Controlling the estimation bias in deep reinforcement learning problems with sparse rewards

Towards robust robotic object manipulation learning

Roland Varga

Master of Science Thesis

Delft University of Technology

Delft Center for Systems and Control

Controlling the estimation bias in deep reinforcement learning problems with sparse rewards

Towards robust robotic object manipulation learning

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Roland Varga

January 18, 2023

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

The work in this thesis was supported by DEMCON. Their cooperation is hereby gratefully acknowledged.

Abstract

Many recent robot learning problems, real and simulated, were addressed using deep reinforcement learning. The developed policies can deal with high-dimensional, continuous state and action spaces, and can also incorporate machine-generated or human demonstration data. A great number of them depend on state-action value estimates, especially the ones in the actor-critic framework. Deriving unbiased estimates for these values is still an open research question, mostly since the connection between accurate value estimates and system performance is not yet well-understood. This thesis work has three main research contributions. Firstly, it analyzes the connection between value estimates and performance for the TD3 algorithm. Secondly, it derives theoretical bounds for the true value function when dealing with environments where a reward is only given for successful completion of a task (sparse/binary reward). Lastly, a deliberate underestimation objective is added to the TD3 algorithm together with the theoretical bounds to improve system performance when using human demonstration data that only covers a specific part of the state and action space. All the algorithms are tested and evaluated using simulated robot manipulation tasks in the robosuite environment, where the robot is first trained on the demonstration data and then can gather more experiences in the simulation. Results show that the deliberate underestimation together with the value bounds enable the robot to learn from human demonstration, which was not possible for the standard TD3. Additionally, applying just the value bounds speeds up the learning process when using machine-generated datasets.

Table of Contents

List of Figures

Acknowledgements

Firstly, I would like to thank my permanent supervisors Dimitris Boskos (TU Delft) and Michiel Plooij (Demcon) for their guidance and patience during the thesis project. I am also grateful to Arno Stienen (Demcon) for initiating my journey in this topic. Jens Kober (TU Delft) and his PhD students, Bas van der Heijden and Jelle Luijkx, helped with their insights at critical points during my thesis.

Jan van Beek (TU Delft) deserves my appreciation for the guidance in the tougher times of the thesis work.

I cannot be grateful enough for all the help and support from my partner, Emma, and my family. Last but not least, my friends helped me overcome numerous obstacles along the way and made my days brighter.

Delft, University of Technology
January 18, 2023

Roland Varga

“As I said, the problem is a classic one (the multi-armed bandit problem); it was formulated during the war, and efforts to solve it so sapped the energies and minds of Allied analysts that the suggestion was made that the problem be dropped over Germany, as the ultimate instrument of intellectual sabotage.”

— *Peter Whittle*

Chapter 1

Introduction

Robotics is spreading from the well-structured factory environments to our environments, even to our homes. This brings up numerous engineering challenges that need to be overcome to design and deploy systems that can operate safely and efficiently.

One of the most researched area in robotization is robotic manipulation [2, 3, 4, 5, 6, 7, 8, 9]. Seemingly easy tasks, such as picking up a plastic cup or a fruit are hard to achieve with robots using the standard tools. So what makes it challenging? For one, variability. The object properties are not necessarily known when designing the grasping systems and they often would be hard or tedious to estimate. For example, a tomato can be relatively solid when it just ripened but it might be a bit mushy as it further ripens. We humans often grab soft objects differently than solid ones. When picking up a tomato, we might reach under it, so we need to squeeze it less on the sides. Same with slippery objects, like a bar of soap. We learn our grasping behaviour via trail and error, dropping objects sometimes in the process. A similar approach exists in robotics: reinforcement learning [10, 11, 12, 2, 13].

In reinforcement learning the robot (agent) interacts with an environment receiving feedback from it in the form of rewards based on its performance. For example, if the goal is to pick up an object with a robot arm, the environment could give a reward of 1 whenever the object was lifted successfully and 0 otherwise. It is up to the robot to "figure out" a way to actually lift the object and thus achieving the reward of 1.

One can imagine, that if only this 0 or 1 reward is given and no other instructions to improve itself, then the robot will need quite some time to figure out anything useful [9]. Therefore it would be beneficial to use algorithms that can incorporate demonstrations of the execution of certain tasks, so the robot does not need to start from scratch. This demonstration might come from a human operator, as shown later in Chapter 4, guiding the robot, showing the task [8, 9, 5, 14].

The majority of state-of-the-art reinforcement learning algorithms in robotics, that can also utilize demonstration data, use deep neural networks (deep reinforcement learning – introduced in Chapter 2) to estimate the value of different states and actions. The "states" refer to the collection of variables that describe the current status of the system (e.g. sensor signals).

In case of the object lifting example, the state variable could be the distance of the robot gripper from the object. The "actions" gather all the possible choices that the robot could do in a given state, for example closing the gripper or moving closer to the object.

When the agent fails to accurately estimate the value of certain states and actions, the performance of these systems can quickly degrade. In our example, the robot might estimate that it is better (more "valuable") to stay far away from the object rather than picking it up. A great number of state-of-the-art algorithms are prone to consistently either overestimate or underestimate the value of states and actions (estimation bias – introduced in Chapter 3). This master thesis project aims at mitigating the estimation bias in certain algorithms in order to improve their robustness, reliability and performance while investigating the following research question.

Research question How to mitigate the effect of estimation bias in deep reinforcement learning algorithms that can utilize demonstration data, in order to improve their performance in robotic grasping?

1-1 Structure of the report

To provide a look into the state of the great field of deep reinforcement learning based robotics, there are three literature survey chapters.

Chapter 2 introduces the concept and tools of deep reinforcement learning through definitions and examples. As you will see, there are several algorithms to consider. However, the focus will be narrowed down to state-of-the-art policy gradient based algorithms that can learn from data, such as DDPG and TD3.

Chapter 3 provides a look into the reasons behind estimation bias and its effect on the performance of deep reinforcement learning agents. Next to the literature survey, an intuitive explanation for the possible reasons for both overestimation and underestimation bias is provided. The tools utilized by the TD3 algorithm to avoid overestimation are also presented.

Chapter 4, the last introductory chapter, presents how algorithms can incorporate demonstration datasets. Training on machine-generated datasets (e.g. data from other trained agents) is generally an easier task compared to human demonstrations. The difference between these two is also explained in this chapter. Different repositories for available datasets to test algorithms are also listed.

Chapter 5 describes the algorithm adjustments in order to potentially mitigate the effect of estimation bias. Additionally, an extra modification is proposed, that might help increasing the stability of algorithms when using human demonstration data. These are the main theoretical contributions of this thesis work.

Chapter 6 describes the simulation tools and datasets that are used for the implementation and testing of the algorithms, followed by Chapter 7 presenting the results. Chapter 8 discusses and interprets the results and Chapter 9 concludes the thesis.

Chapter 2

Deep Reinforcement Learning for Robotic Manipulation

2-1 Deep reinforcement learning in robotics

In the field of robot motion control, machine learning based approaches achieve better and better results. They can be used for tasks that would otherwise be hard or tedious to implement using the traditional approaches. For example, how should a door opening algorithm work? Following traditional approaches we could design a path-planner and a simple controller, that makes the robot follow the generated path. But how should this path be generated? What should happen if the door handle slips or the rotation is stiffer than expected? To lift the burden from the shoulders of the programmers, why not let the robot experiment and "figure out" what could be done in different scenarios? That is where reinforcement learning comes into the picture.

Furthermore, recent robotics research often uses high dimensional and multimodal sensor input signals, such as image data, lidar, force sensors, end-effector position and orientation. The development of standard controllers that can deal with such input signals is challenging, to say the least. For this reason, learning controllers became more and more popular for a number of application. For example, the development of robust robot manipulators can benefit from the fusion of these multimodal sensor signals via learning, and thus reinforcement learning is studied extensively in the recent years [10, 11, 12, 2, 13].

Sim2Real There are recently successful approaches to close the gap between simulation and the real world [4, 15]. Training policies (controllers) in simulation and deploying them in the real world is referred to as sim2real in the literature. In 2019, the OpenAI team built a real-world robot hand, mimicking a human one, and trained a policy purely in simulation, which after deployment manages to execute the desired moves on the real Rubik's cube [4]. In the same year, Hwangbo et al. at ETH Zurich published a paper about training a policy for a four-legged robot, again in simulation, to tackle different locomotion tasks and could deploy it on the real robot [15].

Choice of algorithm The previous results were not achieved using the same approach, in fact, there is not yet a universally best algorithm for all the problems. Researchers achieved promising results with algorithms including but not limited to the followings:

- Proximal Policy Optimization (PPO) [16]: published in 2017 by the OpenAI team
- Trust Region Policy Optimization (TRPO) [17]: this was used for the locomotion policy development in the ANYmal project [15, 18]
- Guided Policy Search (GPS) [19]
- Soft Actor Critic (SAC) [20]: Balakuntala et al. demonstrated successful learning of multimodal contact-rich skills on real robot [5]
- Deep Deterministic Policy Gradient (DDPG) [21]: Hansen et al. achieved promising results on robot manipulation benchmarks in simulation [3]
- Deep Deterministic Policy Gradient from Demonstration (DDPGfD) [8]: Luo et al. developed robust, multi-modal policies for industrial assembly tasks on a real robot [2]
- Twin-Delayed Deep Deterministic Policy Gradient (TD3) [22]
- Batch Deep Corrective Advice Communicated by Humans (BD-COACH) [6]: incorporating human corrective feedback in the learning

This list might look extensive, but the most successful algorithms in the recent years are from the so-called actor-critic framework. Therefore, in the following chapters the theoretical foundations are laid out for the following topics, where the consecutive ones build on or extends the previous ones:

1. Reinforcement learning
2. Value function based methods
3. Deep Q-learning
4. Policy gradient methods (a subset of the actor-critic algorithms)

2-2 Introduction to reinforcement learning

In reinforcement learning the controlled system is typically modeled as a fully or partially observable Markov Decision Process (MDP).

Definition 2-2.1 (Markov Decision Process)**.** A Markov decision process is defined by the 4-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R)$, where $\mathcal{S}$ denotes the set of possible states $s \in \mathcal{S}$, $\mathcal{A}$ is the set of possible discrete and/or continuous actions $a \in \mathcal{A}$, T is the probabilistic state transition function $T(s'|s,a)$, which describes the probability of arriving to the $s' \in \mathcal{S}$ state from $s \in \mathcal{S}$ in the next time step by taking action $a \in \mathcal{A}$. Finally, R describes the reward function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Some definitions include the distribution of the initial state d_0 and the scalar discount factor $\gamma \in (0, 1]$ in the tuple [23], because they are a common elements of MDPs (they will be used extensively).

One of the most important properties of MDPs is that the next state of the system s' only depends on its current state s and the chosen action a (Figure 2-1). This is also known as the **Markov-property**. This means that the current state s should incorporate all the effects of the previous states and actions. However, the state variables are often not measured directly, or the measurements are corrupted by measurement noise. That is why partially observed Markov decision processes become relevant.

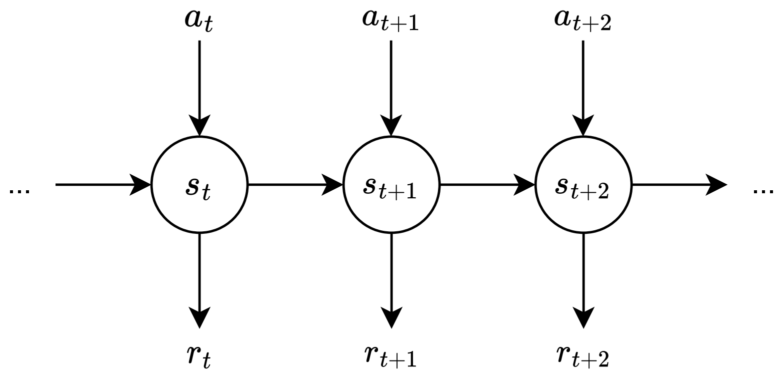


Figure 2-1: Diagram of a run with the Markov decision process. The next state only depends on the current state and the current action. The observed reward depends on the state and optionally the action. Note that the transition between states is not necessarily deterministic.

Definition 2-2.2 (Partially observed Markov decision processes)**.** A partially observed Markov decision process is defined by the 6-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R, \mathcal{O}, E)$, where $\mathcal{S}$, $\mathcal{A}$, T and R are the same as before, the set of observations is denoted by $\mathcal{O}$ and E is the emission function that describes the probability of observing $o \in \mathcal{O}$ given $s \in \mathcal{S}$.

Remark. The $E(o|s)$ emission function can be thought of as a measurement function, and the o measurement (observation) gives an indication of the value of the underlying s state.

Example Let us show a toy example for a reinforcement learning problem, where the previous (and future) notions can be demonstrated. So imagine a robot living in a 4×6 grid world, which is shown on Figure 2-2. In this case the $\mathcal{S}$ state-space is the set of possible, discrete x and y positions. The robot always starts at the bottom left corner, which is state (0,0), so the d_0 distribution of initial state has non-zero probability only at the (0,0) location. The goal of the robot is to reach the bottom right corner (5,0). The set of possible actions in $\mathcal{A}$ are moving one in the direction of a) x, b) y, c) $-x$ or d) $-y$. Our robot is a bit clumsy, so it does not necessarily move to the desired direction, which is captured by the probabilistic T transition function. So 90% of the time the robot makes a step in the direction requested by the action, but 10% of the time it ends up going to the any other 3 directions. On the boundary if the robot moves toward the boundary, it will just end up at the same place (state). Finally, the R reward function is designed such that the robot is "encouraged" the reach the goal:

- A reward of 10 when it steps on the goal
- A reward of -5 when going into the forest (stepping on the tree on Figure 2-2)

- A reward of -15 when meeting with the bear, which waits on the (3,0) location
- A reward of -1 after every action executed (to encourage the robot to finish as fast as possible)

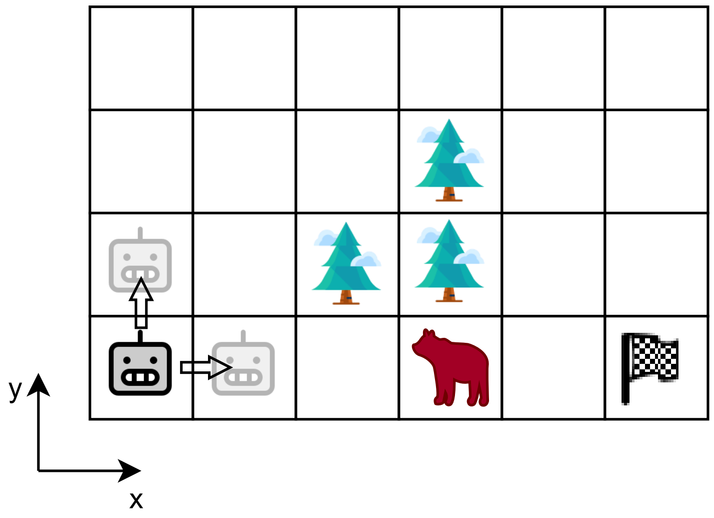


Figure 2-2: Toy example of a reinforcement learning problem. The agent (robot) starts at the bottom left field and its objective is to reach the goal (checkered flag). For the defined reward function, state and action-space, and transition dynamics refer to the main text.

In a reinforcement learning problem a so-called agent interacts with an environment, which is a Markov decision process (Figure 2-3). In the previous example, the agent would be embedded inside the robot and is responsible for strategic decisions. At every discrete, fixed time step the agent chooses an action a_t to execute in the environment, and observes an instantaneous r_t reward from the environment and the next observed state s_{t+1}.

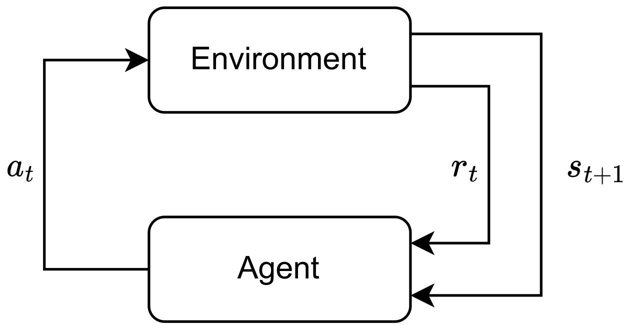


Figure 2-3: Diagram of a general reinforcement learning problem. The agent decides on an a_t action based on the last observed s_t state, which after execution results in r_t instantaneous reward and the new observed state s_{t+1}

.

Notations The control theory and the Reinforcement Learning (RL) literature often uses different notations for the same variables. In control theory the state variable is often denoted by "x" while in RL it is "s". RL works only in discrete time, and it is denoted by "t". Finally, a RL algorithms often only consider the current and the next time steps, so for convenience the time index is often omitted for the current time step (e.g. $a_t \to a$, $s_t \to s$, $r_t \to r$) and the next time step gets a prime (e.g. $s_{t+1} \to s'$). The previously mentioned notations are used extensively in this report.

In a reinforcement learning problem the goal of an agent is to learn a policy π (basically a controller), which is a distribution $\pi(a_t|s_t)$ of actions a_t conditioned on the observed state s_t. In case of a partially observed MDP, the actions are conditioned on the observation o_t instead of the "true" state s_t, so the distribution is denoted by $\pi(a_t|o_t)$. This means that in general we are dealing with a non-deterministic policy. In case of our robot example, the policy function would receive the current state (position on the grid) and would assign different probabilities for the possible actions (direction of movement).

The agent can do explorations in the environment by following the previously mentioned policy. A "trial run" using a policy is called an episode. An episode is essentially a trajectory τ, so a sequence of consecutive states and actions:

$$\tau = (s_0, a_0, s_1, a_1, \ldots, s_H, a_H)$$

The length of the episode is called the horizon and it is often denoted by H and it is possibly infinite (infinite-horizon problems). Since we are dealing with MDPs, the transition between states based on the chosen actions is characterised by the probabilistic transition function T. The policy function π, which determines the actions and the initial state distribution $d_0(s_0)$ are also probabilistic. This enables us to compute the probability of a certain observed trajectory τ:

$$p_\pi(\tau) = d_0\left(s_0\right) \prod_{t=0}^{H} \pi\left(a_t \mid s_t\right) T\left(s_{t+1} \mid s_t, a_t\right)$$

Intuitively, this equation is built upon the fact that the choice of action, the transitions and the initial state distribution all have independent probabilities. So the probability of observing a given initial state s_0 is exactly $d_0(s_0)$ and this is multiplied with the probability of choosing a_0 which is $\pi(a_0|s_0)$. Applying this action will result in the given state s_1 with probability $T\left(s_1 \mid s_0, a_0\right)$, which is also independent of the other probabilities, so it will become a multiplicative term. This goes on for every state, action and transition probabilities. But why do we care about the probability of certain trajectories? It is all about the accumulated reward that we can expect from a policy. During an episode the agent will receive rewards for the different state-action combinations, which can be summed up. So what makes a policy better than an other one? A common objective is the expected accumulated reward following the policy:

$$J(\pi) = \mathbb{E}_{\tau \sim p_\pi(\tau)}\left[\sum_{t=0}^{H} \gamma^t R\left(s_t, a_t\right)\right] \tag{2-1}$$

In the equation $\gamma \in (0, 1]$ is a discount factor, which help weighing future rewards. Usually γ has a value close to one, but its exact value can significantly influence the performance of long-horizon problems (problems which require long-term planning).

2-3 Value function based methods

Now that we showed the objective of general reinforcement learning problems in (2-1), the aim is to come up with a policy π, that will result in the highest expected reward. For this a more traditional approach in machine learning is the so-called value function based methods. Getting to know its mechanisms, advantages and shortcomings helps also understanding policy gradient methods, on which this thesis work builds on. So in this section ultimately the value function based Q-learning will be introduced, which is closely related the Deep Deterministic Policy Gradient (DDPG) and Twin-Delayed Deep Deterministic Policy Gradient (TD3) policy gradient algorithms (used later on). So let us see first what are value function based methods.

Formally, we could define the value of a certain state s following the policy π by the expected accumulated reward that is about to come when starting in that state:

$$V_\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R\left(s_{t+k+1}\right) \mid S_t = s\right]$$

where t is any time step, which means that the value of a state does not depend on the actual time index. This can only be assumed for infinite-horizon problems, that is why the k index in the sum tends to infinity. Note that in this definition the reward function R only depends on the state. In general it could also depend on the action taken, but for the thesis work the instantaneous rewards will only depend on the state. There is one problem though: in practice we usually do not know either the value function or the optimal policy (maximizing the reward), but we can estimate both from data.

At this point we can distinguish between potential algorithms, which will determine how we determine and update the value function and the policy:

- Level of interaction / source of data:
  - Offline: data is collected in advance
  - Online: data is collected by interacting with the process
- Type of experience sampling:
  - Off-policy: use experiences of different policies other than π to improve π
  - On-policy: use experience of π to improve π
- Model knowledge:
  - Model-free: the T transition and R reward functions are not known by the reinforcement learning agent, only the gathered transition data
  - Model-based: both T and R are known (e.g. dynamic programming)
  - Model-learning: T and R are estimated from the transition data

One of the most intuitive tools for solving value function based reinforcement learning problems is the Q-learning, which is an off-policy, online, model-free algorithm. It can be introduced using the toy example from the previous section. In a Q-learning approach the agent tries to estimate the general value of not just states but state-action pairs. In our example, the robot agent could store for each 2D state a numerical value for all four possible actions. This means that for the 6 possible x states, 4 y states and 4 possible actions in each state we need to store altogether $6 \times 4 \times 4 = 96$ values. The function that maps the states and action to a value is called the action-value function (Q-function / Q-table). It can be defined similarly to the state-value function, where an agent takes action a in the current time step and then follows policy π:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R\left(s_{t+k+1}\right) \mid S_t = s, A_t = a\right] \tag{2-2}$$

Storing these values is not an issue in this simple problem, but we can already see, that this number will increase fast as the number of state dimensions and possible actions increase. In any case, let us assume that the robot in our example has no knowledge about the location of the goal and about the obstacles along the way, so this Q-table is initialized with all 0s (initializing with non-zero values can help exploration but let us keep it simple).

Now the agent is free to experiment in the environment using its policy to gather (s, a, s', r) tuples that can be later used to correct the values in the Q-table. A common choice for an exploration policy in these simple problems is an ϵ-greedy policy, where there is an ϵ probability in every time step that the agent will choose a random, possible action, and otherwise it will exhaustively search for the highest valued action of all possible actions in the Q-table for the current state.

We have seen how the Q-table can be initialized and how it can be used to choose the actions to gather (s, a, s', r) tuples (experiences). However, the values in the Q-table need to be adjusted to improve the policy. So how can we tweak these values based on the agents experiences in the environment? The Bellman equation enables the derivation of algorithms that aid the improvement:

$$Q^{\pi}(s,a) = E\left[R(s') + \gamma Q^{\pi}(s', \pi(s')) \mid S_t = s, A_t = a\right]$$

where s' respects the transition dynamics. It essentially describes that the values in the a Q-table should be equal to the instantaneous reward and the discounted future rewards following policy π. Furthermore, since we would like to refine our policy and Q-table, such that the expected accumulated reward will be the highest, the optimal policy π^* and optimal Q-table Q^* are the following:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) = Q^{\pi^*}(s,a) \tag{2-3}$$

In case of a greedy policy (maximum exploitation of the Q-table values):

$$\pi^*(s) = \arg\max_{a} Q^*(s,a) \tag{2-4}$$

Putting Equation (2-2), (2-3) and (2-4) together results in the Bellman optimality principle:

$$Q^*(s,a) = \mathbb{E}\left[R(s') + \gamma \max_{a'} Q^*(s',a') \mid S_t = s, A_t = a\right]$$

Now we know how the optimal Q-function (Q-table in case of discrete states and actions) would look like but it is still not clear how to estimate it from data. The final 2 pieces of this puzzle are provided by the bootstrapping method and the usage of temporal difference (TD) error. Bootstrapping in this context refers to the update of a variable using the values of its successor states. So in case of the Q-function we can use the previous Bellman optimality principle and turn it into an iterative update using the (s, a, s', r) transition samples, that the agent gathers throughout the environmental interactions:

$$Q^\pi(s,a) \leftarrow R(s',a) + \gamma \max_{a'} Q^\pi(s',a')$$

In theory, this kind of update would work, but in practice numerical instability can be avoided using a learning rate α:

$$Q^\pi(s,a) \leftarrow (1-\alpha)Q^\pi(s,a) + \alpha \left[R(s',a) + \gamma \max_{a'} Q^\pi(s',a')\right] \tag{2-5}$$

$$Q^\pi(s,a) \leftarrow Q^\pi(s,a) + \alpha \underbrace{\left[R(s',a) + \gamma \max_{a'} Q^\pi(s',a') - Q^\pi(s,a)\right]}_{\Delta:=\text{temporal difference}} \tag{2-6}$$

Note that the temporal difference Δ can be extracted from the previous update:

Summary of Q-learning: Q-learning belongs to the family of value function based reinforcement learning algorithms. In addition to the state value function $V(s)$ it estimates the action value function $Q(s,a)$ ($V^*(s) = \max_a Q^*(s,a)$). The algorithm works as follows:

Algorithm 1 Q-learning

```
Initialize $\mathbf{Q}(\mathbf{s},\mathbf{a})$ arbitrarily
**ep** $\leftarrow 0$ (index of current episode)
**while** (**ep** $<$ Max number of episodes) **do**
    $\mathbf{s} \leftarrow$ sample $d_0(s_0)$
    $\mathbf{t} \leftarrow 0$
    **while** ($\mathbf{t} <$ Length of episode) **do**
        Choose action $\mathbf{a}$ based on policy derived from $\mathbf{Q}(\mathbf{s},\mathbf{a})$ (e.g. $\epsilon$-greedy)
        Take action $\mathbf{a}$ and observe $\mathbf{r}$ and $\mathbf{s}'$
        $\mathbf{Q}(\mathbf{s},\mathbf{a}) \leftarrow \mathbf{Q}(\mathbf{s},\mathbf{a}) + \alpha \left[\mathbf{r} + \gamma \max_{\mathbf{a}'} \mathbf{Q}(\mathbf{s}',\mathbf{a}') - \mathbf{Q}(\mathbf{s},\mathbf{a})\right]$
        $\mathbf{s} \leftarrow \mathbf{s}'$
        $\mathbf{t} \leftarrow \mathbf{t} + 1$
    **end while**
    **ep** $\leftarrow$ **ep** $+ 1$
**end while**
```

Q-learning has the benefit that it can use samples generated by any policy, since it is an off-policy algorithm. However, there are some challenges/shortcomings that are tackled by more advanced algorithms:

- The system either needs to have only discrete states, or the continuous ones need to be discretized.
- It can only deal with discrete action spaces, where to determine the action a that maximizes $Q(s,a)$ needs to be exhaustively searched.

- The computational complexity of searching the action space grows exponentially as the action space dimension increases. In practice, due to this limitation and the previous ones, problems with high dimensional state and action spaces can quickly become intractable.

2-4 Deep Q-learning

As mentioned in the previous section, Q-learning can only deal with systems that have discrete state and action spaces. So systems with continuous states and actions needs to be discretized, which is not straightforward, and can cause all sorts of problems and undesired behaviour later on. To avoid it, deep learning might be the answer. The 2015 success of the company DeepMind with Atari games demonstrated the potential of Deep Q-learning Network (DQN) approaches to achieve human-level (or even better) control performances [24]. But how do neural networks can boost the performance of Q-learning agents?

First of all, in standard Q-learning a table needs to be stored with values for all possible combinations of states and actions (Figure 2-4). In problems with high-dimensional state- and/or action-spaces, such as robotics problems, the values can be difficult to store and deal with. So the idea of DQN is to approximate the $Q(s, a)$ function with a neural network, which requires less parameters to store and might provide generalization capabilities for states that the agent has not explored before.

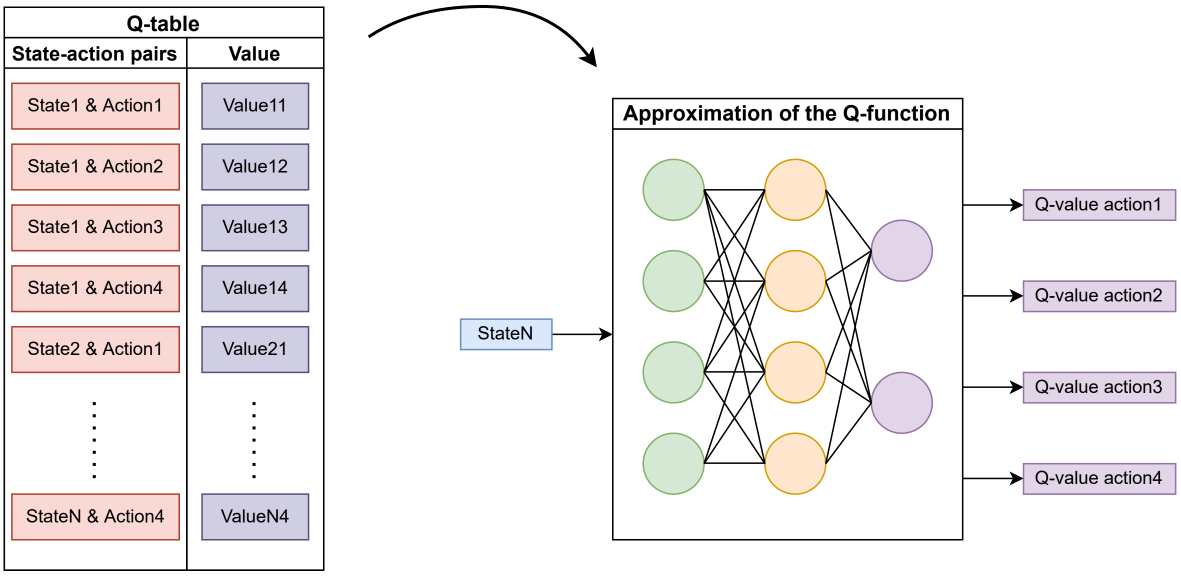


Figure 2-4: In Q-learning the state-action pairs and their corresponding values are stored one-by-one. This means that only discrete states can be considered. Deep Q-learning enables the generalization to continuous state-spaces. In this example, the agent in each N state can execute 4 discrete actions. Instead of storing these pairs directly, a deep neural network can be fitted on the states.

This basically becomes a regression problem. Let us take another look at (2-5):

$$Q^\pi(s,a) \leftarrow Q^\pi(s,a) + \alpha\Big[\underbrace{R(s',a) + \gamma \max_{a'} Q^\pi(s',a')}_{\text{target } Q \text{ value}} - Q^\pi(s,a)\Big]$$

We can see that via the optimization the goal is to shape the Q function more like this target. Also an important distinction between Q-learning and DQN is that now we are not adjusting individual values in the Q-table for a specific state-action pair, but rather we need to adjust the parameters of the network that approximates the Q-function (Figure 2-4). So let us denote the parameterized Q-function by Q_ϕ, where ϕ denotes the network parameters. The loss used for backpropagation can be the following:

$$J(\phi) = \mathbb{E}\left[\left(R(s',a) + \gamma \max_{a'} Q^\pi_\phi(s',a') - Q^\pi_\phi(s,a)\right)^2\right]$$

It is important to highlight, that we can approximate this expectation from data, and then the only variable in this objective will be the ϕ network parameter:

$$J(\phi) \approx \frac{1}{n_B} \sum_{(s,a,s',r)\in\mathcal{B}} \left[\left(r + \gamma \max_{a'} Q^\pi_\phi(s',a') - Q^\pi_\phi(s,a)\right)^2\right]$$

The (s,a,s',r) tuples are the experiences of the agent that are sampled from the so-called replay buffer in $\mathcal{B}$ mini-batches with n_B samples. The replay buffer is a storage for the selected previous experiences and the mini-batch is a subset of that. Using replay buffer is actually necessary when dealing with neural networks. These are global function approximators and thus previous experiences need to be stored and replayed later, otherwise the agent will "forget" them. So a common practice is to sample $\mathcal{B}$ mini-batches from the replay buffer and use stochastic gradient descent (SGD) to update ϕ:

$$\phi_{k+1} = \phi_k - \alpha \nabla_{\phi_k} J(\phi_k)$$

This update looks straightforward, however, there is a problem. The previously mentioned target value changes with every iteration, since the Q-function changes. This means that with the gradient steps the algorithm chases a non-stationary value. To mitigate this problem, a second neural network, a target network can be used, parameterized by ϕ_{targ}. This network has the same structure and parameters, as the Q-function approximator, but its update is delayed with a given number of steps. To avoid confusion it is important to highlight, that the target network does not approximate the whole $r + \gamma \max_{a'} Q^\pi_\phi(s',a')$ term, only the Q-function in this target. To summarize the idea:

1. Both the Q-function approximator Q_ϕ and the target network $Q_{\phi_{\text{targ}}}$ are initialized with the same parameters: $\phi = \phi_{\text{targ}}$.
2. While the target network remains constant, the parameter ϕ is updated.
3. After predefined number of updates to ϕ, these parameters are copied over to update the target network: $\phi_{\text{targ}} \leftarrow \phi$.
4. Steps 2-3 are repeated till convergence.

So the approximation of the objective function is the following:

$$J(\phi_k, \phi_{\text{targ},k}) \approx \frac{1}{n_B} \sum_{(s,a,s',r)\in\mathcal{B}} \left[\left(r + \gamma \max_{a'} Q^{\pi}_{\phi_{\text{targ},k}}(s', a') - Q^{\pi}_{\phi_k}(s,a) \right)^2 \right]$$

$$\phi_{\text{targ},k} \leftarrow \phi_k \text{ after given steps, otherwise } \phi_{\text{targ},k} \text{ remains constant}$$

2-5 Deterministic policy gradient methods, DDPG

DQN is already an improvement compared to Q-learning, as it can deal with continuous state-spaces. However, most physical systems, such as robots, are controlled by continuous actions. For example, we might want to control the joint torques in a robot without manually discretizing the range of possible torque values.

Policy gradient methods provide a way to naturally deal with continuous action spaces. Since the thesis work will focus on deterministic policy gradient based, off-policy, actor-critic algorithms, the theoretical background and motivations are restricted to those in the following. So the concept of deterministic policy gradient will be introduced via the DDPG algorithm developed by Lillicrap et al. in 2015 [21].

So let us consider a deterministic policy $\mu_\theta : \mathcal{S} \rightarrow \mathcal{A}$, where θ denotes the parameter vector of the μ policy (in case of neural network parametrization this contains the weights and biases). The DQN algorithm used a greedy policy, so $\mu(s) = \arg\max_a Q_\phi(s,a)$, which is dependent on the action-value function. However, if we represent the policy and action-value functions separately, we arrive to the actor-critic algorithms. The policy is called the *actor*, since it determines the action based on the current state, while the Q action-value function, the *critic*, gives an indication on how valuable is the chosen action in the current state (Figure 2-5).

As mentioned previously, the focus is now restricted to off-policy algorithms, as they can derive μ_θ from the data collected by any other behaviour policy $\beta(s)$. Let us also denote by $p(s \rightarrow s', t, \beta)$ the density at state s' after transitioning from state s via t time steps using the behaviour policy β. Finally, the discounted state distribution can be expressed as $\rho^\beta(s') := \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} d_0(s) p(s \rightarrow s', t, \beta)\, \mathrm{d}s$, similar to [25]. Using this notation, the performance of a deterministic policy μ_θ can be expressed as:

$$J_\beta(\mu_\theta) = \int_{\mathcal{S}} \rho^\beta(s) Q^{\mu_\theta}(s, \mu_\theta(s))\, \mathrm{d}s$$

Silver et al. [25] derived the derivative of this objective function w.r.t. the hyperparameter θ as:

$$\nabla_\theta J_\beta(\mu_\theta) = \mathbb{E}_{s\sim\rho^\beta} \left[\nabla_a Q^{\mu_\theta}(s,a) \mid_{s=s_t, a=\mu_\theta(s_t)} \nabla_\theta \mu_\theta(s) \mid_{s=s_t} \right]$$

The update of the θ parameters can be implemented using this gradient. The DDPG algorithm extends this deterministic policy update by approximating the Q action-value function similarly to the DQN algorithm (it was actually motivated by the success of DQN). So after similarly approximating the Q^{μ_θ} by $Q^{\mu_\theta}_\phi$ we get:

$$\nabla_\theta J_\beta(\mu_\theta) = \mathbb{E}_{s\sim\rho^\beta} \left[\nabla_a Q^{\mu_\theta}_\phi(s,a) \mid_{s=s_t, a=\mu_\theta(s_t)} \nabla_\theta \mu_\theta(s) \mid_{s=s_t} \right]$$

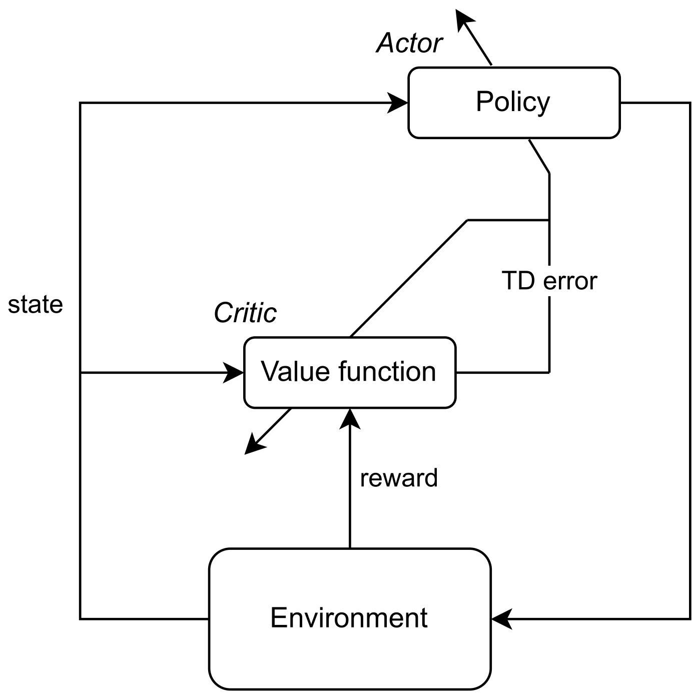


Figure 2-5: Schematics of the actor-critic framework. The actor chooses and executes an action in the environment based on the current state. The critic observes the next state and the reward, then computes the Temporal difference (TD) error. Based on this error both the actor and critic are adjusted [1].

Unfortunately, by approximating the action-value function we lose the guarantee that following the gradient the θ parameter will converge. In any case, we can give a Monte Carlo estimate of the previous expectation based on the collected data [21]:

$$\nabla_\theta J_\beta(\mu_\theta) \approx \frac{1}{n_B} \sum_i^{n_B} \nabla_a Q_\phi^{\mu_\theta}(s,a) \mid_{s=s_i, a=\mu_\theta(s_i)} \nabla_\theta \mu_\theta(s) \mid_{s=s_i}$$

where n_B is the mini-batch size, as before, and the (s_i, a_i, s'_i, r_i) tuples are randomly selected, independent samples from the replay buffer in the mini-batch. Now we have a gradient to update the θ policy network parameters, and the ϕ parameters can be updated as described earlier for the DQN. The creators of DDPG proposed to use target networks for both the policy and the action-value function, similarly to DQN. However, the target networks do not get the parameters from the other networks directly, but via polyak-averaging using the φ polyak coefficient (hyperparameter):

$$\begin{aligned}\theta_{\text{targ},k} &\leftarrow \varphi\theta_{\text{targ},k} + (1-\varphi)\theta_k \text{ , when there is update}\\ \phi_{\text{targ},k} &\leftarrow \varphi\phi_{\text{targ},k} + (1-\varphi)\phi_k \text{ , when there is update}\end{aligned}$$

The interested reader can find the pseudo-code of the DDPG algorithm in the paper published by the creators [21]. The thesis project is built upon the TD3 algorithm [26], which is closely related to DDPG. It will only be introduced in the next chapter after providing additional background knowledge.

2-6 Summary of the chapter

Reinforcement learning is a data-driven way of addressing Markov Decision Process (MDP) problems. Value function based methods assign general values to the points of the state-space, and the best action is chosen such that it maximizes the expected return. Although it is possible to extend value function methods to continuous state-spaces, only discrete actions can be considered.

Actor-critic methods decouple the selection of the action (actor) from the value function (critic), and this enables the use of continuous action spaces. One of the most successful actor-critic algorithms are policy gradient methods. In this chapter, the update equations of the deterministic policy gradient algorithms were derived. From this group, the thesis work is built upon the TD3 algorithm, an improved version of DDPG, that will be discussed still in more details in the next chapter, after the introduction of estimation bias.

Chapter 3

Estimation Bias of Actor Critic Algorithms

In Q-learning the phenomenon of learning unrealistically high values for certain actions and states, the so-called overestimation bias, has been studied in the last 2-3 decades. This chapter focuses on the sources and effects of overestimation bias, also in actor-critic algorithms. Furthermore, as part of a current research area, the characteristics and effects of underestimation bias are also discussed, which are among the main focus points of this thesis project.

3-1 Sources of overestimation bias

Thrun et al. in their paper published in 1993 [27] attributed the overestimation bias in Q-learning to insufficiently flexible function approximation. In 2010, van Hasselt showed that noise in either the rewards or the state transitions can also result in overestimation [28]. Unifying these two views, in 2016 van Hasselt et al. proposed the deep double Q-learning algorithm [29], which is effective in avoiding overestimation bias in deep reinforcement learning problems. But can we get an intuition of the reason for overestimation bias?

Van Hasselt et al. discuss this in much more detail in their paper [29], but let us highlight a simple case when overestimation might occur. In Figure 3-1 a dummy Q-learning problem is shown. In this example, the blue, "true value" function, $Q(s, a)$, only depends on the single state, so in a certain state the instantaneous reward and discounted future reward does not depend on the action (assumption only necessary for the visualization). A couple points from the value function are sampled for the approximation with two polynomial functions, which represents the $Q_\phi(s, a)$ for two possible actions, a_1 and a_2. It can be seen that it is a clear case of overfitting, as both polynomials fit the sampled points really well (denoted by X), but in other regions the approximation is unsatisfying (the two polynomials are fitted of different set of samples). Now remember, that the target value in the Q-learning contained a maximization step over all possible actions:

$$\text{target} = r + \gamma \max_a Q(s', a)$$

and we wanted to make the current Q-function more like this target. It is crucial to highlight, that as we approximate the Q-function, either with a polynomial or a neural network, in certain regions of the state-space in our simple example (Figure 3-1) the values corresponding to a certain actions will sometimes underestimate, sometimes overestimate the true value function, but generally this estimation error for the two possible actions will be different. For example, at state -2.5 taking action 1 has a value close to 1, while action 2 has a value below -2, so one is overestimating and the other one is underestimating the value of those actions. Now due to the maximization step in the target network update we will chose the action with the overestimated value. Even if the value for both actions underestimate the true value, we will choose the higher one.

Overestimation bias in discrete action setting Putting it all together, the approximation errors in the value estimates of certain actions will sometimes result in underestimation, sometimes overestimation, independently of each other. However, due to the maximization step in the target network update, we tend to select the overestimated values over underestimated or even exact values. This is the intuitive reason for the overestimation bias. In a single update step a small overestimation error has little effect, but the error can propagate through the whole network because of the bootstrapping. For a more exact, mathematical formulation deriving the overestimation bias (even in case of underfitting) the paper by van Hasselt et al. [29] is highly recommended.

Overestimation bias in actor-critic framework So far the intuitive reasoning behind overestimation in value function based methods were presented, but does that problem persists in actor-critic settings? A main difference is that in this case instead of the maximization over the discrete actions, which was the root of the overestimation bias, a gradient is used for the parameter update. Fujimoto et al. proved that overestimation is also expected in deterministic policy gradient based actor-critic algorithms, such as Deep Deterministic Policy Gradient (DDPG) [26].

3-2 Effects and reduction of overestimation bias

Overestimating the value of every state uniformly has little-to-no detrimental effect on the learnt policy. In fact, initializing the value of all states with the same positive value is a common practice in tabular Q-learning, as it can help exploration. It can be thought of as an optimism in the face of uncertainty [28].

However, when the overestimation bias is not uniform, so different state values are overestimated by varying amounts, it can harm the policy learning. Van Hasselt et al. demonstrated in six atari games how the reduction of overestimation bias via deep double Q-learning can improve the performance in discrete action problems [29]. In actor-critic domain (continuous action-space), Fujimoto et al. showed the benefits of reducing overestimation bias of the DDPG algorithm, and thus achieving state-of-the-art performance in different MuJoCo environments, outperforming even the soft actor critic algorithm. So what modifications did they make to the original DDPG algorithm for this?

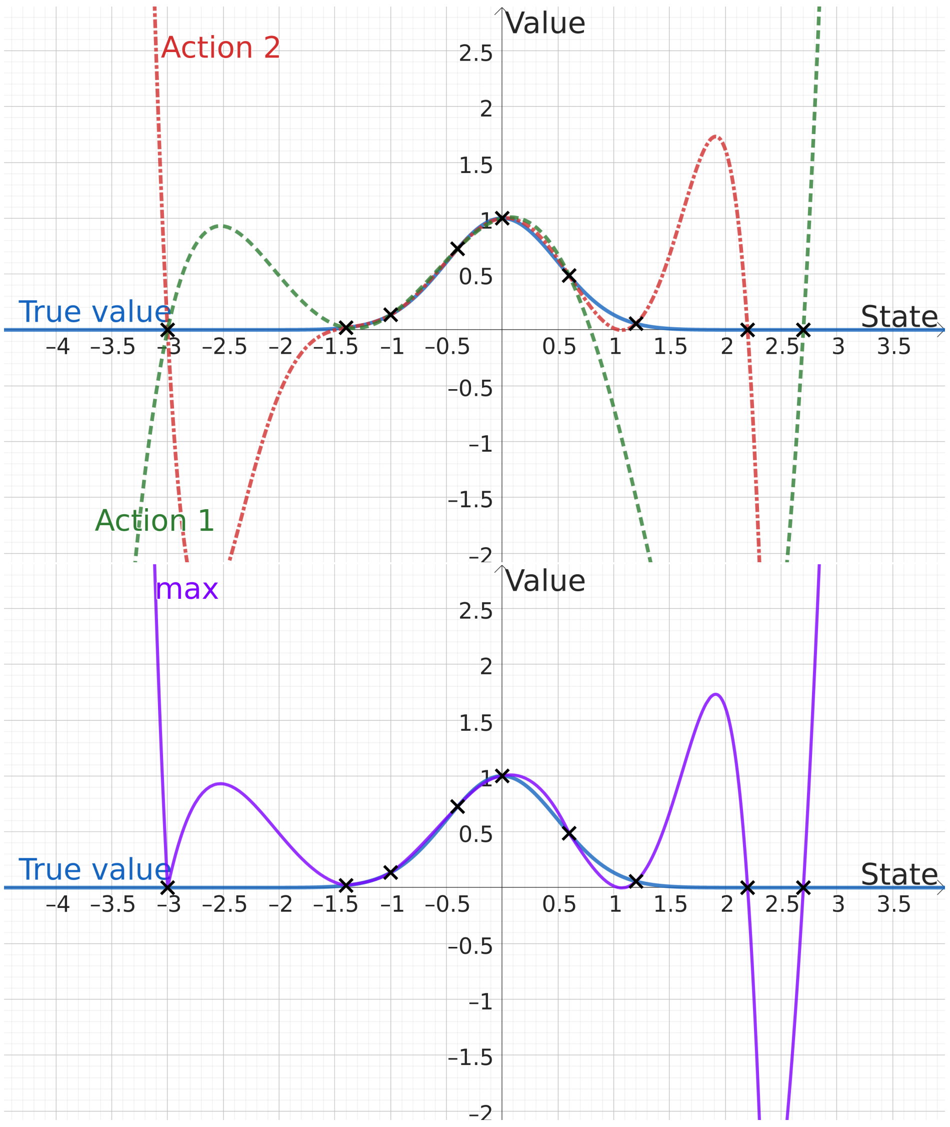


Figure 3-1: Visualization of overestimation bias introduced by the maximization operator in Q-learning. The true value function has a single state input and its value output is the same for all possible actions in a certain state for simplicity. The crosses indicate the samples of the true value function that are used to create the approximations for the state-action pairs. Due to approximation errors, the values for different actions (green and red curves on the top graph) sometimes overestimate, sometimes underestimate the true function. However, the maximization between the two will result in selecting overestimated values more often over underestimated ones (bottom graph).

To address the issue of overestimation, Fujimoto et al. [26] proposed the Twin-Delayed Deep Deterministic Policy Gradient (TD3) algorithm, which applies a couple of modifications to DDPG (or any actor-critic), that help reducing this bias:

1. Delayed policy updates: updating the critic more often than the actor. This helps avoiding that the actor exploits errors in the value estimated by the critic. Otherwise the actor could overfit to a mistakenly overestimated value.
2. Target policy smoothing: a noise term is added to the target to avoid overfitting on specific actions (in this case Gaussian noise with saturation). This also generally helps developing more robust policies [26].

$$\text{target} = r + \gamma Q_{\phi_{\text{target}}}(s', \mu_{\theta_{\text{target}}}(s') + \varepsilon), \text{ where } \varepsilon = \text{clip}\left(\mathcal{N}(0,\sigma), -c, c\right)$$

3. Clipped double Q-learning for actor-critic: maintaining two value functions which are used to update the other one (in the following equation this feature is already combined with the previous one)

$$\text{target} = r + \gamma \min_{i=1,2} Q_{\phi_{\text{target,i}}}(s', \mu_{\theta_{\text{target},1}}(s') + \varepsilon), \text{ where } \varepsilon = \text{clip}\left(\mathcal{N}(0,\sigma), -c, c\right) \quad (3\text{-}1)$$

Together these three modifications reduce the overestimation bias and avoid overfitting to specific actions, resulting in more stable learning and higher performance. However, it avoids overestimation by introducing underestimation bias. The creators of TD3 claim that it is much more favorable over overestimation bias, but this still harms the potential performance.

3-3 Dealing with underestimation bias

Although underestimated values will not propagate through the whole network like the overestimated ones [26], Wu et al. showed theoretically and empirically that it still negatively effects the performance in practical applications [30].

The main source of underestimation bias in the previously mentioned TD3 algorithm is the minimization between two critics, which is shown in Eq. (3-1). Wu et al. proved that this bias becomes even worse as the number of critics increase [30]. Also instead of taking the minimum, averaging the value of several critics results in overestimation (although lower variance in the estimates). So if increasing the number of critics does not directly help, what can be done?

Wu et al. proposed the triplet-averaged deep deterministic policy gradient (TADD) algorithm, which weighs the estimates of multiple critics, where some have overestimation bias and some have underestimates bias [30]. Although they proved that under mild assumptions, the right weighing constant parameter ($\beta \in (0, 1)$) will result in unbiased estimates, its value is dependent on the application and could only be found via trial end error.

Addressing the same issue, Wei et al. [31] recently proposed the Quasi-Median Delayed Deep Deterministic Policy Gradient (QMD3) algorithm, that selects the value from an ensemble of n critics $(Q_{(1)}, Q_{(2)}, \ldots, Q_{(n)})$ using the quasi-median operator. If the value estimates are sorted in increasing order, the quasi-median operator selects the value estimate at index

$\lfloor n/2 \rfloor$. So for even number of critics, it will be the value that is the lower from the two middle values and the value that is one lower from the median in case of odd number of value estimates.

Although the mentioned publications suggest that the reduction of underestimation bias can result in policies with higher performance, the exact effects and nature of underestimation bias is still an active research area with many publications [32, 33, 34, 35].

Chapter 4

Learning from Demonstration

It is desirable to have learning algorithms, which you can just show the task a couple of times and then it can also replicate the behaviour. These would require relatively short time to set up and get them running. Can we use such algorithms in deep reinforcement learning problems to replace carefully shaped rewards, which requires a lot of engineering effort and often result in suboptimal solutions?

4-1 Replacing shaped rewards with demonstrations

Shaped rewards can be replaced by demonstration data [8, 9, 5, 14]. For example, Vecerik et al. from Deepmind [8] proposed the Deep Deterministic Policy Gradient from Demonstration (DDPGfD) algorithm to be able to use the DDPG algorithm with sparse reward environments using demonstration data. They showed both in simulation and on a real robot that the DDPGfD algorithm using demonstrations and sparse reward could outperform DDPG with carefully shaped rewards. The performed task were industry assembly exercises, such as hard drive and cable insertion, and the agent could gather more experiences from the environment, not just the demonstration data (it will be important when comparing to "offline" reinforcement learning algorithms). The key components of their algorithm were the following:

- Adding demonstration to the replay buffer
- Prioritized experience replay for both the demonstration and the newly gathered data
- Using a mix of one-step and n-step return for training, to enable faster propagation of values in the critic
- Making multiple gradient steps per environmental step
- L2 normalization on the critic and the actor network weights

- Kinesthetically controlled robot to gather human demonstration data

Using demonstration data generated by human operators can introduce different, non-trivial challenges which requires special attention and is a focus of several research groups.

4-2 Importance of the source of demonstration data

Different sources of demonstration data and their utility in offline reinforcement learning is an active research area. "Offline" reinforcement learning refers to a framework where the policy needs to be developed using only the demonstration data, so without any live interaction of the agent with the environment.

The creators of the robomimic framework (used for the implementation of this thesis) identified the following challenges when learning from demonstration data [7]:

- Data from Non-Markovian Decision Process: human demonstration substantially differs from machine-generated one. The decisions made by human operators are affected by several factors, such as teleoperation device, past experiences and past actions taken. On the other hand, data generated by trained deep reinforcement learning agents are Markovian.
- Variance in Demonstration Quality from Multiple Humans: demonstration data collected from several participants has a diverse set of strategies, which vary in efficiency and general quality [36, 37]
- Dependence on dataset size: in offline reinforcement learning the state and action space coverage is crucial, as the agent has no opportunity to explore. More data generally results in better coverage of both spaces and thus mostly enable better policies.
- Mismatch between training and evaluation objective: the training objective (e.g. minimizing Q loss) only indirectly effects the evaluation performance (e.g. success rate). This point is also true for online reinforcement learning, such as TD3.
- Sensitivity to hyperparameter choices: studies in offline RL on machine generated datasets suggest that they are often extremely sensitive to hyperparameter choices [38, 39]

From this list of challenges the non-Markovian nature of the human demonstration data can be quite problematic. In deep reinforcement learning the agent is basically trying to optimize a Markov decision processes. So how can one still use non-Markovian demonstration data for learning?

Mandlekar et al. [36] created an algorithm called IRIS, which incorporated temporal abstraction (memory) in the form of recurrent neural network (RNN) when learning from human demonstration and showed promising results in standard benchmarking environments. Similarly, behavioral cloning [40] with an RNN-policy (BC-RNN) and hierarchical behavioral cloning (HBC) [41] are quite successful in utilizing temporal abstraction.

Interestingly, state-of-the-art batch (offline) reinforcement learning algorithms such as batch-constrained Q-learning (BCQ) [42] and conservative Q-learning (CQL) [33] perform very well on machine-generated datasets, but fail to accomplish the task when human demonstration is provided [7]. This highlights the importance of testing on human datasets.

4-3 Available datasets for learning

There are several available datasets and environments to test novel algorithms. The RoboTurk project developed by the Stanford Vision and Learning Lab (SVL) enables users to remotely operate either real or simulated robot environments to gather robotic manipulation human demonstrations using only a web browser and a phone [37]. The datasets from this project is used in this thesis work.

DeepMind also released several datasets for different environments, such as the DM Control Suite, DM Locomotion, the popular Atari 2600 arcade learning environment and even real robot [38]. Although it provides access to diverse problems in terms of action space, observation space, difficulty of exploration, action delay etc., the "only" data available is generated by other trained agents.

The Farama Foundation released datasets generated either by human operators, non-markovian algorithms or partially trained DRL agents [43]. The tasks include 2D navigation, locomotion, MuJoCo Gym environments, Androit object manipulation etc. By the time of writing this report all the environments except the PyBullet and Flow ones are still maintained.

Chapter 5

Algorithm Proposal

This thesis work has two main ideas for algorithm improvement:

1. Applying upper and lower bounds on the value function target values
2. Intentional underestimation of the value function around the demonstration data distribution for improved exploration

In the following sections the motivations, theoretical backgrounds and implementation details behind these modifications are discussed. Let's look at the value function bounds first!

5-1 Lower and upper bounds on the value functions

The algorithms presented in Chapter 3 aimed at either avoiding overestimation bias by deliberate underestimation or reducing the bias itself, often by the introduction of additional critics (neural networks). These extra neural networks, however, increase the computational requirements of the algorithms.

In this thesis work a different approach is proposed, which is based on the insight into the possible, true critic values when using sparse rewards. As discussed in Chapter 4 using sparse rewards over shaped one in reinforcement learning problem is preferred, since these are easier to design and implement, suffer less from local minima. Also the increased difficulty in exploration can be overcome by demonstration data.

The sparse reward function used from now on is as follows:

$$R(s', a) = \begin{cases} 1, & \text{if } s' \text{ is a terminal state} \\ 0, & \text{otherwise} \end{cases}$$

A terminal state is a state, which when reached by the agent, the current episode would normally terminate. Now let's take another look at the bootstrapping step that is used to update the target values in general actor-critic algorithms:

$$\text{target} = \begin{cases} R(s', a), & \text{if } s' \text{ is a terminal state} \\ R(s', a) + \gamma Q(s', \mu(s')), & \text{otherwise} \end{cases} \tag{5-1}$$

These target values are used to update the current critic estimates, by making them more like this target (MSE loss). This means that the target values could already give an indication if underestimation or overestimation happens. If there would be access to the optimal, "true underlying value function", then we could determine the amount of estimation bias in the current estimates.

It is easy to recognize, that based on (5-1), when using sparse rewards the optimal target value of a state-action pair with which a terminal state is reached in a single step is 1, independently of the current state (whether it is terminal or not).

$$\text{target} = 1, \text{ for terminal } s'$$

Remark. If the chosen action in the previously mentioned state is not optimal and it does not bring the system to a terminal state, then the target value for that specific state-action pair is not necessarily equal to 1.

Lemma 5-1.1. In actor-critic framework with sparse rewards, the optimal target values of any state-action pair is upper bounded by 1 for $\gamma \in (0, 1)$ discount factor.

Proof. Since we are using sparse rewards, the instantaneous reward of transitioning to non-terminal s' state from state s using action a is 0. This means that the target value for these transitions will only be determined by the discounted future rewards.

$$\text{target} = \gamma Q(s', \mu(s')), \text{ for non-terminal } s'$$

A reward of 1 is only given for reaching a terminal state, which means that the maximum cumulative reward that can be gathered is 1. Since $\gamma \in (0, 1)$, this means that the discounted future reward is upper bounded by 1 independently of the state or action.

$$Q^*(s, a) \leq 1, \quad \forall s, a$$

□

Because of the either positive or zero instantaneous reward, one might suspect that the optimal state-action value function does also not take negative values. This is idea is captured and proved in the following lemma.

Lemma 5-1.2. In actor-critic framework with sparse rewards, the optimal target values of any state-action pair is lower bounded by 0 for $\gamma \in (0, 1)$ discount factor.

Proof. Similarly to the previous proof, it can be used that the target values of terminal states independent of the action is 1, which is non-negative. Also similarly to the previous case, for non-terminal states the optimal value of a state-action pair only has contribution from the expected discounted future reward. Since $\gamma \in (0, 1)$ and the only possible future reward is either 0 or 1, the expected discounted future reward cannot be negative.

$$Q^*(s, a) \geq 0, \quad \forall s, a$$

□

Sticking together these two lemmas we can state the following theorem.

Theorem 5-1.3. When using the actor-critic framework with target networks and sparse rewards (reward of 1 for task completion, 0 otherwise) the optimal target values do not exceed the $[0, 1]$ range.

$$0 \leq Q^*(s, a) \leq 1, \quad \forall s, a$$

When applying these bounds to the target critic values, the resulting algorithm is named Bounded Twin-Delayed Deep Deterministic Policy Gradient (BTD3).

Algorithm 2 Bounded TD3

```
Initialize the critic networks $Q_{\theta_1}$, $Q_{\theta_2}$, and the $\mu_\phi$ arbitrarily with parameters $\theta_1$, $\theta_2$ and $\phi$
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
$ep \leftarrow 0$ (index of current episode)
while ($ep <$ Max number of episodes) do
    $s \leftarrow$ sample $d_0(s_0)$
    $t \leftarrow 0$
    while ($t <$ Length of episode) do
        Choose action with exploration noise $a \sim \mu_\phi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$
        Take action $\mathbf{a}$ and observe $\mathbf{r}$ and $\mathbf{s'}$
        Store transition tuple $(s, a, r, s')$ in $\mathcal{B}$

        Sample mini-batch of $N$ transitions $(s, a, r, s')$ from $\mathcal{B}$
        $\tilde{a} \leftarrow \mu_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
        $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
        Lower and upper bounding the value estimate:
        $y \leftarrow \max(y, 0)$
        $y \leftarrow \min(y, 1)$
        Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, a))^2$
        if $t$ mod $d$ then
            Update $\phi$ by the deterministic policy gradient:
            $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\mu_\phi(s)} \nabla_\phi \mu_\phi(s)$
            Update target networks:
            $\theta'_i \leftarrow \tau\theta_i + (1 - \tau)\theta'_i$
            $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
        end if
        $s \leftarrow s'$
        $t \leftarrow t + 1$
    end while
    $ep \leftarrow ep + 1$
end while
```

5-2 Underestimation outside the demonstration

As discussed in Chapter 4, most online reinforcement learning algorithms cannot utilize demonstration data efficiently, as they tend to explore the state-space also outside the demonstration data distribution. Most algorithms that work efficiently with human demonstration data utilize imitation learning tools, such as behaviour cloning, and restricts the policy to the demonstration data distribution. Getting behaviour cloning to offline with mixed-quality human demonstration data is still an open challenge [7], so it might be beneficial to look into further alternatives.

As it will also be shown in Chapter 7 in the results on a simulated cube picker robot, without behaviour cloning or using other ways to "force" the agent to choose actions similar to the ones in the demonstration dataset, the agent will start to explore the state-space before doing anything similar to the dataset trajectories.

Alternatively, this thesis work proposes a novel way of utilizing demonstration data by deliberately underestimating the values of state-action pair around the demonstration data distribution. Intuitively, this could be considered as a pessimism in the face of uncertainty.

The main challenge that this thesis' author sees regarding with demonstration data comes from the fact that these only cover a specific part of the state and action space. This is especially true for human demonstration data. For example, when we demonstrate a pick-and-place task to a robot via teleoperation multiple times, the trajectories and actions will be similar to each other. This is not necessarily a problem in itself, but actor-critic algorithms often use global function approximators, such as neural networks, to develop global value functions and policies. The previously mentioned demonstration data only covers a very specific part of the state and action space, and it is left to the critic network to extrapolate to unseen scenarios, which can easily go wrong.

Let's consider an example to show why global function approximators might handle demonstration data poorly. For example, in the previously mentioned pick-and-place task, in all of the demonstration the robot will be guided closer to the object from the initial position. Therefore, when using this data to training the critic, a neural network, what should be the value of being in the initial state and then moving away from the object? There is no data to determine this. The core idea of a proposed solution in this thesis project is to introduce underestimation bias in the critic for states that the agent might encounter.

To do this, in the demonstration data the maximum and minimum values of all state and action dimensions are stored to construct a bounding box in which all the state and action trajectories from the demonstration data would fit inside.

$$\begin{aligned} s_{\min,i} \leq & s_i \leq s_{\max,i} \quad \forall s \in \mathcal{B}, \forall i \text{ state dimension} \\ a_{\min,j} \leq & a_j \leq a_{\max,j} \quad \forall a \in \mathcal{B}, \forall j \text{ action dimension} \end{aligned}$$

These lower- and upper-bounds can be used to generate random state-action pairs that are in a proximal Euclidean distance from the demonstration data distribution. However, since the transition dynamics is not known to the agent and could only be approximated from data at best, we do not have the s' next state when choosing arbitrary action a at state s, if these are not in the demonstration data. This means that we cannot compute the target values similarly to the ones in the dataset. The proposed solution is to generate random $(\hat{s}, \hat{a})$ pairs and use the underestimation bias to "pull down" the values of these pairs:

$$\hat{y} \leftarrow \min_{i=1,2} Q_{\theta'_i}(\hat{s}, \hat{a})$$

This new target can be incorporated in the critic loss additionally to the original loss:

$$\text{loss}Q = N^{-1} \sum (y - Q_{\theta_i}(s, a))^2 + N^{-1} \sum (\hat{y} - Q_{\theta_i}(\hat{s}, \hat{a}))^2$$

The last piece of the puzzle is the generation of random state-action pairs. The previously defined s_{min}, s_{max}, a_{min} and a_{max} variables can be used to define a high dimensional $(\dim(s) + \dim(a))$ box that incorporates all the demonstration data:

$$\begin{aligned}
s_{\text{mean}} &= (s_{\text{max}} + s_{\text{min}})/2 \\
s_{\text{range}} &= (s_{\text{max}} - s_{\text{min}}) \\
a_{\text{mean}} &= (a_{\text{max}} + a_{\text{min}})/2 \\
a_{\text{range}} &= (a_{\text{max}} - a_{\text{min}}) \\
\hat{s} &\sim \mathcal{U}([-0.5, 0.5]) K_{bbm,s} s_{\text{range}} + s_{\text{mean}} \\
\hat{a} &\sim \mathcal{U}([-0.5, 0.5]) K_{bbm,a} a_{\text{range}} + a_{\text{mean}}
\end{aligned}$$

Setting the $K_{bbm,s}$ and $K_{bbm,a}$ parameters to 1 will result in a box that covers the whole demonstration data, and setting it to a value higher than one will incorporate a greater area of the state and action space.

5-3 Putting it all together

Combining the proposed modification and applying them to TD3, the resulting algorithm, called Underestimated Bounded Twin-Delayed Deep Deterministic Policy Gradient (UBTD3), will work as follows:

Algorithm 3 Underestimated Bounded TD3

```
Initialize the critic networks $Q_{\theta_1}$, $Q_{\theta_2}$, and the $\mu_\phi$ arbitrarily with parameters $\theta_1$, $\theta_2$ and $\phi$
Initialize target networks $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
Load the normalized demonstration data into $\mathcal{B}$
Pretrain on demonstration for given number of gradient steps
$ep \leftarrow 0$ (index of current episode)
while ($ep <$ Max number of episodes) do
    $s \leftarrow$ sample $d_0(s_0)$
    $t \leftarrow 0$
    while ($t <$ Length of episode) do
        Choose action with exploration noise $a \sim \mu_\phi(s) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$
        Take action $\mathbf{a}$ and observe $\mathbf{r}$ and $\mathbf{s}'$
        Store transition tuple $(s, a, r, s')$ in $\mathcal{B}$
        Sample mini-batch of $N$ transitions $(s, a, r, s')$ from $\mathcal{B}$
        $\tilde{a} \leftarrow \mu_{\phi'}(s') + \epsilon, \quad \epsilon \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$
        $y \leftarrow r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a})$
        Lower and upper bounding the value estimate:
        $y \leftarrow \max(y, 0)$
        $y \leftarrow \min(y, 1)$
        Compute $s_{\text{max}}$, $s_{\text{min}}$, $a_{\text{max}}$, $a_{\text{min}}$
        Compute $s_{\text{mean}}$, $s_{\text{range}}$, $a_{\text{mean}}$, $a_{\text{range}}$
        Sample uniformly $N$ number of $(s, a)$ pairs from the following ranges:
        $\hat{s} \sim \mathcal{U}([-0.5, 0.5]) K_{bbm,s} s_{\text{range}} + s_{\text{mean}}$
        $\hat{a} \sim \mathcal{U}([-0.5, 0.5]) K_{bbm,a} a_{\text{range}} + a_{\text{mean}}$
        $\hat{y} \leftarrow \min_{i=1,2} Q_{\theta'_i}(\hat{s}, \hat{a})$
        $\hat{y} \leftarrow \max(\hat{y}, 0)$
        $\hat{y} \leftarrow \min(\hat{y}, 1)$
        Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} \left[N^{-1} \sum (y - Q_{\theta_i}(s, a))^2 + N^{-1} \sum (\hat{y} - Q_{\theta_i}(\hat{s}, \hat{a}))^2\right]$
        if $t$ mod $d$ then
            Update $\phi$ by the deterministic policy gradient:
            $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_a Q_{\theta_1}(s, a)|_{a=\mu_\phi(s)} \nabla_\phi \mu_\phi(s)$
            Update target networks:
            $\theta'_i \leftarrow \tau \theta_i + (1 - \tau)\theta'_i$
            $\phi' \leftarrow \tau \phi + (1 - \tau)\phi'$
        end if
        $s \leftarrow s'$
        $t \leftarrow t + 1$
    end while
    $ep \leftarrow ep + 1$
end while
```

Chapter 6

Implementation

To test the standard and proposed algorithms, both a standard robot manipulation simulation environment and demonstration datasets are necessary. In the following sections these choices are discussed.

6-1 Simulation environment, robot model

The benchmarking of different reinforcement learning algorithms can be problematic, since authors often use custom environments (either physical or simulated) and code for the implementation. To address this issue, the Stanford Vision and Learning Lab (SVL) and the UT Robot Perception and Learning Lab (RPL) maintains the robosuite repository, which implements standard simulated robotic reinforcement learning environments using the MuJoCo physics engine [44]. It can be used to implement and test custom reinforcement learning algorithms on tasks such as cube lifting, can pick and place, tool hanging and even collaborative tasks between two robots. The cube pickup and can pick-and-place tasks are illustrated on Figure 6-1.

The robot model, the environment and the deep reinforcement learning relevant features (e.g. task definition, reward function, random initialization, sensor signal preprocessing, etc.) are all provided in these benchmarking environments. The robot that is used in the simulation is the model of the Franka Emika Panda robot arm, with 7 joints and a two finger gripper. The only "missing" component is the policy. This thesis work focused on deriving two policy learning algorithms, that are tested in this environment.

6-2 Demonstration datasets

The sister project of robosuite, robomimic, was implemented to provide tools for learning from human demonstration data, collected in the robosuite environment [7]. It utilizes the

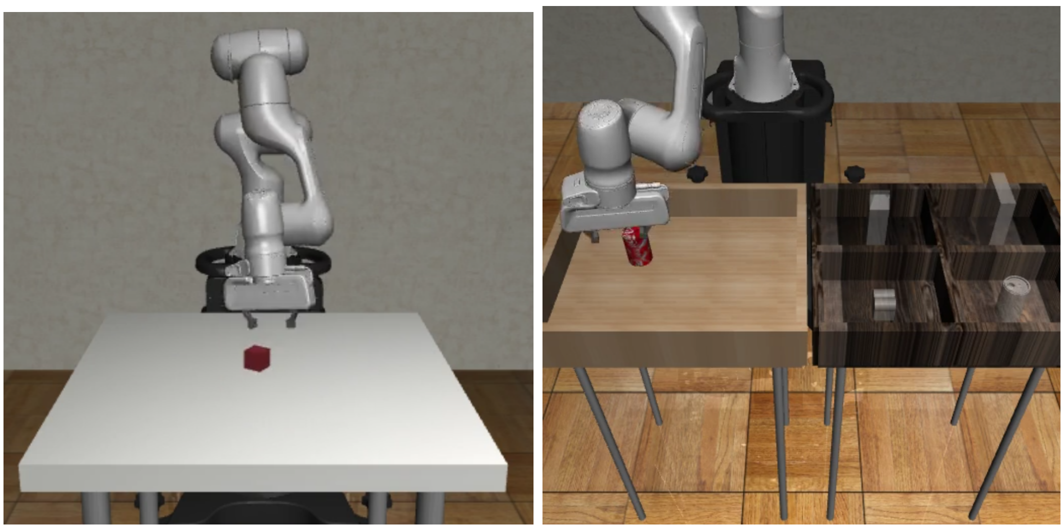


Figure 6-1: Robosuite environments used for the testing of the proposed and standard algorithms. On the left the cube lift environment is shown, where the objective of the agent is to lift the cube from the table above a certain height. On the right, the can pick-and-place environment can be seen, where the aim of the robot is to pick up a can randomly initialized in the left box and move it to the bottom right compartment.

datasets provided by the RoboTurk project, which contains both machine generated and human demonstrations [37]. For the discussion about the importance of including both datasets see Chapter 4.

In these frameworks there are available machine generated and human demonstration datasets for both previously mentioned environments (can pick-and-place and cube lifting):

- Machine generated dataset ($\approx 200 - 600,000$ samples): data from a Soft Actor Critic algorithm that has learnt the task from scratch using shaped reward function. The dataset consists of trail runs with different policy checkpoints during its learning process (so not all successful).
- Multi-human dataset ($\approx 30,000$ samples): demonstration dataset generated by 6 human operators with different experience level and proficiency
- Proficient human dataset ($\approx 9,000$ samples): demonstration dataset generated by a single, skilled operator (currently not used for the tests, but could be included in future work)

The algorithms are tested with binary completion reward, so the corresponding datasets are used (dense reward is also available). Furthermore, the done signal is set to be one only in case of successful task completion. Pardo et al. [45] highlighted the importance of choosing the done signal carefully in case of time limited tasks and their work is a recommended material for the interested reader.

The datasets used for the two tasks contain the following selected sensor signals:

- Robot arm proprioceptive data:

  - End-effector orientation
  - End-effector linear velocity (only for can pick-and-place task)

- Object information:
  - Object position in Cartesian space
  - Object orientation in space
  - Linear distance of object from the end-effector
  - Orientation difference of object and end-effector

6-3 Hardware and software

All the experiments were performed on an Asus Vivobook N580GD laptop with 8th generation Intel Core i7 processor and 16 GB of RAM. The benefits of using a GPU instead of CPU is less clear in case of reinforcement learning, which is generally a CPU extensive task. However, in future work moving the project to a server and testing with GPU computing is strongly recommended.

Regarding the software, the operating system used was Pop!_OS 22.04, which is an Ubuntu-based Linux distribution. Both the robosuite and robomimic components are implemented in Python and were installed from source. The implementation of the TD3 algorithm (later will be denoted as "standard") is based on the OpenAI's Spinning Up project. The handling of the demonstration data, pretraining on it and several logging functionalities were added to this code. In future work, this project could be migrated to be based on the latest Stable Baselines.

6-4 Investigated test cases

The necessary training length for the different tasks and datasets were found empirically. In general, the more difficult can pick-and-place environment requires more trials and longer training time than the cube lifting task. On the setup described in the previous section, the training for the cubelift task takes around 2.5-3 hours, while the can pick-and-place task requires roughly 6 hours. The main bottleneck for the speed is currently posed by the environmental interaction, so the additional steps for the proposed algorithms do not influence the training time drastically.

Due to these already quite long training times, the current tests are restricted to cubelift and can pick-and-place tasks (even more difficult tasks could be tested if the project was moved to a server):

1. Lift task
   (a) Machine generated dataset: consists of 1500 trials (5 policy with 300 trials each) where each episode contains 150 steps, which results in 225,000 samples altogether. For the tests in this thesis work the episode lengths were set to 200 steps (10 seconds in real time) instead of the 150.

   (b) Multi-human dataset: consists of 300 trials (6 operators with 50 trials each) where the episodes have different lengths based on the proficiency of the operator, but all episodes are successful. The dataset consists of 31,127 transition samples. For the tests the episode lengths were set to 200 steps (10 seconds in real time).

2. Can pick-and-place task

   (a) Machine generated dataset: consists of 3900 trials (13 policy with 300 trials each) where each episode contains 150 steps, which results in 585,000 samples altogether. For the tests in this thesis work the episode lengths were set to 200 steps (10 seconds in real time) instead of the 150.

In all three previously mentioned test cases all the algorithms are tested from Chapter 5:

- TD3 with pretraining on demonstration data ("standard")
- Bounded TD3 ("BTD3")
- Underestimated Bounded TD3 ("UBTD3")

Finally, to assess the potential in controlling the value estimate of random state-action pairs around the demonstration data distribution (UBTD3), the evolution of these values are also investigated. The main plots in general are the critic values and performance graphs. These are shown in Chapter 7.

Chapter 7

Results

7-1 Critic value estimates of dataset samples

There are several signals and comparison metrics that could be considered. Since the main idea behind the proposed algorithms was to constraint the critic values (Q values), first these are shown on Figure 7-1. The following can be concluded from the different experiments:

1. **Cube lifting task with machine generated dataset**: The maximum assigned critic value stay close to the theoretical maximum of 1 for all three algorithm. However, the standard algorithm does reach values at approximately 1.3, while the other two stay closer. The standard algorithm produces high negative value estimates, which the BTD3 and the UBTD3 algorithms successfully avoid via the target bounds.
2. **Can pick-and-place task with machine generated dataset**: The two proposed algorithms produce similar value estimates as in the previous case. The value estimate of the standard algorithm, however, diverge from the theoretical range and have big variance. There were less gradient steps performed in case of the standard algorithm, since the pretraining had to be omitted due to instabilities in the value estimates.
3. **Cube lifting task with human demonstration data**: The two proposed algorithms, again, produce similar value estimates as in the previous case. The value estimates of the standard algorithm is unstable and they do not converge.

The previously mentioned Figure 7-1 only show the value of 2 transition samples (minimum and maximum) assigned by the critic. So what if in case of the standard algorithm the highly negative values are outliers and most values are close to the $[0, 1]$ range? The actor loss is actually the mean of the assigned critic values, so it is suitable to get more information about it. The mean critic values for the three test cases are shown on Figure 7-2:

1. **Cube lifting task with machine generated dataset**: For both proposed algorithms the mean critic value estimate is converging to approximately 0.5, however, as the replay

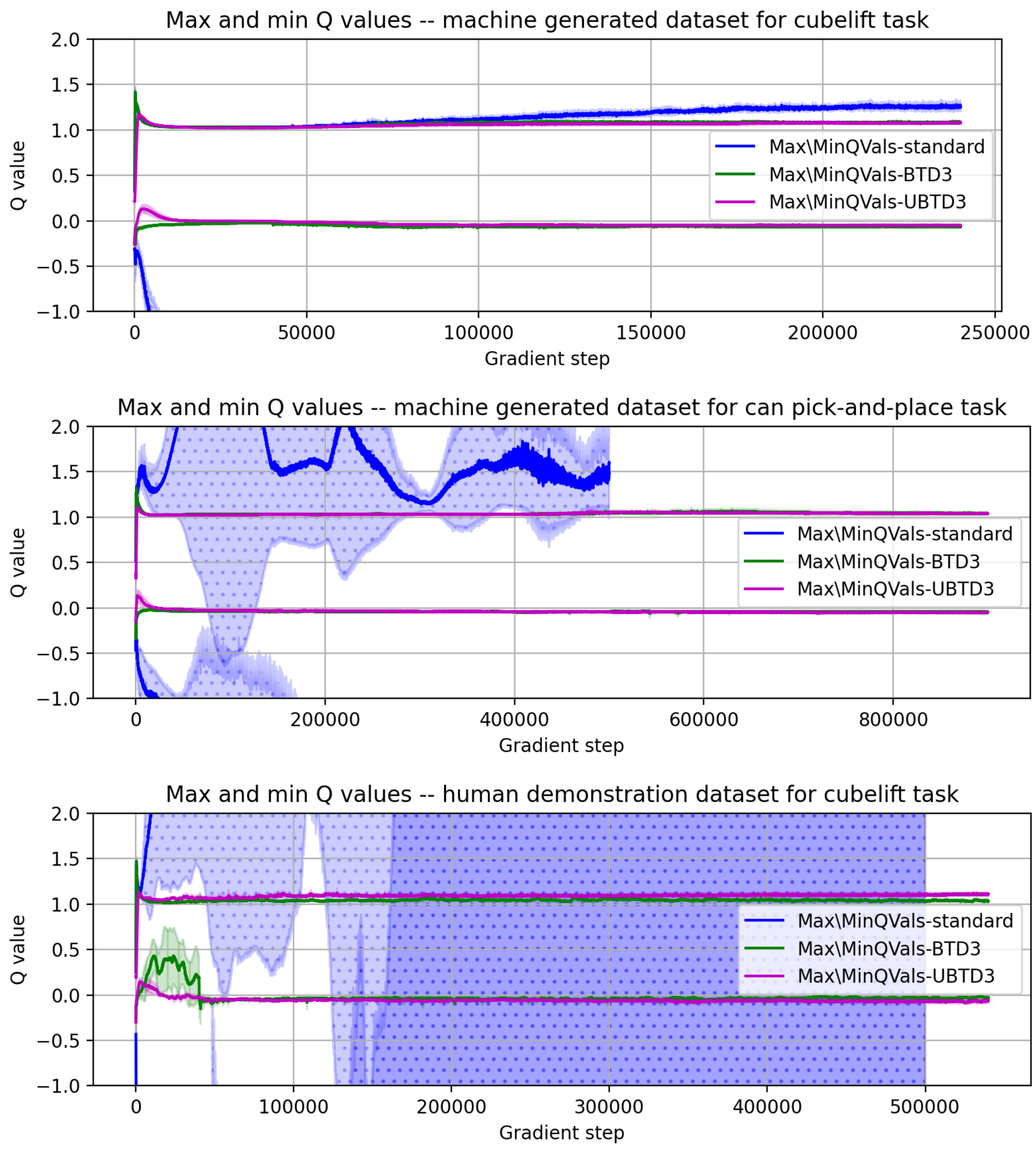


Figure 7-1: Minimum and maximum Q values assigned by the critic to transition samples randomly selected from the replay buffer (different datasets and tasks). The thick, solid lines show the mean of four runs with four different seed values (0, 15, 22, 96). The shaded area shows 1 standard deviation of the different runs. The plots show that the 2 proposed algorithm manages to keep the values close to the desired $[0, 1]$ range. In case of the cubelift task with human demonstration and the can pick-and-place task with machine generated data the standard algorithm could not be pretrained before the environmental interactions, due to instability, and that is why there are less gradient steps in case of the corresponding (blue) curve.

buffer is filling up with more and more successful episodes (since good performance is achieved as will be seen), the slight but steady increase is expected to continue. The mean value estimates of the standard algorithm exit the $[0, 1]$ range at the negative end, but still stay close to 0.

2. **Can pick-and-place task with machine generated dataset**: For the two proposed algorithms the situation is similar to the cube lift task; the value estimates stay around the middle of the $[0, 1]$ range with slight but steady increase, which can be explained by the replay buffer filling with successful trials (as will be shown later). The standard algorithm produces on average even more negative value estimates than before.

3. **Cube lifting task with human demonstration data**: The value estimates of the two proposed algorithms stay in the $[0, 1]$ range, but now closer to 0 than before. For the standard algorithm the instability that we saw in case of the minimum and maximum critic value estimates also affect the mean value estimate and produces a diverging, oscillatory behaviour.

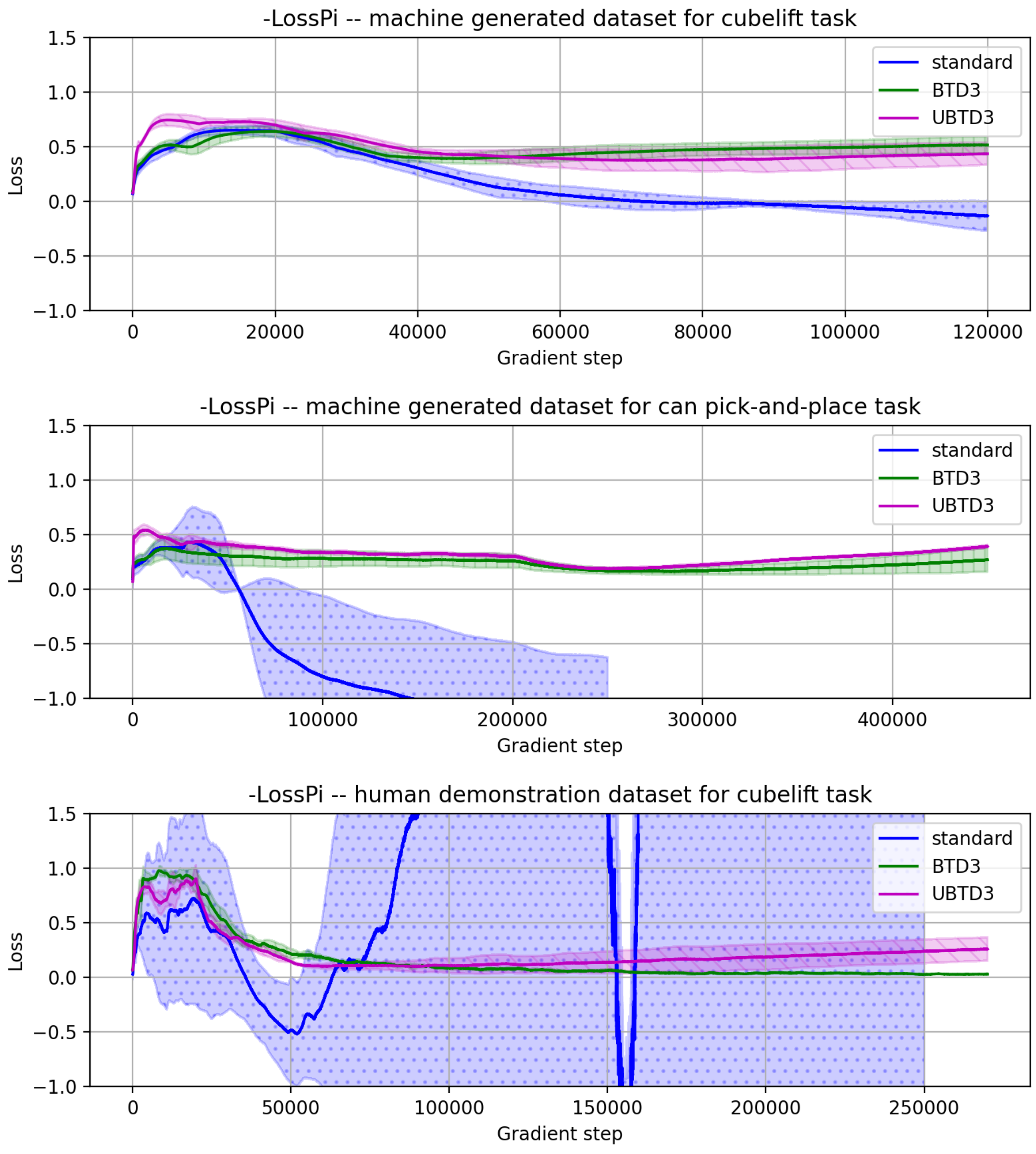


Figure 7-2: Mean of the Q values assigned by the critic to samples in the batch randomly selected from the replay buffer (different datasets and tasks). The thick, solid lines show the mean of four runs with four different seed values (0, 15, 22, 96). The shaded area shows 1 standard deviation of the different runs. The plot shows that as the learning progresses the standard algorithm is becoming more and more pessimistic about the state-action pairs in the replay buffer (despite filling it with successful trials as will be shown later). The two proposed algorithm has a higher mean value (more optimistic). In case of the cubelift task with human demonstration and the can pick-and-place task with machine generated data the standard algorithm could not be pretrained before the environmental interactions, due to instability, and that is why there are less gradient steps in case of the corresponding (blue) curve.

7-2 Critic value estimates of random state-action samples

The main idea behind the UBTD3 algorithm was to control the value estimates of state-action pairs, that are around (and in) the demonstration data distribution. As discussed in Chapter 5, if the state and action space coverage of the demonstration data is not extensive, which is usually the case for human demonstration data, then the value estimates of relevant state-action pairs that are not in the dataset can get arbitrarily bad. UBTD3 addresses this issue by generating random state-action pairs around the demonstration data distribution and deliberately introduces an underestimation bias when calculating their target value. Therefore, it is expected that in case of this algorithm the value of random state-action pairs will converge towards 0, the lowest possible target value due to the target bounds.

Figure 7-3 shows the critic value estimates of the random state-action pairs during pretraining. This means that there is no environmental interaction, since the main purpose with UBTD3 was to enable the agent to perform similar behaviour to the demonstrations right after pretraining, without extensive exploration of the state and action space. The following observations can be made based on Figure 7-3:

1. **Cube lifting task with machine generated dataset**: The UBTD3 algorithm manages to regulate the value of the random state-action pairs as expected. The maximum value estimate of the standard and the BTD3 algorithm is converging to a stable value of around 4.5, but it is actually higher than the value of 1 that is assigned to the dataset state-action pairs during the pretraining phase (see Figure 7-1). This means that without the deliberate value underestimation of random state-action pairs their values can get higher than the values assigned to any of the demonstration samples. The minimum values estimates are also lower than the ones for the dataset transitions.
2. **Can pick-and-place task with machine generated dataset**: The UBTD3 algorithm manages to regulate the value of the random state-action pairs as before. The value estimates of the standard algorithm is not plotted, since it was highly unstable. The maximum value estimates of the BTD3 algorithm is showing an increasing trend, at the end of the experiment it took values around 10. The minimum values are more steady, they converge to around -10. However, these mentioned values are well out of the $[0, 1]$ range.
3. **Cube lifting task with human demonstration data**: The UBTD3 algorithm manages to regulate the value of the random state-action pairs as before. The value estimates of the BTD3 algorithms are stable, and they converge to similar values as in case of the machine generated dataset (≈ 4 for maximum and ≈ -5 for minimum). The standard algorithm is unstable, its value estimates are diverging.

The maximum and minimum critic values during learning for the different seeds for all three algorithms and tasks can be found in the Appendix on Figure A-1, Figure A-3 and Figure A-5.

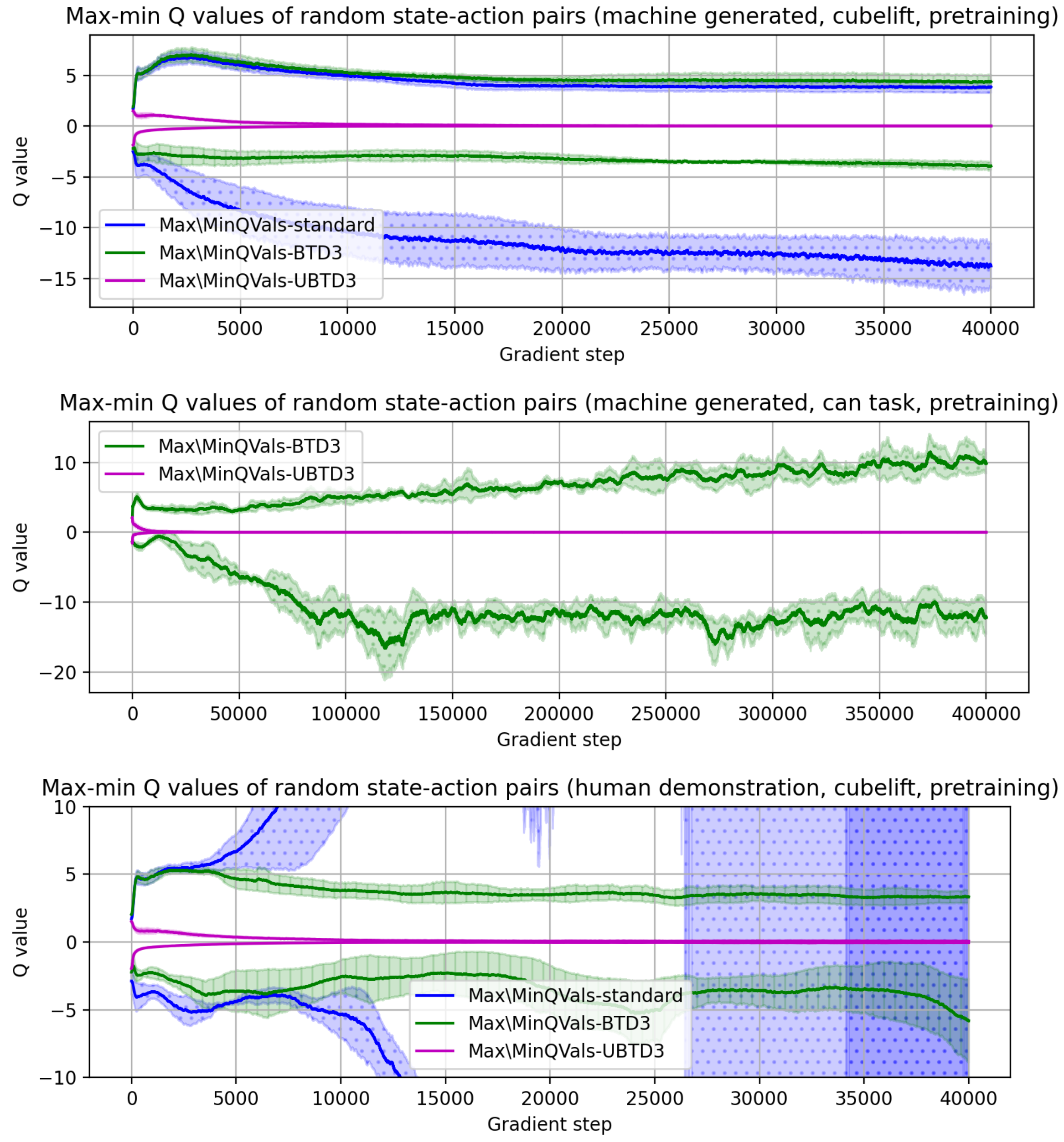


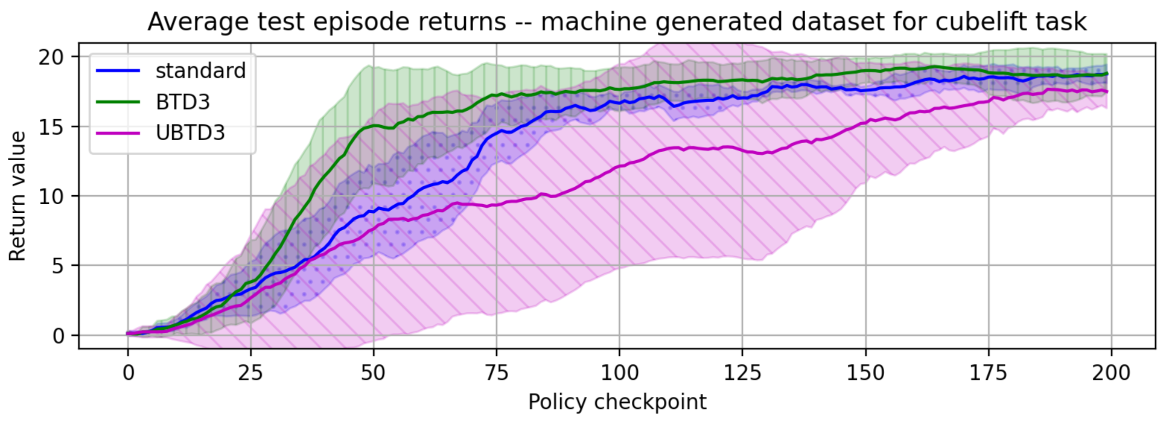
Figure 7-3: During pretraining (no env. interaction), maximum and minimum Q values assigned by the critics to randomly generated state-action pairs around the demonstration data distribution (machine generated dataset, cubelift task). The thick, solid lines show the mean of four runs with four different seed values (0, 15, 22, 96). The shaded areas show 1 standard deviation of the different runs. It can be observed, that the UBTD3 algorithm manages to regulate the values of state-action pairs that are not in the dataset and successfully assign low values to them in the $[0, 1]$ range. In case of the other algorithms, the value of random state-action pairs is not directly regulated, only through the interpolation/extrapolation capabilities of the critic neural networks. In case of the can pick-and-place task the value estimates of the standard algorithm is not plotted, since pretraining on the demonstration data was not possible due to instabilities.

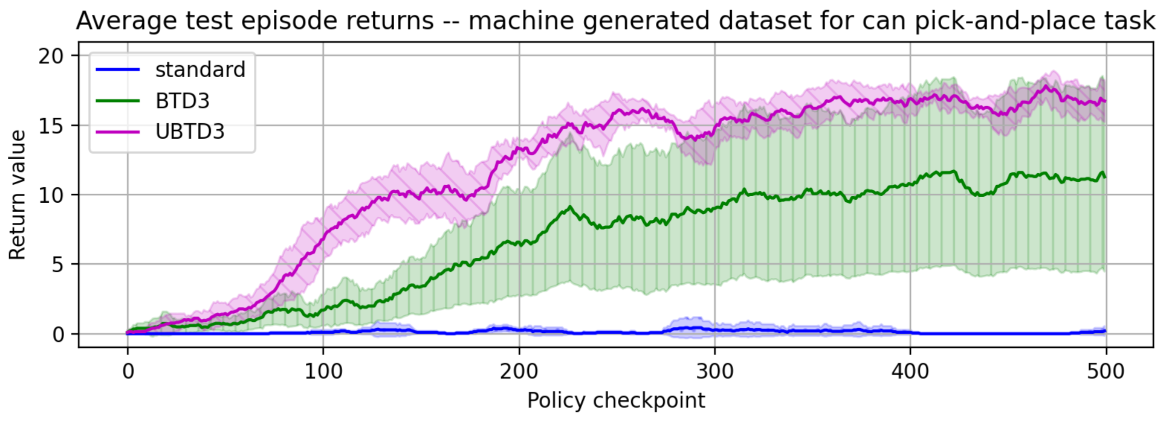
7-3 Performance of the different algorithms

The performance of improving policies trained by the different algorithms is shown on Figure 7-4. In case of the cube lifting task, the return value of 20 means that the robot arm could pick up the cube and hold it for 20 time steps, which is 1 second. The experiment is stopped at this point, so the maximum achievable return value is 20. In case of the can pick-and-place task, the can needs to stay in the correct compartment for the same 20 samples for the episode to be fully successful. It is important to highlight, that the plots on Figure 7-4 show average of 5 test episode returns, so it can be thought of as a form of success rate measure, where the value 20 would correspond to 100% expected success rate. The following observations can be mode from the different test cases:

1. **Cube lifting task with machine generated dataset**: All three algorithms manage to learn the task. The lower and upper bounding the critic values (BTD3) speeds up the policy improvement, since there are higher average return values after the same training steps. Although the learning curve is steeper for the BTD3, the standard algorithm converge to policies with similar (excellent) performance after enough iterations. In case of the UBTD3 algorithm the performance is improving slower than the standard algorithm or BTD3. Its variance is also higher, indicating that the algorithm is more sensitive to initial conditions.
2. **Can pick-and-place task with machine generated dataset**: While the standard algorithm fails, the BTD3 and the UBTD3 algorithms both manage to achieve the task, although they both seem to converge toward less than 100% success rate. The UBTD3 algorithm, however, has much less variance among the different seeds and it also converges to a good performance (it would be roughly 85% success rate).
3. **Cube lifting task with human demonstration data**: The only algorithm that manages to reliable achieve the task is the UBTD3 algorithm. Note that compared to the test case with the machine generated dataset on the same task, reaching similar level of performance here took 2500 trials instead of the previous 1000 (there is a policy checkpoint after every fifth trial, so 500 policy checkpoint = 2500 trials).

The learning curves for the different seeds for all three algorithms and tasks can be found in the Appendix on Figure A-2, Figure A-4 and Figure A-6.





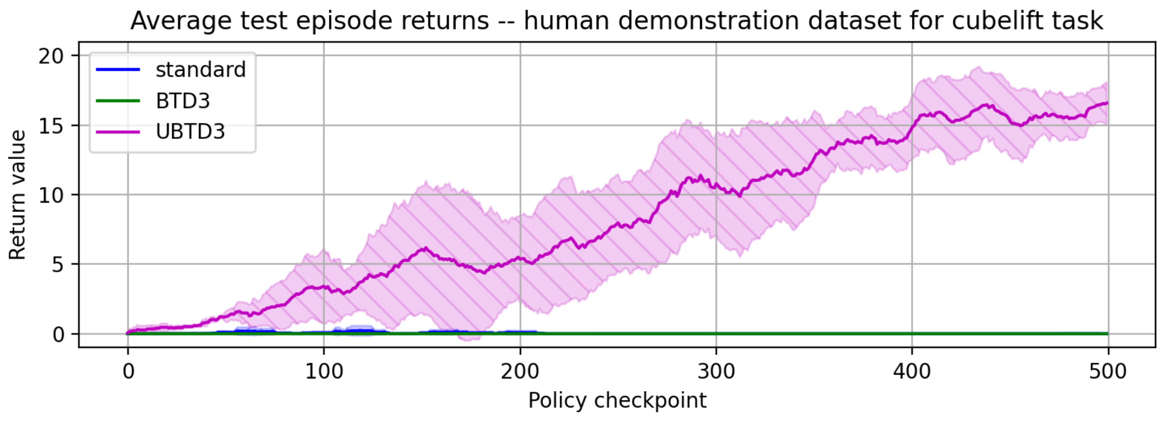


Figure 7-4: Average of 5 test episode returns using the policies trained by the different algorithms as the learning progresses (different datasets and tasks). Although binary completion reward is used, in case of the cubelift task the trial is only stopped if the robot managed to lift the cube for 1 second, which is 20 samples. Values below 20 indicate that the grasping of the cube was not robust and it was dropped. In case of the can pick-and-place task, the value 20 indicate that the robot placed the can in the correct compartment and the can stayed there for the same 20 time steps (1 second). The thick, solid lines show the mean of four runs with four different seed values (0, 15, 22, 96). The shaded areas show 1 standard deviation of the different runs.

Chapter 8

Discussion

8-1 Connection between value estimates and performance

In the previous chapter the value estimates of samples from both the given datasets and random state-action pairs were presented. Additionally, the performance for the same test cases were also shown. As discussed in the literature study chapters, the connection between the critic value estimates and the system performance is an active research area. This section aims at providing possible theories based on the empirical results.

Target bounds and performance In case of the cubelift task with the machine generated demonstration data all three algorithm performed well. However, the maximum and minimum value estimates of the BTD3 algorithm were closer to the $[0, 1]$ range than the estimates of the standard algorithm, and BTD3 performed better, so there might be connection between the two aspects. This theory is reinforced in case of the can pick-and-place task with the machine generated data, where simply applying the target bounds (BTD3) managed to avoid the unstable behaviour that could be observed in case of the standard algorithm. In case of the cubelift task with human demonstration data, however, both the standard and BTD3 algorithms fail, despite BTD3 respecting the 0 and 1 target value bounds. All-in-all, it can be stated that applying the target bounds (BTD3) only improved performance when compared to the standard algorithm.

Value of random state-action pairs and performance Comparing Figure 7-1 and Figure 7-3 for the standard and the BTD3 algorithms, the value estimates of randomly generated state-action pairs close to the demonstration data distribution can get higher than the ones in the demonstrations themselves. This is due to interpolation/extrapolation issues probably because of the relatively small-sized training data. This fact actually poses a theoretical problem when trying to apply the standard algorithm or BTD3. It poses the risk for the agent to converge toward unknown states and actions in those states, which just happened to have high estimated values. This is what behavioural cloning also tries to avoid by making the agent favour actions similar to the demonstration dataset. Meanwhile, the proposed UBTD3 algorithm directly drives the value estimates of unknown state-action pairs to zero, which in theory would also encourage the agent to choose actions similar to the dataset ones. The fact that the UBTD3 algorithm was the only one that could effectively utilize the small-sized human demonstration data is promising. It was also an improvement over BTD3 in the can pick-and-place task with the machine generated dataset. However, in case of the cubelift task with machine generated data UBTD3 was more sensitive to the initialization than the other two algorithm, which resulted in on average slower learning.

8-2 Summary, future prospects

Lower- and upper-bounding the critic function by theoretical minimum and maximum values is possible by limiting the target network outputs (BTD3 algorithm). The empirical results showed that it is an effective way to constraint the value estimates both in terms of required gradient steps (it converges fast), computational requirement (applying only min and max operators) and implementation (two lines of code). It results in less extreme critic values, which enabled improvement over the standard algorithm when using the machine-generated datasets both in the cube lifting and the can pick-and-place tasks.

On the human demonstration data the bounded-underestimated algorithm UBTD3 vastly outperforms both the standard algorithm and BTD3. It was also the best performing in the can pick-and-place task with machine generated data, but was sensitive to initialization in case of the cubelift task with machine generated dataset. Further testing is necessary to be able to assess the potential of the proposed algorithms. Currently, the different trends were created by averaging the results of only 4 different runs (seeds) due to the time-demanding nature of these experiments. At least 10 runs, however, would be better to derive more accurate statistics. Other, more difficult tasks could also be tested in the robosuite environment, such as nut assembly.

Other future development possibilities include:

- Moving the project to a server: This would speed up the testing process.
- Experimenting with prioritized experience replay: The data distribution is changing with the online environmental interactions and all data is sampled with the same probability. Vecerik et al. [8] showed that prioritized experience replay can greatly improve sample efficiency.
- Using recurrent neural network (RNN) in the actor and critic networks: Mandlekar et al. [7] (the developers of robomimic) showed that in offline learning incorporating memory in the critic and actor is crucial when dealing with human demonstrations. RNNs are a good choice.

Chapter 9

Conclusion

The nature and effects of estimation bias in deep reinforcement learning algorithms requires further attention from both theoretical and practical point of view. Algorithms that manage to overcome the undesirable overestimation bias by consistently underestimating the true value functions suffer from the effect of underestimation bias, which is not yet researched thoroughly.

This thesis work showed the potential in setting bounds to the value function of the TD3 algorithm in sparse reward problems to limit the estimation bias. Furthermore, it proposed a novel way of utilizing demonstration data that cover only a specific part of the state and action space, which is a common trait of human demonstration data.

The test cases used in the simulation showed promising results, but testing in additional simulation environments is recommended to get a more comprehensive metric of expected performance improvement. Additionally, based on the literature survey, extending both the actor and critic with memory (e.g. using RNN) and using prioritized experience replay could further improve the performance when using human demonstration data.

Appendix A

Individual Runs with Different Seeds

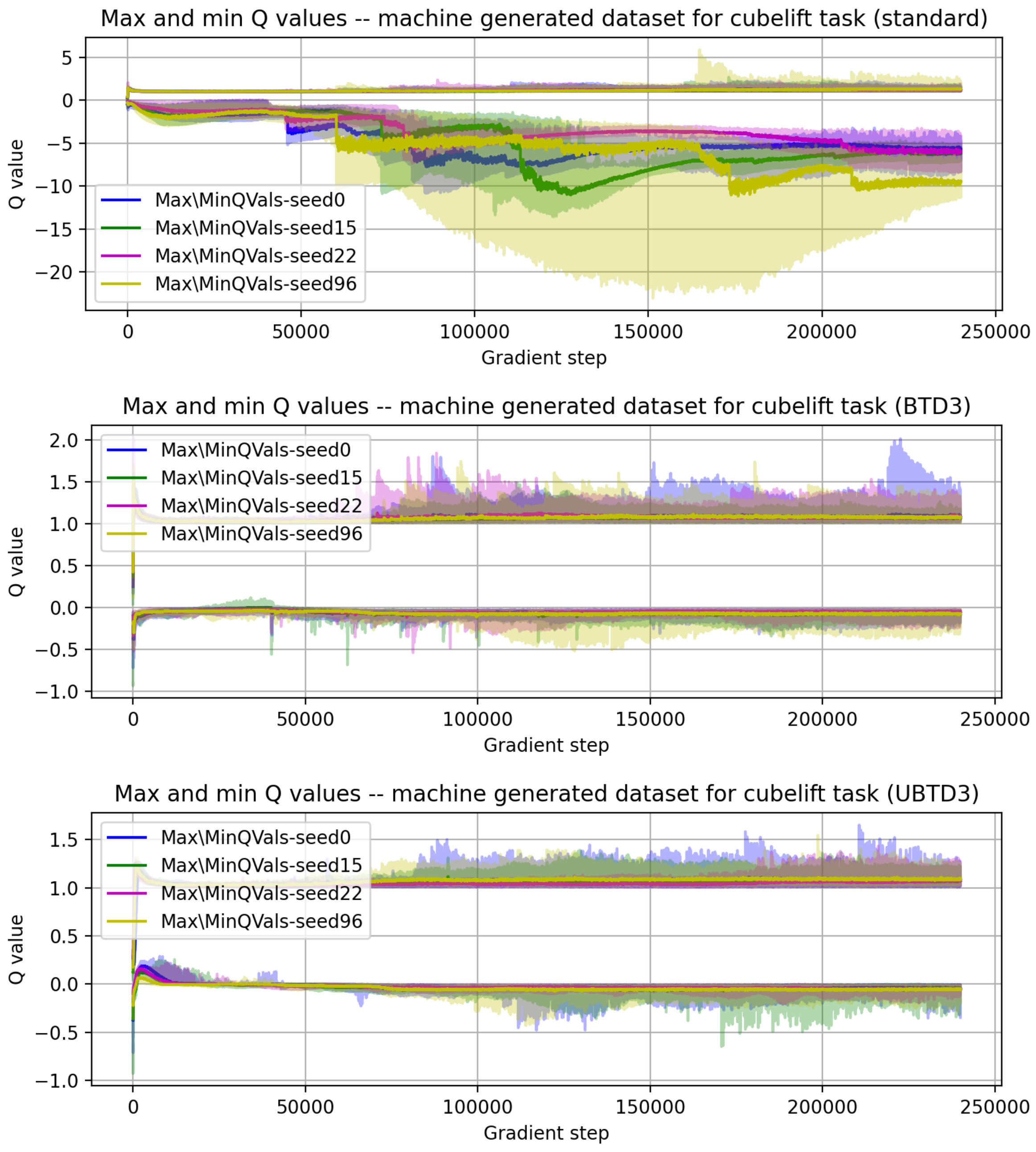


Figure A-1: Maximum and minimum value estimates of the different algorithms while training on the machine generated dataset and the collected experiences in the cubelift task.

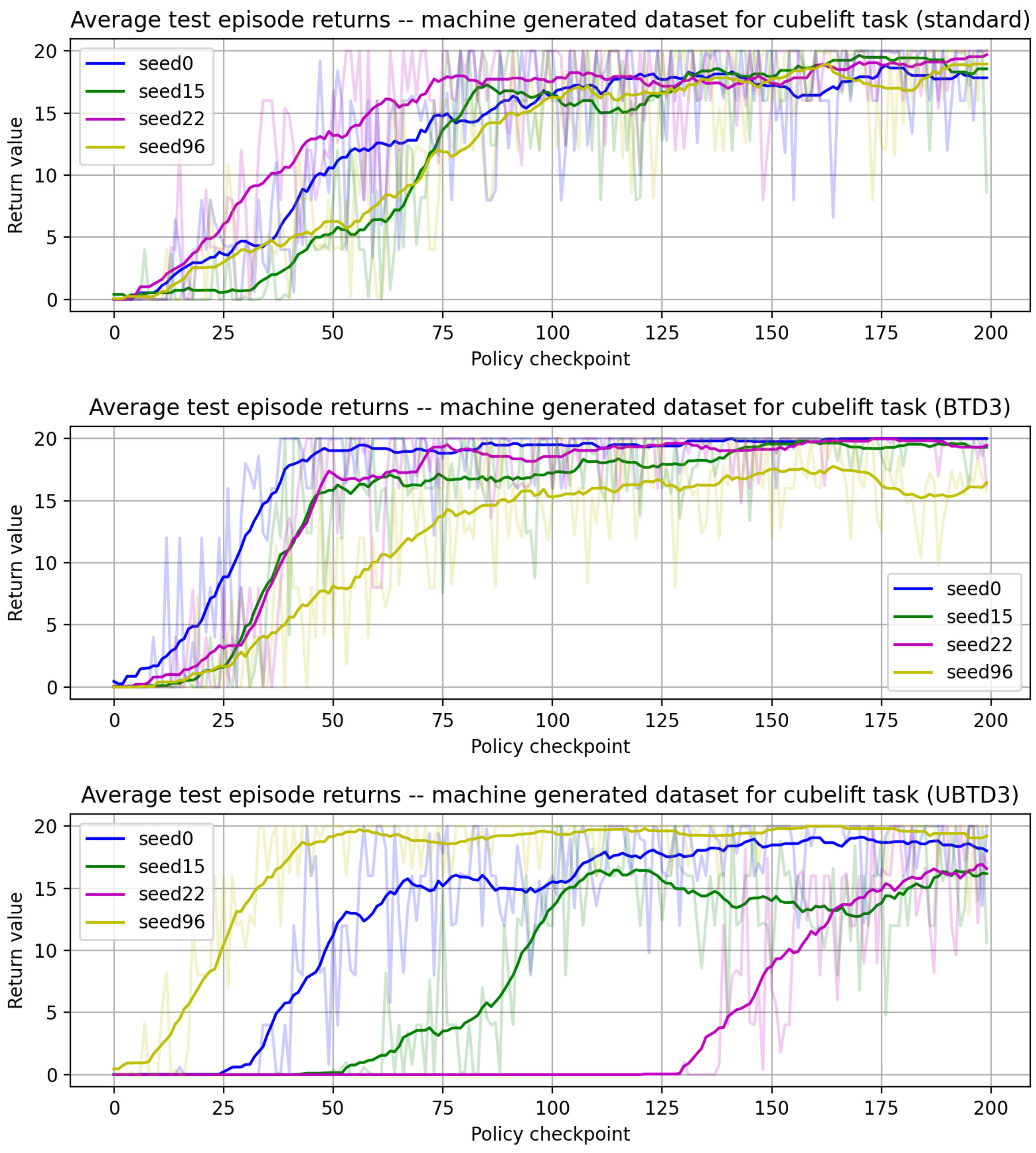


Figure A-2: Performance improvement of the different algorithms as the learning progresses in the cubelift task using the machine generated dataset. During the learning process after every 5th trail (1000 samples) the current policy is evaluated using 5 trials. In each trial the robot needs to lift the cube and hold it for 20 samples (1 second). The figures show the average number of terminal state samples in the 5 trials for every evaluation step.

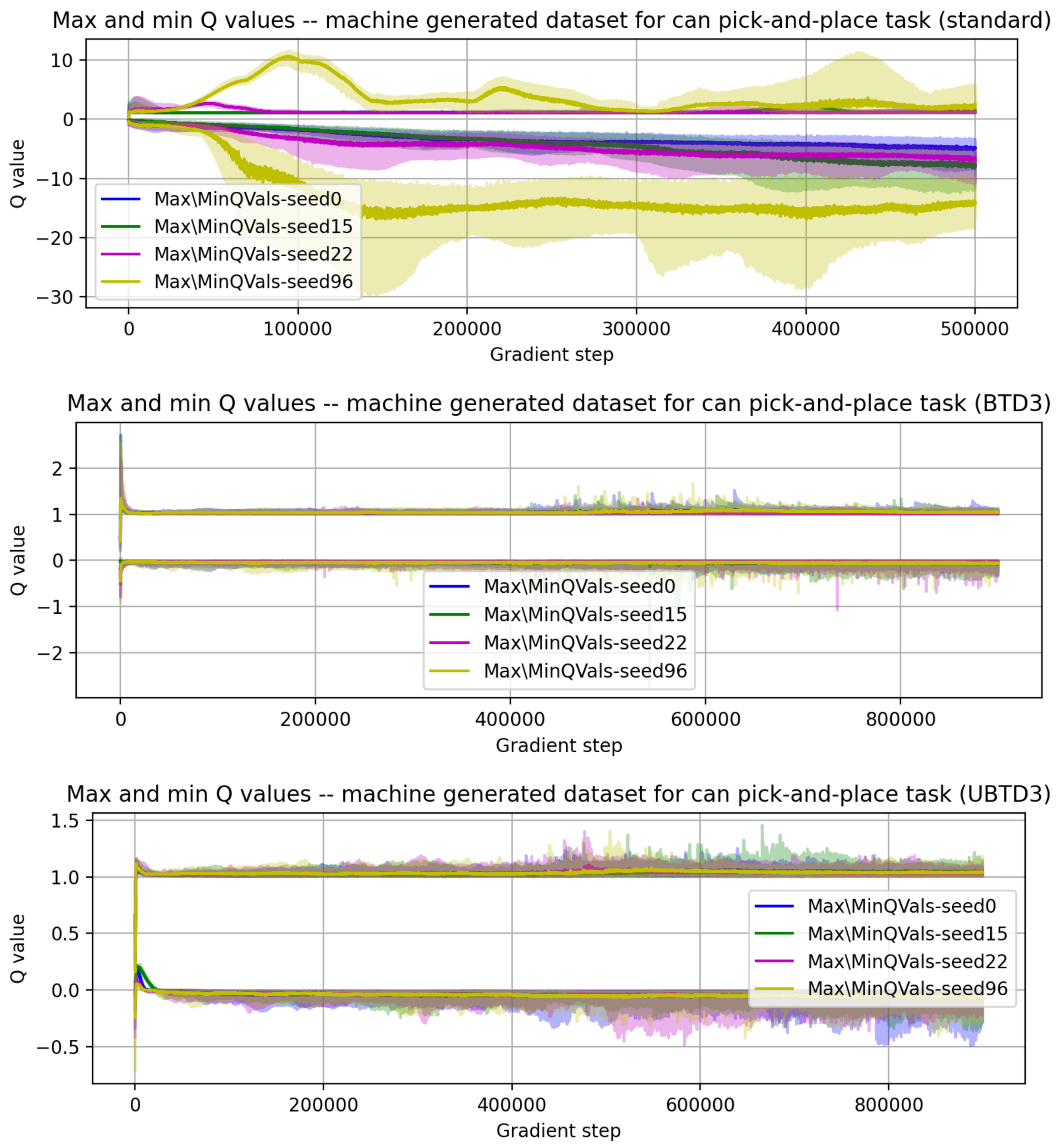


Figure A-3: Maximum and minimum value estimates of the different algorithms while training on the machine generated dataset and the collected experiences in the can pick-and-place task.

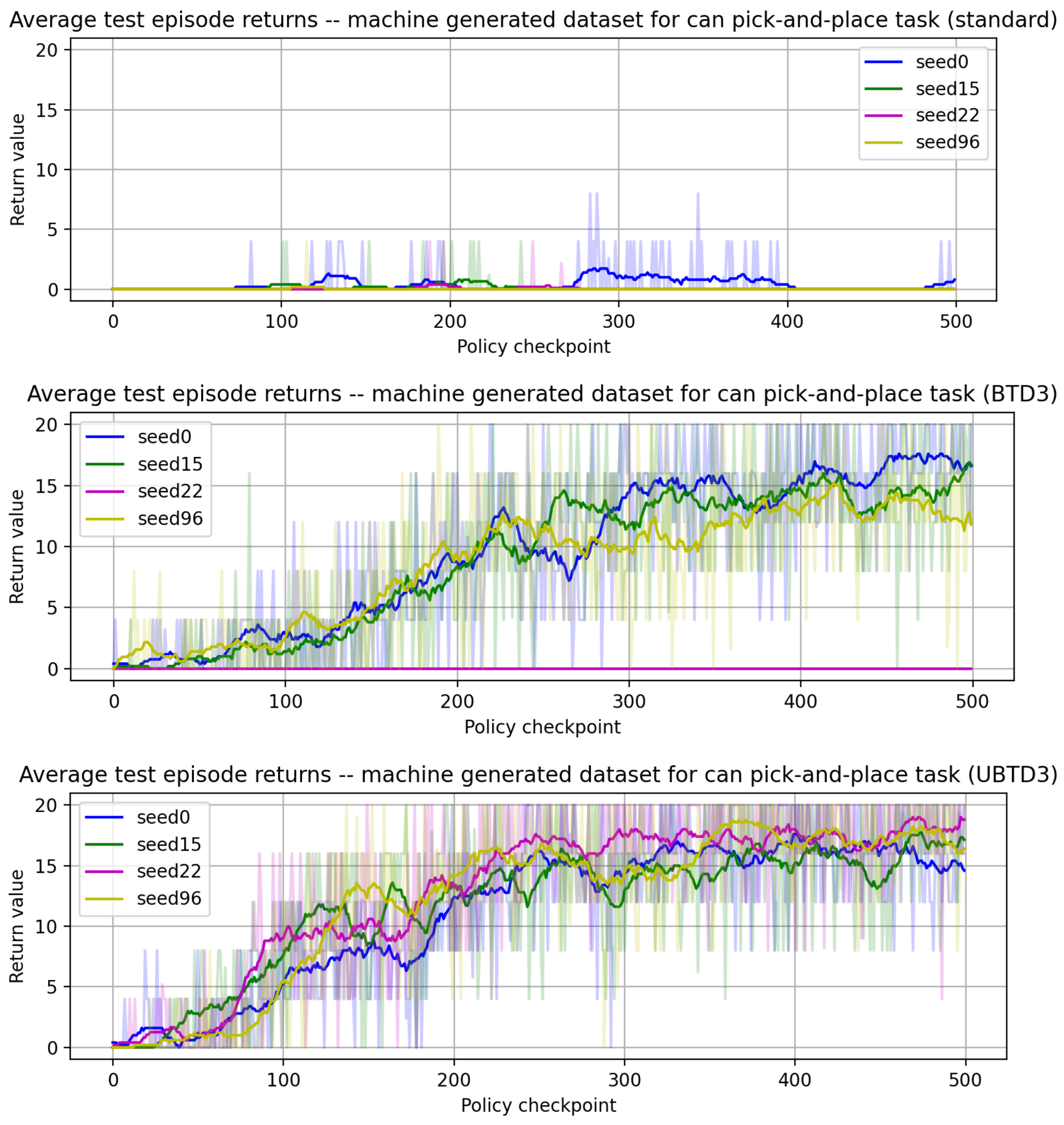


Figure A-4: Performance improvement of the different algorithms as the learning progresses in the can pick-and-place task using the machine generated dataset. During the learning process after every 5th trail (1000 samples) the current policy is evaluated using 5 trials. In each trial the robot needs to lift the can and place it in the designated compartment for 20 samples (1 second). The figures show the average number of terminal state samples in the 5 trials for every evaluation step.

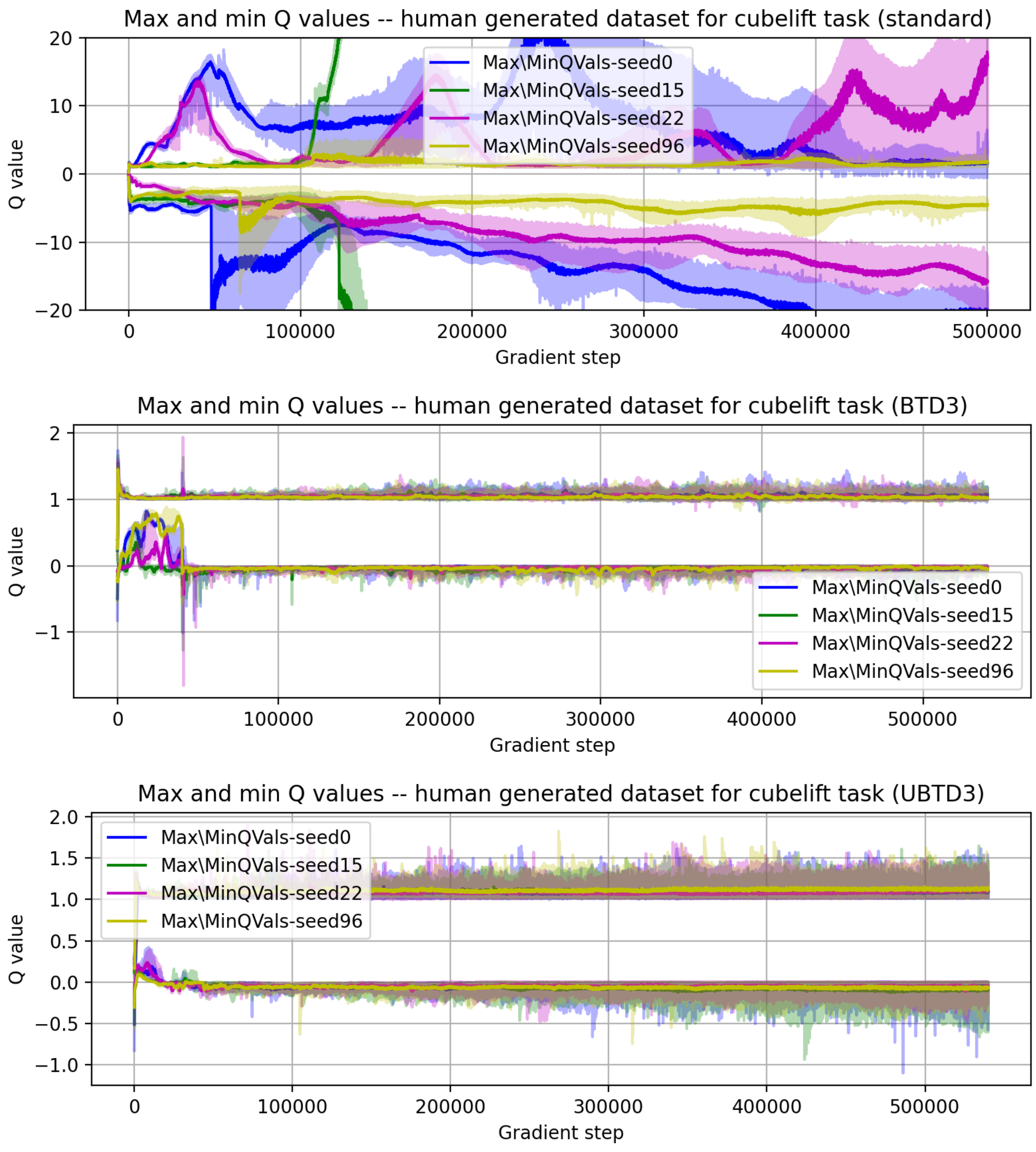
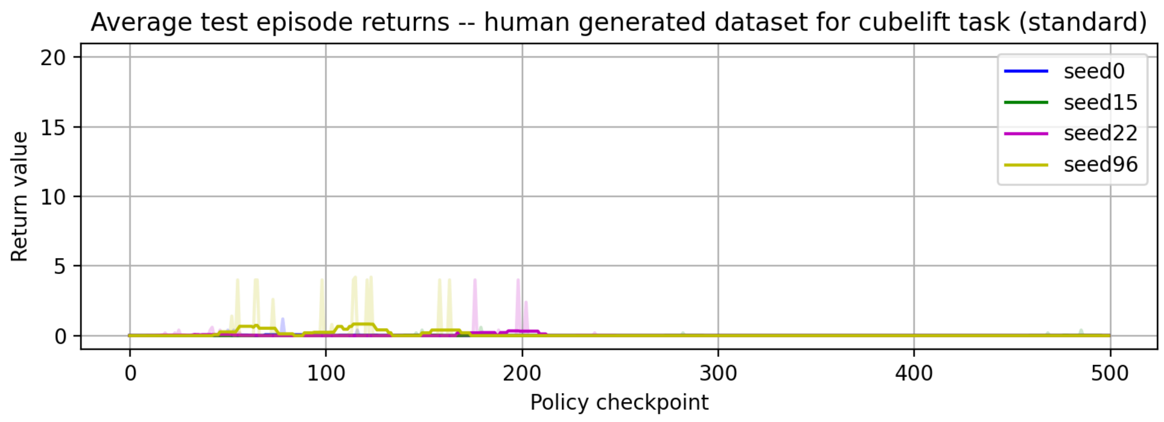
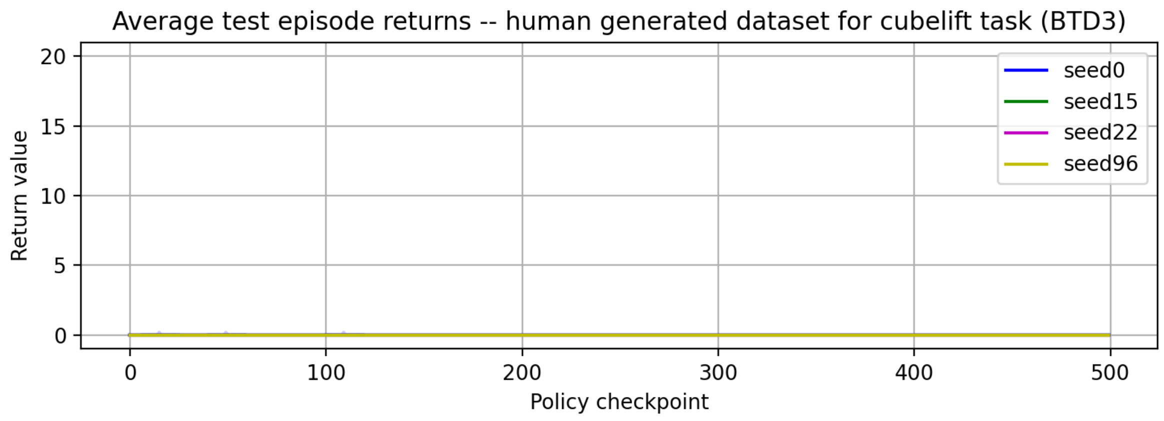


Figure A-5: Maximum and minimum value estimates of the different algorithms while training on the human demonstration dataset and the collected experiences in the cubelift task.





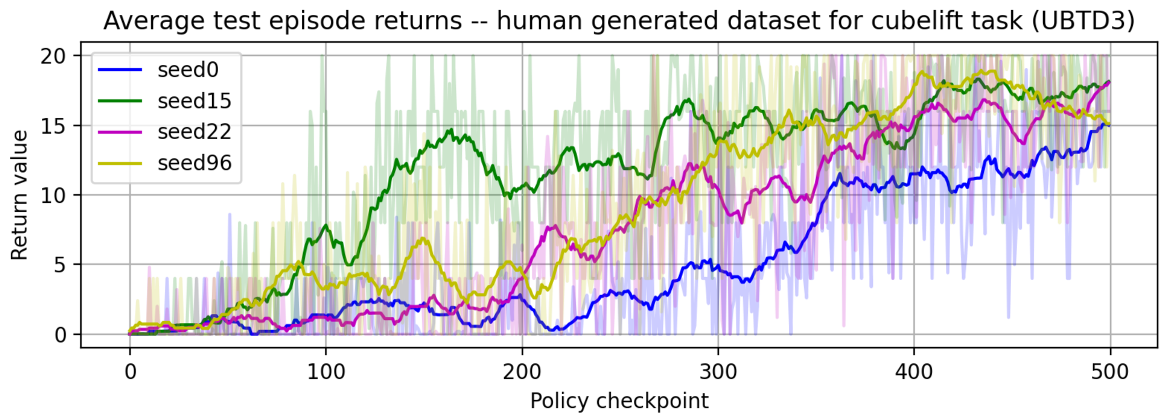


Figure A-6: Performance improvement of the different algorithms as the learning progresses in the cubelift task using the human demonstration dataset. During the learning process after every 5th trail (1000 samples) the current policy is evaluated using 5 trials. In each trial the robot needs to lift the cube and hold it for 20 samples (1 second). The figures show the average number of terminal state samples in the 5 trials for every evaluation step.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction Second edition.* The MIT Press, second edition ed., 2015.

[2] J. Luo, O. Sushkov, R. Pevceviciute, W. Lian, C. Su, M. Vecerik, N. Ye, S. Schaal, and J. Scholz, "Robust Multi-Modal Policies for Industrial Assembly via Reinforcement Learning and Demonstrations: A Large-Scale Study," *CoRR*, vol. abs/2103.11512, 3 2021.

[3] J. Hansen, K. Kastner, Y. Huang, A. Courville, D. Meger, and G. Dudek, "Learning to Manipulate from Pixels on Rigid Body Robots with a Kinematic Critic," tech. rep., Mobile Robotics Lab, McGill University, 2022.

[4] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang, "Solving Rubik's Cube with a Robot Hand," 10 2019.

[5] M. V. Balakuntala, U. Kaur, X. Ma, J. Wachs, and R. M. Voyles, "Learning Multimodal Contact-Rich Skills from Demonstrations Without Reward Engineering," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 3 2021.

[6] I. L. Bosque, *Towards Corrective Deep Imitation Learning in Data Intensive Environments Helping robots to learn faster by leveraging human knowledge.* PhD thesis, TU Delft, Delft, 2021.

[7] A. Mandlekar, D. Xu, J. Wong, S. Nasiriany, C. Wang, R. Kulkarni, L. Fei-Fei, S. Savarese, Y. Zhu, and R. Martín-Martín, "What Matters in Learning from Offline Human Demonstrations for Robot Manipulation," in *Conference on Robot Learning (CoRL)*, 8 2021.

[8] M. Vecerik, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothörl, T. Lampe, and M. Riedmiller, "Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards," *ArXiv*, vol. abs/1707.08817, 7 2017.

[9] A. Nair, B. McGrew, M. Andrychowicz, W. Zaremba, and P. Abbeel, “Overcoming Exploration in Reinforcement Learning with Demonstrations,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6292–6299, IEEE, 5 2018.

[10] M. A. Lee, Y. Zhu, K. Srinivasan, P. Shah, S. Savarese, L. Fei-Fei, A. Garg, and J. Bohg, “Making Sense of Vision and Touch: Self-Supervised Learning of Multimodal Representations for Contact-Rich Tasks,” in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 8943–8950, IEEE, 5 2019.

[11] Y. Chebotar, O. Kroemer, and J. Peters, “Learning robot tactile sensing for object manipulation,” *IEEE International Conference on Intelligent Robots and Systems*, pp. 3368–3375, 10 2014.

[12] Y. Chebotar, K. Hausman, Z. Su, G. S. Sukhatme, and S. Schaal, “Self-Supervised Regrasping using Spatio-Temporal Tactile Features and Reinforcement Learning,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1960–1966, 2016.

[13] Y. Chebotar, M. Kalakrishnan, A. Yahya, A. Li, S. Schaal, and S. Levine, “Path Integral Guided Policy Search,” in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3381–3388, 2017.

[14] T. Hester, O. Pietquin, M. Lanctot, T. Schaul, D. Horgan, J. Quan, A. Sendonaris, G. Dulac-Arnold, J. Agapiou, and J. Z. Leibo, “Deep Q-Learning from Demonstrations,” in *The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, 2018.

[15] J. Hwangbo, J. Lee, A. Dosovitskiy, D. Bellicoso, V. Tsounis, V. Koltun, and M. Hutter, “Learning agile and dynamic motor skills for legged robots,” *Science Robotics*, vol. 4, 1 2019.

[16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” *CoRR*, 7 2017.

[17] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust Region Policy Optimization,” *CoRR*, 2 2015.

[18] V. Tsounis, M. Alge, J. Lee, F. Farshidian, and M. Hutter, “DeepGait: Planning and Control of Quadrupedal Gaits using Deep Reinforcement Learning,” *IEEE Robotics and Automation Letters*, vol. 5, pp. 3699–3706, 9 2020.

[19] S. Levine and V. Koltun, “Guided Policy Search,” in *Proceedings of the 30th International Conference on Machine Learning*, 2013.

[20] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor,” *CoRR*, 1 2018.

[21] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, “Continuous control with deep reinforcement learning,” *CoRR*, 9 2015.

[22] S. Dankwa and W. Zheng, “Twin-Delayed DDPG: A Deep Reinforcement Learning Technique to Model a Continuous Movement of an Intelligent Robot Agent,” in *ACM International Conference Proceeding Series*, Association for Computing Machinery, 8 2019.

[23] S. Levine, A. Kumar, G. Tucker, and J. Fu, “Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems,” 5 2020.

[24] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, pp. 529–533, 2 2015.

[25] D. Silver, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic Policy Gradient Algorithms,” in *31st International Conference on Machine Learning, ICML 2014*, 2014.

[26] S. Fujimoto, H. van Hoof, and D. Meger, “Addressing Function Approximation Error in Actor-Critic Methods,” in *2018 International Conference on Machine Learning*, 2 2018.

[27] S. Thrun and A. Schwartz, “Issues in Using Function Approximation for Reinforcement Learning,” in *Proceedings of the 1993 Connectionist Models Summer School*, 1993.

[28] H. Van Hasselt, “Double Q-learning,” in *Advances in Neural Information Processing Systems*, 2010.

[29] H. van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-learning,” in *AAAI’16: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 9 2016.

[30] D. Wu, X. Dong, J. Shen, and S. C. Hoi, “Reducing Estimation Bias via Triplet-Average Deep Deterministic Policy Gradient,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 31, pp. 4933–4945, 11 2020.

[31] W. Wei, Y. Zhang, J. Liang, L. Li, and Y. Li, “Controlling Underestimation Bias in Reinforcement Learning via Quasi-median Operation,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 8621–8628, 2022.

[32] S. Li, Q. Tang, Y. Pang, X. Ma, and G. Wang, “Balancing Value Underestimation and Overestimation with Realistic Actor-Critic,” *CoRR*, 10 2022.

[33] A. Kumar, A. Zhou, G. Tucker, and S. Levine, “Conservative Q-Learning for Offline Reinforcement Learning,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 6 2020.

[34] H. Jiang, G. Li, J. Xie, and J. Yang, “Action Candidate Driven Clipped Double Q-Learning for Discrete and Continuous Action Tasks,” *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[35] B. Saglam, E. Duran, D. C. Cicek, F. B. Mutlu, and S. S. Kozat, “Estimation Error Correction in Deep Reinforcement Learning for Deterministic Actor-Critic Methods,” in *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, vol. 2021-November, pp. 137–144, IEEE Computer Society, 2021.

[36] A. Mandlekar, F. Ramos, B. Boots, S. Savarese, L. Fei-Fei, A. Garg, and D. Fox, “IRIS: Implicit Reinforcement without Interaction at Scale for Learning Control from Offline

Robot Manipulation Data," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4414–4420, IEEE, 5 2020.

[37] A. Mandlekar, J. Booher, M. Spero, A. Tung, A. Gupta, Y. Zhu, A. Garg, S. Savarese, and L. Fei-Fei, "Scaling Robot Supervision to Hundreds of Hours with RoboTurk: Robotic Manipulation Dataset through Human Reasoning and Dexterity," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1048–1055, IEEE, 11 2019.

[38] C. Gulcehre, Z. Wang, A. Novikov, T. L. Paine, S. G. Colmenarejo, K. Zolna, R. Agarwal, J. Merel, D. Mankowitz, C. Paduraru, G. Dulac-Arnold, J. Li, M. Norouzi, M. Hoffman, O. Nachum, G. Tucker, N. Heess, and N. de Freitas, "RL Unplugged: A Suite of Benchmarks for Offline Reinforcement Learning," in *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*, vol. 33, pp. 7248–7259, 6 2020.

[39] T. L. Paine, C. Paduraru, A. Michi, C. Gulcehre, K. Zolna, A. Novikov, Z. Wang, and N. de Freitas, "Hyperparameter Selection for Offline Reinforcement Learning," *ArXiv*, vol. abs/2007.09055, 7 2020.

[40] D. A. Pomerleau, "ALVINN: AN AUTONOMOUS LAND VEHICLE IN A NEURAL NETWORK," in *Advances in Neural Information Processing Systems 1 (NIPS 1988)*, 1988.

[41] A. Mandlekar, D. Xu, R. Martín-Martín, S. Savarese, and L. Fei-Fei, "Learning to Generalize Across Long-Horizon Tasks from Human Demonstrations," in *Robotics Science and Systems (RSS)*, 3 2020.

[42] S. Fujimoto, D. Meger, and D. Precup, "Off-Policy Deep Reinforcement Learning without Exploration," in *36th International Conference on Machine Learning*, 12 2019.

[43] J. Fu, A. Kumar, O. Nachum, G. Tucker, and S. Levine, "D4RL: Datasets for Deep Data-Driven Reinforcement Learning," *arXiv preprint arXiv:2004.07219*, 4 2020.

[44] Y. Zhu, J. Wong, A. Mandlekar, and R. Martín-Martín, "robosuite: A Modular Simulation Framework and Benchmark for Robot Learning," *CoRR*, vol. abs/2009.12293, 9 2020.

[45] F. Pardo, A. Tavakoli, V. Levdik, and P. Kormushev, "Time Limits in Reinforcement Learning," *CoRR*, 12 2017.

Glossary

List of Acronyms

DDPG	Deep Deterministic Policy Gradient
TD3	Twin-Delayed Deep Deterministic Policy Gradient
DQN	Deep Q-learning Network
SGD	stochastic gradient descent
DDPGfD	Deep Deterministic Policy Gradient from Demonstration
RL	Reinforcement Learning
TD	Temporal difference
MDP	Markov Decision Process
TADD	triplet-averaged deep deterministic policy gradient
QMD3	Quasi-Median Delayed Deep Deterministic Policy Gradient
RNN	recurrent neural network
BD-COACH	Batch Deep Corrective Advice Communicated by Humans
BTD3	Bounded Twin-Delayed Deep Deterministic Policy Gradient
UBTD3	Underestimated Bounded Twin-Delayed Deep Deterministic Policy Gradient

List of Symbols

α	Learning rate
γ	Discount factor
$\mathcal{A}$	Set of possible actions
$\mathcal{O}$	Set of observations (partially observable MDP)
$\mathcal{S}$	Set of possible states
μ_θ	Deterministic policy parametrized by θ

π	Policy, decision making rule
τ	Episode; trajectory of states and actions
a	Action
d_0	Initial state distribution
H	Time horizon of the reinforcement learning problem
o	Observation (partially observable MDP)
Q_ϕ	Q-function approximation using a neural network with ϕ parameter vector
R	Reward function
s	State
s'	Next state
t	Discrete time index