vehicle velocity and the DM requested velocity at time t. In this way, the FC minimization gets targeted including two boundary conditions: the first is battery CS, while the second is the compliance of the vehicle velocity w.r.t. the DM request.
The reward function chosen deals with these boundary conditions and is

$$R_{FCEq} = a + c \cdot FC + d \cdot FC_{eq}, \tag{4.3}$$

where

$$FC_{eq} = -\frac{(SoC - SoC^*) \cdot E_{b,w}}{H_i \cdot \bar{\eta}_{ICE}}. \tag{4.4}$$

a, c, d are numerical coefficients to be tuned; SoC^* is a reference battery SoC value, SoC is the actual battery SoC value and FC_{eq} is the equivalent FC. In Eq. 4.4 $E_{b,w}$ is the battery energy content related to the admitted battery SoC window, H_i is the lower heating value of the fuel considered for the specific ICE application and $\bar{\eta}_{ICE}$ is a fixed average ICE efficiency value.

4.1.1 Software Framework

In this section we present the **Integrated Modular Software Framework** developed by CARS team in [16], on which the experiments have been conducted. The tool is made up of four distinct modular components: the **Simulator**, the **Environment**, the **Communication Interface** and the **Agent**. The Agent and the Environment are written using the Python 3.8.5 programming language [17]; the HEV simulator is written using MATLAB® [18], while the Communication Interface uses the UDP communication protocol. The complete configuration of all parts ensures complete reproducibility of the experiments and, to facilitate, the configuration of each experiment can be stored in separated databases.
The simulator reproduces the pre-transmission functions of the HEV: it receives the action $\tilde{a}_t$ (in the physical representation) from the Environment at time step t;

in turn, it generates the new state s_{t+1}, along with an action **feasibility mask** m_{t+1}: this represents the set of actions which are allowed to be taken by the agent at time step t. In practice, since the action space is discrete, the feasibility mask is a boolean array that indicates whether an action is or not physically possible, for example in the case where the power requested by the road is not realizable or the battery SoC range has been trespassed.
The Environment resembles the OpenAIGym suite's design (see [19] for more info). Within this module, a distinction is made between *physical* and *logical* actions: the physical action $\tilde{a}_t$ is the actual action applied to the HEV powertrain, while a_t is its logical representation.
Regarding the states, the agent doesn't have access to the full state representation: instead, it receives only an observation of it, containing partial information. The state filtering can be done both in a discrete or in a continuous space; in the second case the observation is standardized in the interval [0,1].

4.2 Data collection

The gathering of datasets to experiment with was performed along the standards of state-of-the-art offline RL works. This means that data were collected from different online RL policies, which stem from three different agents, namely **AgentRandom**, **AgentDDQN** and **AgentQLearning**.
AgentRandom is an agent employing a **random** policy, meaning that at each time step the action taken is chosen uniformly at random among the feasible actions. This ensures the maximum degree of **exploration**, i.e., this is the policy which covers the most of the state-action space.
The AgentDDQN instead employs an **ϵ-greedy** policy based on the **Double DQN** algorithm presented in Section 2.8.2, with ϵ-greedy meaning that, at each time step t, the agent chooses an action uniformly at random (like AgentRandom) with probability ϵ; on the other hand, with probability $1-\epsilon$, the action is chosen greedily w.r.t. the q-function, that is

$$a_t = \arg \max_{a \text{ feasible}} Q(s_t, a). \tag{4.5}$$

ϵ is an exploration parameter that linearly decreases during training: this ensures that exploration is predominant in the beginning of the training, when the agent is not confident with its predictions, while towards the end of the training the exploration is really minimal, since by that time the agent will hopefully be almost optimally trained, so that we would like the actions to be chosen based on its predictions and not randomly sampled. We start from an ϵ value of 0.8; every episode it decreases by 0.002, until a value of 0.05 is reached.
Each dataset collected contains transitions in the form $(s, a, r, done, s')$, where s is

the state observed by the agent at a certain time step, a is the action taken, r and s' are respectively the reward and the next state observed, while *done* is a boolean flag that indicates whether the episode is terminated or not (in the state s').
The datasets collected are the following:

- **random**: we collected 250000 transitions in the form $(s, a, r, done, s')$ using the policy of AgentRandom;
- **qlearning**: we collected 250000 transitions using an expert QLearning agent;
- **expert**: we collected 250000 transitions by fully training AgentDDQN; at the end of the training, we perform some episodes, of which each step was collected;
- **medium**: same as the **expert** dataset, with the difference that training was stopped when the return obtained in an episode is half of the maximum which can be obtained after full training of the agent;
- **medium-expert**: the union of **medium** and **expert** datasets (500000 steps);

When collecting qlearning, medium and expert data, we set an ϵ value of 0.25; this is because the environment is deterministic, so a trained agent following a greedy policy (i.e., $\epsilon = 0$) would see the same sequence of transitions at each episode. So, by adding a slight degree of exploration, we ensure that a proper dataset can be gathered. Each episode is made up of **500** steps, so that 250000 transitions correspond to 500 collecting episodes.
In the next sections we delve in the proper experimentation of the offline algorithms.

4.3 CQL vs Baseline

For starters, we trained and tested on the collected data a simple offline version of **AgentDDQN**, which we called **OfflineDDQN**: at each training step, a batch of data is sampled and used to train the critic using the classic DQN target loss (which is the Mean Squared Bellman Error, or MSBE). This was done in order to have a reference baseline with which we would have been able to compare the results of CQL and COMBO. Generally, offline plain adaptations of DDQN are known not to properly work on complex RL tasks (for example, see [20]). However, contrarily to what should be to expect, the agent OfflineDDQN is capable of steadily learning, at least partially, and outperforms the conservative **AgentCQL**, Indeed, as we can see here in Figs. 4.5-4.6, we show the returns on a single test episode: in the first one the metric chosen is the cumulative reward without any discount factor (i.e., the return computed with $\gamma = 1$), while the second figure shows the return using a discount factor of $\gamma = 0.99$. What's interesting to see is that both agents are

capable of learning good and near-optimal policies from **random** data, which are collected from a random policy, while, on the other hand, learning from an expert agent leads to very poor results, unless the expert data is joined with the medium one, as in **medium-expert**. The same happens in the case of **qlearning**, which is also a near-optimal policy.
On the other hand, the baseline yields better results overall than our CQL implementation, showing that this offline algorithm fails to deliver promising results for this task. What's more, for **qlearning**, **medium** and **expert** the CQL policy yields negative nondiscounted returns.
An explanation for this behavior could be found in the narrowness of the data distribution: that is, the data gathered could bring not enough information to cover the interesting portions of the state-action space, making the algorithm unable to leverage said data.
For example, in Fig. (4.1) we can see the distributions for the state s variable from random, medium, expert and qlearning datasets: we can observe that qlearning

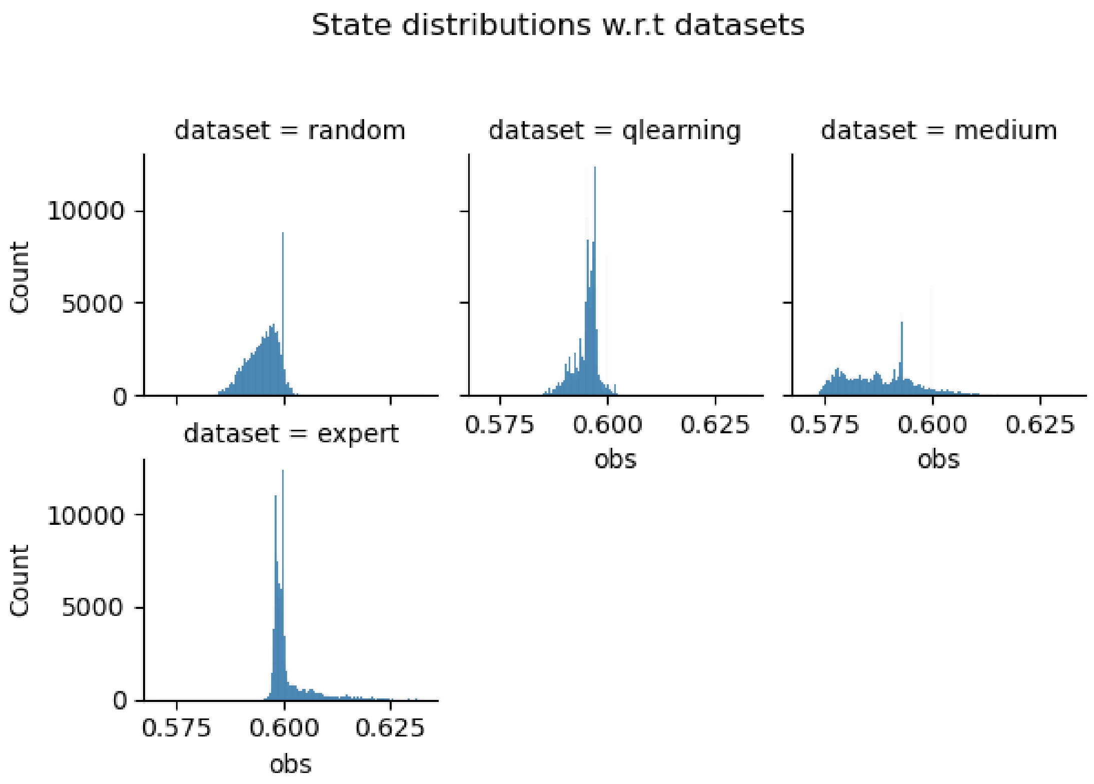


Figure 4.1: State distributions: on the x label, *obs* stands for "observation"

and expert distributions have very sharp peaks: this means that when they are used for training, the agent can only see a small portion of the state space. This

problem could be exacerbated when we implement a conservative approach, like in CQL: since the states collected in qlearning and expert dataset are those visited from near-optimal policies (since said policies are expert DQN and qlearning), when the conservative critic of CQL receives a batch containing these states, it adjusts in a conservative way only those parameters relative to such states. This can results in an overly-pessimistic approach. However, to validate this hypothesis is not that simple, since CQL performs poorly also on **medium** dataset, which state *s* distribution does not present sharp peaks; this means that further analyses are necessary.
Both agents are trained for 250k steps before being tested. For completeness, in Figs. 4.2-4.4 are showed also the distributions of the other variables: as we can

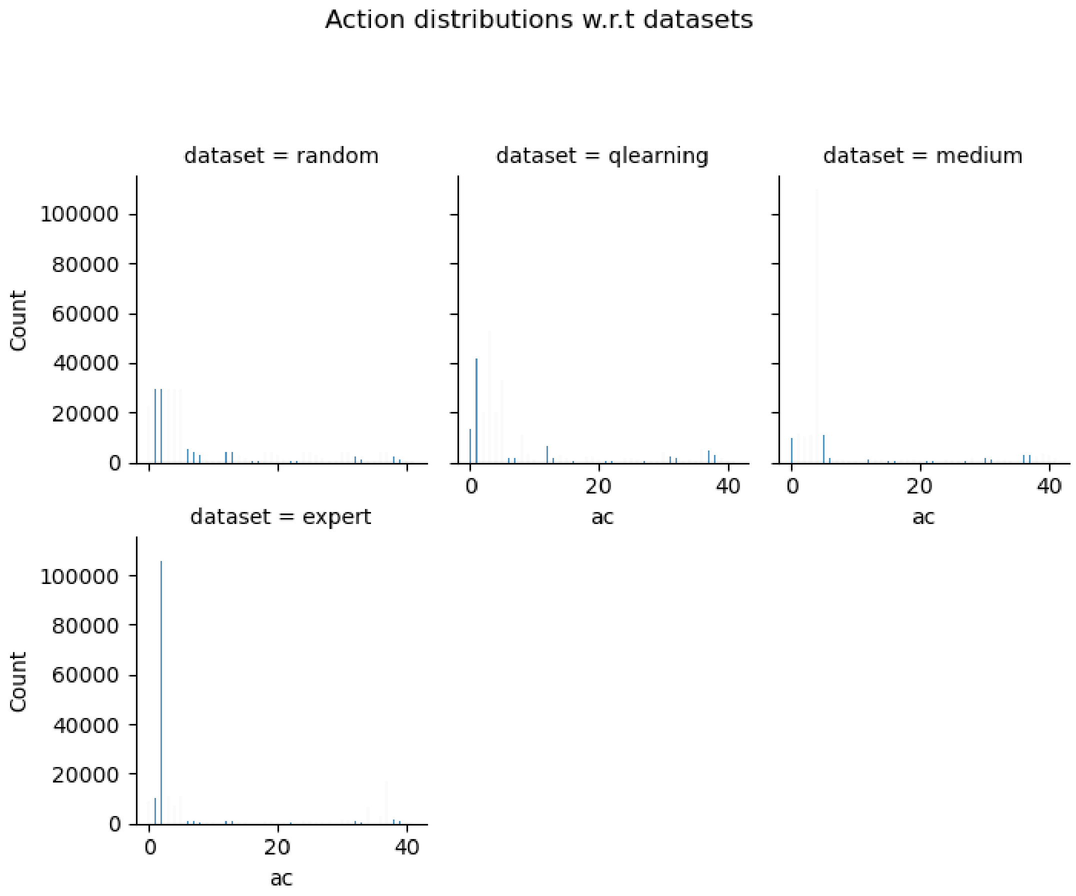


Figure 4.2: Action distributions: on the x label, ac stands for "action"

see, distributions on **qlearning** and **expert** are always very concentrated along their mean.

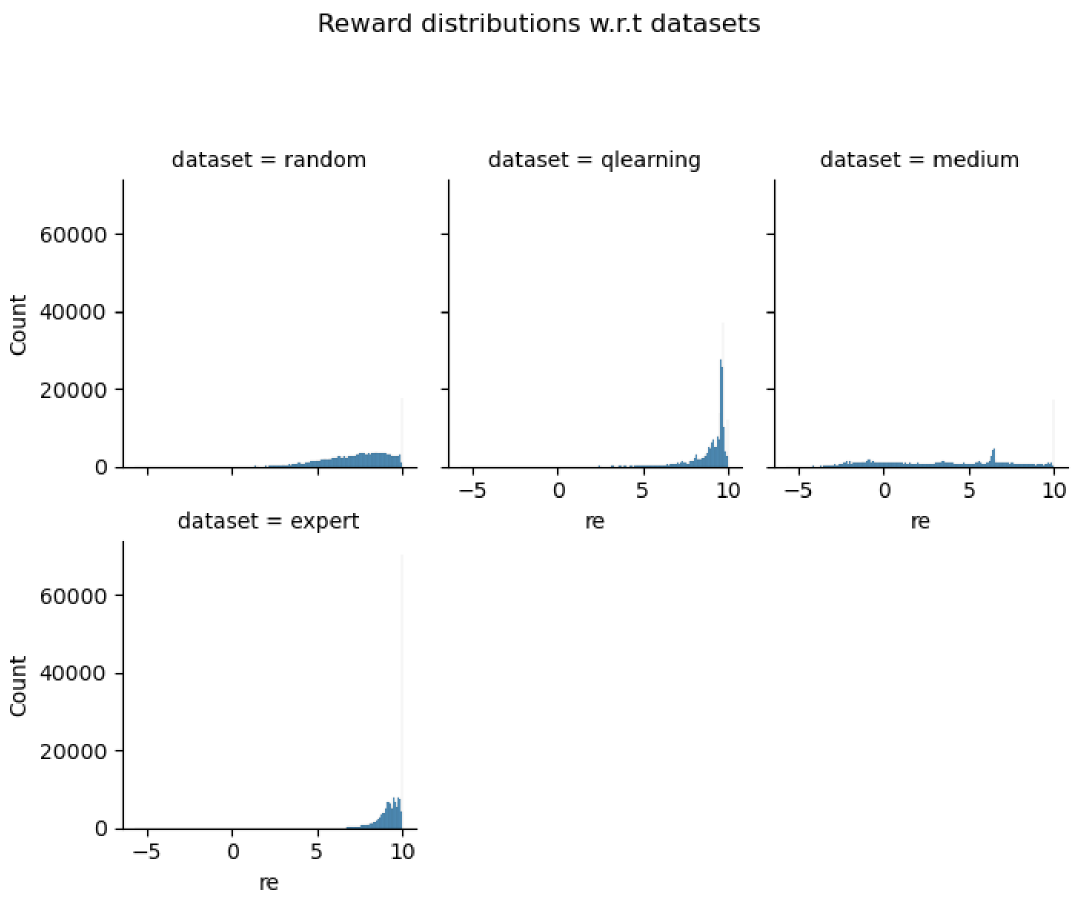


Figure 4.3: Reward distributions: on the x label, re stands for "reward"

Another interesting fact to observe is that on some datasets the agent trained is not able to fully complete the episode; indeed:

- CQL trained on **qlearning** dataset incurs in a *SOC out of boundaries* error, meaning that it's not able to keep the state of charge in the correct range along the trajectory; moreover, the agent trained takes the same action (action 1) at every step;

- CQL trained on **expert** dataset incurs in a *SOC out of boundaries* error; moreover, the agent trained takes only actions 0 and 1 throughout the (prematurely halted) test episode.

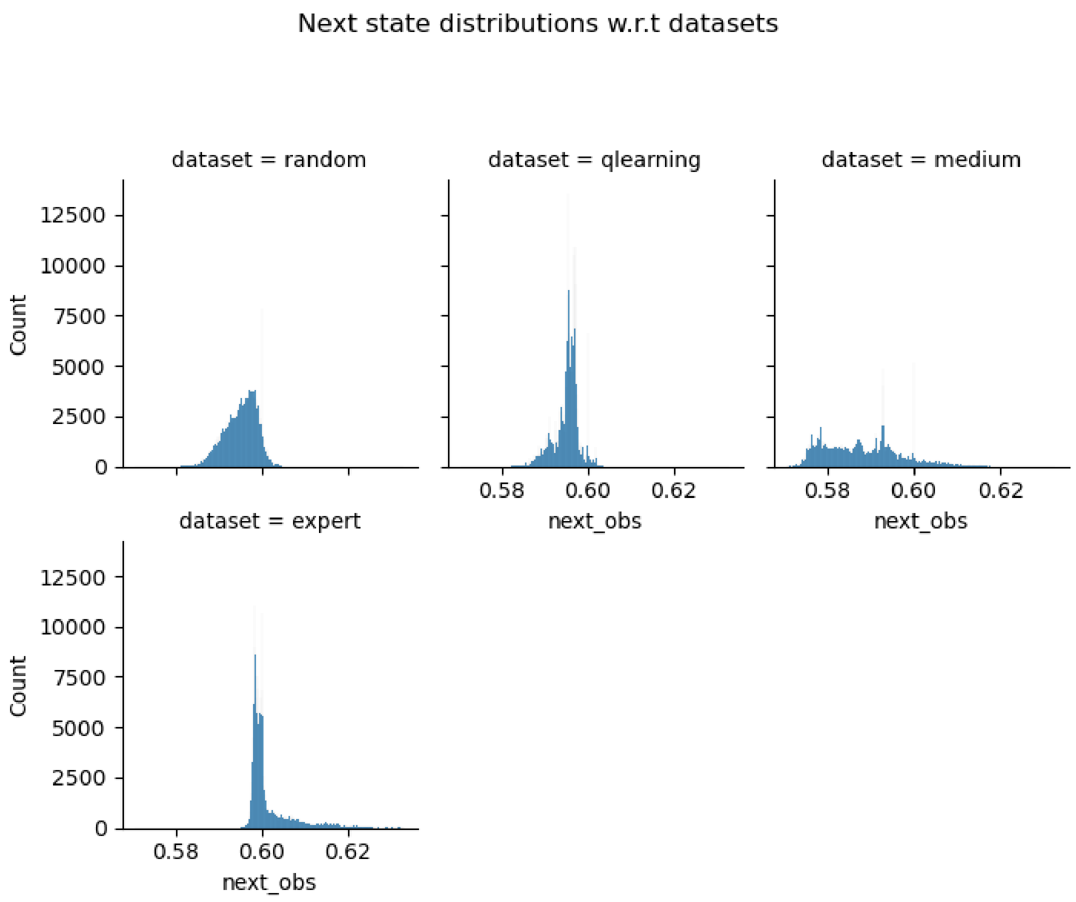


Figure 4.4: Next state distributions: on the x label, $next_obs$ stands for "next observation"

4.4 COMBO vs Baseline

For the COMBO practical implementation we have tested two different transition models: the first is a **single Gaussian network**, that is, we trained a neural network to approximate a transition function of the type

$$T(s_{t+1}, r|s_t, a_t) = \mathcal{N}(\mu(s_t, a_t), \Sigma(s_t, a_t)), \tag{4.6}$$

meaning that the network produces a Gaussian distribution (by outputting the mean and standard deviation conditioned on the observed state-action tuple) over the next state and reward.
The second model is a **Gaussian ensemble**: we trained N single Gaussian networks with the same architecture but with different random initialization of the

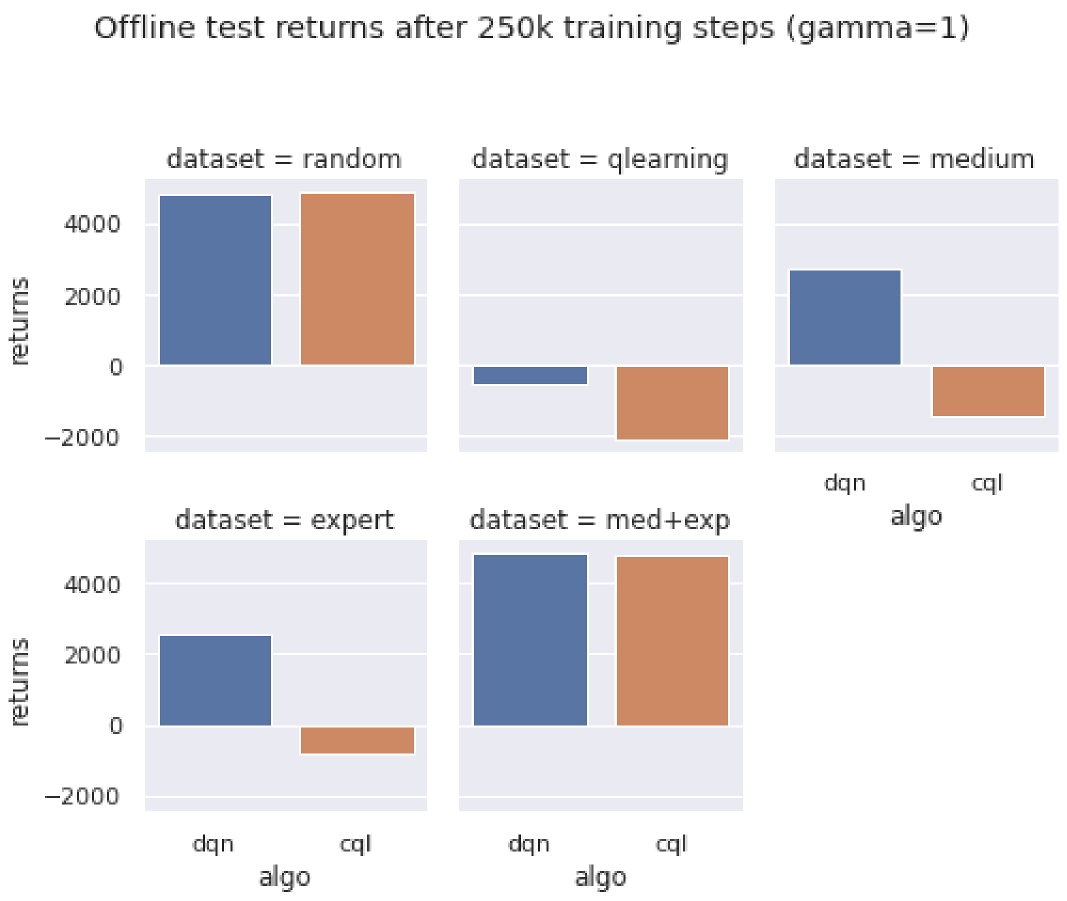


Figure 4.5: Offline Test Returns - CQL vs DQN: CQL produces poor results on **qlearning** and **expert** data

parameters; during validation phase, we only pick the best $K < N$ models and, when collecting rollouts, we randomly pick one dynamics model from the best K; in practice, we set $N = 7, K = 5$. Both the versions (single and ensemble) have been trained via maximum likelihood.

4.4.1 Model Accuracy

The first thing to assess is the model's prediction accuracy, i.e. whether the model used is able to accurately represent the environment dynamics. Regarding the single Gaussian model, in Fig. 4.7, we can see the total MSE along an episode with respect to the number of steps used for training the model. We plotted the trend of the prediction accuracy based on the number of model training steps using the random dataset, while the critic is chosen as random (i.e., each time the action

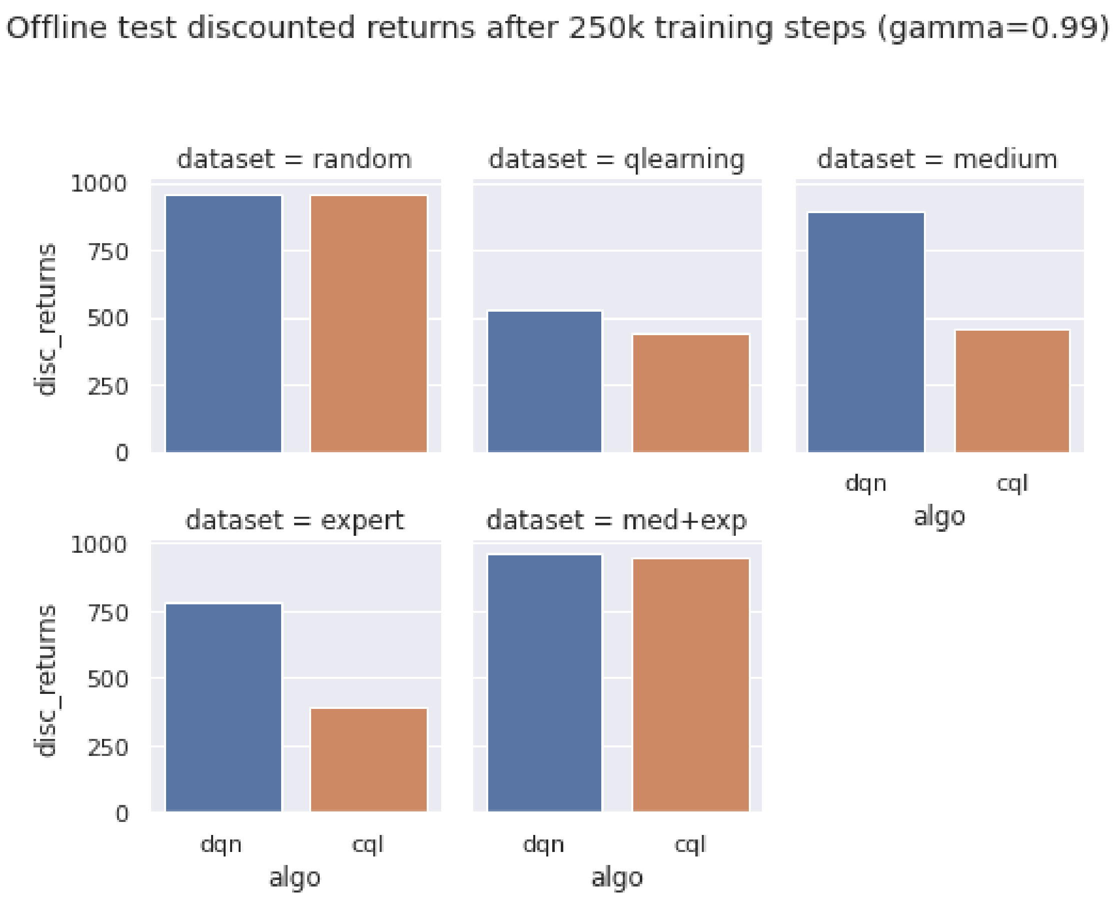


Figure 4.6: Offline Test Discounted Returns - CQL vs DQN: CQL produces poor results on **qlearning** and **expert** data

taken is sampled at uniform).
As we can see, the model is able to basically represent the whole portion of interest of the state-action space. Moreover, we show in Fig.4.8 that also the ensemble has good predictive power, actually better than the single dynamics: indeed, in this case the MSE goes towards zero even sharper.

4.4.2 Horizon

Since COMBO leverages the predictive power of the trained model to collect rollouts, the horizon parameter (that is, the number of consecutive timesteps the model has to infer) must also be calibrated. To this end, we conduct some analyses in this sense. In Fig. 4.9, we show the trend of the absolute relative error in computing multistep predictions: that is, starting from an initial true state s_t, we

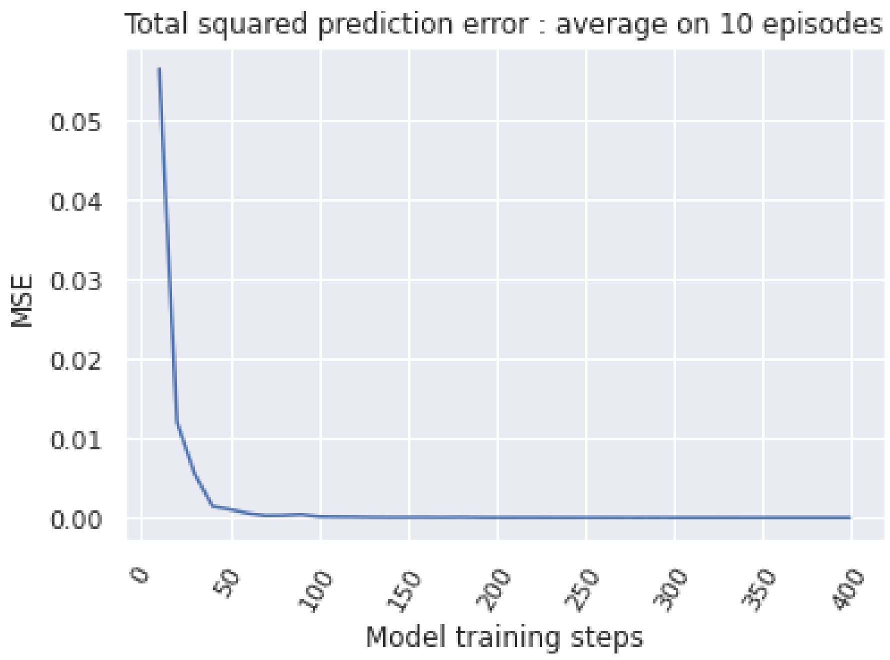


Figure 4.7: Cumulative MSE error of the single gaussian model averaged over 10 episodes with respect to the no. of training steps

use the learned model to predict the next stat, $\hat{s}_{t+1}$; then, this prediction is used to bootstrapping a predicted rollout $\hat{s}_{t+1}, \hat{s}_{t+2}, \ldots, \hat{s}_{t+h}$, where h is the *horizon* parameter. This parameter draws a tradeoff between richness in synthetic data gathering, and accuracy of the same synthetic data generated.
As we can see, values of h larger than 3 result in absolute relative error larger than 50%, which is unacceptable; therefore, in this implementation of COMBO we stuck with a value of $h = 3$.

4.4.3 Results

As for the model-free algorithms, also in this case the agents are trained for a number of 250k steps on each dataset before being tested; however, for each dataset and type of model (single or ensemble, so that in the end we have a total of 10 different models), the transition dynamics to be learned needs a different number of training epochs for each data and model-type configuration.
First, we analyze the results of COMBO equipped with the single Gaussian model (**single-COMBO**): these are depicted in Fig. 4.10 and in Fig. 4.11 . As we can see, single-COMBO performs way better than OfflineDDQN on all types of data, with the exception of the **random** and **medium-expert** datasets, where the results of

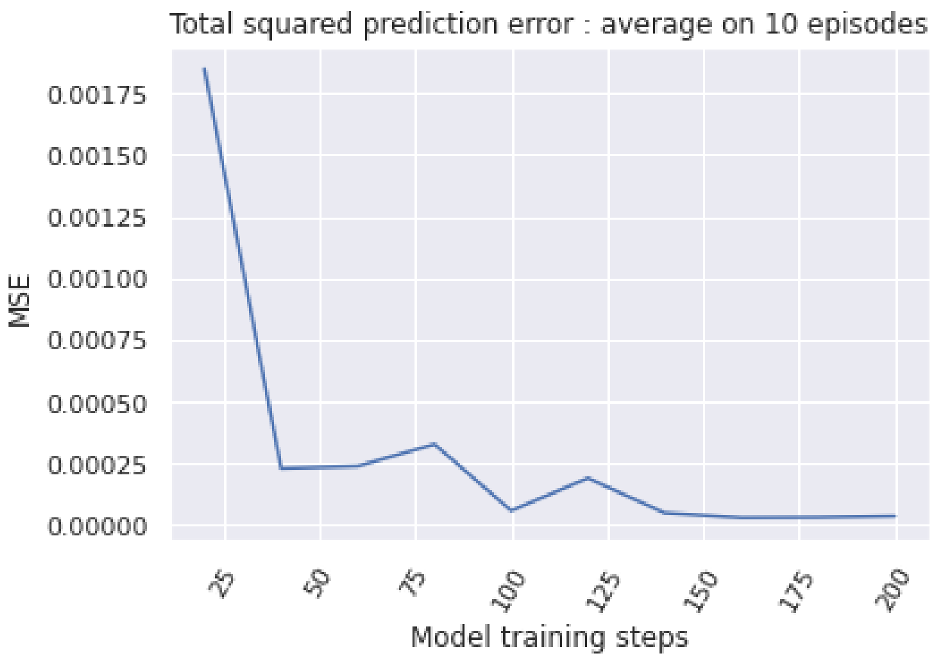


Figure 4.8: Cumulative MSE error of the ensemble model averaged over 10 episodes with respect to the no. of training steps

the model-free baseline are slightly better; what's more interesting, COMBO seems pretty insensitive to the type of dataset used in terms of final result.
Using an ensemble model produces slightly better results: the final outcome is comparable to single-COMBO, but, if we look at the Table 4.1, we can see that Ensemble COMBO is always better than Single COMBO and OfflineDDQN; moreover, the ensemble method proves to be the best one in 4 out of 5 scenarios. As for the single-Gaussian instance, also EnsembleCOMBO proves to be insensitive to the training data distribution. If we compare this observation with the hypothesis made in the case of CQL, we can suggest the idea that learning the transition dynamics and using it to augment the training dataset with synthetically-generated data can heavily boost the final performance. If this is true, then tweaking the d_f parameter in COMBO, which controls the real-synthetic data ratio, can lead to different outcomes. This analysis is illustrated in the next Section.

4.4.4 Sensitivity analysis of d_f

As we explained in Subsection 3.4.2, even when the model is suboptimally accurate, we can bias the updates towards the empirical MDP (which is the MDP represented

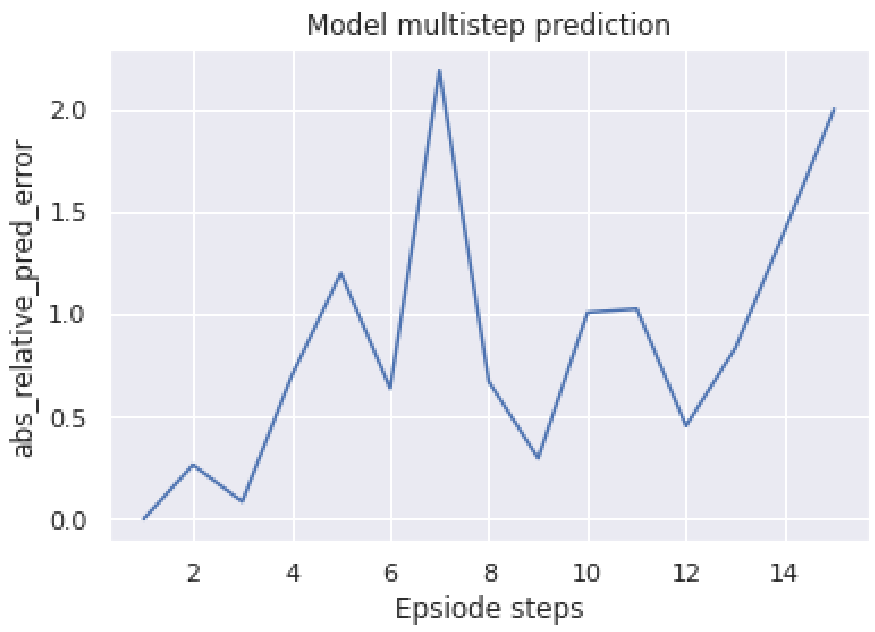


Figure 4.9: Multistep analysis of the single Gaussian model: absolute relative error in multistep predictions

Offline Test Returns (250k training steps)				
	OfflineDDQN	CQL	Single COMBO	Ensemble COMBO
Random	4856	4890	4845	4886
QLearning	-570	-2119	4857	4874
Medium	2726	-1486	4868	4892
Expert	2603	-788	4883	4886
Medium-Expert	4893	4830	4879	4899

Table 4.1: Offline Test Returns: as we can see, Ensemble COMBO is the most powerful method, as it achieves the best results in the majority of cases. In blue are highlighted the best results for each dataset

by the learned model's transition dynamics) by controlling the parameter f.
In practice, f controls the percentage of real vs. synthetic data to use for training the critic. So, if we are confident with our model's predictive power and we think the offline dataset does not adequately cover the true MDP, we can shift the value of f towards 0 to ensure that when training the critic, the larger portion of each batch is made up of synthetic data, which is data generated from the learned model.

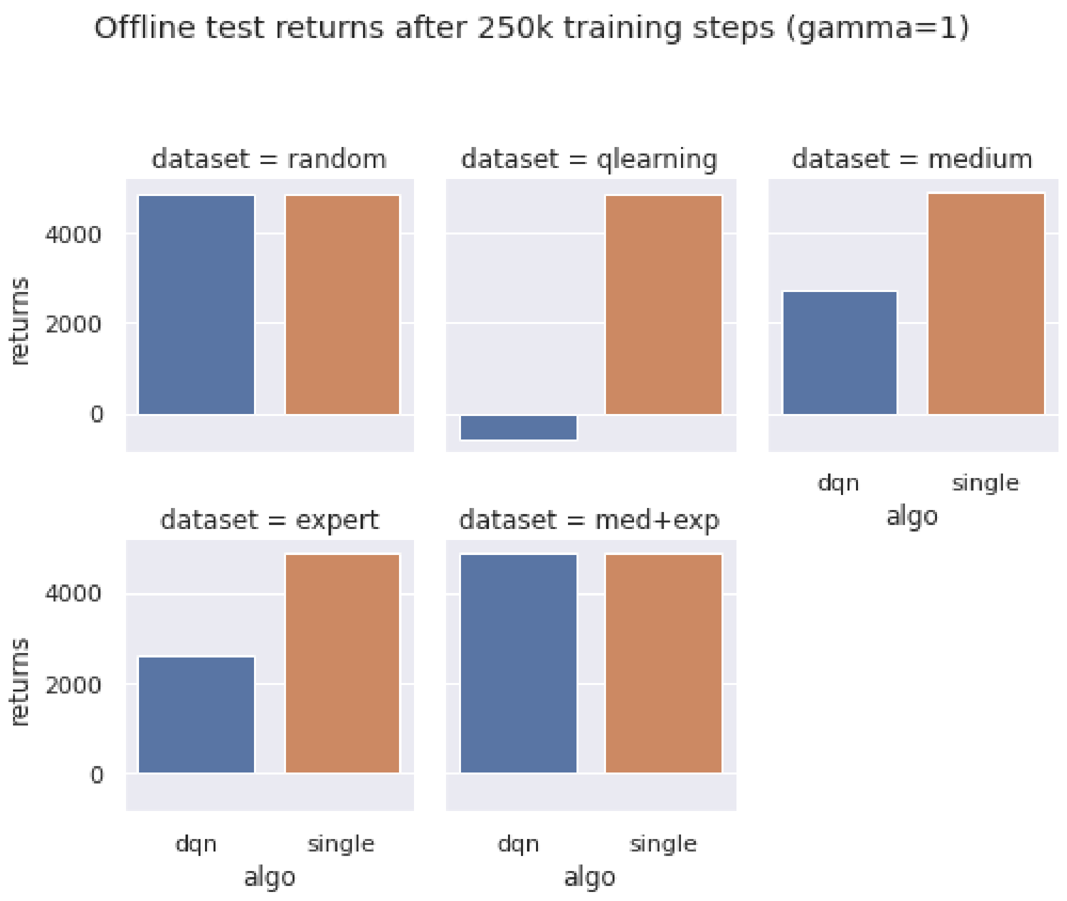


Figure 4.10: Comparison of results between single Gaussian COMBO and OfflineDDQN - Episode test returns

For our experiments, we set $f = 0.25$ (25% real data, 75% synthetic data) and $f = 0.75$. In Table 4.2 we show the results obtained with Ensemble-COMBO on a testing episode for the values of f used. As we can see, the choice of value for this parameter is critical to the success of the algorithm and so must be properly analyzed. For example, this Table shows us that while augmenting the percentage of synthetic data in training produces more or less stable results, when we set a lower value for f the performance sharply declines, especially for **random** and **qlearning** (in the latter case the final return is even negative).
What's peculiar is that for the **expert** data the opposite is true, while the **medium-expert** dataset is really not affected by this sensitivity analysis, probably because the dataset is sufficiently large and widespreaded.
Even if this analysis can't directly prove the hypothesis on the reasons of the poor performance of CQL on certain datasets, it surely gives an insight of how well these

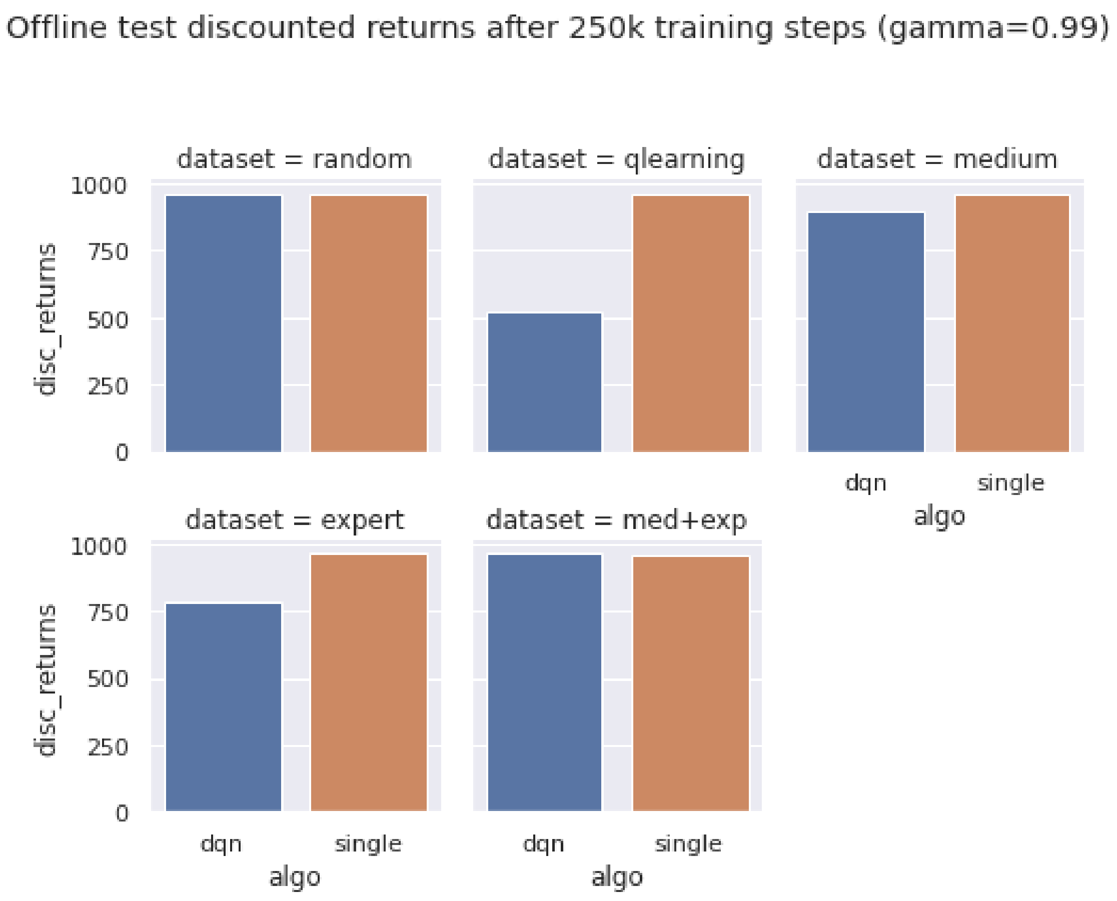


Figure 4.11: Comparison of results between single Gaussian COMBO and OfflineDDQN - Episode test discounted returns

offline algorithms can leverage the different types of data.

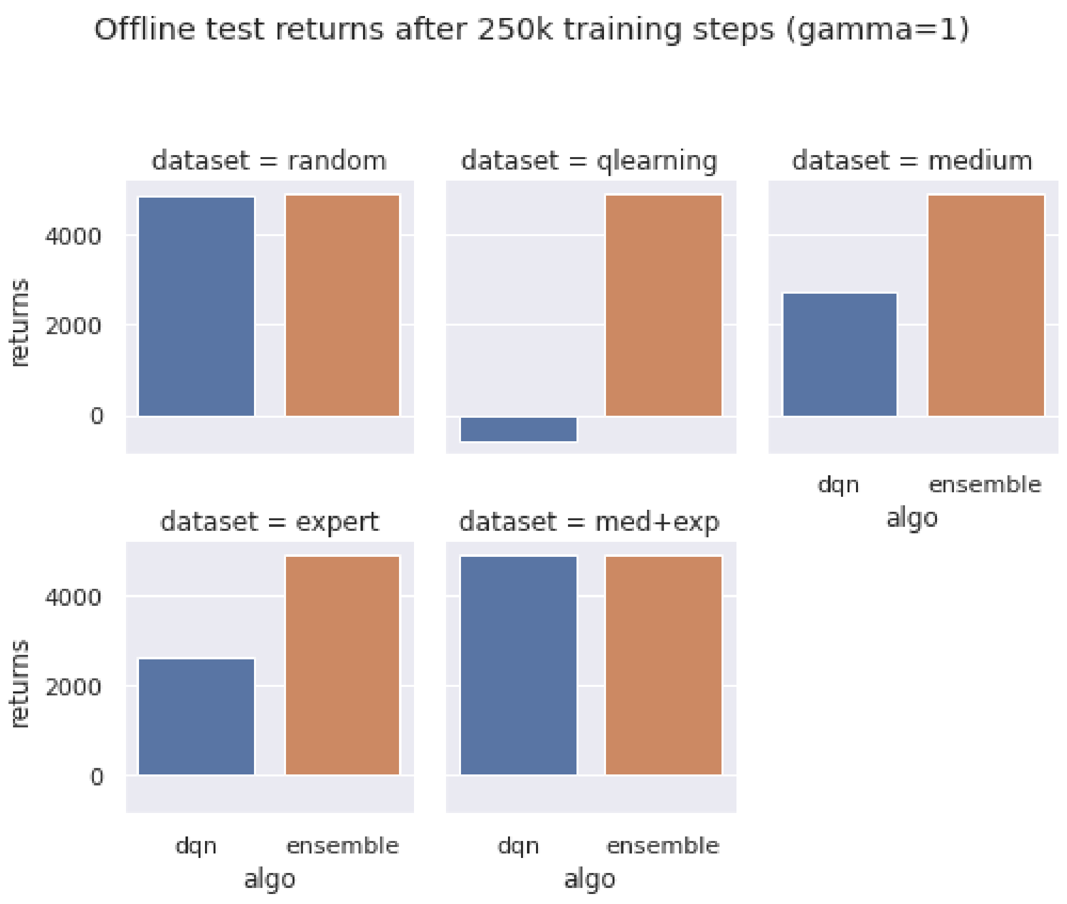


Figure 4.12: Comparison of results between ensemble Gaussian COMBO and OfflineDDQN - Episode test returns

Sensitivity Analysis of d_f (on Ensemble-COMBO)			
	$f = 0.25$	$f = 0.5$	$f = 0.75$
Random	4822	4886	1328
QLearning	3566	4874	-1556
Medium	4359	4892	3858
Expert	2096	4886	4345
Medium-Expert	4848	4899	4878

Table 4.2: Offline Test Returns as function of f: as we can see, a balanced choice of real and synthetic data, which corresponds to $f = 0.5$ delivers the best results: however, COMBO is more inclined towards using synthetic data, as we can see from the low returns (highlighted in red) obtained when the real data is predominant; an exception is the **expert** dataset, which yields opposite results

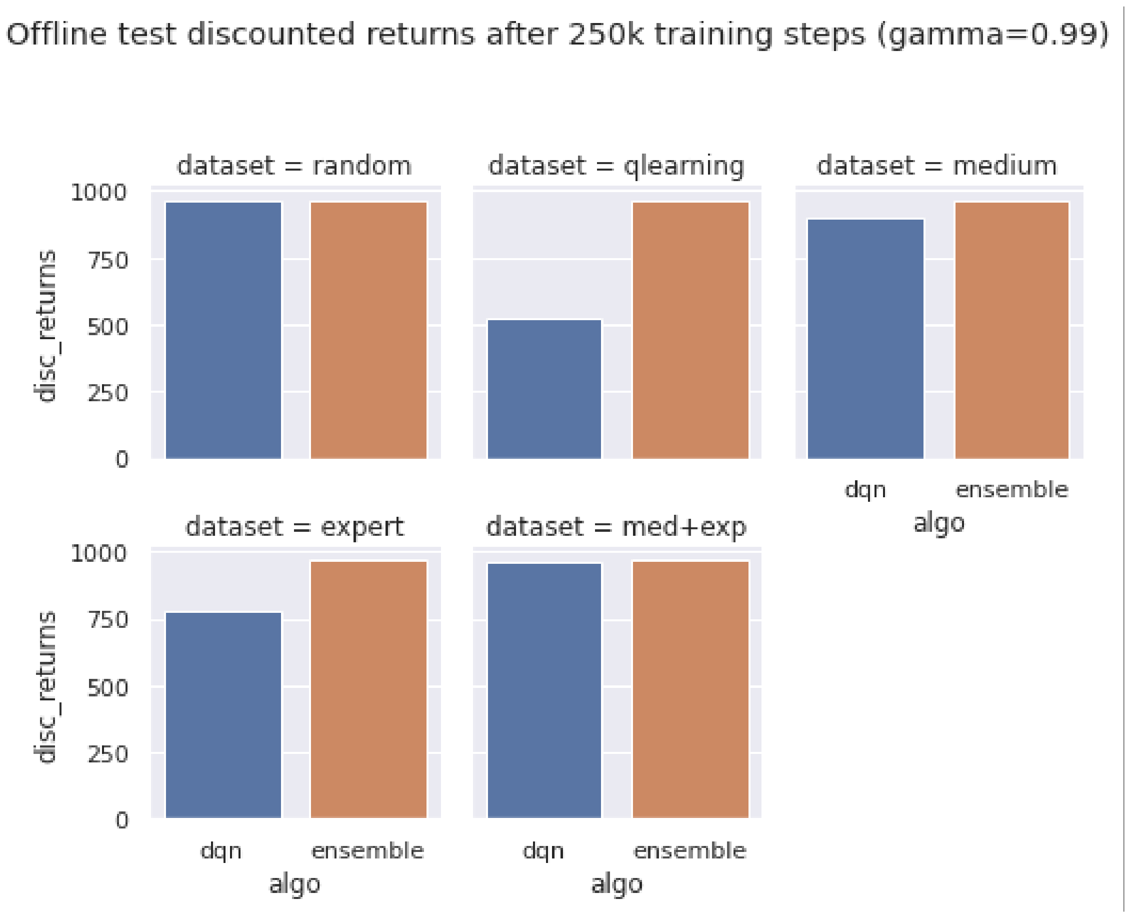


Figure 4.13: Comparison of results between ensemble Gaussian COMBO and OfflineDDQN - Episode test discounted returns

Chapter 5

Conclusions

This work aimed to investigate the power of Offline RL methods in a practical task; in particular, we tried to analyze the differences in terms of final results between **model-free** and **model-based** algorithms. In explicit, the algorithm of reference for the model-free scenario is **C**onservative **QL**earning (**CQL**), while for the model-based approach we chose **COMBO** (**C**onservative **O**ffline **M**odel-**B**ased Policy **O**ptimization): both the algorithms are state-of-the-art methods.
The task to experiment was a driving mission of 500 seconds in the **CARS** framework.
We conduct five different experiments, based on the different data collected. Using various online agents (i.e., a random policy, a policy based on QLearning and one based on DDQN), we gathered 5 different datasets: **random**, **qlearning**, **medium**, **expert** and **medium-expert**.
As a baseline to which we could compare our results, we trained an offline adaptation of DDQN called **OfflineDDQN**: this is simply an algorithm that is trained on batches sampled from the dataset, without the addition of any particular regularization approach. Although such way of procedure is known in literature not to work, in our task the agent has proven to be a mildly strong baseline, at least when trained on **random** and **medium-expert** datasets: the reason could be the simplicity of the environment and the problem related.
Following the results, we evince that CQL is an overly-pessimistic algorithm, since when trained on datasets collected from near-optimal policies (i.e., **expert** and **qlearning**), it yields poor results. Such excessively pessimistic consideration of out-of-distribution actions by the algorithm is worth of further analyses.
On the other hand, when we build a model which is able to correctly interpreting and simulating the environment, things are definitely better: this is what COMBO does.
We developed two different implementation of COMBO, which differ in the choice of model: single-COMBO employs a single Gaussian network (i.e., we assume

the transition probabilities of the environment to follow a normal distribution); ensemble-COMBO is an upgrade of single-COMBO, because the states are predicted by an ensemble of networks, which helps in reducing variance.
We conducted an analysis of the predictive power of both models: we can say that each of them is able to correctly predict the next observation given the current state-action tuple. As we expected, the second method is more powerful than single-COMBO, since it shows a better performance in every type of dataset-based scenario.
Although our DDQN baseline is slightly better than single-COMBO in 4 out of 5 scenarios, DDQN performs poorly on the **expert** dataset; on the contrary, single-COMBO yields good results in every scenario, which makes it a more reliable algorithm than the baseline.
Overall, Ensemble-COMBO delivers the best results: the algorithm performs better than the baseline in each scenario except one (**QLearning**); moreover, Ensemble-COMBO is the best method overall in 3 out of 5 scenarios.
In the end, we have seen that Offline RL proves to be rather succesful for the task chosen, although some algorithms among those used make a distinction (in terms of final result) between the type of dataset used for training. What's more, training and leveraging an approximate transition dynamics (i.e., a *model*) to train an agent pays dividends at the end of the day. Indeed, as we have shown, model-based algorithms like COMBO are insensitive to the dataset used for training, achieving near-optimal performances on all of them.

Bibliography

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning - An Introduction - 2nd Edition*. The MIT Press, 2018 (cit. on pp. 6, 19).

[2] Paolo Brandimarte. *From Shortest Paths to Reinforcement Learning - A MATLAB Based Tutorial on Dynamic Programming*. Springer, 2021 (cit. on p. 8).

[3] Sheldon Ross. *Introduction to Stochastic Dynamic Programming*. Academic Press, 1983 (cit. on pp. 10, 12).

[4] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. King's College, 1989 (cit. on p. 16).

[5] H. van Hasselt. «"Double Q-learning"». In: *NeurIPS Proceedings* (2010) (cit. on pp. 20, 21).

[6] Richard S. Sutton. *Integrated architectures for learning, planning, and reacting based on approximating dynamic programming*. Proceedings of the 7th International Workshop on Machine Learning, 1990 (cit. on p. 24).

[7] E. Fiesler. *Neural Network Classification and Formalization*. IDIAP, 2017 (cit. on p. 26).

[8] V. Mnih et al. «"Human-level control through deep reinforcement learning"». In: *Nature* 518.7540 (2015), pp. 529–533 (cit. on p. 31).

[9] Jianqing Fan et al. «"A Theoretical Analysis of Deep Q-Learning"». In: *arXiv:1901.00137* (2019) (cit. on p. 34).

[10] Levine et al. *Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems*. 2020 (cit. on pp. 36, 44).

[11] A. Kumar et al. «"Conservative Q-Learning for Offline Reinforcement Learning"». In: *arXiv preprint arXiv:2006.04779* (2020) (cit. on p. 38).

[12] *Wikipedia*. URL: `https://en.wikipedia.org/wiki/Total_variation_distance_of_probability_measures` (cit. on p. 42).

[13] Kidambi. *MOReL: Model-based offline reinforcement learning*. 2020 (cit. on p. 45).

[14] Yu. *MOPO: Model-based offline policy optimization.* 2020 (cit. on p. 45).

[15] T. Yu et al. «"COMBO: Conservative Offline Model-Based Policy Optimization"». In: *arXiv preprint arXiv:2102.08363v1* (2021) (cit. on pp. 45, 48).

[16] C. Maino et al. «"Project and Development of a Reinforcement Learning based Control Algorithm for Hybrid Electric Vehicles"». In: *applied sciences* (2021) (cit. on pp. 51, 52).

[17] G. Van Rossum. *The Python Lybrary Reference, release 3.8.5.* (Cit. on p. 52).

[18] MathWorks. *MATLAB Documentation Center.* URL: `http://www.mathworks.it/it/help/matlab/` (cit. on p. 52).

[19] *OpenAI Gym.* URL: `https://gym.openai.com/` (cit. on p. 53).

[20] *Berkeley Artificial Intelligence Research.* URL: `https://bair.berkeley.edu/blog/2020/12/07/offline/` (cit. on p. 54).