# POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering


Master's Degree Thesis

# Offline Reinforcement Learning for Smart HVAC Optimal Control

Supervisors

Prof. Francesco VACCARINO

Candidate

Filippo CORTESE

Academic year 2021-2022

# Summary

Deep Reinforcement Learning provides a mathematical formalism for learning-based control. It presents an agent that, by a trial and error approach, learns how to behave optimally in an environment. Deep Reinforcement Learning has in this *online learning* paradigm one of the biggest obstacle to its widespread adoption. In many settings the interaction between the agent and the environment is either impractical or too dangerous, for example in the healthcare or autonomous driving domain. Offline Reinforcement Learning tries to overcome this issue by proposing a new paradigm, where the learning happens from a fixed batch of previously collected data. Removing the online interaction makes this data-driven approach scalable and practical but introduces also some issues for the learning process. The first is that learning rely completely on the static dataset composition, if this does not cover enough high reward regions, it may be impossible for the agent to learn how to behave optimally. The second is the out of distribution actions overestimation. Actions that are never seen in the data are keen to be overestimated by the agent, that without the reward feedback, can't correct its wrong estimates. This thesis aims at studying in depth the Offline RL approaches with a focus on algorithms that do minimal changes to state-of-the-art deep RL algorithms. Then it will focus on evaluating this approach on a real-case scenario like the smart HVAC control, where the data available is either limited in size or in exploration. To pursuit these objectives we started from a state of the art continuous-action offline RL algorithm, called TD3-BC, and derived a discrete-action algorithm that we call TD4-BC. We compared the two algorithms on a dual action nature environment called LunarLander and tested TD4-BC on the smart HVAC control task. Finally, an additional online fine-tuning approach to TD4-BC is tested on the HVAC environment. The obtained results show comparable performance for TD4-BC with respect to TD3-BC on LunarLander and promising results on the HVAC task, especially with the addition of online fine-tuning. Overall, Offline RL proved to be a powerful paradigm to tackle both a well known benchmark environment and an industry related case, with many open spaces for possible future improvements.

# Acknowledgements

I would like to thank my supervisor Prof. Francesco Vaccarino for giving me the opportunity to do this thesis and for the knowledge and support offered during its writing.

Then I would like to thank my company supervisor Luca Sorrentino for his valuable advice, support and patience during this experience; his help was crucial for me and the development of this thesis. I would also like to express my gratitude to Rosalia Tatano, for all the work and precious help she has given to me during the last phases of this work and to AddFor Industriale S.r.l. for giving me the opportunity to work with them and for letting me access to their infrastructures and knowledge.

A special thanks goes to my family for all the support they gave me in the past years and for always letting me do what I felt was right for me. Finally I would like to to express my deepest gratitude to Lara, for everything she has done for me in the last four years and for always being supportive even when I didn't deserve it.

# Table of Contents

# List of Tables

# List of Figures

# Acronyms

**AI**
Artificial Intelligence

**RL**
Reinforcement Learning

**ML**
Machine Learning

**DL**
Deep Learning

**Deep RL**
Deep Reinforcement Learning

**Offline-RL**
Offline Reinforcement Learning

**HVAC**
Humidity, Ventilation, and Air Conditioning

**MDP**
Markov Decision Process

**FOMDP**
Fully Observable Markov Decsion Process

**POMDP**
Partially Observable Markov Decsion Process

**GPI**

Generalized Policy Iteration

**TD**

Temporal Difference

**FQI**

Fitted Q-Iteration

**NN**

Neural Network

**BC**

Behavioral Cloning

**DQN**

Deep Q-network

**DDQN**

Double Deep Q-network

**TD3**

Twin Delayed Deep Deterministic Policy Gradient

**TD4-BC**

TD3-BC Discrete

**OOD**

Out of Distribution

**SGD**

Stochastic Gradient Descent

**MLP**

Multilayer Perceptron

# Chapter 1

# Introduction

**Reinforcement Learning** (RL) it's a computational approach to the process of learning from interaction. Its core idea consists of a learner, usually called agent, that by interacting with the environment, learns how to behave optimally with respect to an objective. The agent is not told which actions to take, but must discover by itself which actions are good and which are not, by exploiting the reward, a feedback signal that it receives after each action is taken.

With the growing field of Deep Learning in the last years, many algorithms that tried to combine DL (DL) with RL were proposed. This approach, called Deep Reinforcement Learning, has obtained near-human performances on many Atari games, has beaten the world best player of one of the most complex board game, Go, and has reached optimal performances on many other real and simulated tasks. Besides being responsible for these successes, the online nature that characterizes RL is what stopped its adoption in many other applications. In many industrial or healthcare tasks, the possibility of making very coarse mistakes, that the online interaction can bring, is not accepted, either because too costly or impractical, or because potentially unsafe for humans.

To solve this issue, recently a new paradigm called **Offline Reinforcement Learning** was formalized. In this approach, the agent learns how to behave entirely from a fixed dataset of interactions previously collected. This has many practical advantages, starting from the possibility to exploit big amounts of data that are already collected and available, as well as reducing significantly the computational cost for training, since it removes completely the time used for the interaction with the environment. The goal of this thesis is to derive a discrete-action control version of the state-of-the-art Offline-RL algorithm TD3-BC, that we will call TD4-BC, and test its behavior on a practical case of smart HVAC optimal control. Heating,Ventilation and Air Conditioning (HVAC) systems are used to control the temperature and quality of the air in an enclosed space. Smart HVAC optimal control aims at reducing the primary energy consumption of these systems while

maintaining the same level of thermal comfort guaranteed.

Unfortunately, Offline RL paradigm brings some intrinsic issues with it. The first and more obvious is that it is completely dependent on the dataset on which it is trained. If the dataset is not well collected or it is not enough representative of optimal behaviours, the whole learning process is affected. In this work, we evaluate our algorithm TD4-BC with datasets with poor coverage of actions and state to assess their role on the offline agent performance.

The second core issue is the fact that the whole point of Offline-RL is to outperform the policy, or policies, that collected the dataset, thus in its learning process an offline agent has to deviate from them. In doing this, naive Offline-RL algorithms suffer of overestimation of out of distributions actions, that brings to a degradation of the final performance. In this thesis we study one algorithm TD3-BC that tries to address this issue and we derive a discrete-action version from it. We then analyze the smart HVAC optimal control task, where we want to obtain a scalable and practical method to optimize the HVAC control, without the need of heavy customization. The derived algorithm, named TD4-BC, is tested and evaluated on this particular task with datasets, derived from already implemented control algorithms, that are scarce in action representation or have limited state coverage. The experiments carried out in this thesis were done in collaboration with AddFor Industriale S.r.l.

This thesis is organized as follows. Chapter 2 presents the Reinforcement Learning main concepts and introduces to its mathematical framework. Chapter 3 introduces to the concept of deep learning and how is used in RL to obtain Deep Reinforcement Learning algorithms. Some of the main Deep RL algorithms are presented. Chapter 4 presents the Offline Reinforcement Learning, its advantages and its issues, as well as one state-of-the-art algorithm called TD3-BC. It then focus on the derivation of a discrete-action version of TD3-BC, that we call TD4-BC and proposes some paired online fine-tuning approaches. Chapter 5 focus on the environments used for our experiments, LunarLander and HVAC, and on the generation of the datasets used for our offline training. Chapter 6 firstly compare TD4-BC performance on LunarLander with respect to TD3-BC; then it focus on evaluating TD4-BC on the HVAC control task. It also evaluate TD4-BC with a slightly different training paradigm consisting of Offline training plus Online fine-tuning. Chapter 7 discusses the results of the experiments as well as possible future improvements.

# Chapter 2

# Reinforcement Learning

Reinforcement learning is a computational approach to learning from interaction. Its main goal is to solve sequential decision making problems through a trial and error approach. In this chapter we will show the theory and formalism behind Reinforcement Learning and the main types of algorithms that try to solve the RL problem. Further references can be found in chapter 3,4,5,6 of [1].

## 2.1 The Agent-Environment Interface

The **Reinforcement learning** (RL) approach is based on the interaction between two entities, the **agent** and the **environment**. The agent is both the learner and the decision maker. The environment, instead, represent the world where the agent takes its actions. A **Markov Decision Process** (MDP) is a classical formalization of such sequential decision making process and can describe almost all RL problems. The interaction between agent and environment happens at discrete time step $t = 1,2,3...$ . In each of these time step the agent can take an **action** $a_t$ from a set of possible actions $A$, called Action Space end the environment can assume a **state** $s_t$ from a set of possible states $S$, called State Space. The fundamental aspect of a MDP is that the evolution of the environment between states respects the **Markov property**: "The future is independent of the past given the present". This property can be expressed through conditional probabilities in this way:

$$\mathbb{P}[s_{t+1}|s_t, a_t] = \mathbb{P}[s_{t+1}|s_1, s_2, ..., s_t, a_t] \tag{2.1}$$

This means that the probability for the environment to transition from state $s_t$ to a generic state $s_{t+1}$, given the complete knowledge about its states history, is the same if the knowledge is restricted to just the previous state $s_t$. To gain insight on this property we can think of it as the fact that each state must contains all the information that is relevant for the evolution of the environment. This key

assumption in RL it's not easy to make in actual applications and is often addressed through a good engineering of the state. For example by ensembling also past information in the state we can still work with a MDP.

From Eq.(2.1) we can see how the transition between states is also influenced by the action $a_t$ that the agent takes. From the agent perspective, the interaction evolves in this way: at each time step $t$ the agent receives some representation of the environment state $s_t$ which is usually called observation $o_t$, and select which action $a_t$ to take from a set of possible actions $A$. When the observations $o_t$ and state $s_t$ coincide, the MDP is called Fully Observable Markov Decision Process (FOMDP) , otherwise is called Partially Observable Markov Decision Process (POMDP). In this work from now on we will assume to treat FOMDPs, then we will simply refer to it as MDPs for simplicity.

What determines how the agent choose which action to perform is the **policy function** $\pi$. The policy is basically a mapping function from states $s_t \in S$ to probabilities of selecting one action $a_t$ from the action space $A$. We can then interpret it also as the conditional probability of taking action $a_t$ given that the agent is in the environment state $s_t$:

$$\pi(s, a) = \mathbb{P}[a_t|s_t] \tag{2.2}$$

Having performed the action determined by the policy $\pi$, at the next time step, the agent will receive from the environment a scalar value $r_t$, called Reward. The reward can be thought as a feedback signal telling the agent how good was the action it just took and we can formally define it as a mapping function, called **Reward Function**, from tuples of state, actions and next state to scalar values: $R(s_t, a_t, s_{t+1}) \rightarrow \mathbb{R}$. Each step of this interaction process can be thought as tuple called **Transition**: $s_t, a_t, r_t, s_{t+1}$. The interaction process can be graphically summarized in Fig.2.1.

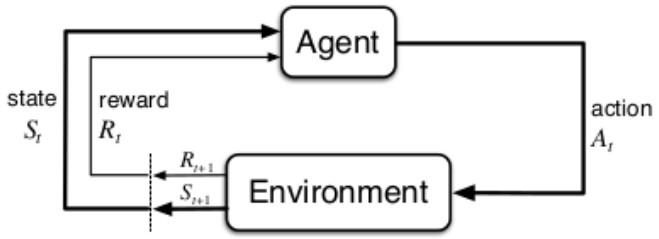



**Figure 2.1:** Agent-Environment Interaction scheme. (Image taken from [1])

Usually the MDP presents some desired states that, once reached, determine its end. Then the list of all transitions happened from the initial state to the goal state is called **Episode**. A particular class of MDP, called Infinite MDP, does

not have a final state but accept an infinite evolution over time, but will not be specifically treated in this work.

We can then introduce the concept of cumulative reward from time step $t$ to final step $T$:

$$G_t = R_t + R_{t+1} + R_{t+2} + ... + R_T \tag{2.3}$$

Usually in RL another parameter $\gamma \in [0,1]$ , called **discount factor**, is introduced in addition to the MDP. Its purpose is to favour the immediate reward with respect to the future reward. Then the cumulative discounted reward can now be defined as:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ... = \sum_{k=t}^{T} = \gamma^k R_k \tag{2.4}$$

There are many different motives for this, firstly it makes mathematically tractable the Infinite MDP, but more importantly it express some uncertainty about the future and about future estimates: the more we go into the future the less we care about it. Uncertainty about future estimate will be often the case for many of the algorithms we will see lather in the chapter.

RL itself is based on the **Reward Hypothesis**: *All goals can be described by the maximization of the expected Cumulative Reward.* In fact the purpose of a RL agent, assuming a parametric policy function with parameters $\theta(\pi_\theta)$, is to find the parameters that maximizes the expected cumulative discounted reward:

$$\pi^*_\theta = \arg\max_\theta \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{k=t}^{T} \gamma^k R(s_t, a_t)\right] \tag{2.5}$$

with $\tau$, the **trajectory**, being the complete sequence of states and action of a general episode: $s_1, a_1, s_2, a_2, ..., a_{T-1}, s_T$.

To summarize we can then describe a MDP by five main entities:

- a finite set of states $S$ called State Space
- a finite set of action $A$ called Action Space
- a Transition Probability Function $P(s, a, s') = \mathbb{P}[s' = s_{t+1}|s = s_t, a = a_t]$
- a Reward function $R(s_t, a_t, s_{t+1}) \in \mathbb{R}$
- a discount factor $\gamma \in [0,1]$

## 2.2 Solving Reinforcement Learning Problem: Prediction vs Control

In this section we will present the two principal problems that MDP presents, the prediction and control problem. The prediction problem basically consists in how

to evaluate a policy and it is needed to solve also the control problem, that is, instead, to find the optimal policy for a MDP.

### 2.2.1 Prediction Problem

This problem consists of evaluating a policy function in an unknown MDP. To do this we need to introduce a metric function useful to evaluate each state, called **Value Function**. The Value Function estimates how good for an agent is to be in a certain state following a certain policy $\pi$, it is like a proxy for what the agent will encounter in its future transitions following the policy of interest $\pi$. It does this in terms of expected future discounted reward and can be defined as:

$$\begin{aligned} &V_\pi : S \to \mathbb{R} \\ &V_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s] \end{aligned} \tag{2.6}$$

Starting from this definition and exploiting Eq.(2.4) it's possible to derive an iterative formulation:

$$\begin{aligned} &V_\pi(s) = \mathbb{E}_\pi[G_t | s_t = s] \\ &V_\pi(s) = \mathbb{E}_\pi[R_t + \gamma G_{t+1} | s_t = s] \\ &V_\pi(s) = \textstyle\sum_a \pi(a|s) \sum_{s'} P(s,a,s') \Big[ R(s,a,s') + \gamma \mathbb{E}_\pi[G_{t+1} | s_{t+1} = s'] \Big] \end{aligned} \tag{2.7}$$

To get in the end:

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s,a,s') \Big[ R(s,a,s') + \gamma V_\pi(s') \Big] \tag{2.8}$$

The Eq.(2.8) is the **Bellman Expectation Equation** for the Value Function and it shows how value function can be decomposed in two parts, an immediate reward and the discounted value from the next state onward. If the transition probability function $P(s,a,s')$ is known, this equation can be solved directly but this is not always computationally feasible since its cost is dependent on the number of states in the state space $S$. We will present one alternative solution later in this work.

### 2.2.2 Control Problem

The control problem instead consists in finding the policy $\pi^*$ that has maximum expected cumulative reward or in other words that is linked to the maximum Value Function $V_\pi^*(s)$. Actually we need first to introduce another function the Action-value function, also called **Q-Value Function**, defined as:

$$\begin{aligned} &Q_\pi : S \times A \to \mathbb{R} \\ &Q_\pi(s,a) = E_\pi[G_t | s_t = s, a_t = a] \end{aligned} \tag{2.9}$$

This means that the Q-value is equal to the expected cumulative reward the agent could get since it has taken action $a$ in state $s$ and following policy $\pi$ for the rest of its episode. In other words the Q-value can be interpreted as a metric of how good is taking a specific action $a$ being in a state $s$. We can apply same reasoning previously used for the Value Function to obtain the Bellman Expectation Equation for Q-value function:

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi\Big[R_t + \gamma G_{t+1} | s_t = s, a_t = a\Big] \\ Q_\pi(s,a) &= \textstyle\sum_{s' \in S} P(s,a,s') \Big[R_t + \gamma \sum_{a' \in A} \pi(s',a') \Big[\mathbb{E}_\pi[G_{t+1} | s_{t+1} = s', a_{t+1} = a']\Big]\Big] \\ Q_\pi(s,a) &= \textstyle\sum_{s' \in S} P(s,a,s') \Big[R_t + \gamma \sum_{a' \in A} \pi(s',a') \Big[Q_\pi(s',a')\Big]\Big] \end{aligned} \tag{2.10}$$

As we said in control problem we need to find the optimal policy for a MDP, that means that if we find the optimal Q-value function $Q^*_\pi(s,a)$ then we get also the optimal policy $\pi^*$ for free, because in every state we know the best action to take. We can write the optimal action-value function as:

$$Q^*_\pi(s,a) = \max_\pi Q_\pi(s,a) \tag{2.11}$$

To obtain this optimal Q-value function we need to derive the **Bellman Optimality Equation**, to do this we start from the optimal value function:

$$V^*_\pi(s) = \max_a Q^*_\pi(s,a) \tag{2.12}$$

expressed in terms of Q-values and the optimal Q-value function expressed in terms of Value Function:

$$Q^*_\pi = \sum_{s' \in S} P(s,a,s') \Big[R(s,a,s') + \gamma V^*_\pi(s')\Big] \tag{2.13}$$

. Putting this together we get the Bellman Optimality Equation:

$$Q^*_\pi(s,a) = \textstyle\sum_{s' \in S} P(s,a,s') \Big[R(s,a,s') + \gamma \max_{a'} Q^*_\pi(s',a')\Big] \tag{2.14}$$

This equation is non-linear because of the *max* operator and generally it does not present a closed form solution.

Now we will present one simple iterative algorithm that, assuming to know the Transition Probability Function $P(s,a,s')$, can solve the Bellman Optimality Equation. It's called **Generalized Policy Iteration** (GPI) and it consists of two interactive parts. The first is called **Policy Evaluation** and it's basically the same process described in subsection 2.2.1, but here the objective is to to evaluate a Q-value Function given a policy $\pi$ over a MDP. The second part is called **Policy**

**Improvement** and its consists of improving the policy $\pi$ with respect to the just computed Q-value Function:

$$\pi' = \arg\max_{a \in A} Q_\pi(s, a) \tag{2.15}$$

The algorithm continues to iterate over these two process obtaining a new Q-value function improvement for each step until the convergence to the optimal Q-value function $Q^*$ and policy $\pi^*$.

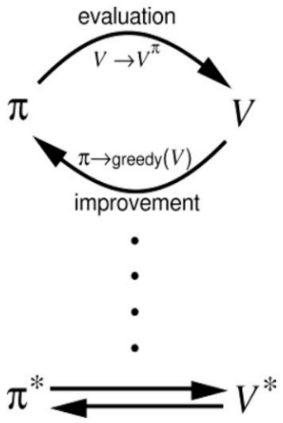



**Figure 2.2:** General Policy Iteration scheme. (Image taken from section 4.6 of [1]

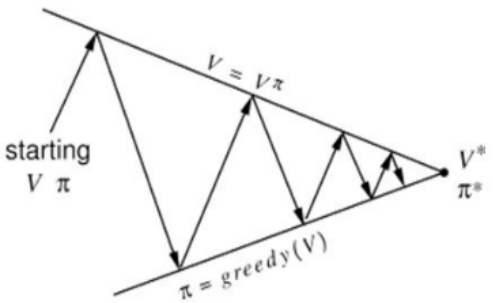



**Figure 2.3:** General Policy Iteration convergence scheme. (Image taken from section 4.6 of [1]

## 2.3 Taxonomy of Reinforcement Learning Algorithms

In this section we will presents the main categories of RL algorithms that tries to solve the MDP control problem when the Transition Probability Function $P(s, a, s')$ is not known and we will describe in particular some specific algorithms. We will restrict our focus on Temporal Difference Methods (TD) generally referred as TD Learning algorithms. TD methods can learn directly from raw experience without a model of the environment dynamics, (no Transition Probability Function) without having to wait to complete episodes of experience to improve their Value estimates. In fact they update their estimates of the Value Functions by exploiting both real experience and the estimate itself, in a process called Bootstrapping.

### 2.3.1 TD Prediction

The simplest TD method is for evaluating a policy in terms of Value or Q-value and is based on this bootstrapped update. Starting from an initial Value function estimate $V_\pi(s, a)$ we let the agent collect real experience from the environment and construct the **TD Target** as the sum of the reward just sampled and the discounted current Value estimate for the next state: $R_t + \gamma V_\pi(s_{t+1})$. The TD target is simply a slightly better estimate of the value function and the algorithm moves its estimate towards this value. To do this the estimate is actually moved towards the **TD Error** that is the difference between the TD target and current old Value Function estimate:

$$R_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t) \tag{2.16}$$

The amount of the update is regulated by a hyperparameter $\alpha$ called learning rate. Putting all together we obtain the TD prediction update:

$$V_\pi(s_t) \leftarrow V_\pi(s_t) + \alpha\Big[R_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)\Big] \tag{2.17}$$

Same structure for the Q-value function:

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha\Big[R_t + \gamma Q_\pi(s_{t+1}, a_{t+1}) - Q_\pi(s_t, a_t)\Big] \tag{2.18}$$

This method is still considered as Tabular Method since it assume to have finite state space $S$ and action space $A$, and that the Value Function estimate can be stored in a simple table. Most of the TD control methods are based on the TD update, we will now show the main categories of these methods and describe some practical algorithms. There are three main type of algorithms:

- Value Based: algorithm that exploits Value Function or Q-value function without the mean of an explicit policy

- Policy Based: algorithm that has no explicit Value Function and tries to optimize the policy by directly searching in the policy parameters space
- Actor Critic: algorithm that put together the previous methods, presenting both a Value Function and a Policy Function. Usually composed by two phase similar to GPI, a Value Function estimate and a subsequent Policy Update phase that exploits the freshly evaluated Value Function.

### 2.3.2 Q-Learning

We will now describe Q-learning, a simple TD control algorithm that exploits the TD Update, for further information refer to the author's paper [2]. Q-learning assumes to let an agent act in the environment following any arbitrary policy $\pi^{\beta}$, called Behavioral Policy, and collect transitions $\tau = (s, a, s', r)$ that are used to update its Q-function estimate. Its objective is to keep improving its Q-value estimates until the optimal ones $Q^*(s, a)$ are reached. Then it can also obtain for free an optimal policy $\pi^*$, by simply selecting for each state the action that has maximal Q-value:

$$\pi^* = \arg\max_{a \in A} Q^*(s, a) \tag{2.19}$$

This kind of algorithms where the policy learned is not necessarily the same used to collect experience are defined as **Off-Policy** algorithms. The only assumption needed for the convergence of this algorithm in the tabular case is that all state,action pairs are continued to be updated. To address this requirement usually as Behavioral Policy is used an $\epsilon-Greedy$ policy. $\epsilon-Greedy$ policy behave greedily with respect to the Q-values most of the time, but with probability $\epsilon$ instead select randomly an action from the Action space $A$. Q-learning starts from the already seen in Eq.(2.14) Bellman Optimality Equation, but modifies it substituting the Expectation with a single trajectory sample:

$$Q(s, a) = R(s, a, s') + \gamma \max_{a' \in A} Q(s', a') \tag{2.20}$$

Then it uses this Q-value target in a classic TD Prediction update like the one seen in Eq.(2.17):

$$Q(s, a) \leftarrow Q(s, a) + \alpha\Big(R(s, a, s') + \gamma \max_{a' \in A} Q(s', a') - Q(s, a)\Big) \tag{2.21}$$

### 2.3.3 Fitted Q-Iteration

We will now describe another Off-Policy Value Based algorithm, that is strictly related to Q-learning: Fitted Q-value Iteration (FQI). More information and

**Algorithm 1** Q-learning: off-policy TD control algorithm to find optimal Q-value function $Q^*_\pi(s,a)$

Initialize $Q(s,a)$ randomly for all $s \in S, a \in A$, except for $Q(s,*)$ where s is terminal state
**for** each episode **do**
  Initialize $s$
  **while** episode not ended **do**:
    Choose $a \in A$ with $\epsilon - greedy\pi^\beta$
    Take action $a$ and observe consequent reward $R_t$ and next state $s'$
    Update Q-value estimate:
    $Q(s,a) \leftarrow Q(s,a) + \alpha\Big[R_t + \gamma \max_{a' \in A} Q(s',a') - Q(s,a)\Big]$
  **end while**
**end for**

demonstration can be found in [3]. FQI is a modified version of Fitted Value Iteration (info in [4]) for working with the Q-value function. FQI makes the same assumption as Q-learning but works in what is called batch-mode and is then not an Online algorithm. Differently from Q-learning FQI is not a tabular method but actually present a function $\hat{Q}(s,a) \rightarrow \mathbb{R}$ that maps state,action pairs to a scalar value, as the Q-value estimator. FQI assumes to have a dataset of transitions $\tau = (s,a,s',r)$ that exploits to calculate target Q-values with the same equation of Q-learning:

$$Q(s,a) = R(s,a,s') + \gamma \max_{a' \in A} Q(s',a') \tag{2.22}$$

Then it use these Q-values estimate obtained from the dataset as target values for a regression problem to update its Q-value function. Usually FQI needs to add some on-policy transition to the dataset to reach convergence to good performance.

**Algorithm 2** FQI: off policy batch TD control algorithm to find optimal Q-value function $Q^*_\pi(s,a)$

Given data set $D$ of transitions $\tau = (s,a,s',r)$
Start with $\hat{Q}(s,a) = 0 \forall s, a \in SxA$ ▷ Initial Q-value function equal to 0 for all states and actions
**while** stopping condition not reached **do**:
  $Q(s,a) = R(s,a,s') + \gamma \max_{a' \in A} Q(s',a'), \forall (s,a,s',r) \in D$ ▷ Calculate Q-value targets for all trajectories $\tau$ in the data set $D$
  Feed to the regression algorithm the targets to fit a new $\hat{Q}(s,a)$
**end while**

## 2.4 Exploration vs Exploitation

In the Q-learning algorithm seen in section 2.3.2 the key assumption for convergence to optimal Q-values was that all state and action pairs were to keep updated during the learning process. To better understand this requirement we need to deepen on one of the main dilemma in RL: the Exploration versus Exploitation trade-off. Exploration is the process of favour an unexplored choice to gather more information, but potentially lose some reward, instead Exploitation is to simply make the best decision that the agent can do given its actual belief. This exploration issues arise since the RL agent learns **online**. In this setting, the action, that the agent chooses, affects which states the latter will see and which data will collect. In this like trial and error process, sometimes for the agent is worth to take new or different actions, to get new data and sometimes it's better if it simply exploits its experience with the environment, without loosing to much reward. There are two main spaces where the agent can explore: State-Action space $S \times A$ and the parameter space when the agent uses some policy or value parametric function. In this works we will address the first kind of exploration. One simple and very naive solution that often works well in practice to this issue is the Random Exploration approach. Like the name suggest this simply consists in to add some random exploration to the learning algorithms. For example the $\epsilon$-greedy policy approach that we have seen in the Q-learning algorithm 1, simply add the possibility of taking a random action with probability equal to $\epsilon$. In continuous-action domain, instead, one solution, is to add Gaussian noise to the action that the agent chooses. We will see an example of this technique in the TD3 algorithm that we will present in chapter 3.

# Chapter 3

# Deep Reinforcement Learning

In this chapter we will firstly give a brief introduction on what are Deep Learning and Neural Networks and then present their use in Reinforcement Learning. We will then describe some of the principal Deep RL algorithms.

## 3.1 Introduction to Deep Learning

In the book Deep Learning [5], the authors describe machine learning as the capability of algorithms to acquire their own knowledge, by extracting patterns from raw data, without the need for humans to formally specify it. Usually ML algorithms are used to learn how to solve tasks that are too difficult for humans or that are intuitive for humans but that classic hard-coded algorithms fails to solve. There are various types of problems that can be solved with machine learning: classification,regression, anomaly detection, clustering and many more. The types of tasks that can be solved is strictly related to the type of data that the algorithms can see.

- **Supervised Learning**: when data comes with labels (for ex. classification task) or target values (for ex. regression task)
- **Unsupervised Learning**: when data does not come with label or target values, usually the case of tasks in which the algorithm look autonomously for trends or patterns in data (for ex. clustering).
- **Reinforcement Learning**: as seen before where the algorithm is in charge of collecting its own data trough interaction with its environment

In this thesis we will only use ML algorithms for Supervised Learning problems and for Reinforcement Learning, thus we will focus on these kind of approaches.

### 3.1.1 Supervised Learning Setting

The two main type of task that falls in the Supervised Learning categories are:

- Classification: the objective is to predict a discrete class label from the observed features. Then the algorithm tries to train a function approximator to mimic a function that maps the features of each data point $x \in \mathbb{R}^n$ to a specific class y from the $K$ classes available: $\hat{f} : \mathbb{R}^n \to \{1, ..., K\}$.
- Regression: the objective is to predict a continuous real value from features that characterize the data point. Then the algorithm tries to train a function approximator to mimic a function that maps the features for each data point $x \in \mathbb{R}^n$ to a real value: $\hat{f} : \mathbb{R}^n \to \mathbb{R}$.

A generic Supervised Learning ML approach will present three main components:

- The model (function approximator) that is in charge of extracting useful knowledge and to perform the desired task
- The Metric that measure the performance of the algorithm on the desired task. Is usually the quantity that is maximized or minimized in the training process
- The Dataset that contains all data samples that can be seen by the algorithm and the labels or target data associated. Usually is divided in three set: Training, Validation and Test, to evaluate the model on unseen data both during and after the training phase.

The dataset separation in these sets are a crucial component of a machine learning procedure. This because of the **Overfitting** phenomenon, one of the main problem in this field. In the book Machine Learning [6], the author Tom Mitchell propose a definition for overfitting: *"Given an hypothesis space $H$, a hypothesis $h \in H$ is said to **overfit** the training data if there exists some alternative hypothesis $h' \in H$, such that $h$ has smaller error than $h'$ over the training examples, but $h'$ has a smaller error than $h$ over the entire distribution of instances"*. Where the hypothesis space $H$ represent the function approximator class and the hypothesis $h, h'$ are different instances of that class. Often the overfitting issue is referred to as the **Bias and Variance trade-off**, where high variance means a model that overfits the training data and fails to understand the real underlying data distribution. High bias, on the other hands, means that the bias inherited with the model error is prevalent and then fails to learn anything from the data (**underfitting**). The goal in machine learning is to find a good balance between the two, we can see a graphical example

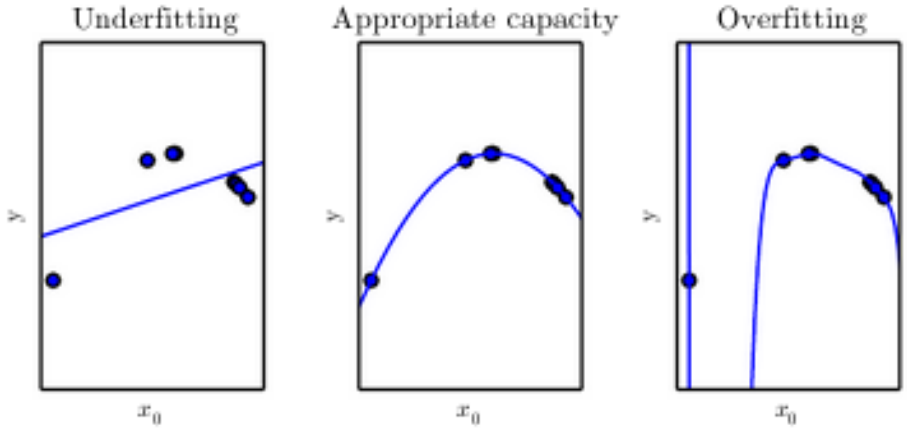



**Figure 3.1:** Bias vs Variance trade-off graphical representation, image taken from [5]

of this in Fig.3.1. Then is easy to understand why in such setting the dataset place a key role in the success of machine learning, when not enough representative data is provided the learning process might easily overfit. Indeed we can define Machine Learning as a data driven approach, where the more quality data is available the better. When the function approximator class used for machine learning is the Neural Network (NN) model, then we can call this approach **Deep Learning**. We will now describe Neural Networks and their main training techniques, based on the book Deep Learning [5].

### 3.1.2 Multilayer Perceptron

The artificial Neural Networks are based on a core unit called **perceptron** or neuron, of which we can see a representation in Fig.3.2. Given a set of inputs features $x_1, x_2, ..x_n \in \mathbb{R}^n$ and a set of weights $\theta_1, \theta_2, ..., \theta_n \in \mathbb{R}^n$, the neuron mathematical function is a weighted sums of the inputs as:

$$net = \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n + b = \sum_{i=0}^{i=n} (\theta_i x_i) + b \tag{3.1}$$

or expressed in a matrix formulation:

$$net = \theta x + b, \tag{3.2}$$

where $\theta$ is the vector containing all the weights of the neuron $[\theta_1, \theta_2, ...\theta_n]$, x is the vector containing all the inputs $[x1, x_2, ...x_n]^T$ and $b \in \mathbb{R}$ is an optional additional

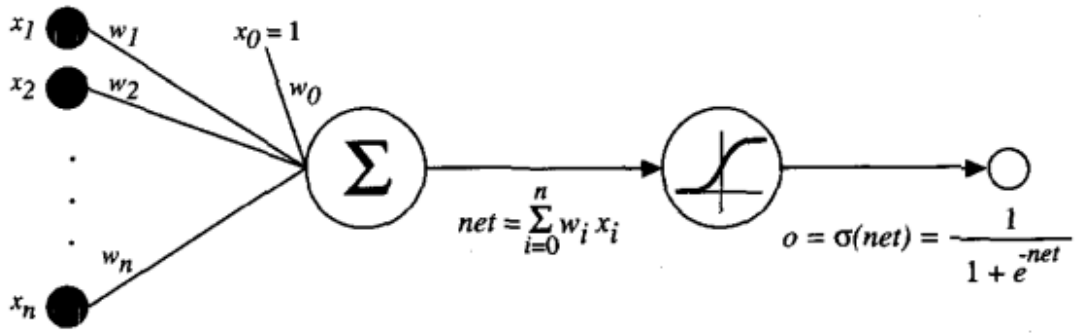



**Figure 3.2:** Perceptron graphical representation, image taken from [6]

bias to augment its representational power. Up to now this is just a simple linear function approximator, then to add non-linearity to the model, the perceptron present an additional cell called **Activation Function**, that is in charge of adding non linearity. In the early perceptrons this was usually a sigmoid function, indicated with $\sigma$, that computes its output $o$ as:

$$o = \sigma(net) = \frac{1}{1 + e^{-net}} \tag{3.3}$$

The sigmoid function maps the input to an output in the range [0,1]. The last development in deep learning has made become very popular a different activation function, the Rectified Linear Unit, referred to as **ReLu**. ReLu computes the output as:

$$ReLu(net) = \max\{0, net\} \tag{3.4}$$

This has many positive attributes, one is the ability to contrast the phenomenon of vanishing gradient that is crucial in the training procedure that we will see in the next section. Just one perceptron is still a very similar approximation to a linear function, that can't represent very complex functions. Then, inspired by biological studies that showed how our brain is composed of neurons connected to each other, the Multilayer Perceptron MLP was proposed. In the MLP we have multiple neurons connected together with a hierarchical structure, each neuron receives as input the output of its predecessor and pass its output to its successors. Usually neurons are grouped in layers to make the visualization easier, layers are just a group of neurons not connected with each other but, in the MLP, connected to the same neurons in input and output. MLPs are the quintessential of Neural Networks and can be thought as a sequential set of **layers**, with the layer in charge of receiving the input called *input layer*, the one in charge of output the prediction *output layer* and all the intermediate layers called *hidden layers*. This hierarchical structure gives us the opportunity to think to layers, as units that start building

simple concepts at the beginning of the network, that end up, being built on top of each other through the network, as a being much more complex. More practically the connections between neurons create a way to put together many different simple non-linear function approximators to reach a high representational power. Another very useful aspect of neural networks lies in the end to end training approach. They are fed with raw data, without the need of hand engineered features and are able to extract automatically from them what they need to perform the task, resulting in a potentially unlimited types of application. We can see a graphical example of Multilayer perceptron in Fig.3.3

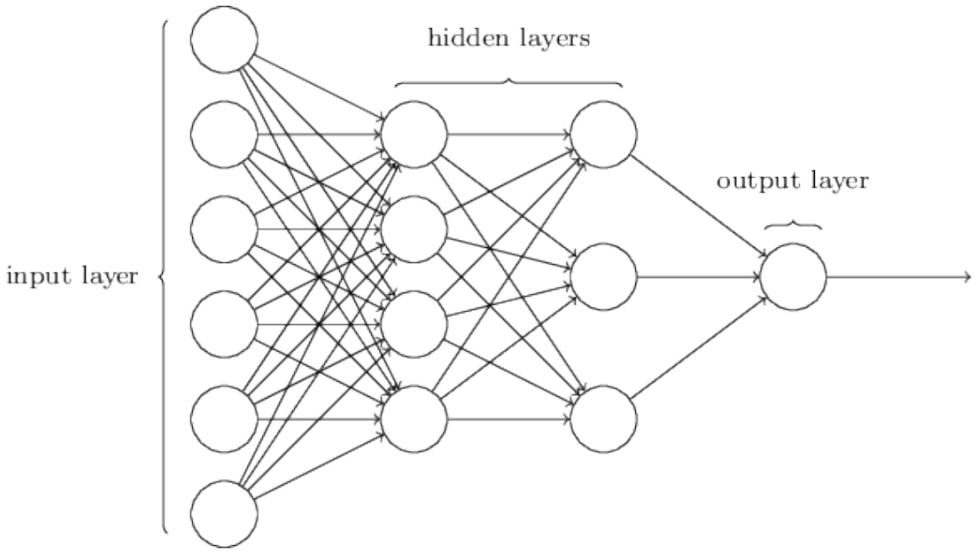



**Figure 3.3:** Example of Neural Network, image taken from [7]

### 3.1.3 Neural Networks Optimization

Neural networks are trained with the **backpropagation** method combined with **stochastic gradient descent**. To address stochastic gradient descent (SGD) we need to first introduce gradient descent. Gradient descent is an optimization technique, that aims to minimize a function $f(x)$ usually called the objective function or, as usually done in deep learning, the loss function by altering $x$. Referring to neural networks we can think of $f(x)$ as the loss of our network, so a measure of how bad the model is performing and $x$ as the set of weights that characterize all the neurons in the net. The derivative $f'(x)$ of $f(x)$ w.r.t. $x$ gives the slope of $f(x)$ at the point $x$. Then the main idea is that if we are able to calculate the gradient of the loss with respect to the parameters we then know the direction towards we need to move them to minimize it. We can then by moving

$x$ of a small step $\epsilon$, usually referred as learning rate, in the opposite direction of the gradient, reduce $f(x)$. Optimization with gradient descent in deep learning is very difficult for two main reason. The first is that usually the function we try to minimize may present lots of local minima or saddle points surrounded by very flat regions where gradient descent becomes useless. The second is the high computational cost of this approach. Usually in deep learning we define the loss function as per-example loss function:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(x_i, y_i, \theta) \tag{3.5}$$

with $L(x, y, \theta)$ being the per data point loss, $m$ being the number of samples $(x, y)$ in our training dataset and $\theta$ the weights of our neural network. For these additive cost function, gradient descent requires computing the gradient for each example with a cost of $O(m)$:

$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta L(x_i, y_i, \theta) \tag{3.6}$$

This as $m$ grows becomes prohibitively long and in deep learning we require to have big dataset in order to reach the desired generalization property. This is where SGD comes in help. Since the gradient itself is an expectation we can estimate it with a set of samples called mini-batch. On each step of the training algorithm we sample uniformly from the dataset a mini-batch of $m'$ data, evaluate the loss function on it and apply a gradient descent step on this estimate:

$$g = \frac{1}{m'}\nabla_\theta \sum_{i=1}^{m'} L(x_i, y_i, \theta) \tag{3.7}$$

SGD has also an additional very good quality, the fact that since its based on estimate it can sometimes avoid to fall in local minima that otherwise, moving towards the true gradient, would not be avoidable. To apply SGD to neural network we exploit the **backpropagation** algorithm. Backpropagation is composed of two phase, a forward pass and a backward pass. In the forward pass a mini-batch is passed through the network to obtain predictions, then these predictions are evaluated with a specific loss function with respect to the true data. Then the loss gradient estimate with respect to the weights of the output layer is computed and is then backpropagated through all the layers weights following the chain rule:

$$\frac{d_y}{d_x} = \frac{d_y}{d_z}\frac{d_z}{d_x} \tag{3.8}$$

Once the gradient with respect to each weight is calculated the gradient descent step is performed:

$$\theta \leftarrow \theta - \epsilon g \tag{3.9}$$

## 3.2 Deep Reinforcement Learning Introduction

In the previous chapter we have just seen RL algorithms applied to tabular cases where we could store Q-values for state,action pairs simply in a lookup table. Now we will extend the methods presented in chapter 2 to be applied with arbitrarily large state and action spaces. In many real world case where we would like to use RL, the state space is enormous. For example assuming to use images as state, the number of possible camera images is much larger than the number of atoms in the universe. It is necessary to say that in such cases we cannot expect to find optimal policy or value function but we will be limited to find good approximate solutions. The other main problem of dealing with large state space, beside the memory needed for large storage tables, is the time and cost to gather enough data to estimate them correctly. As we already seen previously, in RL there is the need of enough exploration and as the state and action space grows this become harder and harder. To address this we need to be able to generalize from observed state and actions to the one that we can think are similar but that we will never see in our experience, this kind of generalization is often called *function approximation*. Luckily this problem has already been extensively studied and many type of function approximation solutions are available, in this work we will focus on the use of Deep learning (DL) and Neural Networks (NNs) as big non linear parametric function approximators. Then RL provides a framework for decision making and NNs should provide their ability to handle unstructured data and to generalize from seen to unseen states or actions. Unfortunately bringing Deep RL brings also some issues that need to be addressed in order to get Neural Networks trained. The main problem is about the training procedure of neural networks combined with the interactive learning paradigm. In supervised learning with NNs we make the assumption of having independent and identically distributed data (i.i.d) on the training set, so that each mini batch sampled from it is statistically representative of the whole dataset. This in Reinforcement Learning is never the case, the data the agent collects is in major part the consequence of its own action, its own belief change the data that will see in the future. Furthermore after each step of update of the Value or Policy function, in theory the agent could change drastically its behaviour, causing a significative shift of the distribution of states that are explored. In deep learning instead, we make the assumption that the underlying data distribution we are trying to learn is always the same. We will now present some of the approaches that try to overcome this issues and try to put together RL and DL.

## 3.3 DQN and DDQN Algorithms

In section 2.3.2 we described the Q-learning algorithm in tabular case, we will now describe two derived algorithms called DQN [8] and DDQN [9], that introduce a neural network, called Q-network $Q_\theta$, with $\theta$ being its set of weights,as the Q-value function approximator. The Q-network receives states $s \in S$ as input and output values for all possible actions $a \in A$. This choice limits the use of this algorithms without additional modification to discrete-action settings where $A$ is a finite space. These two algorithms reached human-level performance over the majority of games in the Atari 2600 framework for RL algorithms evaluation. For further details and information refer to the original papers.

### 3.3.1 DQN

As stated before, the are two main cause for training instabilities of RL with non linear function approximator, the correlation in the sequence of states the agent visit and the fact that small updates to Q may significantly change the policy and therefore change the data distribution and the correlation between the Q-values and target Q-values. To address the first issue, an **Experience Replay Buffer** is introduced. This works like a big memory where at each time step $t$ a transition $\tau = (s, a, s', r)$ is stored, then when a gradient step has to be executed to update the Q-network beliefs, a random mini-batch of transitions are sampled from it, with the aim to decorrelate the selected transitions . The second issue is addressed by introducing a target Q-network $\hat{Q}_{\theta'}$, that is basically a twin Q-network that is only periodically updated during the training process. This should reduce the correlation that was present between the target Q-values and the Q-values that were updated:

$$Q(s, a) \leftarrow r + \max_{a' \in A} \hat{Q}_{\theta'}(s', a') \tag{3.10}$$

### 3.3.2 DDQN

One issue that wasn't still addressed with DQN is the one of overestimation of the Q-values. In some of the Atari games DQN suffers from substantial overestimation. In the DDQN paper [9] the authors demonstrate how this issue is due to the still present correlation between the target value and the Q-value in its update.

$$y = r + \gamma \max_{a' \in A} \hat{Q}_{\theta'}\Big(s', a'\Big) \tag{3.11}$$

As we can see in this equation the maximization over the actions to select the best one is done on the same target Q-network that is also used to estimate its value. This means that we are sampling the same error twice. To try to decouple this

**Algorithm 3** DQN algorithm

Initialize replay memory buffer $D$ to capacity $N$
Initialize Q-network $Q$ with random weights $\theta$
Initialize target Q-network $\hat{Q}$ with random weights $\theta' = \theta$
**for** episode=1 to M **do**
  Initialise environment $E$ and get initial state $s$
  **for** t=1 to T **do**
    With probability $\epsilon$ select a random action $a_t \in A$
    Otherwise select $a_t = \max_{a \in A} Q_\theta(s_t, a)$
    Execute action $a_t$ in environment and observe reward $R_t$ and state $S_{t+1}$
    Store transition $\tau = (s_t, a_t, s_{t+1}, r_{t+1})$ in $D$
    Sample random mini-batch of transitions $(s_j, a_j, s_{j+1}, r_j)s$ from $D$
    **if** episode terminate in $s_{j+1}$ **then**
      Set $y_j = r_j$
    **else**
      Set $y_j = r_j + \gamma \max_{a_{j+1} \in A} \hat{Q}_{\theta'}\Big(s_{j+1}, a_{j+1})\Big)$
    **end if**
    Perform a gradient descent step on $(y_j - Q(s_j, a_j))^2$ w.r.t parameters $\theta$
    Every $C$ steps update target networks weight $\theta^- \leftarrow \theta$
  **end for**
**end for**

**Algorithm 4** DDQN algorithm

Initialize replay memory buffer $D$ to capacity $N$
Initialize Q-network $Q$ with random weights $\theta$
Initialize target Q-network $\hat{Q}$ with random weights $\theta' = \theta$
**for** episode=1 to M **do**
  Initialise environment $E$ and get initial state $s$
  **for** t=1 to T **do**
    With probability $\epsilon$ select a random action $a_t \in A$
    Otherwise select $a_t = \max_{a \in A} Q_\theta(s_t, a)$
    Execute action $a_t$ in environment and observe reward $r_t$ and state $s_{t+1}$
    Store transition $\tau = (s_t, a_t, s_{t+1}, r_{t+1})$ in $D$
    Sample random mini-batch of transitions $(s_j, a_j, s_{j+1}, r_j)$ from $D$
    **if** episode terminate in $s_{j+1}$ **then**
      Set $y_j = r_j$
    **else**
      Set $y_j = r_j + \gamma \hat{Q}_{\theta'}\Big(s_j, \arg\max_{a_{j+1} \in A} Q_\theta(s_{j+1}, a_{j+1})\Big)$
    **end if**
    Perform a gradient descent step on $(y_j - Q(s_j, a_j))^2$ w.r.t parameters $\theta$
    Every $C$ steps update target networks weight $\theta^- \leftarrow \theta$
  **end for**
**end for**

noise, the solution proposed is to, without introducing a new network, use again the target network but this time selecting the action with the standard Q-network:

$$y = r + \gamma \hat{Q}_{\theta'}\Big(s, \arg\max_{a' \in A} Q_\theta(s', a')\Big) \tag{3.12}$$

## 3.4 TD3 Algorithm

Now we will present an actor-critic method named TD3 [10], that builds on top of DDQN but add the use of an explicit policy network. Actor-critic algorithms employ both a parameterized policy and a parameterized value function, and use the value function as a better and more stable estimate of the expected reward for policy gradient calculation. This is a key addition since it also allows this algorithm to be used for agents with continuous-action spaces, removing the direct maximization over Q-values that was only feasible with a finite action space. TD3 updates its policy by the deterministic policy gradient [11] and exploits a very similar Q-value function update to the one of DDQN. The one of TD3 is called **Clipped Double**

**Q-Learning**. DDQN update exploits an additional target Q-network $Q(\hat{s}, a)$, to disentangle the target from the estimates to be updated. In TD3, since we have an explicit policy that is found to be slow-changing, using just the target Q-network is not enough since it ends up being still too similar to the current Q-network. To address this two new Q-network are introduced, with one that still plays the role of target network and the other the role of the current network. Then the update for both $Q_1$ and $Q_2$ current networks will be formalized like this:

$$y_1 = r + \gamma \min_{i=1,2} Q_{\theta_i'}(s', \pi_\phi(s')) \tag{3.13}$$

where we use the policy to estimate the action but we select the more pessimistic estimate between the two target Q-values. This target will be the same for both Q-networks. Furthermore the authors expressed the need to reduce even more the variance in the value estimates, since they are then used to calculate the gradient for the policy update and thus can cause reduced performance and slow learning speed. Then TD3 introduces the use of a target policy, and the delayed policy updates paired with the slow update of all the target networks. This is done to make the Q-values estimate as stable as possible when used by the policy update. In practice what the algorithm does is to update the policy every $d$ steps of Critic Evaluation, where $d$ is an hyperparameter of the algorithm, that we will call **p-freq** and at the same time update of a small quantity $\tau$ the parameters of the target networks:

$$\begin{aligned} \theta' &\leftarrow \tau\theta + (1-\tau)\theta \\ \phi' &\leftarrow \tau\phi + (1-\tau)\phi \end{aligned} \tag{3.14}$$

The last issue addressed by TD3 is the target policy smoothing regularization. Since we have a deterministic policy that can overfit to narrow peaks in the value estimate and since we can reasonably assume that similar actions should have similar values, TD3 in addition to what already the function approximator does automatically, explicitly put a regularization term in the Q-value update:

$$\begin{aligned} y &= r + \gamma Q_{\theta'}(s', \pi_{\phi'}(s') + \epsilon) \\ \epsilon &\sim \text{clip}\mathcal{N}(0, \sigma), -c, c) \end{aligned} \tag{3.15}$$

This is basically a little random noise added to the target policy in the Q-values target calculation that should smooth the Q-values estimate over a small area around the target action.

**Algorithm 5** TD3 algorithm.

Initialize critic Q-networks $Q_{\theta_1}, Q_{\theta_2}$ with random parameters $\theta_1, \theta_2$
Initialize actor network $\pi_\phi$ with random parameters $\phi$
Initialize target networks $\theta_1' \leftarrow \theta_1, \theta_2' \leftarrow \theta_2, \phi' \leftarrow \phi$
Initialize replay memory buffer $B$
**for** $t$=1 to $T$ **do**
  Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon$,
  $\epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward $r$ and new state $s'$
  Store transition tuple $(s, a, s', r)$ in $B$
  Sample random mini-batch of $N$ transitions $(s, a, s', r)s$ from $B$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon$, $\epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min_{i \in \{1,2\}} Q_{\theta_i'}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \arg\min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  **if** $t$ mod $d$ **then**
    Update $\phi$ by deterministic policy gradient:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
    Update target networks:
    $\theta_i' \leftarrow \tau\theta_i + (1 - \tau)\theta_i'$
    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
  **end if**
**end for**

# Chapter 4

# Offline Reinforcement Learning

In the last years we have seen the success of deep learning methods on a various range of tasks and practical problems, where in general the more data was available the better the algorithms ended up being. In the previous chapter we have also seen how Reinforcement Learning and Deep Learning together, try to solve the sequential decision making problem and often obtain very good results in many different settings. However the online trial and error approach that characterize Reinforcement Learning is also the major obstacle to its widespread diffusion. In many settings the interactions with the environment are costly or dangerous. For example in autonomous driving or healthcare we can't simply let our agent to try potentially unsafe actions for possible future reward. To address this issue we will now describe a new paradigm for RL, called **Offline-Reinforcement Learning** (Offline-RL) where we aim to train our agents completely from already collected data, without the need of online interaction, resulting in a data-driven reinforcement learning approach. We can see a graphical example of this paradigm in Fig.4.1 In

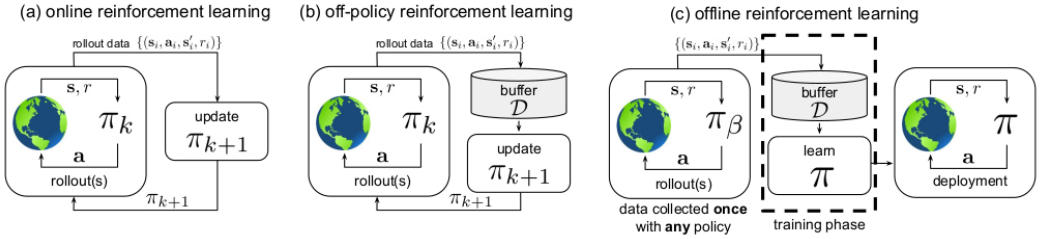



**Figure 4.1:** Graphical representation of classic online RL, classic off-policy RL and Offline-RL, image taken from [12]

this chapter we will at first explore one of the simplest data-driven formulation to solve the sequential decision making problem, called Behavioral Cloning, then we will formally introduce the Offline Reinforcement Learning approach, its positive aspects and its main issues, following the paper [12]. Finally we will present a state of the art actor-critic Offline-RL algorithm, TD3-BC and the proposed variation for discrete-action control TD4-BC.

## 4.1 Behavioral Cloning

The first and more naive solution that comes to mind if we are to put together data driven approach and Reinforcement Learning is the one defined as **Behavioral Cloning** (BC). As behavioral cloning we mean to cast a RL problem as a simple Supervised learning problem. Given a data-set of state-action tuples $D := \{(s, a)\}$ we want to derive a policy $\pi_\phi$, in our case a Neural Network parameterized by $\phi$, that tries to imitate the behavioral policy $\pi_\beta$ that collected the data. We can then write the optimization objective as:

$$\phi^* = \arg\max_\phi \prod_{i=0}^{N} \pi_\phi(a_i|s_i) \tag{4.1}$$

We can imagine to apply this kind of solution when we have already data of a good performing policy or for example data collected by humans. Then the derived policy, trying to imitate a near optimal policy, should work well when deployed in the real environment. However this is also the first limitation of this approach, BC can't improve w.r.t to the behavioral policy, thus resulting in a high limitation of application when no good data is available. Furthermore such a naive approach has another major issue, the distributional shift from the training data when the model is deployed. Learning algorithms make mistakes, even if they are very well trained they usually make some errors when deployed in real world, because real data brings inherently noise and uncertainty in it. What happens is that after some interactions in the real environment our learned policy is queried with states that, even if slightly different from the one seen during training, bring our policy to make error. This in theory should be addressed by the generalization that we aim to obtain with neural networks, but in practice a degradation in performance is often observed. What actually happens is that after the policy is queried with a slightly different state than the one on which was trained, it makes a little mistake and it might ends up in a even more different state, and then might do an even bigger error. This summation of errors starts a divergence process that ends up with the agent following trajectories very different from the ones seen in training, thus resulting in poor quality behavior. Then Behavioral Cloning suffers also from the loss of potential information that not exploiting a Reward Function like RL

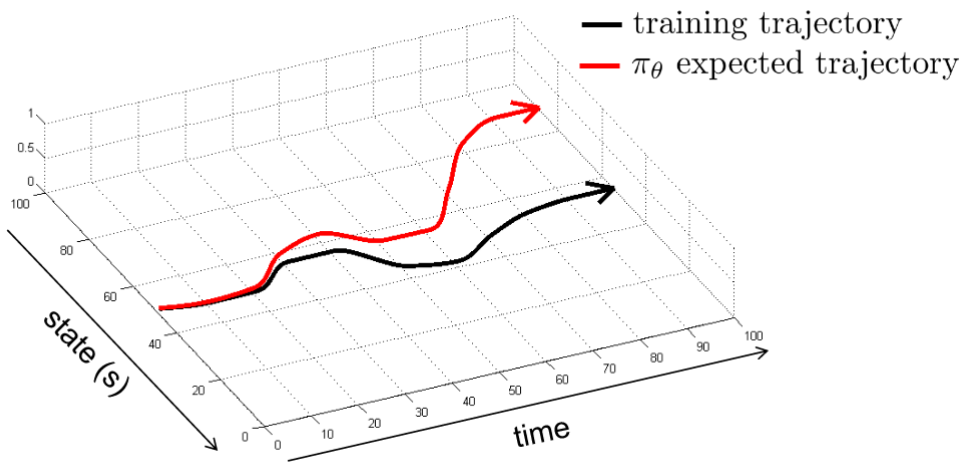



**Figure 4.2:** Distributional shift graphical example, image taken [13]

cause. This limits a lot what the agent can learns from the data both in terms of the environment dynamics and optimal actions selection.

## 4.2 Offline Reinforcement Learning

As previously stated, Offline Reinforcement Learning problem can be defined as a data-driven formulation of the reinforcement learning problem. The end goal is still to optimize the expected cumulative discounted reward like written in Eq.(2.5), but now the algorithm is provided with a fixed dataset of transitions $D = (s_t^i, a_t^i, s_{t+1}^i, r_t^i)$ and its objective is to find the best policy it can, only using this data. We can then intuitively think that Offline Reinforcement Learning requires the learning algorithms to derive a sufficient understanding of the dynamics of the underlying MDP from a fixed dataset $D$, and to then extract the best possible policy $\hat{\pi}_\phi^*$. The first issue with Offline Reinforcement Learning is that, since the algorithm must rely only on a static dataset $D$, there is no possibility of augmenting the exploration, that as underlined in section 2.4 is a key assumption for RL algorithms to work. Indeed if the regions of the state-action space that yield high reward are completely unexplored in the dataset, then our algorithms would probably never discover them. This is a underlying defect of an offline approach and we can't address this in anyway, then when presenting Offline-RL algorithms we will assume that $D$ covers adequately the state-action space $SxA$. We will later see in chapter 6 how this assumption is not always realistic when we tries to use this approach

to tackle real problems where it's even not easy to have already collected data. In theory any of the off-policy algorithm that we presented before, like DDQN in section [9], should work as an offline RL algorithm, simply by fixing a priori the Replay Buffer with the dataset of transitions $D$ and removing the collection of new online transitions. In practice in many settings these algorithms do not work very well. The reason for this is to be searched in the distribution shift, similarly as what we have seen before happening in BC. Here the need for the learned policy to differ from the behavioral policy that collected the data is even more exacerbated since in Offline-RL we want our learned policy to outperform the collection policy. Even more the distributional shift in Offline-RL, does not just lie in the unseen states the learned policy visits at deployment time, but also in the maximization of the expected return in the training process, that is a key step of many RL algorithms. If we look at the Bellman Expectation Equation for Q-values, Eq.(2.8), the target values requires evaluating $Q_\pi(s_{t+1}, a_{t+1})$ where $a_{t+1} \sim \pi(s_{t+1})$. So the accuracy of the target Q-values depends on the estimate of the Q-values for actions that potentially are out of the distribution w.r.t to the actions that the Q-value was ever trained on. The issue of out of distributions (OOD) actions is aggravated by the fact that the policy $\pi(s, a)$ is explicitly optimized to maximize the expected Q-value for state-action pairs:

$$\mathbb{E}_{a\sim\pi(s,a)}[Q_\pi(s, a)] \tag{4.2}$$

This is true even when no explicit policy $\pi(s, a)$ is defined and the maximization of Q-values is done greedily like in Q-learning:

$$Q_\pi^{target}(s, a) = r + \arg\max_{a'\in A} Q_\pi(s', a') \tag{4.3}$$

This means that if the model thinks erroneously that an action is very good and has consequently an high Q-value, it maximizes its policy toward it, but without the online interaction,can never correct its wrong assumptions, thus maximizing the error. This results in an accumulation of error at each training iteration and degradation of the performance. In the paper [14] the authors tested with SAC [15], a state of the art off-policy actor-critic Deep RL algorithm, the existence of this unlearning effect. We can see the results in Fig.4.3. The learning curve have a behaviour that might at a first sight resemble the classical overfitting that we can usually encounter in Supervised Learning, as seen in section 3.1. However it can also be seen how this effect is invariant to the number of data points used to train the algorithms, suggesting that it's actually a different phenomenon. To sum up what we have seen, to effectively deploy and train offline reinforcement learning algorithms we first need to address this OOD actions overestimation issue. We will now see one approach to this problem called TD3-BC.

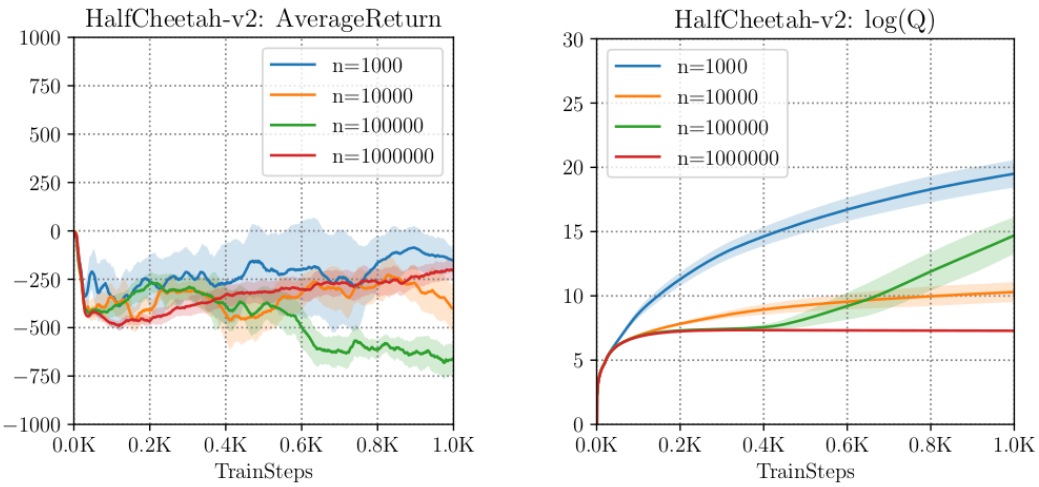



**Figure 4.3:** SAC performance on HalfCheeta-v2 in offline setting, showing return as a function of gradient steps (left) and average learned Q-values on a log scale (right), for different numbers of training points($n$). Image taken from [12]

## 4.3 Minimalist approach to Offline RL: TD3-BC

TD3-BC [16] is an offline reinforcement learning algorithm for continuous control, derived from the TD3 algorithm, seen in section 3.4, that tries to address the extrapolation error on OOD actions. Many of the proposed algorithm to address this issue are usually very complex and even when considered simple, they usually make a significant amount adjustments to the underlying implementation. Those additions make difficult to disentangle the cause of possible performance gain between the algorithmic and implementation changes. TD3-BC, instead, tries to adapt one deep reinforcement learning algorithm with minimal algorithmic changes and reaches state of the art performance on the D4RL evaluation framework [17]. The algorithm simple make one addition to the policy update step of TD3:

$$\pi = \arg\max_{\pi} \mathbb{E}_{(s)\sim D}[Q(s, \pi(s))] \tag{4.4}$$

The addition is a behavioral cloning term with the objective to regularize the policy:

$$\pi = \arg\max_{\pi} \mathbb{E}_{(s,a)\sim D}\Big[\lambda Q(s, \pi(s)) - (\pi(s) - a)^2\Big] \tag{4.5}$$

This BC-term has the goal of pushing the policy toward favoring actions that are close to the ones seen in the dataset $D$. This approach of policy regularization deeply relies on the optimistic idea that the neural networks should be able to generalize so that they still end up in discovering policy that are better than the behavioral

one, even if regularized. To balance between the RL component, expressed as the maximization of the Q-values and the imitation component, expressed through the BC term, a single hyperparameter $\lambda$ is introduced. Actually since this balance is highly susceptible to the scale of the Q-values $\lambda$ is defined by adding a normalization term based on the absolute Q-values:

$$\lambda = \frac{\alpha}{\frac{1}{N}\sum_{(s_i,a_i)} |Q(s_i, a_i)|} \tag{4.6}$$

In practice this term is usually estimated over mini-batches that are sampled during the training process rather than on the entire dataset, to limit the additional computational cost. The fact that this approach add just one single hyperparameter should not be underestimated. Indeed in the offline setting the tuning of hyperparameters, that in other settings would be trivial, becomes an issue, since any real-world interaction is costly or dangerous. Is then clear that just having this one $\lambda$ hyperparameter is one of the positive aspects of this minimalist approach. The second and last algorithmic change that TD3-BC makes is a simple state features normalization over the entire dataset:

$$s_i = \frac{s_i - u_i}{\sigma_i + \epsilon} \tag{4.7}$$

with $u_i$ dataset mean, $\sigma_i$ standard deviation mean and $\epsilon$ being a small normalization constant($10^{-3}$). This normalization is a standard procedure in many Deep RL algorithms, that the authors found to have a good effect on the performance in many tasks. Furthermore is an especially very well suited procedure for Offline-RL where the dataset remains fixed. One last positive aspect of this algorithm with respect to other state of the art offline RL approaches is that the minimal implementation change adds basically no computational overhead on the training process, resulting as we can see in Fig.4.4 in much faster training with respect to CQL[18] and Fisher-BRC[19], two other state of the art Offline-RL algorithms. This is a key quality for a data-driven algorithm, since it allows to train on big and high representative datasets more easily. The authors of this method in the main paper [16], expose also the issue of the **instability of the trained policy**. This means that analyzing the final trained policies of offline algorithms, their average performance can be reasonable but has usually very high variance. This reflects on some evaluation episodes with very poor performance. In online-rl this can be easily addressed by keeping training or revert to a previous version of the policy that during training was more stable, but this is not possible in this setting. This trait of Offline-RL algorithms is found to be consistent across all Offline-RL algorithms that they evaluated. Furthermore the fact that even a very simple approach as TD3-BC presents this issue, fall in favour to the hypothesis of this being a inherently shared offline reinforcement learning issue.

**Algorithm 6** TD3-BC algorithm. Optimal values from the authors are $\alpha = 2.5$, $\epsilon = 10^{-3}$

Initialize critic Q-networks $Q_{\theta_1}, Q_{\theta_2}$ with random parameters $\theta_1, \theta_2$
Initialize actor network $\pi_\phi$ with random parameters $\phi$
Initialize target networks $\theta_1' \leftarrow \theta_1, \theta_2' \leftarrow \theta_2, \phi' \leftarrow \phi$
Store dataset $D$ in replay buffer $B$
Normalize states over the dataset $D$: $s_i = \frac{s_i - u_i}{\sigma_i + \epsilon}$
**for** $t$=1 to $T$ **do**
  Sample random mini-batch of $N$ transitions $(s, a, s', r)s$ from $B$
  $\tilde{a} \leftarrow \pi_{\phi'}(s') + \epsilon$, $\epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
  $y \leftarrow r + \gamma \min_{i \in \{1,2\}} Q_{\theta_i'}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \arg\min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  **if** $t$ mod $d$ **then**
    Update $\phi$ by deterministic policy gradient:
    Calculate hyperparameter $\lambda$: $\lambda = \frac{\alpha}{\frac{1}{N} \sum_{(s_i, a_i)} |Q(s_i, a_i)|}$
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_\phi \pi_\phi(s) \Big[ \lambda Q_{\theta_1}(s, \pi(s)) - (\pi(s) - a)^2 \Big]$
    Update target networks:
    $\theta_i' \leftarrow \tau \theta_i + (1 - \tau) \theta_i'$
    $\phi' \leftarrow \tau \phi + (1 - \tau) \phi'$
  **end if**
**end for**

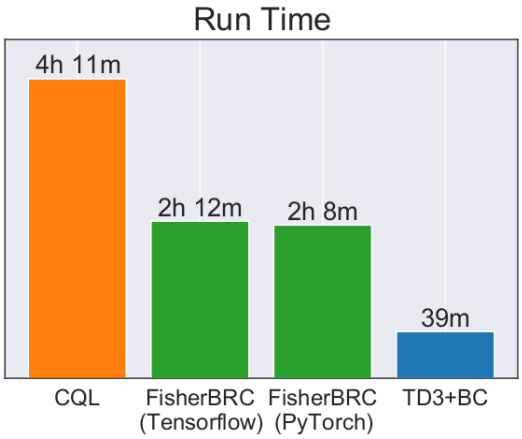



**Figure 4.4:** Total training time comparison of training each Offline-RL algorithm. Image taken from [16]

## 4.4 Proposed discrete-action control algorithm: TD4-BC

We will now present the algorithm that we propose in this thesis: TD4-BC(Discrete TD3-BC). This is an adaptation for the discrete-action control setting of the TD3-BC algorithm. As seen in the previous section [16] TD3-BC reaches state of the art performance with one single modification to a Deep RL algorithm. We then can ask if we can translate the minimal approach of this algorithm to the discrete-action setting, in order to have a valid offline reinforcement learning algorithm for any real case scenario. The first issue to address is the Behavioral Cloning term calculation. In TD3-BC this is obtained as the squared difference between dataset action and policy action $(\pi(s) - a)^2$. Here we have a fixed number of discrete possible actions and we define our policy as a neural network that outputs unnormalized scores for each of the possible actions. Then inspired by supervised learning, we think at actions seen in the dataset $D$ as true labels and we use one of the most common classification loss, the Cross Entropy Loss, to penalize policy for assigning low probabilities to actions seen in $D$.

$$BC = -log\frac{exp(x_n)}{\sum_{c=1}^{C} exp(x_n)} \times t_n \tag{4.8}$$

where $x_n$ are the unnormalized scores that our policy output for state $s_n$, $t_n$ the one-hot true action vector for $a_n$, given $(s_n, a_n) \sim D$, and $C$ the number of available actions. Then we need also to update the policy update step with respect to the Q-values maximization part. To do this we decided to maximize the expected

Q-values under the policy's categorical probability distribution, resulting in :

$$\pi = \arg\max_{\pi} \mathbb{E}_{(s,a)\sim D}\Big[\lambda\pi(s)Q(s,*)\Big] \tag{4.9}$$

Putting the two together we obtain:

$$\pi = \arg\max_{\pi} \mathbb{E}_{(s,a)\sim D}\Big[\lambda\pi(s)Q(s,*) - BC\Big] \tag{4.10}$$

This update follow a similar approach to what can be found in [20]. As it can be seen, we still normalize the Q-values in the policy update step through an additional normalization factor inserted in $\lambda$. This is now composed by dividing expected Q-values as the mean absolute expected Q-value.

$$\lambda = \frac{\alpha}{\frac{1}{N}\sum_{(s_i)} |\pi(s_i)Q(s_i,*)|} \tag{4.11}$$

There are other minor discrete algorithmic changes that differs from the continuous one, for example we are not adding anymore clipped noise to the action selection for the Q-values update but we try to still incorporate some randomness by sampling the action according to the normalized probabilities from the policy:

$$\mathbb{P}(a_i) = \frac{exp(x_i)}{\sum_{j=1}^{C} exp(x_j)} \tag{4.12}$$

Furthermore in TD4-BC we still normalize the states features over the entire dataset:

$$s_i = \frac{s_i - u_i}{\sigma_i + \epsilon} \tag{4.13}$$

**Algorithm 7** TD4-BC algorithm.

Initialize critic Q-networks $Q_{\theta_1}, Q_{\theta_2}$ with random parameters $\theta_1, \theta_2$
Initialize actor network $\pi_\phi$ with random parameters $\phi$
Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$
Store dataset $D$ in replay buffer $B$
Normalize states over the dataset $D$: $s_i = \frac{s_i - u_i}{\sigma_i + \epsilon}$.
**for** $t$=1 to $T$ **do**
  Sample random mini-batch of $N$ transitions $(s, a, s', r)s$ from $B$
  Sample one action from the learned policy categorical distribution:
  $\tilde{a} \leftarrow \pi_{\phi'}(s')$
  $y \leftarrow r + \gamma \min_{i \in \{1,2\}} Q_{\theta'_i}(s', \tilde{a})$
  Update critics $\theta_i \leftarrow \arg\min_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
  **if** $t$ mod $d$ **then**
    Calculate discrete BC term:
    BC=CrossEntropyLoss$(a, \pi(s))$
    Update $\phi$ by regularized categorical policy update:
    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_\phi \pi_\phi(s) \Big[ \lambda \pi_\phi(s) Q_{\theta_1}(s, *) - BC \Big]$
    Update target networks:
    $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
    $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
  **end if**
**end for**

### 4.4.1 Online Fine-Tuning

Offline-RL can train agents that outperform the behavior policy that collected the offline dataset. However as stated in [21]: *"thusly trained agents may in the end be suboptimal in the environment where they are deployed for (a) the dataset they were trained on may be suboptimal; and (b) environment in which they are deployed may be different from the environment in which the dataset was generated"*. (a) can happen when the dataset doesn't cover enough high reward regions of the state action space or even more when it has not enough action or state space coverage. (b) can be interpreted as the general shift between training and testing distributions, usually found in machine learning. This is not just limited to the deployment on a completely different environment but can also manifest when the training-data environment presents extemporaneous anomalous events or it continuously mutates over time. We will explore these issues later in chapter 5 when we will present the HVAC related datasets and their limitations. We will now show our two minimal approaches to make our algorithm TD4-BC able to be fine-tuned through online interaction. The first approach, that we will call Naive Fine-Tuning (TD4-BC-FT), consists of adopting an $\epsilon$-greedy strategy during action selection, with $\epsilon$ decayed over time; the online transition sampled in this way are added to the same memory buffer containing offline data; and at each new trajectory sampled a training update step is carried out by the agent. In the policy update of TD4-BC-FT the BC term and the hyperparameter $\alpha$ are completely removed resulting in this formulation:

$$\pi = \arg\max_{\pi} \mathbb{E}_{(s,a)\sim D}\Big[\lambda \pi(s) Q(s, *)\Big] \tag{4.14}$$

where $\lambda$ is:

$$\lambda = \frac{1}{\frac{1}{N}\sum_{(s_i)} |\pi(s_i) Q(s_i, *)|} \tag{4.15}$$

The second approach, that we will call Conservative Fine-Tuning (TD4-BC-FTC), still consists of adopting an $eps - greedy$ strategy during action selection, with $\epsilon$ decayed over time and to add the online transitions to the memory buffer. In this approach however we keep the regularized policy update in the training exactly as it is in TD4-BC. This is done with two objectives in mind. The first is to still keep the regularization on OOD actions for the offline data. The second is that, since the online data could be seen as OOD from the agent point of view, the policy regularization of TD4-BC could be useful to limit the Q-values overestimation also on this kind data. We expect this approach to provide a slower fine-tuning with respect to the Naive one, but with less chances of diverging estimates.

# Chapter 5

# Testing Environments

In this chapter we introduce the two environments on which we tested our approach to discrete-action control. We describe their main characteristics and objectives, explaining also how the datasets used for training our offline algorithm were obtained.

## 5.1 Lunar Lander

LunarLander-v2 is an open source benchmark environment for optimal control [22]. For each episode a lander is positioned in a random position on the x-axis and at the top of y-axis, then the objective is to pilot it on a pad positioned always at coordinates (0,0), without crashing it. We selected this environment, between many other standard environment for RL algorithms evaluation, because of its duality.

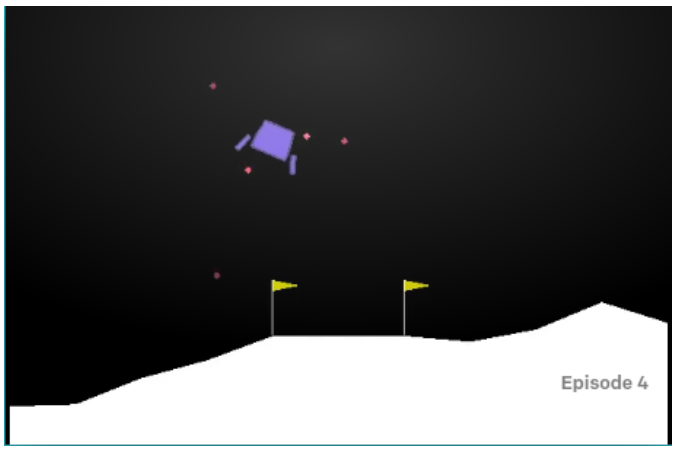



**Figure 5.1:** LunarLander-v2 frame example. Image taken from [22]

It presents indeed both a discrete-action control and continuous-action control version. This was extremely useful for our work since it allowed us to test the TD3-BC algorithm on its standard continuous domain and then to test our proposed discrete variation TD4-BC on the discrete domain.

### 5.1.1 Observation Space and Reward Function

The observations in LunarLander-v2 environment are composed by the eight features detailed in table 5.1.

| Feature Name | Feature meaning | Feature Domain |
|---|---|---|
| $x$ | X-axis coordinate | (-1,1) |
| $y$ | Y-axis coordinate | [0,1] |
| $v_x$ | Horizontal speed | $\mathbb{R}$ |
| $v_y$ | Vertical speed | $\mathbb{R}$ |
| $\theta$ | Angle | [-180,180] |
| $v_\theta$ | Angular speed | $\mathbb{R}$ |
| $c_1$ | First leg contact to the ground | {0,1} |
| $c_2$ | Second leg contact to the ground | {0,1} |

**Table 5.1:** LunarLander-v2 Observation space Description

The reward functions is defined as

$$R_t = \begin{cases} (r_t^{xy} - r_{t-1}^{xy}) + (r_t^v - r_{t-1}^v) + (r_t^\theta - r_{t-1}^\theta) + r_t^p & \text{if environment not done} \\ (+100 \times l - 100 \times c) & \text{if environment done} \end{cases} \tag{5.1}$$

where the four main components are

$$r_t^\theta = -100 \times \mid {\theta_t}^2 \mid +10 * c_1 + 10 * c_2 s, \tag{5.2}$$

$$r_t^{xy} = -100 \times \sqrt{({x_t}^2 + {y_t}^2)}, \tag{5.3}$$

$$r_t^v = -100 \times \sqrt{(v_x t^2 + v_y t^2)}, \tag{5.4}$$

$$r_t^p = -0.3 \times e_t^m - 0.03 \times e_t^s, \tag{5.5}$$

where $e_t^m$ and $e_t^s$ are two boolean variables related to the the main engine and side engine respectively, that take value equal to 1 if the engine is fired at time-step t. $l$ is another boolean variable indicating if the ship has correctly landed on the pad and $c$ a boolean that indicates instead if the ship has crashed. The first component $r_t^\theta$ is encouraging the ship to stay as vertical as possible during its landing process,

the second $r_t^{xy}$ is penalizing any types of action that moves the lander away from the landing pad. Then the third component ensures that the agent tries to reach the coordinates (0,0) with null velocity in order to avoid damages and the forth component is a sort of fuel cost, basically a penalty for any main engine firing or lateral engine firing. This reward design choice is also crucial to don't let the agent to just learn how to fly for an infinite amount of time. Lastly it's important to note that when the episode is ended the reward can only takes two mutually exclusive values, a bonus of +100 points, if the lander has correctly landed, and a malus of -100 points, if it has crashed with the ground. On average the reward for moving from the top of the frame to the landing pad and zero speed is from 100 to 140 points. The environment is considered to be solved if the agent is able to reach an average total cumulative reward over the last 100 episodes superior or equal to 200 points.

### 5.1.2 Action Spaces

As showed in table 5.2, in the continuous environment, the lander is controlled by a bi-dimensional vector. Its first dimension controls the firing of the main engine and how much throttle to use. Its second dimension controls the lateral engines, selecting which one to activate.

| Action Dimension | Value Ranges | Meaning |
|---|---|---|
| 1 | [-1,0) | Main engine off |
| | [0,1] | Fire Main engine with throttle varying from 50% to 100% |
| 2 | [-1,-0.5) | Fire right engine |
| | (0.5,1] | Fire left engine |
| | [-0.5,0.5] | Both lateral engines off |

**Table 5.2:** LunarLander-v2 continuous action space description

Instead in the discrete-action environment, as we can see in table 5.3, the lander can be controlled with four different discrete actions: do nothing, fire left orientation engine, fire right orientation engine, fire main engine. According to Pontryagin's maximum principle, this formulation where you can just decide fire engines with full throttle, still allows the agent to reach an optimal performance.

| Action Encoding | Action meaning |
|---|---|
| 0 | Do nothing |
| 1 | Fire left engine |
| 2 | Fire right engine |
| 3 | Fire main engine |

**Table 5.3:** LunarLander-v2 discrete action space description

### 5.1.3 Dataset Collection and Generation

Following the approach suggested by the authors of D4RL dataset [17], a standard dataset for Offline-RL algorithms evaluation, we collect three different datasets of different qualities: random, medium, and expert. Additionally we can also introduce reference values for each dataset to normalize scores roughly to the range between 0 and 100, by computing the D4RL score:

$$D4RL\ score = 100 \times \frac{score - random\ score}{expert\ score - random\ score} \tag{5.6}$$

In table 5.4 are shown, for each one of the dataset derived for the continuous-action environment, its quality, D4RL score and a brief description on how it was obtained. Table 5.5 does the same but for the datasets derived for the discrete-action environment.

| **Dataset Quality** | **Dataset Code** | **D4RL score** | **Collection Policy** |
|---|---|---|---|
| Random | R | 0 | Let an agent behave randomly and collect $10^6$ transitions |
| Medium | M | 70.83 | Train a TD3 agent until it reaches an average reward over last 10 episodes major or equal than 100 points ("medium" performance), then stop its training and collect $10^6$ transitions |
| Expert | E | 100 | Train a TD3 agent until it reaches an average reward over last 10 episodes major or equal than 200 ("expert" performance), then stop its training and collect $10^6$ transitions |

**Table 5.4:** LunarLanderContinuous-v2 Datasets composition

| **Dataset Quality** | **Dataset Code** | **D4RL score** | **Collection Policy** |
|---|---|---|---|
| Random | R | 0 | Let an agent behave randomly and collect $10^6$ transitions |
| Medium | M | 72.20 | Train a DDQN agent until it reaches an average reward over last 10 episodes major or equal than 100 points ("medium" performance), then stop its training and collect $10^6$ transitions |
| Expert | E | 100 | Train a DDQN agent until it reaches an average reward over last 10 episodes major or equal than 200 ("expert" performance), then stop its training and collect $10^6$ transitions |

**Table 5.5:** LunarLander-v2 Datasets composition

## 5.2 HVAC Control Retrofitting

Heating, Ventilation, Air Conditioning, HVAC, systems are in charge of maintaining thermal comfort and good air quality in enclosed spaces. In this thesis we start from the work *Reinforcement Learning Control Algorithm for HVAC Retrofitting: Application to a Supermarket Building Model by Dynamic Simulation* [23]. The main goal of this research was to test, through a dynamic simulation of an HVAC system deployed in a supermarket, if an online RL approach could bring a reduction in its energy consumption, while satisfying some comfort constraints. Our objective is to proceed further and test if with an Offline-RL approach we can match or even improve the performances of the online RL, avoiding its costly and potentially unfeasible online training phase, by exploiting already available data. We will now discuss some further details regarding this simulation environment, for further information refer to [23]. The framework used for this thesis, can support up to two years of simulations, from 01/01/2016 to 31/12/2017. Each episode of the environment consists of one opening day of the supermarket. The episodes start 30 minutes before the opening hour of the supermarket, they are divided in 15 minutes long time-steps and end when the supermarket closes. The framework allows simulation on two different climate zones in Italy, zone E and zone B. The simulations for zone E are obtained exploiting meteorological data from the city of Bergamo and for zone B, from the city of Catania. The agent can control just the HVAC of the sales area of the supermarket, while all the other zones are controlled automatically by the environment.

### 5.2.1 Observation Space and Reward Function

The observations that the agent will receive are composed, as it can be seen in table 5.6, by 17 features, with $T_{in}$ being the temperature of the room considered and $T_{send}$ being temperature of the air sent to the room. In addition it's important to note that some of these signals are coming from 4 previous time-steps and some from 4 time-steps ahead. The External air Temperature at t+4 is obtained through real weather forecasting for the specific day and city considered.

The objective of the RL algorithm is to save energy while satisfying comfort constraints. The translation of this objective into a reward function in this environment is done with a trade-off between two reward components tuned by an hyperparameter $\lambda$. The reward at each time-step $r_t$ is then defined as the sum

$$r_t = r_E + r_C \tag{5.7}$$

with $r_E$ being the energy-related component and $r_C$ the comfort component. The formulation of these two components, naming the temperature of the zone controlled

| Feature meaning | Feature Domain | Feature Time-step |
|---|---|---|
| Season | $\{Winter, Summer\}$ | $t$ |
| Weekday | {1,2,3,4,5,6,7} | $t$ |
| Time-step of the day | 1, ....,50 | $t$ |
| Electric Cost | $\mathbb{R}$ | $t$, $t-4$ |
| Solar Radiance | $\mathbb{R}$ | $t$, $t-4$ |
| Gas Cost | $\mathbb{R}$ | $t$, $t-4$ |
| External air temperature | $\mathbb{R}$ | $t$, $t+4$ |
| $T_{in}$ Bakery, $T_{in}$ Deli, $T_{in}$ Dry Storage, $T_{in}$ Office, $T_{in}$ Produce, $T_{in}$ Produce | $\mathbb{R}$ | $t$, $t-4$ |
| Humidity Sales | $\mathbb{R}$ | $t$, $t-4$ |
| $T_{send}$ Sales | $\mathbb{R}$ | $t$, $t-4$ |

**Table 5.6:** HVAC State space

as $T_{in}$ and the range of acceptable temperatures as $[\underline{T_b}, \overline{T_b}]$, is given by

$$r_E = -\lambda c$$
$$r_C = \begin{cases} 0 \quad if\, T_{in} \in [\underline{T_b}, \overline{T_b}] \\ -\exp(p) \quad \text{otherwise} \end{cases} \tag{5.8}$$

where $c$ is the sum of the electric and thermal energy costs at time-step $t$ and $p = \max(\underline{T_b} - T_{in},0) + \max(T_{in} - \overline{T_b},0)$ is the comfort constraint component. Following the *stochastic constraint* tuning done by the HVAC paper authors we fix $\lambda = 2$ as the best value for the RL-agent performance.

### 5.2.2 Action Space

Differently from the LunarLander-v2 environment, not all the actions are always accepted by the environment. More specifically the actions that the agent can take are listed in table 5.7, where it's also specified the specific condition for which some actions are made available by the environment.

| Action Index | Action Meaning | Action Feasibility Period |
|---|---|---|
| 0 | Set point=15.0°C | Winter |
| 1 | Set point=15.5°C | Winter |
| 2 | Set point=16.0°C | Winter |
| 3 | Set point=16.5°C | Winter |
| 4 | Set point=17.0°C | Winter |
| 5 | Set point=17.5°C | Winter |
| 6 | Set point=18.0°C | Winter |
| 7 | Only-Ventilation winter | Always |
| 8 | Set point=24.0°C | Summer |
| 9 | Set point=24.5°C | Summer |
| 10 | Set point=25.0°C | Summer |
| 11 | Set point=25.5°C | Summer |
| 12 | Set point=26.0°C | Summer |
| 13 | Only-Ventilation summer | Always |
| 14 | Turn Off | 30 minutes windows before the opening and closing hours |

**Table 5.7:** HVAC Action space

### 5.2.3 Dataset Collection and Generation

Approaching this Industry-related problem we switched focus from the D4RL standard dataset composition to a more task-oriented approach. Since we had already two different type of agents, a classic rule-based control method and a Fitted-Q-Iteration (FQI) Algorithm implemented and tested on this environment we derived from them, for each available climate zone, two different type of datasets.

#### FQI Dataset

This dataset was obtained by online training the FQI agent on the 2016 year simulation, following the approach proposed in [23], and simultaneously collecting its transition with the environment. The datasets result in 18092 transitions $(s, a, s', r)$, that we can expect to cover enough high reward regions of the state-action space. We can grasp an idea of how the data is distributed in terms of actions with respect to the time step of the episode looking at Figures 5.2, 5.3.

For the winter related data in both climate zones we have good coverage of almost all actions, while for the summer data, the action choices look more concentrated at a few actions, especially for zone E. One issue that is inherent of this kind of state space and thus of this data, is that climate data as well as energy costs data, that compose our states, can be highly variable from year to year and can also present some anomalous events that in the end could drastically change the states distribution. Then, limit our data to just one year of experience is for sure a limitation, but at the same time a good benchmark of a real case scenario for our

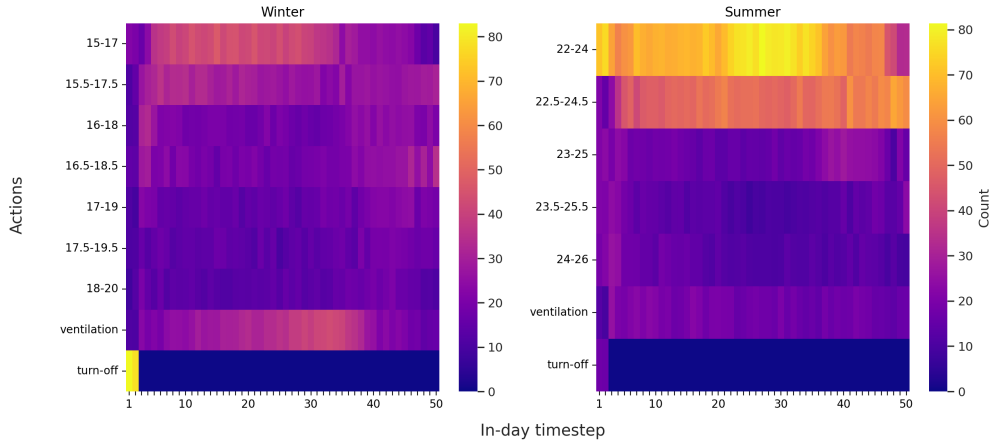



**Figure 5.2:** Heatmap of the number of times an action has been selected during the winter (left) and summer (right) training period over each time step in a generic Episode by the FQI policy on climate zone E

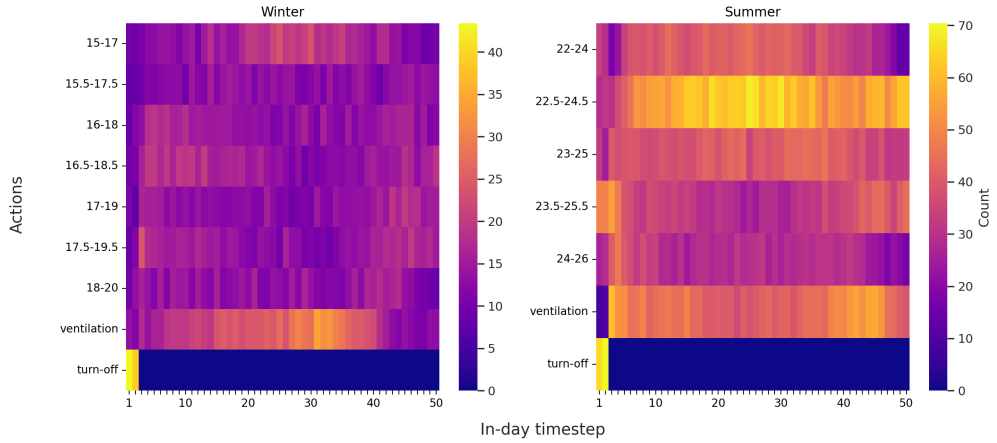



**Figure 5.3:** Heatmap of the number of times an action has been selected during the winter (left) and summer (right) training period over each time step in a generic Episode by the FQI policy on climate zone B

algorithm.

### PI-CONST Dataset

This dataset was obtained by letting the Rule-Based agent (PI-CONST) act in the environment in the 2016 year and collecting its transitions. The number of transitions sampled is the same of the FQI Data and it also presents the same issue of reduced state space coverage. What is very interesting of this dataset is that only three actions are selected by the agent and thus are represented in the dataset, as we can see in Figures 5.4, 5.5. This condition makes the data almost pathological for Offline-RL algorithms. One of the key assumption seen in Chapter 4 for those algorithms, was indeed to be trained on a dataset with enough exploration of high reward regions in the state-action space. Nevertheless we will evaluate our algorithm on this data because it represents the most simple and cheaper data to be collected from an industry point of view, since the Rule-based control method represent an industry standard according to [23].

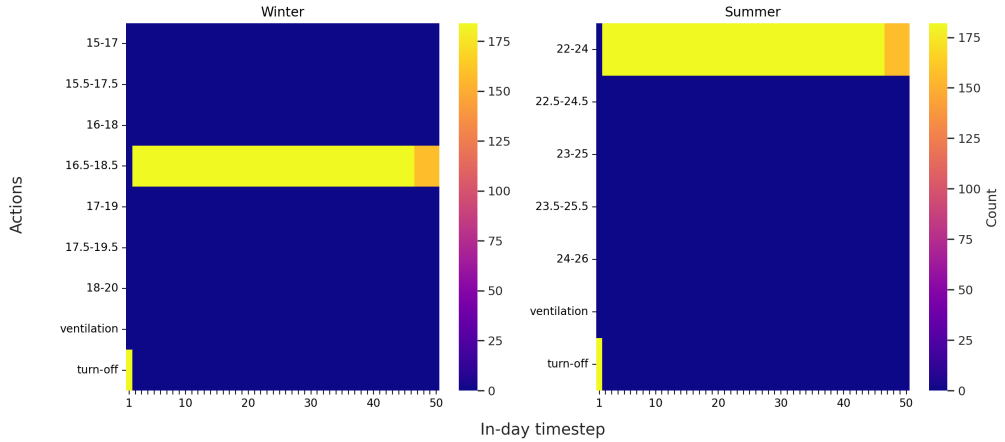



**Figure 5.4:** Heatmap of the number of times an action has been selected during the winter (left) and summer (right) training period over each time step in a generic Episode by the PI-CONST policy on climate zone E simulation

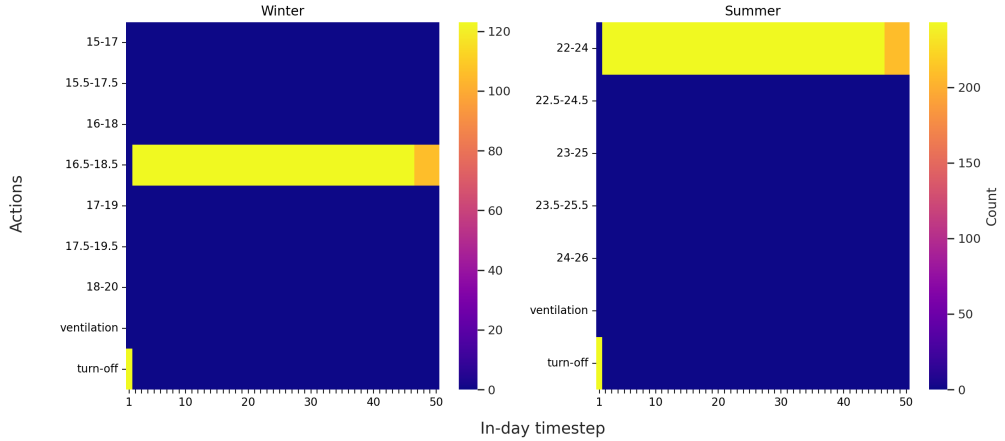



**Figure 5.5:** Heatmap of the number of times an action has been selected during the winter (left) and summer (right) training period over each time step in a generic Episode by the PI-CONST policy on climate zone B simulation

# Chapter 6

# Experiments Results

In this chapter we show the results obtained by our proposed algorithm TD4-BC on the two environments presented in chapter 5. In particular, at first, we focus on validating our algorithm on the standard framework Lunar Lander with respect to its continuous-action parent algorithm TD3-BC, and then we show the results on the industry-related task of HVAC control.

## 6.1 Lunar Lander

Since the LunarLander-v2 framework presents a continuous-action control version (*LunarLanderContinuous-v2*) with the same goals and the same Reward function formulation, we now test the state of the art algorithm TD3-BC on this task to gather baseline performance for our derived discrete approach TD4-BC.

### 6.1.1 TD3-BC on LunarLanderContinuous-v2

The algorithm is trained for 1 million steps on each of the presented datasets and evaluated every 5000 steps. Each evaluation consists of letting the trained agent act in the environment for ten episodes.

All the hyperparameters were set following the original TD3-BC paper [16], even the main hyperparameter to control the balance between RL and BC as $\alpha = 2.5$. Each experiment was repeated over five different random seeds.

We now briefly present the learning curves of the algorithm for each of the dataset it was trained on and later, in subsection 6.1.3, we show its performance after the completion of the training process.

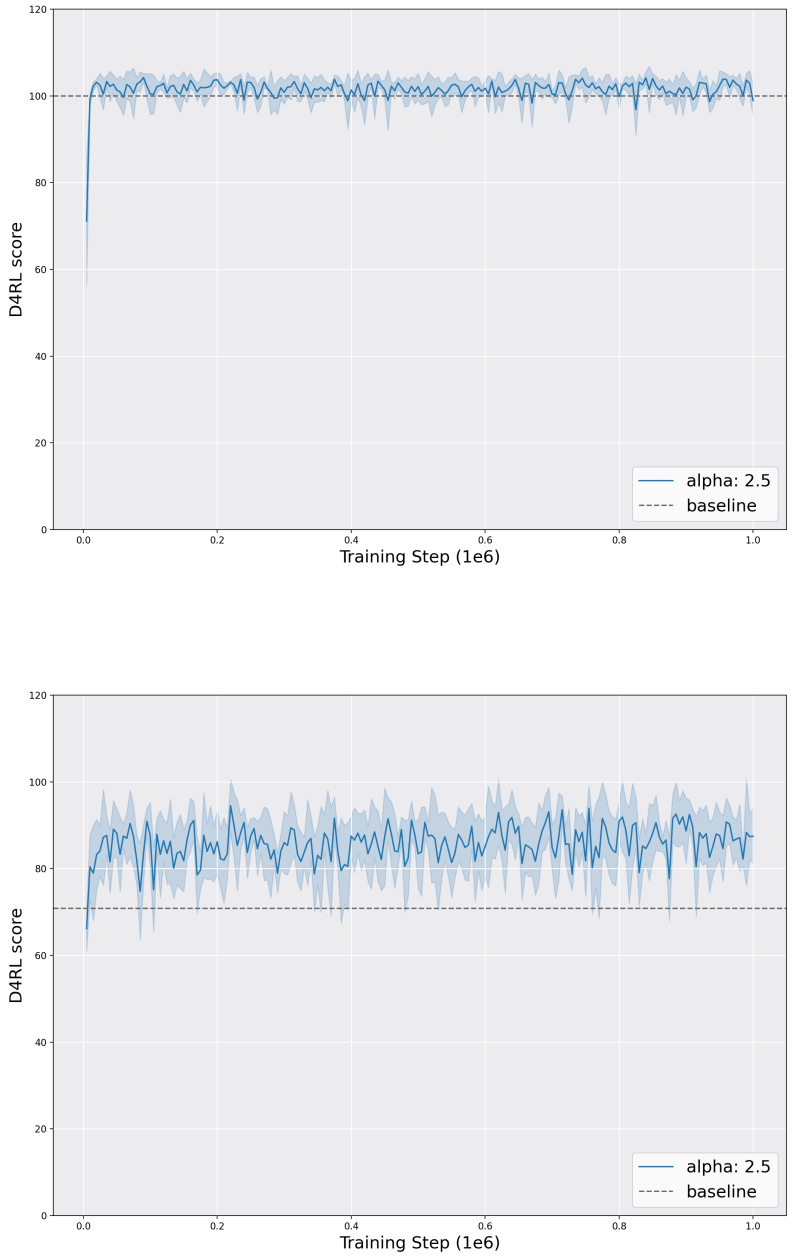



**Figure 6.1:** Learning Curves for TD3-BC trained on expert (top) and medium (bottom) quality datasets. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds. Baseline is the D4RL score associated with the policy that collected the dataset

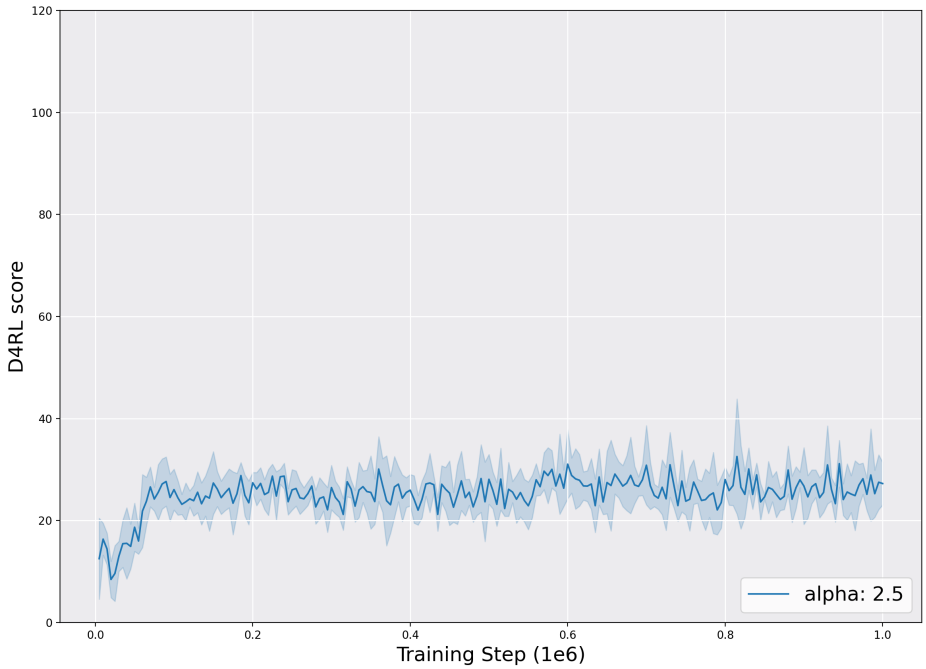



**Figure 6.2:** Learning Curves for TD3-BC trained on random quality dataset. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds. Baseline is the D4RL score associated with the policy that collected the dataset

As we can see in Fig.6.1, for the medium and expert dataset, after few thousand iterations, TD3 is consistently outperforming the performance of the behavioral policy. Instead for random dataset, as it can be seen in Fig.6.2, the process is slower, taking approximately $10^5$ training steps to reach a stable performance that outperforms the random baseline.

### 6.1.2 TD4-BC on LunarLander-v2

Now we focus on the proposed algorithm TD4-BC on the LunarLander-v2 environment for the three quality datasets. Since as we presented in section 4.4, the objective for the policy update is changed with respect to TD3-BC, we decided to, at first, tune our algorithm with respect to two main hyperparameters that have influence on that update: the learning rate **lr** and alpha $\alpha$ that controls the balance between RL and BC. We evaluate our algorithm over the product of these

two hyperparameter set: $\{3 \times 10^{-4}, 3 \times 10^{-5}, 3 \times 10^{-6}\} \times \{2.5, 5, 7.5\}$, all other hyperparameters are set following the TD3-BC paper settings. The same training and evaluation procedure, used for the TD3-BC experiments before, are used here with each experiment repeated over five different random seeds.

We now show the TD4-BC learning curves as well as the progression over training step of the expected Q-value estimate used in the policy update Eq.(4.10):

$$\pi(s)Q(s, *) \tag{6.1}$$

and of the critic loss:

$$\Big[r + \gamma \min_{i=1,2} Q_{\theta'_i}\Big(s', \pi_{\phi'}(s')\Big)\Big] - Q_{\theta_i}(s, a) \tag{6.2}$$

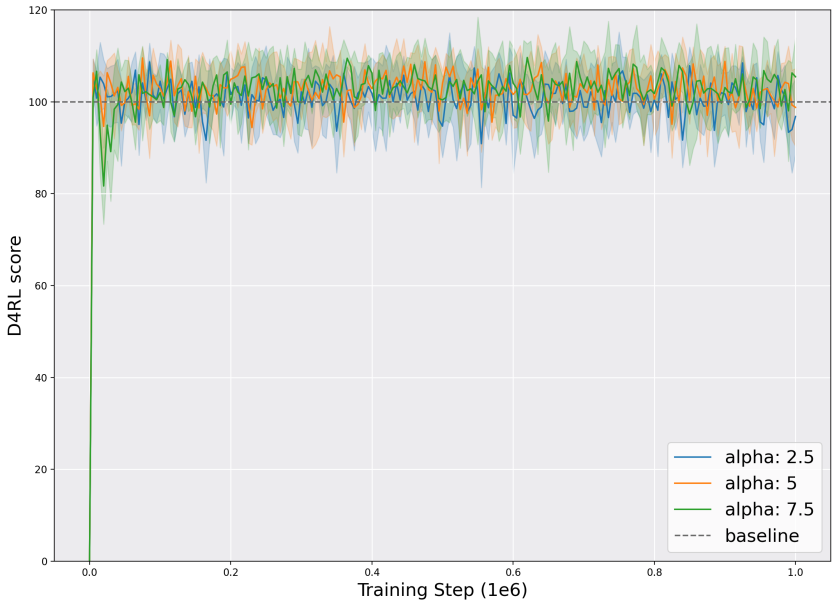
### Expert Dataset

From Fig. 6.3a we can see how, as we could expect from an high reward regions dataset, the alpha hyperparameter is not determinant on the learned policy performance. The algorithm is indeed able to reach the baseline for all the tested alphas and even outperforms it during different parts of the training process.

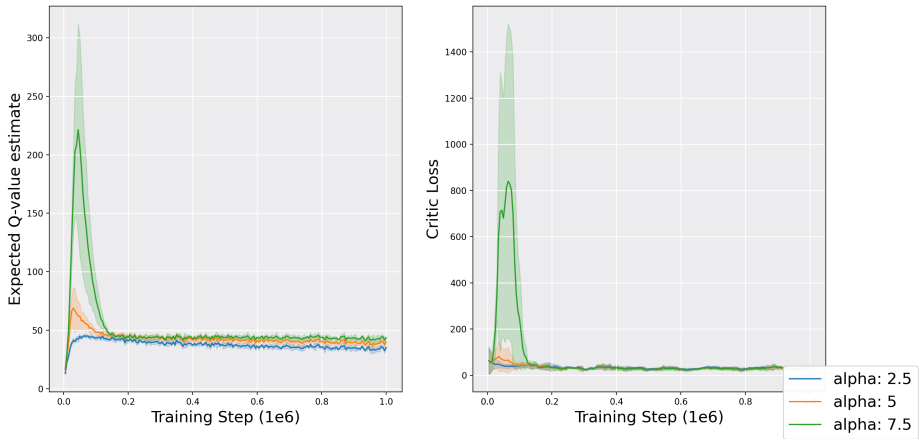
Looking at Fig. 6.3b, instead, we can note how bigger alpha cause an initial overestimation of the Q-value. This can also be seen in Fig. 6.3a in the temporary drop in performance in the initial evaluations for both $\alpha = 5$ and $\alpha = 7.5$. However the algorithm is able to recover from the overestimation and still perform well on the task.

Reducing the learning rate to $3 \times 10^5$ we still obtain a policy that reaches on average the baseline without a critical reduction in convergence speed. From Fig.6.4b we can note an interesting phenomenon, for $\alpha = 7.5$ the algorithm starts to overestimate the expected Q-values, without any consecutive significant loss in performance. We think that this shows how even if an overestimation happens, in high reward dataset like this, it can affects the majority of actions and not just OOD ones. This permits to avoid the collapse of the learning procedure usually associated with overestimation in Offline-RL.

Lastly, with learning rate equal to $3 \times 10^{-6}$ we actually observe a slower curve to convergence in Fig.6.5a. The algorithm still reach the baseline and partially outperforms it for the two bigger alphas around $2 \times 10^5$ training steps. Again with a reduce learning rate we can observe from Fig.6.5b that the algorithm starts, after $500 \times 10^3$ training steps, to overestimate the expected Q-value, again without an observable reduction in performance. We think that this phenomenon can still be explained with the same reasoning that we presented before.

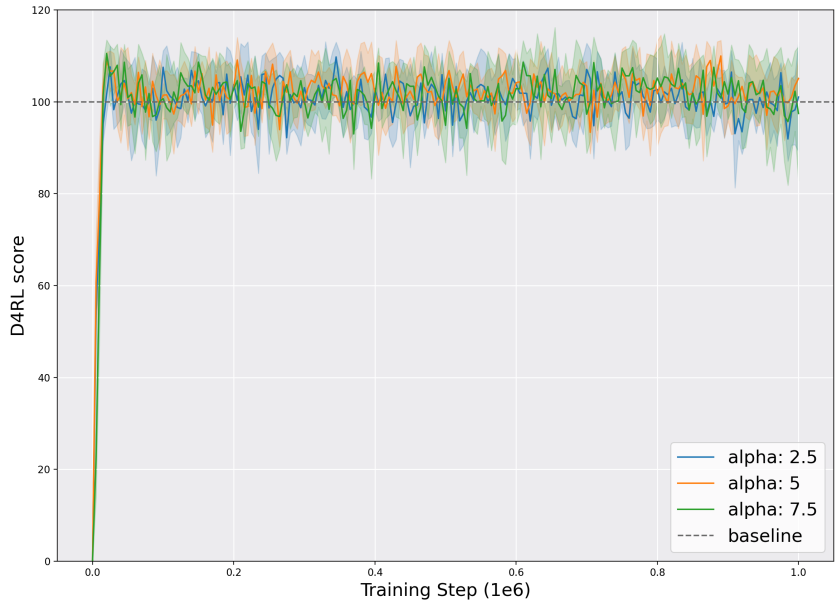



**(a)** Learning Curves for TD4-BC.

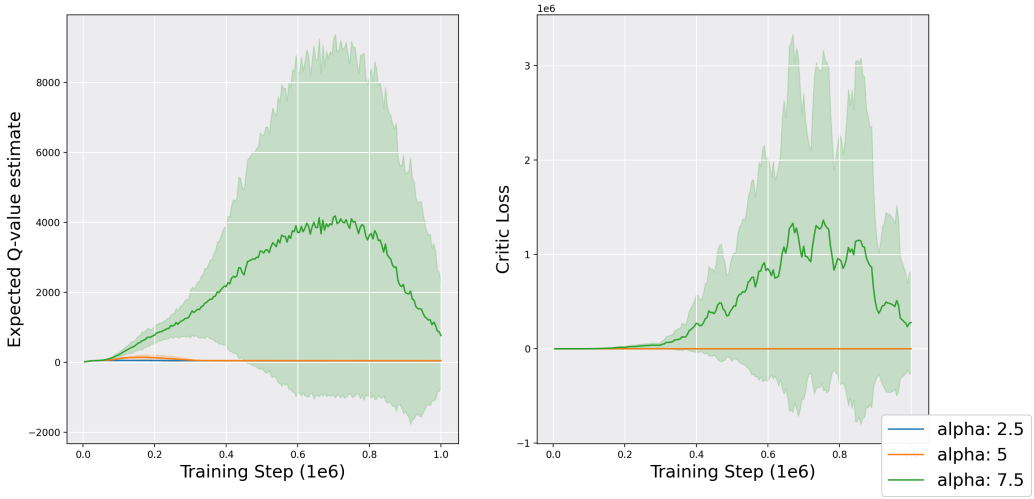



**(b)** Expected Q-values evolution(left) and Critic loss evolution(right) for TD4-BC.

**Figure 6.3:** TD4-BC trained on expert quality dataset and learning rate equal to $3 \times 10^{-4}$. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

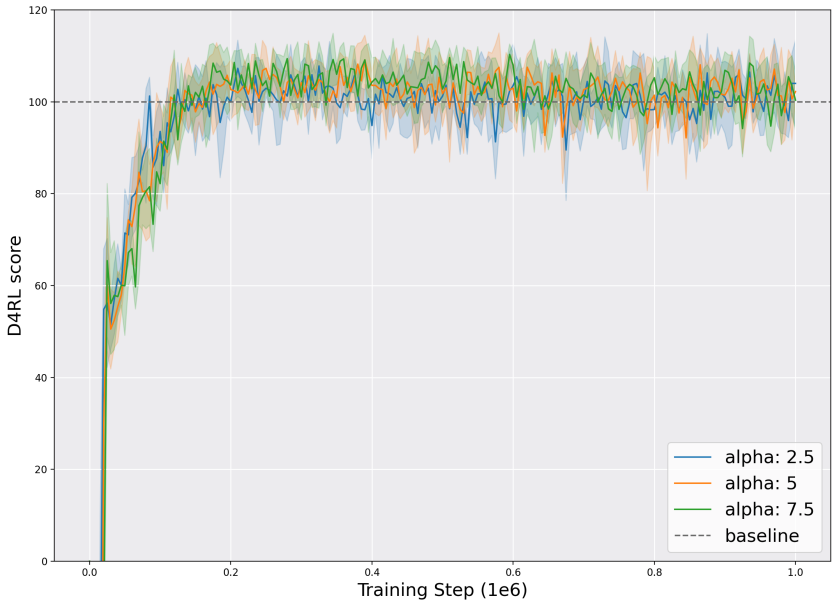



**(a)** Learning Curves for TD4-BC.

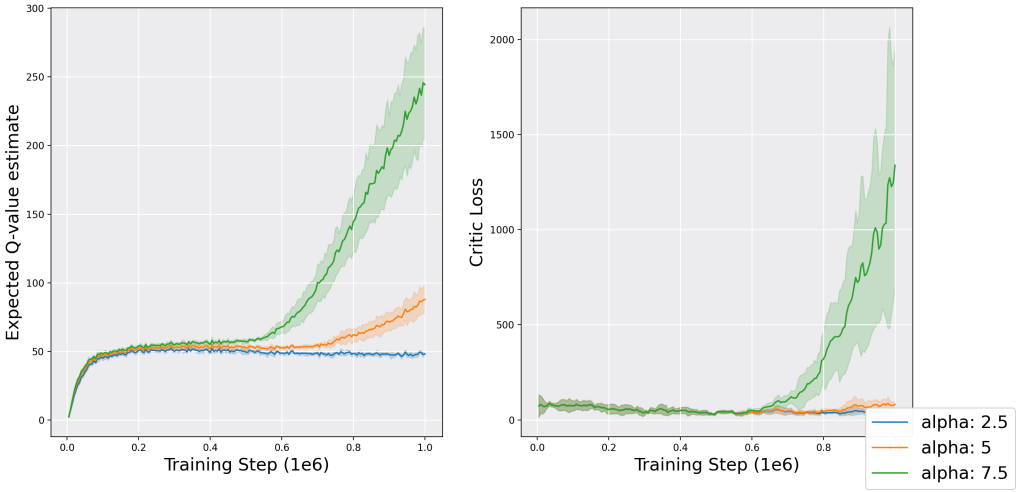



**(b)** Expected Q-values evolution(left) and Critic loss evolution(right) for TD4-BC.

**Figure 6.4:** TD4-BC trained on expert quality dataset and learning rate equal to $3 \times 10^{-5}$. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.




**(a)** Learning Curves for TD4-BC.




**(b)** Expected Q-values evolution(left) and Critic loss evolution(right) for TD4-BC.

**Figure 6.5:** TD4-BC trained on expert quality dataset and learning rate equal to $3 \times 10^{-6}$. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

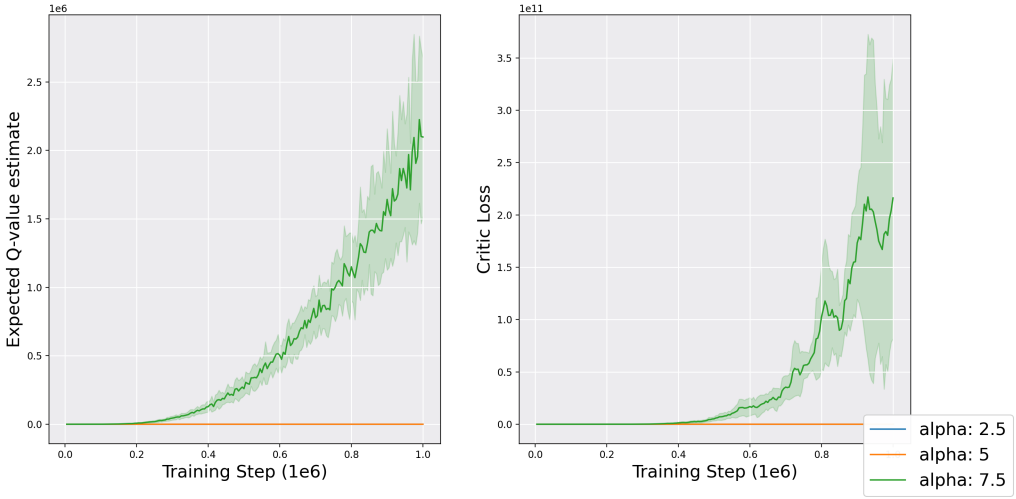
### Medium Dataset

Contrary to what could be naively expected, the performance for the algorithm with $\alpha = 7.5$ not only does not outperforms the baseline but it is significantly worst. This can be explained with the overestimation of OOD actions that characterize Offline-RL and that can be noted also in Fig.6.6b, where the expected Q-value estimate as well as the critic loss increase exponentially over training steps. For smaller alpha this does not happen and the algorithm reaches and overcomes the baseline for the majority of the training steps.

Reducing the learning rate to $3 \times 10^{-5}$, makes the learning curves present an interesting behaviour. In Fig. 6.7a, we can observe how after a few thousand training steps all algorithms reach near-optimal performance, but then progressively return to a near baseline performance level. More specifically for $\alpha = 7.5$, the decrease continue until it reaches sub-baseline performance level. This last drop in D4RL score can be again explained through the overestimation phenomenon, as it can be observed in Fig. 6.7b as well.

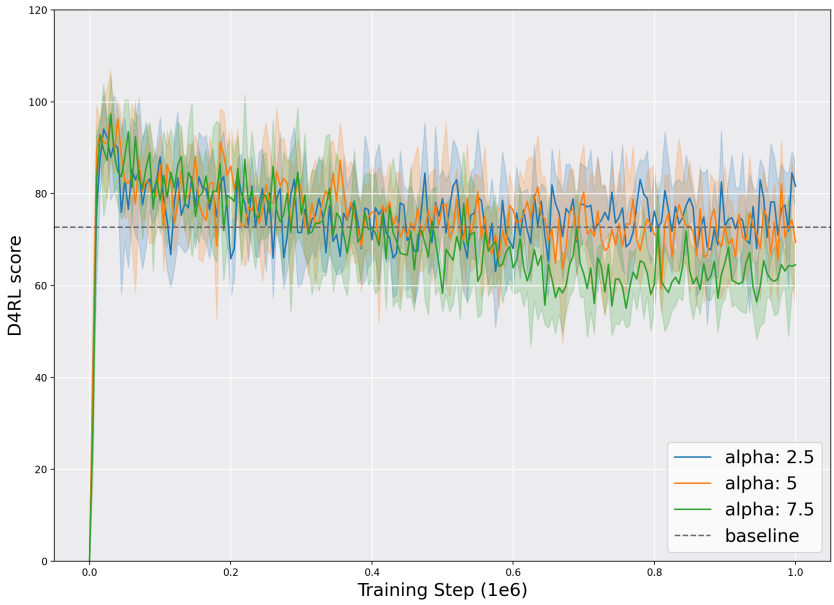
The experiments with the smaller learning rate as expected present a slower learning curve, that reaches again near-optimal performance. Unfortunately even with a smaller learning rate the curves start decreasing after reaching their maximum. This behavior resemble the one found for learning rate equal to $3 \times 10^{-5}$, and is not related to OOD actions overestimation, especially for $\alpha = 2.5$ and $\alpha = 5$ as we can see in 6.8b. We think that this phenomenon resembles more the classical statistic overfitting, where the agents became too well suited for the data seen in training and start to perform worse on the evaluations as the training steps increase, especially since the algorithm present a Behavioral Cloning loss component.


**(a)** Learning Curves for TD4-BC.

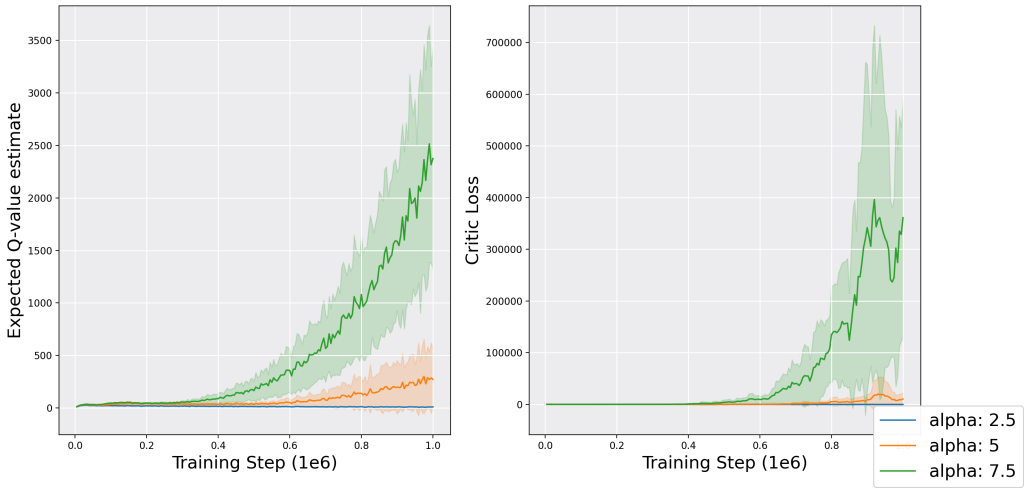



**(b)** Expected Q-values evolution(left) and Critic loss evolution(right) for TD4-BC.

**Figure 6.6:** TD4-BC trained on medium quality dataset and learning rate equal to $3 \times 10^{-4}$. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.



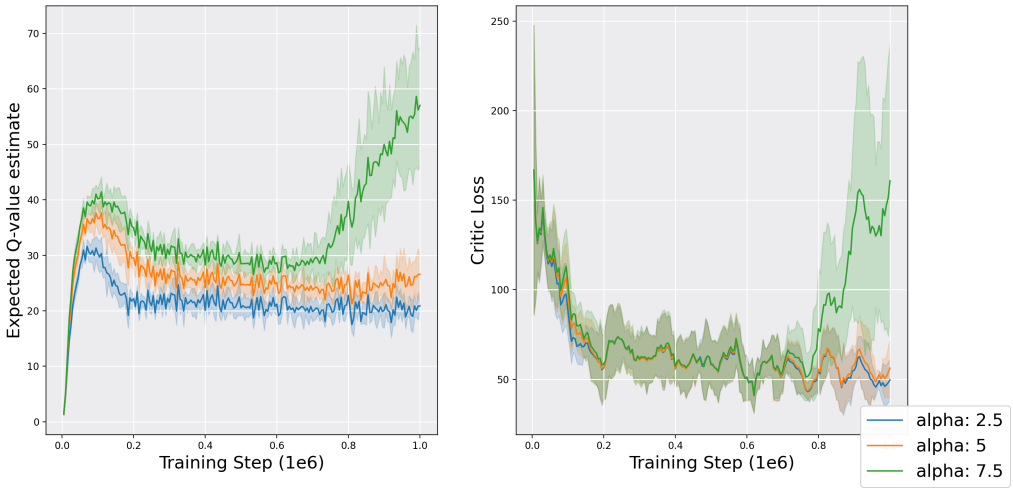

**(a)** Learning Curves for TD4-BC.




**(b)** Expected Q-values evolution(left) and Critic loss evolution(right) for TD4-BC.

**Figure 6.7:** TD4-BC trained on medium quality dataset and learning rate equal to $3 \times 10^{-5}$. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.


**(a)** Learning Curves for TD4-BC.




**(b)** Expected Q-values evolution(left) and Critic loss evolution(right) for TD4-BC.

**Figure 6.8:** TD4-BC trained on medium quality dataset and learning rate equal to $3 \times 10^{-6}$. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

### Random Dataset

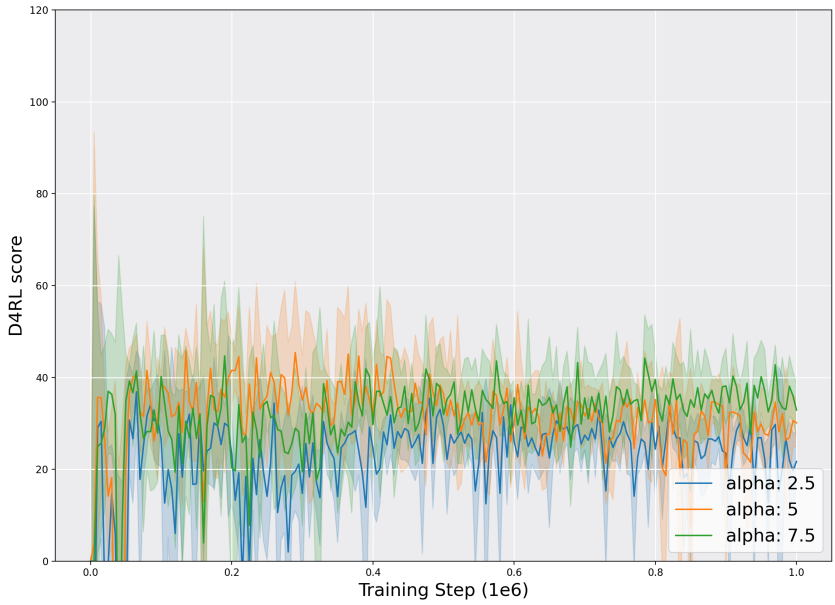
The random dataset is the one in which our algorithm performs better, reaching a peak of more than 50 points in D4RL score with respect to its baseline, as we can see in Fig.6.10a. From both Fig.6.9a and Fig.6.10a we can see a significant improvement with respect to the baseline and a direct dependency with respect to the hyperparameter $\alpha$. Bigger alphas correspond to average better performance. More specifically the algorithm trained with learning rate equal to $3 * 10^{-5}$ and $\alpha = 7.5$ reaches a convergence of increase in performance with respect to the baseline of around 40 points in D4RL score.

For the last set of experiment with learning rate equal to $3 \times 10^{-6}$ the algorithm struggle a little more in the training, with slower curves and to less performing policies. Furthermore the algorithms trained with $\alpha = 7.5$ and $\alpha = 5.0$ are even prone to OOD actions overestimation. Indeed we can see in Fig.6.11a and Fig.6.11b how after around $7 \times 10^5$ training steps the expected Q-values estimates start an increasing trend and the performances a steep decreasing one.

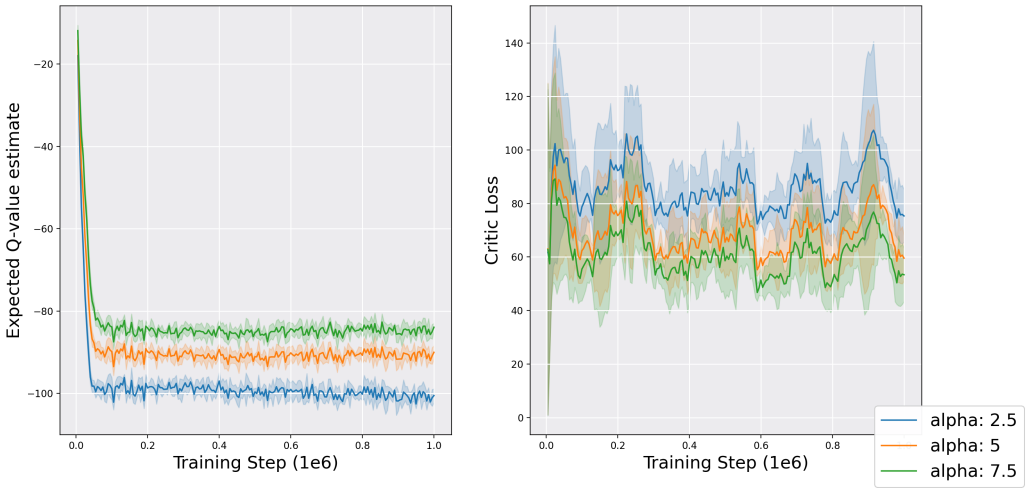
Table 6.1 presents the D4RL scores of the algorithm on the last ten evaluation episodes to determine the learning rate and alpha pair that guarantees overall best performance level. We found that the best values of hyperparameters that bring to a better and stabler algorithm on average across the three quality datasets are $\alpha = 5$ and learning rate equal to $3 \times 10^{-4}$.

| | | Expert | Medium | Random | Total |
|---|---|---|---|---|---|
| $\alpha = 2.5$ | lr=3e-4 | 99.08 ± 4.38 | 73.55 ± 3.07 | 21.88 ± 7.28 | 194.51 ± 14.73 |
| | lr=3e-5 | 99.31 ± 2.47 | 76.42 ± 3.46 | 10.09 ± 9.6 | 185.82± 15.53 |
| | lr=3e-6 | 101.21 ± 3.36 | 80.32 ± 4.96 | 23.92 ± 1.66 | 205.45 ± 9.98 |
| $\alpha = \mathbf{5.0}$ | **lr=3e-4** | 101.51 ± 2.71 | 75.36 ± 3.91 | 29.38 ± 2.79 | **206.25 ± 9.41** |
| | lr=3e-5 | 101.46 ± 1.97 | 72.64 ± 5.41 | 23.85 ± 8.95 | 197.95 ± 16.33 |
| | lr=3e-6 | 102.55 ± 2.52 | 80.82 ± 5.97 | 7.36 ± 6.95 | 190.73 ± 15.44 |
| $\alpha = 7.5$ | lr=3e-4 | **103.82** ± 4.58 | 66.59 ± 2.27 | 35.58 ± 3.69 | 205.79 ± 10.54 |
| | lr=3e-5 | 99.01 ± 9.97 | 63.26 ± 3.53 | **36.66** ± 2.66 | 198.93 ± 16.16 |
| | lr=3e-6 | 102.02 ± 2.22 | **81.52** ± 4.58 | -1.41 ± 5.59 | 182.13 ± 12.39 |

**Table 6.1:** Average D4RL scores over the final 10 evaluations and 5 seeds of TD4-BC on the three quality datasets. ± captures the standard deviation over seeds.

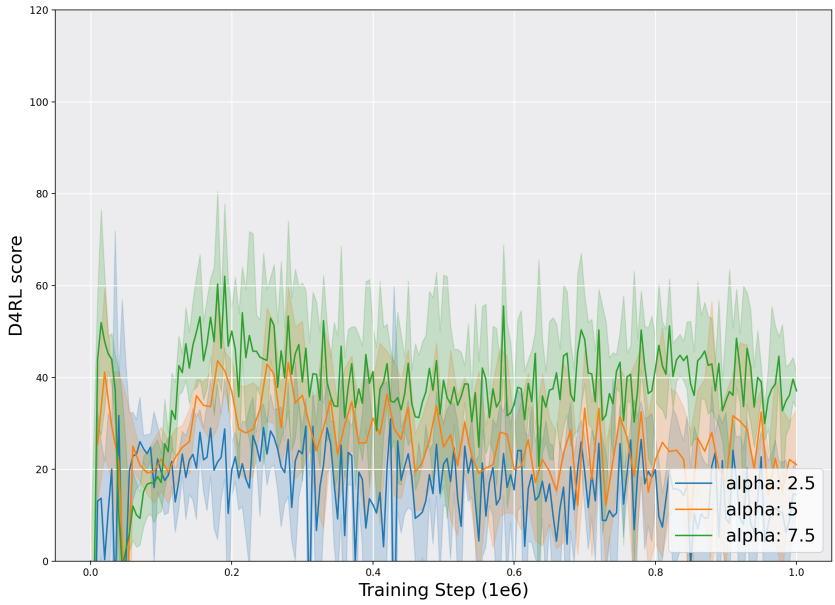



**(a)** Learning Curves for TD4-BC.

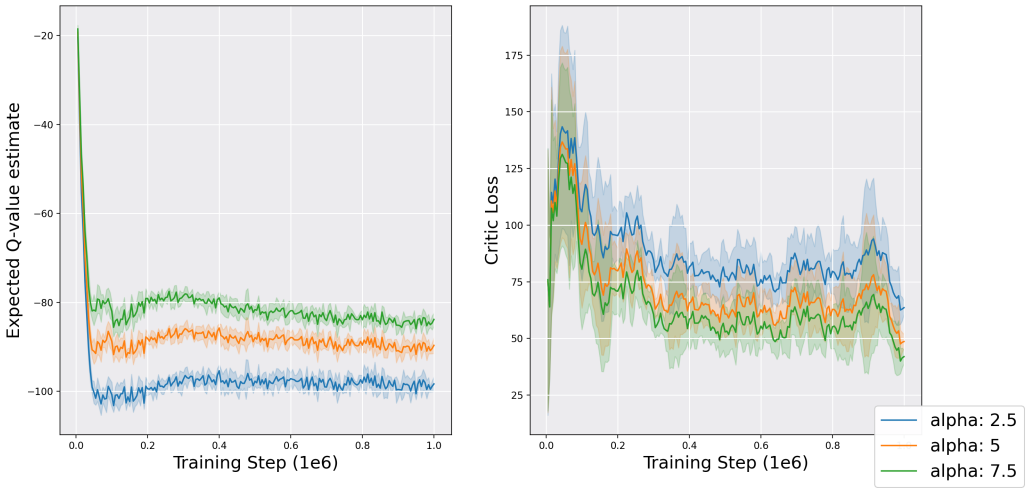



**(b)** Expected Q-values evolution(left) and Critic loss evolution(right) for TD4-BC.

**Figure 6.9:** TD4-BC trained on random quality dataset and learning rate equal to $3 \times 10^{-4}$. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

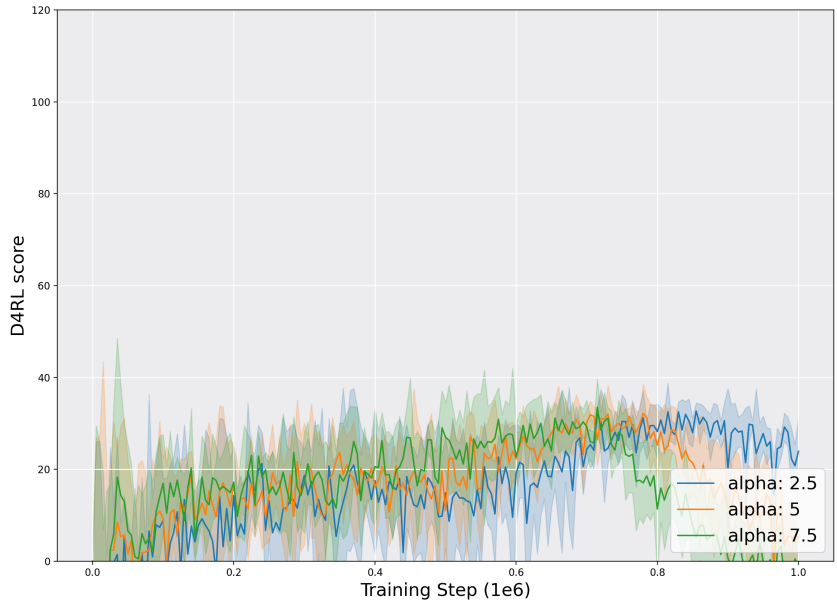



**(a)** Learning Curves for TD4-BC.

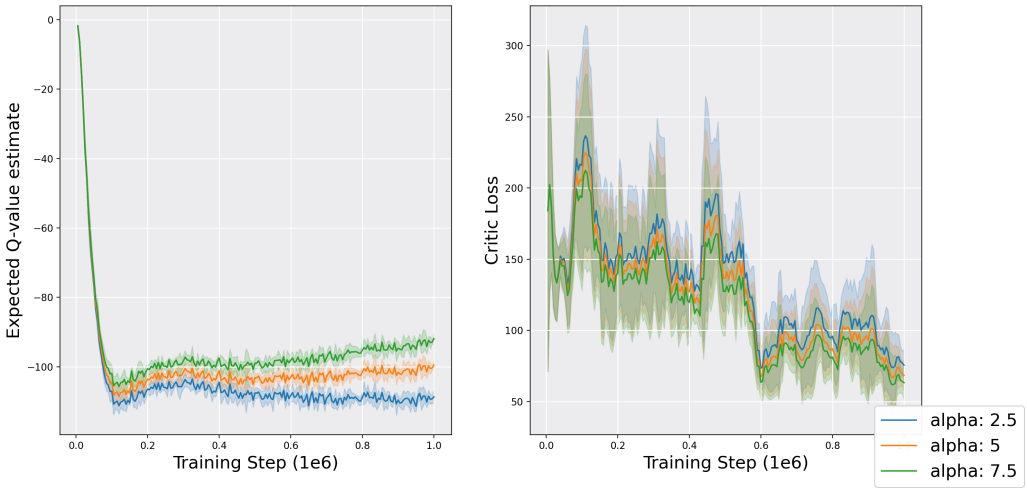



**(b)** Expected Q-values evolution(left) and Critic loss evolution(right) for TD4-BC.

**Figure 6.10:** TD4-BC trained on random quality dataset and learning rate equal to $3 \times 10^{-5}$. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.




**(a)** Learning Curves for TD4-BC.




**(b)** Expected Q-values evolution(left) and Critic loss evolution(right) for TD4-BC.

**Figure 6.11:** TD4-BC trained on random quality dataset and learning rate equal to $3 \times 10^{-6}$. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

### Policy Update Frequency

In Offline-RL we need to try to limit the overestimation errors as much as we can. To this end, we also want to test if augmenting the policy update frequency (**p-freq**) of TD4BC, with respect to the standard value used for TD3-BC in its paper, would make the agent benefits of more stable and less variance policy updates. We then evaluate TD4-BC varying the policy update frequency from 2 to 3 and with the three different learning rates used before. In these experiments we use $\alpha = 5$ as the default value for this hyperparameter, following the previous findings.

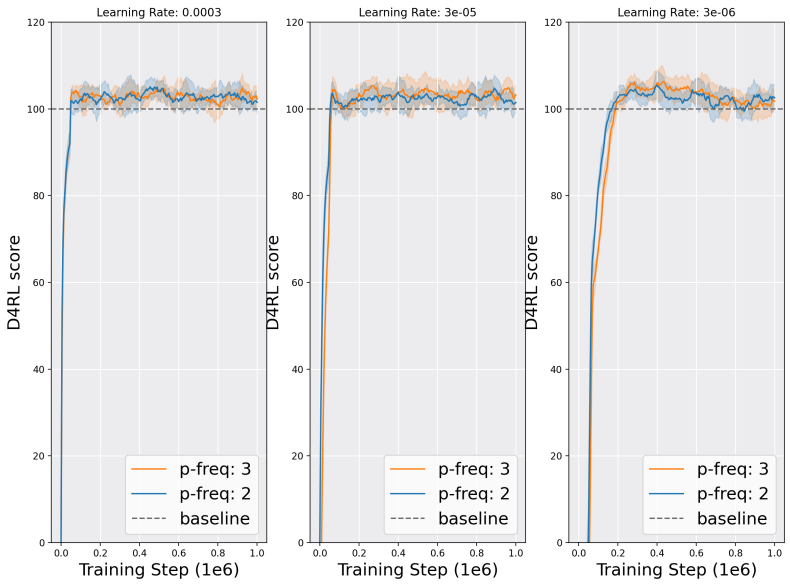



**Figure 6.12:** Learning Curves for TD4-BC trained on expert quality dataset and $\alpha = 5$ for each learning rate and policy update frequency tested. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds. Baseline is the D4RL score associated with the policy that collected the dataset.

In Figures 6.12, 6.13, 6.14, we can see how on average the agent trained with p-freq=3 reaches higher D4RL score on basically all the datasets and with all the learning rates tested. This advantage is significant but it has to be underlined how changing this hyperparameter doesn't change the overall shape of the learning curves, that suffer from the same issues found before. For example in Fig. 6.13 we can see how the slow performance degradation phenomenon is present for both policy update frequencies. Looking at final results in table 6.2, we identify the

top two performing configurations in p-freq=3 and learning rate equal to $3 \times 10^{-4}$ and $3 \times 10^{-5}$, we then decided to stick, for the rest of the experiments, with the $lr = 3 \times 10^{-4}$ as the best hyperparameters configuration since while being just two points of D4RL score behind the other, it cuts in half the standard deviation across seeds, significantly improving the algorithm stability.

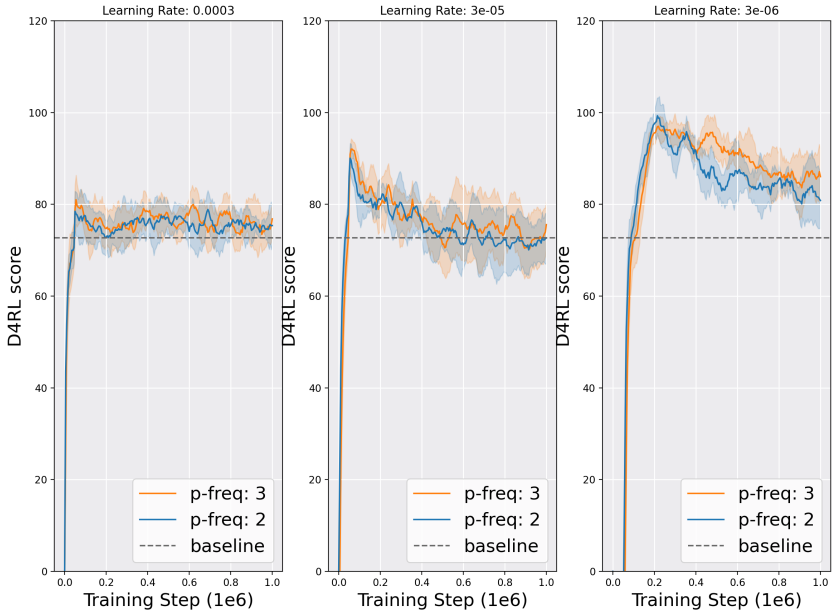



**Figure 6.13:** Learning Curves for TD4-BC trained on medium quality dataset and $\alpha = 5$ for each learning rate and policy update frequency tested. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds. Baseline is the D4RL score associated with the policy that collected the dataset.

| | | Expert | Medium | Random | Total |
|---|---|---|---|---|---|
| p-freq=3 | lr=$3 \times 10^{-4}$ | 102.26 ± 2.96 | 76.82 ± 3.00 | 34.72 ± 1.33 | **213.80 ± 7.29** |
| | lr=$3 \times 10^{-5}$ | 103.20 ± 2.35 | 75.57 ± 5.30 | 36.73 ± 7.68 | **215.50** ± 15.33 |
| | lr=$3 \times 10^{-6}$ | 101.80 ± 2.11 | 86.03 ± 6.07 | 15.29 ± 7.36 | 203.12 ± 15.54 |
| p-freq=2 | lr=$3 \times 10^{-4}$ | 101.51 ± 2.71 | 75.36 ± 3.91 | 29.38 ± 2.79 | 206.25 ± 9.41 |
| | lr=$3 \times 10^{-5}$ | 101.46 ± 1.97 | 72.64 ± 5.41 | 23.85 ± 8.95 | 197.95 ± 16.33 |
| | lr=$3 \times 10^{-6}$ | 102.55 ± 2.59 | 80.82 ± 5.97 | 7.36± 6.95 | 190.73 ± 15.44 |

**Table 6.2:** Average D4RL scores over the final 10 evaluations and 5 seeds of TD4-BC with $\alpha = 5.0$ on the three quality datasets and with different learning rates (lr) and policy update frequencies (p-freq). ± captures the standard deviation over seeds.

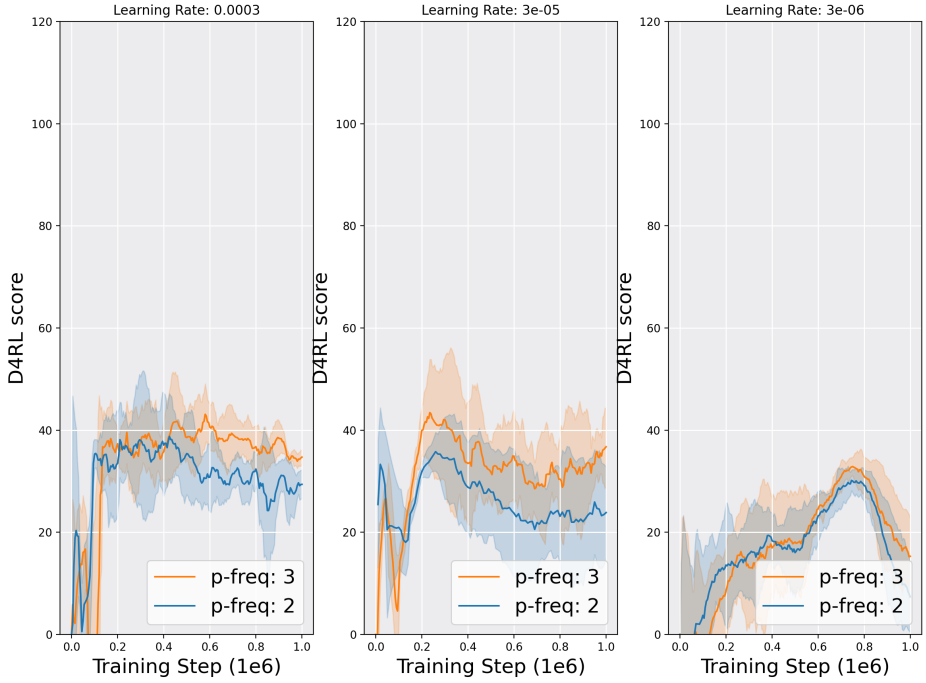



**Figure 6.14:** Learning Curves for TD4-BC trained on random quality dataset and $\alpha = 5$ for each learning rate and policy update frequency tested. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

### 6.1.3 TD3-BC and TD4-BC Comparison

| | TD3-BC | TD4-BC |
|---|---|---|
| Random | 26.48 ± 2.62 | 34.72 ± 1.33 |
| Medium | 87.13 ± 2.63 | 76.82 ± 3.00 |
| Expert | 102.36 ± 0.25 | 102.26 ± 2.96 |
| Total | 215.97 ± 5.5 | 213.80 ± 7.29 |

**Table 6.3:** Average D4RL scores over the final 10 evaluations and 5 seeds of TD3-BC and TD4-BC. ± captures the standard deviation over seeds.

Finally, from table 6.3 we can state that TD4-BC has overall a similar level of performance with respect to its continuous-action parent TD3-BC. It outperforms it in the random setting while falling behind with the medium quality dataset.

Looking at the graphs summarizing the training process presented before we can also point out how the D4RL score variation between evaluations across different point in the learning curve is much higher with respect to TD3-BC. This can be explained by the fact that the available actions in the discrete Lunar Lander environment are drastically different one from the other, while the continuous environment accepts small variations in the actions selected. This means that the discrete policy updates have inherently more drastic consequences in the agent behaviour, thus causing more variance over the evaluations scores at different training process points.

## 6.2 HVAC Control Retrofitting

In this section, we switch the focus on the industry related task of smart HVAC optimal control. As seen in chapter 5, the datasets that we use in these experiments are limited in state, action space coverage, especially the PI-CONST dataset that presents samples of just three actions over the fifteen actions available. Keeping this in mind, we still want to investigate the behavior of TD4-BC in such case where the fundamental assumption, that we discussed in chapter 4, of high reward regions represented in the data, cannot be guaranteed.

For both the datasets presented in chapter 5, the number of transitions collected (18092) is much lower than the one million transitions of the Lunar Lander datasets. Since in the previous section 6.1, we observed a progressive performance degradation across almost all datasets at the increase of training steps, for these experiments we evaluate our algorithm at different points in the training procedure with a maximum number of training updates fixed to $150 \times 10^3$.

Then, we also test the two different Online Fine-Tuning approaches presented in chapter 4. The metrics that we use are the mean return over the whole test year and over the last six months of testing. We also show the evolution over testing episodes of the difference in return between the agents trained with TD4-BC and the behavior agent that collected that the data.

At first, we focus on the results for the FQI-Data and then we switch on the PI-CONST Data. Each analysis is performed on both climate zones simulation, zones E and B, and repeated over five different random seeds.

### 6.2.1 FQI Data Experiment

For this set of experiment we use the FQI agent used to collect the data as the baseline performance. This agent is let to act and simultaneously train during the testing episodes.

Looking at Fig. 6.15, we can determine how the learned policy has similar performance to the baseline except for two particular areas in the testing episodes that show significantly difference between baseline and TD4-BC policies. The first region shows high variability in terms of return gains with respect to the baseline while we have a strong and stable return gain in the second region.

Looking at the different training lengths we can derive that too many training steps like $150 \times 10^3$ do not help the agent, even more they make it performs slightly worse. At the same time if we look at the agent trained for only $10 \times 10^3$ steps we also see a slight performance reduction in the zone around the $300^{th}$ episode, suggesting the existence of a sweet spot between $25 \times 10^3$ and $75 \times 10^3$ training steps.

Switching the focus to climate zone B, we plot the return deltas evolution in Fig.

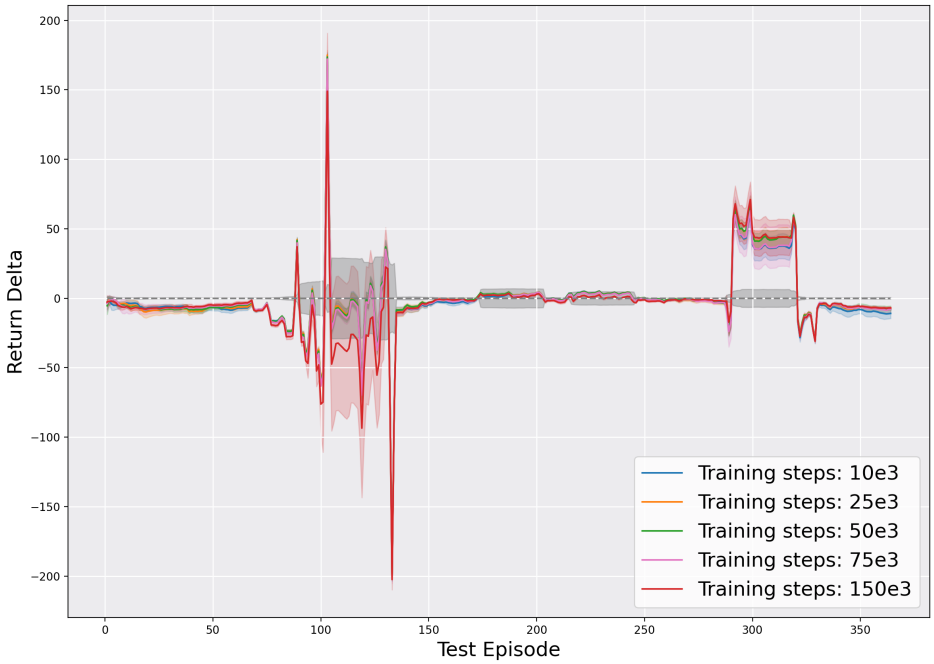



**Figure 6.15:** Curves of Return deltas with respect to behavioral baseline, over testing episodes on zone E, for policies trained with different number of training steps. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

6.16, where we can still observe a similar behavior of our agent with respect to the other climate zone, both in terms of return and of dependency on training steps.

We summarize these results into table 6.4, from which we can derive the best number of training iterations.

From this table, we can confront also the average performance of our agent with respect to the baseline. The best performing agent on the two zones is the one trained for $50 \times 10^3$ steps and if compared to the baseline has overall similar performance. As already seen in Figures 6.15 and 6.16, the agent struggle at the beginning while being much more efficient on the last six months. It's important to note that with respect to the FQI agent our TD4-BC agents are not trained simultaneously during testing, thus it's even more significant that our best agent, without any online training interaction is able to reach such level of performance. Furthermore we can also state that these learned policies are not just copy of their behavioral policies. Indeed if we look at Figures 6.17, 6.18 with respect to the

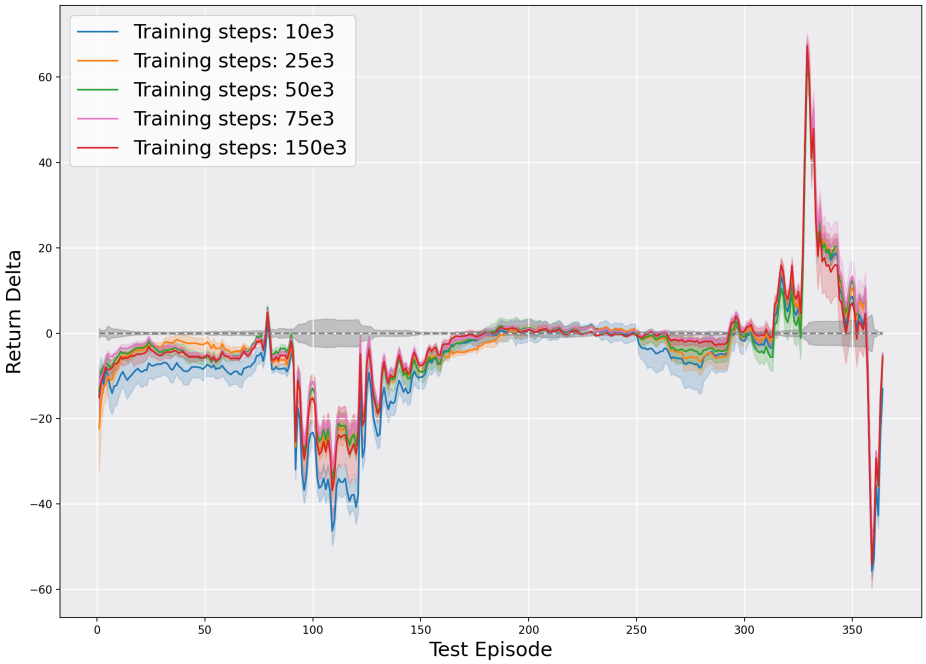



**Figure 6.16:** Curves of Return deltas with respect to behavioral baseline, over testing episodes on zone B, for policies trained with different number of training steps. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

| | | Zone E (12m) | Zone E (6m) | Zone B (12m) | Zone B (6m) |
|---|---|---|---|---|---|
| FQI-baseline | | -317.26 ± 2.91 | -201.41 ± 1.63 | -104.95 ± 0.37 | -117.98 ± 0.43 |
| TD4-BC | tr-steps=$10 \times 10^3$ | -319.08 ± 0.99 | -197.77± 1.40 | -110.91± 1.36 | -117.71 ± 1.88 |
| | tr-steps=$25 \times 10^3$ | -318.14± 0.76 | -196.11± 0.91 | -108.01± 0.42 | -115.94 ± 0.69 |
| | tr-steps=$50 \times 10^3$ | -317.98± 0.92 | -196.08 ± 1.10 | -107.99± 0.76 | -116.28 ± 1.03 |
| | tr-steps=$75 \times 10^3$ | -318.86 ± 1.89 | -197.24 ± 2.88 | -106.71 ± 0.55 | -114.58 ± 0.78 |
| | tr-steps=$150 \times 10^3$ | -320.52± 4.31 | -196.40± 2.21 | -108.09± 1.08 | -116.04 ± 1.18 |

**Table 6.4:** Average Return over 5 seeds for each climate zone over the total test period (12m) and its last six months (6m), ± captures the standard deviation across seeds

heatmaps of their respective behavioral policies in Figures 5.2, 5.3, it's evident how the TD4-BC agent is not simply imitating the behavioral policies.

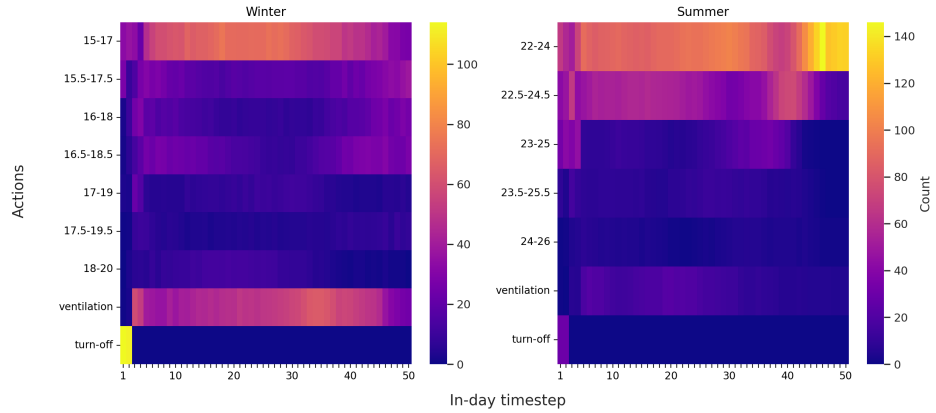



**Figure 6.17:** Heatmap of the number of times an action has been selected during the winter test period(left) and summer test period (right) over each time step in a generic Episode by the TD4-BC, trained for $50 \times 10^3$ steps policy, on climate zone E.

We then investigate the two online fine-tuning approaches proposed in chapter 4, namely TD4-BC-FT and TD4-BC-FTC, on our best performing agent. As stated, we implement an $\epsilon$-greedy strategy at action selection, with $\epsilon$ decayed linearly until it reaches a final epsilon ($\epsilon = 0.02$) that is kept constant for the rest of the test episodes. We test two different starting epsilon (eps-start=0.2, eps-start=0.02) for each approach.

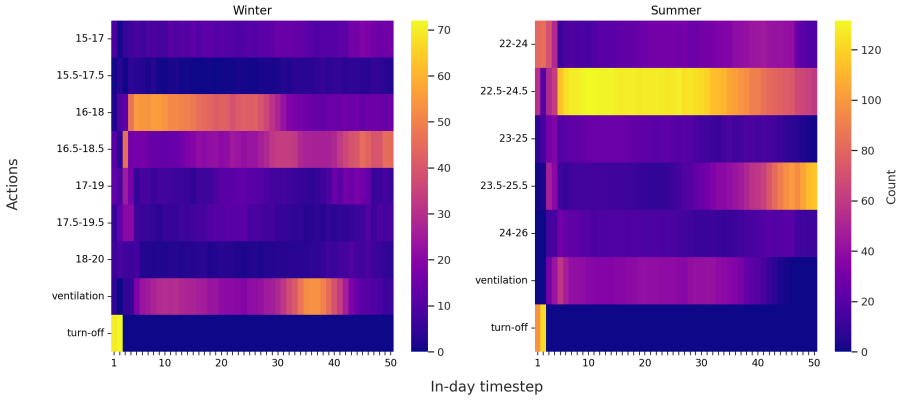



**Figure 6.18:** Heatmap of the number of times an action has been selected during the winter test period(left) and summer test period (right) over each time step in a generic Episode by the TD4-BC, trained for $50 \times 10^3$ steps policy, on climate zone B.

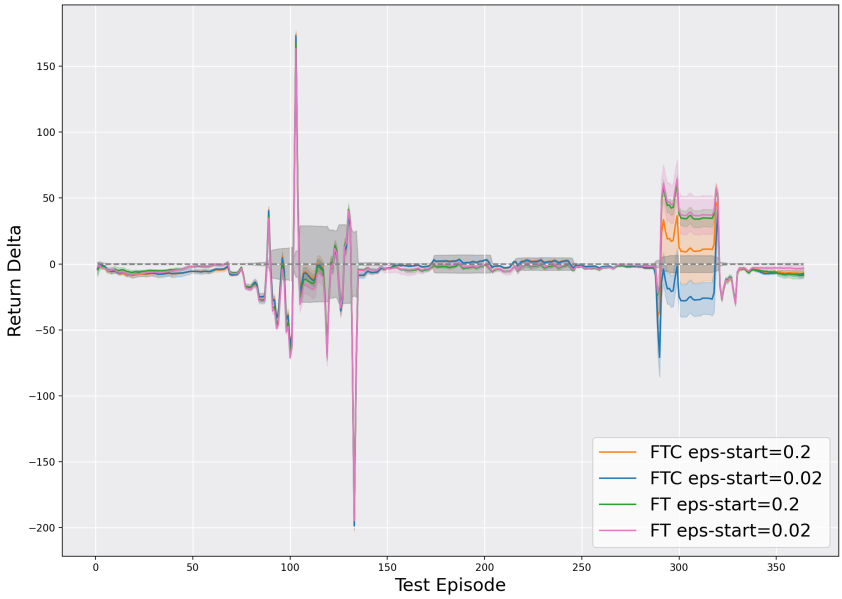



**Figure 6.19:** Curves of Return deltas with respect to behavioral baseline, over testing episodes on zone E, for TD4-BC trained for $50 \times 10^3$ steps, fine-tuned with different approaches. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

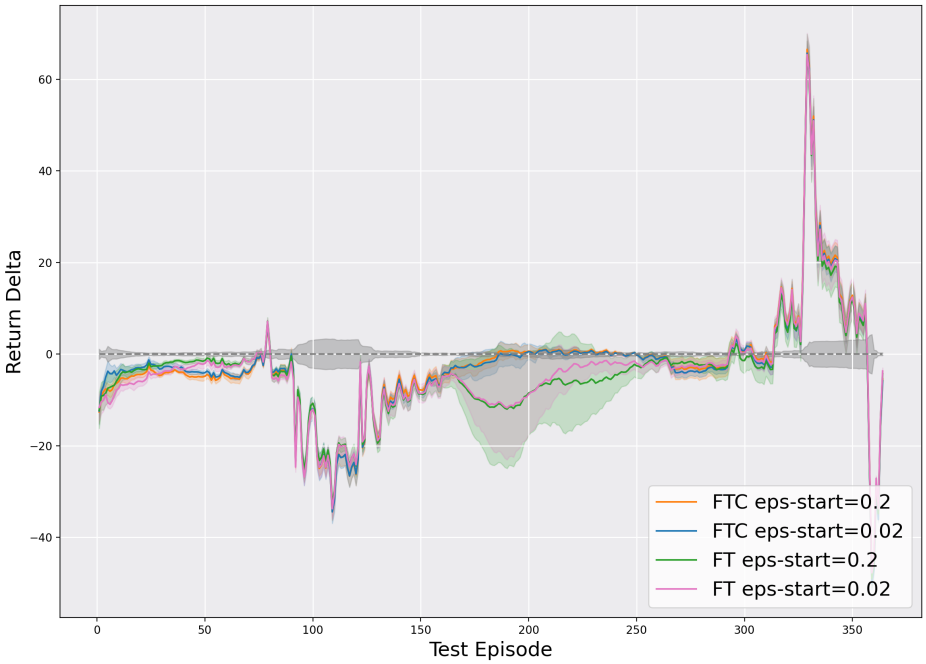



**Figure 6.20:** Curves of Return deltas with respect to behavioral baseline, over testing episodes on zone B, for TD4-BC trained for $50 \times 10^3$ steps, fine-tuned with different approaches. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

The curves of Figures 6.19 and 6.20 are not so much different than the ones of the agent trained without it. This is even clearer in table 6.5, where there is no significant worsening or improvements of the learned policies and no fine-tuning approach comes out as better with respect to the other. The lack of improvement on the first six months, that we hoped would be solved with this additional fine-tuning, can be explained by the fact that both approaches randomly sample from the replay buffer transitions. Then the adaptation to the new data is very slow thus making difficult to our agents to adapt fast in the first six months. We can still get an idea of the action distribution over the time steps of the generic episodes, through Figures 6.21 , where we can see the results of the two different fine-tuning approaches. It's interesting to see how the Naive fine-tuning is making the policy move more with respect to the Conservative one ,as we could expect, from the one trained offline.

| | | Zone E (12m) | Zone E (6m) | Zone B (12m) | Zone B (6m) |
|---|---|---|---|---|---|
| FQI-baseline | eps-start=0.02 | -317.26 ± 2.91 | -201.41 ± 1.63 | -104.95 ± 0.37 | -117.98 ± 0.43 |
| TD4-BC | eps-start=0 | -317.98± 0.92 | -196.08 ± 1.10 | -107.99± 0.76 | -116.28 ± 1.03 |
| TD4-BC-FTC | eps-start=0.2 | -320.90 ± 2.13 | -201.89± 3.84 | -107.48± 0.36 | -115.35 ± 0.65 |
| | eps-start=0.02 | -324.14± 1.35 | -208.22± 1.88 | -107.61± 0.40 | -115.82± 0.54 |
| TD4-BC-FT | eps-start=0.2 | -319.36± 1.79 | -197.60 ± 2.64 | -108.72± 0.90 | -116.86 ± 1.21 |
| | eps-start=0.02 | -319.60± 0.95 | -198.94 ± 1.69 | -108.98± 1.49 | -117.85 ± 1.94 |

**Table 6.5:** Average Return over 5 seeds, for each climate zone, over the total test period (12m) and its last six months (6m), ± captures the standard deviation across seeds. TD4-BC is the starting point for the fine-tuning experiments

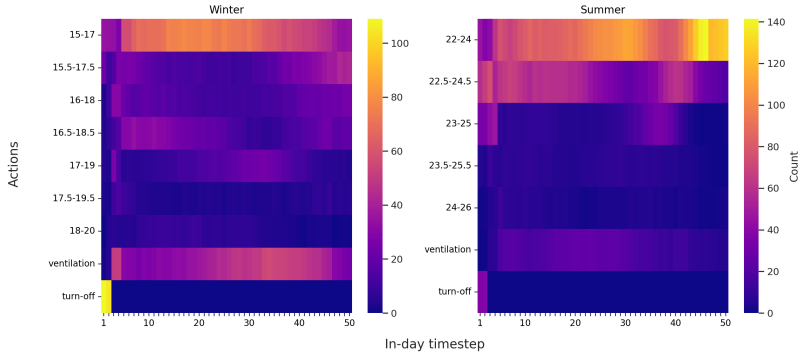



**Figure 6.21:** Heatmap of the number of times an action has been selected during the winter test period(left) and summer test period (right) over each time step in a generic Episode by the TD4-BC-FTC, with eps-start=0.02, on climate zone E.

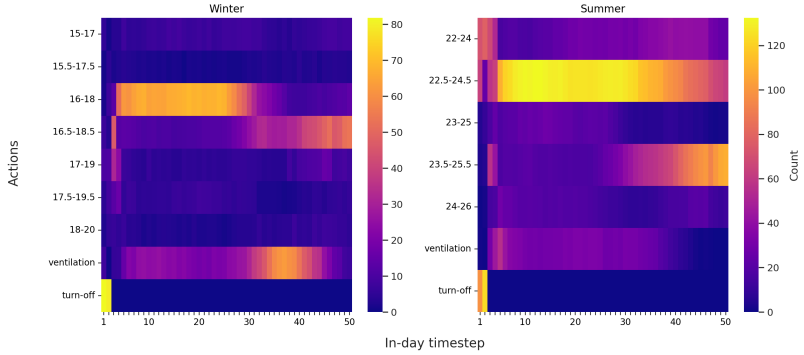



**Figure 6.22:** Heatmap of the number of times an action has been selected during the winter test period(left) and summer test period (right) over each time step in a generic Episode by the TD4-BC-FTC, with eps-start=0.02, on climate zone B.

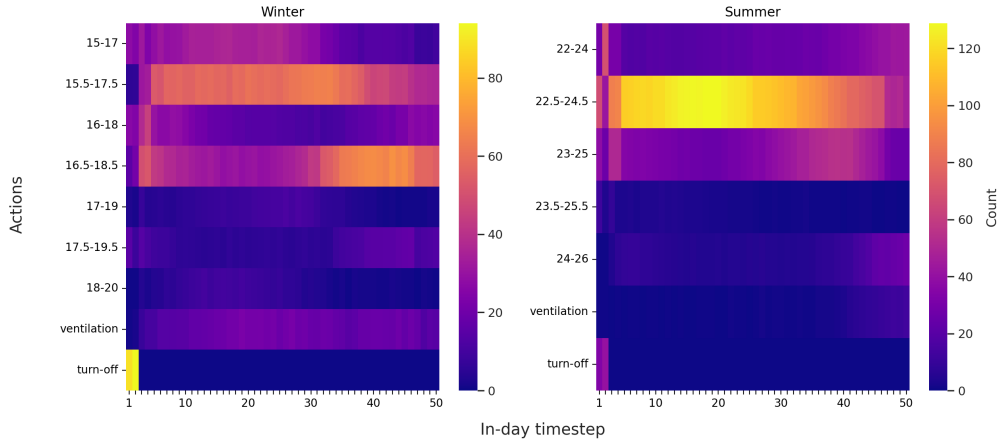



**Figure 6.23:** Heatmap of the number of times an action has been selected during the winter test period(left) and summer test period (right) over each time step in a generic Episode by the TD4-BC-FT, with eps-start=0.2, on climate zone E.

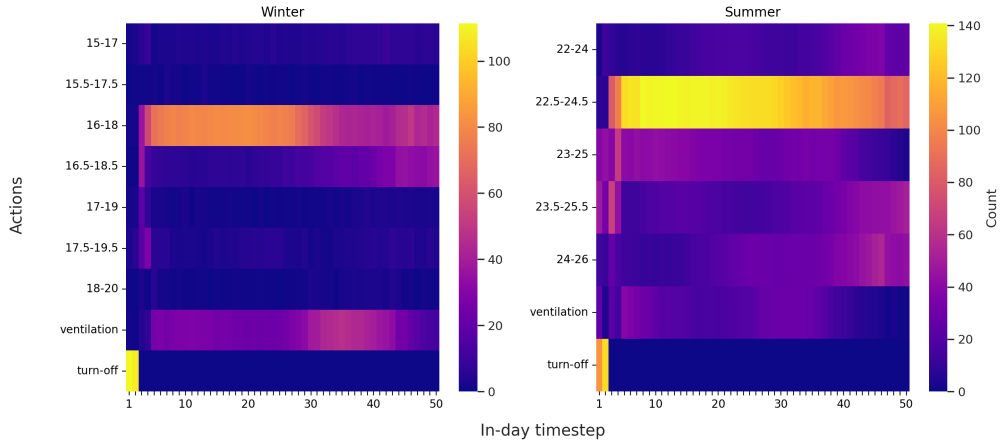



**Figure 6.24:** Heatmap of the number of times an action has been selected during the winter test period(left) and summer test period (right) over each time step in a generic Episode by the TD4-BC-FT, with eps-start=0.2, on climate zone B.

### 6.2.2 PI-CONST Data

For this set of experiments we refer to the PI-CONST agent used to collect the data that is let to act during the testing episodes, as the baseline performance. We also compare the results obtained with this setting with a TD3 discrete-action implementation trained online starting from the beginning of the test year, called

TD4. From table 6.6 and from Figures 6.25 and 6.26 we can see that TD4-BC learning process with PI-CONST data does not work very well. This was expected, as pointed out in chapters 4 and 5. In fact, this kind of data is pathological for Offline-RL algorithms and indeed OOD actions overestimation has played for sure a role in these results. We still want to investigate if there is a relation between training steps and performance. Unfortunately looking at table 6.6, the resulting average returns estimates have to much variance to derive any conclusion from them.

We can check by looking at the heatmaps of Figures 6.27 and 6.28, how the derived policy diverge completely with respect to the behavioral one for the TD4-BC trained for $50 \times 10^3$ step, favoring heavily one single action, like in the climate zone E, that was never seen in the data during training.


**Figure 6.25:** Curves of Return deltas with respect to behavioral baseline, over testing episodes on zone B, for policies trained with different number of training steps. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

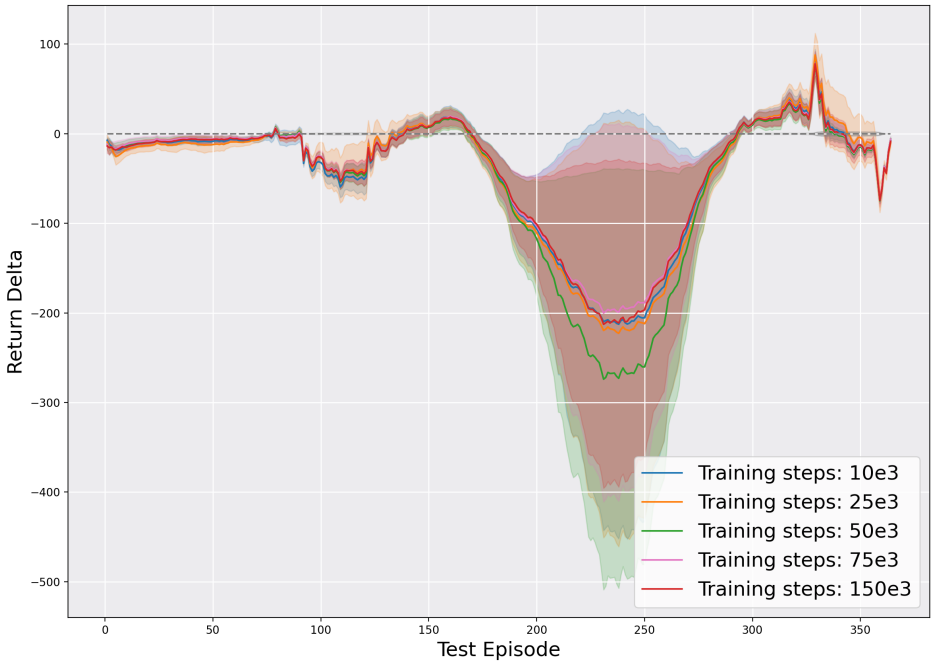



**Figure 6.26:** Curves of Return deltas with respect to behavioral baseline, over testing episodes on zone B, for policies trained with different number of training steps. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

| | | Zone E (12m) | Zone E (6m) | Zone B (12m) | Zone B (6m) |
|---|---|---|---|---|---|
| PI-CONST-baseline | | -328.63 ± 0.42 | -200.05 ± 0.089 | -118.44 ± 0.18 | -133.37± 0.40 |
| TD4-BC | tr-steps=$10 \times 10^3$ | -409.01 ± 99.82 | -336.80 ± 177.66 | -162.71± 40.28 | -207.15 ± 76.10 |
| | tr-steps=$25 \times 10^3$ | -363.48± 18.21 | -254.18± 26.32 | -163.83± 40.68 | -209.64 ± 74.28 |
| | tr-steps=$50 \times 10^3$ | -379.88± 22.16 | -275.25 ± 43.04 | -171.58± 41.17 | -226.23 ± 76.23 |
| | tr-steps=$75 \times 10^3$ | -363.87 ± 23.93 | -250.97 ± 31.08 | -159.64 ± 37.01 | -203.25± 68.00 |
| | tr-steps=$150 \times 10^3$ | -365.63± 16.25 | -247.21± 24.04 | -161.18± 32.36 | -205.61 ± 59.72 |

**Table 6.6:** Average Return over 5 seeds for each climate zone over the total test period (12m) and its last six months (6m), ± captures the standard deviation across seeds

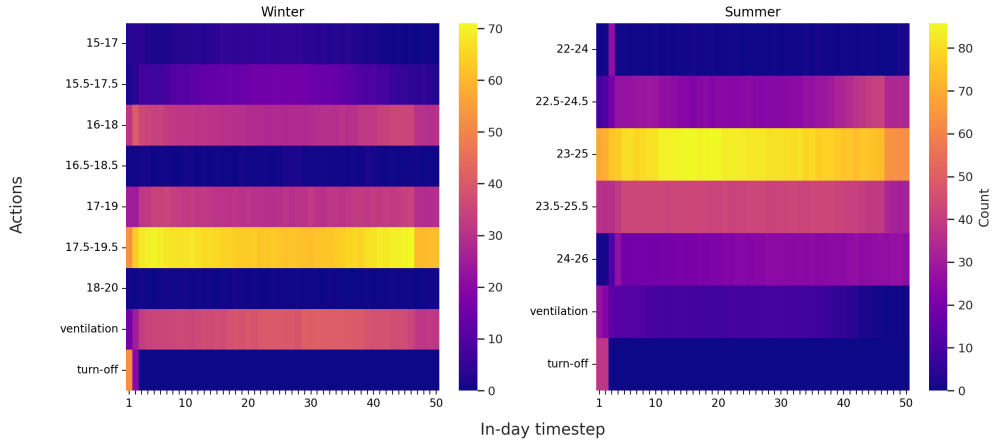



**Figure 6.27:** Heatmap of the number of times an action has been selected during the winter test period(left) and summer test period (right) over each time step in a generic Episode by the TD4-BC ,trained for $50 \times 10^3$ steps policy, on climate zone E.

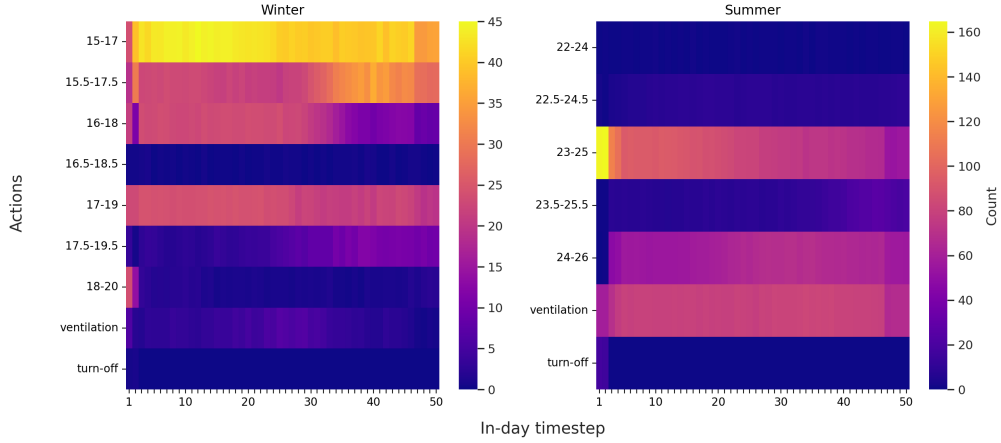



**Figure 6.28:** Heatmap of the number of times an action has been selected during the winter test period(left) and summer test period (right) over each time step in a generic Episode by the TD4-BC ,trained for $50 \times 10^3$ steps policy, on climate zone B.

At this point we want to then test if our fine-tuning approaches, combined with the offline training can be a viable solution to exploit such challenging data for smart HVAC control. The Return Deltas curves of Figures 6.29 and 6.30 shows the gains of the Conservative fine-tuning, with respect to the Naive one, on both

climate zones. This again was expected in such particular scenario. With limited action exploration in the dataset keeping the policy BC regularization term should have helped to avoid overestimation and consequent degradation of the policy quality. Instead the Naive fine-tuning, does not help our agent, for almost all episodes up to the $300^{th}$ on climate zone E and for the region between episodes $200^{th}$ and $300^{th}$ on climate zone B, it actually reduce our agent quality. The shaded area in those curves, representing the standard deviation across seeds, suggests that when such quality degradation happens, the policies are very unstable cause probably based on unstable value estimates. This behavior could be due to the combined role of an ill learned policy and online samples that are seen as OOD by the agent, as anticipated in chapter 4.

The averaged results of the fine-tuning can be seen in table 6.7. Here it is clear to see how Conservative Fine-tuning paired with TD4-BC, regardless of the starting exploration parameter, allowing us to train on very limited data and to still beat the baseline on the second half of the testing period for both climate zones. Furthermore, in zone B, the fine-tuned agents overperform on average the baseline on the whole testing period.

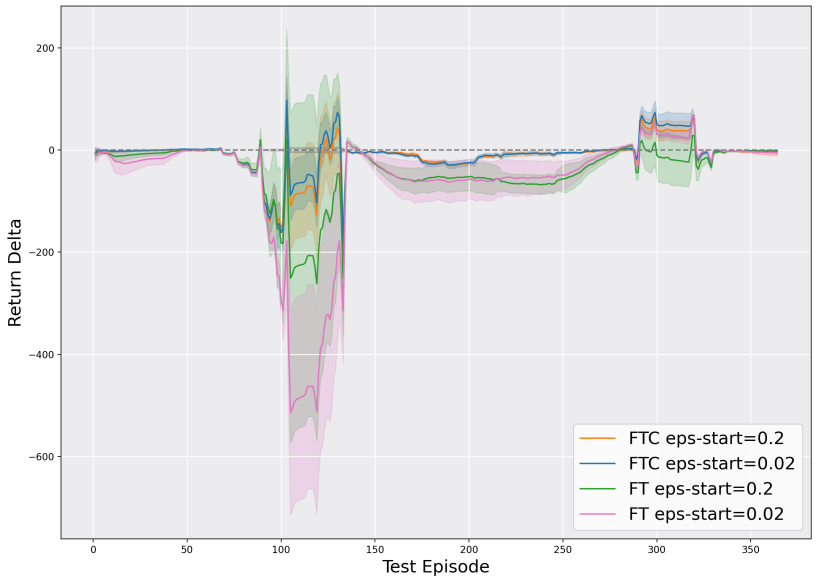



**Figure 6.29:** Curves of Return deltas with respect to behavioral baseline, over testing episodes on zone E, for policies trained with different fine-tuning approaches. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

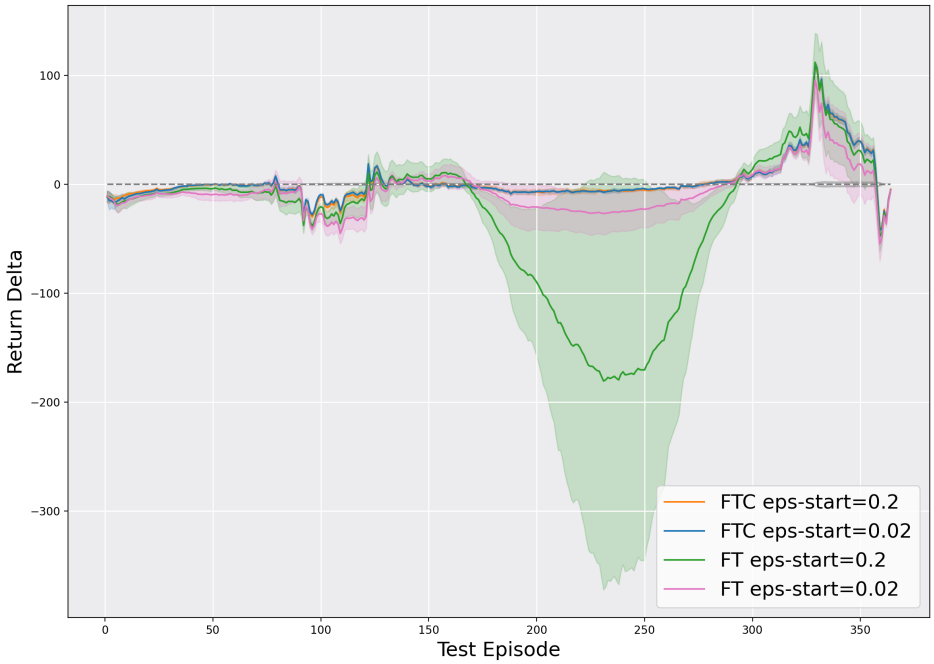



**Figure 6.30:** Curves of Return deltas with respect to behavioral baseline, over testing episodes on zone B, for policies trained with different fine-tuning approaches. Curves are averaged over 5 seeds, with shaded area representing the standard deviation across seeds.

Then to validate the role of the offline training plus online Conservative fine-tuning approach, we implemented an online discrete TD3 algorithm that is let to train on the testing period. The results, in table 6.7, show how the gain obtained by TD4-BC-FTC is not brought by the online training of its base algorithm TD3. These results also suggest how the offline training can be a viable alternative to smart HVAC optimal control, with respect to a simple implementation of an online RL agent, even in such challenging settings.

Finally, we can look at the evolution of TD4-BC-FTC policies through the heatmaps in Figures 6.31 and 6.32. These are very similar to the heatmaps of the behavior policy (Figures 5.4, 5.5), with just few different actions selection in some particular time step. Instead the policies shown in Figures 6.33 and 6.34 for TD4-BC-FT without the action regularization end up being completely different from the behavior policy, probably victim of high OOD actions overestimation.

| | | Zone E (12m) | Zone E (6m) | Zone B (12m) | Zone B (6m) |
|---|---|---|---|---|---|
| PI-CONST-baseline | | -328.63 ± 0.42 | -200.05 ± 0.089 | -118.44 ± 0.18 | -133.37± 0.40 |
| TD4 Online | eps-start=1.0 | -344.63 ± 16.97 | -219.74 ± 26.98 | -134.49 ± 12.40 | -160.57 ± 22.09 |
| TD4-BC-FTC | eps-start=0.2 | -339.62 ± 6.40 | -198.76± 2.94 | -115.83 ± 0.74 | -123.90 ± 1.27 |
| | eps-start=0.02 | -336.75± 4.63 | -196.16± 4.16 | -115.79 ± 0.59 | -123.92 ± 1.24 |
| TD4-BC-FT | eps-start=0.2 | -369.38± 26.24 | -227.53 ± 10.64 | -150.18± 36.59 | -185.83 ± 68.15 |
| | eps-start=0.02 | -387.18± 13.37 | -217.48 ± 13.73 | -124.72± 6.10 | -135.91 ±9.61 |

**Table 6.7:** Average Return over 5 seeds for each climate zone over the total test period (12m) and its last six months (6m), ± captures the standard deviation across seeds

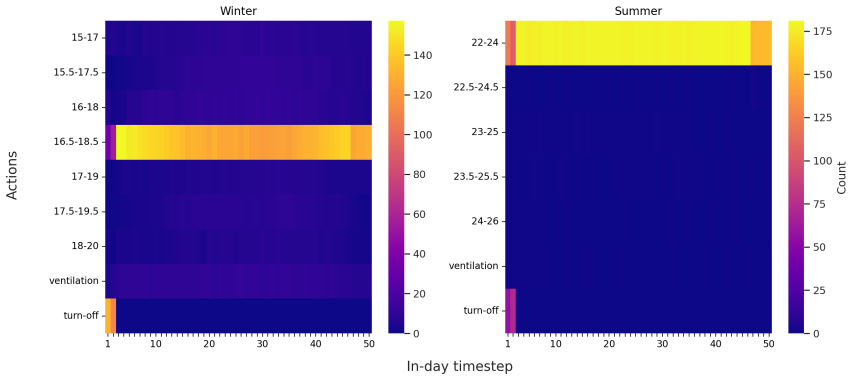



**Figure 6.31:** Heatmap of the number of times an action has been selected during the winter (left) and summer (right) test periods over each time step in a generic Episode by the TD4-BC-FTC, with eps-start = 0.02, policy, on climate zone E.

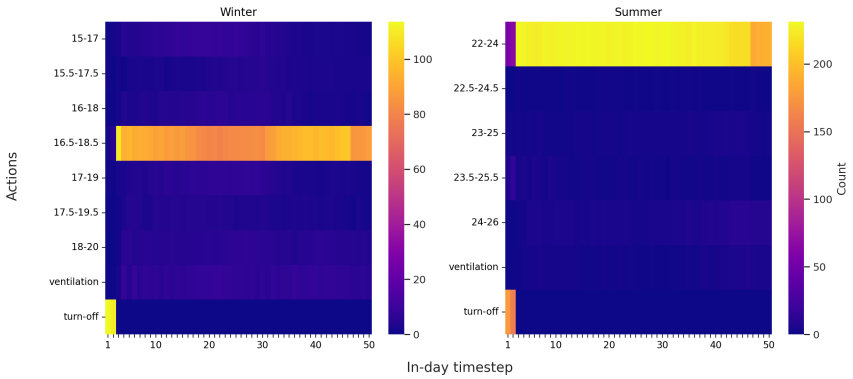



**Figure 6.32:** Heatmap of the number of times an action has been selected during the winter (left) and summer (right) test period over each time step in a generic Episode by the TD4-BC-FTC, with eps-start = 0.02, policy, on climate zone B.

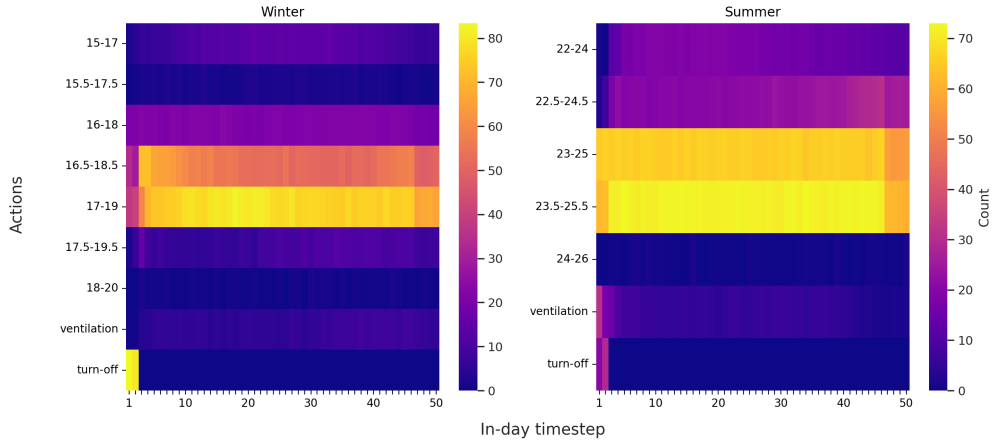



**Figure 6.33:** Heatmap of the number of times an action has been selected during the winter (left) and summer (right) test period over each time step in a generic Episode by the TD4-BC-FT, with eps-start = 0.2, policy, on climate zone E.

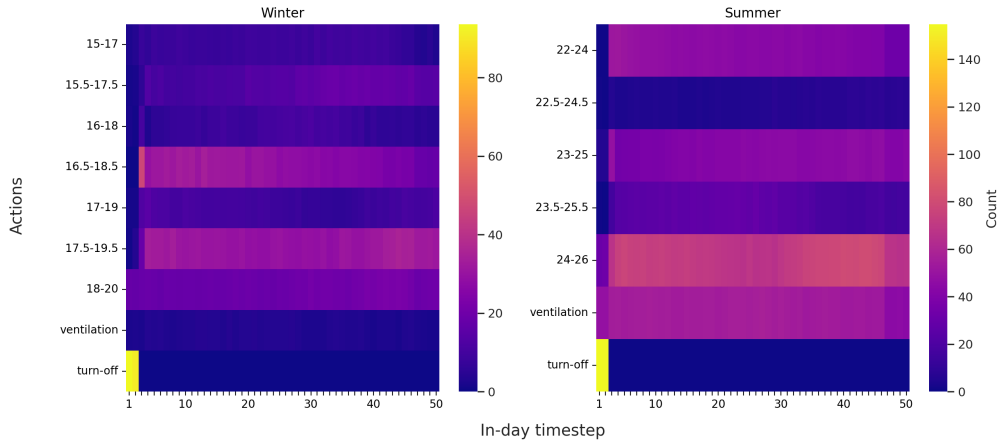



**Figure 6.34:** Heatmap of the number of times an action has been selected during the winter (left) and summer (right) test period over each time step in a generic Episode by the TD4-BC-FT, with eps-start = 0.2, policy, on climate zone B.

# Chapter 7

# Conclusions

This thesis aimed at testing the Offline Reinforcement Learning algorithm TD4-BC on the smart HVAC optimal control task, as well as testing if such minimalist approach could be successfully translated to a discrete-action domain.

At first, a set of tests was carried out on the LunarLander environment to establish if TD4-BC was competitive with its close relative TD3-BC. TD3-BC was deployed with three different datasets, each with a specific level of performance of the associated collection policy and with the best hyperparameters found by its authors. TD4-BC was tested with comparable datasets, but at the same time, nine different sets of hyperparameters were tried, with the objective of finding the set that yielded better performance across all three datasets. Furthermore, we tested and found that delaying policy updates, bring stabler updates and better quality agents. In the end, we found the two approaches to be comparable and we found TD4-BC to even outperforms TD3-BC with a dataset-specific hyperparameter selection.

During these experiments, we also observed some of the issues that TD4-BC presents. The first is the slow performance degradation, after the reach of its peak, at the growing of training steps. The second is the high variability between evaluation runs: this was inherited by the TD3-BC algorithm but became exacerbated in TD4-BC. We propose one motivation to this phenomenon, that is strictly related to the discrete-action nature of the algorithm. A discrete policy is just wrong or correct in a specific state, in general its action can't be just a little bit far from the optimal one. In addition, in an environment like LunarLander discrete, where the four available actions are completely different one from the other.

We then switched focus on the main objective of the thesis, the HVAC control. At first, we evaluated our approach, with the previously found best set of hyperparameter $\alpha$, learning rate and policy frequency updates, on the FQI dataset. Building on the findings on the LunarLander task, we investigated the role of the numbers of training steps with respect to the average return, finding that too

many training steps deteriorate our policy performance. Then we picked the best agent, and we compared it with respect to the reference behavioral FQI policy. We obtained comparable results, finding TD4-BC to be a good option for HVAC retrofitting, since it reaches similar performance, without the cost of both money and time of one year of online training that the behavioral policy required.

Furthermore, we tested two approaches to online fine-tuning for TD4-BC, a complete Naive online Fine-Tuning, where the regularization of the policy is removed, and a Conservative Fine-Tuning with the policy regularization still active. These techniques didn't make significant improvements or worsening to the TD4-BC policy trained offline.

Then, we moved to the PI-CONST dataset. This dataset contains almost no action exploration, resulting in a poor state-action space coverage. We evaluated TD4-BC on it, obtaining a worsening of the policy with respect to the behavioral one and observing indeed an high OOD actions overestimation. To help the algorithm in such challenging environment the two online fine-tuning were again tested. The naive approach, especially with higher starting exploration, worsen even more the offline agent, almost on all the periods and climate zones considered, probably victim of OOD samples overestimation coming from both offline and online data. Instead, Conservative Fine-Tuning proved to be useful for the offline agent trained on PI-CONST Data, making it beat the reference policy for the last six months of the testing period for climate zone E and making it overcome the baseline over the whole testing year for climate zone B. These results suggest that Offline Reinforcement Learning can be a useful approach to smart HVAC optimal control, especially when some requirements in terms of dataset composition are respected.

Overall, the minimalist offline approach considered in this thesis, that enables the possibility of online fine-tuning with no additional cost, revealed to be an interesting approach for the smart HVAC optimal control. Since the impact that number of training steps had on our experiments, we think that future works should focus on techniques of early stopping for offline training and in general of offline-evaluation. Moreover, the integration between offline training and online tuning should be further explored, specially for industrial tasks like HVAC control, where is not always easy to have big amount of various and representative data available and ready to be exploited. Approaches that explicitly take in to consideration the balance between offline and online data to control the trade-off between blind maximization and regularization, for example, could be tested in this task.

# Bibliography


[1] R. S. Sutton and A. G. Barto. «Reinforcement Learning An Introduction». In: Cambridge, MA: The MIT Press, 2018. Chap. 1 (cit. on pp. 3, 4, 8).

[2] C. J.C.H. Watkins and P. Dayan. «Q-Learning». In: *Machine Learning*. Vol. 8. Boston, MA: Kluwer Academic Publishers, 1992, pp. 279–292 (cit. on p. 10).

[3] D. Ernst, P. Geurts, and L. Wehenkel. «Tree-Based Batch Mode Reinforcement Learning». In: *Journal of Machine Learning Research*. Apr. 2005 (cit. on p. 11).

[4] G. J. Gordon. «Approximate Solutions to Markov Decision Processes». PhD thesis. Pittsburgh, PA, June 1999 (cit. on p. 11).

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016 (cit. on pp. 13, 15).

[6] T. M. Mitchell. *Machine Learning*. New York: McGraw-Hill, 1997 (cit. on pp. 14, 16).

[7] M. A. Nielsen. *Neural Networks and Deep Learning*. misc. 2018. URL: `http://neuralnetworksanddeeplearning.com/` (cit. on p. 17).

[8] V. Mnih et al. «Human-level control through deep reinforcement learning». In: *Nature* 518 (Feb. 2015), pp. 529–533 (cit. on p. 20).

[9] Hado van Hasselt, Arthur Guez, and David Silver. «Deep Reinforcement Learning with Double Q-Learning». In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30.1 (Mar. 2016) (cit. on pp. 20, 28).

[10] Scott Fujimoto, Herke van Hoof, and David Meger. «Addressing Function Approximation Error in Actor-Critic Methods». In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. PMLR, Oct. 2018, pp. 1587–1596 (cit. on p. 22).

[11] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. «Deterministic Policy Gradient Algorithms». In: *International Conference on Machine Learning*. 2014, pp. 387–395 (cit. on p. 22).

[12] S. Levine, A.l Kumar, G. Tucker, and J. Fu. «Offline Reinforcement Learning: Tutorial, Review and Perspectives on Open Problems». In: *arXiv pre-print* (Nov. 2020). arXiv: 005.01643 (cit. on pp. 25, 26, 29).

[13] Berkeley University of California. *Deep Reinforcement Learning Course CS285*. URL: https://rail.eecs.berkeley.edu/deeprlcourse/ (cit. on p. 27).

[14] Aviral Kumar, Justin Fu, Matthew Soh, George Tucker, and Sergey Levine. «Stabilizing Off-Policy Q-Learning via Bootstrapping Error Reduction». In: *Advances in Neural Information Processing Systems*. Vol. 32. 2019 (cit. on p. 28).

[15] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. «Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor». In: *Proceedings of the 35th International Conference on Machine Learning*. Vol. 80. PMLR, Oct. 2018, pp. 1861–1870 (cit. on p. 28).

[16] S. Fujimoto and S. S. Gu. «A Minimalist Approach to Offline Reinforcement Learning». In: *Advances in Neural Information Processing Systems*. Vol. 34. 2021 (cit. on pp. 29, 30, 32, 47).

[17] J. Fu, A. Kumar, O. Nachum, G. Tucker, and S. Levine. «D4RL: Datasets for Deep Data-Driven Reinforcement Learning». In: *arXiv pre-print* (Feb. 2021). arXiv: 2004.07219 (cit. on pp. 29, 39).

[18] A. Kumar, A. Zhou, G. Tucker, and S. Levine. «Conservative Q-Learning for Offline Reinforcement Learning». In: *arXiv pre-print* (Aug. 2020). arXiv: 2006.04779 (cit. on p. 30).

[19] I. Kostrikov, J. Tompson, R. Fergus, and O. Nachum. «Offline Reinforcement Learning with Fisher Divergence Critic Regularization». In: *arXiv pre-print* (June 2021). arXiv: 2103.08050 (cit. on p. 30).

[20] Petros Christodoulou. «Soft Actor-Critic for Discrete Action Settings». In: *arXiv pre-print* (2019). arXiv: 1910.07207 (cit. on p. 33).

[21] Seunghyun Lee, Younggyo Seo, Kimin Lee, Pieter Abbeel, and Jinwoo Shin. «Offline-to-Online Reinforcement Learning via Balanced Replay and Pessimistic Q-Ensemble». In: *5th Annual Conference on Robot Learning*. 2021 (cit. on p. 35).

[22] OpenAI. *A toolkit for developing and comparing reinforcement learning algorithms*. URL: https://gym.openai.com/envs/LunarLander-v2/ (cit. on p. 36).

[23] A. Mastropietro, F. Castiglione, S. Ballesio, and E. Fabrizio. «Reinforcement Learning Control Algorithm for HVAC Retrofitting: Application to a Supermarket Building Model by Dynamic Simulation». In: *16$^{th}$ IBPSA International Conference and Exhibition*. Rome, Italy, Sept. 2019 (cit. on pp. 41, 44, 45).