Kombination von Maximum Entropy Reinforcement Learning mit Distributional Q-Value Approximation

Am Beispiel Autonomes Fahren

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Mathematik

eingereicht von

Tobias Kietreiber, BSc
Matrikelnummer 01526084

an der Fakultät für Mathematik und Geoinformation
der Technischen Universität Wien

Betreuung: Assoz. Prof. Dipl.-Ing. Dr. techn. Clemens Heitzinger

Wien, 16. Mai 2023

Tobias Kietreiber	Clemens Heitzinger

TECHNISCHE
UNIVERSITÄT
WIEN

Combining Maximum Entropy Reinforcement Learning with Distributional Q-Value Approximation Methods

At the Example of Autonomous Driving

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Technical Mathematics

by

Tobias Kietreiber, BSc
Registration Number 01526084

to the Faculty of Mathematics and Geoinformation

at the TU Wien

Advisor: Assoz. Prof. Dipl.-Ing. Dr. techn. Clemens Heitzinger

Vienna, 16th May, 2023

Tobias Kietreiber

Clemens Heitzinger

Erklärung zur Verfassung der Arbeit

Tobias Kietreiber, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 16. Mai 2023

Tobias Kietreiber

Acknowledgements

First of all, I want to thank my advisor, Professor Clemens Heitzinger, for his guidance. It was a lot of fun working on this thesis and it would not have been possible without him.

Thanks as well to all my friends, especially Helmut Horvath for being a great project partner, Alexander Grosz for introducing me to the topic of Reinforcement Learning and countless fruitful discussions, Jakob Deutsch for helping me with the theory around Wasserstein distances, and Luka Ilić and Alexander Grosz for proofreading this thesis.

Last, but definitely not least, I thank my family for always supporting me.

Kurzfassung

Reinforcement Learning hat in den letzten Jahren sehr an Popularität gewonnen, da damit komplexe Probleme nur mithilfe eines Belohnungssignals gelöst werden können, besonders nachdem es auf moderne Deep Learning Architekturen ausgedehnt wurde. Es werden laufend neue Erweiterungen entwickelt, darunter die Approximation der q-Werte in Verteilung und Maximum Entropy Reinforcement Learning. Beide scheinen in Umgebungen des autonomen Fahrens besonders gut zu funktionieren.

In dieser Arbeit werden diese beiden Methoden vorgestellt, indem zunächst ein kurzer Überblick über bestehende Literatur gegeben und danach die Kombination der beiden Methoden präsentiert wird. Schlussendlich werden wir experimentell im CARLA Simulator zeigen, dass dies nicht nur funktioniert, sondern bei Problemen des autonomen Fahrens auch zu besseren Ergebnissen führt.

Abstract

Reinforcement Learning has gained a lot of popularity in recent years due to its capability to learn complex tasks from just a reward signal, especially after the extension to modern Deep Learning architectures. A number of improvements to the concept were introduced, two of them being distributional q-value approximation and Maximum Entropy Reinforcement Learning. In environments dealing with autonomous driving problems, both seem to have a benefit on performance.

In this thesis, these two methods are introduced by giving a short overview of previous work and the idea behind their combination is presented. Lastly, we will show through experiments in the CARLA simulator that this combination not only works but is generally superior in autonomous driving tasks.

Contents

Kurzfassung **ix**

Abstract **xi**

Introduction **1**

1 Reinforcement Learning Concepts **3**
- 1.1 Basic Definitions 3
- 1.2 Temporal Difference Learning 7
- 1.3 Policy Gradient Methods 10
- 1.4 Deep Reinforcement Learning 12

2 Distributional RL **15**
- 2.1 Categorical DQN 16
- 2.2 Quantile Regression DQN 19
- 2.3 Implicit Quantile Networks 24

3 Soft Actor-Critic **27**
- 3.1 Maximum Entropy Objective 28
- 3.2 Theoretical Analysis 29
- 3.3 Soft Actor-Critic, a Practical Approximation to Soft Policy Iteration . 31
- 3.4 Automatic Entropy Temperature Tuning 32
- 3.5 Discrete Variant 33
- 3.6 Distributional Extension 35

4 Experimental Results **39**
- 4.1 Implementation Details 39
- 4.2 Overview of the Environments 40
- 4.3 Results 43

5 Conclusion **51**

A Hyperparameters **53**
- A.1 Stability – DQNs 53

A.2 Stability – SACs 56
A.3 Cruise Control 59
A.4 Discrete Lane Keeping 65
A.5 Discrete Lane Keeping – Lower Capacities 71
A.6 Continuous Lane Keeping 73
A.7 Combined Adaptive 76

Bibliography **79**

Introduction

I was first introduced to Reinforcement Learning during a project concerning autonomous driving in the CARLA simulator, [Dosovitskiy et al., 2017]. The way a car was able to learn how to drive just from being told how well it was doing, even using only classical algorithms that are now decades old, fascinated me. I then proceeded to dive into more recent techniques using deep learning and distributional approximation methods, and with that came a huge leap in performance, previously impossible challenges were easy for those algorithms. I was then introduced to the Soft Actor Critic algorithm, the probably most important representant of Maximum Entropy Reinforcement Learning, which has an approach to exploration of the environment that is very well suited for autonomous driving tasks. From this came the idea to combine the two, and the topic for this thesis was decided.

Chapter 1 will serve as an introduction to the topic of Reinforcement Learning, presenting a few selected topics we will need later on. Chapter 2 and 3 will describe the methods from the title, distributional q-value approximation and Maximum Entropy Reinforcement Learning's most important representative, Soft Actor Critic. The theory behind those methods will be presented with some select proofs to provide some insight into the inner workings. After combining the two concepts at the end of Chapter 3, Chapter 4 will feature some experimental results mainly around autonomous driving, and discussion of these results.

A quick note on notation: This thesis combines quite a few books and papers, which all use different notations. In the interest of uniformity, a common notation will be used for all concepts, not the notation of the original work cited.

Furthermore, while striving to use as few abbreviations as possible, there are a couple we will use throughout the thesis: PMF (probability mass function), PDF (probability density function), and CDF (cumulative distribution function). Also, we will usually refer to the algorithms in the established way, so e.g. for DQN we will always use the abbreviation after introducing it.

CHAPTER 1

Reinforcement Learning Concepts

This chapter aims to first briefly introduce the basic ideas and terms behind Reinforcement Learning and then extend them to use a modern Deep Learning framework.

1.1 Basic Definitions

All definitions and results of this and the two following sections are taken from [Sutton and Barto, 2018], although sometimes generalized a bit to accommodate extensions used in later chapters. For some of the proofs, additional detail is added.

1.1.1 Markov Decision Processes

The *Markov Decision Process*, or *MDP* for short, is a formalization of the type of problem we want to solve with Reinforcement Learning. Informally they describe an environment with discrete time steps, with which an agent can interact by executing an action that in turn modifies the environment and yields a reward for the agent, see Figure 1.1. The goal of Reinforcement Learning will be to make the agent choose actions in a way that maximizes the sum of those rewards.

Definition 1.1 (Markov Decision Process)**.** An *MDP* is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{S}_0, p, \gamma)$, where

- $\mathcal{S}$ is a set of states,
- $\mathcal{A}$ is a set of actions,
- $\mathcal{S}_0 \subseteq \mathcal{S}$ is a set of starting states,
- $p\colon\ \mathcal{S} \times [r_{\min}, r_{\max}] \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, $r_{\min} < r_{\max} \in \mathbb{R}$ is the so-called *dynamics function* or *transition function*, representing the density function of the transition from a state-action pair (s, a) to next state s' and reward r,

- $\gamma \in [0, 1]$ is the *discount factor*.

We call a (finite or infinite) series of random variables

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$$

of states S_t, actions A_t and rewards R_t a *trajectory*, with support of S_0 in $\mathcal{S}_0$. For all trajectories generated by the MDP, we require

$$f(s', r | S_{t-1} = s, A_{t-1} = a') = f(s', r | S_i = s_i, A_i = a_i, i \in I),$$

for all $s, s_i \in \mathcal{S}, a_i \in \mathcal{A}$, $I \subseteq \{0, \ldots, t-1\}, t-1 \in I$, where f is the (conditional) CDF of the state-reward distribution, i.e. the probability of transitioning from a state-action pair to the next state-reward pair may only depend on the last time step, not the whole history of steps. We call this the *Markov Property* of the MDP. Note that p would not be well-defined if the Markov Property were not satisfied.

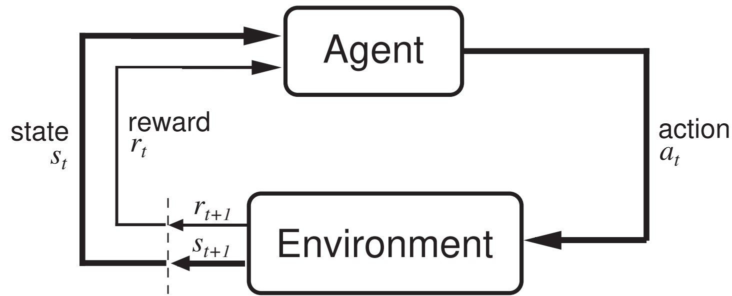

Figure 1.1: The Agent-Environment Interaction. [Sutton and Barto, 2018] [Sutton, 2023]

Definition 1.2 (Episodes)**.** Let $(\mathcal{S}, \mathcal{A}, \mathcal{S}_0, p, \gamma)$ be an MDP. If there exists a set $\mathcal{S}_\dagger \subseteq \mathcal{S}$, such that all trajectories of the MDP are finite and end in a state $s \in \mathcal{S}_\dagger$ and no states of $\mathcal{S}_\dagger$ occur before the end of a trajectory, we call the MDP *episodic* (or say the MDP represents an *episodic task*) and call $\mathcal{S}_\dagger$ the terminal states.

In the real world, many applications are not episodic, e.g. many control tasks can theoretically go on forever, like a car that never crashes. We call these types of tasks *continuing*. For most of the algorithms and ideas presented here, the notion of episodes is not crucially important, but still a helpful concept.

1.1.2 Return, Value- and q-Function

We now have the terminology to talk about the goal of Reinforcement Learning more formally:

Definition 1.3 (Return)**.** Given a trajectory $S_0, A_0, R_1, S_1, A_1, \ldots$ of an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{S}_0, p, \gamma)$, we call the random variable

$$G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+i+1}$$

the *return.* Since the reward is required to be bounded, G_t is also bounded if $\gamma < 1$. For episodic tasks, this sum is always finite and thus also exists for $\gamma = 1$.

Definition 1.4 (Policy)**.** A function π, that maps a state $s \in \mathcal{S}$ to a probability distribution over $\mathcal{A}$ is called a *policy.*

Definition 1.5 (q- and Value-Function)**.** Let $(\mathcal{S}, \mathcal{A}, \mathcal{S}_0, p, \gamma)$ be an MDP, π a policy. The function

$$v_\pi \colon \mathcal{S} \to \mathbb{R}; \quad s \mapsto \mathbb{E}_{A_t \sim \pi, S_t, R_{t+1} \sim p}\left[G_t \middle| S_t = s\right]$$

is called the *value function* of state s under policy π. The function

$$q_\pi \colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}; \quad (s, a) \mapsto \mathbb{E}_{A_t \sim \pi, S_t, R_{t+1} \sim p}\left[G_t \middle| S_t = s, A_t = a\right]$$

is called the *action-value function* (or simply q-function) of state action pair (s, a) under policy π.

Using these notions we can now define what it means to solve a Reinforcement Learning problem:

Definition 1.6 (Optimal Policy)**.** Let Π be the set of all policies of an MDP. If for some $\pi^* \in \Pi$ the condition

$$\forall \pi \in \Pi \colon \quad \forall s \in \mathcal{S} \colon \quad v_{\pi^*}(s) \geq v_\pi(s)$$

holds, we call π^* an *optimal policy* or *solution* of the MDP.

Instead of the value function, we could have also used the q-function in the definition:

Theorem 1.7. *$\pi^* \in \Pi$ is optimal iff*

$$\forall \pi \in \Pi \colon \quad \forall (s, a) \in \mathcal{S} \times \mathcal{A} \colon \quad q_{\pi^*}(s, a) \geq q_\pi(s, a),$$

Proof. The statement follows immediately from

$$v_\pi(s) = \mathbb{E}_{A \sim \pi}[q_\pi(s, A)] \tag{1.1}$$

and

$$q_\pi(s, a) = \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a]. \tag{1.2}$$

□

By plugging π^* into equation (1.1), we see that

$$\begin{aligned} v_{\pi^*}(s) &= \mathbb{E}_{A\sim\pi^*}[q_\pi(s,A)] \\ &= \max_{a\in\mathcal{A}} q_\pi(s,A), \end{aligned}$$

because otherwise π^* could be modified to always choose such a maximal action, which would increase v_{π^*} (see the proof of Theorem 1.9). Combining this with (1.2), we get:

Theorem 1.8 (Bellman Equation)**.** *An optimal policy π^* satisfies*

$$v_{\pi^*}(s) = \max_{a\in\mathcal{A}} \mathbb{E}[R_{t+1} + \gamma v_{\pi^*}(S_{t+1}) | S_t = s, A_t = a]$$

and

$$q_{\pi^*}(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'\in\mathcal{A}} q_{\pi^*}(S_{t+1},a') | S_t = s, A_t = a]. \tag{1.3}$$

This equation will become very useful in the following chapters as it allows us to reformulate the search for the q- and value-function of the optimal policy as a fixed point problem. From there we can construct a (deterministic) policy as

$$\pi_q(s) := \arg\max_{a\in\mathcal{A}} q(s,a). \tag{1.4}$$

There still is the question of whether an optimal policy always exists. In the generality of our MDP definition, the answer is *no* (consider an MDP with just one state, action space $[0,1)$ and deterministic reward $r(a)=a$), but the following important special case holds:

Theorem 1.9. *Let $(\mathcal{S},\mathcal{A},\mathcal{S}_0,p,\gamma), \gamma<1$ be a finite MDP (states, actions and possible rewards are finite), then an optimal deterministic policy exists.*

Proof. Let π_1,π_2 be two deterministic policies. Define π by

$$\pi(s) = \begin{cases} \pi_1(s), & \text{if } v_{\pi_1}(s) \geq v_{\pi_2}(s), \\ \pi_2(s), & \text{otherwise.} \end{cases}$$

Then we have

$$\begin{aligned} v_{\pi_1}(s) &\leq q_{\pi_1}(s,\pi(s)) \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma v_{\pi_1}(S_{t+1}) | S_t = s\right] \\ &\leq \mathbb{E}\left[R_{t+1} + \gamma q_{\pi_1}(S_{t+1},\pi(S_{t+1})) | S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_{\pi_1}(S_{t+2}) | S_t = s\right], \end{aligned}$$

so by induction

$$v_{\pi_1}(s) \leq \mathbb{E}_\pi\left[\sum_{n=1}^{N} \gamma^{n-1} R_{t+n} + \gamma^N v_{\pi_1}(S_{t+N+1}) | S_t = s\right] \qquad \forall N\in\mathbb{N}.$$

Since the reward is bounded and $\gamma < 1$, we have $\gamma^N v_{\pi_1}(S_{t+N+1}) \to 0$, so the right side converges to $v_\pi(s)$, proving $v_{\pi_1}(s) \leq v_\pi(s)$.

For any two uncomparable (by their value functions) policies, there exists an upper bound policy. Since for any finite MDP, there are only finitely many deterministic policies, we arrive at a maximum in finite steps by repeatedly applying this maximization operation to pairs of policies.

By using the same arguments as above, we can show that for any stochastic policy π_{stoch}, the (deterministic) policy defined by

$$\pi(s) := \arg\max_{a \in \mathcal{A}} q_{\pi_{\text{stoch}}}(s, a)$$

fulfills $v_{\pi_{\text{stoch}}} \leq v_\pi$, so the maximal deterministic policy is the optimal policy we were looking for. □

1.2 Temporal Difference Learning

We already know how to construct a policy when given an optimal q function, so let us look at how to approximate this q-function: From the definition

$$q_\pi(s, a) = \mathbb{E}_{A_t \sim \pi, S_t, R_{t+1} \sim p} \left[G_t | S_t = s, A_t = a \right],$$

we could just generate trajectories, for each (s, a) calculate the respective returns G_n starting in state s with action a and then approximate the q function by

$$\begin{aligned} q_n(s, a) &= \frac{1}{n} \sum_{k=1}^{n} G_k \\ &= \frac{1}{n} \sum_{k=1}^{n-1} G_k + \frac{1}{n} G_n \\ &= \frac{n-1}{n} \frac{1}{n-1} \sum_{k=1}^{n-1} G_k + G_n \\ &= \frac{1}{n-1} \sum_{k=1}^{n-1} G_k + \frac{1}{n} \left(G_n - \frac{1}{n-1} \sum_{k=1}^{n-1} G_k \right) \\ &= q_{n-1}(s, a) + \frac{1}{n} \left(G_n - q_{n-1}(s, a) \right), \end{aligned}$$

giving us an incremental update rule. This is called a Monte-Carlo update. While yielding an interesting class of algorithms, it has two main disadvantages:

1. we have to keep track of the number of updates for each state-action pair and

2. we still have to generate the whole episode to calculate the return.

The first disadvantage is resolved by replacing the update rule by

$$q(s,a) \leftarrow q(s,a) + \alpha\left(G - q(s,a)\right)$$

for some α called the learning rate. (We would still call this a Monte-Carlo update!)

The second disadvantage is solved by replacing G by the estimate $R_{t+1} + \gamma q(S_{t+1}, A)$ for A sampled from π to obtain

$$q(s,a) \leftarrow q(s,a) + \alpha\left(R_{t+1} + \gamma q(S_{t+1}, A) - q(s,a)\right). \tag{1.5}$$

This replacement of G by an estimate from the reward and the current approximation of the q function is called a temporal difference learning update, with this specific approach of using the sampled action in the update rule called a SARSA update (because we need the current state and action, the reward we got and the next state and sampled action).

We could combine this update rule with (1.4) and expect it to learn an optimal policy over time. Unfortunately, this will not be the case in the vast majority of running this algorithm, as we are currently missing one crucial component of Reinforcement Learning: *exploration*!

Consider the following scenario depicted in Figure 1.2: An agent (the blue dot) placed in a one-dimensional grid world. Its actions are moving left and right, the left-most and right-most cells being terminal, and moving onto the left-most cell gives a reward of 1, moving onto the rightmost cell gives a reward of 2, and all other actions give a reward of 0. For simplicity let $\gamma = 1$. Looking at the policy that always moves left, the algorithm described above will never update and be stuck in a suboptimal policy, as it never sees the better option of moving right.

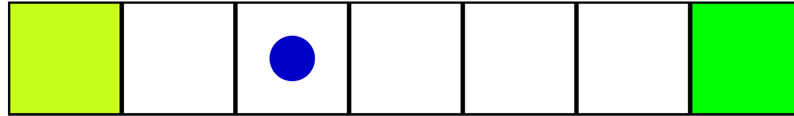

Figure 1.2: An example of a 1d grid world.

We call this the *exploration vs. exploitation* problem. A Reinforcement Learning algorithm has to explore so as not to get stuck in suboptimal policies, but if it explores too much, it will not use the knowledge gained, never generating good rewards. So a good algorithm has to strike a balance between gaining new knowledge by *exploring* vs. *exploiting* the gained knowledge to get good rewards.

We will return to different solutions to this problem multiple times in this thesis, but for now, we can solve it by replacing (1.4) by

$$\pi_q^{\varepsilon}(s) := \begin{cases} \arg\max_{a \in \mathcal{A}} q(s,a) & \text{with probability } 1-\varepsilon, \\ \text{a random } a \in \mathcal{A} & \text{with probability } \varepsilon \end{cases}$$

for some $0 \leq \varepsilon \leq 1$. We call π_q^ε an ε-*greedy* policy. This yields Algorithm 1.1, which we call *SARSA*.

Initialize: $q(s, a)$ for all states/actions, $q(s_t, a) = 0$ for all terminal s_t
1 **foreach** *episode* **do**
2 | Sample start state s;
3 | Choose a according to π_q^ε;
4 | **while** s *not terminal* **do**
5 | | Take action a, observe reward r, next state s';
6 | | Choose a' according to π_q^ε;
7 | | $q(s, a) \leftarrow q(s, a) + \alpha\,(r + \gamma q(s', a') - q(s, a))$;
8 | | $s \leftarrow s'$;
9 | | $a \leftarrow a'$;
10 | **end**
11 **end**

Algorithm 1.1: The SARSA algorithm. [Sutton and Barto, 2018]

A very important variation of this algorithm is *Q-Learning*, shown in Algorithm 1.2. The difference between the two is that the q-value used in the update rule does not originate from the action sampled by the ε-greedy policy, but rather from the action retrieved from (1.4). Therefore, the approximated q-function is not that of the policy used to generate the trajectories, but rather of the corresponding deterministic policy. Thus, we call Q-Learning an *off-policy* algorithm, while SARSA is called *on-policy*. Off-policy algorithms have the advantage of being easily extendable to using experience replay, which we will return to when discussing the use of artificial neural nets in Reinforcement Learning.

Initialize: $q(s, a)$ for all states/actions, $q(s_t, a) = 0$ for all terminal s_t
1 **foreach** *episode* **do**
2 | Sample start state s;
3 | **while** s *not terminal* **do**
4 | | Choose a according to π_q^ε;
5 | | Take action a, observe reward r, next state s';
6 | | $q(s, a) \leftarrow q(s, a) + \alpha\,(r + \gamma \max_{a' \in \mathcal{A}} q(s', a') - q(s, a))$;
7 | | $s \leftarrow s'$;
8 | **end**
9 **end**

Algorithm 1.2: The algorithm Q-Learning. [Sutton and Barto, 2018]

1.3 Policy Gradient Methods

Another rich class of Reinforcement Learning algorithms are the so-called *Policy Gradient Methods*. Here, instead of using (1.4) for the definition of an ε-greedy policy, we directly obtain a stochastic policy from the q-function that is in some way proportional to the q-values for each action. For simplicity let $\gamma = 1$ in this section except in the final algorithm pseudocode.

To do this, we need some

$$\pi\colon\ \mathcal{S} \times \mathbb{R}^d \to \mathcal{D}(\mathcal{A}),$$

where $\mathcal{D}(\mathcal{A})$ are all distributions over $\mathcal{A}$. We can interpret this π as a function that yields a policy for any $\theta \in \mathbb{R}^d$. We therefore also write π_θ for $\pi(\cdot, \theta)$.

Our goal is to maximize the return, i.e. maximize

$$J(\theta) = v_{\pi_\theta}(s).$$

If J is differentiable, we can use gradient ascent,

$$\theta \leftarrow \theta + \alpha \nabla J(\theta),$$

to improve our policy. While this gradient may seem unfeasible in practice, the following theorem greatly simplifies its calculation:

Theorem 1.10 (Policy Gradient Theorem)**.** *Let $\mathcal{S}$ be a discrete space, $b\colon \mathcal{S} \to \mathbb{R}$ an arbitrary function (called* baseline*). Let the image of π contain only discrete distributions, $p(s, a, \theta)$ the PMF of $\pi(s, \theta)$, then*

$$\nabla J(\theta) \propto \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} (q_{\pi_\theta}(s, a) - b(s)) \nabla p(s, a, \theta),$$

where μ is the PMF of the occurrences of the states under the policy $\pi_\mathcal{T}$. Similarly, if the image of π contains only distributions that have a PDF (denoted by $f(s, a, \theta)$),

$$\nabla J(\theta) \propto \sum_{s \in \mathcal{S}} \mu(s) \int_{\mathcal{A}} (q_{\pi_\theta}(s, a) - b(s)) \nabla f(s, a, \theta)\ da,$$

holds. If $\mathcal{S}$ is a subspace of $\mathbb{R}^n$, the sums over $\mathcal{S}$ can be replaced by integrals.

Proof. The proof of the theorem with $b \equiv 0$ is a straightforward but somewhat lengthy calculation, so the reader is referred to Section 13.2 of [Sutton and Barto, 2018]. For

$b \neq 0$ note

$$\begin{aligned}\nabla J = \nabla v_\pi(s) &= \nabla \left[\sum_{a\in\mathcal{A}} \pi(s,a) q_\pi(s,a)\right] \\ &= \nabla \left[\sum_{a\in\mathcal{A}} \pi(s,a) q_\pi(s,a) - b(s)\right] \\ &= \nabla \left[\sum_{a\in\mathcal{A}} \pi(s,a) q_\pi(s,a) - b(s)\sum_{a\in\mathcal{A}} \pi(s,a)\right] \\ &= \nabla \left[\sum_{a\in\mathcal{A}} \pi(s,a)(q_\pi(s,a) - b(s))\right]\end{aligned}$$

and similarly for continuous distributions. □

Note that by multiplying with $\frac{p(s,a,\theta)}{p(s,a,\theta)}$ (or $\frac{f(s,a,\theta)}{f(s,a,\theta)}$ in the continuous case) we can reformulate the gradient as

$$\nabla J(\theta) \propto \mathbb{E}_{\pi_\theta}\left[(G - b(s))\nabla \log p(s,a,\theta)\right]$$

with return G. This gives us an easily implementable update rule by estimating this expected value. We see that the gradient is small if the baseline is close to the expected return. This gives rise to the idea of simultaneously approximating v_π and using it as the baseline. The resulting algorithm is called Actor-Critic (because the value function is used as a critic to "evaluate" how good the actor, i.e. the policy is) and is depicted in Algorithm 1.3, where the TD error (1.5) is used to obtain the gradient for the value function.

Initialize: $\pi(s,a,\theta)$: a differentiable policy parametrization
Initialize: $v(s,\tau)$: a differentiable state-value function parametrization, $v(s_t,\tau) = 0$ for terminal states

```
1  foreach episode do
2      Sample start state s;
3      while s not terminal do
4          Let n be the current step number;
5          Choose a according to π_θ;
6          Take action a, observe reward r, next state s';
7          δ ← R + γv(s', τ) − v(s, τ);
8          τ ← τ + αδ∇v(s, τ);
9          θ ← θ + αγ^n δ∇ log π(s, a, θ);
10         s ← s';
11     end
12 end
```

Algorithm 1.3: The algorithm Actor-Critic, π is identified with its PMF/PDF. [Sutton and Barto, 2018]

To conclude this section, let us look at two important examples of how to choose such a π in practice:

- Let $\mathcal{A} = \{a_1, \ldots, a_n\}$ be a discrete space. Choose any differentiable

  $$h(s, a, \theta) \colon \ \mathcal{S} \times \mathcal{A} \times R^d \to \mathbb{R},$$

  e.g. a neural net with input $\mathcal{S} \times \mathcal{A}$ or just a linear function $\theta \cdot x(s, a)$ for some feature extraction $x \colon \ \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$. Then

  $$\pi(s, a, \theta) := \frac{e^{h(s,a,\theta)}}{\sum_{i=1}^{n} e^{h(s,a_i,\theta)}}$$

  defines a PMF.

- Let $\mathcal{A} = \mathbb{R}$. Let $h \colon \ \mathcal{S} \times R^d \to \mathbb{R} \times \mathbb{R}^+; \ \ (s, \theta) \mapsto (\mu_\theta(s), \sigma_\theta(s))$ be any differentiable function, then

  $$\pi \colon \ \mathcal{S} \times \mathbb{R}^d \to \mathcal{D}(\mathcal{A}); \quad (s, \theta) \mapsto \mathcal{N}(\mu_\theta(s), \sigma_\theta(s)),$$

  where $\mathcal{N}(\mu, \sigma)$ denotes a normal distribution with mean μ and standard deviation σ. This can be extended in a straightforward manner to $\mathcal{A} = \mathbb{R}^d$ using diagonal normal distributions, i.e. normal distributions in each dimension of $\mathcal{A}$.

1.4 Deep Reinforcement Learning

So far we have mentioned the usage of artificial neural nets as the policy and as the value function in Actor-Critic. These share the problems described in this section, and we will get back to them in Chapter 3, but for now, we will focus on the extension of Q-Learning (see Algorithm 1.2) to Deep Learning techniques. This section is based on [Mnih et al., 2015].

If we just replaced q in Algorithm 1.2 by a neural net parametrized by θ and use (1.5) for gradient descent as we did in Actor-Critic (Algorithm 1.3), we would see it diverging miserably when applied to more complex environments. The major reasons for this are:

1. Typically consecutive states and actions in an episode are highly correlated, leading to biases when applying gradient descent updates to the networks in the order the experiences occurred.

2. Even small updates to the q-network can change the policy at a given state by the definition of π_q^ε. Coupled with the fact that every update changes the q-function globally, not just in the state the update originated from, this may lead to very unstable policies changing greatly with updates.

3. The update for $q(s, a)$ uses the target value $r + \gamma q(s', a')$, making these two highly correlated. Since the update for $q(s, a)$ also changes $q(s', a')$ this may lead to divergence of the network.

To address these issues we use the following two techniques:

- Experience decorrelation with uniform sampling from a buffer: Instead of updating the network in each step using the current experience as we did in Algorithm 1.2, we store all trajectories the agent has experienced (or at least a lot of them) in a replay buffer, and update by uniformly sampling a certain number of experiences (a minibatch) from the buffer and calculating the gradient of the mean error obtained from them. This reduces problem 1 greatly.
- Introduction of a target network to stabilize updates: Use two sets of parameters, θ^- and θ, where θ^- determines the policy and τ is updated according to the error

$$\left(r + \gamma \max_{a' \in \mathcal{A}} q(s', a', \theta^-) - q(s, a, \theta)\right)^2,$$

  and every $C \in \mathbb{N}$ steps θ^- is set to θ. This introduces a kind of inertia to the policy and the updates, keeping the q-values stable for some time. This greatly mitigates problems 2 and 3.

The resulting algorithm is called *Deep Q-Learning* (or *DQN* for short) and is depicted in Algorithm 1.4. Note that gradient descent is replaced by the Adam optimizer, see [Kingma and Ba, 2014]. In the original paper, RMSProp, [Geoffrey Hinton, 2012], was used, which Adam usually outperforms, so for the sake of comparability all Deep Reinforcement Learning algorithms described in this thesis will use Adam.

```
    Initialize: Minibatchsize n, inertia C
    Initialize: θ⁻, θ: parameters of the neural net q
    Initialize: Buffer B of fixed size
 1  foreach episode do
 2  |  Sample start state s;
 3  |  while s not terminal do
 4  |  |  Choose a according to π^ε_{θ⁻};
 5  |  |  Take action a, observe reward r, next state s', store (s, a, r, s') in B;
 6  |  |  Sample minibatch M of size n from B;
 7  |  |  L ← (1/n) Σ_{i=1}^{n} (r_i + γ max_{a'∈A} q(s'_i, a', θ⁻) − q(s_i, a_i, θ))^2;
 8  |  |  Update θ by using Adam, minimizing L;
 9  |  |  s ← s';
10  |  |  if C steps taken then
11  |  |  |  θ⁻ ← θ;
12  |  |  end
13  |  end
14  end
```

Algorithm 1.4: The algorithm DQN.

CHAPTER 2

Distributional Q-Value Approximation

So far we looked at the state- and action-value functions,

$$v_\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t R_{t+1} | S_0 = s\right],$$
$$q_\pi(s,a) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t R_{t+1} | S_0 = s, A_0 = a\right]$$

and how to approximate an optimal policy from them. In this chapter, we will have a closer look at the random variable

$$Z^\pi(s,a) := \sum_{t=0}^{\infty}\gamma^t R_{t+1},$$

its distribution, and different ways to approximate it.

We will start with a bit of historical context in looking at C51, the first algorithm to utilize distributional q-value approximation. Then we will have a more in-depth look at QRDQN, the first algorithm with a theoretically sound basis in the category. Lastly, we will extend the idea of QRDQN to IQN. The basis for all these algorithms will be DQN, described in Section 1.4.

In the interest of uniform notation, this whole chapter will use the notation of [Dabney et al., 2018b]. Also, we always assume an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{S}_0, p, \gamma)$ with a discrete action space in this chapter.

2.1 Categorical DQN

In [Bellemare et al., 2017], the authors first explore the idea of distributional approximation of the return-value distribution, developing a theoretical framework around it. They use a categorical distribution with 51 atoms as their representation, hence the name Categorical DQN or C51, which performs very well in practice, but does not completely fit the theory yet. In this section, we will explore this theory and give a sketch of the algorithm, so all the results are taken from [Bellemare et al., 2017].

2.1.1 The Distributional Bellman Operator

First, note that

$$Z(s,a) \stackrel{D}{=} R(s,a) + \gamma P^{\pi} Z(s,a),$$

where $R(s,a)$ is the random variable describing the reward gained in state s taking action a, $\stackrel{D}{=}$ means equality in distribution and P^{π} is defined as

$$\begin{gathered} P^{\pi} Z(s,a) \stackrel{D}{=} Z(s',a') \\ s' \sim p(s,a), \quad a' \sim \pi(s). \end{gathered}$$

We denote the space of all such Z by $\mathcal{Z}$. Now using the Bellman equation (1.3),

$$\begin{aligned} \mathbb{E}\left[Z^{\pi^*}(s,a)\right] &= q_{\pi^*}(s,a) \\ &= \mathbb{E}[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} q_{\pi^*}(S_{t+1}, a') | S_t = s, A_t = a] \\ &= \mathbb{E}[R(s,a)] + \gamma \max_{a' \in \mathcal{A}} \mathbb{E}_{s' \sim p}[Z^{\pi^*}(s',a') | S_t = s, A_t = a]. \end{aligned}$$

We see that not only the q function of the optimal policy but also the underlying distribution satisfies the equation

$$Z^{\pi^*}(s,a) \stackrel{D}{=} R(s,a) + \gamma \max_{a' \in \mathcal{A}} Z^{\pi^*}(s',a').$$

We denote the set of all Z representing the return value distribution of an optimal policy by $\mathcal{Z}^*$.

From this, we can define:

Definition 2.1. The *Distributional Bellman Operator* is defined as

$$\mathcal{T}^{\pi} Z(s,a) :\stackrel{D}{=} R(s,a) + \gamma P^{\pi} Z(s,a),$$

where, again, $:\stackrel{D}{=}$ means that the random variable on the left is any random variable sharing the distribution with the right side.

Similar to the non-distributional Bellman equation, the distribution Z of the return when following π is given by a fixed point of $\mathcal{T}^{\pi}$, so using it will enable us to perform policy evaluation. For analysis of the control setting, we will use the following operator:

Definition 2.2. Let $\mathcal{G}_Z$ be the set of *greedy policies* regarding Z, i.e. all policies maximizing the expected value:

$$\mathcal{G}_Z := \{\pi \in \Pi : \sum_{a \in \mathcal{A}} \pi(s,a)\mathbb{E}\left[Z(s,a)\right] = \max_{a' \in \mathcal{A}} \mathbb{E}\left[Z(s,a')\right]\}.$$

Then the *Distributional Bellman Optimality Operator* is defined as any $\mathcal{T}$ satisfying

$$\mathcal{T}Z \overset{D}{=} \mathcal{T}^{\pi}Z \quad \text{for some } \pi \in \mathcal{G}_Z,$$

i.e. $\mathcal{T}$ realizes a policy evaluation step using some greedy policy regarding the current Z.

2.1.2 The Wasserstein Metric

In the next section, we want to check whether or not the operators $\mathcal{T}$ and $\mathcal{T}^{\pi}$ satisfy the Banach fixed-point theorem. We, therefore, need a metric on distributions:

Definition 2.3. For random variables U, Y with respective CDFs F_Y, F_U, we define the *p-Wasserstein distance* (or *p-Wasserstein metric*) as

$$W_p(U,Y) = \left(\int_0^1 |F_Y^{-1}(\omega) - F_U^{-1}(\omega)|^p \, d\omega\right)^{\frac{1}{p}}.$$

For $p = \infty$ we set

$$W_\infty(U,Y) = \sup_{\omega \in [0,1]} |F_Y^{-1}(\omega) - F_U^{-1}(\omega)|.$$

That the Wasserstein distance is a metric follows directly from the fact that it is the L_p metric on the inverse CDFs of the random variables, positivity follows from the fact that CDFs are right-continuous.

Having a notion of distances between distributions, we can extend this idea to define a metric over value distributions $Z(s,a)$, enabling convergence analysis of the operators $\mathcal{T}^{\pi}$ and $\mathcal{T}$:

Definition 2.4. Let $Z_1, Z_2 \in \mathcal{Z}$ be two value distributions, then we define

$$\bar{d}_p(Z_1, Z_2) := \sup_{s \in \mathcal{S},\ a \in \mathcal{A}} W_p(Z_1(s,a), Z_2(s,a)).$$

2.1.3 Convergence Analysis

The first result concerns policy evaluation:

Theorem 2.5. *The operator $\mathcal{T}^{\pi}$ is a γ-contraction in $\bar{d}_p$.*

Proof. Let $Z_1, Z_2 \in \mathcal{Z}$ be two value distributions. Then

$$\begin{aligned}
\bar{d}_p(\mathcal{T}^\pi Z_1, \mathcal{T}^\pi Z_2) &= \sup_{s,a} d_p(\mathcal{T}^\pi Z_1(s,a), \mathcal{T}^\pi Z_2(s,a)) \\
&= \sup_{s,a} d_p(R(s,a) + \gamma P^\pi Z_1(s,a), R(s,a) + \gamma P^\pi Z_2(s,a)) \\
&\overset{(*)}{\leq} \gamma \sup_{s,a} d_p(P^\pi Z_1(s,a), P^\pi Z_2(s,a)) \\
&\leq \gamma \sup_{s',a'} d_p(Z_1(s',a'), Z_2(s',a')) \\
&= \bar{d}_p(Z_1(s',a'), Z_2(s',a')),
\end{aligned}$$

where $(*)$ holds because

$$d_p(A + \gamma U, A + \gamma V) \leq \gamma d_p(U, V)$$

for independent random variables A, U, V. □

So $\mathcal{T}^\pi$ has a unique fixed point and iteration of $\mathcal{T}^\pi$ on any $Z \in \mathcal{Z}$ converges to this fixed point. By definition of $\mathcal{T}^\pi$, this fixed point represents for each state-action pair (s,a) a distribution with expected value $q_\pi(s,a)$.

Unfortunately, the same is not true for the optimality operator:

Theorem 2.6. *The operator $\mathcal{T}$ is not a contraction.*

But still, the following result holds:

Theorem 2.7. *Denote by $\mathcal{Z}^{**}$ the set of value distributions that correspond to a sequence of optimal policies. Let $Z_k = \mathcal{T} Z_{k-1}$ with $Z_0 \in \mathcal{Z}$. Then*

$$\lim_{k \to \infty} \inf_{Z^{**} \in \mathcal{Z}^{**}(s,a)} d_p(Z_k(s,a), Z^{**}(s,a)) = 0 \quad \forall s \in \mathcal{S},\ \forall a \in \mathcal{A}.$$

If $\mathcal{S}$ is finite, then Z_k converges to $\mathcal{Z}^{**}$ uniformly.

Furthermore, if there is a total ordering $\prec$ on the set of optimal policies Π^*, such that for any $Z^* \in \mathcal{Z}^*$

$$\mathcal{T} Z^* = T^\pi Z^* \text{ with } \pi \in \mathcal{G}_{Z^*}, \pi \prec \pi' \forall \pi' \in \mathcal{G}_{Z^*} \setminus \{\pi\},$$

then $\mathcal{T}$ has a unique fixed point $Z^* \in \mathcal{Z}^*$.

We see that the control setting is much less well-behaved than the policy evaluation setting, not even guaranteeing a fixed point.

2.1.4 The Algorithm C51

Since this section's purpose is to motivate ideas of Distributional Reinforcement Learning and to give historical context, we will just sketch the algorithm based on the above idea, for the full details see the original paper [Bellemare et al., 2017]. First, choose $V_{\min}, V_{\max} \in \mathbb{R}$ and $N \in \mathbb{N}$. By $\Delta z := (V_{\max} - V_{\min})/(N-1)$ we get the atoms

$$z_i := V_{\min} + i\Delta z, \qquad 0 \leq i < N.$$

Next choose some parametric model $h : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}^N$, then we can define Z_h by

$$Z_\theta(s,a) = z_i \quad \text{with probability} \quad p_i(s,a) := \frac{e^{h_i(s,a,\theta)}}{\sum_{j=1}^N e^{h(s,a,\theta)}},$$

so we model our distribution by using a discrete PMF (compare also the example in Section 1.3).

Since the support of $\mathcal{T}Z_\theta$ and Z_θ will almost always be mostly disjoint, see Figure 2.1, we will project the support of $\mathcal{T}Z_\theta$ onto the support of Z_θ. Denote this projection by $\Phi\mathcal{T}Z_\theta$.

By the theory we looked at so far, we would now want to minimize some Wasserstein distance. But in this setting, the Wasserstein distance cannot be used as a loss from samples, as the gradient obtained from sample trajectories will in general differ from the gradient of the underlying distribution. So for now, we will use the Kullback-Leibler divergence,

$$D_{\mathrm{KL}}(\Phi\mathcal{T}Z_{\tilde{\theta}}(s,a) \parallel Z_\theta(s,a))$$

as the loss function, where $\tilde{\theta}$ means we view θ as fixed, not contributing to the gradient. The Kullback-Leibler divergence is defined as

$$D_{\mathrm{KL}}(P \parallel Q) := \mathbb{E}_{x\sim P}\left[\log \frac{p(x)}{q(x)}\right],$$

where p, q are the densities of P, Q respectively. This creates a theory-practice gap, which we will set out to solve in the next section.

2.2 Quantile Regression DQN

Unsatisfied with the theory-practice gap of C51, the authors of [Dabney et al., 2018b] set out to create an algorithm that actually minimizes the Wasserstein distance. They do so by finding a clever representation of the distributions involved and by borrowing techniques from economics, both of which we will explore in this section. Again, if not otherwise stated, all results are from [Dabney et al., 2018b].

2.2.1 The Quantile Approximation

Instead of approximating the density function of the distribution, we will approximate the distribution function, or more precisely its inverse F^{-1}, the quantile function.

This has three main advantages:

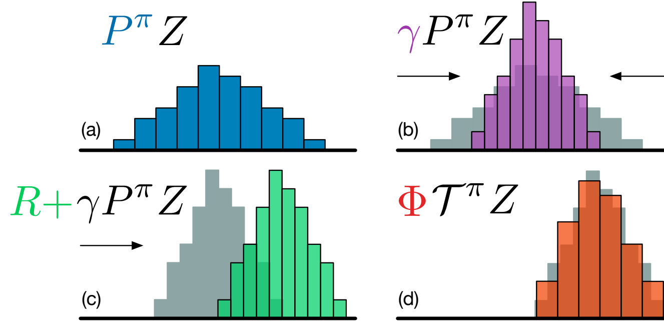

Figure 2.1: Visualization of a distributional Bellman operator with a deterministic reward function on a discrete distribution: (a) Next state distribution under policy π, (b) Discounting shrinks the distribution towards 0, (c) The reward shifts it, and (d) Projection step, taken from [Bellemare et al., 2017]

1. We do not have to know the support of the distribution of the return beforehand, as F^{-1} is always defined on $(0, 1)$, therefore eliminating the hyperparameters $V_{\min}, V_{\max}$.
2. As there is no issue with disjoint supports anymore, we do not need the projection step of C51.
3. We can now actually minimize the Wasserstein distance without problems with biased gradients when approximating using sample trajectories using quantile regression. We will discuss this in more detail in the next subsection.

Formally, we choose $N \in \mathbb{N}$ to get

$$\tau_i = \frac{i}{N}, \qquad i = 0, \ldots, N$$

Our approximation of F^{-1} is a step function that is constant between τ_i and τ_{i+1}, $i = 0, \ldots N-1$. Denote by $\mathcal{Z}_Q$ all distributions with such a quantile function.

The first question concerns how to best approximate any given distribution in $\mathcal{Z}_Q$. As we previously established, the Wasserstein distances yield a good notion for approximation distributions, so the question is solved by determining an explicit form of the projection onto $\mathcal{Z}_Q$,

$$\Pi_{W_1} Z := \arg \min_{Z_\theta \in \mathcal{Z}_Q} W_1(Z, Z_\theta).$$

Since $Z_\theta \in \mathcal{Z}_Q$ are piecewise constant, the answer is given by the following lemma:

Lemma 2.8. *For any $\tau, \tau' \in [0, 1]$ with $\tau < \tau'$ and CDF F with inverse F^{-1}, any θ with*

$$F(\theta) = \frac{\tau + \tau'}{2}$$

minimizes

$$\int_{\tau}^{\tau'} |F^{-1}(\omega) - \theta| \; d\omega.$$

Therefore $F^{-1}\left(\frac{\tau+\tau'}{2}\right)$ *is a valid minimizer, and if* F^{-1} *is continuous at* $\frac{\tau+\tau'}{2}$*, it is the unique minimizer.*

Defining the quantile midpoints as

$$\hat{\tau}_i := \frac{\tau_{i-1} + \tau_i}{2}, \qquad i = 1, \dots, N$$

the projection of $\Pi_{W_1} Z$ is given by the values

$$F^{-1}_{\Pi_{W_1} Z} = \theta_i := F^{-1}_Z(\hat{\tau}_i), \qquad i = 1, \dots, N,$$

where F^{-1}_Z denotes the quantile function of Z. The projection is visualized in Figure 2.2. We also denote by F^{-1}_θ any quantile function defined that way from a vector $\theta \in \mathbb{R}^N$.

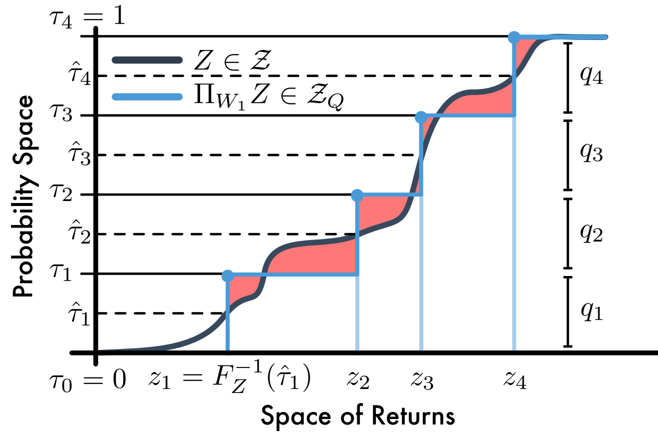

Figure 2.2: Visualization of the projection $\Pi_{W_1} Z$ with $N = 4$. The red regions show the 1-Wasserstein distance. Figure taken from [Dabney et al., 2018b]

We end the subsection with the result, that projecting in policy evaluation still attains a fixed point:

Theorem 2.9. *Let* $\mathcal{S}, \mathcal{A}$ *be countable, then*

$$\bar{d}_\infty \left(\Pi_{W_1} \mathcal{T}^\pi Z_1, \Pi_{W_1} \mathcal{T}^\pi Z_2\right) \leq \gamma \bar{d}_p(Z_1, Z_2)$$

for any $Z_1, Z_2 \in \mathcal{Z}$.

2.2.2 Quantile Huber Loss

Above we mentioned that one advantage of the quantile representation is that we can directly minimize the Wasserstein distance by using gradients from sampled experience. Unfortunately, this is not true directly, but there is a method for unbiased stochastic approximation of the quantile function, called *quantile regression*, which is a technique used in economics:

Definition 2.10. We call

$$\mathcal{L}_{\mathrm{QR}}^{\mathcal{T}}(\theta) := \mathbb{E}_{\hat{Z}\sim Z}[\rho_\tau(\hat{Z}-\theta)]$$

with

$$\rho_\tau\colon\ \mathbb{R}\to\mathbb{R};\quad u\mapsto u(\tau-\delta_{u<0})$$

the *quantile regression loss.*

This loss attains a minimum at the value $F_Z^{-1}(\tau)$ for a given distribution Z and quantile τ. However it is not smooth at zero, so we modify it using the Huber loss, [Huber, 1964],

$$\mathcal{L}_\kappa(u) = \begin{cases} \frac{1}{2}u^2, & \text{if } |u|\leq\kappa, \\ \kappa(|u|-\frac{1}{2}\kappa), & \text{otherwise,} \end{cases}$$

to better suit our needs, giving:

Definition 2.11. We call

$$\rho_\tau^\kappa(u) = |\tau-\delta_{u<0}|\mathcal{L}_\kappa(u)$$

the *quantile Huber loss.* Furthermore, we use $\rho_\tau^0 := \rho_\tau$.

So in summary, we can find the $\theta\in\mathbb{R}^N$ minimizing Wasserstein distance by minimizing the loss

$$\sum_{i=1}^N \mathbb{E}_{\hat{Z}\sim Z}[\rho_{\tau_i}^\kappa(\hat{Z}-\theta_i)]$$

which gives us unbiased sample gradients, enabling the usage of stochastic gradient descent.

2.2.3 The Full Algorithm

Now we can describe the algorithm based on distributional policy improvement using quantile representations:

1. Use a neural net

   $$\theta\colon\ \mathcal{S}\times\mathcal{A}\to\mathbb{R}^N$$

   to represent the quantile functions $Z_\theta\in\mathcal{Z}_Q$. In practice for discrete action spaces, it is usually more efficient to use $\theta\colon\mathcal{S}\to\mathbb{R}^{|\mathcal{A}|}\times\mathbb{R}^N$.

2. Use π_q^ε based on

$$q(s,a) = \mathbb{E}\left[Z_\theta(s,a)\right] = \frac{1}{N}\sum_{i=1}^{N}\theta(s,a)_i$$

to generate trajectories, and store these experiences in a buffer.

3. To update θ, sample experiences uniformly from the buffer to calculate

$$L = \sum_{i=1}^{N}\mathbb{E}_{\hat{Z}\sim Z}[\rho^\kappa_{\tau_i}(\mathcal{T}Z - \theta_i)] \approx \frac{1}{N}\sum_{j=1}^{N}\sum_{i=1}^{N}\rho^\kappa_{\hat{\tau}_i}(\mathcal{T}\theta_j - \theta_i),$$

where

$$\mathcal{T}\theta_j = r + \gamma\theta_j(s', a^*),$$
$$a^* = \arg\max_{a\in\mathcal{A}} q(s', a),$$

and run an Adam optimization step on L. Note that $\mathcal{T}\theta_j$ is implicitly treated as a constant when calculating ∇L, similarly to the temporal difference error we used in DQN.

Pseudocode is provided in Algorithm 2.1.

Initialize: Minibatchsize n, inertia C
Initialize: ξ^-, ξ: parameters of the neural net $\theta : \mathcal{S}\times\mathcal{A}\to\mathbb{R}^N$
Initialize: Function $q(s,a,\xi) = \frac{1}{N}\sum_{i=1}^{N}\theta(s,a,\xi)_i$
Initialize: Buffer B of fixed size

```
1  foreach episode do
2  |  Sample start state s;
3  |  while s not terminal do
4  |  |  Choose a according to π^ε_{q(·,·,ξ⁻)};
5  |  |  Take action a, observe reward r, next state s', store (s, a, r, s') in B;
6  |  |  Sample minibatch M of size n from B;
7  |  |  Sample actions a* ← arg max_{a∈A} q(s', a, ξ⁻) for each s' in M;
8  |  |  Tθ_j(r, s') ← r + γθ_j(s', a*, ξ⁻);
9  |  |  L(s, a, r, s') ← 1/N Σ_{j=1}^N Σ_{i=1}^N ρ^κ_{τ̂_i}(Tθ_j(r, s') − θ_i(s, a, ξ));
10 |  |  Update ξ by using Adam, minimizing the mean of L;
11 |  |  s ← s';
12 |  |  if C steps taken then
13 |  |  |  ξ⁻ ← ξ;
14 |  |  end
15 |  end
16 end
```

Algorithm 2.1: The algorithm QRDQN.

2.3 Implicit Quantile Networks

This section is based on [Dabney et al., 2018a].

In QRDQN, the quantile function F_Z^{-1} is parametrized as a step function, meaning that the error of our approximation is largely dependent on the hyperparameter N. We now aim to learn the full quantile function.

For notational simplicity, write $Z_\tau := F_Z^{-1}(\tau)$. We can now view Z_τ as a random variable given $\tau \sim U([0,1])$, leading to $Z_\tau(s,a) \sim Z(s,a)$. We can therefore view Z_τ as samples from the (implicitly defined) return distribution, giving rise to the name "Implicit Quantile Network" when approximating Z_τ by an artificial neural net.

This perception of Z_τ as a sample from the return distribution allows us to deduce

$$q(s,a) = \mathbb{E}[Z(s,a)] = \mathbb{E}_{\tau \sim U([0,1])}[Z_\tau(s,a)] \approx \frac{1}{K} \sum_{k=1}^{K} Z_{\tau_k}(s,a)$$

for $\tau_k \sim U([0,1])$, $k = 1, \ldots, K$. From this, we can define π_q^ε as usual.

So, lastly, we have a look at how to optimize: Given an experience (s, a, r, s') we define a temporal difference error similar to the one used in QRDQN,

$$\delta^{\tau,\tau'}(s,a,r,s') = r + \gamma Z_{\tau'}(s', \pi_q(s')) - Z_\tau(s,a).$$

Note, as usual, that $Z_{\tau'}(s', \pi_q(s'))$ is treated as a constant when differentiating.

This gives us, again similar to QRDQN, the IQN loss function

$$L(s,a,r,s') = \frac{1}{N'} \sum_{i=1}^{N} \sum_{j=1}^{N'} \rho_{\tau_i}^{\kappa}(\delta_t^{\tau_i,\tau_j'}(s,a,r,s'))$$

for $\tau_i, \tau_j' \sim U([0,1])$ for $i = 1, \ldots, N, j = 1, \ldots N'$, where $N, N' \in \mathbb{N}$ are hyperparameters. The full algorithm is provided in Algorithm 2.2.

2.3.1 A Note on Risk Sensitive RL

So far in this whole chapter, we have only talked about distributional q-value approximation as a means to faster or better convergence. But it also enables us, especially in the case of IQN, to transform the distribution when selecting an action. We could, for example, give more weight to low values of return, therefore giving bad experiences more weight, which would likely lead the agent to avoid actions that lead to high negative rewards, however unlikely they may be, therefore making it *risk-averse*. Similarly, we could put more weight on high values of return, making the agent shift its actions more towards high returns, ignoring possible low returns, therefore making it *risk-seeking*.

Initialize: Minibatchsize n, inertia C
Initialize: ξ^-, ξ: parameters of the neural net $\theta : \mathcal{S} \times \mathcal{A} \times \mathbb{R} \to \mathbb{R}$
Initialize: (Stochastic) Function $q(s, a, \xi) = \frac{1}{K} \sum_{k=1}^{K} \theta(s, a, \tau, \xi)$ for $\tau_i \sim U([0, 1])$
Initialize: Buffer B of fixed size

1 **foreach** *episode* **do**
2   Sample start state s;
3   **while** *s not terminal* **do**
4   Choose a according to $\pi^{\varepsilon}_{q(\cdot,\cdot,\xi^-)}$;
5   Take action a, observe reward r, next state s', store (s, a, r, s') in B;
6   Sample minibatch M of size n from B;
7   Sample actions $a^* \leftarrow \arg\max_{a \in \mathcal{A}} q(s', a, \xi^-)$ for each s' in M;
8   Sample $\tau_i, \tau'_j \sim U([0, 1])$, $i = 1, \ldots, N$, $j = 1, \ldots, N'$;
9         $\delta^{\tau_i, \tau'_j}(s, a, r, s') \leftarrow r + \gamma\theta(s', a^*, \tau'_j, \xi^-) - \theta(s, a, \tau_i, \xi)$;
10         $L(s, a, r, s') \leftarrow \frac{1}{N} \sum_{j=1}^{N} \sum_{i=1}^{N} \rho^{\kappa}_{\tau_i}(\delta^{\tau_i, \tau'_j}(s, a, r, s'))$;
11   Update ξ by using Adam, minimizing the mean of L;
12         $s \leftarrow s'$;
13   **if** *C steps taken* **then**
14             $\xi^- \leftarrow \xi$;
15   **end**
16   **end**
17 **end**

Algorithm 2.2: The algorithm IQN.

Formally, this is done by choosing a monotonous, bijective $\beta : [0, 1] \to [0, 1]$. We then change our definition of the q-function to

$$q_\beta(s, a) = \mathbb{E}_{\tau \sim U([0,1])}[Z_{\beta(\tau)}(s, a)] \approx \frac{1}{K} \sum_{k=1}^{K} Z_{\beta(\tau_k)}(s, a),$$

where $\tau_k \sim U([0, 1]), k = 1, \ldots, K$. A convex β would make the agent act risk-seeking, whereas a concave β would make it risk-averse.

CHAPTER 3

Soft Actor-Critic

Algorithms based on Q-Learning have a quite simple approach to exploration, namely a uniform one. Imagine you want to use Reinforcement Learning to train a robot to complete some kind of obstacle course. To succeed, it has to overcome the obstacles in succession, i.e. it has to be able to overcome the first obstacle before it even sees the second one. Now, the first one may be a bit tricky, requiring the robot to follow a precise sequence of actions. This requires exploration to be very low to succeed, otherwise, the robot will still act too randomly. But because of the uniform approach to exploration, the robot would not be able to explore enough at the second obstacle if it tries to succeed at the first one.

This is where Actor-Critic methods, which we mentioned in Section 1.3, come into play: They have a more individual approach to exploration, exploring more in states where they do not have a lot of experience, and exploiting more in states where they have a lot of knowledge about the environment. But when implementing an Actor-Critic algorithm, another weakness becomes apparent: Sometimes they just think to know a lot about the environment, dropping exploration more or less completely in some states. Then, after more about the environment has been revealed to them in other states, some other actions may be better in those states, but they are unable to adapt.

In this chapter, we look at a method to control the level of exploration Actor-Critic methods exhibit, by using the entropy of the policy, a measure of the randomness of the policy, as a regulizer.

Results from the first three sections are taken from [Haarnoja et al., 2018b] if not noted otherwise.

3.1 Maximum Entropy Objective

First, we need to formally define entropy:

Definition 3.1. Let X be a random variable with PMF or PDF p. Then we call

$$\mathcal{H}(X) := -\mathbb{E}_X [\log p]$$

the *entropy* of X.

This concept comes from information theory, where entropy can be interpreted as the expected *information* to be gained from sampling from X. Uniform distributions have the highest entropy, and distributions, where all events except one have zero probability, have the lowest entropy.

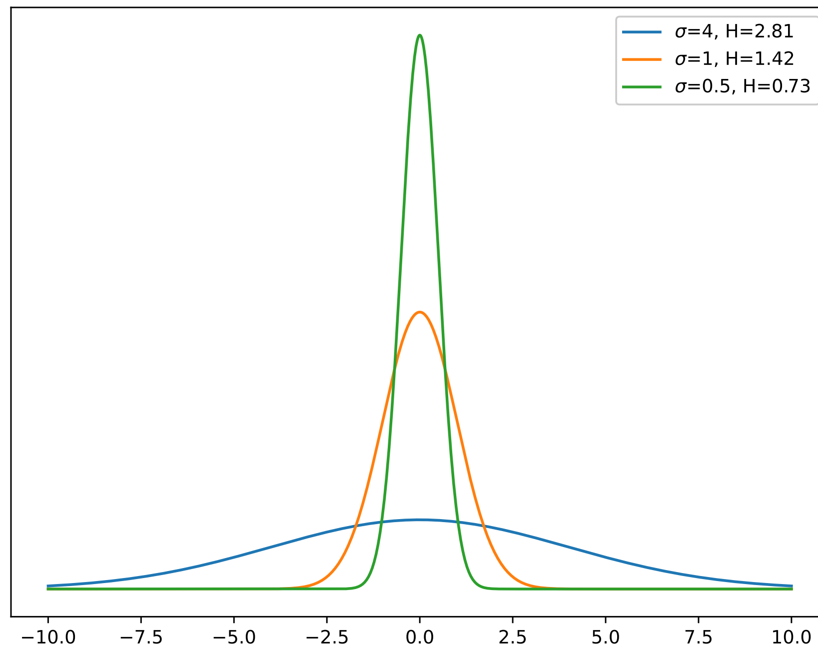

Figure 3.1: Densities of different normal distributions with mean 0 and standard deviation σ, with entropy noted in the legend.

This makes sense in the interpretation of expected information since following a uniform distribution will yield a lot of information about the underlying system, whereas following a deterministic distribution will only reveal one of its features. In the context of Reinforcement Learning, this yields a nice interpretation of $\mathcal{H}(\pi(s))$ as the expected

gained information about the environment dynamics when following the policy π in state s, i.e. $\mathcal{H}(\pi(s))$ quantifies the exploration characteristics of π.

This leads us to Maximum Entropy Reinforcement Learning, see e.g. [Ziebart, 2010] for an at-length introduction to the topic. Consider, for some $\alpha > 0$,

$$J(\pi) := \mathbb{E}_{A_t \sim \pi, S_t, R_{t+1} \sim p}\left[\sum_{t=0}^{T} R_{t+1} + \alpha \mathcal{H}(\pi(s_t))\right].$$

By our argument above, the term $\mathcal{H}(\pi(s_t))$ is a result of the *exploration* of the agent, while the reward term results from *exploitation* of the current knowledge. Thus swapping out the maximum expected return objective of Reinforcement Learning with this maximum entropy objective gives us the means to control the balance of exploration vs. exploitation by setting (the hyperparameter) α.

Because of these ideas we now define

$$\mathcal{T}^{\pi} q(s,a) := R(s,a) + \gamma \mathbb{E}_{s' \sim p}\left[v(s')\right], \tag{3.1}$$

$$v(s) := \mathbb{E}_{a \sim \pi}\left[q(s,a) - \alpha \log \pi(s,a)\right], \tag{3.2}$$

where we identify $\pi(s,a)$ with the PMF/PDF of $\pi(s)$ at action a.

From this we want to define a policy, so let $\bar{\Pi}$ be a set of policies, i.e. policies defined by normal distributions in each state. As we want our policy to explore according to the current q-function, giving higher probabilities to actions with higher q-values, we want to get close to a distribution with density $\exp(q^{\pi}(s))$. We, therefore, define the policy improvement operator $\mathcal{T}$ as a projection into $\bar{\Pi}$,

$$\mathcal{T}\pi(s) := \arg\min_{\pi' \in \bar{\Pi}} D_{\mathrm{KL}}\left(\pi'(s_t, \cdot) \,\middle\|\, \frac{\exp(\alpha^{-1} q^{\pi}(s_t, \cdot))}{Z^{\pi}(s_t)}\right), \tag{3.3}$$

where $Z^{\pi}(s)$ is a normalizing factor, chosen in such a way that $\frac{\exp(\alpha^{-1} q^{\pi}(s_t, \cdot))}{Z^{\pi}(s_t)}$ describes a distribution. This $Z^{\pi}(s)$ is generally intractable, but as we will see in Section 3.3, it is just a constant in the loss resulting from this projection and therefore has gradient zero, meaning we do not have to explore it further.

3.2 Theoretical Analysis

First, we show that the augmented objective does not change the convergence of policy evaluation:

Lemma 3.2. *Consider a mapping $q^0 : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ with $|\mathcal{A}| < \infty$ and define $q^{k+1} = \mathcal{T}^{\pi} q^k$. Then the sequence q^k will converge to the soft q-value of π as $k \to \infty$.*

Proof. As π is fixed in this setting, define

$$\bar{R}(s,a) := R(s,a) - \alpha \mathbb{E}_{s' \sim p}[\log \pi(s', \cdot)].$$

As $|\mathcal{A}| < \infty$, this augmented reward is still bounded. This makes it possible to rewrite

$$\mathcal{T}^\pi q(s,a) = \bar{R}(s,a) + \gamma \mathbb{E}\left[q(s',a)\right],$$

meaning the result then follows from the corresponding policy evaluation result from classical Reinforcement Learning, see [Sutton and Barto, 2018] Section 4.1. □

Lemma 3.3. *Let $\pi_{old} \in \bar{\Pi}$ and let*

$$\pi_{new} = \mathcal{T}\pi_{old}$$

Then

$$q^{\pi_{new}}(s_t,a_t) \geq q^{\pi_{old}}(s_t,a_t)$$

for all $(s_t,a_t) \in \mathcal{S} \times \mathcal{A}$ with $|\mathcal{A}| < \infty$.

Proof. By definition of $\mathcal{T}$,

$$\begin{aligned}\pi_{\text{new}}(s_t,\cdot) &= \arg\min_{\pi' \in \bar{\Pi}} D_{\text{KL}}\left(\pi'(s_t,\cdot) \,\middle\|\, \frac{\exp(\alpha^{-1} q^\pi(s_t,\cdot))}{Z^\pi(s_t)}\right) \\ &= \arg\min_{\pi' \in \bar{\Pi}} \mathbb{E}_{a_t \sim \pi'}\left[\log \pi'(s_t,a_t) - \alpha^{-1} q^{\pi_{\text{old}}}(s_t,a_t) + \log Z^{\pi_{\text{old}}}(s_t)\right].\end{aligned}$$

Since $\pi_{\text{old}} \in \bar{\Pi}$, it participates in the minimum, therefore

$$\begin{aligned}&\mathbb{E}_{a_t \sim \pi_{\text{new}}}\left[\log \pi_{\text{new}}(s_t,a_t) - \alpha^{-1} q^{\pi_{\text{old}}}(s_t,a_t) + \log Z^{\pi_{\text{old}}}(s_t)\right] \\ \leq\ &\mathbb{E}_{a_t \sim \pi_{\text{old}}}\left[\log \pi_{\text{old}}(s_t,a_t) - \alpha^{-1} q^{\pi_{\text{old}}}(s_t,a_t) + \log Z^{\pi_{\text{old}}}(s_t)\right].\end{aligned}$$

As $Z^{\pi_{\text{old}}}$ does not depend on a_t, its expected value is the same on both sides, therefore

$$\mathbb{E}_{a_t \sim \pi_{\text{new}}}\left[\log \pi_{\text{new}}(s_t,a_t) - \alpha^{-1} q^{\pi_{\text{old}}}(s_t,a_t)\right] \leq \mathbb{E}_{a_t \sim \pi_{\text{old}}}\left[\log \pi_{\text{old}}(s_t,a_t) - \alpha^{-1} q^{\pi_{\text{old}}}(s_t,a_t)\right]$$

and, by definition of v and after multiplying by α,

$$\mathbb{E}_{a_t \sim \pi_{\text{new}}}\left[q^{\pi_{\text{old}}}(s_t,a_t) - \alpha \log \pi_{\text{new}}(s_t,a_t)\right] \geq v^{\pi_{\text{old}}}(s_t).$$

This makes it possible to expand from the definition of q:

$$\begin{aligned}q^{\pi_{\text{old}}}(s_t,a_t) &= R(s_t,a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p}\left[v^{\pi_{\text{old}}}(s_{t+1})\right] \\ &\leq R(s_t,a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p}\left[\mathbb{E}_{a_{t+1} t \sim \pi_{\text{new}}}\left[q^{\pi_{\text{old}}}(s_{t+1},a_{t+1}) - \alpha \log \pi_{\text{new}}(s_{t+1},a_{t+1})\right]\right],\end{aligned}$$

and from here we can expand $q^{\pi_{\text{old}}}(s_{t+1},a_{t+1})$ again. This series converges, by Lemma 3.2, to $q^{\pi_{\text{new}}}(s_t,a_t)$, proving the statement.

Note that this was possible because the projection regarding the Kullback-Leibler divergence equals the soft policy evaluation term. □

This yields:

Theorem 3.4. *Assume $|\mathcal{A}| < \infty$. Then repeated application of Lemma 3.2 and Lemma 3.3 starting from any $\pi \in \bar{\Pi}$ converges to a policy optimal in $\bar{\Pi}$, i.e.*

$$q_{\pi^*}(s,a) \geq q_\pi(s,a)$$

for all $(s,a) \in \mathcal{S} \times \mathcal{A}$ and $\pi \in \bar{\Pi}$.

Proof. By Lemma 3.3, the sequence of policies generated by this repeated application is monotonous. Since the q-function is bounded (see proof of Lemma 3.2), this sequence converges to some π^*. At convergence, the minimum of the Kullback-Leibler divergence has to be attained, so we can argue similarly to the previous lemma that

$$q^{\pi^*}(s,a) \geq q^\pi(s,a)$$

for all $\pi \in \bar{\Pi}$, so π^* is optimal in $\bar{\Pi}$. □

3.3 Soft Actor-Critic, a Practical Approximation to Soft Policy Iteration

To derive a practical approximation to soft policy iteration, we will focus on continuous action spaces, we restrict ourselves even further to $\mathcal{A} = \mathbb{R}^n$. We, therefore, can use a neural net

$$\pi\colon\ \mathcal{S} \times \mathbb{R}^d \to \mathbb{R} \times \mathbb{R}^+; \quad (s,\phi) \mapsto (\mu,\sigma)$$

and then sample actions from $\mathcal{N}(\pi(s,\phi))$, i.e. a normal distribution with mean μ and standard deviation σ, compare the end of Section 1.3. We will therefore write $\pi(s,a,\phi)$ for the PDF of the normal distribution with parameters $\pi(s,\phi)$. Note that we can easily generalize this approach to box spaces $[a_1,b_1] \times \cdots \times [a_n,b_n] \subseteq \mathbb{R}^n$ by applying tanh and shifting each dimension accordingly.

Next, we can define neural nets

$$q(s,a,\theta)\colon\ \mathcal{S} \times \mathcal{A} \times \mathbb{R}^{\tilde{d}} \to \mathbb{R};$$

and

$$v(s,\psi)\colon\ \mathcal{S} \times \mathbb{R}^{\hat{d}} \to \mathbb{R}$$

as the approximations to the q- and value function. Given an experience replay buffer element (s,a,r,s'), definitions (3.1) and (3.2) lead to losses

$$L_q(\theta) = \frac{1}{2}\left(r + \gamma v(s',\psi) - q(s,a,\theta)\right)^2$$

and

$$L_v(\psi) = \frac{1}{2}\left(\mathbb{E}_{a'\sim\pi}\Big[q(s,a',\theta) - \alpha \log \pi(s,a',\phi)\Big] - v(s,\psi)\right)^2$$

respectively.

To derive the loss for π, we will use (3.3). Here, the problem arises that we cannot directly differentiate the output of $\mathcal{N}(\pi(s, \phi))$, as we have to sample from the normal distribution, which is not a differentiable operation directly. To still be able to obtain gradients, we use what is called the *reparametrization trick*: Instead of sampling a from $\mathcal{N}(\pi(s, \phi))$, we sample ε from $\mathcal{N}(0, 1)$ and then calculate

$$a = \sigma\varepsilon + \mu =: f(\varepsilon, \mu, \sigma),$$

making a distributed according to $\mathcal{N}(\mu, \sigma)$, therefore allowing differentiation of a with respect to μ and σ, meaning we can also obtain gradients for neural net optimization of $\pi(s, \phi)$. Using this trick and ignoring the factor $Z^\pi(s_t)$ like mentioned before, the loss becomes

$$L_\pi(\phi) = \mathbb{E}_{\varepsilon\sim\mathcal{N}(0,1)} \left[\alpha \log \pi(s, f(\varepsilon, \pi(s, \phi)), \phi) - q(s, f(\varepsilon, \pi(s, \phi), \theta))\right].$$

Before giving pseudocode, we will explore one more aspect of Soft Actor-Critic, tuning the entropy temperature α.

3.4 Automatic Entropy Temperature Tuning

In practice, tuning α is quite hard. Usually, we want our policy to exhibit a certain level of exploration throughout training. But leaving α constant will change this level of exploration over time, since as rewards from the environment get larger, the entropy component of the soft return dwindles in comparison. This leads to exploration getting smaller over time. While this is, in principle, not undesirable (as we have noted previously, too much exploration will lead to poor performance and possibly never experiencing some states), α does not give us enough control over the reduction in exploration. It would be much better if we could choose a desired mean entropy $\bar{\mathcal{H}}$ and then tune α in such a way that $\mathcal{H}_\pi := \mathbb{E}_s \left[\mathcal{H}(\pi(s))\right]$ is close to $\bar{\mathcal{H}}$. This idea was first presented by [Haarnoja et al., 2019], where they use constrained optimization to obtain the following results:

Theorem 3.5. *Let $\pi^*(s, a, \alpha)$ denote the optimal policy given entropy temperature α. Then the optimal value of α, retaining $\mathcal{H}_\pi \geq \bar{\mathcal{H}}$, is given by*

$$\alpha^* = \arg\min_\alpha \mathbb{E}_{a\sim\pi^*} \left[-\alpha \log \pi^*(s, a, \alpha) - \alpha\bar{\mathcal{H}}\right].$$

Note that $\mathcal{H}_\pi \geq \bar{\mathcal{H}}$ is enough to have $\mathcal{H}_\pi$ close to $\bar{\mathcal{H}}$ in practice, as higher values of α typically lead to worse performance in the Reinforcement Learning problem itself.

Furthermore, this optimal value of α is infeasible to calculate by gradient descent in practice, as the gradient of π^* with respect to α is hard to compute. We will, therefore, use the surrogate objective

$$L(\alpha) = \mathbb{E}_{a\sim\pi^*} \left[-\alpha \log \pi(s, a) - \alpha\bar{\mathcal{H}}\right],$$

i.e. we drop the dependence of π on α completely. Note that the gradient is given by

$$\nabla L(\alpha) = \mathbb{E}_{a\sim\pi^*}\left[-\log\pi(s,a)\right] - \bar{\mathcal{H}} = \mathcal{H}_\pi - \bar{\mathcal{H}}.$$

Thus, this optimization enlarges α when the current mean entropy $\mathcal{H}_\pi$ of the policy is smaller than the desired one, and reduces it when the mean entropy is too big.

Pseudocode of the algorithm combining all the concepts of the last few sections is given in Algorithm 3.1. Note that we, again, use the concept of inertia for updating our neural nets, this time in v, since v is used for the Bellman update term in L_q. Also, we use the concept of Polyak updates, i.e. we update the parameters exponentially every C steps, so for $\tau \in (0,1)$, which is called the Polyak coefficient,

$$\psi^- = \tau\psi + (1-\tau)\psi^-.$$

Furthermore, to reduce overestimation bias, two q-functions can be approximated at the same time, and the q-value used for the updates is taken as the minimum of the two approximations.

3.5 Discrete Variant

In the last two sections, we described the algorithm [Haarnoja et al., 2018b] derived from Soft Policy Iteration, which only works on continuous action spaces, as it uses normal distributions. We now want to look at a version that works on discrete, finite action spaces by using the softmax policy we described at the end of Section 1.3. We will be following [Christodoulou, 2019].

So let $\mathcal{A} = \{a_1, \dots, a_n\}$, and $h(s,a,\phi)\colon \mathcal{S}\times\mathcal{A}\times\mathbb{R}^d \to \mathbb{R}$ be a neural net parametrized by $\phi \in \mathbb{R}^d$. Define

$$\pi\colon\ \mathcal{S}\times\mathbb{R}^d \to \mathbb{R}^{|\mathcal{A}|};\quad (s,\phi)\mapsto \left(\frac{e^{h(s,a_i,\phi)}}{\sum_{j=1}^n e^{h(s,a_j,\phi)}}\right)_{i=1,\dots,|\mathcal{A}|},$$

so $\pi(s,\phi)$ describes the PMF of the action distribution in state s, with the i-th entry being the probability of taking action a_i.

Now several things are a little easier than in the continuous action case: We can directly calculate the entropy of the action distribution in each state,

$$\mathcal{H}(\pi(s,\phi)) = -\pi(s,\phi)\cdot\log\pi(s,\phi),$$

making sampling of the entropy obsolete.

Furthermore, instead of using $q(s,a,\theta)\colon \mathcal{S}\times\mathcal{A}\times R^{d'} \to \mathbb{R}$, we can use $q(s,\theta)\colon \mathcal{S}\times R^{d'} \to R^{|\mathcal{A}|}$. Therefore,

$$v(s,\theta) := q(s,\theta)\cdot\pi(s,\phi) + \alpha\pi(s,\phi)\cdot\log\pi(s,\phi)$$

```
    Initialize: Minibatchsize n, inertia C, polyak coefficient τ, target entropy H̄
    Initialize: φ: parameters of the neural net π
    Initialize: θ: parameters of the neural net q
    Initialize: ψ⁻, ψ: parameters of the neural net v
    Initialize: Buffer B of fixed size
 1  foreach episode do
 2  |   Sample start state s;
 3  |   while s not terminal do
 4  |   |   Choose a according to π(s, φ);
 5  |   |   Take action a, observe reward r, next state s', store (s, a, r, s') in B;
 6  |   |   Sample minibatch (s_i, a_i, r_i, s'_i), i = 1, ..., n from B;
 7  |   |   Sample new actions a'_i from π(s_i, φ) using the reparametrization trick;
 8  |   |   L_q ← 1/n Σ_{i=1}^n 1/2 (r + γ v(s'_i, ψ⁻) − q(s_i, a_i, θ))²;
 9  |   |   L_v ← 1/n Σ_{i=1}^n 1/2 (q(s_i, a'_i, θ) − α log π(s_i, a'_i, φ) − v(s_i, ψ))²;
10  |   |   L_π ← 1/n Σ_{i=1}^n (α log π(s_i, a'_i, φ) − q(s_i, a'_i, θ));
11  |   |   L_α ← −α log π(s_i, a'_i, φ) − αH̄;
12  |   |   Update θ by using Adam, minimizing L_q;
13  |   |   Update ψ by using Adam, minimizing L_v;
14  |   |   Update φ by using Adam, minimizing L_π;
15  |   |   Update α by using Adam, minimizing L_α;
16  |   |   s ← s';
17  |   |   if C steps taken then
18  |   |   |   ψ⁻ ← τψ + (1 − τ)ψ⁻;
19  |   |   end
20  |   end
21  end
```

Algorithm 3.1: The algorithm Soft Actor Critic.

is the soft value function, so we do not need an extra neural net or approximation step for the calculation of the L_q.

Of course no reparametrization trick is needed, and similar to the above simplifications, the expected value in the policy loss can be simplified to

$$L_\pi = \pi(s, \phi) \cdot (\alpha \log \pi(s, \phi) - q(s, \theta)) \,.$$

The pseudocode of this algorithm, which we will call Discrete Soft Actor-Critic, is given in Algorithm 3.2. Note that overestimation bias counteraction by approximating two q-functions simultaneously we discussed above for the continuous variant is also possible here.

Initialize: Minibatchsize n, inertia C, polyak coefficient τ, target entropy $\bar{\mathcal{H}}$
Initialize: ϕ: parameters of the neural net π
Initialize: θ^-, θ: parameters of the neural net q
Initialize: ψ^-, ψ: parameters of the neural net v
Initialize: Buffer B of fixed size

1 **foreach** *episode* **do**
2 Sample start state s;
3 **while** *s not terminal* **do**
4 Choose a according to $\pi(s, \phi)$;
5 Take action a, observe reward r, next state s', store (s, a, r, s') in B;
6 Sample minibatch (s_i, a_i, r_i, s_i'), $i = 1, \ldots, n$ from B;
7 $L_q \leftarrow \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} \left(r + \gamma \pi(s_i', \phi) \cdot q(s_i', \theta^-) - q(s_i, a_i, \theta)\right)^2$;
8 $L_\pi \leftarrow \frac{1}{n} \sum_{i=1}^{n} \pi(s_i, \phi) \cdot (\alpha \log \pi(s_i, \phi) - q(s_i, \theta^-))$;
9 $L_\alpha \leftarrow -\alpha \pi(s_i, \phi) \cdot \log \pi(s_i, \phi) - \alpha \bar{\mathcal{H}}$;
10 Update θ by using Adam, minimizing L_q;
11 Update ψ by using Adam, minimizing L_v;
12 Update ϕ by using Adam, minimizing L_π;
13 Update α by using Adam, minimizing L_α;
14 $s \leftarrow s'$;
15 **if** *C steps taken* **then**
16 $\theta^- \leftarrow \tau\theta + (1 - \tau)\theta^-$;
17 **end**
18 **end**
19 **end**

Algorithm 3.2: The algorithm Discrete SAC.

3.6 Distributional Extension

The distributional q-value approximation we discussed in Chapter 2 can be seen as a separate component of the actual algorithms given, so one might wonder if it is possible to extend Soft Actor-Critic to use this component. In this section, we will give two variations of Soft Actor-Critic using quantile regression and implicit quantile networks. The distributional extension described here was done as part of the thesis, but other work exists on the topic, e.g. [Duan et al., 2022], which I was not aware of while deriving this extension.

Unifying notation from QRDQN and IQN, let

$$q(s, a, \tau)\colon\ \mathcal{S} \times \mathcal{A} \times [0, 1] \to \mathbb{R},$$

which corresponds to $Z_\tau(s, a)$ from Chapter 2. Furthermore, let

$$v(s, \tau)\colon\ \mathcal{S} \times [0, 1] \to \mathbb{R}$$

be a representation of the quantile function of the distribution underlying the value function, similar to $Z_\tau(s,a)$ for the q-function.

Using this notation and the quantile Huber loss ρ^κ_τ, the distributional Soft Actor Critic losses for q and v become

$$L_q(s,a,r,s') = \frac{1}{N'} \sum_{i=1}^{N} \sum_{j=1}^{N'} \rho^\kappa_{\tau_i} \Big(r + \gamma v(s', \tau'_j) - q(s,a,\tau_i) \Big)$$

and

$$L_v(s,a,r,s') = \frac{1}{N'} \sum_{i=1}^{N} \sum_{j=1}^{N'} \rho^\kappa_{\tau_i} \left(\mathbb{E}_{a' \sim \pi} \Big[q(s,a',\tau'_j) - \alpha \log \pi(s,a',\phi) \Big] - v(s,\psi,\tau_i) \right).$$

The loss of π stays the same, using $q(s,a) = \mathbb{E}_\tau\left[q(s,a,\tau)\right]$.

We, therefore, get a quantile regression variant of Soft Actor-Critic using $N = N'$ and equidistant quantile midpoints $\tau = \tau'$, and an implicit quantile network variant by using $\tau_i \sim U([0,1]), i = 1, \ldots, N$ and $\tau'_j \sim U([0,1]), j = 1, \ldots, N'$. Pseudocodes for both variants, which we will call QRSAC and IQNSAC, are given in Algorithms 3.3 and 3.4 respectively.

Unfortunately, the trick of using two separate approximations of the q-function to reduce overestimation bias does not work here anymore, as the quantiles of the minimum over two functions are not the minimum of their quantiles.

In the discrete case, we have to omit the trick of representing v in terms of π and q for the loss of q, since similarly to the minimum, the quantiles of a linear combination of functions are not the linear combination of those quantiles. We can still apply that trick for L_π however. This gives two algorithms which we will call Discrete Quantile Regression Soft Actor-Critic (Discrete QRSAC) and Discrete Implicit Quantile Network Soft Actor-Critic (Discrete IQNSAC). Since the pseudocodes are a mixture of Algorithms 3.3 and 3.4 with Algorithm 3.2 (lines 10–11 and 11–12 of Algorithms 3.3 and 3.4 respectively are replaced by lines 8–9 of Algorithm 3.2 and network architectures are adjusted to fit the discrete setting), they will be omitted here.

Initialize: Minibatchsize n, inertia C, polyak coefficient τ, target entropy $\bar{\mathcal{H}}$
Initialize: Quantile midpoints $\hat{\tau}_i$, $i = 1, \ldots, N$
Initialize: ϕ: parameters of the neural net π
Initialize: θ: parameters of the neural net for $q(s, a, \tau)$, τ a quantile midpoint $\hat{\tau}_i$
Initialize: ψ^-, ψ: parameters of the neural net v
Initialize: Buffer B of fixed size

1 **foreach** *episode* **do**
2 Sample start state s;
3 **while** s *not terminal* **do**
4 Choose a according to $\pi(s, \phi)$;
5 Take action a, observe reward r, next state s', store (s, a, r, s') in B;
6 Sample minibatch (s_i, a_i, r_i, s'_i), $i = 1, \ldots, n$ from B;
7 Sample new actions a'_i from $\pi(s_i, \phi)$ using the reparametrization trick;
8 $L_q(s_i, a_i, r_i, s'_i) \leftarrow \frac{1}{N} \sum_{k=1}^{N} \sum_{j=1}^{N} \rho^{\kappa}_{\hat{\tau}_k} \left(r_i + \gamma v(s'_i, \hat{\tau}_j, \psi^-) - q(s_i, a_i, \hat{\tau}_k, \theta)\right)$;
9 $L_v(s_i, a_i, r_i, s'_i) \leftarrow$
$\frac{1}{N'} \sum_{k=1}^{N} \sum_{j=1}^{N'} \rho^{\kappa}_{\hat{\tau}_k} \left(q(s_i, a'_i, \hat{\tau}_j, \theta) - \alpha \log \pi(s_i, a'_i, \phi) - v(s_i, \hat{\tau}_k, \psi)\right)$;
10 $L_\pi(s_i, a_i, r_i, s'_i) \leftarrow \alpha \log \pi(s_i, a'_i, \phi) - \frac{1}{N} \sum_{k=1}^{N} q(s_i, a'_i, \hat{\tau}_k, \theta)$;
11 $L_\alpha(s_i, a'_i) \leftarrow -\alpha \log \pi(s_i, a'_i, \phi) - \alpha \bar{\mathcal{H}}$;
12 Update θ by using Adam, minimizing the mean over all L_q;
13 Update ψ by using Adam, minimizing the mean over all L_v;
14 Update ϕ by using Adam, minimizing the mean over all L_π;
15 Update α by using Adam, minimizing the mean over all L_α;
16 $s \leftarrow s'$;
17 **if** C *steps taken* **then**
18 $\psi^- \leftarrow \tau\psi + (1 - \tau)\psi^-$;
19 **end**
20 **end**
21 **end**

Algorithm 3.3: The quantile regression variant of Soft Actor-Critic, QRSAC, in practice $q(s, a, \hat{\tau}_i)$ would be implemented as $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^N$.

Initialize: Minibatchsize n, inertia C, polyak coefficient τ, target entropy $\bar{\mathcal{H}}$
Initialize: ϕ: parameters of the neural net π
Initialize: θ: parameters of the neural net for $q(s, a, \tau)$
Initialize: ψ^-, ψ: parameters of the neural net v
Initialize: Buffer B of fixed size

1 **foreach** *episode* **do**
2 Sample start state s;
3 **while** *s not terminal* **do**
4 Choose a according to $\pi(s, \phi)$;
5 Take action a, observe reward r, next state s', store (s, a, r, s') in B;
6 Sample minibatch (s_i, a_i, r_i, s_i'), $i = 1, \ldots, n$ from B;
7 Sample new actions a_i' from $\pi(s_i, \phi)$ using the reparametrization trick;
8 Sample $\tau_k, \tau_j' \sim U([0,1])$, $k = 1, \ldots, N$, $j = 1, \ldots, N'$;
9 $L_q(s_i, a_i, r_i, s_i') \leftarrow \frac{1}{N} \sum_{k=1}^{N} \sum_{j=1}^{N} \rho_{\tau_k}^{\kappa} \left(r_i + \gamma v(s_i', \tau_j, \psi^-) - q(s_i, a_i, \tau_k, \theta)\right)$;
10 $L_v(s_i, a_i, r_i, s_i') \leftarrow$
$\frac{1}{N'} \sum_{k=1}^{N} \sum_{j=1}^{N'} \rho_{\tau_k}^{\kappa} \left(q(s_i, a_i', \tau_j, \theta) - \alpha \log \pi(s_i, a_i', \phi) - v(s_i, \tau_k, \psi)\right)$;
11 $L_\pi(s_i, a_i, r_i, s_i') \leftarrow \alpha \log \pi(s_i, a_i', \phi) - \frac{1}{N} \sum_{k=1}^{N} q(s_i, a_i', \hat{\tau}_k, \theta)$;
12 $L_\alpha(s_i, a_i') \leftarrow -\alpha \log \pi(s_i, a_i', \phi) - \alpha \bar{\mathcal{H}}$;
13 Update θ by using Adam, minimizing the mean over all L_q;
14 Update ψ by using Adam, minimizing the mean over all L_v;
15 Update ϕ by using Adam, minimizing the mean over all L_π;
16 Update α by using Adam, minimizing the mean over all L_α;
17 $s \leftarrow s'$;
18 **if** *C steps taken* **then**
19 $\psi^- \leftarrow \tau\psi + (1 - \tau)\psi^-$;
20 **end**
21 **end**
22 **end**

Algorithm 3.4: The implicit quantile network variant of Soft Actor-Critic, IQNSAC.

CHAPTER 4

Experimental Results

The main part of this thesis was implementing and evaluating the algorithms described in Chapters 2 and 3 and DQN. This was done in `Python 3.8` using `PyTorch`, [Paszke et al., 2019] as the Deep Learning library.

Before discussing experimental results, some implementation details important for interpreting these results will be provided and an overview of the environments will be given.

4.1 Implementation Details

Implementation was done as single-file implementations to ensure independence and ease of adaptability of the algorithms. Only the buffer architecture is shared between files.

All of them are OpenAI Gym compatible, [Brockman et al., 2016], which is distributed in Python as `gym` by `pip`.

Some specific important details:

1. For better performance, the neural net approximating the distributional q-function in IQN was split into three parts: one part q_s that takes the state-action pairs as input, another part q_q that takes the quantiles as input, and a net q_c combining the outputs of these two nets by a Hadamard product to give the final approximation of the distributional q-function.

   Therefore, the complete net approximating the quantile function is given by

   $$q(s, a, \tau) = q_c(q_s(s, a) \odot q_q(\tau)).$$

   The Hadamard product is used to force interaction between the outputs of q_s and q_q, so in theory, a one-layered q_c is enough. In practice, experiments showed that

multiple layers might still provide some benefit in q_c, but the Hadamard product is beneficial nonetheless.

2. Experiments showed that a Polyak update is in some cases beneficial also for updates to the policy parameters we usually denoted with ϕ. Therefore, an additional Polyak parameter τ_π is provided in all Soft Actor-Critic variants.

3. In [Haarnoja et al., 2018a], another variant of Soft Actor-Critic is introduced, where v is not approximated by a neural net, but

$$v(s) = \mathbb{E}_{a\sim\pi}\left[q(s, a)\right]$$

is used to approximate v by sampling an action a from π and then setting $v(s) = q(s, a)$ in that update step. This can also be done similarly in the distributional case.

The variant where v is approximated separately is more interesting when discussing the theory, which is why it was the one we covered in Chapter 3, but the variant approximating v by using q performs a little better in practice, so this variant is used for the presented results. Note that when doing this, you have to perform the Polyak update on θ.

4. Wherever possible, `stable-baselines3`, [Raffin et al., 2021], was used to verify the performance of the implementations.

4.2 Overview of the Environments

4.2.1 CartPole-v1

In `CartPole-v1`, the agent controls a cart that carries an upright pole. The goal is to balance the pole on the cart by moving left and right. The observation space is a 4-dimensional box space, the action space is discrete with two actions, “move left” and “move right”. The reward is always 1 and the episode ends if the pole tilts too far. After a maximum of 500 steps, the episode terminates either way, making the maximum undiscounted return 500.

This environment is part of the classic control environments available by default in the `gym` package. We will use it to show some basic stability properties of the algorithms, as it is relatively computationally inexpensive.

4.2.2 Autonomous Driving Environments

The main interest of this section lies in autonomous driving. The gym environments described here were developed by Helmut Horvath and me in the CARLA simulator, version 0.9.13 [Dosovitskiy et al., 2017].

We will use the following scenarios:

- `carla-lanekeeping-relative-v0`: An environment featuring a 6-dimensional state space containing, among other things, the offset from the center of the lane, the angle to the lane, and the lane's curvature. The action space is discrete with 9 actions controlling the steering of the car, while the throttle is set randomly at the start of the episode. The car has to learn to drive along a highway, the map used in this scenario is depicted in Figure 4.1. The reward, therefore, is high for small offsets from the center of the lane.

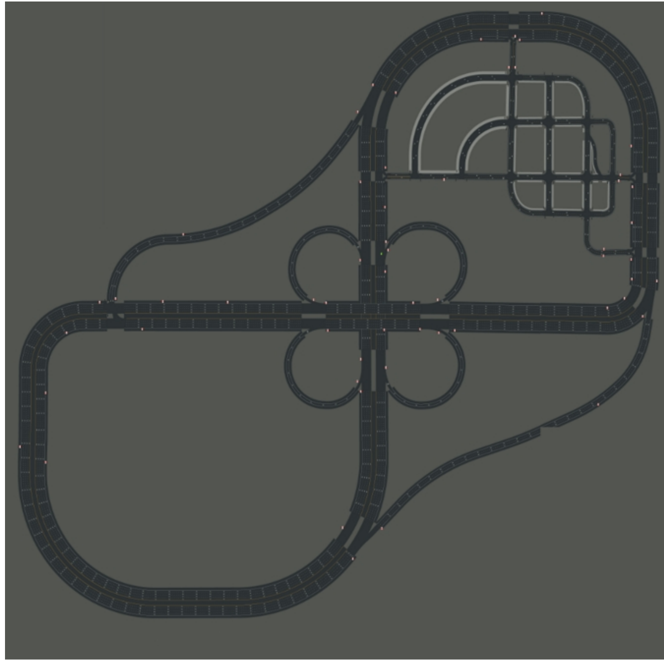

Figure 4.1: A schematic of `Town04` in the CARLA simulator, used for the lane keeping and the combine-adaptive scenario. In both, the car has to drive on the infinite-loop highway. [CARLADocs, 2022]

  The actions work in a relative way, which means that the agent controls how much to change the current steering. By doing this, the car can fine-tune the individual steering angles needed for different curves much more and can reach much higher performance than if the actions worked in an absolute way.

  The car is placed on the same part of the road at the start of every episode, so the agent has to first learn how to drive a right turn before it even experiences any left turn.

- `carla-cruisecontrol-v0`: Another discrete environment. The car is placed at the start of one of the long, straight roads in the map depicted in Figure 4.2. The agent can control the throttle and brake, discretized as 7 actions, and using throttle and brake at the same time is not possible. At the start of each episode, a random target velocity is chosen. This target velocity changes every timestep with

a probability of 1/150, or about 0.7%. After going over a specified line near the end of the straight part of the road, the target velocity is set to zero, so the car has to perform an emergency brake.

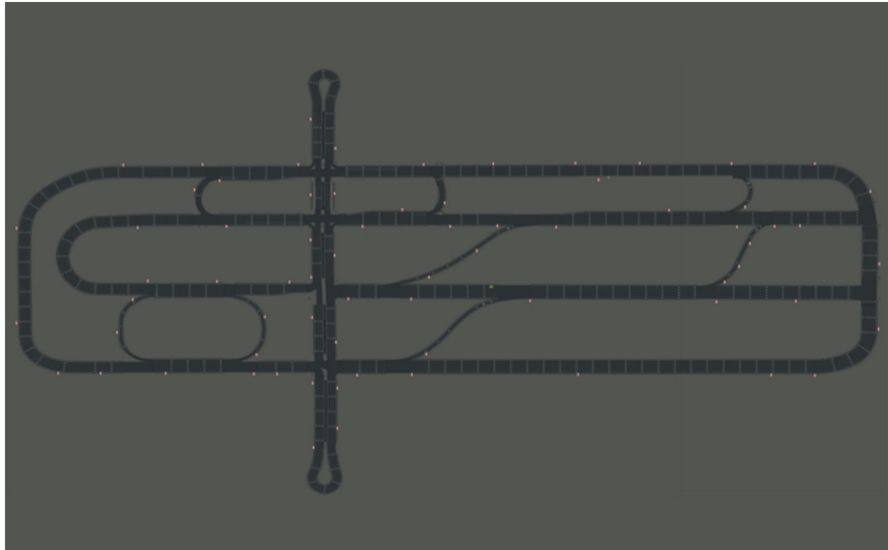

Figure 4.2: A schematic of `Town06` in the CARLA simulator, used for the cruise control scenario. [CARLADocs, 2022]

Reward is given for being closer to target velocity, but being over this target gives a little less reward than being below the target by the same amount, so the car learns to respect speed limits.

- `carla-lanekeeping-v0`: The continuous counterpart of `carla-lanekeeping-relative-v0` works in exactly the same way, but the agent has the full range of steering, i.e. the interval $[-1, 1]$, available.

- `carla-combined-adaptive-v0`: A continuous combination of cruise control and lane keeping, where the agent controls the car by setting throttle/brake actions in $[-1, 1]$ (values smaller than zero correspond to braking, values bigger than zero to throttle) and steering actions in $[-1, 1]$ at the same time, meaning the action space is $[-1, 1] \times [-1, 1]$.

  The map is the same as in the lane keeping scenarios, but in addition, another car with a different velocity is occasionally spawned in front of the agent. Reward is given as a combination of cruise control and lane keeping rewards, scaled so the agent keeps a safe distance from the other car.

4.3 Results

The plots in this section show the average (undiscounted) return per step during training. To generate them, 50 runs of each algorithm and parameter set depicted were generated, and the return for each episode was saved. Then, the return for each step in each run is generated, so for example, if a run had 20 steps in the first episode, and 30 in the second, the first episode had a return of 2 and the second episode a reward of 3, an array consisting of 20 entries of 2, then 30 entries of 3 is generated. Lastly, a moving average of 5000 steps is applied such that trends in training are more visible, and the mean performance and a 95% confidence interval are plotted. This approach to plotting the performance should give a good representation of the training process, including exploration properties.

Hyperparameters were chosen by a mixture of automatic tuning using `optuna`, [Akiba et al., 2019], and manual adaptation so the parameters used match the ones from the papers presented in the previous chapters more closely. In the interest of reproducibility, but so as not to clutter this section, all hyperparameters are included in Appendix A.

All computational results presented were obtained using the CLIP cluster (`https://clip.science`). For early testing and hyperparameter tuning, the Vienna Scientific Cluster was used (`https://vsc.ac.at/`).

4.3.1 Stability

The first things we want to take a look at are the stability properties of the DQN-based algorithms. Figure 4.3 shows performance plots in `CartPole-v1` in dependence of different values of the inertia C, which increases stability for higher values.

For $C = 1$, only the performance of DQN even increases, QRDQN and IQN diverge completely. At $C = 20$, IQN starts to learn, while QRDQN still diverges. At $C = 50$, all algorithms increase in performance over time, but QRDQN starts to diverge a bit again after the initial bump in performance, although it will probably retain better average returns than a random policy.

At $C = 100$, DQN and QRDQN both have very good performance, while IQN's performance has not really changed from $C = 50$, as it seems to hit a ceiling at about 400 average return. This is due to network capacity and not stability, and we will return to this problem later.

Note that the performance curve of DQN notably flattens from $C = 1$ to $C = 100$. This is due to C slowing down training speed, which is also how the stability properties improve, but this necessitates a balance between stability and learning speed, making C a not-so-easy-to-tune parameter.

Also, note that no algorithm really hits the maximum of 500 average reward. This is due to how exploration works in this class of algorithms. The exploration parameter

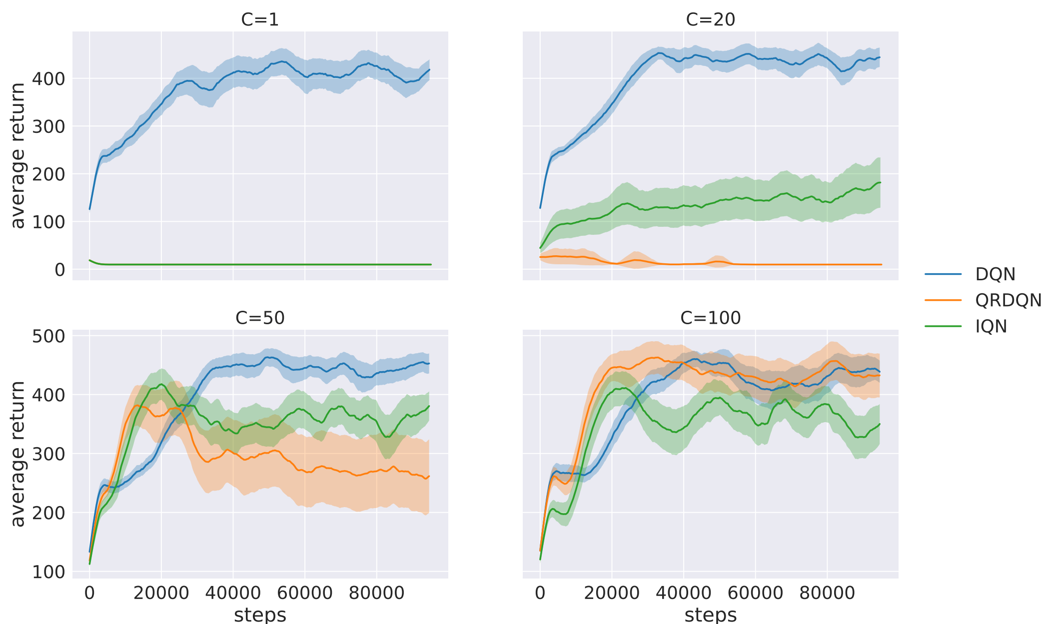

Figure 4.3: Performance plots depicting the effect of the stability parameter C on training the DQN-based algorithms.

ε was set to 5% after 5000 steps, meaning on average every 20th action is completely random. Therefore, if a couple of successive actions are wrong, the agent will not be able to recover, ending the episode early.

In contrast, Discrete SAC reaches this theoretical maximum for $C = 1$, as depicted in Figure 4.4. Note that the SAC variants employ an extra stability parameter, τ, so C of the DQN variants is not directly comparable to the C of the SAC variants.

Discrete IQNSAC is more or less stable, but not able to reach as high a performance, while Discrete QRSAC improves in performance initially, but quickly starts diverging again.

At $C = 2$ all algorithms are mostly stable, not dropping in performance, but Discrete SAC still is the only one to come close to the theoretical maximal performance. At $C = 10$, Discrete QRSAC also reaches very high performances, and at $C = 20$ all three variants reach high performance, although Discrete IQNSAC drops it again after the initial high and Discrete SAC has a very flat learning curve, meaning that this high focus on stability hurts its performance quite a bit already.

Altogether, the conclusion for both the DQN and SAC variants is, that the non-distributional versions behave the most stable, while the implicit quantile network variants initially perform second best, but do not benefit as quickly from added stability as the quantile regression variants do.

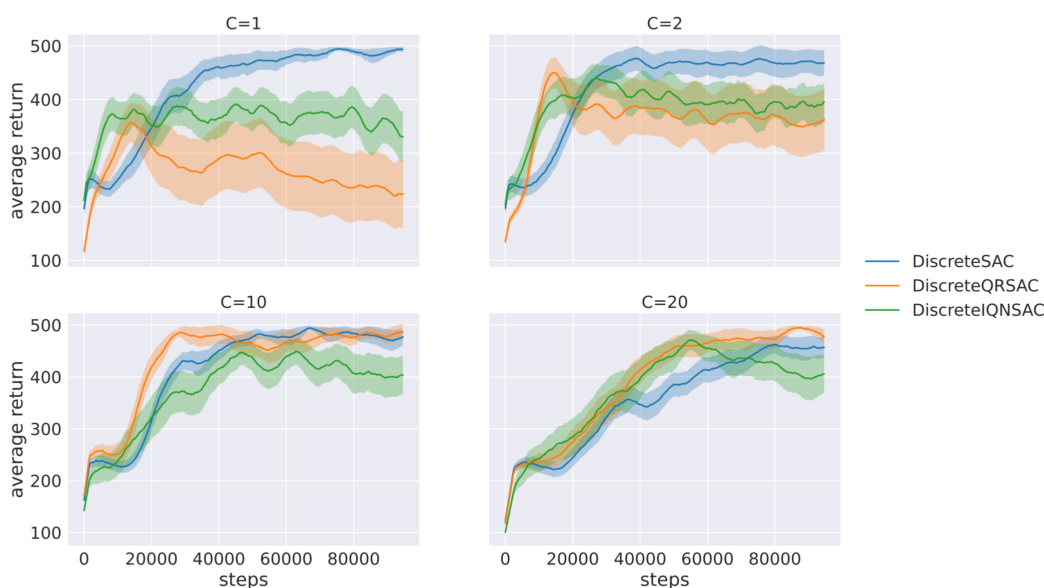

Figure 4.4: Performance plots depicting the effect of the stability parameter C on training the SAC-based algorithms.

4.3.2 Performance in Autonomous Driving

From the previous section, one would get the picture that distributional q-value approximation makes no sense since it behaves much more unstably than its non-distributional counterpart. But `CartPole-v1` is a relatively easy environment. In this section, we will look at the much more complex autonomous driving environments and will show that distributional extensions of DQN and SAC can make a lot of sense to use.

Cruise Control

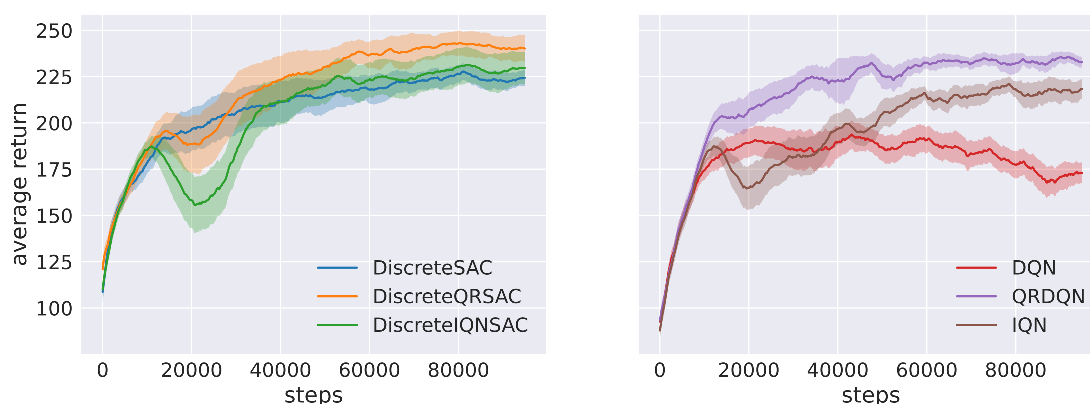

Figure 4.5: Performance in `carla-cruisecontrol-v0`.

We start with the discrete environments. In `carla-cruisecontrol-v0`, depicted in Figure 4.5, all algorithms generally reach good performance. The distributional SAC variants have a slight bump in performance at the start, most likely due to stability, but they recover quickly, therefore no increase in the stability parameters is needed. DQN has the worst performance, but both QRDQN and IQN perform well, with performances inside of the confidence intervals of the SAC variants, and therefore no actual difference is obvious between the two classes of algorithms. It is notable, however, that quantile regression performs best in both classes, closely followed by implicitly quantile networks.

Discrete Lane Keeping

A different picture is given for `carla-lanekeeping-relative-v0`, depicted in Figure 4.6. Here, the DQN-based algorithms perform notably worse than their SAC counterparts. Thinking back to the analogy of the obstacle course in the introduction of Chapter 3, this makes sense: While the first and second turns of the lane keeping scenario are not so hard even on higher velocities, the third turn is very tight and therefore has quite narrow error margins. Even a single wrong action can result in the agent not being able to recover, corresponding to the first obstacle from the analogy. The soft ε-policies used in the DQN variants will not be able to consistently overcome this turn for higher values of ε, but lower values will result in the next turns not being learnable since these are the first left turns the agent encounters, therefore needing some exploration. This results in the low returns seen on the right side of Figure 4.6, while the SAC variants are all able to satisfactorily perform in the environment.

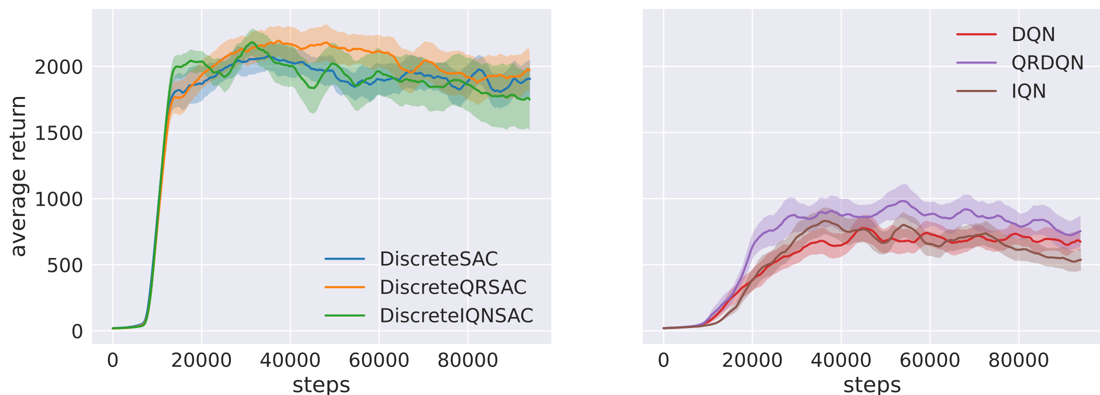

Figure 4.6: Performance in `carla-lanekeeping-relative-v0`.

A Note on Capacity

This is also a good opportunity to talk about how the network capacities impact performance and how they were chosen: As we have seen in Figure 4.4, capacity can act like a ceiling for performance. If the network is not expressive enough to capture the environment dynamics, a certain return can never be surpassed.

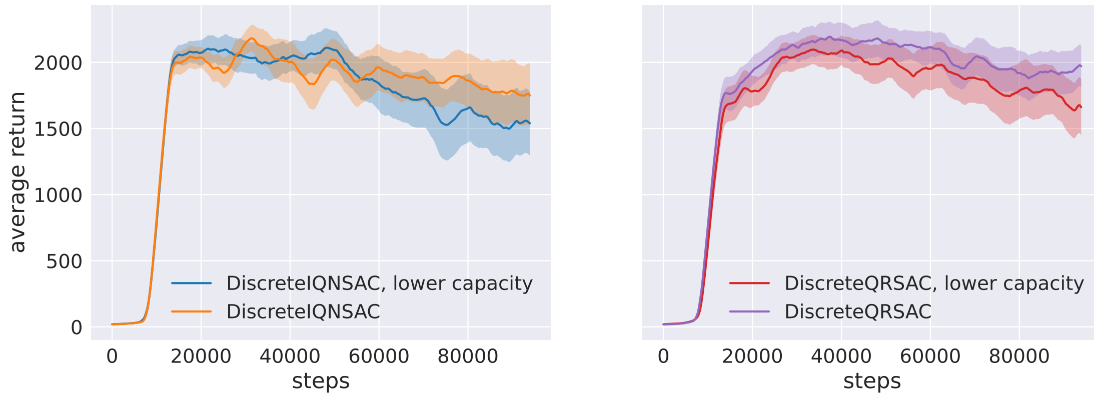

Figure 4.7: Performance in `carla-lanekeeping-relative-v0`.

To assess whether or not capacity is the reason for a bounded performance, one has to increase the network capacity and re-run the algorithm. (Note that, usually, the q-networks are the bottleneck as they have to capture the environment dynamics, the policy is seldom the culprit for low performance due to low network capacity.) Figure 4.7 shows the difference between the runs of Discrete IQNSAC and Discrete QRSAC from Figure 4.6 and the runs from the same hyperparameters, except that the q-networks were kept at lower capacity, removing two hidden layers for both. For the exact hyperparameters used see Tables A.15, A.20 for the Discrete IQNSACs, Tables A.14, A.19 for the Discrete QRSACs. We can see that the higher capacity improves performance a bit, although maybe not as much as one might expect from such a drastic increase in network parameters.

Since higher network capacities usually lead to higher training times, this is a tradeoff between runtime and performance. To ensure statistically sound results, all graphs in this thesis show confidence intervals of the return over 50 runs. Therefore, the capacity was tested using lower run counts, based on which it was chosen so runtime would be reasonable, but performance would not be notably impacted, in order to show results representative of the ideas behind the algorithms and not results constrained by low network capacities.

Also, note that the network capacity needed is highly dependent on the algorithms used. Discrete SAC only has to approximate q-values, while Discrete QRSAC has to approximate the distribution underlying those q-values at set quantile points, and Discrete IQNSAC even has to approximate the whole quantile function of those distributions. Their continuous counterparts have to do this for a continuous action space, requiring

even more capacity. This leads to even more careful considerations when choosing the algorithm for a given environment. In a deterministic environment, for example, using distributional q-value approximation does not make much sense, whereas, in a highly stochastic environment, one might benefit largely from having access to the whole quantile function describing environment interactions in much more detail.

Continuous Lane Keeping

In continuous lane keeping, the agent has the whole action space of $[-1, 1]$ for steering at its disposal. But taking actions outside the interval $[-0.1, 0.1]$ will result in steering that is too extreme, crashing the car with high probability. Therefore, the agent has to learn to ignore the actions outside of this interval early on in exploration, making SAC-based algorithms work well for this scenario.

As we can see, the added expressiveness of implicit quantile networks seems to pay off here. Although slower in learning, IQNSAC significantly overtakes SAC and QRSAC in performance. Also, training in this environment seems to benefit from distributional q-value approximation, as QRSAC and IQNSAC have about 30% more performance than SAC.

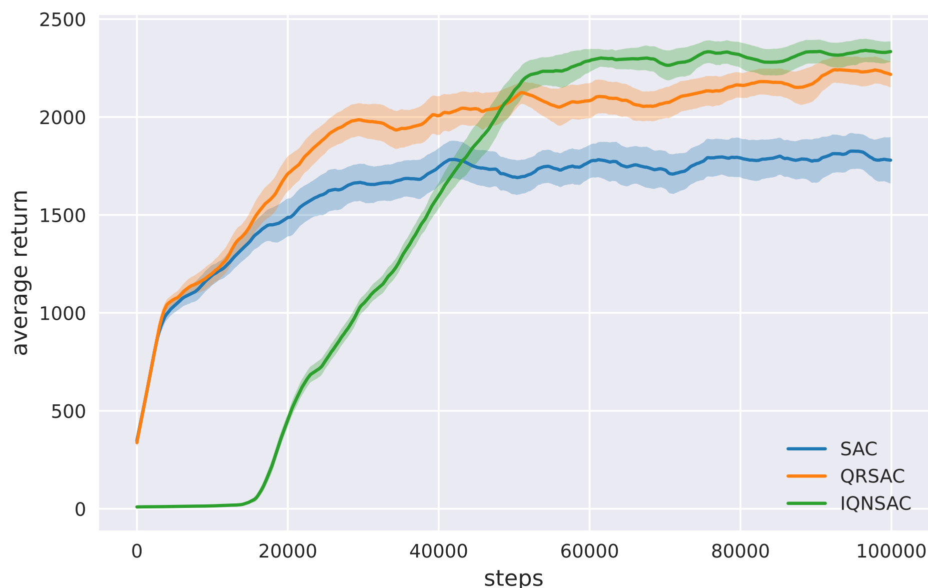

Figure 4.8: Performance in `carla-lanekeeping-v0`.

Combined Adaptive

The last environment presented is the combination of cruise control and lane keeping, where the agent has to combine all the skills needed for the previous environments and additionally keep a safe distance from a car driving in front. The learning curve is depicted in Figure 4.9. Here, QRSAC performs best from the beginning and also reaches the highest return overall. IQNSAC comes close, at about 100 000 steps the two confidence intervals are about the same, but performs a bit worse altogether, with learning also needing a longer initial phase. SAC performs comparably to QRSAC at first, but then seems to reach bounds in performance it cannot break through. Overall, distributional methods seem to perform better in this task as well, but the additional information of approximating the whole quantile function does not provide any benefit.

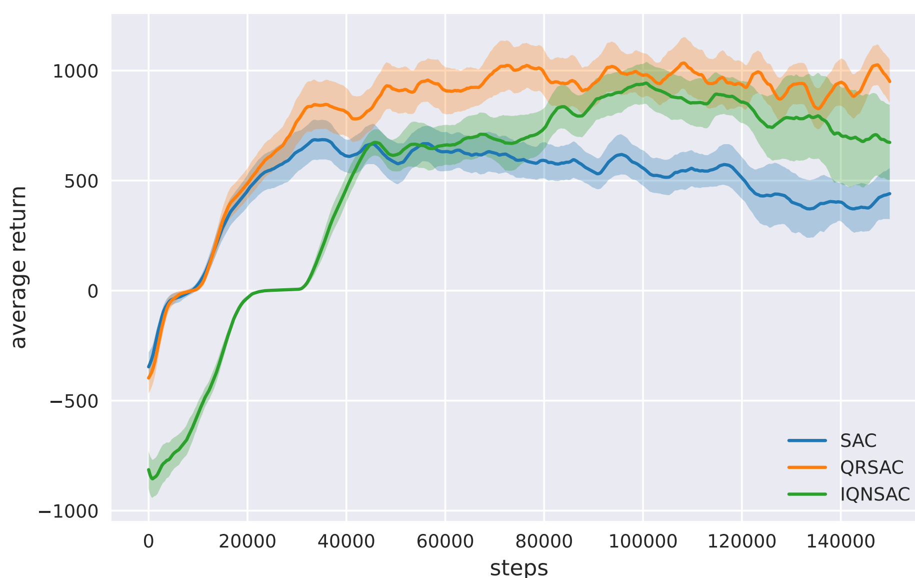

Figure 4.9: Performance in `carla-combined-adaptive-v0`.

Summary

Altogether, the exploration provided by Soft Actor-Critic outperforms the soft ε-exploration in all autonomous driving environments we studied. Additionally, distributional q-value approximation yields higher returns, justifying the combination of the two approaches.

CHAPTER 5

Conclusion

Motivated by autonomous driving tasks, we started looking at the basics of Reinforcement Learning, defining concepts and extending them to modern deep learning architectures.

In the next two chapters, we looked at the methods in the title of the thesis, introducing the theory behind approximation of the return distribution and Maximum Entropy Reinforcement Learning, giving an overview of the literature and proofing some of the results. At the end of Chapter 3, we combined the two approaches.

Chapter 4 then served as an experimental justification for this combination. After giving some details on the implementation and environments used, we showed that the resulting algorithms work and that using distributional Soft Actor-Critic methods will outperform the two methods in isolation.

But deciding whether to use quantile regression or implicit quantile network-based methods, the answer is not so clear. If highest possible performance is not so important, the recommendation is to go with quantile regression, as it is the simpler approach and also runs a little faster due to relying on simpler network architecture. Also, it often outperforms implicit quantile networks, seemingly if the environment dynamics are not so complicated that they cannot be accurately represented by discrete quantile points. If an increase in performance by a few percentage points is important, both methods have to be tested, as implicit quantile networks may slightly outperform quantile regression.

APPENDIX A

Hyperparameters

A.1 Stability – DQNs

Hyperparameter	**Value**
`buffer_size`	10 000
`eps_coolsteps`	5 000
`eps_end`	0.05
`eps_start`	1
`gamma`	0.99
`learning_starts`	257
`lr`	0.00025
`q_net`	LinearNet((seq): Sequential((0): Linear(in_features=4, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=2, bias=True)))
`sampling_size`	256
`tau`	1

Table A.1: Hyperparameters of DQN in `CartPole-v1`

Hyperparameter	Value
buffer_size	10 000
eps_coolsteps	5 000
eps_end	0.05
eps_start	1
gamma	0.99
learning_starts	257
lr	0.000 25
nr_quantiles	25
q_net	LinearNet((seq): Sequential((0): Linear(in_features=4, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=50, bias=True)))
sampling_size	256
tau	1

Table A.2: Hyperparameters of QRDQN in CartPole-v1

Hyperparameter	Value
buffer_size	10 000
eps_coolsteps	5 000
eps_end	0.05
eps_start	1
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 25
q_final_net	FinalNet((seq): Sequential((0): Linear(in_features=128, out_features=2, bias=True)))
q_quantiles_net	LinearNet((seq): Sequential((0): Linear(in_features=1, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True)))
q_states_net	LinearNet((seq): Sequential((0): Linear(in_features=4, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True)))
quantile_samples	25
sampling_size	256
tau	1

Table A.3: Hyperparameters of IQN in CartPole-v1

A.2 Stability – SACs

Hyperparameter	Value
H	0.4
alpha	0.12
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 25
lr_alpha	0.000 1
policy_net	LinearNet((seq): Sequential((0): Linear(in_features=4, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=256, bias=True) (3): ReLU() (4): Linear(in_features=256, out_features=2, bias=True)))
q1_net	LinearNet((seq): Sequential((0): Linear(in_features=4, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=2, bias=True)))
q2_net	LinearNet((seq): Sequential((0): Linear(in_features=4, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=2, bias=True)))
sampling_size	256
tau	0.05
tau_pi	0.005

Table A.4: Hyperparameters of DiscreteSAC in CartPole-v1

Hyperparameter	**Value**
`H`	0.4
`alpha`	0.12
`buffer_size`	10 000
`gamma`	0.99
`learn_alpha`	True
`learning_starts`	257
`lr`	0.000 25
`lr_alpha`	0.000 1
`nr_quantiles`	25
`policy_net`	LinearNet((seq): Sequential((0): Linear(in_features=4, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=256, bias=True) (3): ReLU() (4): Linear(in_features=256, out_features=2, bias=True)))
`q_net`	LinearNet((seq): Sequential((0): Linear(in_features=4, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=50, bias=True)))
`sampling_size`	256
`tau`	0.05
`tau_pi`	0.005

Table A.5: Hyperparameters of DiscreteQRSAC in `CartPole-v1`

Hyperparameter	Value
H	0.4
alpha	0.12
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 25
lr_alpha	0.000 1
policy_net	LinearNet((seq): Sequential((0): Linear(in_features=4, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=256, bias=True) (3): ReLU() (4): Linear(in_features=256, out_features=2, bias=True)))
q_final_net	FinalNet((seq): Sequential((0): Linear(in_features=128, out_features=2, bias=True)))
q_quantiles_net	LinearNet((seq): Sequential((0): Linear(in_features=1, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True)))
q_states_net	LinearNet((seq): Sequential((0): Linear(in_features=4, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True)))
quantile_samples	25
sampling_size	256
tau	0.05
tau_pi	0.005

Table A.6: Hyperparameters of DiscreteIQNSAC in CartPole-v1

A.3 Cruise Control

Hyperparameter	Value
C	4
H	0.5
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 1
policy_net	LinearNet((seq): Sequential((0): Linear(in_features=3, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=7, bias=True)))
q1_net	LinearNet((seq): Sequential((0): Linear(in_features=3, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True) (5): ReLU() (6): Linear(in_features=512, out_features=7, bias=True)))
q2_net	LinearNet((seq): Sequential((0): Linear(in_features=3, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True) (5): ReLU() (6): Linear(in_features=512, out_features=7, bias=True)))
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.7: Hyperparameters of DiscreteSAC in carla-cruisecontrol-v0

Hyperparameter	Value
C	4
H	0.5
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 1
nr_quantiles	25
policy_net	LinearNet((seq): Sequential((0): Linear(in_features=3, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=7, bias=True)))
q_net	LinearNet((seq): Sequential((0): Linear(in_features=3, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True) (5): ReLU() (6): Linear(in_features=512, out_features=175, bias=True)))
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.8: Hyperparameters of DiscreteQRSAC in `carla-cruisecontrol-v0`

Hyperparameter	Value
C	4
H	0.5
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 1
policy_net	LinearNet((seq): Sequential((0): Linear(in_features=3, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=7, bias=True)))
q_final_net	FinalNet((seq): Sequential((0): Linear(in_features=512, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=7, bias=True)))
q_quantiles_net	LinearNet((seq): Sequential((0): Linear(in_features=1, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True)))
q_states_net	LinearNet((seq): Sequential((0): Linear(in_features=3, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True)))
quantile_samples	10
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.9: Hyperparameters of DiscreteIQNSAC in `carla-cruisecontrol-v0`

Hyperparameter	Value
C	2 000
buffer_size	10 000
eps_coolsteps	20 000
eps_end	0.05
eps_start	1
gamma	0.99
learning_starts	257
lr	0.000 25
q_net	LinearNet((seq): Sequential((0): Linear(in_features=3, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True) (5): ReLU() (6): Linear(in_features=512, out_features=7, bias=True)))
sampling_size	256
tau	1

Table A.10: Hyperparameters of DQN in carla-cruisecontrol-v0

Hyperparameter	Value
C	2 000
buffer_size	10 000
eps_coolsteps	20 000
eps_end	0.05
eps_start	1
gamma	0.99
learning_starts	257
lr	0.000 25
nr_quantiles	25
q_net	LinearNet((seq): Sequential((0): Linear(in_features=3, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True) (5): ReLU() (6): Linear(in_features=512, out_features=175, bias=True)))
sampling_size	256
tau	1

Table A.11: Hyperparameters of QRDQN in carla-cruisecontrol-v0

Hyperparameter	Value
C	2 000
buffer_size	10 000
eps_coolsteps	20 000
eps_end	0.05
eps_start	1
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 25
q_final_net	FinalNet((seq): Sequential((0): Linear(in_features=512, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=7, bias=True)))
q_quantiles_net	LinearNet((seq): Sequential((0): Linear(in_features=1, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True)))
q_states_net	LinearNet((seq): Sequential((0): Linear(in_features=3, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True)))
quantile_samples	25
sampling_size	256
tau	1

Table A.12: Hyperparameters of IQN in carla-cruisecontrol-v0

A.4 Discrete Lane Keeping

Hyperparameter	Value
C	4
H	0.5
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 1
policy_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=9, bias=True)))
q1_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True) (5): ReLU() (6): Linear(in_features=512, out_features=9, bias=True)))
q2_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True) (5): ReLU() (6): Linear(in_features=512, out_features=9, bias=True)))
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.13: Hyperparameters of DiscreteSAC in carla-lanekeeping-relative-v0

Hyperparameter	Value
C	4
H	0.5
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 1
nr_quantiles	25
policy_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=9, bias=True)))
q_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True) (5): ReLU() (6): Linear(in_features=512, out_features=512, bias=True) (7): ReLU() (8): Linear(in_features=512, out_features=512, bias=True) (9): ReLU() (10): Linear(in_features=512, out_features=225, bias=True)))
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.14: Hyperparameters of DiscreteQRSAC in carla-lanekeeping-relative-v0

Hyperparameter	Value
C	4
H	0.5
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 1
policy_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=9, bias=True)))
q_final_net	FinalNet((seq): Sequential((0): Linear(in_features=512, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=9, bias=True)))
q_quantiles_net	LinearNet((seq): Sequential((0): Linear(in_features=1, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True)))
q_states_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True)))
quantile_samples	10
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.15: Hyperparameters of DiscreteIQNSAC in carla-lanekeeping-relative-v0

Hyperparameter	Value
`C`	50
`buffer_size`	10 000
`eps_coolsteps`	40 000
`eps_end`	0.05
`eps_start`	1
`gamma`	0.99
`learning_starts`	257
`lr`	0.000 25
`q_net`	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True) (5): ReLU() (6): Linear(in_features=512, out_features=9, bias=True)))
`sampling_size`	256
`tau`	1

Table A.16: Hyperparameters of DQN in `carla-lanekeeping-relative-v0`

Hyperparameter	Value
`C`	50
`buffer_size`	10 000
`eps_coolsteps`	40 000
`eps_end`	0.05
`eps_start`	1
`gamma`	0.99
`learning_starts`	257
`lr`	0.000 25
`nr_quantiles`	25
`q_net`	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True) (5): ReLU() (6): Linear(in_features=512, out_features=225, bias=True)))
`sampling_size`	256
`tau`	1

Table A.17: Hyperparameters of QRDQN in `carla-lanekeeping-relative-v0`

Hyperparameter	Value
C	50
buffer_size	10 000
eps_coolsteps	40 000
eps_end	0.05
eps_start	1
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 25
q_final_net	FinalNet((seq): Sequential((0): Linear(in_features=512, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=9, bias=True)))
q_quantiles_net	LinearNet((seq): Sequential((0): Linear(in_features=1, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True)))
q_states_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True)))
quantile_samples	25
sampling_size	256
tau	1

Table A.18: Hyperparameters of IQN in carla-lanekeeping-relative-v0

A.5 Discrete Lane Keeping – Lower Capacities

Hyperparameter	**Value**
C	4
H	0.5
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 1
nr_quantiles	25
policy_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=9, bias=True)))
q_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True) (5): ReLU() (6): Linear(in_features=512, out_features=225, bias=True)))
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.19: Hyperparameters of DiscreteQRSAC with lower capacity in carla-lanekeeping-relative-v0

Hyperparameter	Value
C	4
H	0.5
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 1
max_grad_norm	1
policy_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=128, bias=True) (1): ReLU() (2): Linear(in_features=128, out_features=128, bias=True) (3): ReLU() (4): Linear(in_features=128, out_features=9, bias=True)))
q_final_net	FinalNet((seq): Sequential((0): Linear(in_features=512, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=9, bias=True)))
q_quantiles_net	LinearNet((seq): Sequential((0): Linear(in_features=1, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True)))
q_states_net	LinearNet((seq): Sequential((0): Linear(in_features=6, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True)))
quantile_samples	10
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.20: Hyperparameters of DiscreteIQNSAC with lower capacity in carla-lanekeeping-relative-v0

A.6 Continuous Lane Keeping

Hyperparameter	Value
C	1
H	−1
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 15
q1_net	LinearNet((seq): Sequential((0): Linear(in_features=7, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=256, bias=True) (3): ReLU() (4): Linear(in_features=256, out_features=256, bias=True) (5): ReLU() (6): Linear(in_features=256, out_features=1, bias=True)))
q2_net	LinearNet((seq): Sequential((0): Linear(in_features=7, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=256, bias=True) (3): ReLU() (4): Linear(in_features=256, out_features=256, bias=True) (5): ReLU() (6): Linear(in_features=256, out_features=1, bias=True)))
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.21: Hyperparameters of SAC in carla-lanekeeping-v0

Hyperparameter	Value
C	1
H	-1
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 15
nr_quantiles	25
q1_net	LinearNet((seq): Sequential((0): Linear(in_features=7, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=256, bias=True) (3): ReLU() (4): Linear(in_features=256, out_features=256, bias=True) (5): ReLU() (6): Linear(in_features=256, out_features=25, bias=True)))
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.22: Hyperparameters of QRSAC in carla-lanekeeping-v0

Hyperparameter	Value
C	1
H	−1
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 15
q_final_net	FinalNet((seq): Sequential((0): Linear(in_features=256, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=1, bias=True)))
q_quantiles_net	LinearNet((seq): Sequential((0): Linear(in_features=1, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=256, bias=True)))
q_states_net	LinearNet((seq): Sequential((0): Linear(in_features=7, out_features=256, bias=True)))
quantile_samples	25
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.23: Hyperparameters of IQNSAC in carla-lanekeeping-v0

A.7 Combined Adaptive

Hyperparameter	Value
C	5
H	−1
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 15
q1_net	LinearNet((seq): Sequential((0): Linear(in_features=12, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=256, bias=True) (3): ReLU() (4): Linear(in_features=256, out_features=256, bias=True) (5): ReLU() (6): Linear(in_features=256, out_features=1, bias=True)))
q2_net	LinearNet((seq): Sequential((0): Linear(in_features=12, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=256, bias=True) (3): ReLU() (4): Linear(in_features=256, out_features=256, bias=True) (5): ReLU() (6): Linear(in_features=256, out_features=1, bias=True)))
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.24: Hyperparameters of SAC in `carla-combined-adaptive-v0`

Hyperparameter	Value
C	5
H	−1
alpha	1
buffer_size	10 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 15
nr_quantiles	25
q1_net	LinearNet((seq): Sequential((0): Linear(in_features=12, out_features=256, bias=True) (1): ReLU() (2): Linear(in_features=256, out_features=256, bias=True) (3): ReLU() (4): Linear(in_features=256, out_features=256, bias=True) (5): ReLU() (6): Linear(in_features=256, out_features=25, bias=True)))
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.25: Hyperparameters of QRSAC in carla-combined-adaptive-v0

Hyperparameter	Value
C	5
H	-1
alpha	1
buffer_size	100 000
gamma	0.99
learn_alpha	True
learning_starts	257
lr	0.000 15
lr_alpha	0.000 15
q_final_net	FinalNet((seq): Sequential((0): Linear(in_features=512, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=1, bias=True)))
q_quantiles_net	LinearNet((seq): Sequential((0): Linear(in_features=1, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True)))
q_states_net	LinearNet((seq): Sequential((0): Linear(in_features=12, out_features=512, bias=True) (1): ReLU() (2): Linear(in_features=512, out_features=512, bias=True) (3): ReLU() (4): Linear(in_features=512, out_features=512, bias=True)))
quantile_samples	25
sampling_size	256
tau	0.005
tau_pi	0.005

Table A.26: Hyperparameters of IQNSAC in carla-combined-adaptive-v0

Bibliography

[Akiba et al., 2019] Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[Bellemare et al., 2017] Bellemare, M. G., Dabney, W., and Munos, R. (2017). A distributional perspective on reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning – Volume 70*, ICML'17, page 449–458. JMLR.org.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI gym.

[CARLADocs, 2022] CARLADocs (2022). Maps and navigation. `https://carla.readthedocs.io/en/0.9.14/core_map/`. Accessed: 2023-05-01.

[Christodoulou, 2019] Christodoulou, P. (2019). Soft actor-critic for discrete action settings.

[Dabney et al., 2018a] Dabney, W., Ostrovski, G., Silver, D., and Munos, R. (2018a). Implicit quantile networks for distributional reinforcement learning. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1096–1105. PMLR.

[Dabney et al., 2018b] Dabney, W., Rowland, M., Bellemare, M. G., and Munos, R. (2018b). Distributional reinforcement learning with quantile regression. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18. AAAI Press.

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

[Duan et al., 2022] Duan, J., Guan, Y., Li, S. E., Ren, Y., Sun, Q., and Cheng, B. (2022). Distributional soft actor-critic: Off-policy reinforcement learning for addressing value estimation errors. *IEEE Transactions on Neural Networks and Learning Systems*, 33(11):6584–6598.

[Geoffrey Hinton, 2012] Geoffrey Hinton, Nitish Srivastava, K. S. (2012). Neural networks for machine learning: Lecture 6a, overview of mini-batch gradient descent. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. Accessed: 2023-04-19.

[Haarnoja et al., 2018a] Haarnoja, T., Ha, S., Zhou, A., Tan, J., Tucker, G., and Levine, S. (2018a). Learning to walk via deep reinforcement learning.

[Haarnoja et al., 2018b] Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018b). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Dy, J. and Krause, A., editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1861–1870. PMLR.

[Haarnoja et al., 2019] Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., and Levine, S. (2019). Soft actor-critic algorithms and applications.

[Huber, 1964] Huber, P. J. (1964). Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*, 35(1):73 – 101.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[Raffin et al., 2021] Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. (2021). Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8.

[Sutton, 2023] Sutton, R. S. (2023). Figures for: Reinforcement Learning: An Introduction. http://incompleteideas.net/book/figures/figures.html. Accessed: 2023-04-19.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT press.

[Ziebart, 2010] Ziebart, B. D. (2010). *Modeling Purposeful Adaptive Behavior with the Principle of Maximum Causal Entropy*. PhD thesis, USA. AAI3438449.