Karlsruhe Institute of Technology

Offline Reinforcement Learning in Autonomous Driving

Bachelor Thesis

by

Pascal Schindler

Degree Course: Industrial Engineering and Management B.Sc.

Matriculation Number: 1969739

Institute of Applied Informatics and Formal Description Methods (AIFB)

KIT Department of Economics and Management

Advisor:	Prof. Dr. J. Marius Zöllner
Second Advisor:	Prof. Dr. Andreas Oberweis
Supervisor:	M.Sc. Mohammd Karam Daaboul
Submitted:	September 13, 2021

Abstract

Current online reinforcement algorithms struggle to utilize large and diverse datasets. In contrast, offline reinforcement learning algorithms offer an efficient solution for this problem. This paves the way for data-driven reinforcement learning. With the help of offline reinforcement learning algorithms, it is now possible to apply reinforcement learning in costly environments such as healthcare or autonomous driving. For this reason, we tested one of the latest offline reinforcement learning algorithm, CQL, in the autonomous driving environments CarRacing-v0 and Carla. We evaluated the CQL performance on different datasets with different α values. The α value controls the conservatism of the algorithm. Thereby, we tested the hypothesis that higher α values perform better the better the dataset and lower α values perform better the worse the dataset. To this end, we created expert datasets with excellent trajectories and imperfect datasets with noisy trajectories. Furthermore, we evaluated the CQL performance in contrast to behavior cloning and the state-of-the-art online reinforcement learning algorithm SAC.

Contents

List of Abbreviations

ABS Anti-Lock Braking Systems.

AC Actor-Critic.

ACC Adaptive Cruise Control.

AI Artificial Intelligence.

ALVINN Autonomous Land Vehicle in A Neural Network.

AVs autonomous vehicle.

BC Behavior Cloning.

BEAR Bootstrapping Error Accumulation Reduction.

BRAC Behavioural Regularized Actor Critic.

CARLA CAR Learning to act.

CNNs Convolutional Neural Networks.

CQL Conservative Q-Learning.

DRL Deep Reinforcement Learning.

GPU Graphical Processing Unit.

LFD Learning from Demonstration.

LSTM Long Short-Term Memory.

MDP Markov Decision Process.

ML machine learning.

MMD Maximum Mean Discrepancy.

NNs Neural Networks.

OEMs original equipment manufacturers.

OOD Out-of-Distribution.

RL Reinforcement Learning.

RNNs Recurrent Neural Networks.

SAC Soft-Actor-Critic.

SQL Soft-Q-Learning.

TD Temporal Difference.

List of Figures

List of Tables

1 Introduction

1.1 Offline Reinforcement Learning in Autonomous Driving

Autonomously driving cars are considered to be the most disruptive technology of this century [16], which will profoundly impact the economy, mobility, and society [7]. Nowadays, the first concepts of autonomous vehicle (AVs) can be found on the streets. Car manufacturers already introduced autonomous driving level 3 or are on the transition to introduce autonomous driving level 4 [69]. Market research estimates that from 2030 to 2050 consumers will begin to accept AVs in the broad mass market, and from 2050 to 2060, AVs will become the primary means of transportation [7] [40]. In the next 40 years, there will be a fundamental change in the car market. The safety aspect of AVs is great hope and, at the same time, a profound argument for the introduction of autonomous vehicles. With the emergence of AVs on the road, the safety of road users will drastically increase. According to [55], approximately 90-95 % of all car accidents are caused by human driving errors. The causes for driving errors are distracted driving, drunken driving, or speeding. Autonomous driving is a promising solution to prevent these unnecessary accidents since it will take away human control over the vehicle [40]. Furthermore, AVs can provide more independent mobility for non-drivers or drivers who cannot safely participate in road traffic [17]. Additionally, on an economic level, more and more business models will emerge. In the future, vehicle unit sales will continue to grow, but with a lower growth rate. New trends will emerge, and the current one is heading towards shared mobility [13]. In all these concepts, the application of AVs is a vast business case. It is expected that mobility services such as car sharing, carpooling, or shuttle services will further increase their growth [61]. Moreover, existing business models like car insurance or maintaining services have to adapt to the new situation. For autonomous technology to be successful, it needs to overcome regulatory and ethical hurdles and the lack of standardized road infrastructure [11]. Key characteristics of an artificial intelligence system for autonomous driving are the ability to process massive amounts of constantly changing data, reason and draw inferences, learn based on historical patterns, and analyze and solve complex problems [57]. In short, the essential ability of an autonomous car is to generate a sequence of intelligent actions. As a part of machine learning, Reinforcement Learning (RL) is about optimizing actions in different situations and, therefore, is theoretically excellent for autonomous driving.

RL is a basic machine learning paradigm, where the agent learns to achieve a specific goal in an uncertain environment. Through rewards and penalties for the performed actions, the agent will be nudged into the desired behavior. The agent's goal is the maximization of its total reward. Increased computational power and further achievements in several areas of machine learning (ML) helped to accelerate the development of highly capable RL algorithms. Recent applications show that agents can perform complex tasks such as playing computer games on a human or even beyond human level [43], mastering coordination games, or simple tool usage [4]. Previous work from the latest years shows promising applications of RL in the area of autonomous driving. It was possible to demonstrate that RL agents can perform intelligent behavior in a highway setting and, in addition, can handle intersection problems [25] [42]. Even though these successful approaches were performed in a simulated environment and, therefore, not directly meaningful for a real-world application, more and more real-world applications are emerging. In 2019, an agent learned with only a few training episodes to follow a lane [29]. Moreover, only with a few hours of training, RL models performed sharp turns, controlled stops, or obstacle avoidance in the real world [17].

However, most RL methods learn in an *active* learning setting, which means an agent performs an action, observes the outcome of the action in the form of a reward, and adapts its behavior. This active interaction with the environment makes such RL methods hard to transfer to a real-world problem, where active interactions can be expensive or even unsafe [37]. For a practical application of RL into the real world, it is necessary to combine RL with data-driven machine learning. The utilization of large and diverse data sets is essential. In *offline* RL, data is collected once in advance and used to train an optimal policy without further online data collection [37]. Since no further interaction with the environment is needed, offline RL might be a solution to enable RL in autonomous driving. An AV could be trained on millions of pictures and videos, representing actual driving behavior and thereby learning to drive without being risky or unsafe [38]. Nevertheless, there are still huge problems, such as distributional, which must be overcome.

This thesis aims to derive a sample efficient and safe policy from a static data set generated from the CARLA simulator. The derived policy should be able to keep in lane. Furthermore, we will evaluate the effectiveness of the approach with respect to performance and sample efficiency and compare the results with state-of-the-art baseline algorithms such as Soft-Actor-Critic (SAC) and Behavior Cloning (BC).

1.2 Outline

The main body of this work is separated into four different chapters. Each chapter will focus on its particular subject.

2 Background and Related Work summarizes the basic techniques and related approaches we will explore in this work. In particular, it will give a brief overview of approaches to deep learning in autonomous driving and necessary system components. After that, a summary of the foundations of reinforcement learning and typical algorithms is given, followed by the main body of the work, offline reinforcement learning. Next, advanced topics such as policy constraint methods or learning lower-bounded policy-values methods will be addressed.

3 Approach discusses the problem faced by implementing online as well as offline algorithms. Explicitly, we address the input representation, input space, reward-function design, and network architecture. In addition, design choices will be expounded and justified.

4 Experiments are records of the experimental evaluation of the proposed approach. We analyze whether conservative q-learning can empirically lead to comparable good results or even succeed other approaches compared to state-of-the-art baseline algorithms.

5 Conclusion summarizes the results of the proposed approach and discusses its applicability as well as its limitations. Moreover, we will describe if offline reinforcement learning might be a promising approach to enable autonomous driving.

1.3 Notation

To the best of our abilities, we choose notations that follow the standard of related recent literature. Temporal dependencies such as the time step along trajectories are generally denoted with a subscript t. At times, next-step dependencies (i.e., t+1) may be replaced by an apostrophe (e.g., s') to reduce clutter. A summary of essential notations is given in table 1.1

Symbol	Meaning
a, a_t	Action and action taken at time step t
α_{temp}	Temperature factor SQL
α_{con}	Conservatism factor CQL
α_{learn}	Learning rate
$\mathcal{A}$	Action space (discrete or continuous) with $\mathbf{a} \in \mathcal{A}$
$\text{A}^{\pi}(s,a)$	Advantage function of state s and action a
$b(s)$, $b(s_t)$	State-dependent baseline function and state-dependent baseline function at time step t
$\mathcal{D}$	Set of transitions available for updating policy
$D_{KL}(p_1 \vert\vert p_2)$	*Kullback–Leibler* divergence between two probability distributions
D	Policy constraints
d_0	Initial state distribution $d_0(\mathbf{s}_0)$
$d^{\pi}(\mathbf{s})$	Overall state visitation frequency and state visitation frequency at time step t
ϵ	Random noise variable
$\mathbb{E}_{X \backsim P}$	Expected value of random variable X, sampled from distribution P
f	Placeholder for arbitrary functions
G_t	Total discounted reward from time step t
γ	Discounting factor for rewards
H	Maximum rollout horizon
$\mathcal{H}(x)$	Entropy of the random variable x
$J(\pi)$	Expected reward under trajectory distribution
$J(\theta)$	Loss function of network θ
$\mathcal{M}$	Tuple consisting of ($\mathcal{S}$, $\mathcal{A}$, T, d_0, r, γ)
$\mathcal{N}(\mu, \sigma^2)$	Gaussian distribution parametrized by mean μ and variance σ^2
∇_{θ}	Gradient w.r.t. θ
$\mathbf{o}$	Observation

$\mathcal{O}$	Observation space with $\mathbf{o} \in \mathcal{O}$
$p_\pi(\tau)$	Trajectory distribution
$P(s'\|s,a)$, $P(\mathbf{s}_{t+1}\|\mathbf{s}_t,\mathbf{a}_t)$	State transition probability for arbitrary state-action tuple and at particular time step
π, $\pi(\mathbf{a}_t\|\mathbf{s}_t)$	Stochastic distribution over actions conditioned on states
π, $\pi(\mathbf{a}_t\|\mathbf{o}_t)$	Stochastic distribution over actions conditioned on observations
π^*	Optimal policy
π_θ	Policy π with network parameters θ
π_β	Policy displayed in dataset
$Q^\pi(s,a)$	State-action value function
$Q^*_\pi(s,a)$	Optimal state-action value function
r, r_t	Reward and reward at time step t
R_t	Return at time step t
s, s_t	State and state at time step t
$\mathcal{S}$	State space (discrete or continuous) with $\mathbf{s} \in \mathcal{S}$
T	Conditional probability distribution of the form $T(\mathbf{s}_{t+1}\|\mathbf{s}_t,\mathbf{a}_t)$
τ	A trajectory of form $(\mathbf{s}_0, \mathbf{a}_0,...,\mathbf{s}_H, \mathbf{a}_H)$
θ, ψ, ϕ	Parameter vector, e.g. might denote weights of deep network
$\bar{\theta}, \hat{\theta}, \hat{\psi}, \bar{\psi}, \hat{\phi}, \bar{\phi}$	Target network parameters
$V^\pi(s)$	State value function
$V^*_\pi(s)$	Optimal state value function

Table 1.1: Notation table

2 Backround and Related Work

The following chapter gives an overview of the frameworks, concepts, and methods relevant to this work. Firstly, 2.1 gives a wrap-up about the current state-of-the-art deep learning methods used for autonomous driving. After that, the fundamental framework of RL is set out. Moreover, some basic algorithms of *online* RL will be introduced before addressing more advanced methods of *offline* RL. At last, Behavioural Cloning will be presented and its differentiation from RL. The introduced concepts are structured such that the derived ideas and mathematical descriptions follow a continuous development.

2.1 Deep Learning in Autonomous Driving

As already mentioned in 1.1, up to 95% of all traffic accidents are caused by human errors. To tackle rising fatal car accidents and meet the demand for advanced safety features in cars [65], original equipment manufacturers (OEMs) started developing better safety systems. As a result, we can distinguish between *passive* safety features such as airbags or seat belts and *active* safety features like Anti-Lock Braking Systems (ABS) or Adaptive Cruise Control (ACC). Active safety systems actively assist the driver to avoid accidents and directly intervene with the driving task [39].

The current state-of-the-art technologies, like the mentioned ABS, enable *automated driving*. These systems enhance the driving behavior by dedicated control of autonomous systems that support the driver while in control or enable to timely get back in control. In contrast to automated driving, *autonomous driving* is the extreme end result of automated driving. In principle, no human driver is needed to operate the vehicle [48]. Today, it is common to distinguish between five different levels of *autonomous driving*, from level 0 with no automation to level 5 with full automation [23].

Firstly, to create an autonomously acting car, it needs sensory impressions. In general, the car observes the environment through its sensors (e.g., several cameras, LiDAR sensors, or radar sensors) and thus creates its image of the environment. In the past, there has been a long-running debate about which technology to use. Cameras are cost-efficient but lack depth perception, cannot work in the dark and are sensitive to bad weather. In contrast, LiDAR sensors have a higher resolution and deliver a precise perception of the environment, but are very costly and are vulnerable to bad weather conditions [5]. Different OEMs decided to follow different approaches in the number of sensors and the usage of the available technology. Nevertheless, all OEMs are connected to the fact that they have to create solutions in the respective disciplines **Vision**, **Perception** and **Planning** as seen in 2.1 [53].

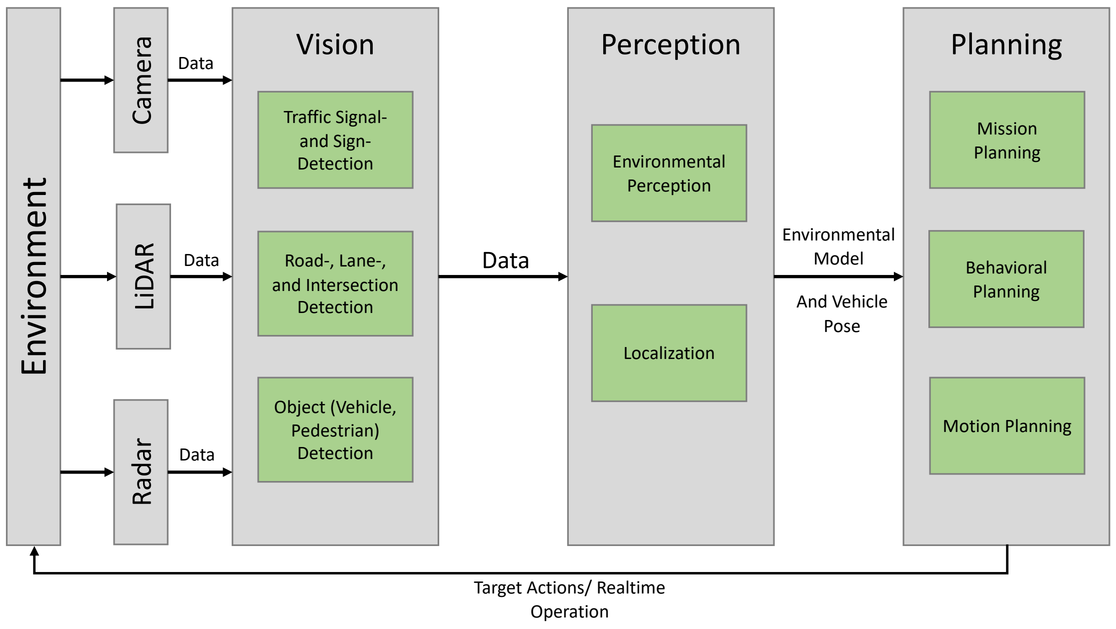


Figure 2.1: A typical autonomous vehicle system overview with its three main components "Vision", "Perception" and "Planning". This figure is based on [6].

Vision refers to the process of data collection and data processing. For a reliable autonomous system, an understanding of the scene is necessary. The environment is scanned with the help of several sensors to get as much information as possible. Modern sensor architectures include cameras, radar, and LiDAR sensors. In particular, *vision* is responsible for detecting all relevant objects by filtering the incoming camera images and LiDAR or radar data.

Perception stands for the ability to extract information and knowledge from previous condensed data. *Environmental Perception* refers to the process of forming an understanding of the environment, e.g., the location of objects or obstacles, driveable zones or location of pedestrians. *Localization* stands for the ability to determine its own location and to measure its motion within the environment. Here, an intermediate representation of the environment is formed and later used for the decision process in *planning* [30].

Planning includes the decision process of the car fulfilling its task. In the context of AVs, the transportation from one location to another location. *Mission Planning* refers to the planning from the starting point to the destination point on a map level. *Behavioral Planning* is responsible for making decisions along the route and ensuring that the ego vehicle follows the road rules safely [48] [54]. *Motion Planning* describes the planning of actions in a closed environment, such as overtaking maneuvers or lane changing.

The first time Machine Learning (ML) and Artificial Intelligence (AI) was acknowledged as a central component of autonomous driving was the DARPA Grand Challenge in 2005 [51].

Here, the winner used ML techniques to navigate through the environment. With this breakthrough, companies worldwide started to work on the introduction of deep learning techniques in cars. For now, autonomous driving components, such as environmental perception, localization, or mission planning, can either be designed with a deep learning or non-learning approach. Currently, non-learning approaches are still predominantly. One reason for the hesitation to integrate learning algorithms into the car is the lack of trust in such techniques [59].

Despite the significant challenges and hurdles integrating deep learning into cars, it is clear that sooner or later, these techniques will be applied in the real world. The most common deep learning techniques are Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs) and Deep Reinforcement Learning (DRL).

CNNs are one of the most popular deep learning techniques so far. Its name is derived from the mathematical operation between two layers of the network. CNNs show excellent performance with the processing of images [2]. Tasks such as obstacle detection, scene recognition, or lane recognition are suited for this deep learning architecture [47].

RNNs are excellent in understanding the time dynamics of an environment. RNNs maintain and store relevant information from the past and create a dependency between past states and current states [58]. A particular form of the RNN is the Long Short-Term Memory (LSTM), which tries to learn long-term dependencies. For a vehicle to understand the dynamics of a changing environment, it is crucial to take the past into account, which makes RNNs or LSTMs very suitable [47].

DRL combines RL and deep learning. Since driving is, in principle, a decision-making task, RL in combination with deep learning makes DRL qualified to enable autonomous driving. Several papers showed the success of DRL with tasks such as lane-keeping, lane changing or overtaking [31] [47].

In figure 2.2, the different deep learning methods are matched to the potential application areas. Not all methods are equally suited for the application areas.

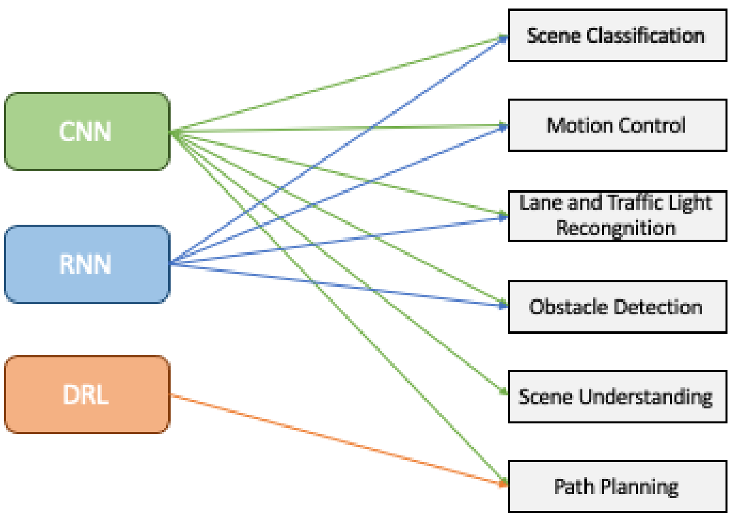


Figure 2.2: Deep Learning methods matched to the application ares in self-driving cars. This figure is based on [47].

Overall, deep learning is becoming more and more prominent in the car market and is a crucial element to enable autonomous driving. As described in this chapter, there are many different design choices for the hardware and ML technique used. Furthermore, there will not be a single deep learning technique that will solve the entire driving process. It will be more of an interplay of different deep learning techniques.

2.2 Reinforcement Learning

Reinforcement Learning (RL) refers to various approaches, where an agent learns to achieve a specific goal through interaction with its environment without any supervision. By trial and error and receiving rewards and penalties, the agent comes up with a solution to a given problem. Human interaction is limited to the adaption of rewards and penalties. A potential application of RL might be autonomous driving.

2.2.1 Markov Decision Process

Normally, a Markov Decision Process (MDP) is a framework used to describe an environment for RL. MDPs are designated to frame the problem of learning from interaction with the environment to achieve a goal. On a fundamental level, a MDP consists of two entities, an *agent* and an *environment*. The agent interacts with the environment by selecting actions, and the environment responds by presenting a new situation to the agent and rewarding him. This interplay is illustrated in 2.3. The goal of the agent is the maximization of the cumulated rewards.

The state, action, the following state, and reward at the discrete time step $t \in \{1,2,3,...\}$

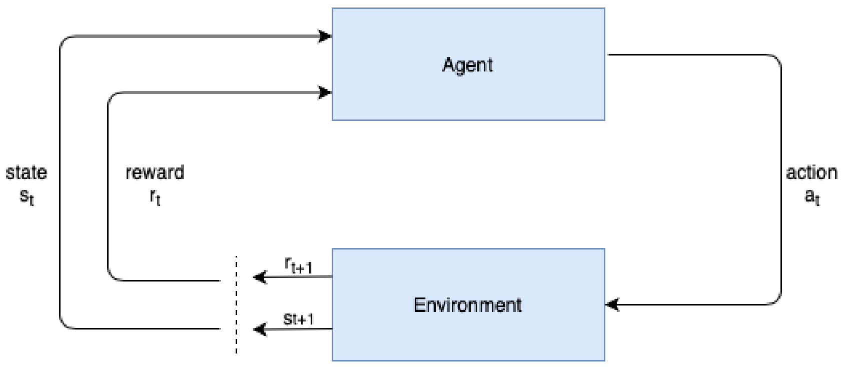


Figure 2.3: The cyclic process of a RL system containing an agent and the environment interacting with each other. This figure is based on [67].

are denoted as $s_t \in \mathcal{S}$, $a_t \in \mathcal{A}$, $s_{t+1} \in \mathcal{S}$ and r_t: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathbb{R}$, where $\mathcal{S}$ and $\mathcal{A}$ refer to a discrete or continuous state-/action space. Usually, the reward function maps state s_t, action a_t and successor state s_{t+1} to a real-valued reward. A fundamental property of MDPs is the *Markov Property*, which states: "The future is independent of the past given the present.". This means that successor state s_{t+1} and received reward r_t only depends on preceding state s_t and preceding action a_t and not on states or actions that are further in the past. This allows the following compact representation of the state transition probability function, which is a probability distribution over the next possible successor states [67]:

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_1, ..., s_t, a_1, ..., a_t). \tag{2.1}$$

The transition function without actions is put into matrix form, where each row sums to 1 [8]. With discrete states, the transition function describes the probability to transition from one state to another state.

$$P = \begin{bmatrix} P_{11} & ... & P_{1n} \\ \vdots & & \vdots \\ P_{n1} & ... & P_{nn} \end{bmatrix}$$

A so-called *policy* is introduced to model the agent's behavior, which is a mapping from states to actions. Therefore, the policy π: $\mathcal{S} \rightarrow \mathcal{A}$ works as a state-dependent selection function of choosing actions given a state in the form

$$\pi(a|s) = P(a_t = a|s_t = s), \qquad \pi : \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]. \tag{2.2}$$

Furthermore, together with the policy, it is now possible to describe the dynamics of a MDP formally as follows: starting in initial state s_0 and choosing $a_0 \in \mathcal{A}$ according to our policy. As a result, the state of the MDP transits from s_0 to s_1, drawn from $P^{a_0}_{s_0 s_1}$.

Then, from s_1, another action a_1 according to the learned policy is picked and again, a transition to a successor state s_2 is performed, drawn from $P_{s_1 s_2}^{a_1}$. The process continues with this scheme and is illustrated as follows:

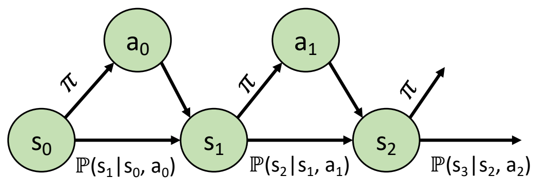


Figure 2.4: Schematic representation of the dynamics MDP

2.2.2 The Reinforcement Learning Goal

As already mentioned, the the agent's goal is to choose better actions over time to maximize the expected value of the *return*. To define the return, the introduction of the reward function R is needed, which depends on the current state s_t, the action a_t and the successor state s_{t+1} [49]:

$$r_t = R(s_t, a_t, s_{t+1}). \tag{2.3}$$

In the following, this might be simplified to just a dependence on the current state $r_t = R(s_t)$ or state-action pair $r_t = R(s_t, a_t)$.

Now, the *return* G_t as the total discounted reward from time step t with discount factor γ is defined as follows:

$$G_t = r_{t+1} + \gamma r_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \quad \gamma \in [0, 1]. \tag{2.4}$$

A sequence of consecutively states and actions yield in a trajectory τ of a specific finite horizon length T. The rewards collected along the trajectory τ form $R(\tau)$. This can be written in the following form:

$$\tau = (s_0, a_0, s_1, a_1, s_2, a_2, ...) \tag{2.5}$$

$$R(\tau) = \sum_{t=0}^{T} r_t \tag{2.6}$$

or with infinite horizon:

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t. \tag{2.7}$$

Most commonly, the discounted future rewards are interpreted as the current value of the future rewards. The discount factor determines how much the agent cares about the rewards far in the future relative to those in the immediate future. With $\gamma = 0$, the agent will be "myopic" and only cares about actions that give an immediate reward. With $\gamma = 1$, the agent will be "far-sighted" and takes actions based on the sum of all future rewards [50]. Reasons for the introduction of discount factor values are [3]:

- Avoids $G_t = \infty$ when trajectory lengths is infinity
- Models human behavior, which shows a preference for immediate reward
- Models uncertainty about future rewards

The goal of RL is to learn a policy that maximizes the **expected** return along the trajectory. With a stochastic environment transition function and policy, the probability of a specific trajectory with horizon T is:

$$P(\tau|\pi) = \underbrace{d_0(s_0)}_{\substack{\text{initial}\\\text{state}\\\text{distribution}}} \prod_{t=0}^{T-1} \underbrace{P(s_{t+1}|s_t, a_t)}_{\substack{\text{transition prob.}\\\text{to } \mathrm{s_t}}} \underbrace{\pi(a_t|s_t)}_{\substack{\text{prob. to}\\\text{take action}\\\mathrm{a_t} \text{ in } \mathrm{s_t}}} \tag{2.8}$$

and the expected return, denoted with $J(\pi)$, is:

$$J(\pi) = \int_{\tau} P(\tau|\pi) R(\tau) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)]. \tag{2.9}$$

The central optimization problem can be written as follows:

$$\pi^* = \arg\max_{\pi} J(\pi) \tag{2.10}$$

with π^* as the **optimal policy**.

2.2.3 Value Functions and Bellman Equations

Since the goal of RL is to derive an optimal policy, a measurement for optimality is needed. A helpful way is to determine the **value** of a state or state-action pair. By value, we mean the expected return of starting in a particular state or state-action pair and follow the policy afterward. Therefore an optimal policy will lead to the highest return.

To express the value of a state, the **On-Policy Value Function** is used, which gives the expected return when starting in s and following the policy afterward:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] = \mathop{\mathbb{E}}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t|s_0 = s\right]. \tag{2.11}$$

In addition to this, it is possible to express the value of a state-action pair with the **On-Policy State-Action-Value Function** also called **Q-function**, which gives the expected return when starting in s and taking action a and following the policy afterwards:

$$Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a] = r(s_0, a_0) + \gamma \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_1 = s']. \tag{2.12}$$

By replacing the previous policy π with the optimal policy π^* in 2.11 and 2.12, the **Optimal Value Function** and the **Optimal State-Action-Value Function** can be derived. This means that instead of following an arbitrary policy, an optimal policy is used, which leads to the following equations:

$$V^*(s) = \max_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] \tag{2.13}$$

and

$$Q^*(s,a) = \max_{\pi} \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s, a_0 = a]. \tag{2.14}$$

Since the computation of the state value function and the state-action value function can be quite resource-intensive. The so-called **Bellman Equation** helps to break down the functions into simpler, recursive sub-problems.

Now, it is possible to write the state value function in the following form:

$$\begin{aligned}
V^{\pi}(s) &= \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_0 = s] \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t r_t|s_t = s\right] \\
&= \mathop{\mathbb{E}}_{\tau\sim\pi}\left[r_t + \gamma\sum_{t=0}^{\infty}\gamma^{t+1} r_{t+1}|s_t = s\right] \\
&= \sum_a \pi(a|s)\sum_{s'} P(s'|s,a)\sum_r P(r|s,a,s')\left[r + \gamma\mathop{\mathbb{E}}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^{t+1} r_{t+1}|s_{t+1} = s'\right]\right] \\
&= \sum_a \pi(a|s)\sum_{s'} P(s'|s,a)\sum_r P(r|s,a,s')[r + \gamma V_{\pi}(s')] \\
&= \mathop{\mathbb{E}}_{\substack{a\sim\pi(a|s)\\ s'\sim P(s'|s,a)\\ r\sim P(r|s,a)}}[r + \gamma V_{\pi}(s')].
\end{aligned} \tag{2.15}$$

With similar procedure, the bellman equation for the state-action value function:

$$Q^{\pi}(s_t, a_t) = \mathop{\mathbb{E}}_{\substack{s'\sim P(s'|s,a)\\ r\sim P(r|s,a,s')}}[r + \gamma\mathop{\mathbb{E}}_{a'\sim\pi}[Q_{\pi}(s_{t+1}, a_{t+1})]]. \tag{2.16}$$

Furthermore, since $V^*(s)$ is the maximum expected total reward when starting in state s, it will be the maximum of $Q^*(s,a)$ over all possible Q^* value of other actions in the state s. With this, the following connection can be established:

$$V^*(s) = \max_a Q^*(s,a) \qquad \forall s \in S \tag{2.17}$$

with the optimal policy:

$$\pi^*(s) = \arg\max_a Q^*(s,a) \qquad \forall s \in S. \tag{2.18}$$

2.3 Off-Policy Reinforcement Learning

On a high level, all RL algorithms can be mapped according to the following overview:

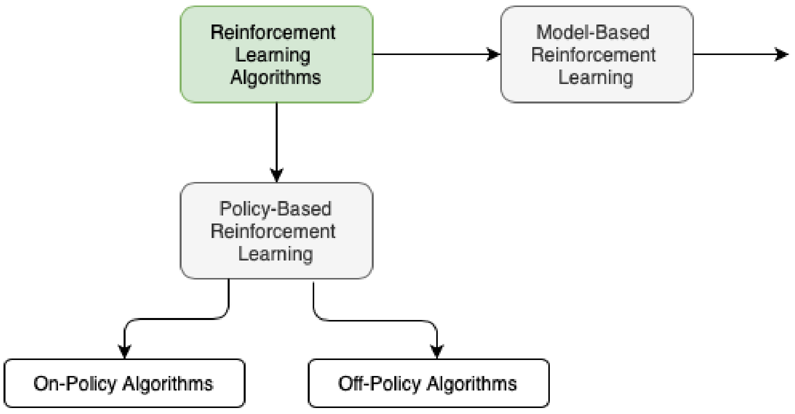

Reinforcement Learning Algorithms
Model-Based Reinforcement Learning
Policy-Based Reinforcement Learning
On-Policy Algorithms
Off-Policy Algorithms

Figure 2.5: Differentiation of the various RL algorithms

In general, it can be differed between two types of policies [66]

1. Target Policy
2. Behavior Policy.

The so-called target policy is the policy the agent tries to learn, whereas the behavior policy is the actual policy in use. Furthermore, it can be distinguished between the learning processes *On-Policy Learning* and *Off-Policy Learning*.

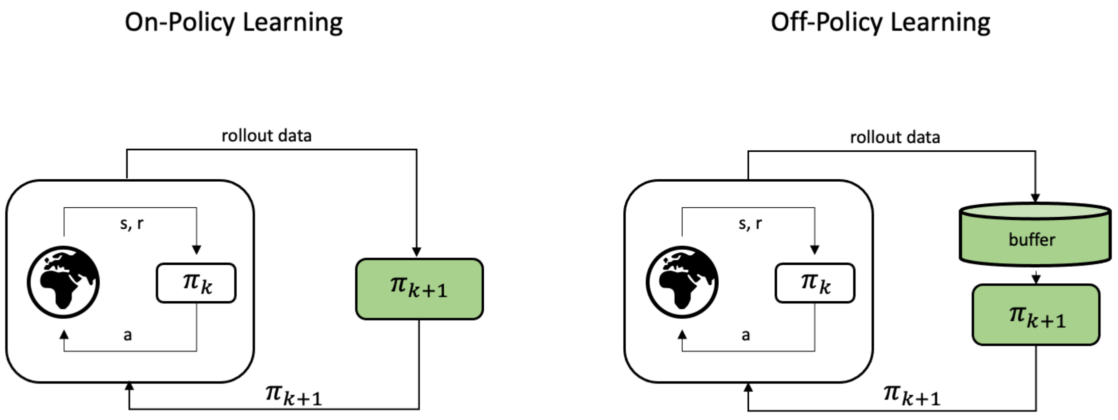


Figure 2.6: Difference between On-Policy Learning and Off-Policy Learning

On-policy learning refers to the concept that the algorithm evaluates and improves the same policy π_k used for the action selection process as seen in Fig. 2.6. The is no difference between the policy used for the action selection and the policy which will be improved. In short, the behavior policy corresponds with the target policy. Known algorithm examples are Policy Iteration, Value Iteration, or Sarsa.

On the other hand, there is off-policy learning which refers to approaches where the policy for evaluation and improvement differs from the policy which is designated to take actions. In short, target policy and behavior policy differ from one another. Here, transitions are stored in a buffer used to learn π_{k+1} while π_k still performs actions. After a specific number of iterations, an update of π_k with π_{k+1}, derived from the buffer, is performed.

The advantage of off-policy methods might be a faster learning process since the knowledge of several policies is stored in the replay buffer [66].

2.3.1 Q-Learning and DQN

Q-learning is a value-based off-policy RL algorithm that uses *Temporal Difference* Learning for the action-value function estimation. Temporal Difference (TD) learning describes a method to estimate the optimal value-function $V^*(s)$ or action-value function $Q(s, a)$. The principle of TD learning is to perform rollouts by using the current policy π. Values are determined by averaging the received returns starting in state s and taking action a. TD methods do not perform a whole episode to approximate the value. Instead, they bootstrap the update by building up on an existing estimation. The TD(0) method is given by:

$$V_\pi(s_t) \leftarrow V_\pi(s_t) + \alpha_{\text{learn}}[r_{t+1} + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)] \tag{2.19}$$

with α as update step-size.

The formula for iterative estimating the Q-values $Q(s_t, a_t)$ is the following [28]:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_{\text{learn}}}_{\text{learning rate}} \cdot \overbrace{[\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\substack{\text{discount}\\ \text{factor}}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\substack{\text{estimate of optimal}\\ \text{future value}}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t.a_t)}_{\text{old value}}]}^{\text{Temporal Difference Error}} \tag{2.20}$$

There are many ways to choose action a for exploration in Q-Learning, but the most common one is the ϵ-greedy strategy:

$$\text{action } a_t = \begin{cases} \arg\max Q_t(a), & \text{with prob. } 1 - \epsilon \\ \text{any action } a, & \text{with prob. } \epsilon \end{cases} \tag{2.21}$$

In this strategy, the action a is selected *greedily* with probability $1 - \epsilon$, $\epsilon \in (0, 1)$. In general, a little randomness is desired to promote exploration of the state space.

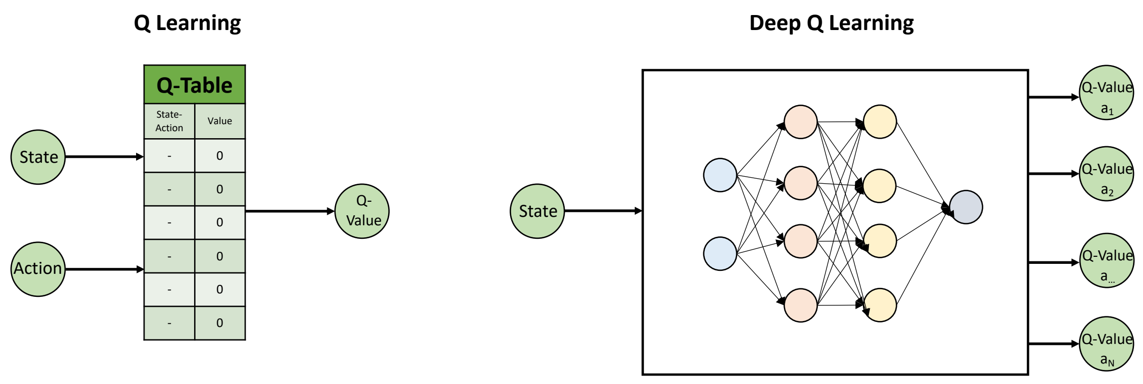


Figure 2.7: Q learning compared to Deep Q Learning

For large state and action spaces such as images or continuous action spaces, the normal Q-Learning quickly reaches its limits. Estimating Q-values with a tabular approach such as Q-tables is infeasible. For the first time, Mnih et al. [43] introduced Neural Networks (NNs) to approximate the action-value function as a *Q-network*. In the paper, a discrete action state was used. The network with the weights θ approximated all Q-values for every different action, as seen in 2.7 on the right.

During the training phase of the Q-network, the following loss function is iteratively minimized:

$$J(\theta) = \underbrace{[(r + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta^{target})}_{\text{Target Q Value}} - \underbrace{Q(s_t, a_t; \theta^{pred})]^2}_{\text{Predicted Q Value}}. \tag{2.22}$$

Hereby, the loss function optimization is done with *stochastic gradient descent* (SGD). The gradient of the loss function can be derived as follows:

$$\nabla_\theta J(\theta) = [(r + \gamma \max Q(s_{t+1}, a_{t+1}; \theta_{i-1})) - Q(s_t, a_t; \theta_i) \nabla_\theta Q(s_t, a_t; \theta_i)]. \tag{2.23}$$

Furthermore, the DQN algorithms utilize experience replay by storing transitions $e_t = (s_t, a_t, r_t, s_{t+1})$ in replay buffer $D = e_1, ..., e_N$. Thereby, a more stable data set is created and, through the random samples, the data is closer to independent and identically distributed [24]. By sampling mini-batches from the replay buffer, the Q-values are updated.

2.3.2 Soft-Q-Learning (SQL)

For the introduction of Soft-Q-Learning (SQL), some intuition is needed before. Generally, in RL, there is a fundamental trade-off between exploration and exploitation. Through exploration, it is possible to escape local optimums and find better solutions. However, too much exploration will not lead to optimal behavior in an appropriate amount of time [10]. This means, agents have to decide between the urge to acquire new knowledge of the environment through exploration and the urge to maximize their reward through exploiting the already gathered knowledge about the environment.

In RL, to measure the randomness of actions of an agent, the *entropy* of a probability distribution is used. Entropy directly relates to the unpredictability of actions and the greater the entropy, the more random actions are taken by the agent. Often and desired, the learning process in RL leads to a decreasing entropy of the action selection due to the convergence to optimal behavior, which has less random behavior [26]. Therefore, entropy can be a tool used to prevent a too fast convergence of a policy by promoting random actions to enhance the awareness of alternatives [68].

A helpful illustration is the following:

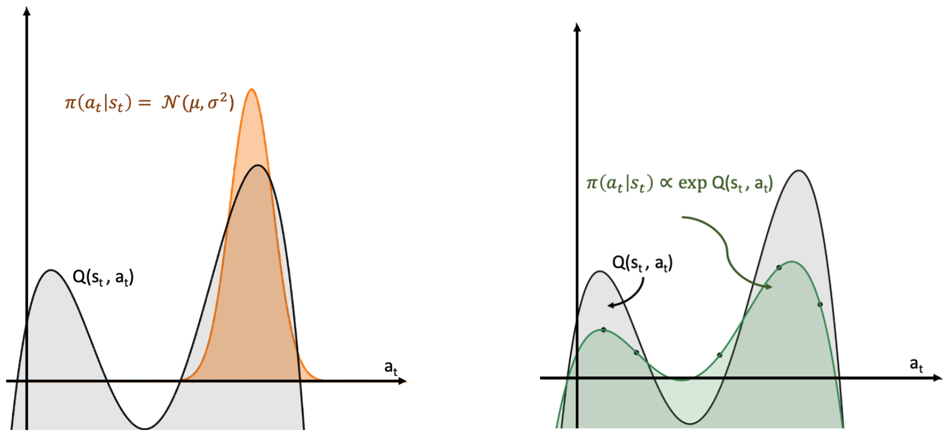


Figure 2.8: Illustration of taking the exponential of the Q-function (on the right) instead of the maximum of the Q-function (on the left). This figure is based on [68].

Again, by taking equation 2.22 into account, in DQN, the maximum of the Q-function is chosen: see 2.8 on the right. A disadvantage of this strategy is the inflexibility of the learned policy in altered environmental situations. A helpful picture is to imagine an agent in a maze with two alternative paths to a goal represented by the grey curve in 2.8. The maximization of the Q-function leads to an ignoring of the alternative path. Whereas a formalization of the policy as an exponential of the Q-function (2.8 on the right) helps the policy to consider other paths [68]:

$$\pi(a|s) \propto \exp Q(s,a) \tag{2.24}$$

The maximum-entropy form of the RL objective in formula 2.10 is:

$$\pi^*_{\text{MaxEnt}} = \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{T} r_t + \mathcal{H}(\pi(\cdot|s_t))\right]. \tag{2.25}$$

The idea behind the entropy formulation of the RL objective is to construct algorithms, where the algorithms learn to receive the highest sum of reward and entropy. Using entropy forces the agent to search for a distribution with maximum entropy, enabling exploration and preventing convergence against local maxima. Therefore, we tackle the exploration-/exploitation-problem with an entropy regularization with an adaptive parameter α_{temp}, called temperature, [41] and come to the slightly different formulation:

$$\pi^*_{\text{MaxEnt}} = \arg\max_{\pi} \mathbb{E}_{\pi}\left[\sum_{t=0}^{T} r_t + \alpha_{\text{temp}}\mathcal{H}(\pi(\cdot|s_t))\right]. \tag{2.26}$$

The temperature parameter is needed to promote exploration. With large α_{temp} encouraging exploration and small α_{temp} reducing exploration. If $\alpha = 0$, the old RL objective of 2.10 is received. Now, in the entropy-regularized framework, we need a slight redefinition of the Q-function to [22]:

$$Q^*_{\text{soft}}(s_t,a_t) = r_t + \mathbb{E}_{(s_{t+1},\ldots)\sim\pi}\left[\sum_{l=1}^{\infty} \gamma^l (r_{t+l} + \alpha_{\text{temp}}\mathcal{H}(\pi^*_{\text{MaxEnt}}(\cdot|s_{t+l})))\right]. \tag{2.27}$$

Whereas the soft value function is given by:

$$V^*_{\text{soft}}(s_t) = \alpha_{\text{temp}} \log \int_{\mathcal{A}} \exp\left(\frac{1}{\alpha_{\text{temp}}} Q^*_{\text{soft}}(s_t,a')\right)da'. \tag{2.28}$$

Furthermore, we train the Q-function with the minimization of the soft Bellman residual:

$$J_Q(\theta) = \mathbb{E}_{s_t,a_t\sim\pi}\left[\frac{1}{2}(\hat{Q}^{\bar{\theta}}_{\text{soft}}(s_t,a_t) - Q^{\theta}_{\text{soft}}(s_t,a_t)^2\right]. \tag{2.29}$$

Hereby, $\hat{Q}^{\bar{\theta}}_{\text{soft}}(s_t,a_t) = r_t + \gamma\mathbb{E}_{s_{t+1}\sim\pi}[V^{\bar{\theta}}_{\text{soft}}(s_{t+1})]$ is the target Q-value with $\bar{\theta}$ as the target parameters.

Advantages of SQL compared to Q-learning is better exploration, policy transfer between similar tasks and a better robustness in terms of adaption to changed environmental situations [28] [68].

2.3.3 Soft-Actor-Critic Algorithm (SAC)

First of all, Actor-Critic (AC) algorithms must be introduced before explaining the SAC algorithm. As well as SQL, AC algorithms can handle continuous action-space and continuous state-space.

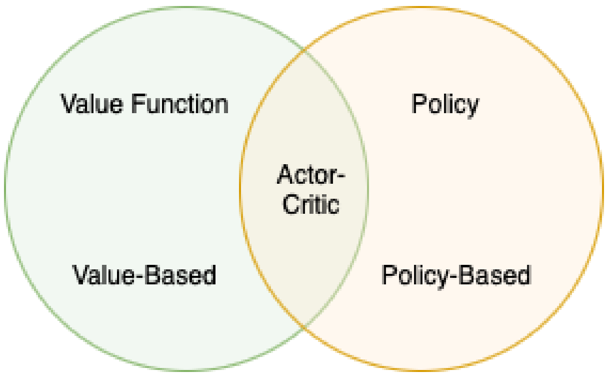


Figure 2.9: The connection between the two policy optimization approaches: Value-based optimization and policy-based optimization. The hybrid between those two approaches are actor-critic approaches. This figure is based on [14].

In general, AC algorithms work with two different structures. The first structure represents the *actor*, which is responsible for the action selection. The second structure, called *critic*, is in charge of evaluating the actions taken by the actor [63]. The actor takes the state from the environment as input and delivers the current best action according to its current knowledge as output. The goal of the actor is to learn an optimal policy π. Whereas the critic approximates the value function to evaluate the actions taken by the actor [27]. This behavior is displayed in the following figure:

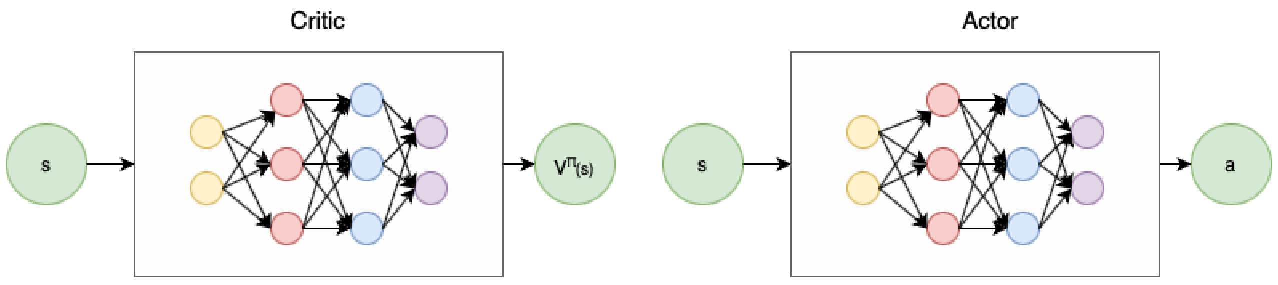


Figure 2.10: Two network design with the actor network (on the right) and critic network (on the left) [37].

The general structure of every actor-critic algorithm can be seen in the following illustration, which shows the loop between the actor-critic structure of figure 2.10 and the environment.

The SAC algorithm uses neural network function approximators for $V_\psi(s_t)$, $Q_\theta(s_t, a_t)$ and

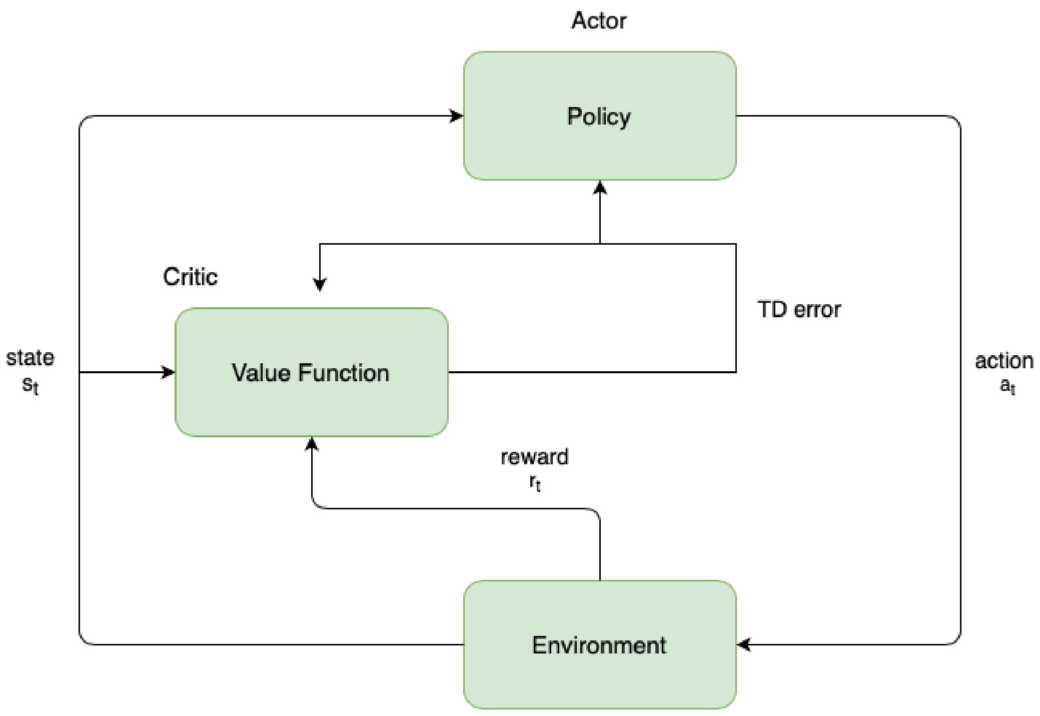


Figure 2.11: The general interaction between the actor-critic structure and the environment with the policy as actor and the value function as critic. The critic evaluates the environment feedback in form of reward r_t with a TD error. The figure is based on [67].

the policy $\pi_\phi(a_t|s_t)$ with the respective network parameters ψ, θ and ϕ. The state-value function $V_\psi(s_t)$ is trained through the minimization of the squared residual error:

$$J_V(\psi) = \mathbb{E}_{s_t \sim D}\left[\frac{1}{2}(V_\psi(s_t) - \mathbb{E}_{a_t \sim \pi_\phi}[Q_\theta(s_t, a_t) - \log \pi_\phi(a_t|s_t)])^2\right]. \tag{2.30}$$

The states are sampled from replay buffer D. The gradient of equation 2.30 can be approximated by:

$$\hat{\nabla}_\psi J_V(\psi) = \nabla_\psi V_\psi(s_t)(V_\psi(s_t) - Q_\theta(s_t, a_t) + \log \pi_\phi(a_t|s_t)), \tag{2.31}$$

with the actions sampled from the current policy. By minimizing the soft Bellman residual, it is possible to train the soft Q-function:

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim D}\left[\frac{1}{2}(Q_\theta(s_t, a_t) - \hat{Q}_\theta(s_t, a_t))^2\right], \tag{2.32}$$

with the target network $\hat{Q}_\theta(s_t, a_t))$ as:

$$\hat{Q}_\theta(s_t, a_t)) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1}\sim\pi}[V_{\bar{\psi}}(s_{t+1})]. \tag{2.33}$$

and the target parameters of the value function as $\bar{\psi}$ .The stochastic gradient of the loss function $J_Q(\theta)$ is given by:

$$\hat{\nabla} J_Q(\theta) = \nabla_\theta Q_\theta(s_t, a_t)(Q_\theta(s_t, a_t) - r(s_t, a_t) - \gamma V_{\bar{\psi}}(s_{t+1})). \tag{2.34}$$

The parameters of the target network $\bar{\psi}$ are an moving average of the value parameters ψ. Furthermore, the policy is learned through the minimization of the KL-divergence:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim D}\left[\mathrm{D}_{\mathrm{KL}}\left(\pi_\phi(\cdot|s_t) \middle\| \frac{\exp\left(Q_\phi(s_t, \cdot)\right)}{Z_\phi(s_t)}\right)\right]. \tag{2.35}$$

Since the function $Z_\phi(s_t)$ does not show up in the gradient, it can be ignored. For the optimization of $J_\pi(\phi)$, the policy is reparametrized such that samples are drawn with:

$$a_t = f_\phi(\epsilon_t, s_t), \tag{2.36}$$

with $f_\phi(\epsilon_t, s_t)$ as a function of state s_t, independent noise ϵ and policy parameters ϕ. Because of the reparametrizatuin, equation 2.35 can be written as follows:

$$J_\pi(\phi) = \mathbb{E}_{\substack{\epsilon_t \sim N \\ s_t \sim D}}[\log \pi_\phi(f_\phi(\epsilon_t, s_t)|s_t) - Q_\phi(s_t, f_\phi(\epsilon_t, s_t))], \tag{2.37}$$

and the gradient of the loss function $J_\pi(\phi)$ as:

$$\hat{\nabla}_\phi J_\pi(\phi) = \nabla_\phi \log \pi_\phi(a_t|s_t) + (\nabla_{a_t} \log \pi_\phi(a_t|s_t) - \nabla_{a_t} Q(s_t, a_t))\nabla_\phi f_\phi(\epsilon_t, s_t). \tag{2.38}$$

Hereby, a_t is evaluated by $f_\phi(\epsilon_t, s_t)$ and ϵ is sampled from a fixed distribution. Furthermore, the SAC algorithm trains two Q-functions independently and then uses the minimum of the Q-functions in the following way:

$$J_\pi(\phi) = \mathbb{E}_{\substack{\epsilon_t \sim N \\ s_t \sim D}}\left[\log \pi_\phi(f_\phi(\epsilon_t, s_t)|s_t) - \min_{j=1,2} Q_\phi(s_t, f_\phi(\epsilon_t, s_t))\right]. \tag{2.39}$$

Samples are collected from the environment and the gradients are calculated from the replay buffer what makes SAC an off-policy algorithm.

2.4 Offline Reinforcement Learning

The algorithms of the previous sections are all *online* RL algorithms. In their respective framework, they either update their policy after every step or update the policy with trajectories from a buffer [67]. So far, the significant disadvantage is that we can not apply RL in a wide variety of fields where online interaction might be harmful [38]. In the context of autonomous driving, online learning means the car is learning during the actual driving process on the street, which is error-prone and risky. An alternative to the training in real life is the usage of high-fidelity simulators. Unfortunately, these are challenging to build. Furthermore, the transition from a simulator to real life is not apparent since good performance in the simulators does not imply good performance in the real world [52]. Nevertheless, Osinski et al. [52] and Capasso et. al. [12] were able to show a fundamental transfer of driving maneuvers from simulation to real-world.

Another promising application area for offline RL despite autonomous driving exists in health care. Since we can interpret the treatment in health scenarios as a sequence of decision-making, it makes the RL framework, as long term decision-making optimizer, suitable for treatment recommendations. Moreover, as in autonomous driving, the real-world application must be considered very carefully because human life is involved. In addition, the selection of the underlying data must be taken very carefully since artifacts might lead to harmful decisions. An illustrating example is that from a more aggressive treatment of sicker patients, which comes along with higher mortality, the agent could conclude the inappropriateness of the treatment even though the mortality comes from the illness and not from the treatment [21]. Furthermore, modeling the reward function and action space is also very error-prone and needs a careful evaluation.

Much of the success of machine learning is based on the use of large amounts of data. So far, in online RL, the advantages of large amounts of data are not used. In the case of off-policy algorithms, data is utilized in the sense of replay buffers but is not comparable with the usage of big data sets like in supervised learning. The goal is to transfer this *data-driven* learning methods into RL, which results in *offline* RL [38]. Firstly, this might help improve RL's performance, and secondly, this solves the problem with the risky online interaction since the agent learns from the given data. Several frameworks emerged, e.g., in pure offline RL, data is collected once in advance with any given policy, and based on the collected data, a policy is derived which will be deployed in the environment (2.12 left). Another variation is the usage of offline RL as a basis to derive a policy that will be fine-tuned afterward online (2.12 right) [46].

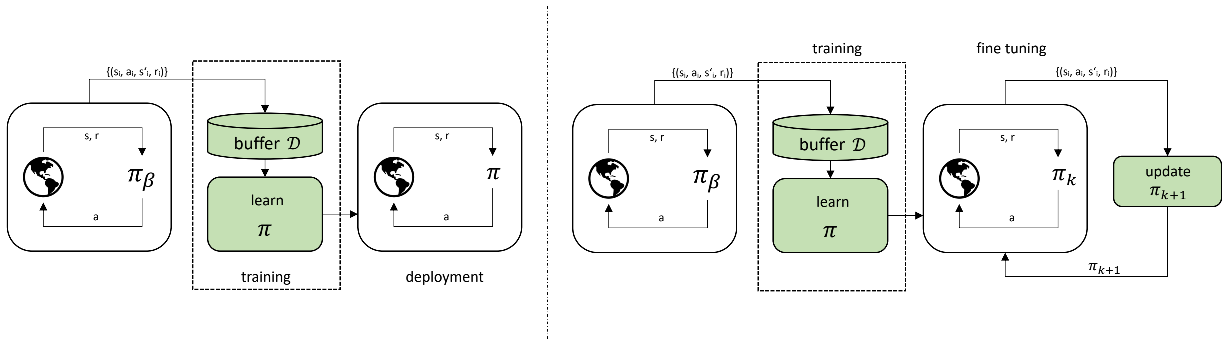


Figure 2.12: Frameworks Offline RL [38] [46]. On the left, pure offline learning with direct deployment in the environment. On the right, offline learning with further online learning afterwards.

2.4.0.1 Fundamental sources of errors in offline RL

In general, the offline data can be delivered by any kind of source and be very diverse. However, the learned policy can only display behavior that is also represented in the underlying data. This correlation stresses the importance of data selection.

Furthermore, another fundamental problem of offline RL is the distributional shift. Distributional shift generally describes the phenomena that the behavior policy, the policy collecting the data (in 2.12 policy π_β), differs from the policy derived from the data (in 2.12 policy π). The reason for this difference is that common off-policy algorithms are prone to overestimate unseen states and end up in a loop of constant overestimation. State-of-the-art off-policy algorithms such as SAC were tested on datasets. By having a look at the Q-values, the performance of SAC on an expert dataset seemed incredible since they have grown very quickly. In reality, the algorithm performed badly [32] and only thinks it performs well. In Q-learning, Q-values are updated by choosing the maximum over the actions $\max_{a'} Q(s', a')$. If the maximum is represented by an out-of-distribution action, the Q-value is updated with this never-seen action. This behavior starts a cascade of constantly increasing overestimations. In classical online learning, there might also occur the effect of overestimation, but with exploration and the actual visitation of the state-action pair, the algorithms corrects its overestimation. Whereas in the offline setting, there is no possibility to correct the overestimation through exploration since the data does not represent the overestimated state-action pair and online interaction is not possible. So far, several approaches have been developed to tackle the problem of distributional shift, under them policy constraint methods, conservative model-based methods or lower-bounded policy-value methods [37].

2.4.1 Policy Constraint Methods

One way to address the problem of the distribution shift is to add some form of pessimism to our policy so that Out-of-Distribution (OOD) actions are not taken anymore [37]. This means that the target value $\pi(a_{t+1}|s_{t+1})$ is forced to be close to the behavior distribution, the one that collected the data $\pi_\beta(a_{t+1}|s_{t+1})$. This pessimism prevents the overestimation cascade since all states and actions for the Q-function are in-distribution.

In general, there exist several ways to implement such constraint, e.g., support matching [34], distribution matching [19], state-marginal constraints [45] or implicit/closed-form distribution constraints [46]. The different types of constraints often lead to an implicitly different type of algorithms approaching the solution differently. The goal is to choose a constraint, which is least restrictive [37]. Empirically, the success of the policy constraint method heavily depends on the tunability of the method.

Generally, we can formalize such an approach by imposing a constraint on the policy:

$$\pi_\phi := \arg\max_\phi \mathbb{E}_{a' \sim \pi_\phi(a|s)}[Q(s,a)] \qquad \text{s.t.} \quad D(\pi_\phi(a|s), \pi_\beta(a|s)) \leq \epsilon \tag{2.40}$$

with β given as the distribution displayed in the dataset.

The constraints alters the optimal solutions in the following way:

$$\max_\pi \quad \mathbb{E}_\pi \left[\sum_t \gamma^t r(s_t, a_t) \right] - \alpha D(\pi(a|s), \pi_\beta(a|s)) \tag{2.41}$$

Examples for the constraint modeling can look like follows:

- $D(\pi_\phi, \pi_\beta) = \text{MMD}((\pi_\phi, \pi_\beta)$
- $D(\pi_\phi, \pi_\beta) = D_{\text{KL}}((\pi_\phi, \pi_\beta)$
- $D(\pi_\phi, \pi_\beta) = D(d^{\pi_\phi}(s,a), d^{\pi_\beta}(s,a))$

2.4.1.1 Bootstrapping Error Accumulation Reduction (BEAR) According to Kumar et al. [34], the weakness of current off-policy algorithms applied to offline datasets stems from the bootstrapping of OOD actions. This bootstrapping error accumulates over time. The Bootstrapping Error Accumulation Reduction (BEAR) algorithm was developed to address this problem, which is, in general, a policy constraint method. On a high level, BEAR constrains the learned policy to place the "non-zero probability mass on actions with non-negligible behavior policy density"[32] and is, therefore, a support matching constraint method. The advantage of BEAR is the robustness when facing a data set not generated by an expert but from a suboptimal policy.

The BEAR algorithm is built on top of the classical SAC algorithm as introduced in 2.3.3 but additionally a static dataset of transitions $D = \{s, a, s', R(s, a))\}$ which is collected by the unknown policy $\beta(\cdot|s)$ is given. Hereby, the state-action distribution of β is denoted as $\mu(s, a)$. For the improvement of our policy, two components are used. Firstly, K Q-functions and their respective minimum Q-values and secondly, a constraint, which ensures that the set of policies $\prod_\epsilon$ shares the same support as the behavior policy μ.

The policy is updated with the maximum over the pessimistic Q-values with the set of Q-functions $\hat{Q}_1, ..., \hat{Q}_K$:

$$\pi_\phi(s) = \max_{\pi \in \prod_\epsilon} \mathbb{E}_{a \sim \pi(\cdot|s)} \left[\min_{j=1,\dots,K} \hat{Q}_j(s, a) \right] \tag{2.42}$$

To enforce the satisfaction of the support constraint, the sampled Maximum Mean Discrepancy (MMD) between actions is used to measure the support divergence. With MMD, the discrepancy can be estimated solely with samples from both distributions, β and actor π. The MMD between P and Q given with the samples $(x_1, ..., x_n) \sim P$ as well as $(y_1, ..., y_m) \sim Q$ and universal kernel $k(\cdot, \cdot)$ can be written in the following way:

$$\text{MMD}^2(\{x_1, ..., x_n\}, \{y_1, ...y_m\}) = \frac{1}{n^2} \sum_{i,i'} k(x_i, x_{i'}) - \frac{2}{nm} \sum_{i,j} k(x_i, y_i) + \frac{1}{m^2} \sum_{j,j'} k(y_j, y_{j'}) \tag{2.43}$$

Putting together 2.42 and 2.43 into 2.40,the following constrained policy improvement step is created:

$$\pi_\phi := \max_{\pi \in \delta_{|S|}} \mathbb{E}_{s \sim D} \mathbb{E}_{a \sim \pi(\cdot|s)} \left[\min_{j=1,\dots,K} \hat{Q}_j(s, a) \right] \quad \text{s.t. } \mathbb{E}_{s \sim D}[\text{MMD}(D(s), \pi(\cdot|s))] \leq \epsilon. \tag{2.44}$$

In [34], a threshold of $\epsilon = 0.05$ has been established. In summary, the actor maximizes the Q-function, while the Q-function performs a constrained Q-learning over a reduced set of policies.

By comparing BEAR to several state-of-the-art baseline algorithms such as BC, BCQ or DQfD with different datasets generated by different policies (from random low-return policy to expert high-return policy), [32] showed an outperformance of every algorithm by BEAR. Furthermore, the BEAR algorithm was able to perform better than the expert high-return policy.

2.4.1.2 Behavioural Regularized Actor Critic (BRAC) Like BEAR, the Behavioural Regularized Actor Critic (BRAC) framework has the goal to regularize a learned policy but tries to generalize the existing approaches such as BEAR or BCQ further [70]. Similar as in SAC, a term is added to the target Q-value calculation, which will push the learned policy π towards the behavior policy π_β (*value penalty).* Therefore, the penalization of the value function looks like following:

$$V_D^\pi(s) = \sum_{t=0}^{\infty} \gamma^t \mathbb{E}_{s_t \sim P_t^\pi(s)}[R^\pi(s_t) - \alpha D(\pi(\cdot|s_t), \pi_\beta(\cdot|s_t))] \tag{2.45}$$

with D measuring the divergence between the actions of policy π and π_β. Common divergence functions are MMD oder KL.

The Q-value objective together with the sample-based estimated penalization $\hat{D}$ and $\bar{Q}$ as target Q-function is given by:

$$\min_{Q_\psi} \mathbb{E}_{\substack{(s,a,r,s')\sim D \\ a' \sim \pi_\theta(\cdot|s')}} \left[r + \gamma(\bar{Q}(s', a') - \alpha \hat{D}(\pi_\theta(\cdot|s'), \pi_\beta(\cdot|s'))) - Q_\psi(s,a))^2 \right]. \tag{2.46}$$

Furthermore, the objective is a slight variation of 2.41 and can be written as,

$$\max_{\pi_\theta} \mathbb{E}_{(s,a,r,s')\sim D} \left[\mathbb{E}_{a'' \sim \pi_\theta(\cdot|s)}[Q_\psi(s, a'')] - \alpha \hat{D}(\pi_\theta(\cdot|s), \pi_\beta(\cdot|s)) \right]. \tag{2.47}$$

Another form of regularization is to constrain the policy only during its optimization with $\alpha = 0$ in 2.46 during the Q-update and with $\alpha \neq 0$ in 2.47 during policy update. This variation is called *policy regularization.*

In addition, another design decision to consider is the choice of D. With BEAR the Kernel MMD was already introduced, but additional options are KL Divergence or the Wasserstein Distance.

In [70], the same evaluation as in [34] was performed with different datasets generated by different policies with a simpler form of BEAR derived from the BRAC framework. It was shown that value penalty performs slightly better than policy regularization. In the case of the divergence choice, no advantage or disadvantage for any regularizer was shown, but a sensitivity towards hyperparameters was observable.

2.4.2 Lower-Bounded Policy-Values Methods

A different approach to tackle the problem of distributional shift is to use lower-bounded policy-value methods, where the expected value of a policy is a lower-bound of the true value. Since the Q-values are overestimated due to bootstrapping OOD actions, learning a conservative estimation of the value function, which lower-bounds the true value addresses this issue [35]. The main idea is to minimize Q-values of unseen actions and afterwards tighten the bound by maximizing Q-values of actions included in the data distribution.

2.4.2.1 Conservative Q-Learning (CQL) The algorithmic framework of Conservative Q-Learning (CQL) aims to learn conservative, lower-bound estimations of the value function by constraining the Q-values during the training phase and therefore prevents overestimating OOD actions. The theoretical analysis in [35] shows that only the *expected* value of the Q-function lower-bounds the true policy value. The conservative off-policy evaluation is addressed through minimization of the expected Q-value under the state-action distribution $\mu(s,a)$ with the restriction to match the state-marginal in D with $\mu(s,a) = d^{\pi_\beta}(s)\mu(a|s)$. The reason for this restriction is that during Q-function training the function queries unseen actions, not unseen states. The iterative update for the Q-function training with trade-off factor α can be given like follows:

$$\begin{aligned}\hat{Q}^{k+1} \leftarrow \arg\min_{Q} \alpha_{\text{con}} \cdot (\mathbb{E}_{s_t \sim D, a_t \sim \mu(a_t|s_t)}[Q(s_t,a_t)] - \mathbb{E}_{a_t \sim D, a_t \sim \hat{pi}_\beta(a_t|s_t)}[Q(s_t,a_t)]) + \\ \frac{1}{2}\mathbb{E}_{s,a,s' \sim D}\left[(Q(s_t,a_t) - \hat{\mathcal{B}}^{\pi}\hat{Q}^k(s_t,a_t))^2\right]. \quad (2.48)\end{aligned}$$

In addition, Kumar et. al. [35] proves when $\mu(a|s) = \pi(a|s)$ that $\mathbb{E}_{\pi(a|s)}[\hat{Q}^\pi(s,a)] \leq V^\pi(s)$ must be true.

To come to the CQL framework, small modification of 2.48 with regularizer $R(\mu)$ is needed:

$$\begin{aligned}\min_{Q} \max_{\mu} \alpha_{\text{con}}(\mathbb{E}_{a_t \sim D, a_t \sim \mu(a_t|s_t)}[Q(s_t,a_t)] - \mathbb{E}_{a_t \sim D, a_t \sim \hat{\pi}_\beta(a_t|s_t)}[Q(s_t|a_t)]) \\ + \frac{1}{2}\mathbb{E}_{s,a,s' \sim D}\left[(Q(s_t,a_t) - \hat{\mathcal{B}}^{\pi_k}\hat{Q}^k(s_t,a_t))^2\right] + \mathcal{R}(\mu). \quad (2.49)\end{aligned}$$

and come to CQL($\mathcal{R}$). A simplified version of 2.49 [33] is :

$$\hat{Q}^{\pi}_{\text{CQL}} \leftarrow \arg\min_{Q} \max_{\mu(a_t|s_t)} \overbrace{(\mathbb{E}_{s_t\sim D, a_t\sim\mu(a_t|s_t)}[Q(s_t,a_t)] - \mathbb{E}_{s_t,a_t\sim D}[Q(s_t,a_t)])}^{\text{CQL regularizer}}$$
$$+ \frac{1}{2\alpha_{\text{con}}} \underbrace{\mathbb{E}_{s,a,s'\sim D}\left[(r(s_t,a_t) + \gamma\mathbb{E}_{\pi}[\bar{Q}(s_{t+1},a_{t+1})] - Q(s_t,a_t))^2\right]}_{\text{TD error}}. \quad (2.50)$$

The regularizer minimizes the Q-values on unseen actions with overestimated values, while at the same time maximizes the expected Q-values on the dataset. The regularizer can e.g. be the KL-divergence and is a variation of 2.49:

$$\min_{Q} \alpha_{\text{con}}\, \mathbb{E}_{s_t\sim D}\left[\log\sum_{a_t}\exp(Q(s_t,a_t)) - \mathbb{E}_{a_t\sim\hat{\pi}_{\beta}(a_t|s_t)}[Q(s_t,a_t)]\right]$$
$$+ \frac{1}{2}\mathbb{E}_{s,a,s'\sim D}\left[(Q - \hat{\mathcal{B}}^{\pi_k}\hat{Q}^k)^2\right] \quad (2.51)$$

The pseudocode to CQL can be given in the following way:

Algorithm 1: CQL [35]

```
Initialize Q-function, $Q_\theta$ and policy $\pi_\phi$ (if using actor-critic method);
for step t in {1,...,N} do
    Train Q-function using $G_Q$ gradient steps on objective from 2.51;
    $\theta_t = \theta_{t-1} - \eta_Q \nabla_\theta \text{CQL}(\mathcal{R})(\theta)$;
    ($\mathcal{B}^*$ for Q-learning, $\mathcal{B}^{\pi}_{\phi_t}$ for actor-critic);
    (only with actor-critic);
    Improve policy $\pi_\phi$ via $G_\pi$ gradient steps on $\phi$ with SAC entropy regularization;
    $\phi_t := \phi_{t-1} + \eta_\pi \mathbb{E}_{s\sim D, a\sim\pi_\phi(\cdot|s)}[Q_\theta(s,a) - \log\pi_\phi(a|s)]$
end
```

Empirically, CQL outperforms other offline RL methods such as BEAR or BRAC variations by the factor 2-5x and is the only method to perform better than behavioural cloning due stitching. Stitching describes the process of stitching together different sub-trajectories of a dataset. As an example, CQL can in theory solve a maze by stitching together several sub-trajectories, which show in combination the correct way, even though the solution way through the maze is not directly displayed in the dataset.

2.5 Behavior Cloning (BC)

A different approach to train an agent is behavioral cloning. Instead of learning from rewards and punishment from the environment, the agent imitates the behavior from an expert (typically a human). This form of learning is called Learning from Demonstration (LFD) [30]. In behavior cloning, a dataset with actions and observations is generated and, afterward, a model is trained with supervised learning [37].

One of the very first applications of imitation learning was Autonomous Land Vehicle in A Neural Network (ALVINN) [56]. The car was able to drive up to 90 miles without human interruption but suffered from one fundamental problem. Due to a normal randomness in the system, the car will sooner or later end up in a state it never was before and will not know which action to take. For the car, it is very hard to recover from this since no data shows a recovery trajectory. In addition, it is hard to provide high quality data sets with a diverse demonstration of actions and states [30]. An improvement in autonomous driving was the adaption to use three cameras [9]. One front camera, one to the left and one to the right. Through the augmentation of the three images the model can understand the environment better and learn from its mistakes.

Since this adaptation was still unable to recover from errors, the DAgger algorithm was developed to solve this problem. The DAgger method helps the agent to recover from unseen states by adding new measurements to the dataset. On the other hand, the DAgger algorithm has the disadvantage of labeling the actions to observation by hand, which makes the algorithm impractical for large data sets [20].

A quite successful application of BC was shown in [62]. Here, they used data generated from experts playing the Go game to initialize the policy network. Nevertheless, LfD was not enough to achieve a super human performance since no intelligence is incorporated. Additionally, they used search tree techniques and another value network. LfD was only the basis for further improvement.

3 Approach

The following section introduces the Gym CarRacing-v0 environment and the CARLA driving simulator. Furthermore, the dataset creation, the different SAC hyperparameters and CQL hyperparameters are outlined. In addition, the pre-processing, the reward function design and the CNN architecture is described.

3.1 CarRacing-v0

The CarRacing-v0 environment is an OpenAI Gym environment. A small formula 1 car drives on a racetrack and the goal is to cover as much of the racetrack as possible.

3.1.1 Environment

The observation space of the environment consists of 96x96 RGB images. CarRacing-v0 is a continuous control task environment with a top-down view. The action space is defined as follows: steering wheel angle $\in [-1, 1]$, gas $\in [0, 1]$ and brake $\in [0, 1]$. The reward of the environment is defined as - 0.1 for every frame and $+\frac{1000}{N}$ for every track tile visited in the episode with N as the total number of track tiles. The episode is finished when all tiles are visited or, the car drove off the map. At the bottom of the image, the following information is displayed: the actual speed, four ABS sensors, the steering wheel position and, a gyroscope.

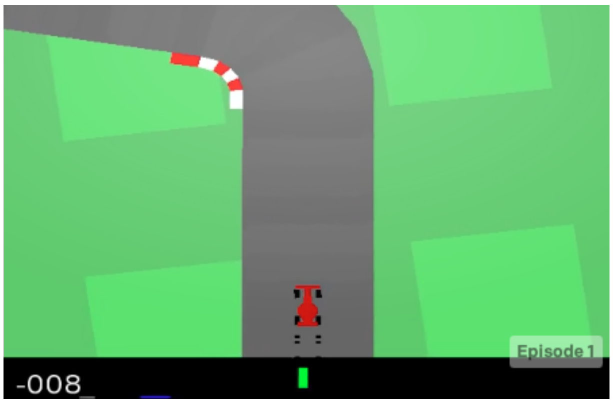


Figure 3.1: Graphic shows example image of the CarRacing-v0 environment

3.1.2 CarRacing-v0 SAC

Pre-Processing Before training SAC on the CarRacing-v0 environment, some pre-processing was necessary. Firstly, the black bar at the bottom of the image was cut off to prevent a learning from the displayed information. Secondly, the RGB image was normalized and turned into a grayscale image to speed up the learning process.

SAC Algorithm For the SAC algorithm, the implementation of Laskin et al. [36] was used. The hyperparameters for the final model can be seen in the following table:

Hyperparameters	SAC (Pixel)
General	
Observation size	96
Frame stack	4
Action repeat	2
Encoder	
Number of layers	4
Number of filters	32
Latent dimensions	128
Encoder learning rate	1×10^{-3}
Encoder tau	0.05
Agent Learning	
Train steps	1×10^{6}
Replay buffer size	5×10^{4}
Batch size	256
Discount factor γ	0.99
Critic learning rate	1×10^{-3}
Critic optimizer beta	0.9
Critic target update frequency	2
Critic tau	0.1
Target interpolation factor	0.005
Min log SD	-10
Max log SD	2
Actor learning rate	1×10^{-3}
Actor optimizer beta	0.9
Actor update frequency	2
Learnable temperature	True
Initial temperature	0.1
Alpha learning rate	1×10^{-4}
Alpha optimizer beta	0.5

Table 3.1: Hyperparameter settings for the SAC algorithm during the CarRacing-v0 training. The first column contains the hyperparameter name and the second column the respective value.

The two hyperparameters frame stack and action repeat in table 3.1 refer to techniques that help stabilizing the training process.

Frame stack Frame stacking is a technique that helps the model to understand the dynamics of an environment. By stacking every k frames together, it is easier for the agent to understand temporal information such as velocity or speed. In addition, the agent can more easily understand the consequences of its actions.

Action repeat Action repeat refers to a technique that let the model repeat an action k times before selecting a new one. In the case of autonomous driving in RL, it helps the car to prevent swerving and stabilizes the training. Instead of choosing every state a new action, it repeats the action for several states.

3.1.3 Dataset generation

To test the CQL algorithm on CarRacing-v0, it is necessary to generate a dataset beforehand. Therefore, a SAC agent was fully trained until it was able to solve the environment. The environment is considered to be solved with a constant reward level above 800. With the generated model, a dataset was created. The following illustration shows the results of several SAC trainings with the hyperparameters of table 3.1. With this hyperparameters, it was possible to achieve a stable and good learning performance. Every time, the agent reached expert level. Furthermore, SAC was very efficient and reached the desired reward-level very fast.

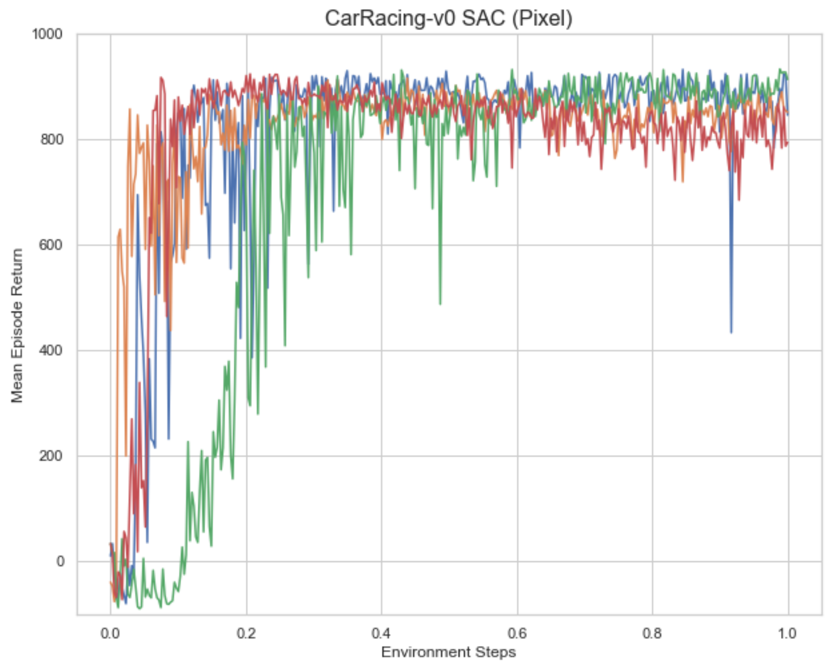


Figure 3.2: SAC (Pixel) results in the CarRacing-v0 environment steadily achieving a reward around 850

During the training, every 1000 steps, the current model was saved. To test the CQL algorithm, we decided to evaluate the performance on two different datasets. A high reward dataset and a low reward dataset. Therefore, from the saved models, we chose one, which achieved on average a reward of 851, and one, which achieved on average a reward of 426. With the two selected models, we created the datasets. Each dataset consisted of 50k transitions.

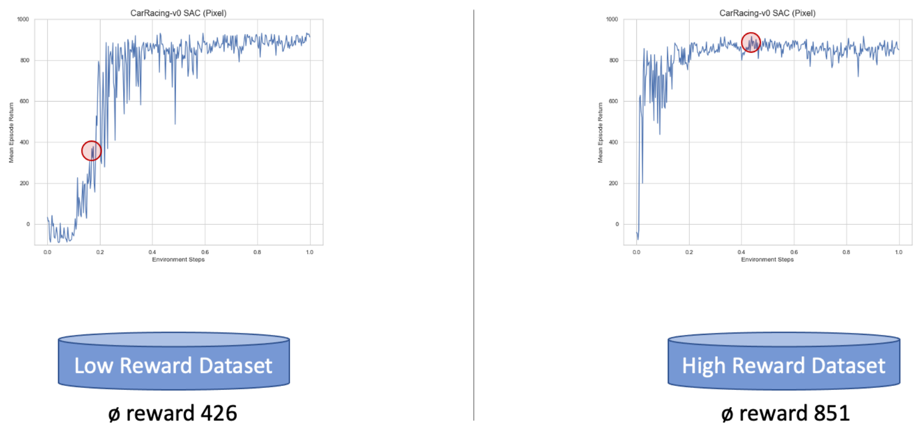


Figure 3.3: Low and high reward dataset generated for the CQL algorithm evaluation

3.1.4 CQL Approach

For CQL, we used the implementation of Seno et al. [60]. The network optimization is done in the following way:

$$L(\theta_i) = \alpha_{\text{con}} \mathbb{E}_{s_t \sim D} \left[\log \sum_{a_t} \exp Q_{\theta_i}(s_t, a_t) - \mathbb{E}_{a_t \sim D}[Q_{\theta_i}(s_t, a_t)] - \tau \right] + L_{\text{SAC}}(\theta_i) \tag{3.1}$$

with the computation of $\log \sum_a \exp Q_{\theta_i}(s_t, a)$ as follows:

$$\log \sum_a \exp Q_{\theta_i}(s_t, a_t) \approx \log \left(\frac{1}{2N} \sum_{a_i \sim \text{Unif(a)}}^{N} \left[\frac{\exp Q(s_t, a_i)}{\text{Unif}(a)} \right] + \frac{1}{2N} \sum_{a_i \sim \pi_\phi(a_t|s_t)}^{N} \left[\frac{\exp Q(s_t, a_i)}{\pi_\phi(a_i|s_t)} \right] \right) \tag{3.2}$$

where N is the number of sampled actions.

Furthermore, it was possible to adjust the α parameter automatically via a Lagrangian dual gradient. If the Q-value is smaller than the threshold τ, the α value becomes smaller. Otherwise, if the Q-values become larger than the threshold, the α value gets increased to penalize large Q-values. Therefore, we decided to test constant and learning alpha values on the two datasets. Hereby, we focused on the variation of the alpha values. We expected that high α values should perform better on the high reward dataset and worse on the low reward dataset. Higher α values imply higher conservatism which means the agent should stay closer to the action distribution shown in the dataset. Lower α values imply lower conservatism and, therefore, should perform better on the low reward dataset. In this environment, we tested the following values: $\alpha \in [0.001, 0.01, 1, 5, 10, 20, 40]$. In addition, we compared the CQL results with behavior cloning.

In the following table, the hyperparameters for the CQL algorithm in the CarRacing environment can be seen. The reason we did not use frame stacking is that the dataset already contained stacked images.

Hyperparameters	CQL (Pixel)
General	
Observation size	96
Frame stack	0
Agent Learning	
Train steps	1×10^6
Batch size	256
Discount factor γ	0.99
Number of critics	2
Critic learning rate	3×10^{-4}
Critic optimizer beta	0.9
Critic tau	0.005
Actor learning rate	1×10^{-4}
Actor optimizer beta	0.9
N-step TD calculation	1
Initial temperature	0.1
Temperature learning rate	1×10^{-4}
Temperature optimizer beta	0.9
Initial alpha	$\alpha \in [0.001, 0.01, 1, 5, 10, 20, 40]$
Alpha learning rate	1×10^{-4} or 0
Alpha optimizer beta	0.9
Alpha threshold	10
Number of actions sampled	10

Table 3.2: Hyperparameter settings for the CQL algorithm during the CarRacing-v0 training. The first column contains the hyperparameter name and the second column the respective value.

As network architecture, we decided for the classical "Nature DQN" architecture style [44]. As activation function was ReLU used. This network architecture was also used for the behavior cloning algorithm.

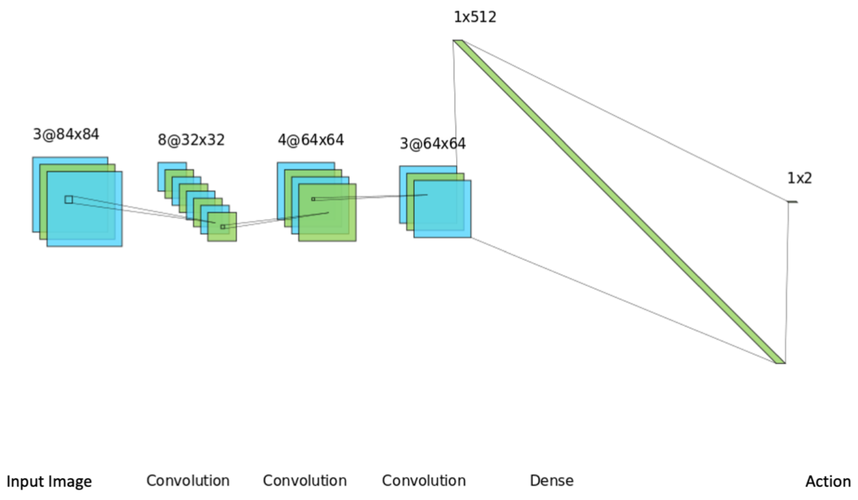
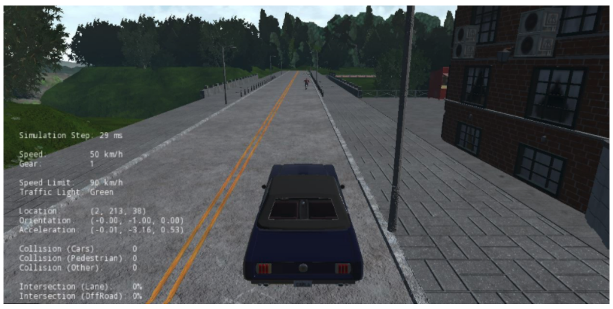


Figure 3.4: Small CNN in LeNet style

3.2 CARLA Environment

CAR Learning to act (CARLA) is an open-source code based driving simulator. It already provides pre-build maps with streets, buildings, pedestrians, and vehicles. In addition, different car sensors such as LiDAR sensors or different cameras are implemented. State information like velocity, acceleration, and location can quickly be withdrawn from the simulator [15].



```
Simulation Step: 29 ms

Speed:          50 km/h
Gear:           1

Speed Limit:    90 km/h
Traffic Light:  Green

Location:       (2, 213, 38)
Orientation:    (-0.00, -1.00, 0.00)
Acceleration:   (-0.01, -3.16, 0.53)

Collision (Cars):        0
Collision (Pedestrian):  0
Collision (Other):       0

Intersection (Lane):     0%
Intersection (OffRoad):  0%
```

Figure 3.5: Graphic shows example image of the Carla environment

3.2.1 Environment

Within the simulator, we decided to use an RGB camera placed at the front of the bonnet. The reason for this is that most car manufacturers only use cameras these days. Furthermore, we chose a resolution of 84x84x3 pixels. This resolution is clear enough to determine street lines and obstacles but blurry enough to ensure a fast computation. Furthermore, the action space of the Carla environment consists of the following variables: steering wheel angle $\in [-1, 1]$, acceleration $\in [0, 1]$ and brake $\in [0, 1]$. To reduce the number of variables, we decided to combine acceleration and brake to throttle/brake $\in [-1, 1]$ with negative values as braking and positive values as accelerating.

Reward function Design The self-defined task in Carla was to keep the lane and drive with a steady velocity. To achieve this behavior, the reward function is crucial. In contrast to the CarRacing-v0 environment, in Carla there is no predefined reward function. Therefore, it was necessary to implement our own. In general, a reward function consists of reward components and *penalty components*. We punish undesired behavior such as speeding, not moving, or driving off the lane with penalty components and reward desired behavior with reward components.

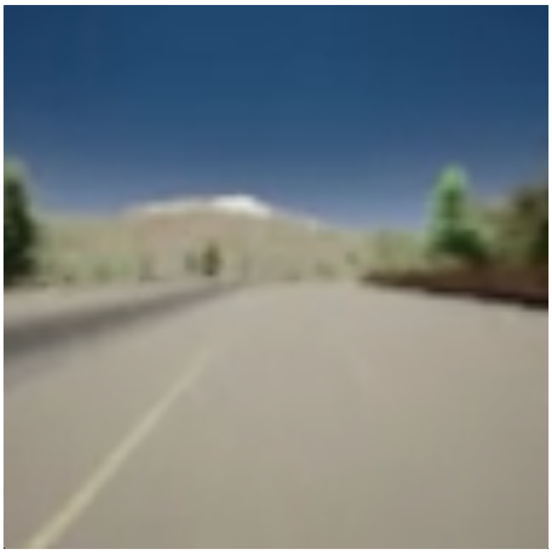


Figure 3.6: Actual camera image in the Carla simulator. Used as decoder input.

The final reward function consists of six

components: r_s as speed reward, r_h as
heading reward, r_d as distance-to-centerline reward, r_{sd} as steering-difference reward, r_s as steering reward and r_b as braking reward.

$$r_{total} = \begin{cases} r_b & v \leq 0 \\ w_s * r_s + w_h * r_h + w_d * r_d + w_{sd} * r_{sd} + r_s & \text{else} \end{cases}$$

The weights w_i sum up to 1. The final results were achieved with $w_s = 0.2$, $w_h = 0.1$, $w_d = 0.5$ and $w_{sd} = 0.2$. Moreover, we achieved better results with the reward functions in function form and not scalar values (figure 3.7). With this combination, a steady driving behavior between the lane markings was accomplished.

Speed Reward The speed reward motivates the agent to drive with a velocity near the speed limit. It receives a fast decreasing positive reward if a velocity bigger than the speed limit is reached.

Heading reward First of all, the heading of the car in relation to the street direction is calculated. Afterwards, the heading reward rewards driving in the street direction. If there is a huge difference between the car heading and the street heading, the reward is decreasing.

Distance-to-centerline Reward The distance-to-centerline reward calculates an imaginary perfect line between the two lane markings. The agent is trained to follow this imaginary line and is nudged to deviate as little as possible from this line. In the case of deviating, it receives a fast decreasing positive reward.

Steering Difference Reward The steering difference reward rewards slow steering maneuvers. The reward calculates the difference between steering_t and steering_{t-1} and rewards minor differences. This approach prevents swerving of the car.

Steering Reward The steering reward is designed to prevent sharp turns of the car or quick lane changes. It only gives negative rewards if the steering wheel angle is larger or less than ± 0.3. With optimal driving behavior, this reward should not be active and is therefore not weighted.

Braking Reward In the beginning, we had the problem that the car was not moving. Even with higher speed reward weights, such as $w_s = 0.8$ the car was not moving. To prevent this behavior, we introduced the braking reward, which penalizes braking if the car's velocity is less or close to zero. During the driving, this reward is not active.

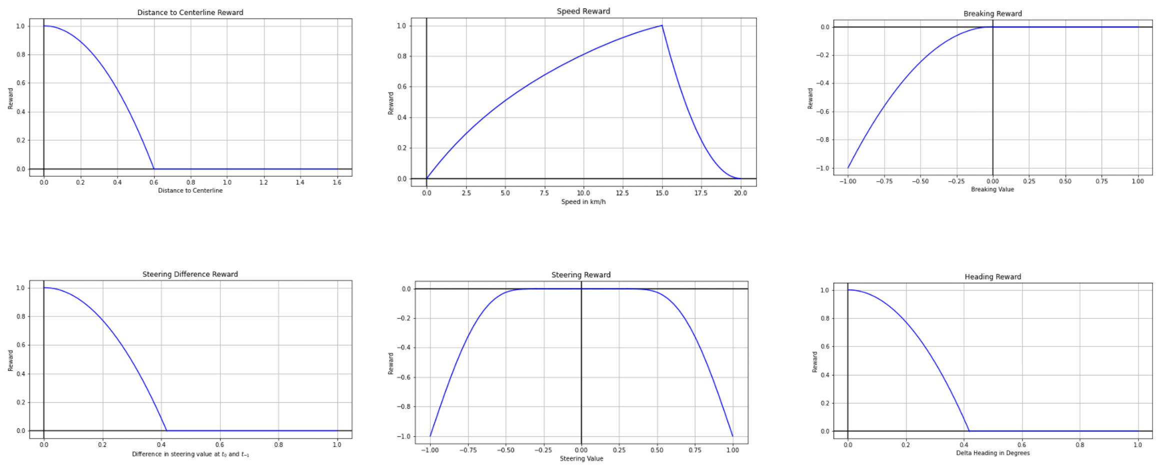


Figure 3.7: Visualization of the respective reward functions. Top left: distance-to-centerline reward, top middle: speed reward, top right: braking reward, bottom left: steering difference reward, bottom middle: steering reward, bottom right: heading reward

3.2.2 Carla SAC

To confirm that the reward function is working, we learned a SAC agent. If the SAC model can solve the task with our own reward function, the CQL algorithm should also be able to solve the task. For the actual training, we used the same hyperparameters as stated in table 3.1.

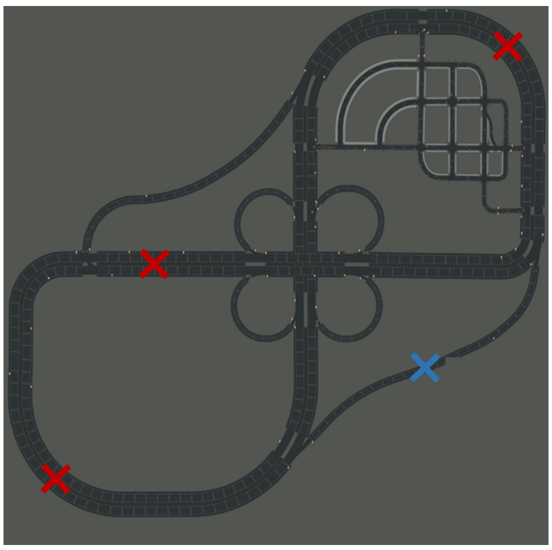


Figure 3.8: Graphic shows the map layout and the three spawn points (in red) during the training phase and the evaluation point (in blue)

During the training phase, we used the Carla Map 04 due to the long highways and the clear streets without traffic lights. In addition, we did not spawn any other cars or passengers on the map to keep the task as simple as possible. Furthermore, to increase the environment diversity during the training, we used three spawn points around the map (see figure 3.8 red points). For the evaluation, another spawn point was used to observe the car's performance (see figure 3.8 blue points). Every time, we evaluated 500 steps.

Because the reward function is designed to add up to 1, a maximum reward of 500 was

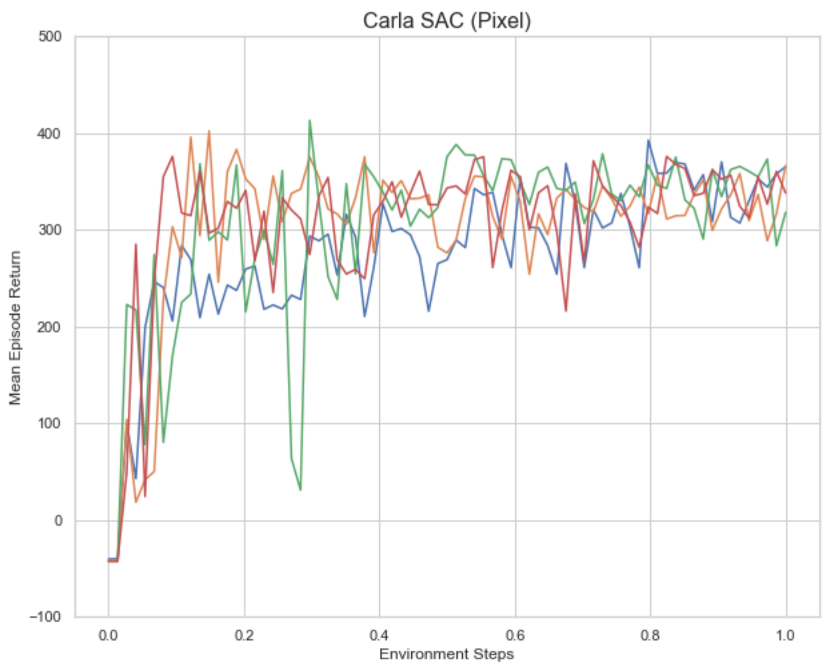
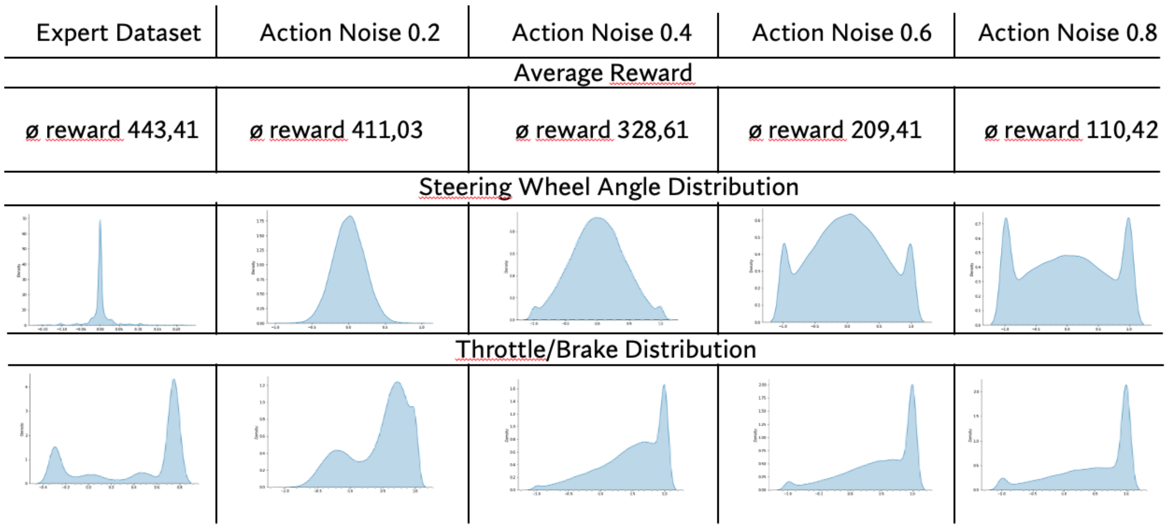


Figure 3.9: SAC (Pixel) results in the Carla environment.

possible. The SAC algorithm achieved a reward between 300 and 400 out of 500 on a constant level. Since we designed the reward function by ourselves, we had to confirm that the car fulfilled the task. For this, we saved the observations and put them in a video. Manually, we confirmed that the desired behavior, lane keeping, was fulfilled. This ensured that the reward function is working properly.

3.2.3 Dataset generation

For the CQL algorithm, we made five different datasets. To create these datasets, we used the Carla autopilot. With the autopilot, the perfect dataset was created. To get worse datasets, a so called action noise was used. Hereby, we used a normal distribution with mean zero and different standard deviations with values from 0.2 to 0.8. The normal distribution was layered over the autopilot actions and clipped so the values are between -1 and 1 for the steering wheel angle and throttle/brake. The following table provides an overview of the average reward, the steering wheel angle, and the throttle/brake distribution of the respective dataset. With increased action noise, the steering wheel angles get pushed to -1 and 1, and more unnatural speeding behavior is observable. Since the map consists primarily of straight lanes and some long curves, the steering wheel angle in the expert dataset is mostly around zero. More unnatural behavior is seen in datasets with higher action noise. High steering wheel angles imply uncontrolled driving behavior and quick swerving during the driving. Nevertheless, the car stays in all datasets mainly in the lane. Additionally, every dataset contains 50k transitions.



Expert Dataset	Action Noise 0.2	Action Noise 0.4	Action Noise 0.6	Action Noise 0.8
Average Reward				
ø reward 443,41	ø reward 411,03	ø reward 328,61	ø reward 209,41	ø reward 110,42
Steering Wheel Angle Distribution				
Throttle/Brake Distribution				

Figure 3.10: Overview of the datasets with the respective reward, steering distribution and throttle/brake distribution.

3.2.4 CQL Approach

In this thesis, we focused on the effect of the α value on the performance. Therefore, we tested the following values: $\alpha \in [0.01, 0.1, 1, 5, 10]$. The goal is to see if higher values perform better on good datasets. Moreover, the performance is set in contrast to behavior cloning. For the hyperparameters, we used the same as in table 3.2 but with observation space 84, frame stack 4 and $\alpha \in [0.01, 0.1, 1, 5, 10]$. Furthermore, the same CNN architecture was used.

In general, we expect that behavior cloning should perform at best with the expert dataset and has a decreasing performance with decreasing dataset quality. On the other way around, we expect CQL to perform better than behavior cloning on the poor datasets.

4 Experiments

Several experiments were conducted to evaluate the performance of CQL in the CarRacing-v0 and Carla environment. The results are summarized in section 4.1.

Advantages Offline RL during the Training During the experiment phase of this thesis, we recognized a notable advantage of offline RL. In general, the most time-intensive phase during online learning is to find an appropriate hyperparameter setting. Usually, for the classical RL environments such as the Atari Games or the PyBullet environments, already tuned models exist. Unfortunately, not for the Box2D environment CarRacing-v0. Much time was spent finding the correct and functional hyperparameter setting. This process is very time-intensive. Due to the lack of resources, we had to train a model for two to four days and evaluate afterwards the performance. This process repeats until an appropriate hyperparameter setting is found. The CarRacing-v0 environment has the advantage that the reward is already predefined. Therefore, we can measure the performance by looking at the achieved reward. In the Carla environment, it was not possible to evaluate the current model by just looking at the average reward because we defined an own reward function. Instead, we had to evaluate the model manually by watching training videos. This made the online training even more time-intensive. Furthermore, every time we restated a training in online RL, the buffer has to be filled again with trajectories. We do not profit from past trainings because all trajectories are deleted from the buffer when the training is restarted. In contrast to this, in offline RL, it is only necessary to create a dataset once. Now, It is not needed to collect trajectories again. The only time-consuming task is the hyperparameter setting, but the already collected dataset is always reused, making offline RL less time-intensive.

4.1 Results

The main experiments of this work evaluate the performance and sample efficiency of CQL on different datasets with different qualities. Hereby, we focused on different α values and the sensitivity of CQL on these different values.

4.1.1 Performance CarRacing-v0

In the following, we see the performance of CQL with different α values on the low reward dataset.

On average, the non-learning α approach performs better than the learning α approach. Only in one scenario, with $\alpha = 0.001$, the learning α performs better than the constant α but cannot keep the performance level. All other learning α values cannot achieve a

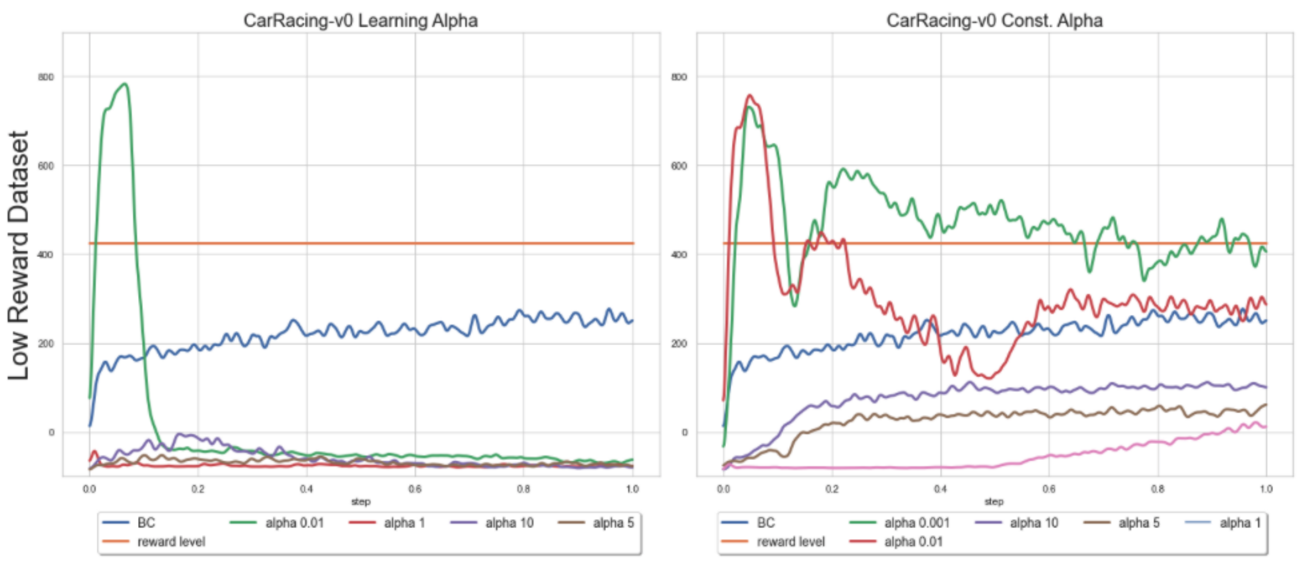


Figure 4.1: Experimental results on the low reward dataset with an average reward of 426. An 1-D gaussian filter with $\sigma = 5$ was used to even out short-term volatility.

positive reward. The learning α approach seems very unstable. In the constant approach, all α values achieve a positive reward. Lower α values perform better than bigger values. With $\alpha = 0.01$ and $\alpha = 0.001$, it was possible to outperform the dataset performance and behavior cloning. Only moderate results are achieved with higher values such as $\alpha \in [1, 5, 10]$.

The experimental results on the high reward dataset are depicted in figure 4.2.

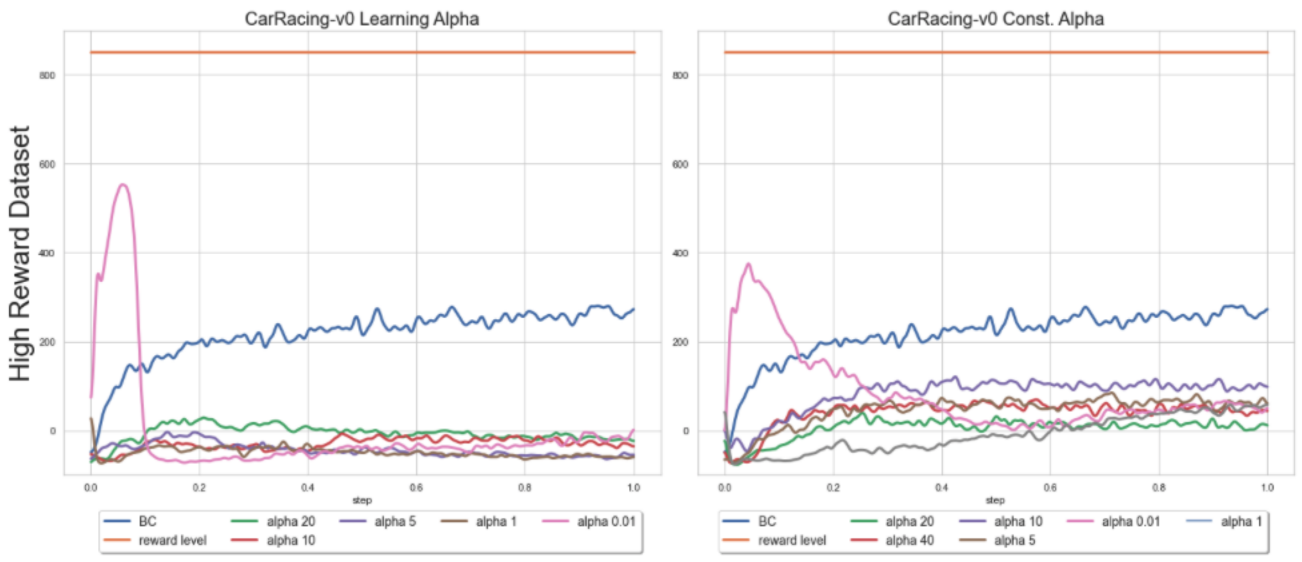


Figure 4.2: Experimental results on the high reward dataset with an average reward of 851. An 1-D gaussian filter with $\sigma = 5$ was used to even out short-term volatility.

Again, the high reward dataset results confirm that constant α values perform better than learning α values. In the learning approach, only $\alpha = 0.01$ was able to outperform behavior cloning, but cannot hold the performance level and quickly decreased performance-wise. Moreover, no α value was able to reach the dataset reward level. Similar in the constant approach, only $\alpha = 0.01$ performed better than behavior cloning, but decreased performance-wise as well. In both settings, all other α values besides $\alpha = 0.01$ did not reach a positive reward or did not reach behavior cloning level. We were not able to confirm that higher α values perform better on good datasets. We assume that this can be attributed to the highly stochastic environment of CarRacing-v0. Reasons for the bad performance might be the underlying dataset. During the evaluation, we tested the current model ten times. After every evaluation run, the car resets, and a completely new map with a new layout is created. We presume that the maps during CQL evaluation differed too much from the maps in the dataset.

Furthermore, by looking at the behavior cloning performance, we can see that it performs on both datasets equally. Therefore, we assume that the network capacity is too limited. Regarding the network architecture, we tested deeper and bigger architecture styles and were able to achieve a performance of approximately 600 on the high reward dataset. Therefore, an additional reason for the bad performance of CQL might be the network capacity limitation. Due to computational limitations, we were not able to test CQL with bigger networks.

4.1.2 Performance Carla

In the following, we see the performance of CQL with different α values on the different Carla datasets. Hereby, we tested $\alpha \in [0.01, 0.1, 1, 5, 10]$ on the expert dataset as well as noise dataset 0.2 - 0.8.

Performance Reward Function

As we can see in the following illustration 4.3, in all cases, behavior cloning was outperformed. Unfortunately, no consistency between the α values and the dataset quality is observable.

On the expert dataset, $\alpha = 0.01$ performed by far the best. On the noise dataset 0.2, $\alpha = 5$ achieved the best reward, whereas $\alpha = 0.1$, $\alpha = 5$ and $\alpha = 10$ were the best on the noise dataset 0.4. On noise dataset 0.6, $\alpha = 5$ was the best and on the noise dataset 0.8, $\alpha = 1$ achieved the highest reward. Moreover, only on the noise dataset 0.6 and noise dataset 0.8, CQL was able to reach dataset reward level. For now, the theory that higher α values perform better on good datasets and lower α values better on bad datasets could not be confirmed.

Nevertheless, since we created our own reward function, the reward performance seen in figure 4.3 is only conditionally meaningful. Behavior such as excessively speeding or driving fast in circles might lead to high rewards. Therefore, it makes more sense to look at other parameters. Since the task was to drive with constant speed steadily in line, we measured the distance to the center line, as indicator for the lane keeping task and the vehicle velocity, as indicator for the steadiness of the speeding behavior. For that reason, we chose the best models of the different α values and measured again the performance with the new metrics. The results are seen in figure 4.4.

Performance New Metrics

Alpha 0.01 In figure 4.4, we see that the smallest $\alpha = 0.01$ does not perform well on any dataset. Often the car stops moving. Furthermore, we see a high volatility in the distance to center line, which indicates fast maneuvering with high steering wheel angles. This contradicts our self-imposed task to drive steadily in lane. Even though $\alpha = 0.01$ was not able to perform very well, we can see on the noise dataset 0.4 that the model learned not to cross the street line. The car crossed the street line, reduced the velocity afterwards and began to recenter to the middle line. Nevertheless, $\alpha = 0.01$ seems too low to perform good on any dataset.

Alpha 0.1 With $\alpha = 0.1$, the same difficulty is observable. Often the car stops moving. Nevertheless, on the noise dataset 0.4, the model was able to keep the lane very well and drove with a high and steady velocity. Furthermore, although stopping at the beginning of the episode, $\alpha = 0.1$ performed very well on the noise dataset 0.6. Besides one large

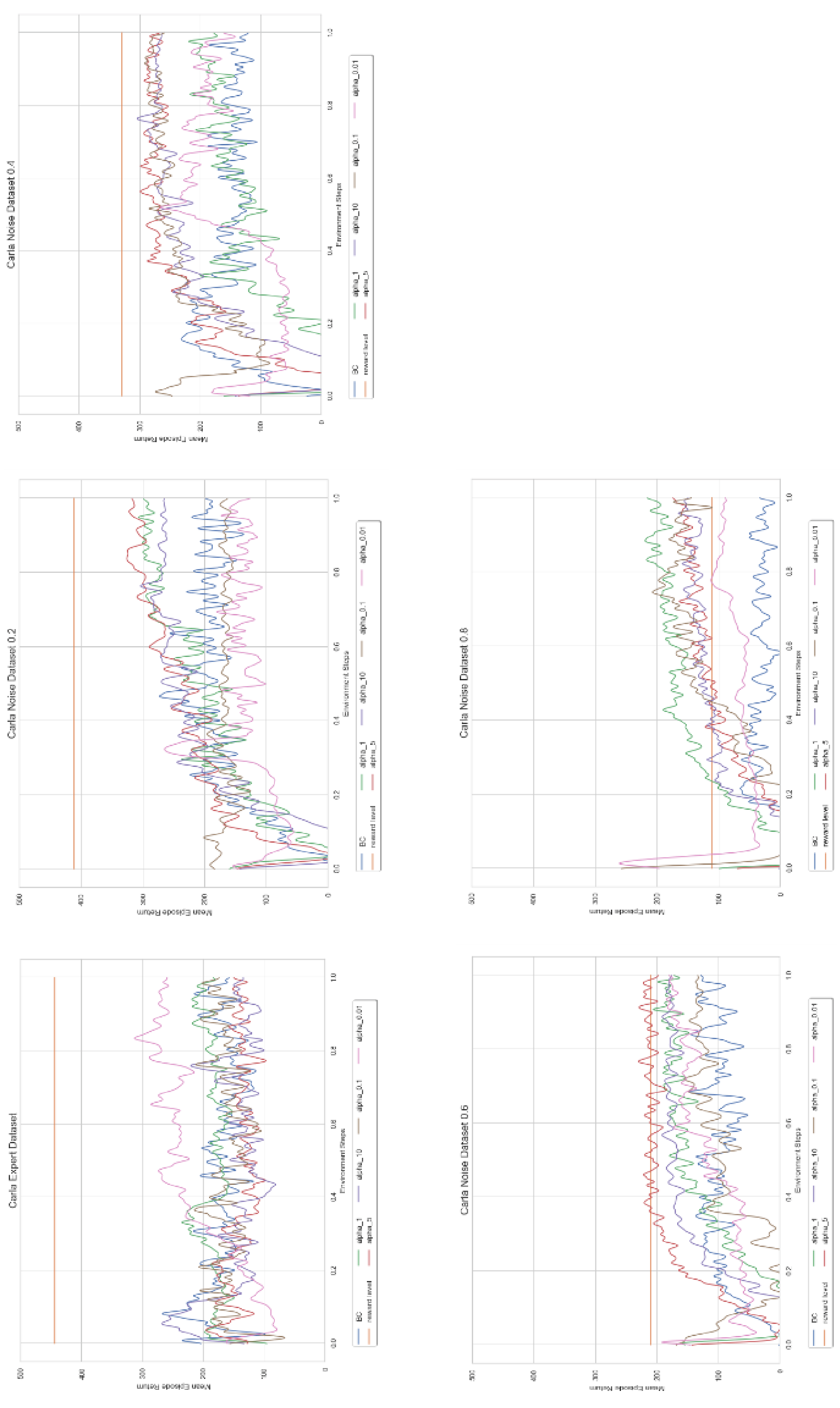


Figure 4.3: The figure shows the performance of the CQL algorithm on the different datasets. Top left is the expert dataset, top middle the noise dataset 0.2, top right the noise dataset 0.4, bottom left the noise dataset 0.6 and bottom middle the noise dataset 0.8. An 1-D gaussian filter with $\sigma = 5$ was used to even out short-term volatility.

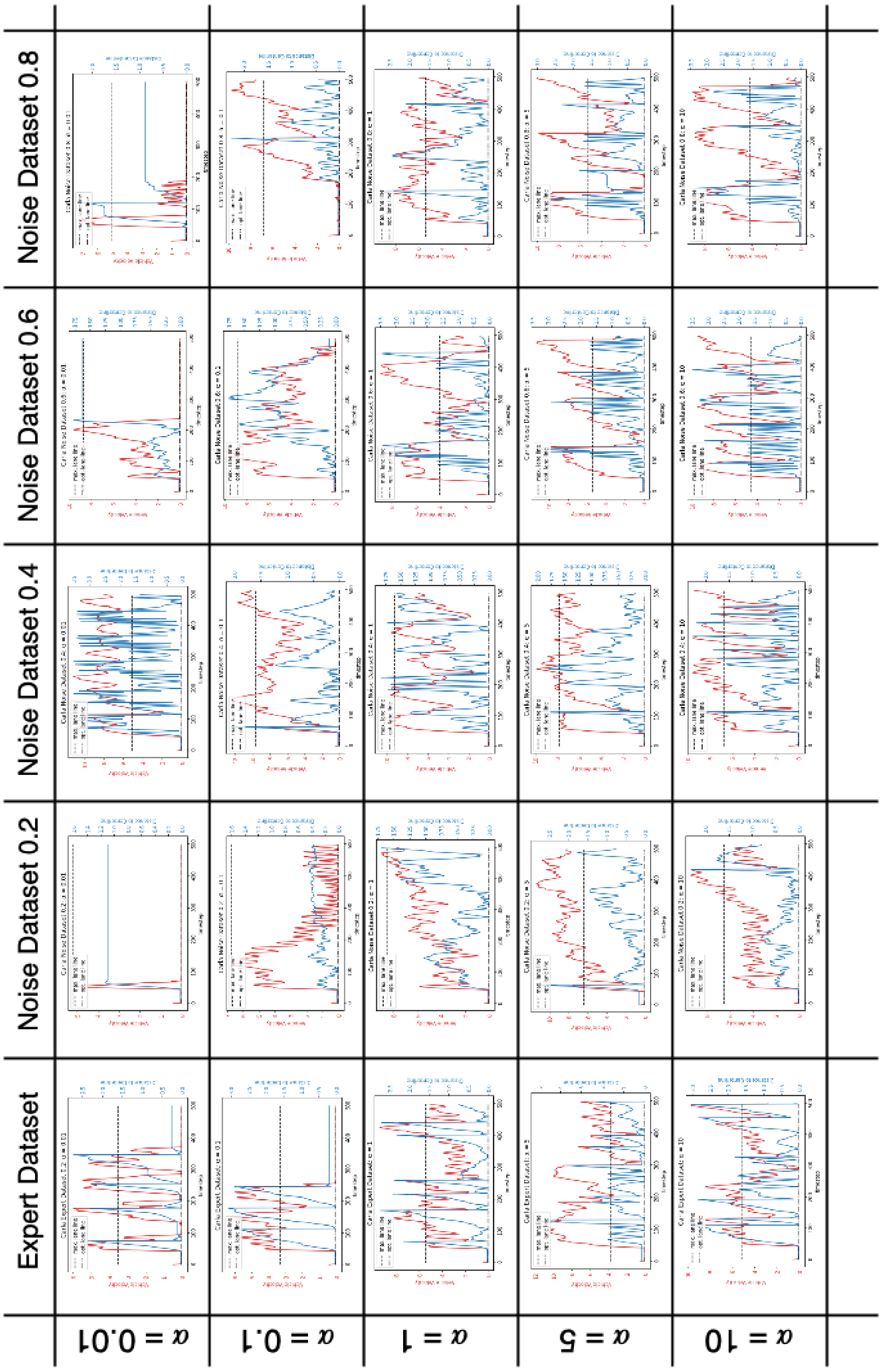


Figure 4.4: The figure shows the performance of the CQL algorithm on the different datasets. Hereby, we focused on the parameters vehicle velocity and distance to center line.

distance to centerline outlier, the overall small distance to centerline values are recognizable. Here, we can see that lower α values perform better on worse datasets. The best fitting α value for the noise dataset 0.4, 0.6 and 0.8 seems $\alpha = 0.1$.

Alpha 1 The $\alpha = 1$ performance on the noise dataset 0.2 must be highlighted. Here, we were able to produce one of the best results. The car kept close to the optimal distance to center line and accelerate constantly without losing the center line. Furthermore, we can observe the trade-off between vehicle velocity and distance to centerline. With increased vehicle velocity, it is harder for the model to keep the car close to the centerline. Overall, $\alpha = 1$ showed a better performance on better datasets. However, on the expert dataset, the performance was not good. Same reasoning as in 4.1.1, we assume that the trajectories in the expert dataset are too good and do not show recovery behavior.

Alpha 5 Here, $\alpha = 5$ performs the best on the noise dataset 0.2. With decreasing dataset quality, a higher distance to centerline oscillation is seen. When we compare the results of $\alpha = 5$ and $\alpha = 1$ on the noise dataset 0.2, we see that $\alpha = 1$ was able to drive steadier. Furthermore, we see that worse dataset quality results in worse driving behavior with $\alpha = 5$. Higher deviation from the centerline is observable.

Alpha 10 The worse the dataset gets, the more prominent is the oscillating driving behavior. Overall, $\alpha = 10$ does not perform very well. Therefore, we assume that $\alpha = 10$ is too high to perform good on the datasets.

Underlying Data The following overview shows the velocity and distance to centerline behavior in the respective data sets. With increased noise, we see a higher variance in the distance to centerline and vehicle velocity. The higher the α value, the closer the CQL algorithms stays to the dataset. Therefore, higher α values perform worse on bad datasets since the underlying data shows more oscillating behavior and the model tries to keep close to the dataset.

In addition, the higher speeding of the CQL models compared to the dataset speeding is explained as follows. Usually, the speed limit is 15 $\frac{\text{km}}{\text{h}}$ in Town04, whereas the autopilot, to stay ideally in the lane, only reaches around 5 $\frac{\text{km}}{\text{h}}$. However, the reward function is designed for 15 $\frac{\text{km}}{\text{h}}$. Therefore, the car reaches higher rewards when driving fast than the 5 $\frac{\text{km}}{\text{h}}$ shown in the dataset.

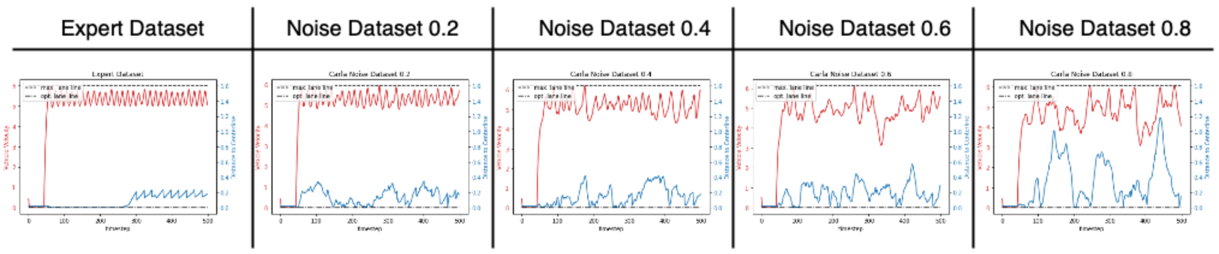


Figure 4.5: The figure shows the velocity and distance to center line behavior in the respective dataset.

In general, the CQL algorithm can solve the lane keeping task . Overall, not as good and not as smooth as online RL, but it is possible. The bad performance on the expert dataset is explained through the lack of recovery trajectories. The model does not see trajectories showing recovery behavior in the dataset and, therefore, cannot recover from bad states. A mixture of expert dataset with noise dataset 0.8 might lead to better results. Here the ratio between good and bad dataset must be studied.

5 Conclusion

This work aimed to train an agent to stay in lane and drive at a constant speed using offline RL. First, the role of deep learning in autonomous driving, the components of autonomous driving, and various deep learning techniques were presented. Then, the theoretical foundations of RL were presented, followed by a discussion of offline RL. Afterward, the main algorithm of this work, CQL, was introduced. For this purpose, CQL was evaluated in the CarRacing-v0 and Carla environments. Hereby, we focused on the sensitivity of the α parameter.

Several experiments were conducted in pixel-based and continuous action environments to evaluate the performance of CQL. First, we focused on the CarRacing-v0 environment. CarRacing-v0 has the advantage that the reward function is already predefined. Here we focused on the optimization of the CQL hyperparameters. Therefore, it was necessary to generate a dataset beforehand. Unfortunately, there was no benchmark dataset in D4RL [18]. Therefore, to create the dataset, we trained a SAC agent to completion so that model could solve the environment. Afterward, we used this agent to generate a high and low reward dataset. Following, we evaluated the performance of very high and very low α values on the two different datasets. We used behavior cloning as a comparison algorithm. In the low reward dataset, CQL outperformed behavior cloning and performed better than the dataset. In the high reward dataset, CQL struggled to exceed behavior cloning and was not able to reach dataset level. We hypothesize that the underlying data set is the problem. In the CarRacing environment, the track layout is recreated after every reset. Therefore, we believe that the layouts in the dataset differ too much from the layouts during the training. In other words, our agent could not generalize well.

In the Carla environment, we created five different datasets. The best one was created by an autopilot. The other four datasets were created through a modified autopilot with continuously increased action noise. On all datasets, we were able to show that CQL can outperform behavior cloning. Furthermore, it was possible to show that lower α values perform better the poorer the dataset and higher α values perform better the better the dataset. Nevertheless, the performance of CQL in the environments, CarRacing-v0 and Carla, is not as stable as online RL, but generally, CQL was able to solve the tasks.

Causes of errors CQL In general, CQL is an algorithm that is very sensitive to the α value. Much fine-tuning is necessary to find the best-working α value. In our case, the range of α values that worked best were between 5 and 0.1, which results in a long fine-tuning process.

Generally, pixel-based observations have a high volatility since a single color change in one pixel represents a new state. This requires high implementation stability. CarRacing-v0 and Carla represent high-dimensional environments which makes it hard to perform on. CQL might perform on low-dimensional environments better.

Another error-prone CQL aspect is the choice of data in the underlying dataset. As we were able to see in the CarRacing-v0 environment, good data does not necessarily mean good performance. Here, it is important to add recovery trajectories to good datasets. A mixture of an expert dataset with a more noisy dataset might help. This stresses the importance of diversity in the dataset to cover as much different states as possible. By that, the algorithm can recognize different states better and facilitates recovery from bad states.

5.1 Discussion

Methods such as CQL offers promising solutions for real-world problems by using large amounts of data. Especially real-world problems such as autonomous driving are very safety-critical. Therefore, learning from collected data helps to solve autonomous driving with RL. Nevertheless, the offline RL approach is still very error-prone, and so far, there exists no deployment in the real world. The experience is limited to simulated environments.

5.2 Outlook

In our approach, we used a simple CNN with three convolutional layers. Due to hardware limitations, we were not able to use bigger network structures with CQL. Even with this relatively small network, the Graphical Processing Unit (GPU) was already fully occupied. For example, in the CarRacing-v0 environment, we achieved a reward of 600 on the high reward dataset with a deeper behavior cloning network. Therefore, we assume that CQL could also benefit from an increased layer size.

In addition, Agarwal et al. [1] highlights the importance of large and diverse datasets and shows that offline RL benefits from them. In this thesis, we worked with 50k transitions in the CarRacing-v0 environment as well as in the Carla environment. In the benchmark D4RL [18] datasets, the size of the datasets was scaled up to 1M transitions. Especially in the expert datasets, it might be essential to increase the diversity and add transitions that show recovery behavior from bad states. Here, the right mixture between good transitions and recovery transitions might be crucial.

Moreover, Sinha et al. [64] was able to show the performance increase with image augmentation. Augmentations such as rotation or color jittering are common in computer vision research. Rotating an image with a slight angle still maintains the vital information but increases the abstractness. It was possible to show that on medium datasets, image augmentation can increase the CQL performance in a wide range of environments and tasks [64]. Therefore, it might improve the performance of CQL in the Carla environment.

References

[1] Agarwal, R., Schuurmans, D., and Norouzi, M. An optimistic perspective on offline reinforcement learning. In *International Conference on Machine Learning* (2020), PMLR, pp. 104–114.

[2] Albawi, S., Mohammed, T. A., and Al-Zawi, S. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)* (2017), pp. 1–6.

[3] Amit, R., Meir, R., and Ciosek, K. Discount factor as a regularizer in reinforcement learning. In *International conference on machine learning* (2020), PMLR, pp. 269–278.

[4] Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., and Mordatch, I. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528* (2019).

[5] Barnard, W. Lidar v. cameras - the autonomous vision race, May 2020.

[6] Behere, S., and Torngren, M. A functional architecture for autonomous driving. In *2015 First International Workshop on Automotive Software Architecture (WASA)* (2015), IEEE, pp. 3–10.

[7] Bertoncello, M., and Wee, D. Ten ways autonomous driving could redefine the automotive world. *McKinsey & Company 6* (2015).

[8] Blackburn. Introduction to reinforcement learningnbsp;: Markov-decision process, Aug 2020.

[9] Bojarski, M., Testa, D. D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., Zhang, X., Zhao, J., and Zieba, K. End to end learning for self-driving cars. *CoRR abs/1604.07316* (2016).

[10] Britz, D. Exploration vs. exploitation, Apr 2014.

[11] Campbell, M., Egerstedt, M., How, J. P., and Murray, R. M. Autonomous driving in urban environments: approaches, lessons and challenges. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 368*, 1928 (2010), 4649–4672.

[12] Capasso, A. P., Bacchiani, G., and Broggi, A. From simulation to real world maneuver execution using deep reinforcement learning, 2020.

[13] Delhi, S. I.-N. Automotive revolution & perspective towards 2030. *Auto Tech Review 5*, 4 (2016), 20–25.

[14] Dong, H., Dong, H., Ding, Z., Zhang, S., and Chang. *Deep Reinforcement Learning.* Springer, 2020.

[15] Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., and Koltun, V. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning* (2017), pp. 1–16.

[16] Fagnant, D. J., and Kockelman, K. Preparing a nation for autonomous vehicles: opportunities, barriers and policy recommendations. *Transportation Research Part A: Policy and Practice 77* (2015), 167 – 181.

[17] Folkers, A., Rick, M., and Büskens, C. Controlling an autonomous vehicle with deep reinforcement learning. In *2019 IEEE Intelligent Vehicles Symposium (IV)* (2019), IEEE, pp. 2025–2031.

[18] Fu, J., Kumar, A., Nachum, O., Tucker, G., and Levine, S. D4rl: Datasets for deep data-driven reinforcement learning, 2020.

[19] Fujimoto, S., Meger, D., and Precup, D. Off-policy deep reinforcement learning without exploration. In *International Conference on Machine Learning* (2019), PMLR, pp. 2052–2062.

[20] Gao, Y., Liu, Y., Zhang, Q., Wang, Y., Zhao, D., Ding, D., Pang, Z., and Zhang, Y. Comparison of control methods based on imitation learning for autonomous driving. In *2019 Tenth International Conference on Intelligent Control and Information Processing (ICICIP)* (2019), IEEE, pp. 274–281.

[21] Gottesman, O., Johansson, F., Meier, J., Dent, J., Lee, D., Srinivasan, S., Zhang, L., Ding, Y., Wihl, D., Peng, X., et al. Evaluating reinforcement learning algorithms in observational health settings. *arXiv preprint arXiv:1805.12298* (2018).

[22] Haarnoja, T., Tang, H., Abbeel, P., and Levine, S. Reinforcement learning with deep energy-based policies. *CoRR abs/1702.08165* (2017).

[23] Hirz, M., and Walzel, B. Sensor and object recognition technologies for self-driving cars. *Computer-aided design and applications 15*, 4 (2018), 501–508.

[24] Hui, J. Rl - dqn deep q-network, Mar 2019.

[25] Isele, D., Rahimi, R., Cosgun, A., Subramanian, K., and Fujimura, K. Navigating occluded intersections with autonomous vehicles using deep reinforcement learning, 2018.

[26] Juliani, A. Maximum entropy policies in reinforcement learning amp; everyday life, Nov 2018.

[27] Karagiannakos, S. The idea behind actor-critics and how a2c and a3c improve them, Nov 2018.

[28] Karunakaran, D. Q-learning: a value-based reinforcement learning algorithm, Sep 2020.

[29] Kendall, A., Hawke, J., Janz, D., Mazur, P., Reda, D., Allen, J.-M., Lam, V.-D., Bewley, A., and Shah, A. Learning to drive in a day. In *2019 International Conference on Robotics and Automation (ICRA)* (2019), IEEE, pp. 8248–8254.

[30] Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Al Sallab, A. A., Yogamani, S., and Pérez, P. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems* (2021).

[31] Kiran, B. R., Sobh, I., Talpaert, V., Mannion, P., Sallab, A. A. A., Yogamani, S., and Pérez, P. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems* (2021), 1–18.

[32] Kumar, A. Data-driven deep reinforcement learning, Dec 2019.

[33] Kumar, A. Offline reinforcement learning: How conservative algorithms can enable new applications, Dec 2020.

[34] Kumar, A., Fu, J., Soh, M., Tucker, G., and Levine, S. Stabilizing off-policy q-learning via bootstrapping error reduction. In *Advances in Neural Information Processing Systems* (2019), pp. 11784–11794.

[35] Kumar, A., Zhou, A., Tucker, G., and Levine, S. Conservative q-learning for offline reinforcement learning. *arXiv preprint arXiv:2006.04779* (2020).

[36] Laskin, M., Lee, K., Stooke, A., Pinto, L., Abbeel, P., and Srinivas, A. Reinforcement learning with augmented data. *arXiv preprint arXiv:2004.14990* (2020).

[37] Levine, S. Decisions from data: How offline reinforcement learning will change how we use ml, Sep 2020.

[38] Levine, S., Kumar, A., Tucker, G., and Fu, J. Offline reinforcement learning: Tutorial, review, and perspectives on open problems, 2020.

[39] Lindgren, A., and Chen, F. State of the art analysis: An overview of advanced driver assistance systems (adas) and possible human factors issues.

[40] Litman, T. *Autonomous vehicle implementation predictions.* Victoria Transport Policy Institute Victoria, Canada, 2017.

[41] Liu, J., Liu, S., and Gu, X. Soft q network, 2020.

[42] Mirchevska, B., Blum, M., Louis, L., Boedecker, J., and Werling, M. Reinforcement learning for autonomous maneuvering in highway scenarios. In *Workshop for Driving Assistance Systems and Autonomous Driving* (2017), pp. 32–41.

[43] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[44] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature 518*, 7540 (2015), 529–533.

[45] Nachum, O., and Dai, B. Reinforcement learning via fenchel-rockafellar duality. *CoRR abs/2001.01866* (2020).

[46] Nair, A., Dalal, M., Gupta, A., and Levine, S. Accelerating online reinforcement learning with offline datasets, 2020.

[47] Ni, J., Chen, Y., Chen, Y., Zhu, J., Ali, D., and Cao, W. A survey on theories and applications for self-driving cars based on deep learning methods. *Applied Sciences 10*, 8 (2020), 2749.

[48] Okuda, R., Kajiwara, Y., and Terashima, K. A survey of technical trend of adas and autonomous driving. In *Proceedings of Technical Program - 2014 International Symposium on VLSI Technology, Systems and Application (VLSI-TSA)* (2014), pp. 1–4.

[49] OpenAI. Part 1: Key concepts in rl.

[50] Or, B. Penalizing the discount factor in reinforcement learning, Oct 2020.

[51] Orenstein, D. Stanford team's win in robot car race nets $2 million prize, Oct 2005.

[52] Osinski, B., Jakubowski, A., Milos, P., Ziecina, P., Galias, C., Homoceanu, S., and Michalewski, H. Simulation-based reinforcement learning for real-world autonomous driving. *CoRR abs/1911.12905* (2019).

[53] Paden, B., Čáp, M., Yong, S. Z., Yershov, D. S., and Frazzoli, E. A survey of motion planning and control techniques for self-driving urban vehicles. *CoRR abs/1604.07446* (2016).

[54] Pendleton, S. D., Andersen, H., Du, X., Shen, X., Meghjani, M., Eng, Y. H., Rus, D., and Ang, M. H. Perception, planning, control, and coordination for autonomous vehicles. *Machines 5*, 1 (2017), 6.

[55] Petridou, E., and Moustaki, M. Human factors in the causation of road traffic crashes. *European journal of epidemiology 16*, 9 (2000), 819–826.

[56] Pomerleau, D. A. Alvinn: An autonomous land vehicle in a neural network. Tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA ARTIFICIAL INTELLIGENCE AND PSYCHOLOGY . . . , 1989.

[57] Proff, Pottebaum, W. Autonomous driving, moonshot project with quantum leap form hardware to software and ai focus.

[58] Quan, Y. S., and Chung, C. C. Approximate model predictive control with recurrent neural network for autonomous driving vehicles. In *2019 58th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)* (2019), pp. 1076–1081.

[59] Raats, K., Fors, V., and Pink, S. Trusting autonomous vehicles: An interdisciplinary approach. *Transportation Research Interdisciplinary Perspectives 7* (2020), 100201.

[60] Seno, T. d3rlpy: An offline deep reinforcement library. `https://github.com/takuseno/d3rlpy`, 2020.

[61] Shaheen, S. A., Cohen, A. P., and Roberts, J. D. Carsharing in north america: Market growth, current developments, and future potential. *Transportation Research Record 1986*, 1 (2006), 116–124.

[62] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature 529*, 7587 (2016), 484–489.

[63] Simonini, T. An intro to advantage actor critic methods: let's play sonic the hedgehog!, Jul 2018.

[64] Sinha, S., and Garg, A. S4rl: Surprisingly simple self-supervision for offline reinforcement learning. *arXiv preprint arXiv:2103.06326* (2021).

[65] Stamatiadis, N., and Deacon, J. A. Trends in highway safety: effects of an aging population on accident propensity. *Accident Analysis & Prevention 27*, 4 (1995), 443–459.

[66] Suran, A. On-policy v/s off-policy learning, Jul 2020.

[67] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction*, second ed. The MIT Press, 2018.

[68] Tang, H. Learning diverse skills via maximum entropy deep reinforcement learning, Oct 2017.

[69] Van Brummelen, J., O'Brien, M., Gruyer, D., and Najjaran, H. Autonomous vehicle perception: The technology of today and tomorrow. *Transportation research part C: emerging technologies 89* (2018), 384–406.

[70] Wu, Y., Tucker, G., and Nachum, O. Behavior regularized offline reinforcement learning. *arXiv preprint arXiv:1911.11361* (2019).

Assertion

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, September 6, 2022 Pascal Schindler