KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Robust End-to-End Autonomous Driving by combining Contrastive Learning and Reinforcement Learning

Paul van Tieghem de Ten Berghe

Thesis submitted for the degree of Master of Science in Mathematical Engineering

Thesis supervisor:
Prof. dr. ir. Johan Suykens

Assessors:
Prof. dr. ir. Wim Michiels
Prof. dr. Nick Vannieuwenhoven

Mentor:
Ir. Bram De Cooman

Academic year 2022 – 2023

Preface

For me, autonomous driving is like the new space race, and it is a subject I've long been passionate about. After two internship experiences focusing on the modular approach towards autonomous diving, it was a real pleasure to discover the fascinating world of end-to-end autonomous driving through reinforcement learning. Therefore I am very thankful to my thesis supervisor Professor dr. ir. Johan Suykens for providing me with this opportunity. Additionally, I would like to extend my gratitude to KU Leuven and the Flemish Supercomputer Center, who generously provided the computational resources for the experiments conducted in this thesis. Special thanks go to my mentor, Bram De Cooman, for consistently being available and supportive when I needed guidance or encountered any problems. Last but not least, I would like to thank my family for their endless support throughout the years, enabling me to pursue a degree in Mathematical Engineering.

Paul van Tieghem de Ten Berghe

Contents

Abstract

When driving, humans observe traffic situations with their eyes (input), process the information in their brains (neural network), and apply control actions to the vehicle (output). The mapping from inputs to outputs is performed by one large neural network from end to end. A possible approach towards autonomous driving, called end-to-end autonomous driving, mimics this process by training artificial neural networks with deep reinforcement learning (RL). RL is a powerful technique to learn policies purely from interactions with an environment. In visual RL, image observations are used as input, similar to how humans rely on visual information. However, visual RL agents suffer from sample-inefficiency and lack of robustness against novel scenarios. Sample-inefficiency can be addressed by using off-policy RL algorithms such as *Soft Actor-Critic* (SAC) and adding an auxiliary task with an unsupervised objective. On the other hand, the robustness of the RL agents can be improved by leveraging data augmentation. One possible approach that combines these solutions and uses contrastive learning as the unsupervised objective is the *'Contrastive Unsupervised Representations for Reinforcement Learning'* (CURL) method proposed by *Srinivas et. al., 2020* [61]. In this thesis, the CURL methodology is applied to an end-to-end autonomous driving scenario, and extended with additional data augmentations. The experiments conducted in this study compare RL agents trained using various configurations of the CURL method with each other on an end-to-end autonomous driving task. Furthermore, this thesis focuses on investigating the generalizability and robustness of CURL agents, areas that were not addressed in the original CURL paper. The findings show that using an auxiliary contrastive learning objective significantly improves the performance and sample-efficiency of RL agents compared to the SAC algorithm, even in the absence of data augmentation. Additionally, the generalizability and robustness of these agents can be improved by leveraging data augmentations. It was found that these data augmentations ought to be carefully designed because data augmentations that are not optimality-invariant or too complex tend to lead to instability during training of the RL agents, due to the brittleness of RL algorithms. Moreover, the data augmentation used in the original CURL paper was found to be unsuitable for autonomous driving. Overall, the CURL method demonstrates its effectiveness in achieving more robust and sample-efficient RL agents for end-to-end autonomous driving. Further research is warranted to explore scaling up models and simulators, among other aspects, in order to fully uncover the potential of this approach.

Samenvatting

Bij het autorijden observeren mensen verkeerssituaties met hun ogen (input), verwerken ze de informatie in hun hersenen (neuraal netwerk) en passen ze stuurhandelingen toe op het voertuig (output). De mapping van inputs naar outputs wordt uitgevoerd door één groot neuraal netwerk op een *end-to-end* manier. Een mogelijke benadering voor autonoom rijden, genaamd *end-to-end* autonoom rijden, bootst dit proces na door kunstmatige neurale netwerken te trainen met *deep reinforcement learning* (RL). RL is een krachtige techniek om *policies* te leren, puur door interactie met een omgeving. Bij visuele RL worden afbeeldingen gebruikt als input, vergelijkbaar met hoe mensen vertrouwen op visuele informatie. Visuele RL-agenten lijden echter aan sample-inefficiëntie en gebrek aan robuustheid tegen nieuwe scenario's. Sample-inefficiëntie kan worden aangepakt door *off-policy* RL-algoritmen zoals *Soft Actor-Critic* (SAC) te gebruiken en een bijtaak toe te voegen met een ongesuperviseerd objectief. De robuustheid van de RL-agenten kan worden verbeterd door gebruik te maken van data augmentatie. Een mogelijke benadering die deze oplossingen combineert en contrastief leren gebruikt als het ongesuperviseerd objectief is de *'Contrastive Unsupervised Representations for Reinforcement Learning'* (CURL) methode voorgesteld door *Srinivas et al., 2020* [61]. In deze thesis wordt de CURL-methodologie toegepast op de taak van *end-to-end* autonoom rijden. Daarnaast wordt de CURL-methode uitgebreid met extra data augmentaties. De in dit onderzoek uitgevoerde experimenten vergelijken RL-agenten die zijn getraind met verschillende configuraties van de CURL-methode met elkaar op de taak van *end-to-end* autonoom rijden. Bovendien focust deze thesis zich op het onderzoeken van de generaliseerbaarheid en robuustheid van CURL-agenten, themas die niet aan bod kwamen in het oorspronkelijke CURL-artikel. De resultaten tonen aan dat het gebruik van een bijkomend contrastief leren objectief de performantie en de sample-efficiëntie van RL-agenten aanzienlijk verbetert in vergelijking met het SAC-algoritme, zelfs zonder het gebruik van data augmentatie. Bovendien kan de robuustheid van deze agenten worden verbeterd door gebruik te maken van data augmentatie. Deze data augmentaties moeten zorgvuldig worden ontworpen omdat data augmentaties die niet optimaliteit-invariant zijn of te complex zijn, vaak leiden tot instabiliteit tijdens het trainen van de RL-agenten, vanwege de broosheid van RL-algoritmen. Verder bleek de data augmentatie die werd gebruikt in de oorspronkelijke CURL-paper ongeschikt te zijn voor autonoom rijden. In het algemeen toont de CURL-methode haar effectiviteit in het bereiken van meer robuuste en sample-efficiënte RL-agenten voor *end-to-end* autonoom rijden. Verder onderzoek is nodig om o.a. modellen

en simulatoren op te schalen, om het volledige potentieel van deze benadering te onthullen.

List of Figures and Tables

List of Figures

List of Tables

List of Abbreviations and Symbols

Abbreviations

API	Application programming interface
A3C	Asynchronous advantage actor-critic
BEV	Birds-eye-view
CPC	Contrastive predictive coding
CURL	Contrastive unsupervised representations for reinforcement learning
CNN	Convolutional neural network
DDPG	Deep deterministic policy gradient
FPS	Frames per second
GAE	General advantage estimation
GPU	Graphical processing unit
KL	Kullback-Leibler
LiDAR	Light detection and ranging
MPC	Model predictive control
MDP	Markov decision process
MLP	Multi-layer perceptron
NPC	Non player character
POMDP	Partially observable Markov decision process
PPO	Proximal policy optimization
ReLU	Rectified linear unit
RL	Reinforcement learning
RNN	Recurrent neural networks
SLAM	Simultaneous localization and mapping
SAC	Soft actor-critic
TD	Temporal-difference
TD3	Twin delayed DDPG
TPU	Tensor processing unit
TRPO	Trust region policy optimization
t-SNE	t-distributed stochastic neighbor embedding

Chapter 1

Introduction

1.1 Background and Motivation

Reinforcement learning (RL) is a powerful technique to learn policies (controllers) purely from interactions with an environment (system). In recent years, it has been applied to a wide variety of problems, ranging from simple control tasks (e.g. balancing an inverted pendulum) to complex control tasks (e.g. solving a Rubik's cube with only one robotic hand [50]). Modern deep reinforcement learning algorithms utilize approximate methods to train deep neural networks to map inputs (e.g. state information or observations) to outputs (e.g. control actions). Most reinforcement learning algorithms only require a reward associated with every interaction to train an agent on how to operate within an environment. This makes it possible to obtain controllers without explicitly programming them, which can be very useful for tasks that are too complex for rule-based systems or conventional control algorithms such as model predictive control.

In recent years, autonomous vehicles have become increasingly popular. Unfortunately, even after years of research and large investments, fully autonomous vehicles have yet to be achieved. One potential way to tackle this problem is with reinforcement learning. Autonomous driving is a complex task compared to standard reinforcement learning benchmarks such as the *Atari 2600 Games* and the *DeepMind Control Suite*, however. This can be illustrated by the fact that autonomous driving has yet to be *solved*, while reinforcement learning research has shown superhuman performance on most of the above-mentioned benchmarks [47, 7] and even on more complex games [49, 20]. As a consequence, new research within the space of reinforcement learning is needed to achieve a comparable level of success.

Ideally, we would like to obtain autonomous driving agents that are able to learn to drive purely from vision, just like humans[1]. This would eliminate the need for expensive automotive sensors by relying on just one or more cameras. In this so-called

[1]Note that this is a simplification. Humans also use their sense of balance and hearing for driving, but driving purely from vision is possible for humans.

end-to-end autonomous driving scenario, a reinforcement learning agent would learn a policy that processes these camera images as inputs, and outputs control actions to be applied to the vehicle such as acceleration, braking, and steering. This type of reinforcement learning, where the agent is given high-dimensional observations instead of state information as inputs is called visual reinforcement learning.

Sample-efficiency is one of the two major challenges in visual reinforcement learning [44, 39, 34], because learning policies directly from pixels is much more challenging than learning from precise state measurements. On top of this, the data collection (interaction with and/or simulation of the environment) in autonomous driving is much slower and computationally intense compared to benchmark RL environments. This makes the importance of sample-efficient RL methods even more important in end-to-end autonomous driving.

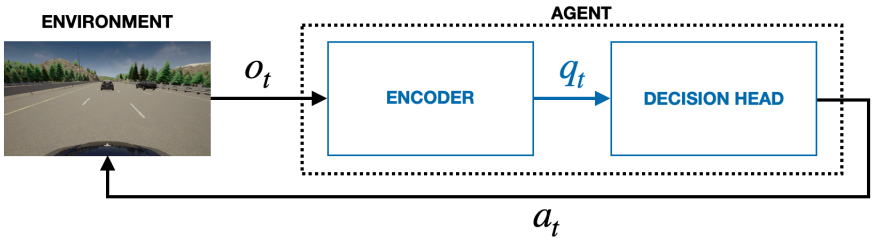


Figure 1.1: General architecture of a reinforcement learning agent in visual reinforcement learning, where actions a_t are derived from an approximate state vector q_t, extracted from observations o_t.

Since most of the state information s_t of an environment is contained within an observation o_t of the environment, it should be possible to extract approximate state information q_t in a so-called *latent space* from the image observations. This is shown in Figure 1.1. Typically, this is done by using a convolutional neural network (CNN) encoder, combined with a multi-layer perceptron (MLP) decision head. In this setting, the encoder and decision head are trained jointly from scratch by the deep reinforcement learning algorithm. It is also possible to start with an encoder that was pre-trained to accelerate training. Other methods use (variational) autoencoders for feature extraction, where these models are either pre-trained or trained jointly with the reinforcement learning objective as an auxiliary task [5, 36, 76].

The second major challenge in visual reinforcement learning (and in end-to-end autonomous driving especially) is that agents need to be able to generalize to new environments and thus be robust to variations to the environment in which they were trained. This can be compared to the human ability to easily adapt themselves to drive in a different country with different rules, lane markings, weather types, and car models for example. For this to be possible, RL algorithms need to be able to extract approximate state information q_t from camera observations in a robust way.

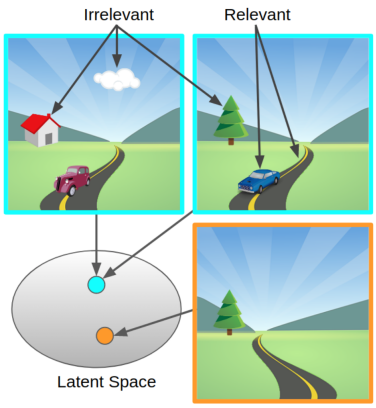


Figure 1.2: Robust representations of the visual scene should be insensitive to irrelevant objects (e.g. clouds) or details (e.g. car types), and encode two observations equivalently if their relevant details are equal (e.g. road direction and location of other cars). Figure and caption are taken from *Zhang et al., 2021* [77].

In this latent space, semantically similar situations should be encoded similarly for a robust agent, as demonstrated in Figure 1.2. If done successfully, such a robust agent should be able to generalize to novel driving environments without having to be trained in this specific environment.

Self-supervised learning methods have gained significant popularity in recent years due to the breakthrough achievements across various domains made possible by these methods. Examples include masked language modeling in the field of natural language processing [15] and contrastive learning in the field of computer vision. Self-supervised contrastive learning has been shown to learn useful representations for downstream tasks [66, 27, 11], making it a possible candidate to extract state information from visual observations in a robust way to improve the generalization of reinforcement learning agents. In self-supervised contrastive learning, data augmentations are leveraged to learn representations of raw image data without labels.

Two solutions to improve the sample efficiency in visual reinforcement learning are i) to use off-policy RL algorithms and ii) to add an auxiliary task with an unsupervised objective [76]. On the other hand, the robustness of the RL agents can be improved by leveraging data augmentation [40, 37]. A possible attempt that utilizes all these solutions was presented in the 2020 paper *CURL: '**C**ontrastive **U**nsupervised Representations for **R**einforcement **L**earning'* by *Srinivas et al.* [61], which inspired the research for this thesis. The CURL methodology combines the off-policy *Soft Actor-Critic* (SAC) reinforcement learning algorithm with an auxiliary self-supervised contrastive learning objective for representation learning. In theory, this combination of robust contrastive representation learning and the sample-efficient SAC algorithm thus seems like the ideal recipe for robust end-to-end autonomous driving.

1.2 Research Goal and Research Questions

The first research goal of this thesis is to apply the CURL methodology to an end-to-end autonomous driving task. In this thesis, end-to-end autonomous driving in a simple highway scenario is attempted using the CURL methodology. The end-to-end autonomous driving agent is trained and evaluated in a simulator and is only provided with one camera sensor in a dash cam configuration as shown in Figure 1.1 on the left. The underlying goal of this research thus aims to uncover whether the effectiveness of the CURL method can be extended from the simple benchmark environments tested in the original paper to the more complex task of end-to-end autonomous driving.

The second research goal aims at answering the question *'Does the CURL methodology improve an agent's performance and generalizability/robustness over plain SAC in end-to-end autonomous driving?'* To answer this question, both plain SAC and CURL agents will be trained and evaluated for the same set of hyperparameters. The agents trained with the plain SAC algorithm will then serve as a baseline to compare the agents trained with the CURL method to.

The third research goal aims at answering the question *'What is the influence of the data augmentations on the performance and robustness of CURL agents in end-to-end autonomous driving? Which data augmentations lead to better results, and why?'*. To answer these questions, multiple CURL agents using different data augmentations will be trained and evaluated for the same set of hyperparameters.

1.3 Contributions

The main contributions of this thesis are:

- The application of the *Soft Actor-Critic* algorithm and the CURL methodology to an end-to-end autonomous driving task in a simulated environment. The implementation of the CURL method[2] is adapted for, and fused with, a custom autonomous driving environment built on top of the CARLA simulator[3]. The code is open-sourced and can be consulted at `https://github.com/paulvantieghem/curla`.
- A comparison in terms of performance and robustness between agents trained with the plain *Soft Actor-Critic* algorithm and agents trained with the CURL method on an end-to-end autonomous driving task.
- An investigation on the effect of different data augmentations on the performance and robustness of agents trained with the CURL method on an end-to-end autonomous driving task.

[2] `https://github.com/MishaLaskin/curl`
[3] `https://github.com/carla-simulator/carla`

1.4 Thesis Structure

This thesis is structured into five chapters. These chapters are organized in the following order:

- *Chapter 1 - Introduction:* This chapter discusses the context and motivation for the research conducted in this thesis.
- *Chapter 2 - Preliminaries:* This chapter contains an overview of the subjects of autonomous driving, reinforcement learning, and self-supervised contrastive representation learning, providing the reader with the necessary theoretical foundation to better understand subsequent chapters.
- *Chapter 3 - Methodology:* In this chapter, the approach toward robust end-to-end autonomous driving by combining contrastive learning and reinforcement learning is explained. The chapter begins with a section on the design of the custom autonomous driving environment. After that, the literature on which this thesis is based is discussed, along with the extensions made to it. Lastly, the chapter presents the setup of the conducted experiments.
- *Chapter 4 - Results:* This chapter discusses the results obtained from the various experiments presented in the previous chapter. Specifically, the trained reinforcement learning agents are evaluated in terms of their performance and robustness, and compared to each other.
- *Chapter 5 - Conclusion and Future Work:* In this final chapter, the findings and conclusions derived from the research are summarized. Additionally, areas for future work and potential improvements are identified.

Chapter 2

Preliminaries

This chapter aims to introduce the reader to the most important theoretical subjects relevant to the research conducted in this thesis. The reader is assumed to have an understanding of mathematics, statistics, machine learning, and modern deep learning concepts. The first section of this text provides a brief overview of the autonomous driving landscape and introduces the reader to the concept of end-to-end autonomous driving. The second section contains a theoretical summary of the most important concepts in reinforcement learning and then branches off to the area of reinforcement learning that is most relevant in the context of this thesis: model-free, visual, deep reinforcement learning in continuous action spaces using the *Soft Actor-Critic* algorithm. The third and final section discusses representation learning, more specifically self-supervised contrastive representation learning. This preliminary knowledge will enhance the reader's comprehension of Chapter 3, where the combination of contrastive learning and reinforcement learning is explained for the pursuit of robust end-to-end autonomous driving.

2.1 Autonomous Driving

The autonomous driving industry and research community aim to develop and research systems that enable vehicles of any kind to operate autonomously. The most well-known application of this technology is the self-driving car, but other vehicles are also included within this definition, such as trucks and rail vehicles. Since cars encounter the most complex traffic situations, one could argue that a potential system that can 'solve' autonomous driving for cars would also solve autonomous driving for all other vehicles (but not the other way around). Figure 2.1 shows the six levels of autonomy according to the Society of Automotive Engineers. These levels serve as clear indicators of autonomy for industry, consumers, and legislators.

Although autonomous driving may seem like a new technology, research around autonomous vehicles has been around for almost 50 years (even using artificial neural networks [53]). In the USA, the Navlab and Autonomous Land Vehicle (ALV) [35] research projects at Carnegie Mellon University, sponsored by the Defense Advanced

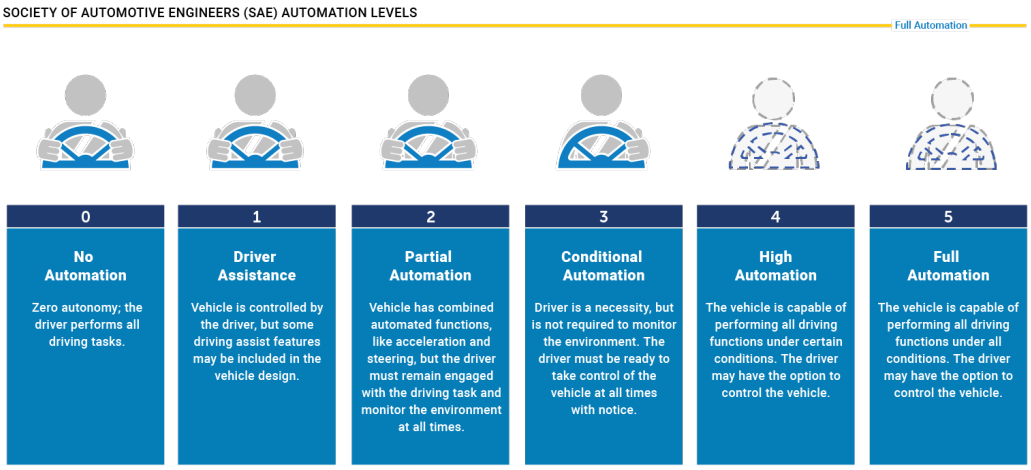


Figure 2.1: The six levels of vehicle autonomy, according to the Society of Automotive Engineers (SAE). Figure source: *Accolade Technology* website [3].

Research Projects Agency (DARPA), started in 1984. In Germany, 1987 saw the birth of the Eureka PROMETHEUS Project (PROgraMme for European Traffic of Highest Efficiency and Unprecedented Safety) by Mercedes-Benz and Bundeswehr University Munich. To this day, DARPA continues to sponsor autonomous driving research through its *Grand Challenges* [73], where university teams compete in various autonomous driving competitions.

Today, the most advanced autonomous vehicle technology and the research around it are in the hands of private companies like Tesla, Google (Waymo), Amazon (Zoox), General Motors (Cruise), Wayve, Comma etc. In the last ten years, autonomous driving has made incredible breakthroughs on all fronts thanks to the recent ($\sim$ 2013) advancements in deep learning, facilitated by the rise in computational power of modern computers [58], particularly in graphical processing units (GPU) and tensor processing units (TPU) [33].

Modern autonomous vehicles are equipped with a sensor suite, allowing the vehicle to perceive its environment in great detail. Most autonomous vehicles make use of at least one camera, but more often than not multiple cameras are used. Next to this, radars and LiDARs (***Li**ght **D**etection **A**nd **R**anging*) are used to measure distances around the vehicles. Radars emit electromagnetic waves in short pulses, which get reflected by objects in their path. Depending on the time interval between emission and reflection, the distance between the radar and the reflecting object can be determined. The problem with radars is that the electromagnetic waves only reflect well on metal objects (such as other vehicles for example). LiDARs work in a similar way but use arrays of lasers mounted on a rotating cylinder to obtain a point-cloud of distances all around the vehicle. LiDARs also suffer less from bad

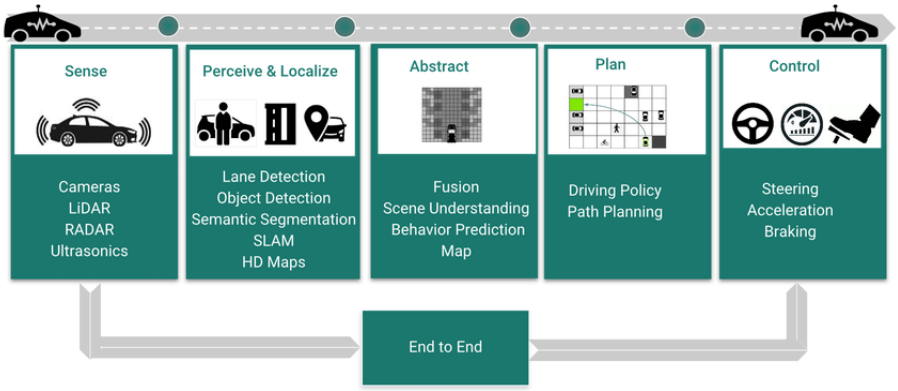


Figure 2.2: Overview of the modules in an autonomous driving pipeline. Figure source: *Ranga et al., 2020* [54].

reflections on non-metallic objects compared to radars. Finally, global navigation satellite system systems (GNSS) and tachometers are used to determine the vehicle's location and velocity, respectively.

Every autonomous driving problem is characterized by the same inputs and outputs: the sensors on the vehicle and the control actions that the vehicle needs to take, respectively, as shown in Figure 2.2. How one goes from input to output is free to choose, but two main approaches persist in modern autonomous driving: the modular approach and the end-to-end approach.

2.1.1 Modular approach

In the modular approach, the sensor information is processed through various consecutive software modules before a final control action is applied to the vehicle. The first possible module consists of *perception* and *localization*. For the *perception*, the cameras are used in combination with one or more deep learning models to perform tasks such as lane detection, object detection, object segmentation etc. The most common technique for *localization* is simultaneous localization and mapping (SLAM). In SLAM, LiDARs, radars, GNSS and tachometers are used to create a 3D map of the vehicle's surroundings, while simultaneously keeping track of the vehicle's location within this map.

In the following *abstract* module, the results from the *perception* and *localization* are fused together to create a virtual world of the vehicle's surroundings, containing all other actors and objects (other vehicles, pedestrians, traffic lights, etc.) needed for safe navigation. This virtual world may be presented from a different view, such as birds-eye-view (BEV). A visualization of such a virtual world can be seen in Figure 2.3. Other tasks may be performed within the *abstract* module, such as object tracking and behavior prediction.

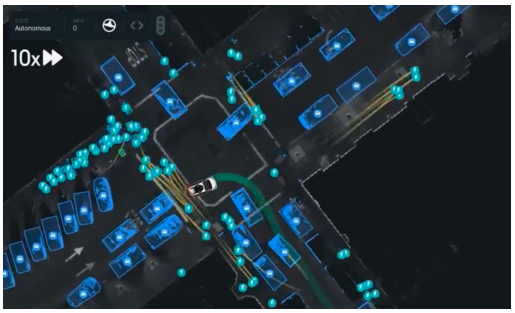


Figure 2.3: Visualization of the virtual SLAM map of a Cruise vehicle in BEV. Figure source: video found on *Cruise's* website (`https://getcruise.com`).

Using the knowledge gathered from the previous modules, the *planning* module tries to find a path for the vehicle to take. In most cases, the planning module makes use of controllers, such as model predictive control (MPC), to navigate within the virtual 3D world created by the previous module. The resulting driving policy then determines the control action to be applied to the vehicle. This control action controls the vehicle's acceleration, steering, and braking. A survey on state-of-the-art technologies in autonomous driving can be found in *Huang et al. 2020* [32].

2.1.2 End-to-end approach

In the end-to-end approach, the mapping (driving policy) between the sensor inputs and the control action outputs is characterized by one deep neural network. This neural network is then (partially) trained using imitation learning or reinforcement learning, or a combination of both [10]. In recent years, end-to-end approaches have started to become more popular, with companies such as Wayve and Comma leading the way. The advantage of the end-to-end approach is that deep neural networks are able to learn tasks that are hard to manually specify, but also to extract useful features from the input sensors that are hard to manually design. The main drawback of end-to-end autonomous driving is the lack of explainability or interpretability. Deep neural networks are essentially black-box models that learn a mapping between inputs and outputs from large amounts of data. This lack of explainability is a major deal-breaker for people (especially legislators), because no line of code or human error can be pinpointed as the culprit in the event of a crash. For this reason, driving simulators are used to validate the safety and performance of these end-to-end models over millions of (virtual) kilometers before deploying them in the real world [1]. Finally, note that hybrid approaches are possible, where aspects of modular and end-to-end autonomous driving are combined.

[1]These blog posts from Wayve explain the importance of simulators in more detail: *'Introducing Wayve Infinity Simulator'* (`https://wayve.ai/thinking/introducing-wayve-infinity-simulator/`), *'Evaluating driving performance in diverse simulated world'* (`https://wayve.ai/thinking/evaluating-driving-performance-in-diverse-simulated-worlds/`)

2.2 Reinforcement Learning

Reinforcement learning is one of the three main branches of machine learning, next to supervised learning and unsupervised learning. In reinforcement learning, one aims to learn an optimal policy for an agent by interacting with an environment. The optimal policy is the set of actions the agent ought the take, following observations given by the environment, to maximize its cumulative reward. For example, reinforcement learning could be applied to the game of *Snake*, where the agent controls the snake and tries to maximize its cumulative reward, the length of the snake, by interacting with its environment, the game.

2.2.1 The Agent-Environment Interaction Loop

The reinforcement learning interaction loop is depicted in Figure 2.4, and is compromised of two main blocks: the *agent* and the *environment* (or *world*). The agent exists and acts within the environment, and is thus also part of the environment.

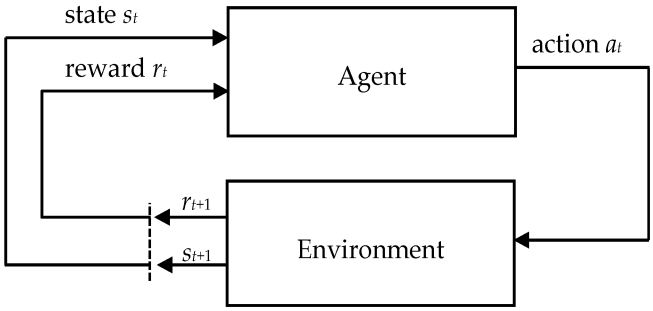


Figure 2.4: General structure of the reinforcement learning agent-environment interaction loop. Figure source: *Sutton & Barto, 1998* [62].

At each moment in time t, the agent receives the *state* s_t of the world and a *reward signal* r_t from the environment. The reward r_t is a scalar value indicating how good or bad the current state of the world is, which is calculated by the *reward function* of the environment. The agent then performs an *action* a_t, calculated from its policy, based on the current state s_t of the world. The *policy* π of the agent is a function that translates the current state of the world to an action the agent will take. At each moment in time, the world changes by itself and is influenced by the action a_t taken by the agent, resulting in a new state s_{t+1} and a reward r_{t+1}. The goal of reinforcement learning is for the agent to maximize its return, which is the cumulative reward it obtained by interacting with the environment. An optimal policy will thus maximize long-term cumulative reward instead of immediate rewards.

2.2.2 Formalizing the General Reinforcement Learning Problem

First, a formal definition of *the Reinforcement Learning Problem* [62] is presented.

Definition 2.2.1 (The Reinforcement Learning Problem [62])**.** *A reinforcement learning task is defined as follows:*

Given

- *a set of states* $\mathcal{S}$
- *a set of actions* $\mathcal{A}$
- *a (possibly unknown) start-state distribution* ρ_0*, from which the initial state* $s_0 \in \mathcal{S}$ *is sampled:* $s_0 \sim \rho_0(\cdot)$
- *a (possibly unknown) transition function (or model)* $\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$
- *a (possibly unknown) reward function* $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$

Find *a policy* $\pi : \mathcal{S} \to \mathcal{A}$ *that maximizes a value function* $V^\pi(s)$ *for all states* $s \in \mathcal{S}$*. The utility value* $V^\pi(s)$ *is based on the rewards received starting from state* s *and following policy* π*.*

Definition 2.2.2 (Policy function)**.** *The policy function* $\pi : \mathcal{S} \to \mathcal{A}$ *of an agent determines which action* a *the agent will take depending on the state* s *of the environment in which the agent is situated. Policies can either be deterministic, denoted by*

$$a_t = \mu(s_t) \equiv \pi(s_t),$$

or stochastic, denoted by

$$a_t \sim \pi(\cdot|s_t).$$

Definition 2.2.3 (Trajectory [4])**.** *A trajectory or rollout* τ *is a sequence of state-action pairs:*

$$\tau = (s_0, a_0, s_1, a_1, ...).$$

The transition from one state to another, depending on the most recent action taken by the agent, is determined by the transition function or world model δ*, which can be deterministic or stochastic. In the deterministic case state transitions are denoted by*

$$s_{t+1} = \delta(s_t, a_t),$$

while in the stochastic case state transitions are denoted by

$$s_{t+1} \sim P_\delta(\cdot|s_t, a_t).$$

In the prior definition, it is assumed that δ and P_δ are only dependent on the current state s_t and action a_t. This assumption is called the Markov property, where the environment can then be modeled as a Markov decision process (MDP). The Markov property allows the agent to make decisions based solely on the current state without having to take its history (prior trajectory) into account.

Definition 2.2.4 (Episode [9]). *Episodic environments are worlds where the agent will eventually end up in a terminal state, after which the environment will be reset for the next trial. In episodic environments, trajectories are finite and will always end in a terminal state. The sequence of states, actions, and rewards that starts in the* ***initial state*** s_0 *and ends in a* ***terminal state*** s_N *is called an* ***episode****. In episodic environments, a* ***done signal*** d *from the environment will be provided to the agent at each time step, indicating whether a terminal state has been reached or not, where*

$$d_t = \begin{cases} 0 & \text{if } s_t \text{ is not a terminal state.} \\ 1 & \text{if } s_t \text{ is a terminal state.} \end{cases}$$

Definition 2.2.5 (Reward and Return [4]). *The reward function determined by the environment depends on the current state of the world, the action just taken, and the next state of the world*

$$r_t = r(s_t, a_t, s_{t+1}),$$

although frequently this is simplified to just a dependence on the current state, $r_t = r(s_t)$, or state-action pair $r_t = r(s_t, a_t)$. This choice depends on how the reward function is specified for the environment. The cumulative reward over a trajectory τ is called the return R. The return of a trajectory can be defined in different ways, depending on the value of the ***discount factor*** $\gamma \in (0, 1]$ *and on the horizon T of the trajectory, namely*

- *the* ***finite-horizon (un)discounted return*** $R(\tau) = \sum_{t=0}^{T} \gamma^t r_t$,
- *the* ***infinite-horizon (un)discounted return*** $R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$,

where the undiscounted returns have a discount factor $\gamma = 1$. A discount factor $\gamma \in (0, 1)$ guarantees that the discounted return $R(\tau)$ is a finite sum (which adds convergence properties in exact solution methods) and also adds an incentive to the agent to collect rewards as soon as possible rather than further down the trajectory.

Summary The very first state of the world, s_0, is randomly sampled from the start-state distribution as $s_0 \sim \rho_0(\cdot)$. At each moment in time, the environment will be situated in a state $s_t \in \mathcal{S}$, for which the agent will execute the action $a_t \sim \pi(\cdot|s_t)$, with $a_t \in \mathcal{A}$. After this action, the new state of the environment $s_{t+1} \in \mathcal{S}$ follows from the transition function P_δ as $s_{t+1} \sim P_\delta(\cdot|s_t, a_t)$. The reward r_t calculated by the reward function r following the action taken by the agent is $r_t = r(s_t, a_t)$, with r_t a scalar. The consecutive state-action pairs (s_t, a_t) form a trajectory τ along which rewards r_t are collected. The cumulative reward obtained along this trajectory is the return $R(\tau)$.

Definition 2.2.6 (Expected return [4])**.** *The goal in reinforcement learning is to obtain a policy that maximizes the expected return. For a stochastic transition function and policy, the probability of a T-step trajectory τ given the policy π is defined as*

$$P(\tau|\pi) = \rho_0(s_0) \prod_{t=0}^{T-1} P_\delta(s_{t+1}|s_t, a_t)\pi(a_t|s_t).$$

The expected return $J(\pi)$ is then defined as:

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

Definition 2.2.7 (Value functions and Q-functions [4])**.** *The value of a state $V^\pi : \mathcal{S} \to \mathbb{R}$ or value of a state-action pair $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the expected return when starting in that state or state-action pair, and then acting according to a particular policy forever after. Value functions are used, one way or another, in almost every RL algorithm. Action-value functions are referred to as "**Q-functions**", and the actual value of a state-action pair is then called the "**Q-value**" of that state-action pair. There are four main functions of note here:*

- *The **On-Policy Value Function**, V^π, which gives the expected return when starting in state s and always acting according to the policy π, denoted by*

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

- *The **On-Policy Action-Value Function**, $Q^\pi(s, a)$, which gives the expected return when starting in state s, taking an action a (which may not have come from the policy), and then forever after acting according to the policy π, denoted by*

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

- *The **Optimal Value Function**, V^*, which gives the expected return when starting in state s and always acting according to the optimal policy π^*, denoted by*

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

- *The **Optimal Action-Value Function**, Q^*, which gives the expected return when starting in state s, taking an action a (which may not have come from the policy), and always acting according to the optimal policy π^*, denoted by*

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

The value function and the action-value function can be linked to each other as follows:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{a\sim\pi}\left[Q^{\pi}(s,a)\right],$$

$$V^{*}(s) = \max_{a} Q^{*}(s,a).$$

To find an optimal policy, one could learn the optimal Q-values Q^* and just select the action with the maximal Q^*:

$$\mu^{*}(s) = \arg\max_{a} Q^{*}(s,a).$$

Definition 2.2.8 (Advantage function [4])**.** *The advantage function $A^{\pi}(s,a)$ corresponding to a policy π quantifies how much better it is for the agent to take a specific action a in state s, over randomly selecting an action according to $\pi(\cdot|s)$, assuming that the agent acts according to π forever after. Mathematically, the advantage function is defined by*

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s).$$

The advantage $A^{\pi}(s,a)$ is thus used to quantify the relative advantage of taking a specific action a over other actions on average when situated in state s.

The Bellman Equations

The four value functions presented above obey the Bellman Equation. Intuitively, this property can be explained as follows: *"The value of your starting point is the reward you expect to get from being there, plus the value of wherever you land next."* [4]. This property allows us to rewrite the value functions as a recursive sequence:

$$\begin{aligned}
V^{\pi}(s) &= \mathop{\mathbb{E}}_{a\sim\pi, s'\sim P_{\delta}}\left[r(s,a) + \gamma V^{\pi}(s')\right], \\
Q^{\pi}(s,a) &= \mathop{\mathbb{E}}_{s'\sim P_{\delta}}\left[r(s,a) + \gamma \mathop{\mathbb{E}}_{a'\sim\pi}\left[Q^{\pi}(s',a')\right]\right], \\
V^{*}(s) &= \max_{a} \mathop{\mathbb{E}}_{s'\sim P_{\delta}}\left[r(s,a) + \gamma V^{*}(s')\right], \\
Q^{*}(s,a) &= \mathop{\mathbb{E}}_{s'\sim P_{\delta}}\left[r(s,a) + \gamma \max_{a'} Q^{*}(s',a')\right].
\end{aligned}$$

with $P_{\delta}(\cdot|s_t, a_t)$ the stochastic transition function of the environment, and with s', a' shorthand notations for s_{t+1}, a_{t+1}, respectively.

2.2.3 Branching off into relevant areas of RL for this thesis

The domain of reinforcement learning is quite vast and intricate. In order to keep this chapter relevant to the reader, the text now branches off into more specific areas of reinforcement learning that are relevant to the research presented in this thesis. Specifically, we will branch off into *model-free, visual, deep reinforcement learning in continuous action spaces.*

Action Spaces: Discrete vs. Continuous

Depending on the environment, the action space in which the agent acts can either be discrete or continuous. An example of an environment with a discrete action space is the game of chess, where only a finite amount of discrete moves can be executed at any moment. Environments where the agent performs a control problem often have a continuous action space. An example of an environment with a continuous action space is a racing game, where actions are two-dimensional real-valued vectors $a \in \mathbb{R}^2$, with $a_i \in [-1, 1]$, indicating how much the car should accelerate and turn. The difference between these two cases has significant implications for reinforcement learning methods. Certain categories of algorithms are applicable only in one situation and would require significant modifications for the other scenario.

Deep Reinforcement Learning

Policies, value functions, and Q-functions are vector-valued functions. For simple environments with discrete action spaces, these can be represented in a tabular manner. *Tabular solution methods* are the methods in traditional reinforcement learning that are concerned with solving these types of problems using dynamic programming (policy iteration, value iteration), Monte Carlo methods, etc. An in-depth overview of these methods can be found in the book *'Reinforcement Learning: An Introduction'* by Richard Sutton and Andrew Barto [62].

For more complex environments however, these simple representations would either be too simple and thus fail to encapsulate the complexity of the environment or be so large that even modern computers would not be able to store and/or compute all possible values. For this reason, modern RL algorithms try to fit functions using *approximate solution methods* to try to find optimal policies.

In deep reinforcement learning, parameterized functions or artificial neural networks are used to represent policies, value functions, and Q-functions. The parameters of these functions can then be tuned or learned by using optimization algorithms such as gradient descent/ascent. This is particularly advantageous for visual environments where the agent does not get a state vector from the environment, but rather a high-dimensional image of (a part of) the environment. In these situations, deep learning architectures such as convolutional neural networks (CNN) can be leveraged to extract spatial features from these images, and recurrent neural networks (RNN) can be leveraged to extract temporal information from these observations [41]. These features can then be combined to approximate state information. Policies, value functions and Q-functions that are represented by an artificial neural network get an adapted notation

$$\pi_\theta(\cdot|s) \quad V_\phi(s) \quad Q_\psi(s, a),$$

where θ, ϕ or ψ indicate that the functions are parameterized by the set of parameters θ, ϕ or ψ, respectively.

Definition 2.2.9 (Experience Replay and Replay Buffer [52]). *Experience replay is a replay memory technique used in reinforcement learning where the agent's experiences* $e_t = (s_t, a_t, r_t, d_t, s_{t+1})$ *or* $e_t = (o_t, a_t, r_t, d_t, o_{t+1})$ *are stored at each time-step in a dataset* $\mathcal{B} = \{e_1, e_2, ..., e_N\}$ *called the replay buffer. This memory is then usually sampled randomly for a mini-batch of experience. This tackles the problem of autocorrelation leading to unstable training in deep reinforcement learning, by making the problem more like a supervised learning problem.*

Visual Reinforcement Learning: Observations vs. States

Visual reinforcement learning refers to the application of RL algorithms in environments where no direct state information about the world is available or provided to the agent. Instead of providing state vectors s_t, the environment only provides *observations* o_t of (a certain part of) the world to the agent. These observations o_t are high dimensional images, from which no quantitative state values are directly available from. The observations only contain partial state information, and the environment is thus modeled as a *partially observable Markov decision process (POMDP)* [2]. To put things in perspective, a simple 100×100 pixel color image (3 channels) forms a $100 \cdot 100 \cdot 3 = 30000$ dimensional observation vector o_t. Visual RL and deep RL thus go hand in hand due to the high dimensional nature of visual RL and due to the fact that no quantitative state values are directly available from observations in visual RL. Deep neural networks are then leveraged to extract relevant state information from high-dimensional observations.

Definition 2.2.10 (Frame-stacking). *Observations* o_t *are image captures of the environment at a particular moment in time, lacking any kind of temporal information. For this reason, most algorithms in visual RL will use a concept called frame-stacking in order to incorporate temporal information into the state representation. The observations in the current episode are kept in a sequence* $\mathcal{S}_t = o_1, o_2, ..., o_t$*. At every time-step, a function* ϕ *recovers the last* k *observations from the sequence* $\mathcal{S}_t$*, preprocess the observations, and stacks the last* k *observations into a* k*-frame 'frame-stack'. This frame-stack is then used as the observation.*

Model-Free vs. Model-Based Reinforcement Learning

An important factor in RL is whether a model of the world is available to the agent. A model of the world contains the transition function δ or P_δ and reward function r. In model-based RL, the model of the world is given, approximated, or learned. It is kind of similar to model predictive control, where one can derive (white box model using differential equations) or approximate/learn (gray/black box model using system identification) a model of the dynamics of a system, and then solve an optimization problem at each step in time to select the optimal action to control the system. In this case, the policy is never explicitly represented. Model-based RL is not discussed any further, because this thesis focuses solely on model-free RL algorithms.

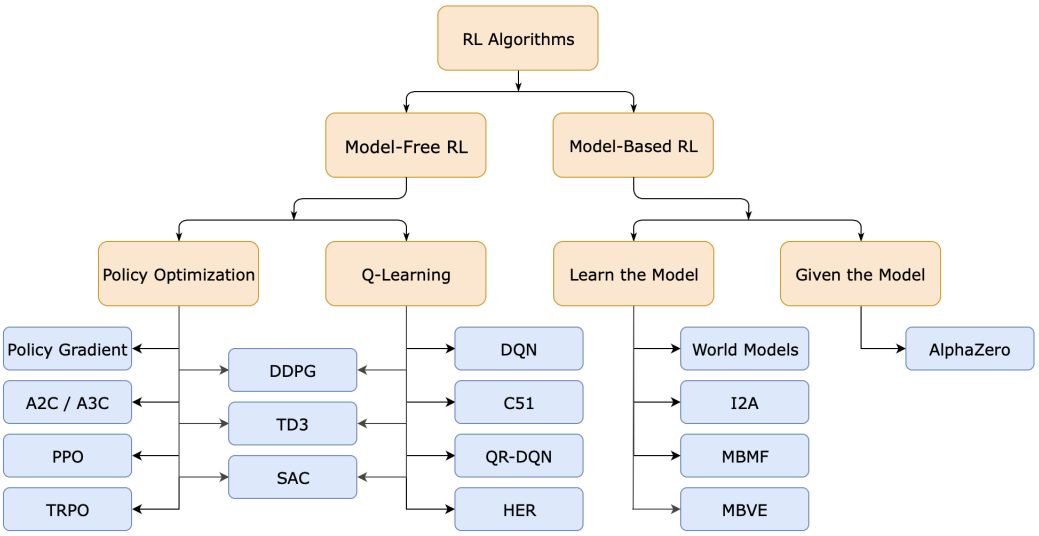


Figure 2.5: A non-exhaustive overview of modern deep reinforcement learning algorithms[2]. Figure source: *OpenAI Spinning Up* course [4].

Model-free RL algorithms are RL methods where the agent does not have access to a model of the world. In this case, sampling-based approximations will have to be made in order to get an understanding of the world and its dynamics. An overview of deep reinforcement learning algorithms for both model-free and model-based RL is shown in Figure 2.5. As shown in Figure 2.5, model-free RL algorithms can be divided into three categories:

1. Value-based algorithms: *Q-learning* and its variants
2. Policy-based algorithms: *Policy optimization*
3. Algorithms that interpolate between the two previous types of algorithms: *actor-critic* methods

All the limitations that one can encounter in RL in the discussion above are relevant to the task of end-to-end autonomous driving. A summary of these limitations, their solutions, and corresponding branch of the subject matter can be found in Table 2.1. The next sections provide an overview of value-based, policy-based, and actor-critic algorithms in model-free reinforcement learning. Note that model-based RL can be used in autonomous driving by learning a model of the world, but this thesis focuses solely on model-free RL.

[2]Algorithm citations: A2C/A3C [45], PPO [57], TRPO [55], DDPG [43], TD3 [21], SAC [24], DQN [46], C51 [8], QR-DQN [14], HER [6], World Models [22], I2A [71], MBMF [48], MBVE [19], AlphaZero [59].

Limitation	Solution	Branch of RL
No access to a dynamics model of the world	Sampling based approximations	Model-free RL
Environment too complex for tabular solution methods	π, Q, V function fitting by gradient-based optimization of artificial neural networks	Deep RL
High-dimensional observations containing indirect, partial state information	Artificial neural network based feature extractors	Visual RL

Table 2.1: Summary of the RL-related limitations encountered in an autonomous driving environment, how these limitations are tackled in this thesis, and what area of RL these topics belong to.

2.2.4 Q-learning and its variants

Value Iteration

Value iteration of the Q-function is an exact solution method based on the Bellman equation

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim P_\delta} \left[r(s,a) + \gamma \max_{a'} Q^*(s',a') \right] \tag{2.1}$$

that performs iterative Bellman-backups

$$Q_{k+1}(s,a) \leftarrow \mathop{\mathbb{E}}_{s' \sim P_\delta} \left[r(s,a) + \gamma \max_{a'} Q_k(s',a') \right], \tag{2.2}$$

which are guaranteed to converge toward the optimal Q-function Q^* in the limit [62]. The optimal policy can then implicitly be found as:

$$\mu^*(s) = \arg\max_a Q^*(s,a).$$

Tabular Q-Learning

Due to the first limitation in Table 2.1, the value iteration algorithm is not suited for autonomous driving. The transition function P_δ is not available in model-free RL, which makes the calculation of the expected values in the Bellman-backup in Equation 2.2 unfeasible. Tabular Q-learning [70] modifies the value iteration algorithm by replacing the straight update-rule of the Q-values using expected values in Equation 2.2 with an update-rule that computes the average of the sampled observed values [9]. To obtain this average (approximation of the expected value), an exponential moving average update with parameter α is carried out instead of a full backup, which results in the update rule

$$Q(s,a) \leftarrow (1-\alpha) Q(s,a) + \alpha \left[r(s,a) + \gamma \max_{a'} Q(s',a') \right]. \tag{2.3}$$

Rewriting Equation 2.3 results in

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a) \right], \tag{2.4}$$

which shows how the update term is proportional to the *temporal difference* (TD): the difference between the new estimate $r(s,a) + \gamma \max_{a'} Q(s',a')$ for $Q(s,a)$ and the current estimate $Q(s,a)$. The Q-learning algorithm is shown in Algorithm 1.

Algorithm 1: The (tabular) Q-learning algorithm [9, 1]

Input : Discount factor γ, exponential moving average learning rate α
Output : The optimal state-action values Q^*

```
Initialize Q randomly
Initialize s randomly
while Q has not converged do
    Select an action a ϵ-greedily
    Take action a, observe r(s, a) and s'
    if s' is terminal then
        Set target T = r(s, a)
        Initialize new s randomly
    else
        Set target T = r(s, a) + γ max_a' Q(s', a')
    Q(s, a) ← (1 − α)Q(s, a) + αT
return Q
```

It can be shown that Q-learning is guaranteed to converge to the optimal Q-values Q^* in the limit, and thus find an optimal policy. For proof of this convergence, the reader is referred to *Watkins et al., 1992* [70]. In theory, this requires every state-action pair to be visited infinitely often, but in practice, convergence can be observed much sooner [4]. Since the Q-learning algorithm can be executed without acting on a policy, it is classified as an *off-policy learning* algorithm.

Exploration vs Exploitation: ϵ-greedy sampling and annealing

In reinforcement learning, the agent's policy is initialized randomly. In Q-learning, this policy inherits its initial randomness from the randomly initialized Q-function the agent utilizes to derive its policy. This randomness causes the agent to *explore* its environment, causing it to eventually have performed each possible action. This is desirable because it enables updates on all possible Q-values. After a while, the learned Q-values will start to approximate the optimal Q-values Q^*, which will in turn lead to a policy that approximates the optimal policy. In this phase, the agent *exploits* its knowledge to perform actions that are more optimal. This has the upside of making some Q-values more and more accurate (close to the optimal Q-values)

but has the downside of lacking exploration. Exploration is important because the optimal policy may not be the most straightforward policy. If an agent starts to exploit a sub-optimal policy too early, it could lack the exploration to learn an alternative policy that is much closer to the optimal policy. This challenge is called the *trade-off between exploration and exploitation.*

A popular technique to force the agent to always keep exploring is ϵ-greedy sampling, where the policy is adapted to select a random action $a \in \mathcal{A}(s)$ with probability ϵ instead of the action normally taken by the policy. For Q-learning, this ϵ-greedy sampling is defined below:

$$a = \begin{cases} \sim \text{uniform distribution over } \mathcal{A}(s) & \text{with probability } \epsilon \\ \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon \end{cases} \tag{2.5}$$

The problem with ϵ-greedy sampling is the fixed ϵ. Choosing ϵ too low might result in too little exploration, causing the agent to start exploiting a sub-optimal policy too early and never learning a close-to-optimal policy. Choosing ϵ too high might result in too little exploitation, causing the agent to never start exploiting a policy and thus acting too randomly forever. To remedy these problems, the ϵ value in ϵ-greedy sampling can be annealed. Annealing ϵ consists of starting with a relatively high value for ϵ, and then gradually lowering it throughout the iterations. This scheme will result in high exploration at the start of training and high exploitation at the end of training, which is the ideal trade-off between exploration and exploitation. The initial value of ϵ and the rate at which ϵ is decreased are hyperparameters that need to be tuned in order to achieve good results.

Deep Q Networks (DQN)

The second and third limitations in Table 2.1 make the use of tabular solution methods like tabular Q-learning unfeasible. For this reason, the table in Q-learning is replaced by a parameterized Q-function or Q-network Q_θ. The targets $\mathcal{T}$ (see Algorithm 1) can then be used as labels in a kind of supervised learning setting to modify the parameters or weights of Q_θ using gradient descent. The main difference with traditional supervised learning is that the labels used in the calculation of the value of the loss function depend on the current iteration i. When optimizing the loss function $L_i(\theta_i)$, the Q-network with weights from the previous iteration θ_{i-1} (called the target Q-network $\hat{Q}$) is used for the calculation of the target value. A gradient descent step can formally be written down as:

$$\theta_{i+1} \leftarrow \theta_i - \alpha \nabla_\theta L(\theta)|_{\theta=\theta_i}$$

with target values T_i and loss function $L_i(\theta_i)$:

$$\begin{cases} \mathcal{T}_i = \mathbb{E}_{s' \sim P_\delta}\left[r(s,a) + \gamma \max_{a'} Q_{\theta_{i-1}}(s',a')\right] \\ L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[\mathcal{T}_i - Q_{\theta_i}(s,a)\right]^2 \end{cases} \tag{2.6}$$

The original DQN paper by *Mnih et al., 2013* [46] combines the adaptations to the Q-learning algorithm presented above, but modified to perform stochastic gradient descent on a mini-batch of experiences sampled from a replay buffer (see Definition 2.2.9), rather than computing the full expectations as in Equation 2.6. The algorithm to train these DQN is called *Deep Q-learning with experience replay* and is presented in Algorithm 2. The algorithm is presented for visual RL, where the agent receives high-dimensional observations o_t (images) rather than direct states s_t. To incorporate temporal information into the observations, the frame-stacking technique is used as in Definition 2.2.10.

Notice that Algorithm 2 utilizes two Q-networks: the Q-network Q_θ and *target* Q-network $\hat{Q}_{\theta^-}$. This modification to the original DQN algorithm of *Mnih et al., 2013* [46] was introduced in a follow-up paper by *Mnih et al., 2015* [47]. The target Q-network $\hat{Q}_{\theta^-}$ is a stable version of the Q-network Q_θ, used for generating the targets $\mathcal{T}_j$ for the loss function. The weights θ^- of the target Q-network are set equal to the weights of the Q-network every C steps, but remain untouched in between these synchronizations. According to *Mnih et al., 2015*, *"this modification makes the algorithm more stable compared to standard online Q-learning, where an update that increases $Q_\theta(s_t, a_t)$ often also increases $Q_\theta(s_{t+1}, a)$ for all a and hence also increases the target $\mathcal{T}_j$, possibly leading to oscillations or divergence of the policy. Generating the targets using an older set of parameters adds a delay between the time an update to Q_θ is made and the time the update affects the targets $\mathcal{T}_j$, making divergence or oscillations much more unlikely"* [47].

2.2.5 Policy Optimization

In policy-based model-free deep reinforcement learning, the policy is represented explicitly as an artificial neural network, and the policy is optimized directly. This is opposed to value-based RL, where the policy is implicitly derived as the action that would maximize the Q-function (an artificial neural network in deep RL). The simplest algorithm within the class of policy-based model-free deep reinforcement learning is called (vanilla) policy gradient or (V)PG. Direct policy learning can only find *locally* optimal policies however: *"Unfortunately, learning the policy π^* directly is not trivial. The relationship between $Q(s, a)$ and $Q(s', a')$ is relatively clear, and Q-values for successor states carry information about the Q-value for a given state. Simply knowing the optimal action for the successor states, however, does not help with finding the optimal action for the current state. Therefore, it is hard to avoid the use of state- or state-action-values altogether. It is possible, however, to generate a locally optimal policy without the use of an explicit representation of the Q-function, or the need to compute Q-values for all possible state-action pairs."* [9].

Policy optimization has the advantage of being an easier problem to solve than Q-learning based algorithms for some environments, because often learning a policy π can be simpler than learning a Q-function Q or a value function V [1]. Next to this, some extra complexity arises when learning Q and/or V. For example, given

Algorithm 2: Deep Q-learning with experience replay [46, 1]

Input : Discount factor γ, learning rate α, target network update frequency C, number of episodes M, episode horizon T

Output : A Q-function Q_θ approximating Q^*

Initialize replay buffer $\mathcal{B}$ with capacity N
Initialize the Q-network Q_θ with random weights θ
Initialize the target Q-network $\hat{Q}_{\theta^-}$ with weights $\theta^- = \theta$
for *episode* $= 1, M$ **do**
    Initialize sequence $\mathcal{S}_1 = \{o_1\}$ and preprocessed frame-stack $\phi_1 = \phi(\mathcal{S}_1)$
    for $t = 1, T$ **do**
        Select an action a_t ϵ-greedily
        Take action a_t, observe reward r_t and observation o_t
        Set $\mathcal{S}_{t+1} = \{\mathcal{S}_t, o_{t+1}\}$ and preprocess $\phi_{t+1} = \phi(\mathcal{S}_{t+1})$
        Store experience $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{B}$
        Sample random mini-batch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$
        $\mathcal{T}_j = \begin{cases} r_j & \text{if } \phi_{j+1} \text{ is terminal} \\ r_j + \gamma \max_{a'} \hat{Q}_{\theta^-}(\phi_{j+1}, a') & \text{if } \phi_{j+1} \text{ is not terminal} \end{cases}$
        Perform a gradient descent step w.r.t. θ on $[\mathcal{T}_j - Q_\theta(\phi_j, a_j)]^2$, according to Equation 2.6
        Every C steps, reset $\hat{Q}_{\theta^-} = Q_\theta$
return Q

a learned value function V, the action to take is not directly available. One would have to obtain (learn) a dynamics model to derive the actions to take from the value function. For Q-learning, there is the problem of computing the optimal action from the learned Q-function using $a = \arg\max_a Q_\theta(s, a)$, which can become really complex (slow) for continuous or high-dimensional action spaces. [1]

(Vanilla) Policy Gradient

In policy gradient, the stochastic policy $\pi_\theta(\cdot)$ is represented as a parameterized function or artificial neural network with parameters or weights θ. The policy is initialized randomly by setting the weights θ to random values. The goal is to optimize the weights θ of the π_θ network with gradient ascent to maximize an objective function $J(\pi_\theta)$. For policy gradient, the objective function $J(\pi_\theta)$ is the expected return of the policy (see Definition 2.2.6), defined as

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta, P_\delta} [R(\tau)] \,. \tag{2.7}$$

The weights θ of the parameterized policy π_θ then get updated by gradient ascent steps with learning rate α as

$$\theta_{i+1} \leftarrow \theta_i + \alpha \nabla_\theta J(\pi_\theta)|_{\theta=\theta_i}.$$

The gradient $\nabla_\theta J(\pi_\theta)$ is called the *policy gradient*. In order to perform the parameter update above, the policy gradient $\nabla_\theta J(\pi_\theta)$ needs to be computable. The analytical derivation of the policy gradient is out of the scope of this text but can be consulted in *Sutton & Barto, 2017; Sec. 13.1* [62] and in *OpenAI Spinning Up; Part 3: Intro to Policy Optimization* [4]. The resulting analytical expression for the policy gradient is

$$\nabla_\theta J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta, P_\delta}\left[\sum_{t=0}^{T} R(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)\right],$$

where the values of $\pi_\theta(a_t|s_t)$ are probabilities. Just as in deep Q learning, we will approximate the gradient by performing stochastic gradient descent on a mini-batch of experienced trajectories instead of calculating full expectations. The difference with deep Q-learning is that policy gradient is an *on-policy* algorithm, meaning that we cannot use random experiences from a replay buffer, but rather a collection of trajectories $\mathcal{D} = \{\tau_i\}_{i=1,\dots,N}$, obtained by following the policy π_θ. The final expression for the approximated policy gradient g is then:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\left[\sum_{t=0}^{T} R(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)\right] \equiv g. \tag{2.8}$$

Exploration vs. Exploitation

If the policy is chosen to be a deterministic policy μ, the same ϵ-greedy scheme as in Q-learning can be used to regulate the trade-off between exploration and exploitation, where actions a are determined as

$$a = \begin{cases} \sim \text{ uniform distribution over } \mathcal{A}(s) & \text{with probability } \epsilon. \\ \mu(s) & \text{with probability } 1-\epsilon. \end{cases} \tag{2.9}$$

Stochastic policies π_θ on the other hand are already stochastic in nature, and thus inherently handle the trade-off between exploration and exploitation. For continuous action spaces, a stochastic policy $\pi_\theta(\cdot|s)$ can be represented by a multivariate Gaussian distribution $\mathcal{N}(\mu_\theta(s), \Sigma)$, with mean vector $\mu_\theta(s)$ and $\sigma_\theta(s)$ the diagonal elements of the covariance matrix Σ. Here μ_θ and σ_θ are parameterized functions or neural networks. Actions can then be sampled from the policy as

$$a = \mu_\theta(s) + \sigma_\theta(s) \odot z, \quad \text{with} \quad z \sim \mathcal{N}(0, I). \tag{2.10}$$

Policy Gradients in General Form

Through the links between expected return $J(\pi)$, the value function V^π and the Q-function Q^π (see Definitions 2.2.6 and 2.2.7), the policy gradient g in Equation 2.8 can actually be expressed in many different forms. As per *Schulman et al., 2018* [56], the approximate policy gradient g can be generally expressed as

$$g = \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\left[\sum_{t=0}^{T} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right] \tag{2.11}$$

where Ψ_t may be on of the following:

$$\Psi_t = \begin{cases} \sum_{t=0}^{\infty} r_t & : \text{the total reward of the trajectory.} \\ \sum_{t'=t}^{\infty} r_{t'} & : \text{the reward following action } a_t. \\ r_t + V^{\pi}(s_{t+1}) - V^{\pi}(s_t) & : \text{the temporal-difference residual.} \\ A^{\pi}(s_t, a_t) & : \text{the advantage function.} \\ Q^{\pi}(s_t, a_t) & : \text{the Q-function.} \end{cases} \tag{2.12}$$

Every option for Ψ_t in Equation 2.12 leads to the same expected value for the policy gradient, but with different variances [56]. High-variance policy gradients will need more samples to approximate the expected value compared to low-variance policy gradients. Variance is thus crucial in settings like autonomous driving where sample-complexity forms an important bottleneck.

When using $\Psi_t = \sum_{t=0}^{\infty} r_t$ in the policy gradient, *"taking a step with this gradient pushes up the log-probabilities of each action in proportion to $R(\tau)$, the sum of all rewards ever obtained"* [4]. This is not ideal, because in this scenario the quality of an action is partially quantified by what happened before taking that action. Using $\Psi_t = \sum_{t'=t}^{\infty} r_{t'}$ fixes this, because now only the cumulative reward following an action a_t (*reward-to-go*) is considered in the gradient, leading to a lower variance in the expected value for the policy gradient.

Using the temporal-difference residual leads to even lower variance, because Ψ_t is now effectively a measure of how much better the current reward r_t compared to the expected reward under the current policy $V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$. The challenge with using value functions in the choice for Ψ_t is that these value functions have to be learned or estimated, which can be done in several ways. For example, in Monte Carlo value function estimation a regression against the empirical return is applied to a parameterized value function. Policy gradient methods where a value function (or Q-function) is learned are called *actor-critic* methods.

2.2.6 (Soft) Actor-Critic

This final section of the preliminary part on reinforcement learning discusses improvements on policy gradients and how policy optimization can be combined with Q-learning in actor-critic methods. After that, maximum entropy reinforcement learning is introduced, to finally arrive at the *Soft Actor-Critic* (SAC) algorithm. SAC is the reinforcement learning algorithm that will be combined with a self-supervised-contrastive learning objective (see Section 2.3) to attempt robust end-to-end autonomous driving in subsequent chapters.

Actor-Critic Algorithms

A well-known challenge in reinforcement learning is the delay between the taking of an action and the effect of that action on the reward, which is called the *credit assignment problem* [62]. As per *Schulman et al., 2018* [56]: *"Value functions offer an elegant solution to the credit assignment problem – they allow us to estimate the goodness of an action before the delayed reward arrives"*. The family of algorithms that use a value function (value function, Q-function, advantage function) instead of empirical returns as a choice for Ψ_t in Equation 2.12 are called *actor-critic* algorithms. Actor-critic methods are able to *"obtain estimators with lower variance, at the cost of introducing bias"* [56].

The simplest form of an actor-critic method consists of two models:

- the *actor*, which updates the weights θ of the policy network π_θ, according the the policy gradient calculated using the critic.
- the *critic*, which updates the weights ϕ of the Q-function network Q_ϕ and/or the value function network V_ϕ, depending on the specification of the algorithm.

Actor-critic methods are thus an interpolation between Q-learning and policy optimization, where a value function and a policy are learned jointly. Research around actor-critic methods has been plentiful in recent years, leading to key algorithms such as *Asynchronous Advantage Actor-Critic* (A2C/A3C) [45], *Deep Deterministic Policy Gradient* (DDPG) [43], *Twin Delayed DDPG* (TD3) [21], *General Advantage Estimation* (GAE) [56], *Soft Actor-Critic* (SAC) [24] etc.

Maximum Entropy Reinforcement Learning

In traditional reinforcement learning, the goal is to find a policy that maximizes the expected return (see Definition 2.2.6). Maximum entropy RL extends this goal by simultaneously maximizing the expected return and the entropy $\mathcal{H}$ of the policy. In other words, in maximum entropy RL the agent will try *"to succeed at a task while acting as randomly as possible"* [24]. The objective to maximize in maximum entropy RL can be formally written down as

$$J(\pi_\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta,P_\delta}[R(\tau)] + \sum_{t=0}^{T} \mathop{\mathbb{E}}_{(s_t,a_t)\sim\pi_\theta,P_\delta}[\alpha\mathcal{H}(\pi_\theta(\cdot|s_t))] \tag{2.13}$$

$$= \sum_{t=0}^{T} \mathop{\mathbb{E}}_{(s_t,a_t)\sim\pi_\theta,P_\delta}[r(s_t,a_t) + \alpha\mathcal{H}(\pi_\theta(\cdot|s_t))]\,, \tag{2.14}$$

where α is a temperature parameter that controls the stochasticity of the policy by regulating the relative importance of the reward against the entropy term [24] and T is the horizon of a trajectory τ (see Definition 2.2.3). By setting $\alpha = 0$, the original reinforcement learning objective of Equation 2.7 is retrieved. The addition of the entropy maximization term results in more robust stochastic policies [79, 17]

and improves exploration during training by acquiring diverse behaviors [23]. As per *Haarnoja et al., 2018*: *"This objective has a number of conceptual and practical advantages. First, the policy is incentivized to explore more widely, while giving up on clearly unpromising avenues. Second, the policy can capture multiple modes of near optimal behavior. In problem settings where multiple actions seem equally attractive, the policy will commit equal probability mass to those actions"* [24].

Soft Policy Iteration

Soft policy iteration is the maximum entropy variant of policy iteration. Policy iteration is an exact method that can be applied in tabular settings (cf. value iteration as the starting point for Q-learning) that alternates between policy evaluation and policy improvement. In soft policy iteration, the *entropy augmented reward* is defined as

$$r_\pi(s_t, a_t) = r(s_t, a_t) + \mathop{\mathbb{E}}_{s_{t+1}\sim P_\delta}\left[\alpha\mathcal{H}(\pi(\cdot|s_{t+1}))\right]. \tag{2.15}$$

Using $R(\tau) = \sum_{t=0}^{T} r_\pi(s_t, a_t)$ in Equation 2.7, we recover the objective in maximum entropy RL of Equation 2.14. The negative log-likelihood of an action a_t given a state s_t for the stochastic policy π is used as a measure of entropy, defined as

$$\mathcal{H}(\pi(\cdot|s_t)) = -\log \pi(a_t|s_t). \tag{2.16}$$

This formula originates from information theory. A predictable policy with low stochasticity, and thus high likelihoods for specific actions given a state, has less entropy than an unpredictable policy with high stochasticity. Using the augmented reward in the definition of the Bellman equation for the Q-function, we can derive

$$\begin{aligned}
Q^\pi(s_t, a_t) &= \mathop{\mathbb{E}}_{s_{t+1}\sim P_\delta}\left[r_\pi(s_t, a_t) + \gamma \mathop{\mathbb{E}}_{a_{t+1}\sim\pi}\left[Q^\pi(s_{t+1}, a_{t+1})\right]\right] \\
&= \mathop{\mathbb{E}}_{s_{t+1}\sim P_\delta}\left[r_\pi(s_t, a_t)\right] + \gamma \mathop{\mathbb{E}}_{s_{t+1}\sim P_\delta}\left[\mathop{\mathbb{E}}_{a_{t+1}\sim\pi}\left[Q^\pi(s_{t+1}, a_{t+1})\right]\right] \\
&= r(s_t, a_t) + \mathop{\mathbb{E}}_{s_{t+1}\sim P_\delta}\left[\alpha_1\mathcal{H}(\pi(\cdot|s_{t+1}))\right] + \gamma \mathop{\mathbb{E}}_{s_{t+1}\sim P_\delta}\left[\mathop{\mathbb{E}}_{a_{t+1}\sim\pi}\left[Q^\pi(s_{t+1}, a_{t+1})\right]\right] \\
&= r(s_t, a_t) + \mathop{\mathbb{E}}_{s_{t+1}\sim P_\delta}\left[\alpha_1\mathcal{H}(\pi(\cdot|s_{t+1})) + \gamma \mathop{\mathbb{E}}_{a_{t+1}\sim\pi}\left[Q^\pi(s_{t+1}, a_{t+1})\right]\right] \\
&= r(s_t, a_t) + \gamma \mathop{\mathbb{E}}_{s_{t+1}\sim P_\delta}\left[\mathop{\mathbb{E}}_{a_{t+1}\sim\pi}\left[Q^\pi(s_{t+1}, a_{t+1})\right] - \alpha_2 \log \pi(a_{t+1}|s_{t+1})\right]
\end{aligned}$$

where $\alpha_2 = \alpha_1/\gamma$ was used as a scaling to make the equations work. Recall that the policy π is a stochastic policy, meaning that $\log \pi(a_{t+1}|s_{t+1})$ represents the log-likelihood of an action a_{t+1} following a state s_{t+1}. The *policy evaluation* step of soft policy iteration uses an adapted version of the Bellman backup for the Q-function

based on the maximum entropy augmented Bellman equation derived above:

$$Q(s_t, a_t) \leftarrow r(s_t, a_t) + \gamma \mathop{\mathbb{E}}_{s_{t+1} \sim P_\delta} \left[\mathop{\mathbb{E}}_{a_{t+1} \sim \pi} [Q(s_{t+1}, a_{t+1})] - \alpha \log \pi(a_{t+1}|s_{t+1}) \right] \quad (2.17)$$

and iterating over this equation will eventually converge to the soft Q-function of π, as per *Haarnoja et al., 2018* [24].

In the *policy improvement step* of soft policy iteration, the policy is updated toward the exponential of the new soft Q-function. The policy is restricted to a set of policies Π (such as a parameterized family of Gaussian distributions) in order for the policy to remain tractable. This restriction is done by projecting the improved policy onto the set of desired policies using the *Kullback-Leibler (KL) divergence* D_{KL}. The policy is then updated for each state as

$$\pi_{new} = \arg\min_{\pi' \in \Pi} D_{KL} \left(\pi'(\cdot|s_t) \,\middle\|\, \frac{\exp\left(\frac{1}{\alpha} Q^{\pi_{old}}(s_t, \cdot)\right)}{Z^{\pi_{old}}(s_t)} \right), \quad (2.18)$$

which *"is proven to lead to an improved policy in terms of its soft Q-value"*, as per *Haarnoja et al., 2018* [24]. Also, "the partition function $Z^{\pi_{old}}(s_t)$ normalizes the distribution, and while it is intractable in general, it does not contribute to the gradient with respect to the new policy and can thus be ignored" [24]. The alternation between policy evaluation and policy improvement in soft policy iteration is proven to converge to the optimal maximum entropy policy $\pi^* \in \Pi$. This proof can be found in *Haarnoja et al., 2018* [24]. Unfortunately, as with other exact methods, this algorithm can only practically be used in a tabular setting for simple environments. For complex environments in continuous domains, an approximate algorithm based on this soft policy iteration can be formulated: the *Soft Actor-Critic* algorithm.

Soft Actor-Critic

The *Soft Actor-Critic* algorithm blends three key components: an actor-critic architecture that concurrently learns a policy and value function, the use of the maximum entropy principle to increase exploration and stability, and an off-policy formulation allowing for efficient utilization of the collected data leading to higher sample-efficiency [24]. The *Soft Actor-Critic* algorithm is an approximate method based on the soft policy iteration algorithm. SAC uses artificial neural networks to represent the policy and the soft Q-function, and uses stochastic gradient descent to optimize the parameters of these networks instead of performing policy evaluation and improvement until convergence. In order for the policy π_ϕ to be tractable, it is represented as a parameterized Gaussian distribution with mean and covariance given by neural networks, as described in Equation 2.10. The soft Q-function network is trained on the objective function $J_Q(\theta)$ that aims to minimize the soft Bellman residual. This objective is based on Equation 2.17 and incorporates the maximum entropy principle in SAC. The expected values in Equation 2.17 are approximated using stochastic samples taken from a replay buffer $\mathcal{B}$, which incorporates the sample

efficient off-policy learning from experience replay. The objective function for the soft Q-function is now defined as

$$J_Q(\theta) = \mathop{\mathbb{E}}_{(s_t,a_t)\sim\mathcal{B}}\left[\frac{1}{2}\left(Q_\theta(s_t,a_t) - \left(r(s_t,a_t) - \gamma \mathop{\mathbb{E}}_{s_{t+1}\sim P_\delta}[W_{\bar{\theta}}(s_{t+1})]\right)\right)^2\right], \tag{2.19}$$

where

$$W_{\bar{\theta}}(s_{t+1}) = \mathop{\mathbb{E}}_{a_{t+1}\sim\pi_\phi}[Q_{\bar{\theta}}(s_{t+1},a_{t+1})] - \alpha \log \pi_\phi(a_{t+1}|s_{t+1}). \tag{2.20}$$

For the calculation of the residual, a target soft Q-network $Q_{\bar{\theta}}$ is used (cf. DQN in Algorithm 2) instead of the actual soft Q-network with parameters θ. Its parameters $\bar{\theta}$ are updated using an exponentially moving average of θ, with coefficient c_m ($\bar{\theta} \leftarrow c_m\theta + (1-c_m)\bar{\theta}$). This trick is proven to improve stability during training [47, 25].

The weights of the tractable policy π_ϕ are trained on the objective function $J_\pi(\phi)$, minimizing the expected KL-divergence from Equation 2.18. Concretely, $J_\pi(\phi)$ is defined as

$$J_\pi(\phi) = \mathop{\mathbb{E}}_{s_t\sim\mathcal{B}}\left[\mathop{\mathbb{E}}_{a_t\sim\pi_\phi}[\alpha \log(\pi_\phi(a_t|s_t)) - Q_\theta(s_t,a_t)]\right] \tag{2.21}$$

As presented in Equation 2.10, a tractable Gaussian stochastic policy can be represented using a reparametrization trick, where

$$a_t = f_\phi(z_t;s_t) = \mu_\phi(s_t) + \sigma_\phi(s_t) \odot z_t \tag{2.22}$$

can be used as a neural network transformation, with z_t sampled from a spherical Gaussian $\mathcal{N}$. Equation 2.21 can then be rewritten as

$$J_\pi(\phi) = \mathop{\mathbb{E}}_{s_t\sim\mathcal{B},z_t\sim\mathcal{N}}[\alpha \log(\pi_\phi(f_\phi(z_t;s_t)|s_t)) - Q_\theta(s_t,f_\phi(z_t;s_t))] \tag{2.23}$$

where the policy π_ϕ is implicitly defined by f_ϕ. The resulting estimated gradient $\nabla_\phi J_\pi(\phi)$ of this objective function is unbiased and has low variance [24]. The gradients $\nabla_\theta J_Q(\theta)$ and $\nabla_\phi J_\pi(\phi)$ of the objective functions $J_Q(\theta)$ and $J_\pi(\phi)$ can be consulted in *Haarnoja et al., 2018* [24].

Recall from maximum entropy RL the temperature parameter α controls the stochasticity of the policy by regulating the relative importance of the reward against the entropy term. Choosing the right value for α can be a difficult hyperparameter to tune because it depends on the magnitude of the rewards. The magnitude of the reward can on one hand be different across environments or tasks, but also changes during training as the policy improves over time. This problem inspired the authors of the original paper for SAC [24] to come up with an improved implementation where the temperature parameter is automatically adjusted during training in their follow-up paper (*Haarnoja et al., 2019* [25]). The theoretical basis on which this automatic entropy adjustment scheme is based and its derivation are out of the scope of this text but can be consulted in *Haarnoja et al., 2019* [25].

The practical *Soft Actor-Critic* algorithm for episodic environments with observations o_t instead of state measurements s_t is shown in Algorithm 3. This algorithm uses two soft Q-functions instead of one in order to reduce the positive bias in the policy improvement step. This trick, called *double Q-learning*, was first introduced by *van Hasselt, 2010* [68] and *van Hasselt et al., 2015* [69], where it is shown that the positive bias due to overestimation of the Q-values can degrade the performance of value-based methods. The two soft Q-networks Q_{θ_1} and Q_{θ_2} have their own parameters θ_1 and θ_2, respectively. Both Q-networks also have their own target networks $Q_{\bar{\theta}_1}$, $Q_{\bar{\theta}_2}$ for the calculation of the residual in Equation 2.19. The minimum soft Q-value between both Q-networks Q_{θ_1} and Q_{θ_2} is then used in the objective function $J_Q(\theta)$ to avoid value overestimation. For episodic environments, the done signal d_t is incorporated into the objective function $J_Q(\theta)$ to correctly attribute the value of terminal states. Finally, for visual RL problems, the inputs to the models are observations o_t rather than states s_t. A final adapted version of the objective function $J_Q(\theta_i)$ for the soft Q-networks Q_{θ_i} can now be written down as

$$J_Q(\theta_i) = \mathop{\mathbb{E}}_{e\sim\mathcal{B}}\left[\left(Q_{\theta_i}(o_t, a_t) - (r_t + \gamma(1 - d_t)\mathcal{T})\right)^2\right], \tag{2.24}$$

where $e = (o_t, a_t, o_{t+1}, r_t, d_t)$ is an experience tuple sampled from the replay buffer $\mathcal{B}$ and $\mathcal{T}$ is the target, defined as

$$\mathcal{T} = \min_{i=1,2}\left(Q_{\bar{\theta}_i}(o_{t+1}, a_{t+1}) - \alpha \log \pi_\phi(a_{t+1}|o_{t+1})\right) \tag{2.25}$$

Algorithm 3: *Soft Actor-Critic* [25] for visual, episodic environments

$\bar{\theta}_1 \leftarrow \theta_1$, $\bar{\theta}_2 \leftarrow \theta_2$ ▷ Initialize target network parameters
$o_0 \sim \rho_0(\cdot)$ ▷ Reset the environment and obtain initial observation
$d_0 = 0$ ▷ Set initial done signal to false
$\mathcal{B} \leftarrow \emptyset$ ▷ Initialize an empty replay buffer
for *each iteration* **do**
  if $d_t = 1$ **then**
    $o_t \sim \rho_0(\cdot)$ ▷ Reset the environment and obtain initial observation
  $a_t \sim \pi_\phi(a_t|o_t)$ ▷ Sample action from the policy
  $o_{t+1} \sim P_\delta(o_{t+1}|o_t, a_t)$ ▷ Sample transition from the environment
  $r_t = r(s_t, a_t)$ ▷ Get the reward from environment
  $d_t = d(s_t, a_t)$ ▷ Get the done signal from environment
  $\mathcal{B} \leftarrow \mathcal{B} \cup \{(o_t, a_t, r_t, d_t, o_{t+1})\}$ ▷ Store the transition in the replay buffer
  $\theta_i \leftarrow \theta_i - \lambda_Q \nabla_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$ ▷ Update the Q-network parameters
  $\phi \leftarrow \phi - \lambda_\pi \nabla_\phi J_\pi(\phi)$ ▷ Update the policy parameters
  $\alpha \leftarrow \alpha - \lambda \nabla_\alpha J(\alpha)$ ▷ Update the entropy temperature parameter
  $\bar{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta}_i$ for $i \in \{1, 2\}$ ▷ Update target network weights

2.3 Self-supervised Contrastive Representation Learning

The following introduction to self-supervised contrastive representation learning starts with an overview of representation learning and self-supervised learning and concludes with a section on how self-supervised representation learning can be achieved by using contrastive learning. This whole introduction is not an exhaustive study of the field, but rather a short overview of selected topics relevant to the following chapters of this thesis, and focuses mostly on methods and applications using image data.

2.3.1 Representation Learning

Representation learning or feature learning is a machine learning discipline that aims at converting raw data to a set of *representations, features, or embeddings*. Most modern deep learning model architectures make use of an encoder or feature extractor in order to obtain a mathematically and computationally convenient feature space to process inputs. The terms *feature space, embedding space, latent space* are used interchangeably. Representation learning is used across all applications within deep learning: reinforcement learning, natural language processing, computer vision, etc.

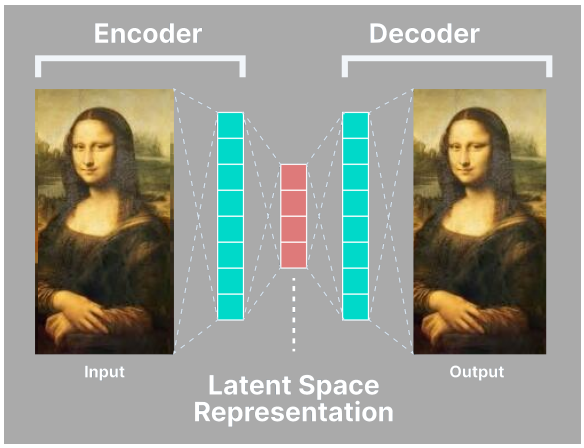


Figure 2.6: General autoencoder architecture. Figure source: *V7Labs* [64].

The autoencoder is perhaps the simplest architecture for representation learning. This model aims to learn an identity mapping between its inputs and outputs and consists of an encoder and a decoder. The place where the encoder and the decoder meet, the *latent space*, is a lower dimensional space that forms a bottleneck. The loss function of an autoencoder aims at minimizing the reconstruction error between input and output. After training an autoencoder on a dataset of images, the encoder should be able to extract lower-dimensional features from the input images into the latent space. The encoder part of the autoencoder can then be used as a feature extractor for other tasks. For example, it can be used as a feature extractor in an

image classification model, as in Figure 2.7. Here, a classifier is trained on top of the embeddings of the feature extractor to classify images of dogs and cats. In this setting the weights (parameters) of the feature extractor network are *frozen*, meaning that no gradient descent steps are performed on these weights. The feature extractor is thus *pre-trained*, but not *fine-tuned* for the classification task. An alternative is to not freeze the weights of the feature extractor and perform gradient descent updates on both the feature extractor and the classifier, which is called *fine-tuning*. Both techniques are forms of *transfer learning*, where knowledge from a related task (here: reconstruction of images through an autoencoder) is applied to another task (here: image classification)

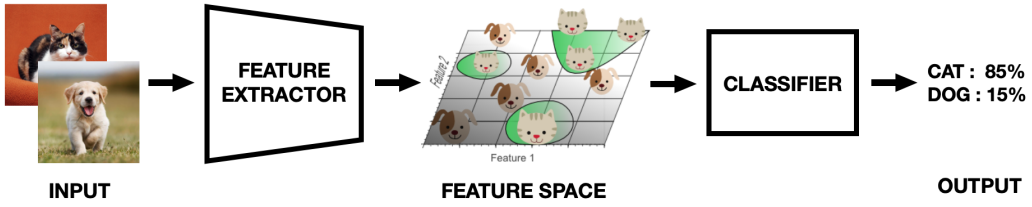


Figure 2.7: Example architecture for image classification making use of a feature extractor followed by a classifier.

2.3.2 Self-supervised Learning

Training an autoencoder as described in the previous section is an example of *unsupervised learning* because the training task only used the raw data without needing any form of supervision like class labels. The image classification example in Figure 2.7 is an example of *supervised learning* because the training of the classifier requires labeled data.

A third form of learning exists in the middle of unsupervised and supervised learning, called *self-supervised learning*. In self-supervised learning, labels are automatically generated for unlabeled data. Self-supervised learning is thus unsupervised because it uses data without human-annotated labels, but also supervised because learning happens in a supervised way using automatically generated labels.

2.3.3 Contrastive Representation Learning

Contrastive representation learning [26, 42, 66, 75, 27] is a deep learning method with the goal of learning an embedding space where inputs of a similar class are close to each other while inputs of dissimilar classes are far from each other, according to some distance metric. Contrastive learning can be applied to labeled data as well as unlabeled data. Contrastive representation learning applied to unlabeled datasets is

one of the most popular and powerful approaches in self-supervised learning.

Definition 2.3.1 (Contrastive Learning [61])**.** *Contrastive learning can be understood as learning a differentiable dictionary lookup task. Given a query or anchor q and keys $\mathbb{K} = \{k_1, k_2, ...\}$ and an explicitly known partition of $\mathbb{K}$ (with respect to q) $P(\mathbb{K}) = (\{k_+\}, \mathbb{K} \setminus \{k_+\})$, the goal of contrastive learning is to ensure that q matches with k_+ relatively more than any of the keys in $\mathbb{K} \setminus \{k_+\}$. Here, q, $\mathbb{K}$, k_+ and $\mathbb{K} \setminus \{k_+\}$ are also referred to as anchor, targets, positive, negatives respectively in the parlance of contrastive learning [66, 27]. Similarities between the anchor and targets are measured by a certain distance metric, for example the dot product ($q^\top k$)) [75, 27], the bilinear product ($q^\top W k$)) [66], the euclidean distance ($\|q - k\|_2^2$) [12] etc.*

Supervised Contrastive Representation Learning

Given a dataset $\mathcal{D} = \{\mathbf{x}_i, y_i\}_{i:1...N}$ of inputs $\mathbf{x}_i$ and corresponding labels y_i, the goal of supervised representation learning is to learn a mapping $f_\theta : \mathcal{X} \to \mathbb{R}^d$ that maps inputs $\mathbf{x}_i \in \mathcal{X}$ to embeddings in a d-dimensional latent space $\mathbb{R}^d$. In supervised contrastive representation learning, this is done by using a loss function that aims to amplify the distance in the latent space ('contrast') between inputs of dissimilar classes while minimizing the distance between inputs of a similar class, according to some metric. This process is visualized in Figure 2.8. An example of a loss function that can be used to achieve the task described above is the contrastive loss [12], defined as

$$\mathcal{L}_{\text{cont}}(\mathbf{x}_i, \mathbf{x}_j, \theta) = \begin{cases} \|f_\theta(\mathbf{x}_i) - f_\theta(\mathbf{x}_j)\|_2^2 & \text{if } y_i = y_j, \\ \max(0, \epsilon - \|f_\theta(\mathbf{x}_i) - f_\theta(\mathbf{x}_j)\|_2^2) & \text{if } y_i \neq y_j, \end{cases} \tag{2.26}$$

where ϵ is a hyperparameter setting the lower bound of the distance between samples of dissimilar classes [72]. Performing gradient descent on this contrastive loss function will thus minimize the square of the Euclidean distance between the latent representations of inputs of similar classes and maximize the square of the Euclidean distance between the latent representations of inputs of dissimilar classes, up to a distance ϵ. Note that many other contrastive loss functions exist[3].

[3] A comprehensive overview can be found in the blog *'Contrastive Representation Learning'* by Lillian Weng (`https://lilianweng.github.io/posts/2021-05-31-contrastive/`)

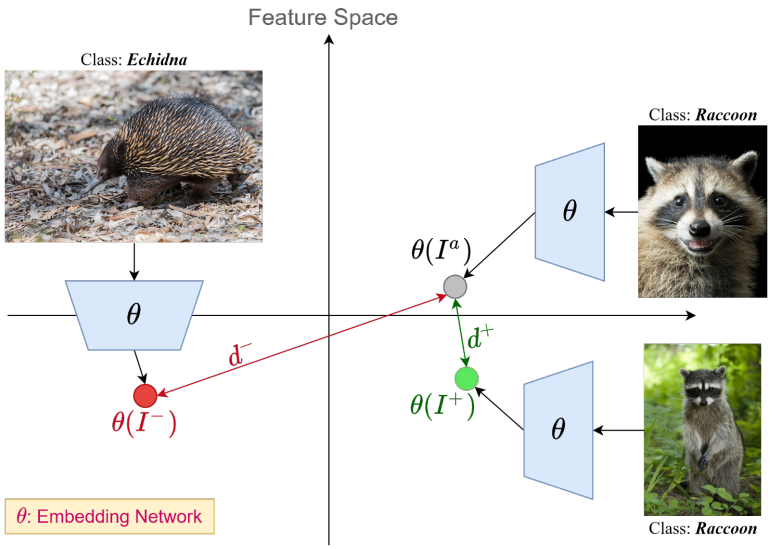


Figure 2.8: Visualization of how contrastive learning works in a supervised setting: the contrastive learning model (represented by θ) tries to minimize the distance d^+ between images of the same class (the anchor I^a and the positive I^+) and maximize the distance d^- between images of different classes (the anchor I^a and the negative I^-) in the feature space. Figure source: *V7Labs* [65]

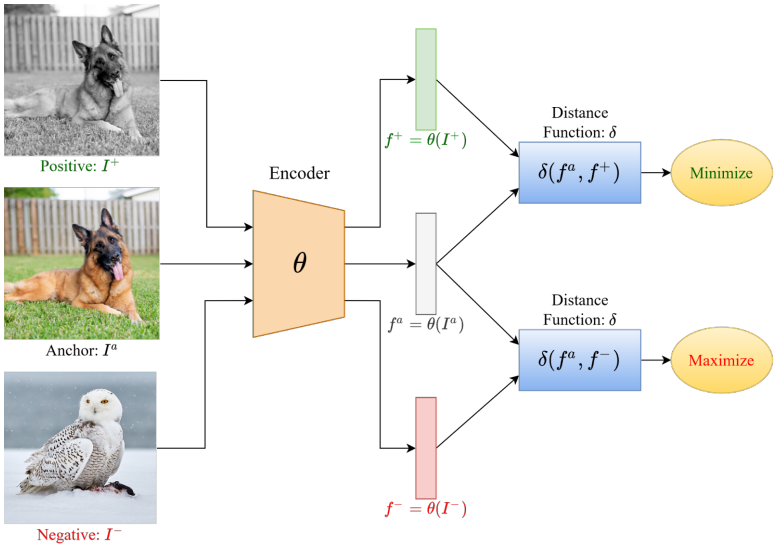


Figure 2.9: Visualization of how contrastive learning works in a self-supervised setting, called *instance discrimination*: the contrastive learning model (represented by θ) tries to minimize the distance $\delta(f^a, f^+)$ between the same image augmented in different ways (the anchor I^a and the positive I^+) and maximize the distance $\delta(f^a, f^-)$ between different images (the anchor I^a and the negative I^-) in the feature space. Figure source: *V7Labs* [65]

Self-supervised Contrastive Representation Learning

In a self-supervised setting, exactly the same techniques as in supervised contrastive representation learning are applied to learn representations from the data, with the only problem being that no class labels are provided along with the raw inputs. In order to surmount this problem, data augmentation is leveraged to create alternate versions of inputs: every input image $\mathbf{x}_i$ in the original dataset is copied and then augmented one or multiple times into alternate version $\mathbf{x}_j$ while keeping track of which augmentations belong to which original input image using a label y_i. The original input images $\mathbf{x}_i$ are the *anchors* or *queries*, the augmented versions of the same original input image are called *positives* and the augmented versions of other original input images are called *negatives*. The collection of positives and negatives are referred to as *targets* or *keys*.

The ultimate goal of self-supervised contrastive learning is to learn robust representations of the inputs against augmentations or perturbations. In general, a heavily augmented or perturbed dataset will result in more robust representations, but in some settings, certain augmentations can lead to wrong representations. For example, when training a reinforcement learning agent in an autonomous driving scenario, one wouldn't want to augment the dataset with horizontally flipped images, because this would lead to a traffic situation that is not equivalent to the original. The specific augmentations to use are thus problem specific and are decided upon using prior knowledge of the problem. Examples of image augmentations include, but are not limited to, horizontal and vertical flipping, random rotations, random shearing, adding noise, random color shifts, etc. Visualizations of some augmentations per category are shown in Figure 2.10. Using the labels y_i created during the augmentation process, the loss function defined in Equation 2.26 (or any other contrastive loss function) can be used in a self-supervised setting. This process is known as *instance discrimination*, and is visualized in Figure 2.9.

Figure 2.10: Overview of common image augmentations, organized by category [51].

Chapter 3

Methodology

This chapter delves into the methodology employed in the attempt towards robust end-to-end autonomous driving in a simulator. This is done by combining a reinforcement learning algorithm with a contrastive learning objective, in line with the background and motivation denoted in Section 1.1. The first section of this chapter discusses the autonomous driving environment used for the training and evaluation of the end-to-end autonomous driving agents. Here, the configuration of the environment and the design of the reward function are discussed. The second section covers the inner workings of the CURL methodology, building on the preliminary knowledge acquired in Chapter 2. After that, the third section discusses data augmentations in visual reinforcement learning and introduces additional image augmentations for CURL used in the experiments. Finally, an overview of the different training experiments is provided in the last section of this chapter. The results of these experiments are then discussed in Chapter 4.

3.1 Custom Autonomous Driving Environment

This section discusses the design of the custom autonomous driving environment on which the reinforcement learning experiments of this thesis are carried out. The environment is programmed on top of the CARLA simulator, which is introduced in the first subsection. This custom reinforcement learning environment should meet several criteria, which include:

- a way to obtain an observation of the environment at every time-step.
- a way to apply a control action to the ego vehicle at every time-step.
- a way to obtain a reward for an action at every time-step (reward function).
- a way to measure when an episode is done.
- a way to reset the environment when an episode is done.

The second subsection of this section provides a detailed overview of the environment configuration and the design of the reward function.

3.1.1 CARLA

CARLA (***Ca**r **L**earning to **A**ct*) is an open-source simulator specifically designed for autonomous driving research. It was introduced alongside the 2017 paper by *Dosovitskiy et al.* [16]. The CARLA simulator is free to use and was purposely built for the training, prototyping, and validation of autonomous driving models. These factors make CARLA the preferred choice for a simulated autonomous driving environment for this thesis, since no free and open-source alternatives of the same quality exist.

Since its creation in 2017, many different updates to the simulator have been released, which all build on top of each other. While experimenting with different versions of CARLA, it was found that the newer releases tend to crash due to memory leaks or other bugs when the simulator is used for a long time (thousands of episodes). For this reason, CARLA version 0.9.8 was used for the experiments conducted in this thesis, because it proved to be the most recent reliable version of the software.

The CARLA software comes in two forms: the simulator itself (server) and a Python API[1] (client) that enables developers and researchers to programmatically control the simulator. The Python API can be used to configure a multitude of aspects of the simulation like the weather, the control action to be applied to the ego vehicle[2], the sensors on the ego vehicle, the behavior of other vehicles, the state of traffic lights, etc. The Python API thus lets a program act on and observe/measure the simulation at every time-step. This makes it possible to create a simulated autonomous driving environment that can interact with a reinforcement learning algorithm.

3.1.2 Environment Implementation and Usage

The custom CARLA environment is implemented as a Python class using the CARLA client-side Python API. It follows the same naming convention for methods as the *Gym interface*[3] from OpenAI, a diverse collection of reference environments used as a benchmark in much of modern reinforcement learning research. The two main methods are the `reset()` method and the `step()` method, which are discussed below.

Environment Initialization

The custom CARLA environment class is initialized with a collection of arguments specifying how the simulator should be set up. Some of these arguments include, but are not limited to, which CARLA town should be used for the simulation, the desired speed of the ego vehicle, the number of other vehicles (called *Non-Player-Character* (NPC) vehicles), the speed at which the ego vehicle is considered to be

[1] Application Programming Interface
[2] In autonomous driving, the ego vehicle refers to the vehicle currently being controlled.
[3] `https://www.gymlibrary.dev`

stalling, the amount of time the ego vehicle can stall before the episode is done, the number of seconds an episode should last, the speed at which the simulation and sensors are rendered in frames-per-second (FPS), the position of the camera sensor on the ego vehicle, the resolution of the images coming from the camera sensor, etc.

The experiments in this thesis are carried out solely in 'Town 4' of the CARLA simulator, which contains a four-lane highway in an infinite loop configuration. This town was chosen because of its simplicity, considering the rather small scale of this thesis (remember that complex autonomous driving tasks have yet to be solved). Figure 3.1 shows the layout of CARLA 'Town 4'.

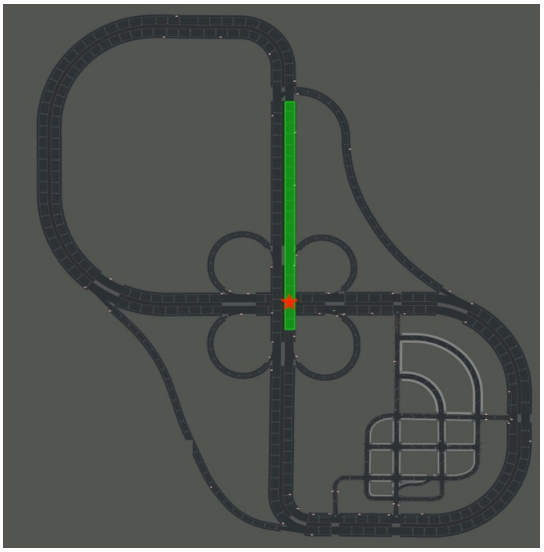


Figure 3.1: The layout of CARLA 'Town 4'. The red star indicates where the ego vehicle is spawned, while the green region shows where the NPC vehicles are spawned.

Resetting the environment

The `reset()` method is used to initialize the environment for the first time or to reset the environment after an episode is done. During this reset, the ego vehicle is randomly spawned in one of the four lanes on the bridge of 'Town 4' (the red star in Figure 3.1), while the NPC vehicles are spawned in random lanes at random distances along the green region in Figure 3.1. The NPC vehicles are then set to autopilot mode (controlled by the *'TrafficManager'* of the CARLA simulator), while the ego vehicle awaits its first control action from the reinforcement learning agent. The weather is chosen randomly from a list of weather presets. Figure 3.2 shows what a typical episode looks like from two perspectives. The `reset()` method also returns the first observation from the camera sensor of the ego vehicle, and thus acts as the start-state distribution function ρ_0, where $o_0 \sim \rho_0(\cdot)$ (see Definition 2.2.1).

(a) Aerial view of the highway

(b) Ego vehicle camera sensor view

Figure 3.2: Views from of a typical episode in CARLA Town 4.

Action Space

The action space of the CARLA environment is continuous and two-dimensional. The two-dimensional action vector a_t contains the throttle/brake input x_t, and the steering input y_t, defined as

$$a_t = \begin{bmatrix} x_t \\ y_t \end{bmatrix} \text{ , with } a_t \in \mathbb{R}^2 \text{ and } x_t, y_t \in [-1, 1].$$

The `step()` method takes in these two-dimensional control action vectors a_t as input, applies the control action to the ego vehicle, and then returns the current observation o_t, reward r_t and done signal d_t. If x_t is positive, the throttle is engaged with magnitude $|x_t|$, and if x_t is negative, the brake is engaged with magnitude $|x_t|$. The sign of y_t determines if the car should steer left or right with magnitude $|y_t|$.

Reward Function

The reward function calculates the current reward r_t as a weighted sum of five different components $r_i \in \mathbb{R}$, with weights $\lambda_{r_i} \in \mathbb{R}$. To calculate the reward, the reward function uses the waypoints of the highway, the position and velocity of the ego vehicle, and a collision sensor that can measure collision impulses of the ego vehicle. The waypoints of the highway are discrete points in the lanes of the highway. When prompted, the CARLA simulator returns the coordinates of the closest waypoint ahead of the ego vehicle's current lane. Using the coordinates of two consecutive waypoints, a vector in the direction of the highway in the current lane can be obtained. The reward r_t at time-step t is calculated as follows:

$$r_t = \lambda_{r_1} r_1 + \lambda_{r_2} r_2 + \lambda_{r_3} r_3 + \lambda_{r_4} r_4 + \lambda_{r_5} r_5 \tag{3.1}$$

$$\text{where} \begin{cases} r_1 = \left(\mathbf{v}_{ego,t}^{\top} \cdot \hat{\mathbf{u}}_{highway,t}\right) \Delta t & \textit{highway progression reward} \\ r_2 = -\min(1, d_{c,t}^3) & \textit{center of lane deviation penalty} \\ r_3 = -|y_t| & \textit{steering penalty} \\ r_4 = -\|\mathbf{i}_t\|_2 & \textit{collision penalty} \\ r_5 = -\max(0, d_{v,t}\Delta t + \Delta t) & \textit{speeding penalty} \end{cases} \tag{3.2}$$

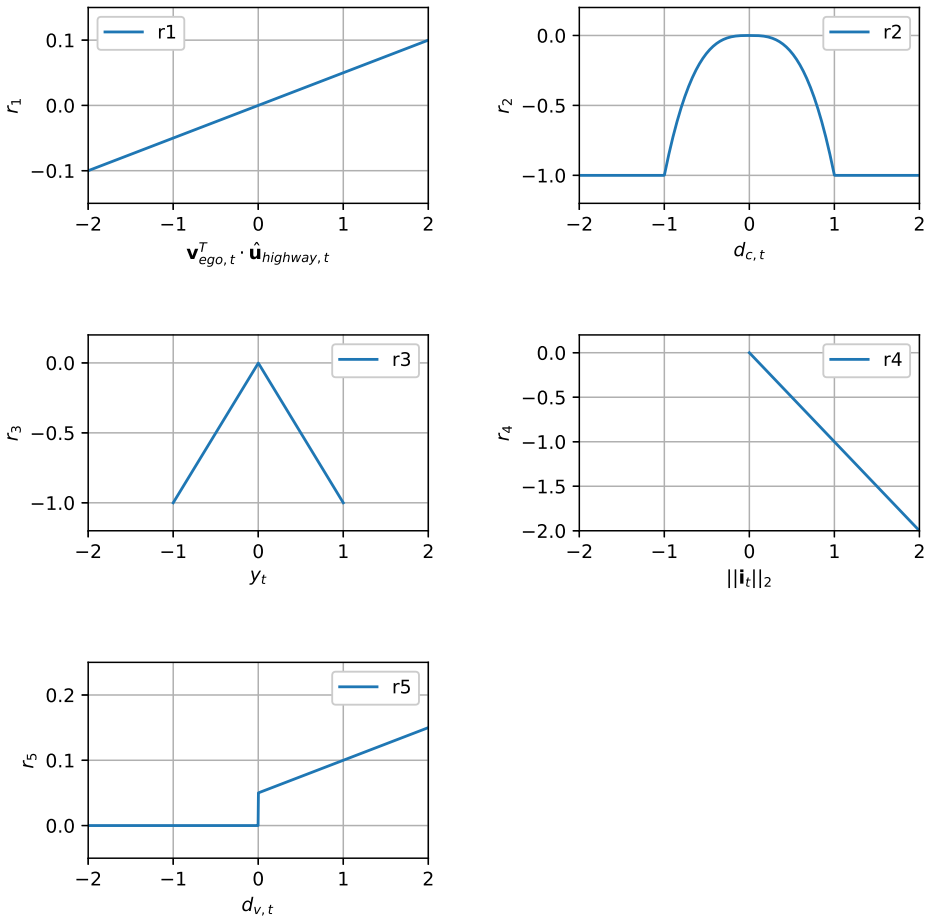


Figure 3.3: Plots of the different components that make up the reward function in Equation 3.2 for $\Delta t = 0.05$ in the range $[-2, 2]$ for visualization purposes.

The weights λ_{r_i} can be used to tune the reward function by giving more or less importance to the individual components r_i. Figure 3.3 shows the shape of the different terms of the reward function, which are discussed in more detail below.

- The first term of the reward, $r_1 \in (-\infty, +\infty)$, represents the progression (in meters) along the direction of the highway during the current time-step. Here, $\mathbf{v}_{ego,t}$ is the two-dimensional velocity vector of the ego vehicle at the current time-step, $\hat{\mathbf{u}}_{highway,t}$ is a unit vector in the direction of the highway (calculated using the highway waypoints) at the current time-step, and Δt is the time between two discrete time-steps (the inverse of the FPS of the simulation). This reward incentivizes the agent to make progress along the highway and to keep its heading aligned with the direction of the highway.
- The second term of the reward, $r_2 \in [-1, 0]$, is a penalty for deviating from the center of the lane, where $d_{c,t}$ is the perpendicular distance (in meters) between the position of the ego vehicle and the center of the closest lane at the current

time-step. This penalty incentivizes the agent to keep the ego vehicle centered in its lane. The absolute value of the penalty is capped at a maximum value of 1 to avoid extreme penalties when the ego vehicle is learning to drive.

- The third term of the reward, $r_3 \in [-1, 0]$, is a penalty for the absolute value of the applied steering angle, motivating the agent to apply as few steering commands as possible to learn to drive smoothly. The fact that $y_t \in [-1, 1]$ leads to $r_3 \in [-1, 0]$.

- The fourth term of the reward, $r_4 \in (-\infty, 0]$, is similar to r_3, but for collisions. It is a penalty for the L_2-norm of the three-dimensional collision impulse vector $\mathbf{i}_t$. The collision impulses are measured by the collision sensor of the ego vehicle, measured in Newton-seconds. This penalty incentivizes the agent to avoid colliding with objects and other vehicles.

- The fifth and final term of the reward, $r_5 \in [0, +\infty)$, is a penalty for going over the desired speed, where $d_{v,t} = \|\mathbf{v}_{ego,t}\|_2 - \|\mathbf{v}_{desired}\|_2$. This penalty is meant to undo the advantage gained in the r_1 term when speeding. The addition of the extra term Δt to the product $d_{v,t}\Delta t$ serves as an extra penalty, meaning that the penalty for speeding (r_5) will be greater than the advantage gained by speeding in r_1.

Note that the design of the reward function was partly inspired by the autonomous driving reward function used in the *'Learning Invariant Representations for Reinforcement Learning without Reconstruction'* paper by *Zhang et al., 2021* [77].

Done Signal

The done signal indicates when an episode is finished and the environment should be reset. The done signal is set to false by default ($d_t = 0$), but is set to true ($d_t = 1$) if the ego vehicle has reached a terminal state. A terminal state can be achieved in three ways:

1. the ego vehicle has been stalling for more than the allowed stall time.
2. the ego vehicle has collided with an object or vehicle.
3. the maximum time per episode has been reached.

3.2 Contrastive Unsupervised Representations for Reinforcement Learning

In this section, the *'**C**ontrastive **U**nsupervised Representations for **R**einforcement **L**earning'* (CURL) method for sample-efficient and robust model-free reinforcement learning is explained in detail. The method can be applied in discrete and continuous action spaces, using the *Rainbow* DQN or SAC algorithm, respectively. The discrete action space version is not discussed, because this thesis is solely concerned with continuous action spaces.

As discussed before, self-supervised contrastive learning can be leveraged in a reinforcement learning setting to learn useful semantic representations from high-dimensional observations. Reinforcement learning agents trained on top of these representations should then be significantly more sample-efficient [76, 44, 61]. The CURL method proposes a framework in which contrastive learning is used to improve an agent's learning ability from online interactions. This means that no prior dataset is used and that the encoder is not pre-trained in advance.

CURL is not the first paper to apply contrastive learning techniques in a model-free visual reinforcement learning setting. In 2018 *van den Oord et al.* attempted this using *Contrastive Predictive Coding* (CPC) [66], but the results were indecisive with marginal gains on some benchmarks [66, 61]. CURL is the first method using contrastive learning in a model-free reinforcement learning setting that leads to substantial sample-efficiency improvements across various benchmarks. It was shown that *"CURL coupled with the Soft-Actor-Critic (SAC) [...] matches the performance of state-based SAC on the majority of 16 [benchmark] environments tested, a first for pixel-based methods"* [61]. These results prove that the CURL method is able to extract state information from raw pixels in benchmark environments. The two following subsections discuss the two main parts of the CURL method: the contrastive learning objective and the *Soft Actor-Critic* algorithm.

3.2.1 Contrastive Learning Objective

In a reinforcement learning setting, no labels are provided alongside observations, eliminating the possibility to perform contrastive learning in a supervised way. Therefore, self-supervised contrastive learning is used, where data augmentations are leveraged to learn representations from unlabeled data in a process called instance discrimination, as explained in Section 2.3.3. In CURL, a batch of B observations is sampled from a replay buffer, augmented, encoded, and then finally the representations are compared using a distance metric or similarity measure. The performance of the encoder is then quantified by a contrastive loss function or objective, on which gradient descent can be performed to optimize the parameters of this encoder. The contrastive objective is used as an *auxiliary loss* during the batch update of the SAC algorithm [61] to improve the representation learning used to extract approximate state information from high dimensional observations. The implementation of these

different parts is discussed below, and an overview of the CURL architecture is shown in Figure 3.4.

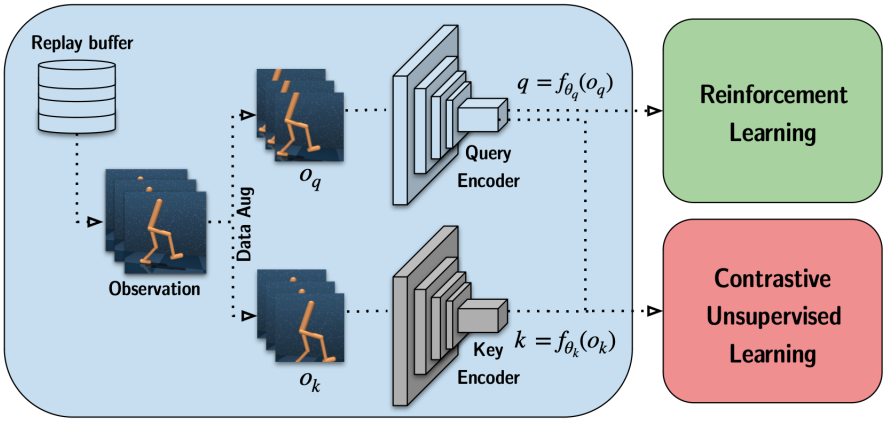


Figure 3.4: Overview of the CURL architecture. Figure source: *Srinivas et al., 2020* [61].

Query & Key Generation

The majority of deep reinforcement learning algorithms use a stack of temporally consecutive frames in a so-called frame-stack as an observation instead of only a single frame [31] (cf. Definition 2.2.10). This is done to incorporate temporal information into the observations, leading to better extraction of state information. Therefore, instance discrimination is performed across frame-stacks. A query o_q and positive key o_{k_+} in Figure 3.4 are data augmentations of the same observation, while a negative key o_{k_-} is a data augmentation of a different observation sampled from the replay buffer. Note that the terms query and anchor are synonyms and that the terms target and key are also synonyms. The data augmentations are defined as $\tau : \mathcal{O} \to \mathcal{O}$, where $\mathcal{O}$ is a set of frame-stacked image tensors. An augmentation τ may or may not preserve the resolution (size) of the original observation.

The original CURL paper uses random cropping with a cropping ratio $r = 0.84$ as the sole augmentation. In random cropping, an image with width w and height h gets cropped at a random location to width $w' = rw$ and height $h' = rh$. This process is depicted in the left part of Figure 3.4, where only a query and a positive key are shown (the random crops come from the same frame-stacked observation). The purpose of this augmentation is to teach the agents to extract spatial information from observations. The experiments conducted in this thesis extend the CURL method with other augmentations in Section 3.3, whose effects on performance and robustness are empirically compared in Chapter 4.

Target Encoding with Momentum

The queries and keys generated by augmenting observations sampled from the replay buffer are encoded into *latent* queries q by the query encoder network f_{θ_q} as $q = f_{\theta_q}(o_q)$ and *latent* keys k by the key encoder network f_{θ_k} as $k = f_{\theta_k}(o_k)$, respectively. This process is shown in the center of Figure 3.4. The latent representations are vectors $q, k \in \mathbb{R}^m$ of approximate state information, where m dictates the dimension of the approximate state-space. Both encoder networks have the exact same architecture and are made up of convolutional layers to extract features from the input observations and fully connected layers to bring these features down to the desired latent space dimension m. The parameters of the key encoder network θ_k are a momentum encoded version of the parameters of the query encoder network θ_q, as presented in the *MoCo* method by *He et al, 2019* [27]. Specifically, the key encoder network parameters θ_k are updated as

$$\theta_k = c_m\theta_k + (1 - c_m)\theta_q, \tag{3.3}$$

where $c_m \in [0, 1)$ is a momentum coefficient that essentially determines how much the parameters θ_k lag behind the parameters θ_q. As per *He et al., 2019* [27], *"A naive solution is to copy the key encoder f_{θ_k} from the query encoder f_{θ_q} [...]. But this solution yields poor results in experiments [...]. We hypothesize that such failure is caused by the rapidly changing encoder that reduces the key representations' consistency."*. The momentum update is thus introduced for reasons of stability.

Contrastive Loss Function

The last step in the contrastive learning objective is the choice of a contrastive loss in function of the latent queries q and keys k, with respect to some similarity measure between q and k (see Definition 2.3.1). The CURL method uses the bi-linear inner product ($q^\top Wk$) as a similarity measure (where $W \in \mathbb{R}^{m\times m}$ is a learnable matrix), which was found to lead to higher performance over the cosine similarity [61]. This similarity measure is then used in the InfoNCE loss function proposed by *van den Oord et al., 2018* [66], where NCE stands for Noise-Contrastive Estimation. The InfoNCE loss on a set of N keys k_i (one positive k_+ and $N-1$ negatives k_-) and a query q *"can be interpreted as the log-loss of a N-way softmax classifier whose label is k_+"* [61], and is defined as

$$\mathcal{L}_{\text{InfoNCE}} = \log\left(\frac{\exp\left(q^\top Wk_+\right)}{\exp(q^\top Wk_+) + \sum_{i=0}^{N-1}\exp(q^\top Wk_i)}\right). \tag{3.4}$$

In Figure 3.4, this loss function is represented by the red block in the bottom-right corner. Note that this loss function is only used to adapt the parameters of the encoder networks f_{θ_q}, f_{θ_k}.

In the CURL method, the contrastive objective is performed across a batch of observations sampled from the replay buffer. The same technique is used in the SAC

algorithm (and off-policy RL algorithms in general) to perform a parameter update on a batch. This makes the auxiliary contrastive task seamless to integrate with the SAC algorithm. First, a batch of B observations $O \in \mathbb{R}^{B \times w \times h \times c \times f_s}$ is randomly sampled from the replay buffer, where w and h represent the width and height of the observation images, c is the number of color channels of the images ($c = 3$ for color images) and f_c is the number of stacked frames in the frame-stack. This batch of observations then gets transformed to a batch of queries $O_q \in \mathbb{R}^{B \times w' \times h' \times c \times f_s}$ and to a batch of keys $O_k \in \mathbb{R}^{B \times w' \times h' \times c \times f_s}$ by the augmentation τ_t, where w' and h' are the possibly reduced width and height of the augmented observation images:

$$O = \begin{bmatrix} o_0 \\ o_1 \\ \vdots \\ o_{B-1} \end{bmatrix}, \quad O_q = \tau_t(O) = \begin{bmatrix} o_{q,0} \\ o_{q,1} \\ \vdots \\ o_{q,B-1} \end{bmatrix}, \quad O_k = \tau_t(O) = \begin{bmatrix} o_{k,0} \\ o_{k,1} \\ \vdots \\ o_{k,B-1} \end{bmatrix} \tag{3.5}$$

After that, the batches of queries O_q and keys O_k get encoded by their respective encoders f_{θ_q} and f_{θ_k} into batches of latent queries $Q \in \mathbb{R}^{B \times m}$ and keys $K \in \mathbb{R}^{B \times m}$:

$$Q = f_{\theta_q}(O_q) = \begin{bmatrix} q_0 \\ q_1 \\ \vdots \\ q_{B-1} \end{bmatrix}, \quad K = f_{\theta_k}(O_k) = \begin{bmatrix} k_0 \\ k_1 \\ \vdots \\ k_{B-1} \end{bmatrix} \tag{3.6}$$

Finally, a similarity matrix $S \in \mathbb{R}^{B \times B}$ is constructed by applying the bi-linear inner product with $W \in \mathbb{R}^{m \times m}$ on the batch:

$$S = Q(WK^\top) = \begin{bmatrix} q_0^\top W k_0 & q_0^\top W k_1 & \dots & q_0^\top W k_{B-1} \\ q_1^\top W k_0 & q_1^\top W k_1 & \dots & q_1^\top W k_{B-1} \\ \vdots & \vdots & \ddots & \vdots \\ q_{B-1}^\top W k_0 & q_{B-1}^\top W k_1 & \dots & q_{B-1}^\top W k_{B-1} \end{bmatrix} \tag{3.7}$$

The batched version of the InfoNCE loss in Equation 3.4 can then be calculated as the cross-entropy loss [78] between the prediction S and labels $I_{B \times B}$ (the identity matrix). This is because the similarities between queries and their positive keys are found on the diagonal of the similarity matrix S, while the similarities between queries and negative keys fill up non-diagonal entries in S.

3.2.2 Soft Actor-Critic

The authors of the CURL paper have selected the SAC algorithm for continuous action spaces due to reasons of sample-efficiency and stability. Off-policy algorithms such as DDPG [43], TD3 [21] and SAC perform more gradient updates per data sample collected, and are thus preferred over state-of-the-art on-policy algorithms such as *Proximal Policy Optimization* (PPO) [57], *Trust Region Policy Optimization* (TRPO) [55] and *Asynchronous Advantage Actor-Critic* (A3C) [45] for their sample-efficiency. Between the state-of-the-art off-policy algorithms, SAC is preferred over

DDPG and TD3 because the latter algorithms are found to often be unstable due to their brittleness w.r.t. hyperparameters. SAC suffers less from this issue because of its maximum entropy formulation, making it more stable than other algorithms [24, 25, 61].

The SAC algorithm has been discussed at length in Chapter 2, and its implementation for episodic environments in visual reinforcement learning can be found in Algorithm 3. The main difference with the implementation in Algorithm 3 is that in CURL, the policy and the two soft Q-functions share the same encoder network f_{θ_q}, as shown in Figure 3.5. The actual network of the policy π_ψ is an MLP network and takes m-dimensional latent representation vectors of observations as input and outputs two-dimensional control action vectors. The two soft Q-networks Q_{ϕ_1}, Q_{ϕ_2} (critic networks) are also MLP networks and take m-dimensional latent representation vectors and two-dimensional control actions as inputs, and output a one-dimensional soft Q-value. The actor π_ψ and critics Q_{ϕ_1}, Q_{ϕ_2} are thus trained on top of latent representations q.

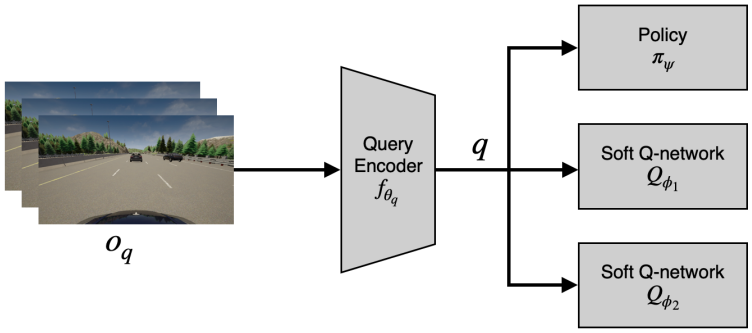


Figure 3.5: Visualization of how the actor (policy π_ψ) and critic (soft Q-networks Q_{ϕ_1}, Q_{ϕ_2}) share the feature extraction encoder network f_{θ_q} in the CURL architecture.

The policy network π_ψ is completely detached from the encoder network f_{θ_q} and the gradients calculated with the loss function of the policy network have no effect on the parameters of the encoder network. The encoder network and critic networks are not decoupled during the optimization step of the SAC algorithm, however. After the calculation of the loss function of the critic, the obtained gradients are backpropagated through both the critic networks and the encoder network. The loss function of the soft Q-networks thus has an effect on the parameters of the encoder network. Note that the contrastive learning objective mentioned earlier only affects the parameters of the encoder network f_{θ_q}, and does not affect the RL agent networks Q_{ϕ_1}, Q_{ϕ_2} and π_ψ.

One last important concept to note is the difference between the augmentation used for sampling actions from the agent, the *evaluation augmentation* τ_e, and the augmentation used when updating the parameters of the different networks, the *training augmentation* τ_t. The evaluation augmentation is used to augment

observations when sampling an action from the policy for evaluation of the agent or when collecting data during training, as in line 8 of Algorithm 3. In CURL, this line becomes $a_t \sim \pi_\psi(a_t|f_{\theta_q}(\tau_e(o_t)))$. The training augmentation is used on observations sampled from the replay buffer in the SAC losses $J_Q(\phi_i)$, $J_\pi(\psi)$ to generate inputs $q = f_{\theta_q}(\tau_t(o_t))$ to the policy and Q-networks ($o_q = \tau_t(o_t)$ in Figure 3.5), and in the contrastive loss $\mathcal{L}_{\text{InfoNCE}}$ to generate queries $q = f_{\theta_q}(\tau_t(o_t))$ and keys $k = f_{\theta_k}(\tau_t(o_t))$. The observations passed to the SAC algorithm are thus also augmented because empirical evidence showed that this improved performance on most benchmarks [61]. The reason why the evaluation augmentation is different is because it is not a random augmentation like the training augmentation, but rather a consistent one, making the inputs to the policy more stable, which makes it easier to select an appropriate action. In the CURL paper, center cropping is used instead of random cropping as the evaluation augmentation.

3.3 Data Augmentations in Visual Reinforcement Learning

The challenge of sample inefficiency in visual reinforcement learning can be alleviated by training with auxiliary tasks such as instance discrimination, among other methods such as learning world models and leveraging pre-training encoders from other domains such as ImageNet [44]. As per *Ma et al., 2022*, [44] *"These methods can significantly improve the sample efficiency of visual reinforcement learning, but the lack of (diverse) training data remains a fundamental issue, which can be effectively solved by data augmentation."*. An important property to keep in mind in this context is *optimality-invariance*. A data augmentation $\tau : \mathcal{O} \to \mathcal{O}$ is optimality-invariant if its mapping from raw observations to augmented observations preserves the values, Q-values, and policy [44]:

$$Q(o, a) = Q(\tau(o), a) \quad V(o) = V(\tau(o)) \quad \pi(a|o) = \pi(a|\tau(o)) \quad \forall o \in \mathcal{O}, a \in \mathcal{A} \tag{3.8}$$

For example, in a setting where an agent is trained to play tennis on grass courts from observations, using data augmentation to change the colors of the observations would not affect the game (and thus the policy and (Q-)values) whatsoever. The random-cropping augmentation used in CURL satisfies the optimality invariance assumption in most benchmark environments, such as in the DeepMind control suite shown in Figure 3.6.

Additionally, learning to extract robust state information in a latent space from high-dimensional observations is not trivial. RL agents can often fail to ignore task-irrelevant information and/or effectively distinguish it from task-relevant information. This can result in RL agents overfitting to the environment they were trained in, leading to bad generalization on novel scenarios. Traditional regularization techniques borrowed from supervised learning such as ℓ_2-regularization, dropout, and batch normalization do not translate well to reinforcement learning settings [44]. Learning robust representations has proven to be much more effective in that regard, where

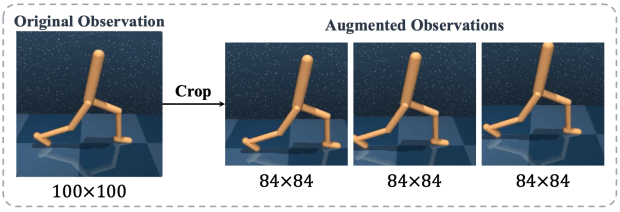


Figure 3.6: Visualization of the optimality-invariance of the random crop augmentation in the DeepMind 'walker' benchmark environment. Figure source: *Ma et al., 2022* [44].

data augmentation is leveraged to generate diverse synthetic data. On top of this, *"data augmentation can implicitly provide prior knowledge to the agent as a type of inductive bias or regularization."* [44]. This important concept of *prior-based diversity* effectively means that we can leverage prior knowledge about the task at hand to select data augmentations that will maximize an agent's robustness to task-irrelevant information contained within observations. For example, in a setting where an agent is trained to play tennis on grass courts from observations, using data augmentation to change the colors of the observations could help the agent to generalize to other surfaces such as clay courts. Another example is shown in Figure 3.7.

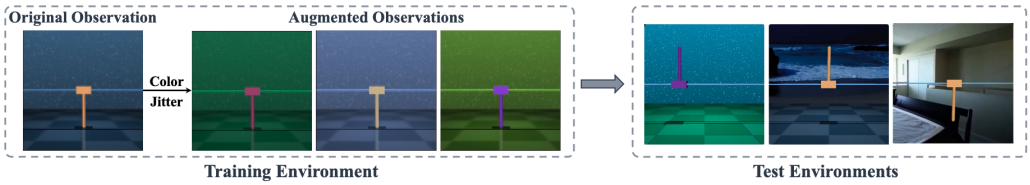


Figure 3.7: Examples of applying data augmentation under the assumption of prior-based diversity in the DeepMind 'cartpole' benchmark environment. Figure source: *Ma et al., 2022* [44].

With the above theory in mind, three additional augmentations (next to random cropping) were introduced for the end-to-end autonomous driving experiments conducted in this thesis. The effects of these augmentations are visualized in Figure 3.8 for both the training and evaluation settings τ_t and τ_e.

The first additional data augmentation, called *identity*, is just an identity mapping in both the training and evaluation setting τ_t and τ_e. This augmentation is introduced to test whether the contrastive learning objective in CURL can have an effect on performance and robustness on its own without any augmentation. This configuration of the CURL method will also serve as a baseline for configurations with actual augmentations.

The second additional augmentation, called *color jiggle*, applies a random transformation to the brightness, contrast, saturation, and hue of observations in the training setting τ_t, and leaves observations unchanged in the evaluation setting τ_e. This augmentation would normally not be optimality-invariant in an autonomous driving context, because it could change the color of traffic lights and other color-dependent signals. In the simple highway environment utilized in the experiments of this thesis, however, such factors are not a concern. This augmentation is specifically targeted at making agents robust against different weather types, car colors, etc.

The third additional augmentation, called *noisy cover*, covers task-irrelevant parts and adds Gaussian noise to observations in the training setting τ_t, and leaves observations unchanged in the evaluation setting τ_e. More specifically, the sky at the top and the hood of the ego vehicle at the bottom of observations get covered by rectangles of a random color by τ_t. Next to this, Gaussian noise gets added to the entire observation by τ_t. As with the previous augmentation, this augmentation would normally not be optimality-invariant in an autonomous driving context but is for the simplified highway driving task. This augmentation is specifically designed to teach the agent to ignore task-irrelevant parts of the observation images such as clouds and the sun.

3.4 Experiments

A total of six experiments are carried out in order to answer the research questions asked in the introductory chapter of this thesis:

- *Does the CURL methodology improve an agent's performance and generalizability/robustness over plain SAC?*
- *What is the influence of data augmentations on the performance and robustness of CURL agents? Which data augmentations lead to better results, and why?*

The experiments consist of training RL agents using different configurations of CURL and SAC in the custom autonomous driving environment for one million environment steps. These experiments are performed on GPU clusters provided by the partnership between KU Leuven and the Flemish Supercomputer Center[4]. Each experiment took between 48 and 120 hours to complete, depending on the GPU and the chosen data augmentation. At a frame rate of 20 FPS, one million environment steps equate to almost 14 hours of simulated driving experience for the RL agents. The fact that training for 14 hours of simulated driving requires 48 to 120 hours of wall-clock time to perform on state-of-the-art hardware[5] once again underlines the importance of sample-efficiency of RL algorithms used in visual RL.

[4] `https://www.vscentrum.be`

[5] GPUs used: NVIDIA P100, V100, and A100.

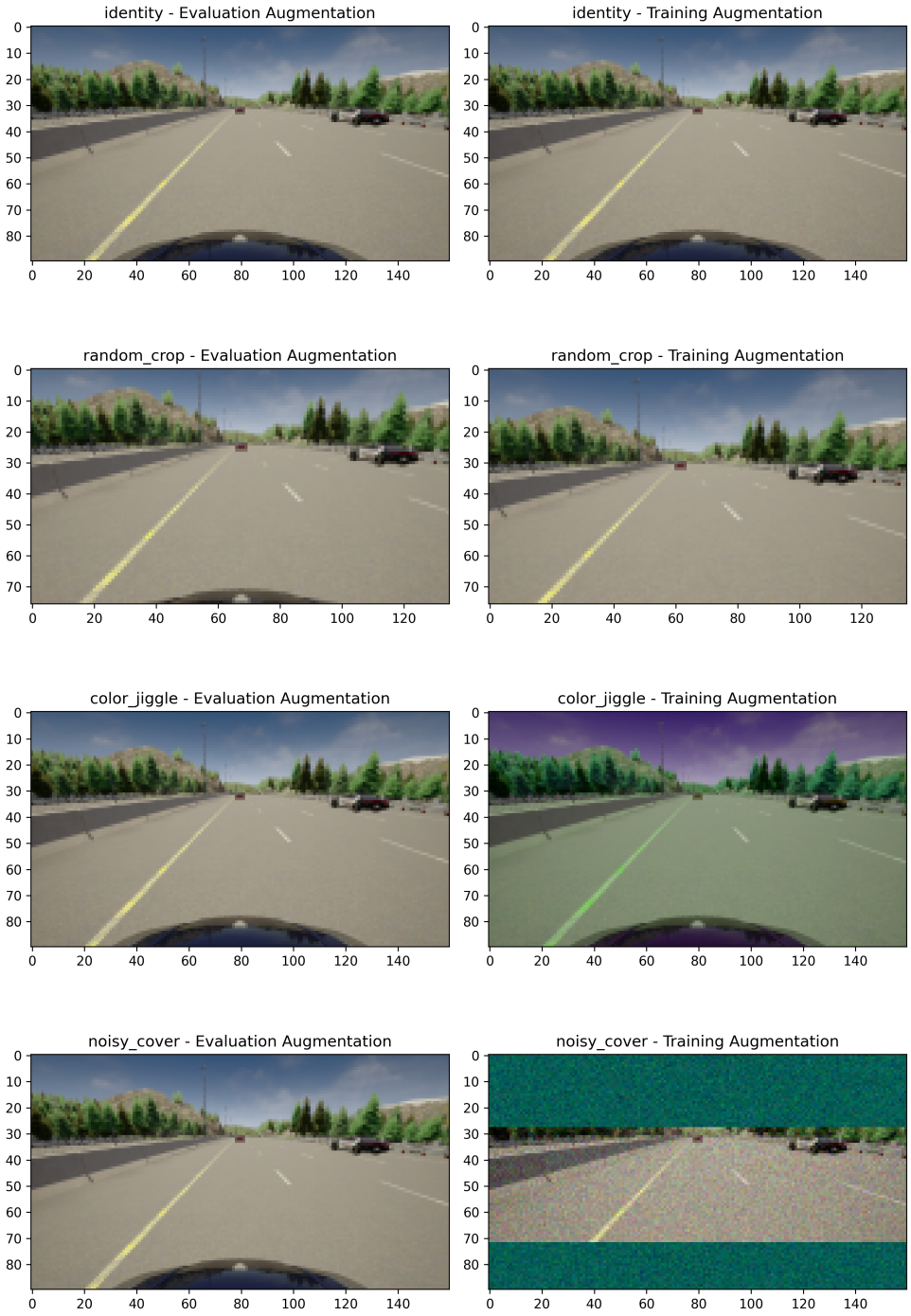


Figure 3.8: Visualization of the different data augmentations used in this thesis on 160×90 observations. *Rows from top to bottom:* identity, random crop, color jiggle, noisy cover. *Left column:* evaluation setting τ_e. *Right-column:* training setting τ_t.

The initial distribution of the parameters of the multiple neural networks, the sequence of chosen weather presets in the simulator, the position of NPC vehicles on the road, etc. are dictated by the random seed of the experiment. In order to mitigate the effect of the random seed on the experiments, each experiment is repeated three times with a randomly chosen random seed value $s \in [1, 10^6]$. Ideally, each experiment would be repeated 10 times or more to get a more accurate metric of an experiment's performance, but this amount of training was unfeasible due to the large computational budget it would require.

Experiment name	**Experiment description**
Pixel SAC	SAC algorithm using frame-stacked observations
CURL identity	CURL with the identity augmentation
CURL random crop	CURL with the random crop augmentation
CURL color jiggle	CURL with the color jiggle augmentation
CURL noisy cover	CURL with the noisy cover augmentation
CURL identity detached	CURL with the identity augmentation, but RL objective does not affect the parameters of the encoder

Table 3.1: Overview of the experiment names and their description.

An overview of the experiments and their description is provided in Table 3.1. The first experiment, *Pixel SAC*, trains the RL agent using only the SAC algorithm without the contrastive learning objective. No augmentation is used to augment the frame-stacked observation in the SAC algorithm either. This experiment serves as a baseline for the CURL experiments. Then four CURL experiments are carried out using the four augmentations defined earlier in this chapter: *identity*, *random crop*, *color jiggle*, and *noisy cover*. The final experiment is the same experiment as *CURL identity* but with the encoder decoupled from the RL objective. Recall from Section 3.2.2 that the parameters of the encoder networks get optimized by both the contrastive loss function and the Q-network loss function from the SAC algorithm. In the *CURL identity detached* experiment, this is not the case because the parameters of the encoder are frozen during the SAC batch update.

Apart from the differences mentioned in Table 3.1, each experiment uses exactly the same set of hyperparameters. The chosen hyperparameters organized by category can be found in Appendix B, Table B.1. The majority of these hyperparameter values were configured to match the settings employed in the CURL paper, as they had already undergone thorough tuning. Other hyperparameters specific to the autonomous driving environment, such as the weights λ_{r_i} of the reward function components, were empirically determined on smaller-scale experiments. Hyperparameters such as the replay buffer size and the batch size have a large impact on the

memory requirement and wall-clock time of the experiments. These hyperparameters were chosen as large as possible within the range of feasibility given by the hardware on the GPU clusters because a large batch size generally improves the contrastive learning objective [11], while a large replay buffer size improves the performance of the RL agent [18].

The CARLA simulator offers a total of 14 different weather presets to choose from at the start of an episode. For training, a total of seven weather presets are used. The other available weather presets are not used during training in order to test the robustness of the trained agents on novel weather scenarios in Chapter 4. Table 3.2 shows which weather presets are used for training and which are kept for testing. Every 25000 steps, the training is halted to perform 10 evaluation episodes under the current policy (using the training weather presets). The mean undiscounted episode return on these 10 evaluation episodes is then tracked as an evaluation metric.

Training weather presets	**Novel weather presets**
ClearNoon	*MidRainyNoon*
ClearSunset	*WetCloudyNoon*
CloudyNoon	*WetCloudySunset*
CloudySunset	*SoftRainNoon*
WetNoon	*SoftRainSunset*
WetSunset	*HardRainNoon*
MidRainSunset	*HardRainSunset*

Table 3.2: Division of the weather presets for training and testing on novel scenarios.

Chapter 4

Results

This chapter discusses the results obtained from the experiments presented in Section 3.4. In accordance with the research goals laid out in Chapter 1, the trained RL agents are assessed and compared based on their performance and robustness on the end-to-end autonomous driving task. The first section of this chapter discusses the agents' training and evaluation performance on the weather presets used for training. The performance on the training weather presets allows us to measure the relative sample-efficiency of agents per experiment setup. The robustness or ability to generalize to different scenarios is assessed in the second section of this chapter, which consists of two parts. First, the performance of the agents on the training weather presets is compared to their performance on the novel weather presets to measure their generalizability. Secondly, the latent space representations learned by the trained agents are visualized and studied.

4.1 Performance

The performance of each experiment during training over the course of one million environment steps is shown in Figures 4.1, 4.2 and 4.3. These figures show the training episode undiscounted return[1], the average batch reward and the mean evaluation episode undiscounted return, respectively, averaged over three random seeds. The data in Figures 4.1 and 4.2 was smoothed using moving average smoothing with a window size of 100 datapoints, while the data in Figure 4.3 was smoothed using moving average smoothing with a window size of 10 datapoints. The reason for the different window size is because the data in Figures 4.1 and 4.2 was logged at least once every 1000 environment steps, while the data in Figure 4.3 was only logged every 25000 environment steps.

CURL Improves Performance and Sample-Efficiency

Figures 4.1 - 4.3 show that the performance on the end-to-end autonomous driving task of the different experiments can be ranked from best to worst as: *CURL color*

[1]The terms *undiscounted return* and *reward* were introduced in Definition 2.2.5

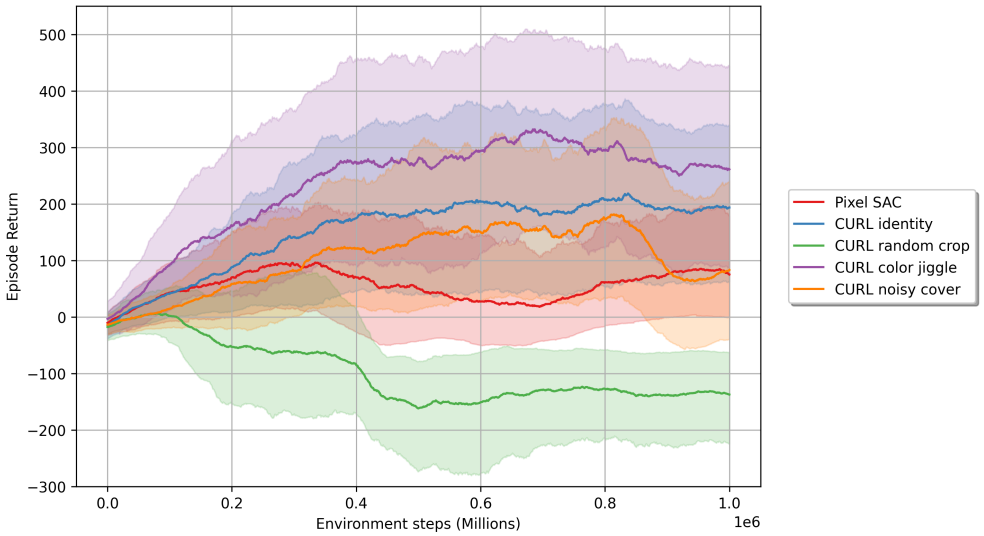


Figure 4.1: Smoothed undiscounted return of the training episodes per experiment during training. Each thickened line represents the smoothed performance averaged over three random seeds, while the slightly transparent areas represent the smoothed maximum-minimum range across the three random seeds. The undiscounted episode return was logged whenever a training episode ended.

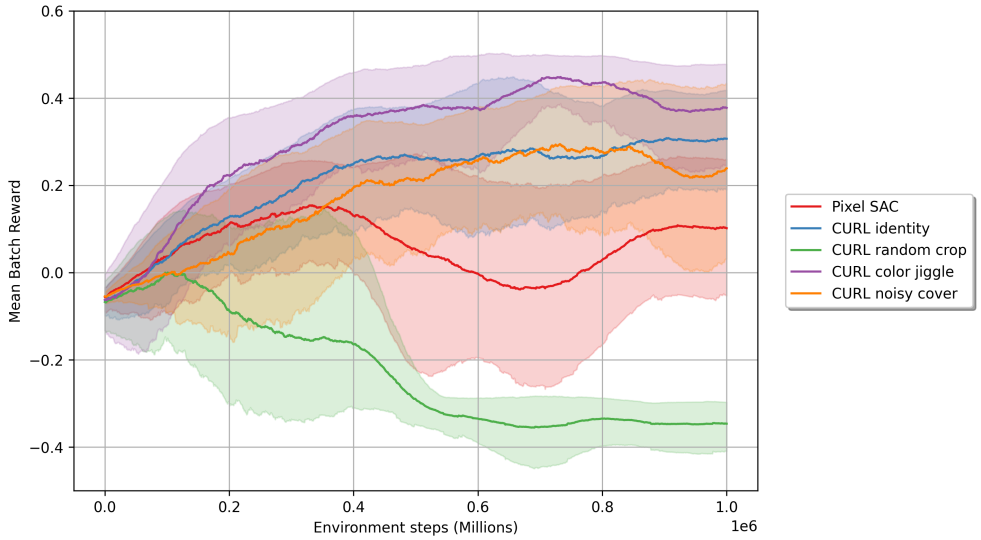


Figure 4.2: Smoothed average batch reward sampled from the replay buffer per experiment during training. Each thickened line represents the smoothed average batch reward averaged over three random seeds, while the slightly transparent areas represent the smoothed maximum-minimum range across the three random seeds. The average batch reward is the average reward over a batch sampled from the replay buffer during the batch update of the SAC algorithm and was logged every 500 environment steps.

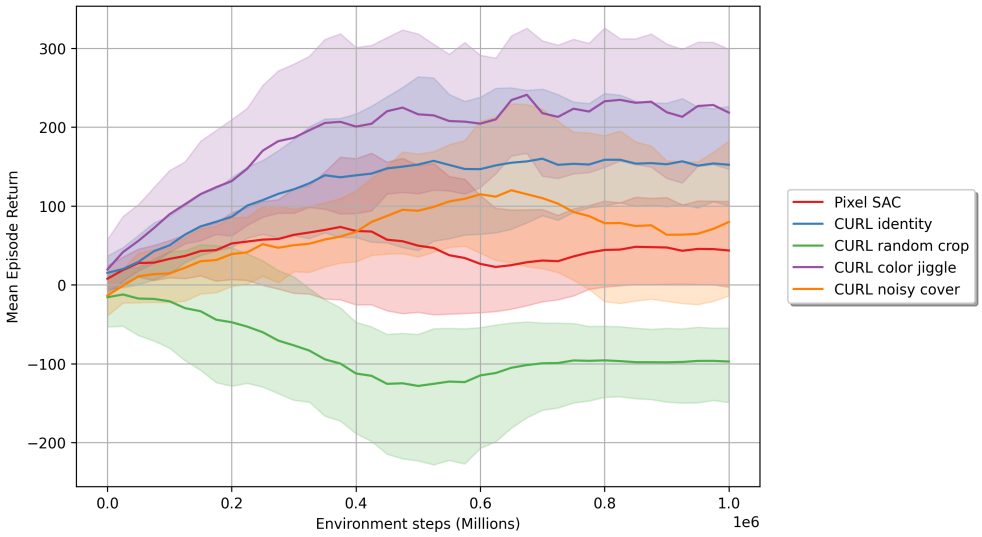


Figure 4.3: Smoothed mean evaluation episode undiscounted return per experiment during training. Each thickened line represents the smoothed mean evaluation episode undiscounted return averaged over three random seeds, while the slightly transparent areas represent the smoothed maximum-minimum range across the three random seeds. The mean evaluation episode undiscounted return is the mean of the evaluation episode undiscounted returns across 10 evaluation episodes, which are performed every 25000 environment steps during training.

jiggle, CURL identity, CURL noisy cover, Pixel SAC, CURL random crop. The performance of the *CURL noisy cover* experiment compared to the other experiments varies the most between the different figures. In this case, the mean evaluation episode undiscounted return metric used in Figure 4.3 should be taken as the most thrust-worthy metric, for reasons discussed below. The metric used in Figure 4.1 logs the undiscounted return of consecutive training episodes, meaning that each data point only contains information about a single episode. The metric used in Figure 4.2 logs the average reward over a batch of 256 transitions sampled from the replay buffer, but this replay buffer is a *first-in-first-out* queue of past transitions. The replay buffer size was set to 10^5 for all experiments (see Appendix B), meaning that the average reward over a batch could contain rewards from up to 10^5 environment steps ago. Finally, the metric used in Figure 4.3 logs the mean undiscounted return of 10 evaluation episodes, using the current policy, which solves the statistical problem of the first metric and the relevancy problem of the second metric.

In general, we can observe that the CURL agents perform better than the *Pixel SAC* agents, except for the CURL agents trained with the *random crop* augmentation. From this observation, it seems that the CURL method generally results in more sample-efficient and more performant training on the end-to-end autonomous driving task. As expected, the auxiliary task of instance discrimination

helped the CURL agents to extract more task-relevant state information from the frame-stacked observations compared to the *Pixel SAC* agents. The first CURL experiment, *CURL identity*, outperformed the *Pixel SAC* agents by roughly a twofold margin in terms of performance. In addition, it ranks as the second-best experiment in terms of performance, trailing only behind the *CURL color jiggle* experiment. This result is remarkable, considering that it significantly improves the performance and sample-efficiency over *Pixel SAC* without even leveraging any data augmentation.

The Importance of Optimality-Invariance

The *CURL random crop* agents seem to have failed to learn a good policy. This is unexpected since this is the sole augmentation method used in the original CURL paper by *Srinivas et al., 2020* [61]. Figure A.4 in Appendix A shows the smoothed logged values of the critic loss (the soft Q-function loss) for each experiment during training. In this figure, we can see that the critic loss of the *CURL random crop* experiments seems to 'explode' to large values early on during training without ever recovering. Other experiments also exhibited minor bursts in the critic loss, but in a more confined manner and regaining stability afterward. Concretely, these explosions mean that the soft Q-value estimates diverge from their targets in Equation 2.19.

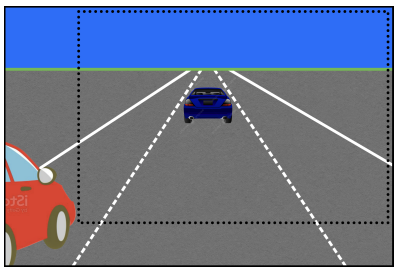


Figure 4.4: Example situation in which the random crop augmentation is not optimality-invariant in autonomous driving. The dotted rectangle represents the random-cropped observation. Note that this is an exaggerated representation.

The high critic losses in the *CURL random crop* experiments are due to the random crop augmentation not being *optimality-invariant* (cf. Section 3.3) in some situations in autonomous driving, as illustrated in Figure 4.4. In the scenario shown in this figure, a policy given the random-cropped observation might decide to overtake the blue vehicle in front, causing it to crash into the red vehicle on its left. At high speeds, this collision might lead to a large negative reward, while the expected reward or Q-value from this observation might be high. If this phenomenon manifests itself often enough within a batch, it might result in very large values of the loss function in Equation 2.19. These large losses lead to large gradients of the loss function, which will have a significant effect on the parameters of the Q-networks through backpropagation, leading to instability. This effect can potentially be mitigated by clipping the loss function or the gradients of the loss function between a maximum

and minimum value to avoid these outlier updates having too much effect on the learning. Another solution for this problem is using a wider camera view or using multiple cameras. Even then, the *random crop* augmentation will always leave out important details compared to other augmentations, making it unsuitable for autonomous driving applications.

Prior-Based Diversity and the Brittleness of RL

The *CURL color jiggle* experiment ranks as the best-performing experiment by quite a margin, which suggests that pairing the CURL method with a good image augmentation can have a significant effect on performance compared to *Pixel SAC*. By introducing random variations to the brightness, contrast, saturation, and hue of observations, prior-based diversity is introduced in the observations, which achieves two outcomes. On the contrastive side, it forces the encoder to learn how to read a traffic situation accurately in order to match queries and keys effectively. It prevents the encoder from solely basing itself on brightness, contrast, saturation, and hue to perform instance discrimination, which leads to better extraction of task-relevant state information. On the RL side, it makes the Q-functions and policy more indifferent to weather presets, which is good for generalizability.

The *CURL noisy cover* experiment shows less impressive results compared to *CURL color jiggle* and *CURL identity*. The *noisy cover* augmentation was designed to help the agents to only extract task-relevant information from observations. The top and bottom covers hide irrelevant details such as the sky, the sun, walls, trees, and the hood of the car with rectangles of a random color. Intuitively, this would lead to a harder instance discrimination task (and thus a better encoder) because only the traffic situation is visible. Next to this, the randomly colored rectangles and noise should make the agents indifferent to changes in weather presets. Unfortunately, the weather still affects the color and brightness of the road, making this augmentation only partially effective. Additionally, the complexity of this augmentation might have the opposite effect of the intentions stated above. As per *Srinivas et al., 2020: "it is preferable to pick a simple discrimination objective in the RL setting [...] considering the brittleness of reinforcement learning algorithms [30], complex discrimination may destabilize the RL objective."* [61]. Another potential failure point of this augmentation is the significant difference between the training augmentation τ_t and the evaluation augmentation τ_e (see Figure 3.8).

Finally, it is worth noting that visual inspection of recorded evaluation episodes showed that each trained agent successfully learned how to drive autonomously[2], except for the *CURL random crop* agents. The trained agents showed the ability to stay in their own lane, brake for potentially dangerous situations and overtake slower-moving vehicles.

[2] A video of a trained *CURL color jiggle* agent can be viewed at `https://github.com/paulvantieghem/curla/blob/main/README.md`

4.2 Generalizability and Robustness

In this section, a closer look is taken at the generalizability of the trained RL agents. This is done by evaluating their performance on novel weather presets that were not included in the possible weather presets during training. Furthermore, the robustness of the agents is investigated by visualizing the latent space of their encoders.

Experiment	Episode Return			Episode Steps		
	mean	max	min	mean	max	min
Pixel SAC	133	540	-186	520	1000	125
CURL identity	235	582	-55	594	1000	97
CURL random crop	-63	-31	-256	183	705	64
CURL color jiggle	**302**	639	-153	**656**	1000	168
CURL noisy cover	213	653	-136	583	1000	86

Table 4.1: Performance of the top-performing model per experiment on 50 evaluation episodes with training weather presets. The episode return is the undiscounted return.

Experiment	Episode Return			Episode Steps		
	mean	max	min	mean	max	min
Pixel SAC	108	478	-215	456	1000	98
CURL identity	194	546	-138	544	1000	97
CURL random crop	-34	-32	-37	150	150	150
CURL color jiggle	**356**	625	-96	**723**	1000	114
CURL noisy cover	264	660	-79	585	1000	87

Table 4.2: Performance of the top-performing model per experiment on 50 evaluation episodes with novel weather presets. The episode return is the undiscounted return.

4.2.1 Generalizability on Novel Scenarios

Tables 4.1 and 4.2 present the performance results of the top-performing model of each experiment on 50 evaluation episodes conducted with training weather presets and novel weather presets, respectively. The top-performing model of each experiment was selected based on its mean evaluation episode undiscounted return near the end of training. As expected, the ranking of experiments in terms of performance in Table 4.1 remains consistent with the findings of the previous section. Table 4.1 serves as a reference for Table 4.2 to measure the relative performance difference between training and novel weather presets. Therefore, the focus should not be on absolute performance, as only the top-performing models from each experiment were tested. Note that the results of the CURL random crop experiments will not be

discussed in this part due to their poor performance.

The data presented in Tables 4.1 and 4.2 shows that the performance of *Pixel SAC* and *CURL identity* experiments is comparatively poorer on the novel weather presets. This isn't surprising considering the fact that both experiments do not employ data augmentation, making them less robust to novel scenarios. Intuitively, one would assume that the *CURL color jiggle* and *CURL noisy cover* experiments would exhibit a less pronounced decline in performance on the novel weather presets compared to the *Pixel SAC* and *CURL identity* experiments, albeit still underperforming compared to the training presets. Surprisingly, however, these experiments show improved performance on the novel scenarios.

The improvement in performance on the novel scenarios can potentially be explained by the weather preset distribution in Table 3.2, where we can see that the set of training weather presets contains three *Noon* presets and four *Sunset* presets, while the set of novel weather presets contains four *Noon* presets and three *Sunset* presets. If the *CURL color jiggle* and *CURL noisy cover* agents were to have learned and generalized better on the *Noon* presets, the improvement in performance on the set of novel weather presets could be explained. This makes sense, as the *Sunset* presets have reduced visibility and contrast, and is supported by evidence declared further in this text. Nonetheless, this surprising improvement in performance still signifies robustness and indicates the generalizability of the *CURL color jiggle* and *CURL noisy cover* experiments, perhaps with a preference for *Noon* presets. The findings of this subsection suggest that the CURL method improves the generalizability of RL agents compared to plain SAC.

4.2.2 Visualization of the Latent Space

Figures A.5 - A.15 in Appendix A show visualizations of the latent space of the top-performing model of each experiment. Additionally, Figure A.17 visualizes the latent space of an untrained model with randomly initialized parameters as a baseline. In order to visualize the 50-dimensional embeddings, a technique called *t-distributed stochastic neighbor embedding* (t-SNE) is utilized *"that visualizes high-dimensional data by giving each datapoint a location in a two or three-dimensional map"* [67]. To obtain these visualizations, the policy of the top-performing model of the *CURL color jiggle* experiment was used to collect 20000 observations in the simulated autonomous driving environment, using all the training weather presets and three novel weather presets. This resulted in 34 episodes of collected data, where each weather preset was used in at least 3 episodes. The collected observations were then augmented with the evaluation augmentation and encoded by the encoders of each model. The 20000 50-dimensional latent representations of these observations were then transformed into a two-dimensional embedding space using t-SNE. Figures A.5 - A.15 present these t-SNE visualizations for each experiment, color-coded by soft Q-value and by weather preset.

Additionally, two fixed observations were selected and indicated with a red circle and a blue circle on the t-SNE plots of each experiment. The red circle observation is taken with the *CloudySunset* weather preset and contains a vehicle in the right lane in front of the ego vehicle. The blue circle observation is taken with the *WetNoon* weather preset and shows an empty highway with the ego vehicle driving on the right-most lane of the highway with a solid white line to its right. For each experiment, the closest neighbor to the embedding of each of these fixed observations was then calculated using the Euclidean distance (on the actual 50-dimensional latent representations, *not* the t-SNE transformed representations). To make things more interesting, a restriction was imposed that the neighbors must be from a weather preset at a different time of day (*Noon* vs. *Sunset*). The neighbors of the red circle and blue circle observations were then indicated by a red cross and a blue cross on all t-SNE plots, respectively. The corresponding images of these observations are also provided. The results reveal that the *CURL identity*, *CURL color jiggle*, and *CURL noisy cover* trained agents found semantically identical neighbors to the given observations, while agents from other experiments failed to do so for one or both observations. Although this evaluation method may not be the most scientifically rigorous, it does offer anecdotal evidence and insights into the robustness of the agents. Note that the absolute (Euclidean) distance is meaningless in a t-SNE plot and that the focus should be on the relative difference and separation between clusters.

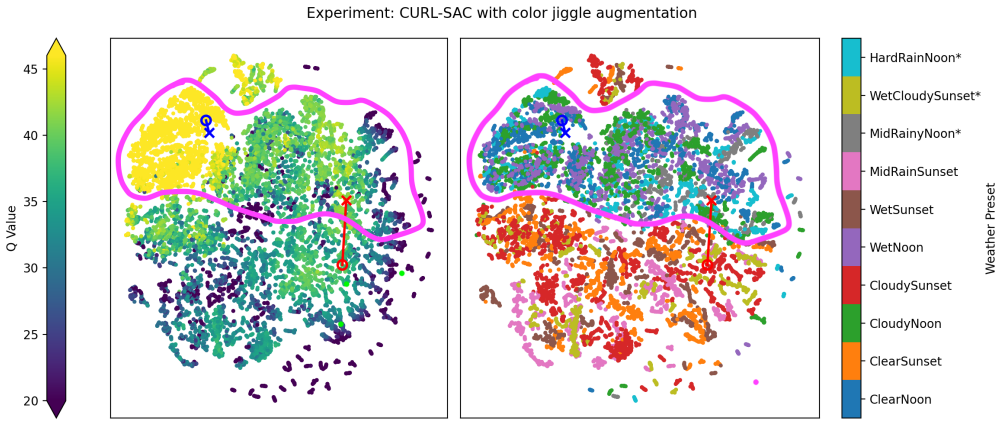


Figure 4.5: Repetition of t-SNE latent space visualization in Figure A.13, but with indication of the *Noon* super-cluster with higher Q-values in magenta.

Noon and *Sunset* Super-Clusters

In all the t-SNE latent space visualization plots in Figures A.5 - A.15, *'super-clusters'* of *Noon* and *Sunset* weather presets can be observed[3] across individual clusters. An

[3]Although the t-SNE plots of the CURL identity experiments don't contains any real clusters, regions of *Noon* and *Sunset* weather presets are still clearly observed.

ideal encoder for RL would not contain such clusters and purely encode observations based on the state information contained within them. These super-clusters are probably formed because the *Sunset* weather presets entail a more complex task due to the reduced visibility compared to the *Noon* weather presets. Since the parameters of the encoder are also affected by the RL objective, observations of similar Q-value will likely be represented close to each other. The Q-value color-coded plots of all the experiments show that generally, higher Q-values are assigned to *Noon* presets, which supports this hypothesis[4].

CURL Improves Robustness

The latent space visualization of the *Pixel SAC* experiment in Figure A.5 shows a clear separation between individual clusters of weather presets, indicating the lack of robustness. The latent spaces of the *CURL color jiggle* and *CURL noisy cover* models in Figures A.13 and A.15 show a greater overlap between data of different weather presets, although the *Noon* and *Sunset* super-clusters are quite clearly separated. The representations look more robust, with the novel weather preset representations overlapping with the training weather preset representations, which was not the case for the *Pixel SAC* model. These findings once again suggest the effectiveness of the CURL method (and data augmentation in general) in improving the robustness and generalizability of RL agents.

Investigation of the *CURL Identity* Experiment

The latent space of the *CURL identity* model in Figure A.7 looks completely different than the latent space of other experiments. The representations are all grouped in one large 'blob', with no clear clusters visible. At first glance, the representations look robust because representations of different weather presets are well mixed. From the testing on novel weather presets, however, we know that this model is not as robust as the *CURL color jiggle* and *CURL noisy cover* models. Various approaches were attempted to better understand this latent space, including adjusting the perplexity of the t-SNE algorithm, employing different distance metrics for the t-SNE transform (such as the cosine similarity), and using principal component analysis (PCA) for dimensionality reduction instead of t-SNE. However, the outcomes consistently indicated that the latent space of the *CURL identity* experiment does not contain any inherent clusters. The instance discrimination objective without data augmentation causes the representations to be maximally scattered over a 50-dimensional unit sphere in a similar process as in *Wu et al., 2018* [75]. The unit sphere aspect of this space is caused by the layer standardization applied to the outputs of the encoder. In other words, the contrastive learning objective without data augmentation causes every observation to be pushed away from others in a 50-dimensional standardized space. The RL objective also affects the latent space by reordering the representations by Q-value, which can clearly be observed in all t-SNE plots. The interaction

[4]This also supports the argument made in the previous subsection about the effect of the weather preset distribution for training and novel scenario testing on the performance of the agents.

between the contrastive objective and the RL objective results in a Q-value sorted hyper-sphere of embeddings with a couple of low Q-value outliers, as in Figure A.7.

In order to investigate this further, an additional experiment was carried out, called *CURL identity detached* (cf. Table 3.1), where the RL objective is refrained from being able to optimize the parameters of the encoder f_{θ_q} (the encoder is *detached* from the RL objective). Figures A.1 - A.3 and Tables A.1 - A.2 extend the training and evaluation metrics previously presented in this chapter with the results of the *CURL identity detached* experiment. The results show that the *CURL identity detached* experiment performs worse than the *CURL identity* experiment as expected, but that its performance is on par with the *Pixel SAC* experiment. The latent space of the additional experiment is visualized in Figure A.9. In this figure, we can observe that its latent space looks very similar to the *CURL identity* latent space when color-coded by weather preset and dissimilar when color-coded by Q-value, which is expected due to the detached encoder.

The fact that the *CURL identity detached* experiment is able to match the performance of the *Pixel SAC* experiment demonstrates that training an encoder on an instance discrimination task using unaugmented data can still lead to an encoder that can extract approximate state information or task-relevant features from observations in RL. This indicates that the encoder does not merely learn a trivial encoding to succeed at the instance discrimination task, but rather develops the ability to extract meaningful information. These findings are similar to *Wu et al., 2018* [75], where instance discrimination without augmentations was found to extract useful features for image classification. The reduced performance of the *CURL identity detached* experiment compared to the *CURL identity* experiment highlights the importance of the fine-tuning effect of the RL objective on the parameters of the encoder.

4.3 Conclusion

From the results discussed in this chapter, it can be concluded that the auxiliary task of instance discrimination plays a crucial role in enabling RL agents' encoders to extract meaningful features from observations, even in the absence of data augmentation. These encoders are simultaneously fine-tuned by the RL objective, and this parallel optimization of the encoder is what leads to improved performance and sample-efficiency over the *Pixel SAC* method. Next to this, the generalizability and robustness of these agents can be significantly improved by leveraging carefully designed data augmentations. Too complex augmentations or augmentations that are not optimality-invariant may lead to instability, however, due to the brittleness of RL algorithms.

Chapter 5

Conclusion and Future Work

In this final chapter, the findings and conclusions derived from the research are summarized. Additionally, areas for future work and potential improvements are identified.

5.1 Conclusion

In this thesis, the effectiveness of combining reinforcement learning with contrastive learning for robust end-to-end autonomous driving was investigated in the area of deep, model-free, visual reinforcement learning in continuous action spaces. This was done by applying the CURL methodology proposed by *Srinivas et al., 2020* [61] to a simulated autonomous driving environment. The CURL methodology combines the *Soft Actor-Critic* (SAC) algorithm with an auxiliary contrastive learning task. In the original paper by *Srinivas et al.*, the CURL methodology is presented as a state-of-the-art method in terms of sample-efficiency for model-free visual reinforcement learning. This thesis extended the CURL methodology with additional data augmentations and investigated the generalizability and robustness of CURL agents on the task of end-to-end autonomous driving.

A total of six experiments utilizing different CURL configurations were conducted in order to fully understand the effect of the different parts of the CURL method, applied to end-to-end autonomous driving. The findings of this thesis showed that the addition of an auxiliary contrastive learning objective significantly improves the performance and sample-efficiency of RL agents for end-to-end autonomous driving compared to the SAC algorithm. These findings align with those presented in the original CURL paper. However, it has been demonstrated that the improvement persists even when data augmentations are not applied, albeit to a lesser degree. This aspect was not addressed in the original CURL research. Furthermore, the experiments conducted in this thesis agree with the CURL paper on the realization that decoupling the representation learning of the encoder networks from the reinforcement learning objective leads to less performant agents (cf. *CURL identity detached* experiment).

The area in which data augmentation proved most powerful was in improving the agents' generalizability and robustness against novel scenarios. However, these data augmentations ought to be carefully designed because data augmentations that are not optimality-invariant or too complex tend to lead to instability during the training of the RL agents. For example, it was discovered that the *random cropping* data augmentation used in the original CURL method was not suitable for the task of end-to-end autonomous driving. This is because this augmentation is not *optimality-invariant* in the context of autonomous driving. Data augmentations that introduce *prior-based diversity*, such as randomly varying the brightness, contrast, saturation, and hue of observations, were found to result in more performant and robust policies.

5.2 Future Work

As mentioned in Section 3.4, training one RL agent for 10^6 environment steps in the simulated autonomous driving environment using an observation size of 160×90 pixels and a batch size of 256 took between 48 and 120 hours on a GPU cluster of the Flemish Supercomputer Center, depending on the available type of GPU and chosen data augmentation. Each of the six experiments conducted in this thesis consisted of training three RL agents using different random seeds. This resulted in the training of 18 RL agents, using more than 1000 hours of computation. Given the sizeable computational needs per experiment, conducting more than six experiments was not feasible given the limited (but still very generous) computational budget allocated for this thesis. The list below provides an overview of potential areas of future work that could either have a significant impact on the performance of RL agents on the end-to-end autonomous driving task or improve the quality of the experiments conducted in this research.

- *Better baselines:* The baseline experiment in this thesis, *Pixel SAC*, utilized the *Soft Actor-Critic* (SAC) algorithm on visual observations. A better baseline for sample-efficiency and robustness would be *State SAC*, where the agent is trained on accurate state information measured by sensors or provided by the simulator. This was not tested due to the limited scope of this thesis, but would certainly be possible by utilizing the vast amount of sensors and information accessible through the CARLA Python API. Next to this, different auxiliary tasks such as reconstruction using (variational) autoencoders could be tested to compare to the contrastive instance discrimination task utilized in this thesis.

- *Improved reward function:* The design of the reward function plays a crucial role in the success of RL since it is essentially the only source of 'labels' in reinforcement learning. The goal of RL is to obtain a policy that maximizes the cumulative reward. If the reward function is badly designed, it is consequently impossible to obtain good policies. The reward function utilized in this thesis was partly based on prior work and partly based on common sense. Next to this, only limited experimentation was available (again, due to computational

limitations) to tune the weights of the components of the reward function λ_{r_i} (cf. Section 3.1). Further research on the design and tuning of reward functions could thus have a notable impact on the autonomous driving abilities of the trained RL agents.

- *Tweaking until perfection:* In the experiments conducted in this thesis, all RL agents were trained using identical hyperparameters. Better results could be achieved by conducting a hyperparameter search to identify the optimal set of hyperparameters. However, performing such a hyperparameter search would require a colossal computational budget.

- *Larger is better:* As per *He et al., 2021*, *"scale is the main driver behind the rise of deep learning [15, 28, 38, 60, 63]. Larger datasets and neural networks consistently yield better performance across all tasks"* [29]. Almost every aspect of the experiments in this thesis could be scaled up to achieve better performance, such as the number of parameters of the networks (more convolutional layers, larger latent space dimension, deeper MLPs), the batch size, the observation size (larger observations contain more detailed information), the number of cameras (a surround view of the ego vehicle would be much better), etc. Unfortunately, every one of these suggestions would also have a significant impact on the computational requirement.

- *Transfer learning:* Repurposing encoders that have been pre-trained on an image classification or object detection task, specifically on a dataset relevant to autonomous driving (such as the *CityScapes Dataset* [13]), would lead to better sample-efficiency. This is because the parameters of the encoders would already be relatively well optimized to extract relevant state information prior to training. Recent research around alternative methods of leveraging pre-training in autonomous driving, such as *policy pre-training* by *Wu et al., 2023* [74] has shown very promising results.

- *Closing the gap between simulation and reality:* Better simulators are crucial for the future of autonomous driving, and this is illustrated by the substantial investments made by self-driving car companies in this space. The version of CARLA utilized in this thesis (version 0.9.8) suffers from unrealistic and unstable turning physics. Because of this, all trained RL agents showed oscillations when applying small control actions to the steering when trying to stay in their lane on the highway. These oscillations sometimes became quite unstable, leading to collisions. Newer versions of the CARLA simulator were tested (up to the latest version at the time of writing, 0.9.14) in which these oscillations were not observed. Unfortunately, these newer versions were not stable enough to consistently run for 10^6 environment steps without crashing. Using future versions of CARLA with improved physics that are stable enough for extensive training of RL agents would lead to much smoother and more performant autonomous driving agents.

Appendices

Appendix A

Additional Result Plots

Experiment	Episode Return			Episode Steps		
	mean	max	min	mean	max	min
Pixel SAC	133	540	-186	520	1000	125
CURL identity detached	171	315	-26	838	1000	105
CURL identity	235	582	-55	594	1000	97
CURL random crop	-63	-31	-256	183	705	64
CURL color jiggle	**302**	639	-153	**656**	1000	168
CURL noisy cover	213	653	-136	583	1000	86

Table A.1: Performance of the top-performing model per experiment on 50 evaluation episodes with training weather presets. The episode return is the undiscounted return.

Experiment	Episode Return			Episode Steps		
	mean	max	min	mean	max	min
Pixel SAC	108	478	-215	456	1000	98
CURL identity detached	161	265	-44	834	1000	181
CURL identity	194	546	-138	544	1000	97
CURL random crop	-34	-32	-37	150	150	150
CURL color jiggle	**356**	625	-96	**723**	1000	114
CURL noisy cover	264	660	-79	585	1000	87

Table A.2: Performance of the top-performing model per experiment on 50 evaluation episodes with novel weather presets. The episode return is the undiscounted return.

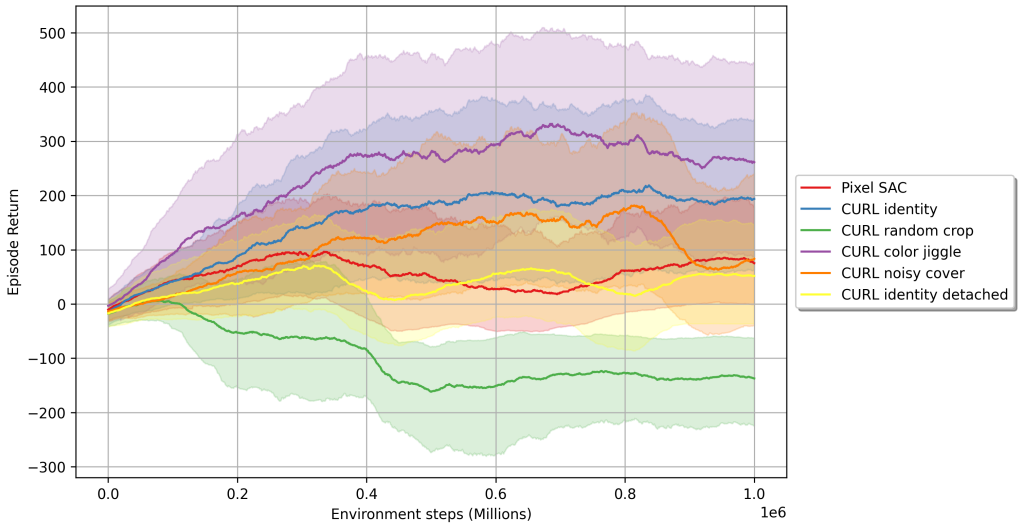


Figure A.1: Smoothed undiscounted return of the training episodes per experiment during training. Each thickened line represents the smoothed performance averaged over three random seeds, while the slightly transparent areas represent the smoothed maximum-minimum range across the three random seeds. The undiscounted episode return was logged whenever a training episode ended.

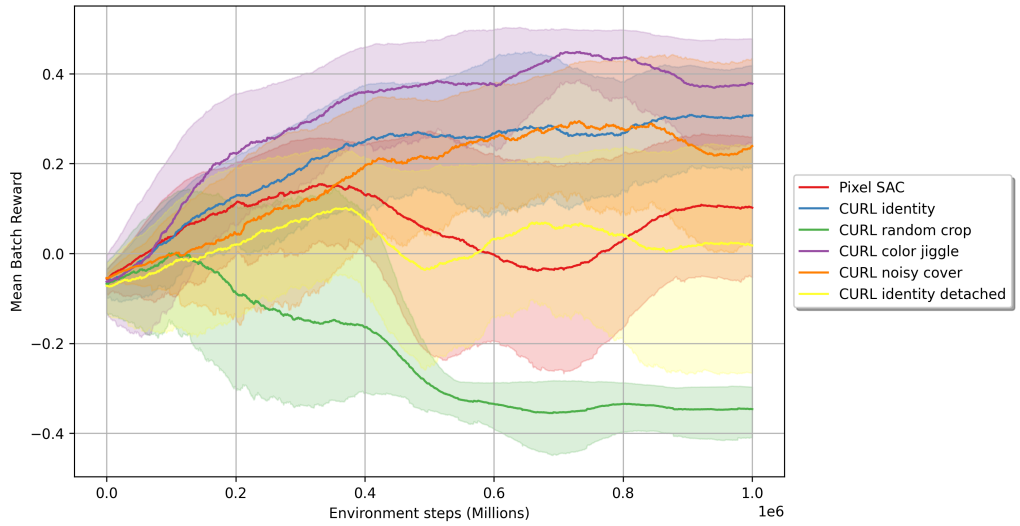


Figure A.2: Smoothed average batch reward sampled from the replay buffer per experiment during training. Each thickened line represents the smoothed average batch reward averaged over three random seeds, while the slightly transparent areas represent the smoothed maximum-minimum range across the three random seeds. The average batch reward is the average reward over a batch sampled from the replay buffer during the batch update of the SAC algorithm and was logged every 500 environment steps.

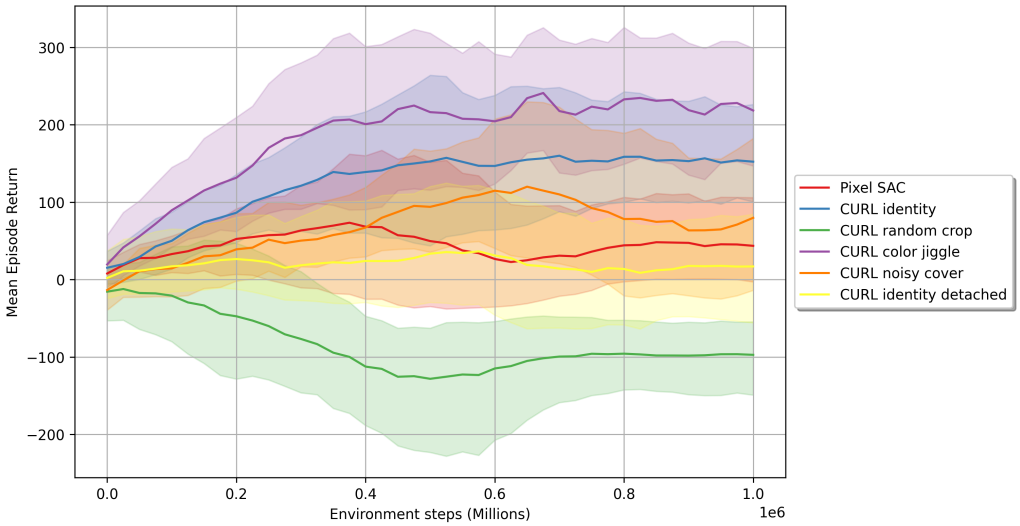


Figure A.3: Smoothed mean evaluation episode undiscounted return per experiment during training. Each thickened line represents the smoothed mean evaluation episode undiscounted return averaged over three random seeds, while the slightly transparent areas represent the smoothed maximum-minimum range across the three random seeds. The mean evaluation episode undiscounted return is the mean of the evaluation episode undiscounted returns across 10 evaluation episodes, which are performed every 25000 environment steps during training.

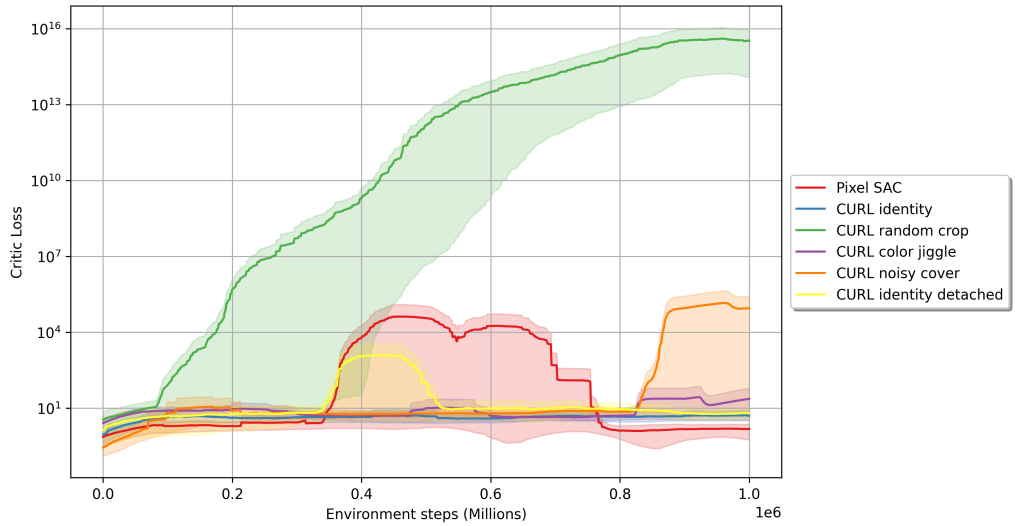


Figure A.4: Smoothed batched critic loss per experiment during training. Each thickened line represents the smoothed critic loss over a batch averaged over 3 random seeds, while the slightly transparent areas represent the smoothed maximum-minimum range across the 3 random seeds. The critic loss over a batch is calculated using Equation 2.24 during the batch update of the SAC algorithm, and was logged every 500 environment steps.

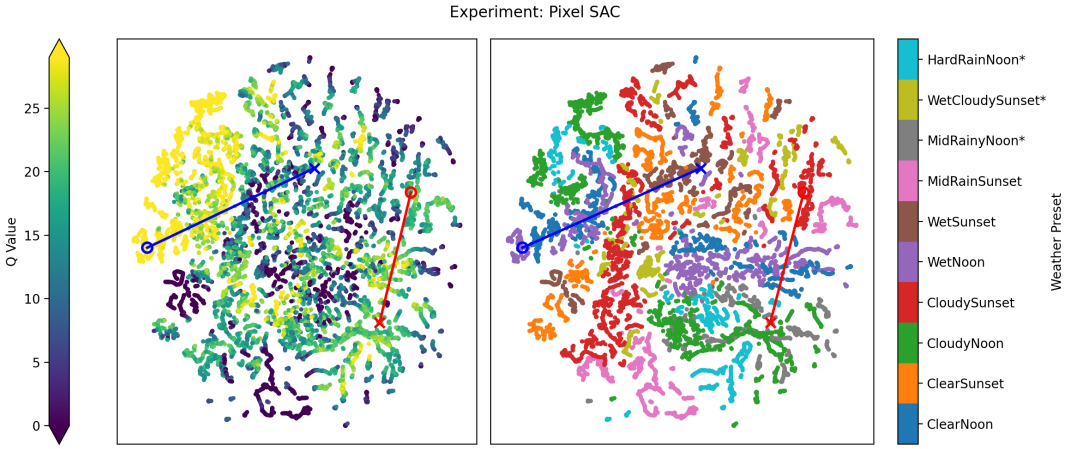


Figure A.5: t-SNE visualization of the latent space representations of 20000 observations for the *Pixel SAC* experiment, color-coded by soft Q-value following the policy (*left plot*) and color-coded by weather preset of the observation (*right plot*). Novel weather presets (not included in the training set) are indicated with an asterisk in the legend of the right plot. *Red circle:* t-SNE transform of the top-left observation in Figure A.6. *Red cross:* t-SNE transform of the top-right observation in Figure A.6. *Blue circle:* t-SNE transform of the bottom-left observation in Figure A.6. *Blue cross:* t-SNE transform of the bottom-right observation in Figure A.6.

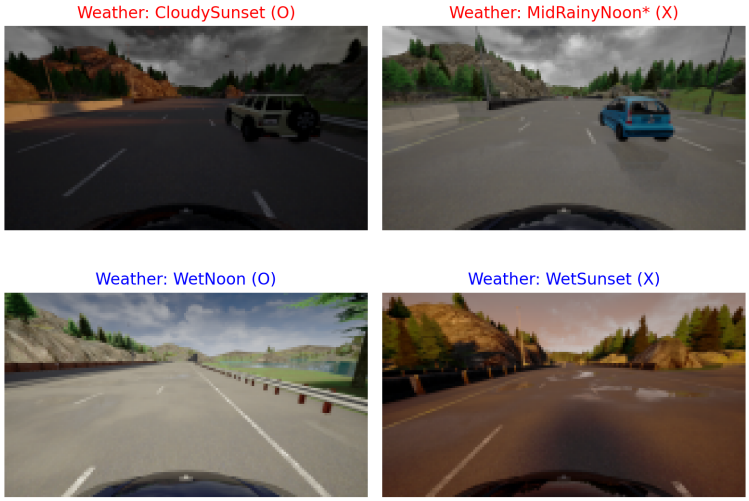


Figure A.6: Observations corresponding to the red and blue circles and crosses in the t-SNE visualization of the *Pixel SAC* experiment in Figure A.5.

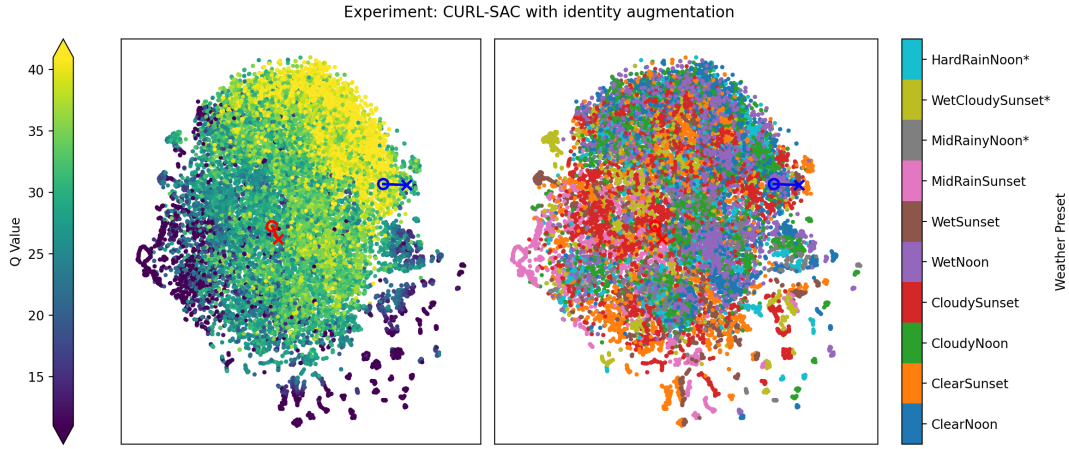


Figure A.7: t-SNE visualization of the latent space representations of 20000 observations for the *CURL identity* experiment, color-coded by soft Q-value following the policy (*left plot*) and color-coded by weather preset of the observation (*right plot*). Novel weather presets (not included in the training set) are indicated with an asterisk in the legend of the right plot. *Red circle:* t-SNE transform of the top-left observation in Figure A.8. *Red cross:* t-SNE transform of the top-right observation in Figure A.8. *Blue circle:* t-SNE transform of the bottom-left observation in Figure A.8. *Blue cross:* t-SNE transform of the bottom-right observation in Figure A.8.

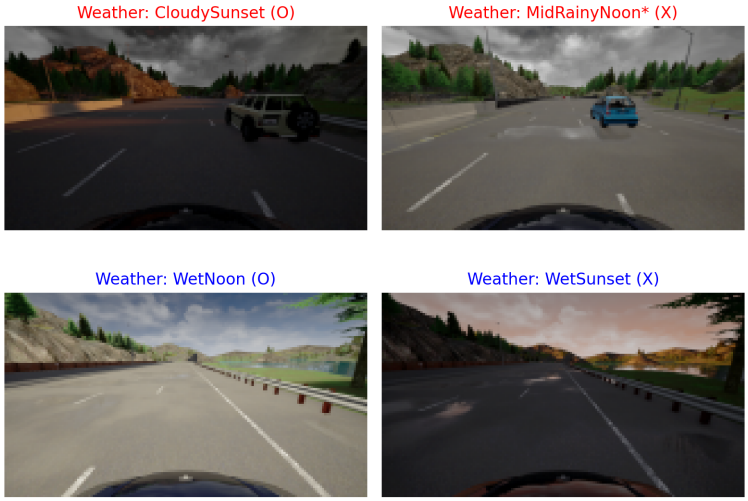


Figure A.8: Observations corresponding to the red and blue circles and crosses in the t-SNE visualization of the *CURL identity* experiment in Figure A.7.

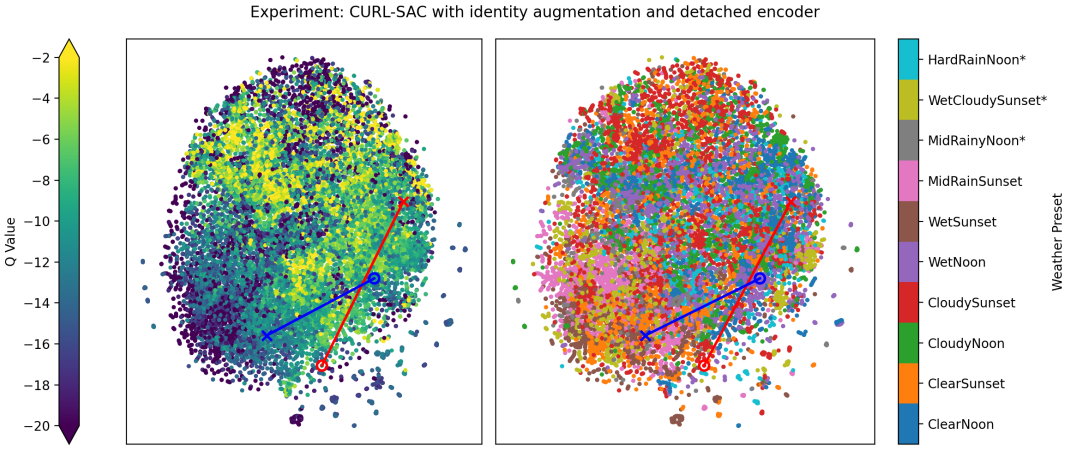


Figure A.9: t-SNE visualization of the latent space representations of 20000 observations for the *CURL identity* experiment, color-coded by soft Q-value following the policy (*left plot*) and color-coded by weather preset of the observation (*right plot*). Novel weather presets (not included in the training set) are indicated with an asterisk in the legend of the right plot. *Red circle:* t-SNE transform of the top-left observation in Figure A.10. *Red cross:* t-SNE transform of the top-right observation in Figure A.10. *Blue circle:* t-SNE transform of the bottom-left observation in Figure A.10. *Blue cross:* t-SNE transform of the bottom-right observation in Figure A.10.

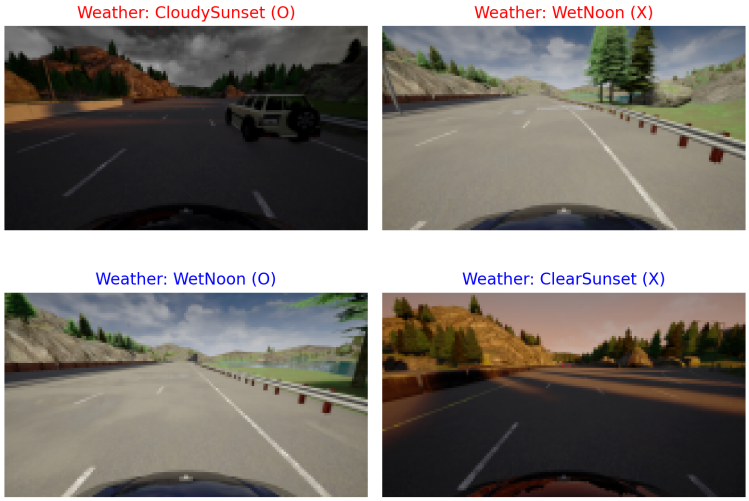


Figure A.10: Observations corresponding to the red and blue circles and crosses in the t-SNE visualization of the *CURL identity* experiment in Figure A.9.

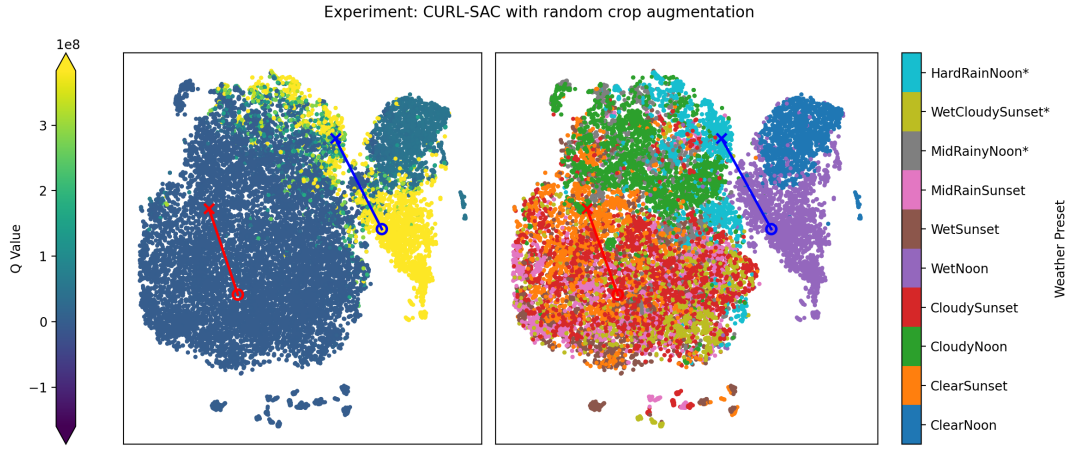


Figure A.11: t-SNE visualization of the latent space representations of 20000 observations for the *CURL random crop* experiment, color-coded by soft Q-value following the policy (*left plot*) and color-coded by weather preset of the observation (*right plot*). Novel weather presets (not included in the training set) are indicated with an asterisk in the legend of the right plot. *Red circle:* t-SNE transform of the top-left observation in Figure A.12. *Red cross:* t-SNE transform of the top-right observation in Figure A.12. *Blue circle:* t-SNE transform of the bottom-left observation in Figure A.12. *Blue cross:* t-SNE transform of the bottom-right observation in Figure A.12.

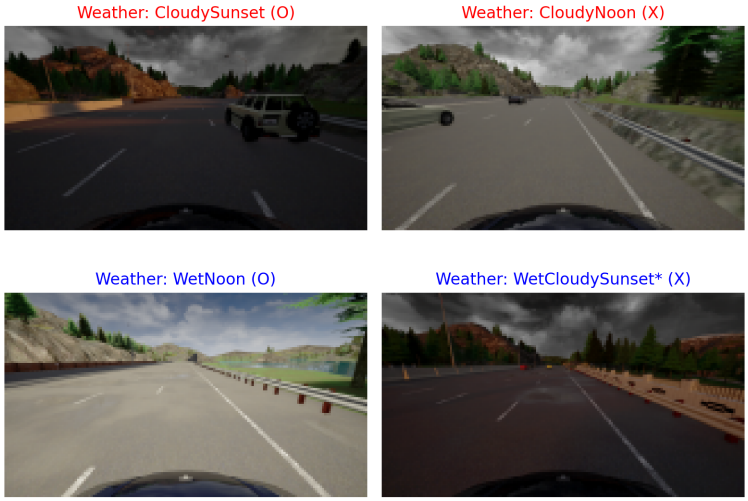


Figure A.12: Observations corresponding to the red and blue circles and crosses in the t-SNE visualization of the *CURL random crop* experiment in Figure A.11.

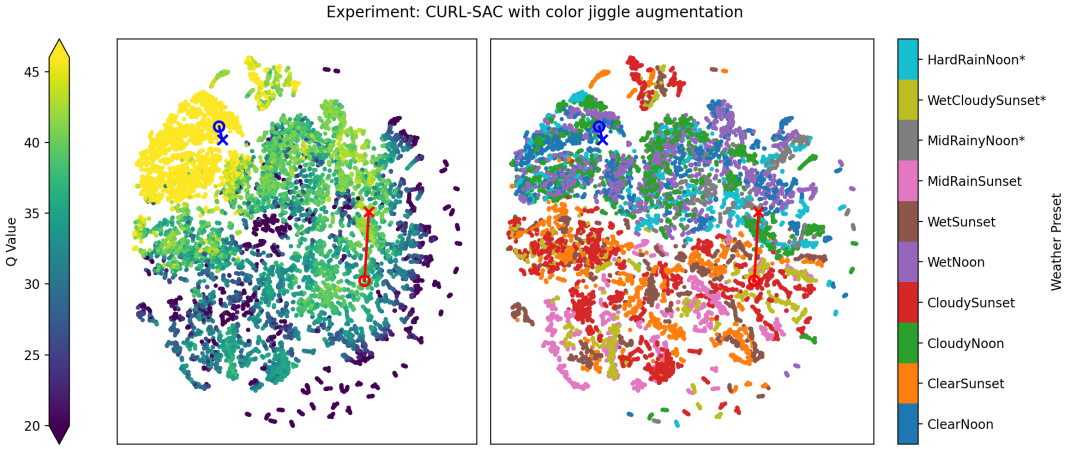


Figure A.13: t-SNE visualization of the latent space representations of 20000 observations for the *CURL color jiggle* experiment, color-coded by soft Q-value following the policy (*left plot*) and color-coded by weather preset of the observation (*right plot*). Novel weather presets (not included in the training set) are indicated with an asterisk in the legend of the right plot. *Red circle:* t-SNE transform of the top-left observation in Figure A.14. *Red cross:* t-SNE transform of the top-right observation in Figure A.14. *Blue circle:* t-SNE transform of the bottom-left observation in Figure A.14. *Blue cross:* t-SNE transform of the bottom-right observation in Figure A.14.

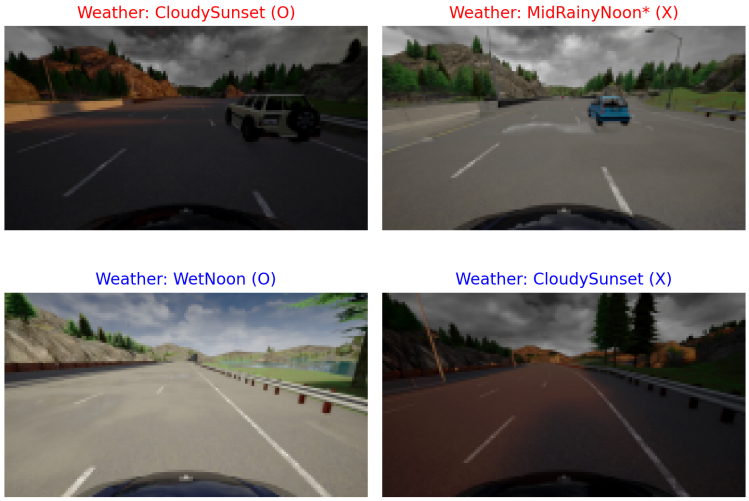


Figure A.14: Observations corresponding to the red and blue circles and crosses in the t-SNE visualization of the *CURL color jiggle* experiment in Figure A.13.

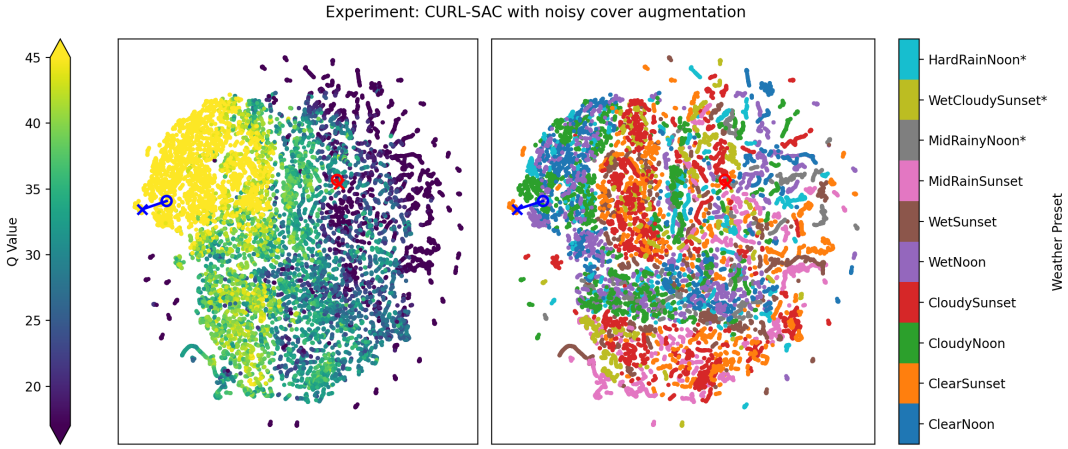


Figure A.15: t-SNE visualization of the latent space representations of 20000 observations for the *CURL noisy cover* experiment, color-coded by soft Q-value following the policy (*left plot*) and color-coded by weather preset of the observation (*right plot*). Novel weather presets (not included in the training set) are indicated with an asterisk in the legend of the right plot. *Red circle:* t-SNE transform of the top-left observation in Figure A.16. *Red cross:* t-SNE transform of the top-right observation in Figure A.16. *Blue circle:* t-SNE transform of the bottom-left observation in Figure A.16. *Blue cross:* t-SNE transform of the bottom-right observation in Figure A.16.

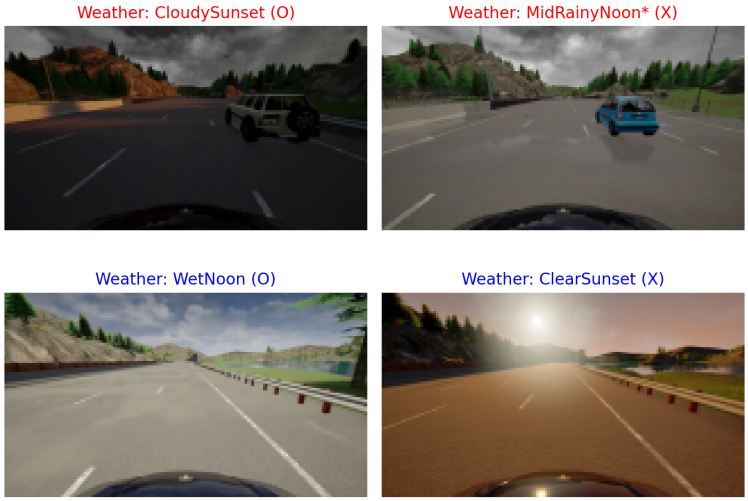


Figure A.16: Observations corresponding to the red and blue circles and crosses in the t-SNE visualization of the *CURL noisy cover* experiment in Figure A.15.

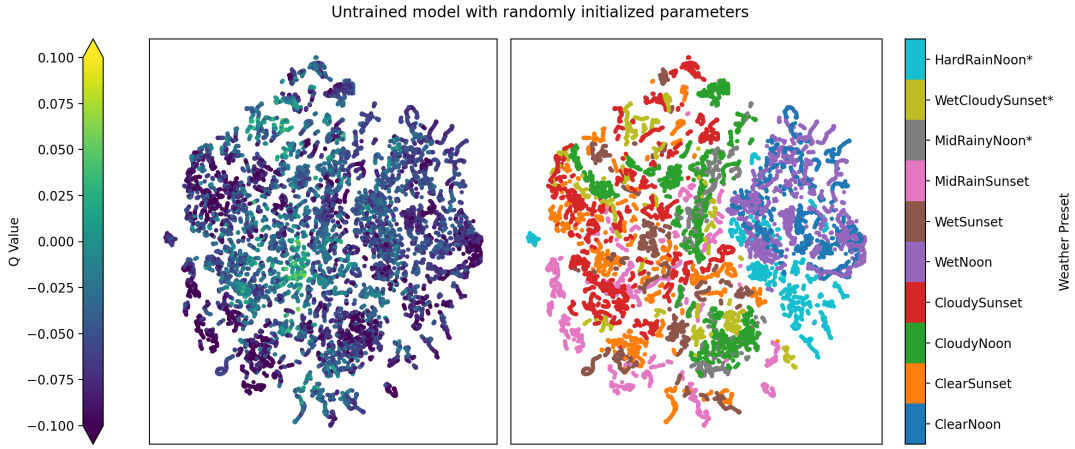


Figure A.17: t-SNE visualization of the latent space representations of 20000 observations for an untrained model with randomly initialized parameters., color-coded by soft Q-value following the policy (*left plot*) and color-coded by weather preset of the observation (*right plot*).

Appendix B

Experiment Hyperparameters

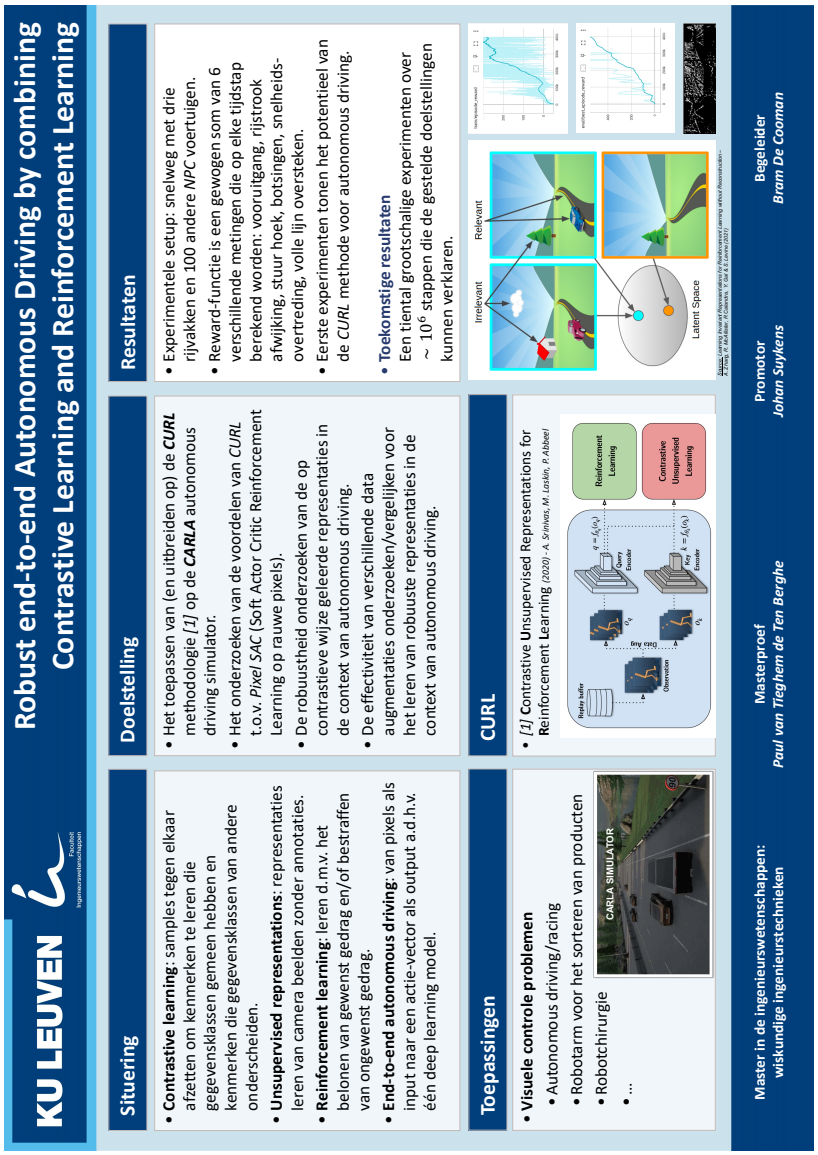
Hyperparameter	Value
Observations	
Observation size	(160,90)
Field of view (FOV)	110
Stacked frames f_s	3
Environment	
CARLA Town	'Town04'
Number of NPC vehicles	10
Desired speed [km/h]	65
Maximum allowed stall time [s]	5
Minimum speed not to stall [km/h]	0.5
Maximum episode duration [s]	50
Frames per second (FPS) [Hz]	20
Reward function	
Highway progression coefficient λ_{r_1}	1.0
Center of lane deviation coefficient λ_{r_2}	0.3
Steering angle coefficient λ_{r_3}	1.0
Collision coefficient λ_{r_4}	0.005
Speeding coefficient λ_{r_5}	1.0
General CURL & SAC	
Initial replay buffer data collection steps without parameter updates	1000
Number of environment steps	10^6
Evaluation frequency	25000
Number of evaluation episodes	10
Discount factor γ	0.99
Initial temperature α	0.1
Learning rate for α	10^{-4}

Replay buffer size	10^5
Batch size	256
Optimizer for all networks	Adam
Learning rate for all networks	10^{-3}
Encoder networks f_θ	
Number of convolutional layers	4
Number of filters per convolutional layer	32
Momentum coefficient c_m for target network parameters	0.05
Latent dimension m	50
Actor network π_ψ	
Hidden layer dimension h	1024
Number of hidden layers	2
Activation function	ReLU
Update frequency	2
Critic networks Q_ϕ	
Hidden layer dimension h	1024
Number of hidden layers	2
Activation function	ReLU
Update frequency	2
Momentum coefficient c_m for target network parameters	0.01

Table B.1: Hyperparameters used for all the experiments carried out in this thesis, per category.

Appendix C

Poster



Bibliography

[1] P. Abbeel. Foundations of Deep Reinforcement Learning Lecture Series. https://www.youtube.com/playlist?list=PLwRJQ4m4UJjNymuBM9RdmB3Z9N5-0IlY0.

[2] P. Abbeel. Partially Observable Markov Decision Processes (POMDPs) - CS 287 Advanced Robotics (Fall 2019) Lecture 15. https://people.eecs.berkeley.edu/~pabbeel/cs287-fa19/slides/Lec15-POMDPs.pdf.

[3] Accolade Technology. 5 Levels of Autonomy (L1 and L2). https://accoladetechnology.com/5-levels-of-autonomy-l1-and-l2/.

[4] J. Achiam and OpenAI. Spinning Up in Deep RL. https://spinningup.openai.com/en/latest/index.html.

[5] A. Amini, W. Schwarting, G. Rosman, B. Araki, S. Karaman, and D. Rus. Variational autoencoder for end-to-end control of autonomous driving with novelty detection and training de-biasing, 10 2018.

[6] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[7] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, D. Guo, and C. Blundell. Agent57: Outperforming the atari human benchmark, 2020.

[8] M. G. Bellemare, W. Dabney, and R. Munos. A distributional perspective on reinforcement learning. *CoRR*, abs/1707.06887, 2017.

[9] H. Blockeel. *Course Text Machine Learning and Inductive Inference*. Katholieke Universiteit Leuven, 2014-2015 edition, 2014.

[10] R. Chekroun, M. Toromanoff, S. Hornauer, and F. Moutarde. GRI: general reinforced imitation and its application to vision-based autonomous driving. *CoRR*, abs/2111.08575, 2021.

[11] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton. A simple framework for contrastive learning of visual representations, 2020.

[12] S. Chopra, R. Hadsell, and Y. LeCun. Learning a similarity metric discriminatively, with application to face verification, 2005.

[13] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding, 2016.

[14] W. Dabney, M. Rowland, M. G. Bellemare, and R. Munos. Distributional reinforcement learning with quantile regression. *CoRR*, abs/1710.10044, 2017.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[16] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. Carla: An open urban driving simulator, 2017.

[17] B. Eysenbach and S. Levine. Maximum entropy rl (provably) solves some robust rl problems, 2022.

[18] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney. Revisiting fundamentals of experience replay, 2020.

[19] V. Feinberg, A. Wan, I. Stoica, M. I. Jordan, J. E. Gonzalez, and S. Levine. Model-based value estimation for efficient model-free reinforcement learning. *CoRR*, abs/1803.00101, 2018.

[20] F. Fuchs, Y. Song, E. Kaufmann, D. Scaramuzza, and P. Durr. Super-human performance in gran turismo sport using deep reinforcement learning. *IEEE Robotics and Automation Letters*, 6(3):4257–4264, jul 2021.

[21] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018.

[22] D. Ha and J. Schmidhuber. Recurrent world models facilitate policy evolution. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[23] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. Reinforcement learning with deep energy-based policies, 2017.

[24] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

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

[26] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality Reduction by Learning an Invariant Mapping, 2006.

[27] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick. Momentum contrast for unsupervised visual representation learning, 2020.

[28] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

[29] X. He, F. Xue, X. Ren, and Y. You. Large-scale deep learning optimizations: A comprehensive survey, 2021.

[30] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. Deep reinforcement learning that matters, 2019.

[31] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

[32] Y. Huang and Y. Chen. Autonomous driving with deep learning: A survey of state-of-art technologies, 2020.

[33] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit, 2017.

[34] L. Kaiser, M. Babaeizadeh, P. Milos, B. Osinski, R. H. Campbell, K. Czechowski, D. Erhan, C. Finn, P. Kozakowski, S. Levine, A. Mohiuddin, R. Sepassi, G. Tucker, and H. Michalewski. Model-based reinforcement learning for atari, 2020.

[35] T. Kanade, C. Thorpe, and W. Whittaker. *Autonomous Land Vehicle Project at CMU*. CSC '86. Association for Computing Machinery, 1986.

[36] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley, and A. Shah. Learning to drive in a day, 2018.

[37] I. Kostrikov, D. Yarats, and R. Fergus. Image augmentation is all you need: Regularizing deep reinforcement learning from pixels, 2021.

[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. Burges, L. Bottou, and K. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[39] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman. Building machines that learn and think like people, 2016.

[40] M. Laskin, K. Lee, A. Stooke, L. Pinto, P. Abbeel, and A. Srinivas. Reinforcement learning with augmented data, 2020.

[41] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[42] Y. LeCun, S. Chopra, R. Hadsell, M. Ranzato, and F. J. Huang. A Tutorial on Energy-Based Learning, 2006.

[43] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning, 2019.

[44] G. Ma, Z. Wang, Z. Yuan, X. Wang, B. Yuan, and D. Tao. A comprehensive survey of data augmentation in visual reinforcement learning, 2022.

[45] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016.

[46] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

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

[48] A. Nagabandi, G. Kahn, R. S. Fearing, and S. Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. *CoRR*, abs/1708.02596, 2017.

[49] OpenAI, :, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Jozefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[50] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang. Solving rubik's cube with a robot hand, 2019.

[51] B. Ozmen. Automl for data augmentation. https://blog.insightdatascience.com/automl-for-data-augmentation-e87cf692c366, 2021.

[52] Papers With Code. Experience replay. https://paperswithcode.com/method/experience-replay#:~:text=Experience\%20Replay%20is%20a%20replay,episodes%20into%20a%20replay%20memory.

[53] D. Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In D. Touretzky, editor, *Proceedings of (NeurIPS) Neural Information Processing Systems*, pages 305 – 313. Morgan Kaufmann, December 1989.

[54] A. Ranga, F. Giruzzi, J. Bhanushali, E. Wirbel, P. Perez, T.-H. Vu, and X. Perrotton. Vrunet: Multi-task learning model for intent prediction of vulnerable road users, 2020.

[55] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[56] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

[57] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[58] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos. Compute trends across three eras of machine learning, 2022.

[59] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[60] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[61] A. Srinivas, M. Laskin, and P. Abbeel. Curl: Contrastive unsupervised representations for reinforcement learning, 2020.

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

[63] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions, 2014.

[64] V7Labs. Autoencoders in Deep Learning: Tutorial & Use Cases [2023]. https://www.v7labs.com/blog/autoencoders-guide.

[65] V7Labs. The Beginner's Guide to Contrastive Learning. https://www.v7labs.com/blog/contrastive-learning-guide.

[66] A. van den Oord, Y. Li, and O. Vinyals. Representation learning with contrastive predictive coding, 2019.

[67] L. van der Maaten and G. Hinton. Viualizing data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605, 11 2008.

[68] H. van Hasselt. Double Q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[69] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning, 2015.

[70] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

[71] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. W. Battaglia, D. Silver, and D. Wierstra. Imagination-augmented agents for deep reinforcement learning. *CoRR*, abs/1707.06203, 2017.

[72] L. Weng. Contrastive representation learning. https://lilianweng.github.io/posts/2021-05-31-contrastive/, 2021.

[73] Wikipedia contributors. Darpa grand challenge. https://en.wikipedia.org/wiki/DARPA_Grand_Challenge, 2023.

[74] P. Wu, L. Chen, H. Li, X. Jia, J. Yan, and Y. Qiao. Policy pre-training for autonomous driving via self-supervised geometric modeling, 2023.

[75] Z. Wu, Y. Xiong, S. Yu, and D. Lin. Unsupervised feature learning via non-parametric instance-level discrimination, 2018.

[76] D. Yarats, A. Zhang, I. Kostrikov, B. Amos, J. Pineau, and R. Fergus. Improving sample efficiency in model-free reinforcement learning from images, 2020.

[77] A. Zhang, R. McAllister, R. Calandra, Y. Gal, and S. Levine. Learning invariant representations for reinforcement learning without reconstruction, 2021.

[78] Z. Zhang and M. R. Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels, 2018.

[79] B. D. Ziebart. Modeling purposeful adaptive behavior with the principle of maximum causal entropy, 2010.

Master's thesis filing card

Student: Paul van Tieghem de Ten Berghe

Title: Robust End-to-End Autonomous Driving by combining Contrastive Learning and Reinforcement Learning

UDC: 621.3

Abstract:
When driving, humans observe traffic situations with their eyes (input), process the information in their brains (neural network), and apply control actions to the vehicle (output). The mapping from inputs to outputs is performed by one large neural network from end to end. A possible approach towards autonomous driving, called end-to-end autonomous driving, mimics this process by training artificial neural networks with deep reinforcement learning (RL). RL is a powerful technique to learn policies purely from interactions with an environment. In visual RL, image observations are used as input, similar to how humans rely on visual information. However, visual RL agents suffer from sample-inefficiency and lack of robustness against novel scenarios. Sample-inefficiency can be addressed by using off-policy RL algorithms such as *Soft Actor-Critic* (SAC) and adding an auxiliary task with an unsupervised objective. On the other hand, the robustness of the RL agents can be improved by leveraging data augmentation. One possible approach that combines these solutions and uses contrastive learning as the unsupervised objective is the *'Contrastive Unsupervised Representations for Reinforcement Learning'* (CURL) method proposed by *Srinivas et. al., 2020* [61]. In this thesis, the CURL methodology is applied to an end-to-end autonomous driving scenario, and extended with additional data augmentations. The experiments conducted in this study compare RL agents trained using various configurations of the CURL method with each other on an end-to-end autonomous driving task. Furthermore, this thesis focuses on investigating the generalizability and robustness of CURL agents, areas that were not addressed in the original CURL paper. The findings show that using an auxiliary contrastive learning objective significantly improves the performance and sample-efficiency of RL agents compared to the SAC algorithm, even in the absence of data augmentation. Additionally, the generalizability and robustness of these agents can be improved by leveraging data augmentations. It was found that these data augmentations ought to be carefully designed because data augmentations that are not optimality-invariant or too complex tend to lead to instability during training of the RL agents, due to the brittleness of RL algorithms. Moreover, the data augmentation used in the original CURL paper was found to be unsuitable for autonomous driving. Overall, the CURL method demonstrates its effectiveness in achieving more robust and sample-efficient RL agents for end-to-end autonomous driving. Further research is warranted to explore scaling up models and simulators, among other aspects, in order to fully uncover the potential of this approach.

Thesis submitted for the degree of Master of Science in Mathematical Engineering

Thesis supervisor: Prof. dr. ir. Johan Suykens

Assessors: Prof. dr. ir. Wim Michiels
Prof. dr. Nick Vannieuwenhoven

Mentor: Ir. Bram De Cooman