From Heightmaps to Cameras: Teacher-Student Reinforcement Learning for Rover Navigation

Master's thesis

By Thomas Schou Sørensen

Aalborg Universitet
Det Tekniske Fakultet for IT og Design

The Technological Faculty for IT og Design
Niels Jernes Vej 10, 9220 Aalborg Øst
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
From Heightmaps to Cameras: Teacher-Student Reinforcement Learning for Rover Navigation

Theme:
Thesis

Project Period:
February 2024 - June 2024

Project Group:
Group 1066

Participant(s):
Thomas Schou Sørensen 20204534

Supervisor(s):
Simon Bøgh
Anton Bjørndahl Mortensen

Copies: 1

Page Numbers: 69

Date of Completion:
4. June 2025

Abstract:

This thesis investigates a DAgger-based teacher-student framework for transferring navigation behavior from a policy trained on privileged heightmap input to one using noisy RGB-D observations. The goal is to support learning under realistic sensor conditions, where reinforcement learning on RGB-D data remains challenging due to its partial and high-dimensional nature. A simulation pipeline was extended to enable separate sensor inputs for teacher and student using the RobuROC4 platform. While the student increasingly aligned with the teacher's actions, it failed to generalize or perform the task effectively. This is attributed to memory-bound dataset handling and limited rollout diversity. The findings suggest that teacher-student imitation learning holds promise for sensor modality transfer, but depends on scalable infrastructure capable of supporting large and diverse training datasets.

Acknowledgments

I would like to express my sincere gratitude to Associate Professor Simon Bøgh and PhD student Anton Bjørndahl Mortensen for their invaluable guidance, support, and supervision throughout this thesis. I would also like to thank Aalborg University for providing the facilities and resources necessary to carry out the work presented in this project.

Aalborg University, 4. June 2025

Thomas Schou Sørensen

tssa20@student.aau.dk

Contents

1 Introduction

The current plan of NASA's Artemis program aims at returning humans to the Moon by 2027 [1]. Building upon this milestone, the broader roadmap envisions humans setting foot on Mars by 2030 [2], [3]. With the successful Artemis I mission in 2022, the future of space exploration is one step closer to creating a more permanent presence in space. Artemis II, the second mission, intends to send a crewed craft near the Moon by 2026 [2]. With the Artemis program, scientific efforts are focused on developing technologies for lunar exploration and habitation, with significant investments from multiple space agencies and associations. These efforts are aimed at advancing technologies for exploring, preparing, settling and utilizing the lunar surface. The Moon is intended as a stepping stone for further deeper space exploration, with the current goal being Mars. It also provides a platform for developing and testing technologies intended for future missions to Mars.

To realize these ambitions, both NASA and the European Space Agency (ESA) emphasize the need for advanced robotic systems capable of autonomous operation in extreme and unstructured environments [4], [5]. In particular, preparing the lunar surface for habitation and infrastructure, including landing pads, shelters and equipment foundations, requires robotic systems with advanced sensing, mobility and decision-making capabilities [6]. Neither direct human nor teleoperated execution of these tasks is feasible due to communication delays, terrain unpredictability and mission constraints.

Reinforcement learning (RL), a machine learning technique, is increasingly being explored for its ability to support autonomous decision-making in complex environments [7], [8]. RL allows agents to learn optimal behaviors through trial-and-error interaction with their environment. However, applying RL in mission-critical systems where human intervention is not possible, raises concerns not only about safety, interpretability, and verification, but also about sim-to-real transfer, as the system must function autonomously in unfamiliar environments [9], [10]. These issues must be addressed for a robotic agent to operate reliably in space and in extraterrestrial contexts.

Mortensen et al. [11] propose a hybrid teacher-student reinforcement learning approach, inspired by the Learning by Cheating method introduced by Chen et al. [12], to reduce the sim-to-real gap. Learning by Cheating is a two-stage imitation learning architecture in which a teacher agent is trained using privileged information (e.g., perfect sensor data or ground-truth obstacle positions in simulation). A student agent is then trained to imitate the teacher's decisions using more realistic, constrained inputs (e.g. raw camera data) [12]. Building on this paradigm, Mortensen and Bøgh developed the RLRoverLab framework [13], [14], a modular

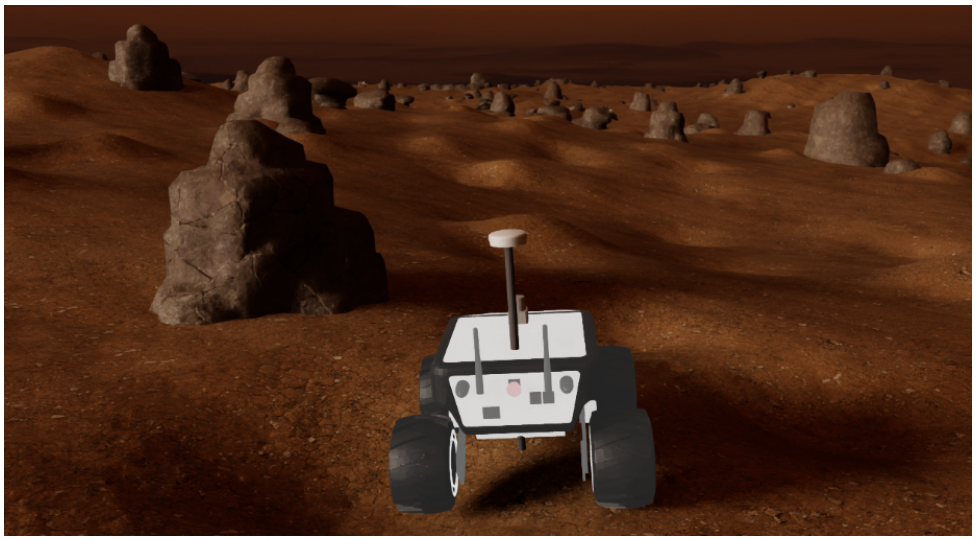

Figure 1.1: Illustratory image of a robot model in a Mars-like terrain inside an Isaac Sim simulation.

RL suite for planetary rover simulation and RL policy development, illustrated in Figure 1.1.

In its current implementation, Motensen et al. employs the same sensor modality, specifically heightmaps, for both the teacher and student agents. The use of similar sensor inputs for both student and teacher simplifies the imitation learning process. However, in domains such as lunar and Martian exploration, where sensor availability is limited, an investigation of using other sensor inputs could offer additional insights into more realistic and robust sim-to-real transfer [11], [13], [15].

Building on the RLRoverLab framework and prior RL robotics research at Aalborg University [16], [17], this thesis extends the work of Mortensen et al. [11] by investigating the effects of using different sensor inputs for teacher and for student agents. This establishes the possibility of utilizing existing policies in training complex or observation redundant agents. Separating sensor inputs can potentially improve output policy robustness and enhance sim-to-sim and sim-to-real transferability. In doing so, this work can contribute to the development of scalable and safety-aware reinforcement learning methods for terrain exploration and preparation in support of NASA's Artemis and ESA's Terrae Novae programs [4], [5].

Initial problem statement

Based on these findings and research, it is evident that further research and development is required in order to bridge the sim-to-real gap and achieve the collective goals of the space agencies. With these goals and associated challenges in mind, the following problem statement is formulated:

> *"How does using separate sensor inputs in a teacher-student framework influence training efficiency, sim-to-sim and sim-to-real transferability of autonomous navigation policies for extraterrestrial rover missions?"*

2 Problem analysis

Building on this motivation and the formulated problem statement, this chapter provides the necessary theoretical foundation for further investigating the RL framework and RL training approach presented by Mortensen et al. [11], [13]. Given that the aim is developing RL training strategies for extraterrestrial navigation tasks, it is essential to first understand the structure of RL problems and the methods used to solve them. Therefore, this chapter begins with an introduction to RL, covering the fundamental theory, learning objectives, and selected actor-critic methods relevant to this work.

Following this, the core ideas behind the teacher-student learning paradigm are introduced, starting with the foundation laid in the Learning by Cheating framework, and then exploring the teacher-student approach in greater detail to establish a basis for its extension. Building upon the background, a review of recent reinforcement learning approaches for navigation based on RGB and depth sensing is conducted, with a particular focus on their potential for sim-to-real transferability. The review further examines key components of the RL framework, such as observation modalities, action spaces, reward functions, and sensor configurations, to identify effective learning strategies for vision-based agents. The chapter concludes by exploring neural network architectures for vision-based navigation, with the aim of establishing a robust and transferable policy network.

2.1 Reinforcement Learning fundamentals

In order to address the issues associated with rover navigation and surface preparation within the bounds of RL it is essential to establish a theoretical foundation, as presented here in a brief overview of the fundamentals of Reinforcement Learning. Additionally, this leads to a more informed discussion of the methods proposed by Mortensen et. al. Given the expansive and complex nature of RL the following overview is limited to the relevant concepts, with the aim of preserving both clarity and brevity. The overview is based on the Book; ”Reinforcement Learning: An Introduction” by Sutton and Barto [18] as well as a review by Levine et. al. [19].

Foundation and Ideas of Reinforcement Learning

RL is a subfield of machine learning that formalizes decision-making through interaction with an environment. It relies on building experience through trial-and-error, thereby optimizing a behavior over time. By definition, RL closely resembles human learning paradigms due to the action-feedback or sensorimotor behavior that humans have, e.g. touching a hot stove resulting in pain. These inputs shape future behavior through stimulus-response.

At the core, RL is a framework consisting of an **agent** which learns to make decisions within a set **environment**. The **environment** is defined by a set of *states* ($\mathcal{S}$) that represent various situations the agent may encounter, including the initial state d_0. Each *state* ($s \in \mathcal{S}$) contains information observable to the **agent**. Based on this information, the **agent** can choose between a set of available *actions* ($\mathcal{A}(s) \subseteq \mathcal{A}$). The choice of *action* ($a \in \mathcal{A}$) is governed by a *policy* ($\pi(a|s)$), which maps *states* to the probability of taking certain *actions* ($p(a|s), \forall a \in A(s)$). When an *action* is taken, the *environment* responds by transitioning to a new *state* and providing a scalar *reward* signal, depending on the resulting *state* ($r(s,a,s')$). This *reward* may be negative and positive, depending on the outcome. An illustration of this framework can be seen in Figure 2.1.

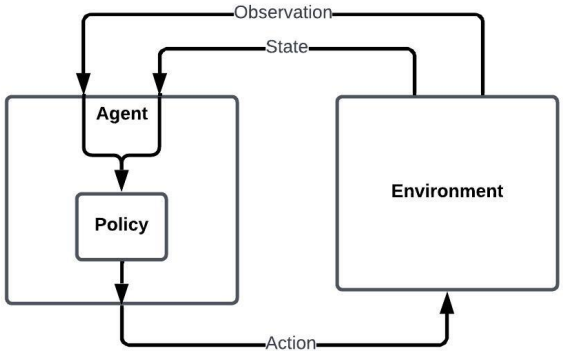

Figure 2.1: An illustration of the basic RL framework.

To improve the behavior of the *agent* over time, the *policy* must adapt accordingly. The agent can refine its policy using various learning algorithms, but before these can be applied, it is necessary to define a formal framework in which the learning process is situated. In reinforcement learning, this framework is typically modeled as a Markov Decision Process (MDP).

Markov Decision Processes (MDPs)

The conceptual framework of RL, as defined previously, is based on a set of states S, which must include the initial state d_0. Furthermore, it presents a set of actions A and a reward function for taking specific actions $r(s,a,s')$. Lastly, it provides the concept of choosing actions based on some probability distribution $p(a|s) \quad \forall a \in A(s)$. In simple terms, this provides the foundation for an MDP and can therefore be modeled as such. For more realistic contexts another approach is to utilize Partially-Observed Markov Decision processes (POMDPs). In general, an MDP assumes that the agent has full knowledge of the state of the environment, such as in a game of tic-tac-toe by knowing which fields contain which playing pieces, whereas POMDPs assume that the agent only has partial knowledge. The latter is often used in real contexts as many systems utilize imperfect sensors which in part only provides a fraction of information about the environment. Both of these frameworks are defined as:

Definition 2.1.1 (Markov Decision Process) *The Markov decision process (MDP) is defined as a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, d_0, r, \gamma)$, where $\mathcal{S}$ is the set of states $s \in \mathcal{S}$, which can be both discrete or continuous. $\mathcal{A}$ is the set of available actions $a \in \mathcal{A}$, which can also be both discrete or continuous. $\mathcal{T}$ defines the conditional probability distribution of the form $\mathcal{T}(s_{t+1}|s_t, a_t)$, also known as the dynamics of the MDP. d_0 defines the initial state distribution $d_0(s_0)$. $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ or as defined earlier $r(s, a, s')$ defines the reward function, while $\gamma : (0, 1]$ is the scalar discount factor for rewards, that defines the value of actions over time.*

Definition 2.1.2 (Partially-Observed Markov Decision Process) *The partially-observed Markov decision process (POMDP) is defined as a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, d_0, E, r, \gamma$, where $\mathcal{S}$, where $\mathcal{S}, \mathcal{A}, \mathcal{T}, d_0, r$ and γ are defined as before. $\mathcal{O}$ is a set of observations, where each observation is given by $o \in \mathcal{O}m$ and E is an emission function, which defines the distribution $E(o_t|s_t)$, or the probability of receiving a certain observation o_t at a particular state s_t.*

For the sake of simplicity and brevity, the MDP Definition 2.1.1 is used for the remainder of this theoretical explanation. Central to RL is the goal of learning some optimal policy, here denoted π^*, that maximizes the expected return for all *trajectories*. For this purpose, a trajectory τ is here defined as a sequence of state-action pairs $\tau = \{s_0, a_0, s_1, a_1, \ldots, s_H, a_H\}$ for some trajectory horizon H. Thus, by this definition, the probability density function of some trajectory, here τ, anchored in some policy π is defined as:

$$p_\pi(\tau) = d_0(s_0) \prod_{t=0}^{H} \pi(a_t|s_t) \mathcal{T}(s_{t+1}|s_t, a_t) \tag{2.1}$$

Where d_0 represents the initial state distribution and t denotes the discrete time step. This probability density function can then be utilized in definition of the optimization problem for some agent can then be defined as:

$$\pi^* = \underset{\pi}{arg\,max} \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[\sum_{t=0}^{H} \gamma^t r(s_t, a_t) \right] \tag{2.2}$$

Where τ represents the sequence of a trajectory and $p_\pi(\tau)$ denotes the probability of a trajectory sequence given some policy π. This problem can be divided into two parts, the optimization problem and the objective function, each denoting a critical part of adapting the behavior of an agent. The optimization problem defines the process through which the agent updates its policy in order to improve performance, while the objective function quantifies the goal the agent aims to achieve. The optimization problem is defined as:

$$\pi^* = \underset{\pi}{arg\,max} J(\pi) \tag{2.3}$$

And the objective function is defined as the expected return of all future rewards given some trajectory τ under some policy π. In other words, the assumed outcome r of taking actions $a_0, \dots a_H$, with some discount factor γ:

$$J(\pi) = \mathbb{E}_{\tau \sim p_\pi(\tau)} \left[\sum_{t=0}^{H} \gamma^t r(s_t, a_t) \right] \tag{2.4}$$

With both the objective function and optimization problem defined the remaining issue is defining algorithms which can be utilized in learning the optimal policy π^*. There exists a multitude of approaches, depending on the architecture of the environment and the observations available to the agent. The following section explores a few of the approaches available. While many fundamental methods assume complete knowledge of the environment, these are omitted here for relevance and brevity.

Policy optimization and Actor-critic methods

Algorithms designed for this purpose are generally categorized as either *value-based*, *policy-based*, or a combination of the two. Value-based methods focus on estimating value functions, such as $V^*(s_t)$ or $Q^*(s_t, a_t)$, which reflect the expected return under a given policy. In contrast, policy-based methods directly optimize the policy $\pi_\theta(a_t|s_t)$ by ascending the gradient of expected return with respect to its parameters.

In the context of methods for learning an optimal policy, another important distinction is between the *prediction* and *control* problems. The prediction problem involves estimating the value function $v_\pi(s)$ for a fixed policy π, thereby evaluating the policy. Contrarily, the control problem is directly concerned with learning the optimal policy π^*, and is part of the focus of this section. A classical *policy-based* solution is policy gradients, which is a more direct optimization method that optimizes the policy π by computing the gradient of the expected reward, with regard to the policy parameters θ. This method is defined as:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\mathcal{T} \sim p_\pi(\tau)} \left[\sum_{t=0}^{H} \gamma^t \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\underbrace{\sum_{t=0}^{H} \gamma^{t'-t} r(s_{t'}, a_{t'})}_{\text{future expected reward}} - \underbrace{b(s_t)}_{\text{baseline}} \right) \right] \tag{2.5}$$

Where π_θ is a policy parameterized by θ and $b(s_t)$ represents the baseline. The baseline, in this case, is utilized in reducing the variance of the gradient estimate. The equation, consisting

of the future expected reward and the baseline is known as the Advantage function, denoted as $A(s_t, a_t)$ and is defined as:

$$A(s_t, a_t) = \sum_{t'=t}^{H} \gamma^{t'-t} r(s_{t'}, a_{t'}) - b(s_t) \tag{2.6}$$

This function can be estimated either through Monte Carlo sampling or by employing a separate neural network, commonly referred to as the *critic*, as utilized in actor-critic methods. Actor-critic methods address the dual objectives of policy optimization and policy evaluation by maintaining two distinct learning components:

- The **Actor** estimates the parameterized policy by using the estimates from the critic to optimize the policy using Equation 2.5.
- The **Critic** estimates the parameterized value function (either state-value $V(s_t)$ or action-value $Q(s_t, a_t)$), thereby providing the advantage function $A(s_t, a_t)$, which can then be utilized in optimizing the policy.

To better understand the role of the critic in estimating long-term returns, we next define the value functions more formally. To enable the RL framework to learn beyond any immediate rewards (greedy policies) a function estimating the expected long-term return (cumulative future rewards) starting from state s, is required. This is known as the *state-value* function, denoted $V^*(s_t)$. Additionally, another function describing the value of taking action a in a state s following a policy π. This function is known as the *action-value* function (or Q-function) and is denoted $Q^*(s_t, a_t)$. The definitions of each using the Bellmann optimality equations are:

$$V^*(s_t) = r(s_t, a_t) + \gamma \max_{a \in \mathcal{A}} \mathbb{E}_{s' \sim \mathcal{T}(s_{t+1}|s_t, a_t)} \left[V^*(s_{t+1}) \right] \tag{2.7}$$

$$Q^*(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s' \sim \mathcal{T}(s_{t+1}|s_t, a_t)} \left[\max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \right] \tag{2.8}$$

Depending on the application, a finite MDP can be solved by estimating either $Q^*(s_t, a_t)$ or $V^*(s_t)$. The actor-critic architecture enables the agent to simultaneously evaluate and improve its behavior, making actor-critic methods a powerful and general approach to policy optimization. Moreover, since the critic uses bootstrapped estimates and the actor optimizes a differentiable policy, the method is both more sample-efficient and suitable for continuous action spaces. The critic network contributes by estimating either the state-value or action-value function. This is done by using regression to minimize the loss function for $V^*(s_t)$:

$$\mathcal{L}_{critic} = \frac{1}{2N} \sum_{n=1}^{N} ||V_\phi^\pi(s_t) - (r(s_t, a_t) + \gamma V_\phi^\pi(s_{t+1}))||^2 \tag{2.9}$$

With ϕ being the parameter for the value function. In order to optimize the policy, using $V^*(s_t)$ to estimate the advantage function $A(s_t, a_t)$, which in this context is defined as:

$$A(s_t, a_t) = r(s_t, a_t) + \gamma V_\phi^\pi(s_{t+1}) - V^\pi(s_t) \tag{2.10}$$

Which can then be utilized in Equation 2.5 to obtain the policy gradient for the actor network. In general, $V_\phi^\pi(s_{t+1})$ is considered easier for neural networks to estimate since it only depends on a single condition. However, using the state-value function $V^\pi(s_{t+1})$ assumes that the next action is drawn from the current policy π_θ. Unlike on-policy methods, off-policy approaches use experience collected from a different behavior policy $\pi_{\theta_{old}}$, which invalidates the estimate due to a policy mismatch. Instead the action-value function $Q_\phi^\pi(s_t, a_t)$ can be used, which is more suitable for off-policy RL, which is beyond the scope of this section, as it depends on sampled data $\mathcal{D} = \{s_t, a_t, r(s_t, a_t), s_{t+1}\}$, that does not depend on a specific policy. This is crucial, since the action-value function does not depend on the state-action pairs originating from a specific policy. For estimating the action-value function $Q_\phi^\pi(s_t, a_t)$, the sampled data is used in recalculating it, by assuming the current policy π_θ. Here the sampled action $a_{t+1} \sim \pi_\theta(a_{t+1}|s_{t+1})$ can be recalculated to bootstrap future return. This leads to the critic loss function:

$$\mathcal{L}_{critic} = \frac{1}{2N} \sum_{n=1}^{N} ||Q_\phi^\pi(s_t, a_t) - (r(s_t, a_t) + \gamma Q_\phi^\pi(s_{t+1}, a_{t+1})||^2 \tag{2.11}$$

Where both Q_ϕ^π and the policy π_θ are updated iteratively. The policy gradient is then reformulated as:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s_t \sim d_t^\pi, a_t \sim \pi_\theta(a_t|s_t)} \left[\nabla_\theta \log \pi_\theta(a_t|s_t) Q_\phi^\pi(s_t, a_t) \right] \tag{2.12}$$

This formulation ensures consistency between the critic and the actor by using current parameters ϕ and θ, enabling stable learning in the off-policy regime. Building on this foundation, several widely-used actor-critic algorithms have been developed, each implementing the underlying principles in distinct ways to improve stability, efficiency, and generalization in various settings. Notable examples include:

- **PPO (Proximal Policy Optimization)**: Introduces a clipped surrogate objective to constrain policy updates, improving training stability while retaining simplicity. [20]
- **TRPO (Trust Region Policy Optimization)**: Optimizes policies with a constraint on the KL divergence between successive policies, providing strong theoretical guarantees at the cost of increased complexity. [21]
- **RPO (Robust Policy Optimization)**: Refers to a class of methods aimed at learning poli-

cies that generalize across uncertain or variable environments by optimizing performance under worst-case scenarios. [22]

- **SAC (Soft Actor-Critic)**: An off-policy algorithm based on the maximum entropy framework, encouraging diverse and robust policies by maximizing both expected return and policy entropy. [23]
- **TD3 (Twin Delayed DDPG)**: An off-policy deterministic algorithm that reduces overestimation bias through the use of double critics, target smoothing, and delayed policy updates. [24]

These algorithms are typically implemented within the framework of deep reinforcement learning, where both the actor and the critic are represented by neural networks. This allows them to approximate complex value functions and policies from high-dimensional sensory input, such as images or proprioceptive data. These algorithms highlight the flexibility of the actor-critic paradigm, differing in whether they operate on-policy or off-policy, how they manage stability, and the strategies they use for exploration. These foundational elements serve as the basis for more advanced reinforcement learning techniques applied in real-world robotics scenarios, including the work by Mortensen et al. The following sections explore practical applications of these concepts, specifically in rover navigation.

2.2 Background

Efficient navigation and surface interaction are critical challenges in applying RL to autonomous extraterrestrial rovers. To address the difficulty of learning complex behaviors in sparse environments with limited real-world data availability, recent work has explored structured training setups that guide the learning process. One such approach is Learning by Cheating, introduced by Chen et al. [12], which leverages privileged information during training to accelerate policy learning. Building on this idea, Mortensen et al. [11] proposed a teacher-student framework tailored to the domain of extraterrestrial rover navigation. In this framework, a teacher policy with access to high-level planning data trains a student policy that operates under realistic constraints.

While the teacher-student framework is not available in the current implementation of *RLRover-Lab*, this simulation framework, developed by Mortensen and Bøgh [13], provides a modular and extensible platform designed to support the development and evaluation of reinforcement learning strategies for planetary rover navigation. The following sections briefly review the core ideas behind these contributions.

2.2.1 Learning-by-cheating

Deploying machine learning-based control systems in robotics often rely on training within simulation due to the challenges of acquiring large-scale real-world data. However, the gap

between simulation and reality, known as the sim-to-real gap, can lead to degraded performance when transferring trained agents to physical environments, due to the differences in dynamics, sensor noise and environmental complexity [10].

An approach presented by Chen et al. [12] introduces a two-stage imitation learning framework aimed at addressing the sim-to-real gap in autonomous systems. The core idea is to decompose policy learning into two separate stages: first, a *privileged agent* is trained to imitate expert demonstrations using full access to high-level environmental information in simulation (e.g., object states, maps and signals), which would not be available in real-world deployment. This agent is used to generate expert trajectories. Secondly, a *sensorimotor agent* is trained to imitate these trajectories using only realistic inputs, such as raw camera data. The term ”cheating” originates from the privileged agent’s access to full state observations [12].

This approach is implemented in the CARLA simulator [25] for autonomous driving tasks. The privileged agent outputs driving waypoints using a ResNet18 backbone, which are then used to train a sensorimotor agent that operates under more realistic input conditions processed through a ResNet34 backbone. Training is conducted using both Behavior Cloning (BC) and the Data Aggregation (DAgger) [26] algorithm for comparison. BC is an offline imitation learning method, where the student policy is trained on a fixed dataset of trajectories generated by an expert policy. In contrast, DAgger is an on-policy method in which the expert is queried during training, allowing the dataset to be iteratively expanded with states encountered by the student. This two-stage process improves generalization and significantly reduces the sim-to-real gap. The architecture is illustrated in Figure 2.2. [12]

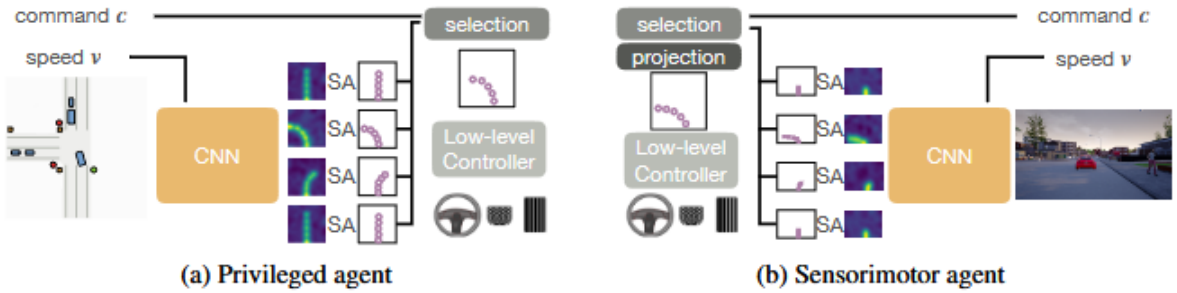

Figure 2.2: The agent architectures used by Chen et. al. **(a)** The privileged agent receives a complete perspective of the environment from which a heatmap is produced that is passed through a soft-argmax layer (SA), providing waypoints for all commands. The input commands selects one conditional branch. The waypoints produced by this, is then passed on to a low-level controller that outputs the actions. **(b)** The sensorimotor agent receives raw sensor inputs from a front-facing camera. It produces waypoints centered in the camera frame. Waypoints are then selected based on the command, projected into the vehicle frame and passed on to the low-level controller. Taken from [12].

The learning by Cheating paradigm provides a structured way to decouple task-level reasoning from raw perception, which Mortensen et al. [11] also draws inspiration from. Their work

adapts this two-stage approach to the RL domain, which is discussed in the following section. Furthermore, vision-based input modalities align more closely with the sensor configurations typically available in extraterrestrial missions. Consequently, RGB-D sensing is adopted in this thesis to provide both visual and spatial information.

2.2.2 Teacher-student

While the previous approach follows a traditional imitation learning pipeline, the method proposed by Mortensen et al. [11] introduces a two-stage RL framework that draws from the Learning by Cheating paradigm by Chen et al. [12]. This framework is specifically designed to enhance sim-to-real transferability in mapless navigation tasks, particularly within planetary rover missions where policies must operate robustly under severe environmental uncertainty and sensory noise.

The first stage focuses on learning an optimal policy for navigation in simulation using privileged, noise-free information. The teacher policy receives proprioceptive inputs, including the Euclidean distance to the goal $d(x,y)$ and the heading angle θ_{goal}. Additionally, the agent is given access to two exteroceptive modalities: a dense local heightmap o_t^d and a sparse, medium-range heightmap o_t^s. Each heightmap is processed by a dedicated Multi-Layer Perceptron (MLP)-based encoder, denoted e_d and e_s, each composed of two fully connected layers with sizes $[60,20]$ and LeakyReLU activations. These produce latent representations l_t^d and l_t^s, respectively. These latents are concatenated with the proprioceptive input and passed through a policy MLP head defined as:

$$MLP_{policy} : [512,256,128] \xrightarrow{LeakyReLU} \text{Action Mean} \tag{2.13}$$

The policy is optimized using PPO, modeling a Gaussian distribution $pi_\theta(a_t|o_t)$ to enable stochastic exploration. The architecture of the teacher framework is illustrated in Figure 2.3. In this stage, the teacher learns high-quality behavior by leveraging ideal, noise-free observations from simulation.

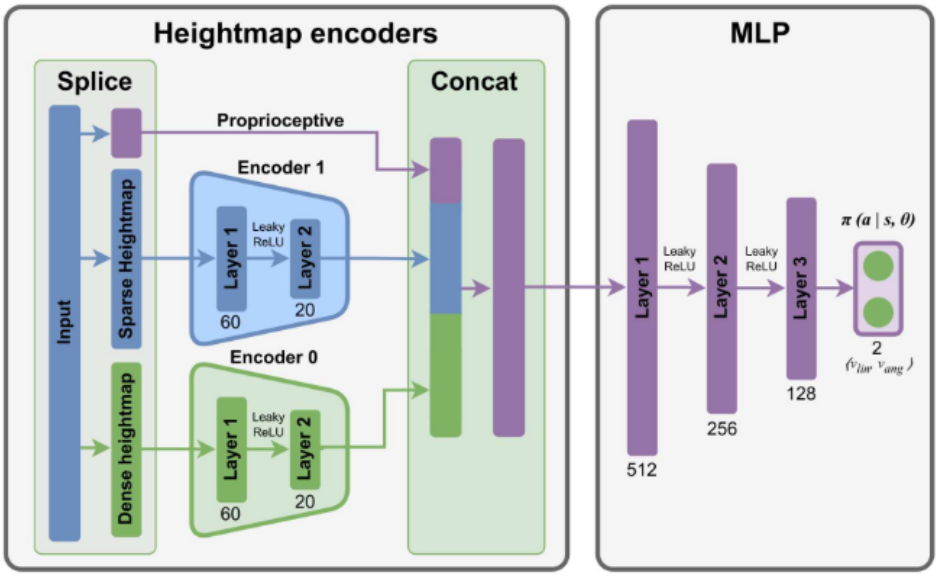

Figure 2.3: Illustration of the teacher framework. Taken from [11]

In the second stage, a student policy is trained to imitate the teacher's behavior using observations corrupted by realistic noise. The student architecture is showcased in Figure 2.4. The noise model includes perturbed data from proprioceptive and exteroceptive inputs using stochastic noise. This introduces Gaussian noise with multiple levels of deviation, fixed sensor offsets simulating drift, and occlusions implemented by masking out a subset of the heightmap data.

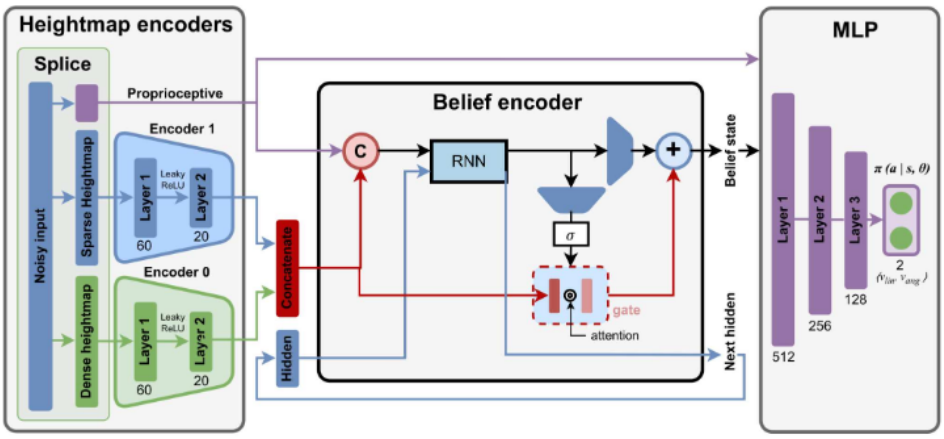

Figure 2.4: Illustration of the student framework. Taken from [11]

In order to alleviate this degradation in observations, the student network employs a belief encoder constructed around a Gated Recurring Unit (GRU) to form a temporal model. At each timestep, the perturbed inputs and the previous hidden state h_{t-1} are processed as:

$$x'_t, h_t = GRU(o_t^p, \tilde{l}_t^s, \tilde{l}_t^d, h_{t-1}) \tag{2.14}$$

To selectively modulate the influence of the latent GRU features, an attention gate AG is applied to x'_t. This gate is computed as:

$$AG = \sigma(e_\alpha(x'_t) \tag{2.15}$$

Where e_α is a 4-layer MLP that modulates how much of the original latent features are passed through. The belief state is computed by combining the GRU output with the gated latent features:

$$x_t = e_b(x'_t) + [\tilde{l}_t^s, \tilde{l}_t^d] \odot AG \tag{2.16}$$

Here, e_b is another 4-layer MLP with increasing layer sizes $[128, 256, 512, 1024]$, and $\odot$ denotes the element-wise Hadamard product. The final belief state x_t is concatenated with the proprioceptive input and passed through a final MLP head to produce deterministic action outputs:

$$\alpha_t = MLP_{student}(x_t, o_t^p) \tag{2.17}$$

The student policy is effectively trained in a supervised learning manner, minimizing the mean squared error (MSE) (squared L2-norm) between its predicted actions and the ground truth actions generated by the teacher policy:

$$\mathcal{L}_{student} = ||\alpha_t^{student} - \alpha_t^{teacher}||_2^2 \tag{2.18}$$

This two-stage training paradigm effectively decouples the acquisition of optimal behavior from the challenge of robustness to observation noise. The teacher is focused on exploiting perfect, simulated information to learn an optimal strategy, while the student learns to denoise and generalize from imperfect, noisy sensory inputs. The training process is conducted using the NVIDIA Isaac Sim [27] platform for physically realistic, GPU-accelerated simulation of planetary terrain, with support for high-fidelity heightmap rendering and terrain interaction. The agent itself is implemented using the *skrl* [28] reinforcement learning framework, which integrates with Isaac Gym [6]. The teacher policy is trained using on-policy optimization via PPO [20], leveraging this parallelized infrastructure to accelerate convergence.

As demonstrated in the paper, this architecture significantly improves sim-to-real transfer performance by leveraging both domain randomization and temporal modeling. However, the approach does not explore sensor decoupling, the concept of training the teacher and student with different sensor modalities or observation types. Particularly in the context of surface preparation and navigation on extraterrestrial bodies, this might improve deployment robustness since sensor configurations and capabilities differ significantly between Earth and other celestial bodies. Furthermore, in contrast to Chen et al., who apply the DAgger framework for behavioral cloning, Mortensen et al. train their GRU-based student policy using supervised learning. DAgger could potentially be used to improve the generalizability of agents trained with this approach.

2.2.3 RLRoverLab - A framework

Building on this approach, RLRoverLab is an open-source reinforcement learning suite developed by Mortensen and Bøgh [13], specifically designed for planetary rover simulation and training. The suite provides an accessible, modular framework for developing, training, and benchmarking RL algorithms. It is built on NVIDIA's Isaac Lab framework, leveraging Isaac Sim's GPU-accelerated physics engine, and, at the time of writing, supports the *skrl* RL library [28]. In addition, it includes pre-trained policies, terrain assets, and rover models to support rapid experimentation and evaluation.

The suite provides a comprehensive set of features, making it a versatile platform for reinforcement learning in planetary rover applications. It includes a range of terrain environments, such as hilly and obstacle-laden landscapes, allowing for evaluation under increasing complexity and partial observability. The framework comes with pre-integrated rover models, including a high-fidelity AAU rover inspired by NASA's Perseverance and the 3D-printable ExoMy rover, compatible with multiple control schemes. These models are illustrated in Figure 2.5

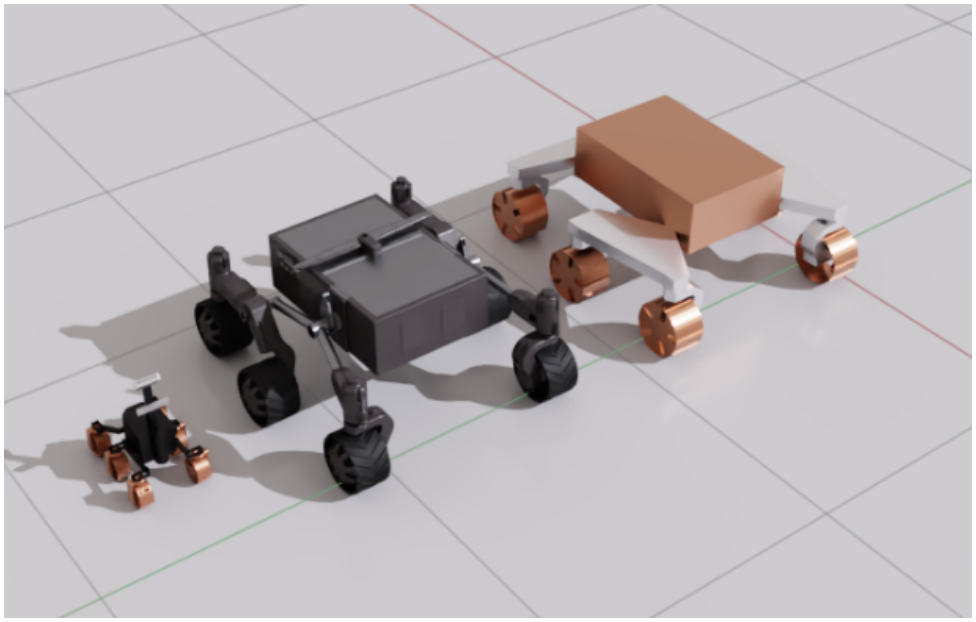

Figure 2.5: Depiction of the rover models currently available in RLRoverLab, from the left; Exomy [29], AAU rover, simplified AAU rover.

To support diverse robotic platforms, the suite currently offers Ackermann steering. The principles of Ackermann can be seen in Figure 2.6. Furthermore, it includes pre-defined task environments for both navigation and grasping, implemented through the OpenAI Gym API [30] that retains the comparability with the major RL frameworks. This collective functionality enables support for a variety of RL solutions. Lastly, this suite also includes benchmarking tools and pre-trained policies for standardized evaluation across algorithms, such as PPO [20], TRPO [21] and TD3 [24], thus enhancing reproducibility and fair comparison between learning strategies.

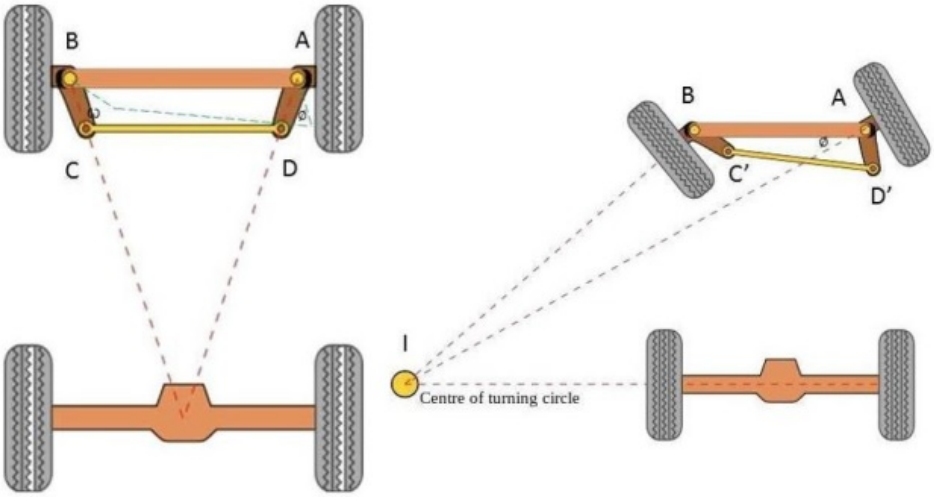

Figure 2.6: Illustration of the Ackermann steering principle. The key principle in Ackermann steering is that the geometry of the steering linkage (typically forming a trapezoid), here denoted **ABCD**, ensures that the front wheels follow concentric circular paths during a turn. This means each wheel is aligned to be perpendicular to a shared turning center point **I**, reducing tire slip and improving maneuverability [31]. Taken from [32]

This framework is supported by detailed documentation, including structured installation guides, implementation details, benchmarking results and development instructions, all hosted on its GitHub repository. Due to the suite's highly configurable and easy-to-use foundation, RL-RoverLab presents an ideal platform for experimental research in the sim-to-real transfer and sensor decoupling strategies and is therefore also considered an ideal platform for development in this project.

2.3 Related work

While the previously discussed approach contributes to improving sim-to-real transfer, a variety of alternative methods are also being actively explored. Nevertheless, the sim-to-real gap remains a critical challenge, particularly for autonomous navigation in extraterrestrial environments. Transferring policies from simulation to real-world deployment exposes them to numerous sources of error: environmental variability, sensor noise, and actuation inaccuracies often lead to degraded performance. As a result, recent research has focused on developing learning systems capable of generalizing across domains, enhancing navigation safety, and coping with unstructured or dynamic terrains [10].

Several studies have explored RL for navigation in simulated lunar environments, where terrain complexity and sensor uncertainty pose significant challenges. Yu et al. [33] propose an end-to-end path planning framework for lunar rovers, employing PPO due to its sample efficiency and implementation simplicity. The system architecture consists of a CNN encoding the data from a depth image, a lidar 2D-pointcloud and proprioceptive and target data. Then deep neural

networks (DNNs) is used to infer the non-linear approximation of value and policy functions to effectively realize the path planning of the lunar rover, Figure 2.7 showcases the architecture utilized in the implementation.

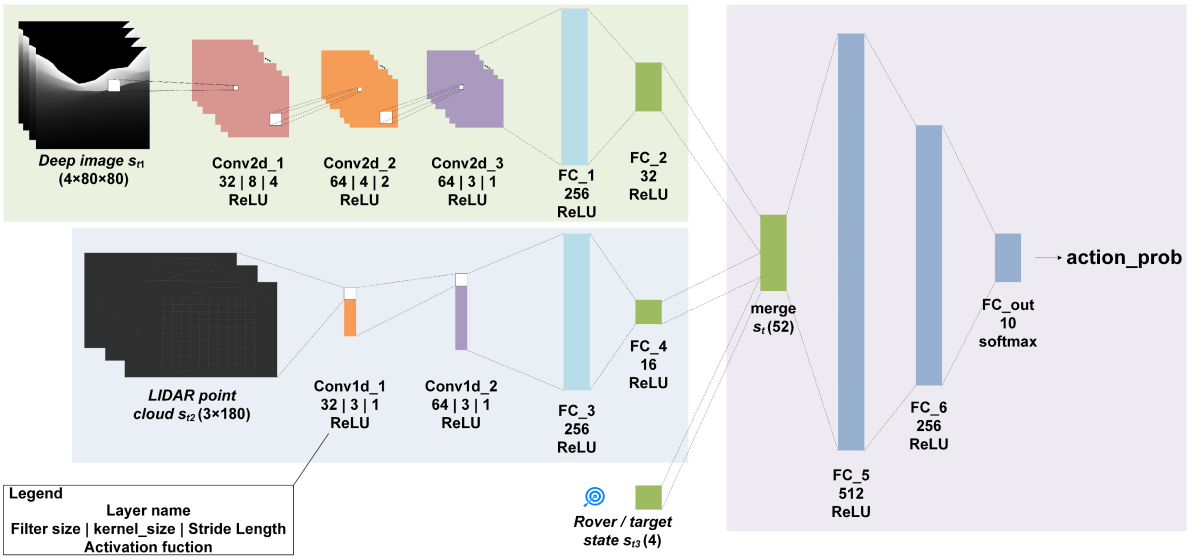

Figure 2.7: Illustration of the DNN architecture used by Yu et al. [33]

Yu et al. employs a reward function explicitly shaped by terrain slip and predicted slippage. It is trained on high-fidelity simulated lunar environments, and curriculum learning is used to gradually increase the complexity of the environments to improve adaptability in the agent. The purpose of this approach is to increase policy safety and generalizability across varying terrain topographies. Focusing on a similar issue is the SAC-based learning-based end-to-end navigation approach for planetary exploration rovers (LEN-PER) proposed by Feng et al. [34]. This approach uses a sequence of RGB and Depth images alongside proprioceptive, including wheel-terrain interactions, where the former is passed through individual CNNs and the latter is passed to an MLP encoder. All feature-terms are then concatenated into a feature vector then passed to the policy optimization module, as illustrated in Figure 2.8.

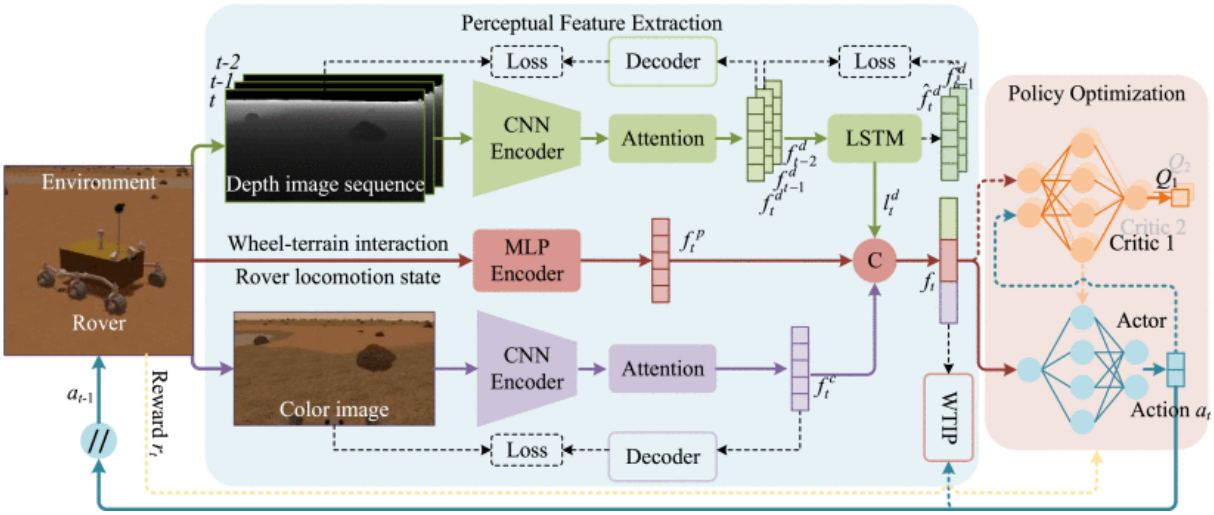

Figure 2.8: Illustration of the end-to-end navigation architecture used by Feng et al. The blue-highlighted network extracts perceptual features from depth images, color images, and proprioceptive states. The red module is the SAC-based policy that outputs control commands. Modules indicated by dashed lines are used only during training. Taken from [34].

The system utilizes the wheel-terrain interaction prediction (WTIP) module to predict slip rates on various terrains, to further enhance navigation capabilities and safety in extraterrestrial settings. A Mars-analog site is used to verify the approach's capabilities in handling the unstructured environments found on both the Lunar and Martian surfaces. The action space used for this solution is, similarly to Yu et al. linear and angular velocities. Continuing the extraterrestrial approach is a solution proposed by Park and Chung [35] focusing on a failure-safe motion planning method for a four-wheeled, two-steering lunar rover. The framework proposed, acts on cases such as actuator failure, ensuring continuous PointGoal navigation accuracy. The method utilizes Deep Q-Network (DQN) for path planning and prediction of and compensation for lost mobility. The system utilizes a discretized action space to reduce training time, allowing the selection of *forward*, *left* and *right*. For the reward function, a high focus was attributed to distance deviation from the path, angular deviation from the goal direction and the distance to the goal. The experiments were conducted in a 2D simulation evaluated in a flat test-bed.

An approach not specifically designed for extraterrestrial missions is a unknown rough terrain point-goal navigation strategy proposed by Zhang [36] aimed at urban search and rescue missions. This approach uses Depth data, estimated robot positions in an elevation map as inputs into a A3C-based pipeline. The pipeline uses a custom NN framework. Depth images and elevation maps (from ROS2 Octomap) are processed through identical convolutional and pooling layers. The elevation map output is combined element-wise with 3D orientation data, which is first passed through a fully connected layer and reshaped. The resulting feature map goes through an additional convolutional layer. All feature vectors are then concatenated and passed to an LSTM, followed by policy optimization. The reward function focuses on navigational suc-

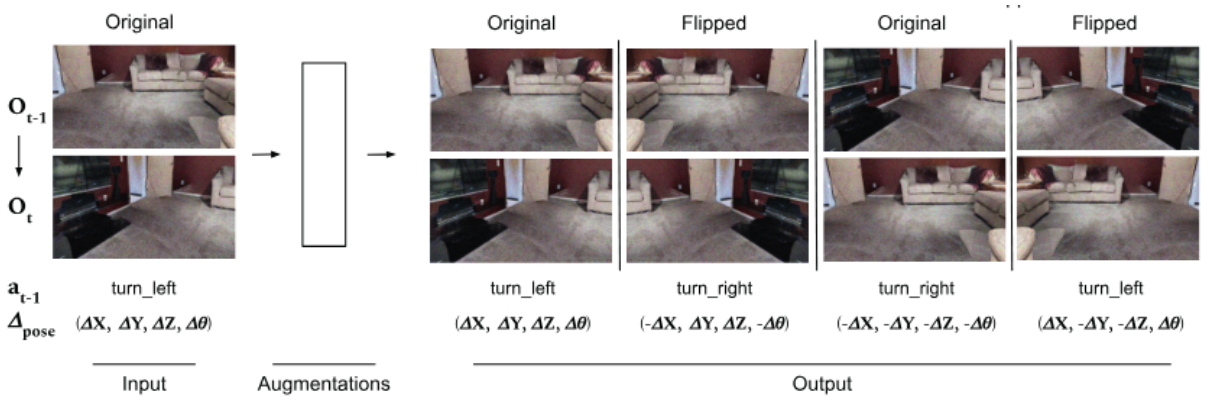

Figure 2.9: Illustration of the applied augmentation for each RGB-D input. Each input is both flipped and swapped to increase generalizability in environments where mapping is infeasible. Taken from [37]

cess and failure- and collision avoidance and time-limitation. A discrete action space was used during training, limited to forward/backward movement and left/right turns. The terrain-based approach randomizes the terrain, thus enhancing the generalization to varied terrains.

Beyond the more classical map-based strategies, Partsey et al. [37] explores whether mapping is necessary for point-goal navigation. In this context, Partsey et al. presents a drop-in approach where the agent is first trained with privileged localization ground truth from a GPS and compass sensor array. During evaluation, GPS and compass data are replaced with noisy RGB-D input, using only the depth channel for ego-motion estimation. The approach employs a navigation policy alongside a visual odometry (VO) module. The policy is implemented using a two-layer LSTM and a half-width ResNet50 encoder within a Decentralized Distributed PPO (DD-PPO) framework. The reward function encourages progress and successful goal completion. The discrete action space includes four actions: move forward 0.25m, stop, turn left, and turn right. To improve generalization during training, observation inputs are augmented by flipping and swapping the input images as illustrated in Figure 2.9.

Although not RL-based, Egan and Göktogan [38] propose a terrain classification system designed specifically for Mars-like environments. Their approach uses a CNN trained on RGB imagery from the Mars Analog MAAS Lab to perform semantic segmentation of traversable terrain classes. This output is fused with geometric obstacle data to generate traversability maps for rover path planning. The model is trained and validated on both synthetic and real Mars imagery, with a focus on on-board inference and computational efficiency, supporting autonomous decision-making without ground intervention.

Table 2.1: Overview of RL studies in robot navigation

Paper	RL algo.	Actions	Observations	Rewards	Citation
Zhang	A3C	Discrete movement commands	RGB-D, elevation map, IMU	Goal success, collision and slippage penalties	[36]
Park and Chung	DQN	Discrete steering and drive commands	Local states, proximity sensing	Mobility maintenance, slippage avoidance, target reaching	[35]
Yu et al.	PPO	Continuous velocity and angular control	RGB, Lidar	Progress reward, collision and slippage penalties	[33]
Feng et al.	SAC	Continuous velocity and angular control	RGB-D, wheel-terrain interaction, vehicle pose	Slippage reduction, hazard avoidance, goal reaching	[34]
Partsey et al.	DD-PPO	Discrete movement commands	RGB-D, visual odometry	Goal success, efficient travel distance	[37]
Egan and Göktogan	–	–	RGB images, depth maps	Traversability cost minimization	[38]

Across the reviewed literature, PPO or similar analogs are the most commonly adopted RL algorithm, primarily due to their balance between sample efficiency and training stability. Observation spaces are typically constructed around RGB-D inputs and elevation data, while action spaces tend to remain discrete to ensure robustness and interpretability. Reward functions are consistently shaped to promote goal completion and penalize collisions or unstable terrain traversal, reflecting a clear emphasis on safety and sim-to-real transferability. DRL is employed throughout, enabling the processing of high-dimensional inputs such as RGB and depth images through integrated neural network architectures. Accordingly, the design and integration of such networks must be examined more closely.

2.3.1 Neural networks for planetary navigation

Hu et al. [39] present a system for cluttered rough terrain navigation using elevation maps generated from fused RGB-D, IMU, and 3D LiDAR data. The elevation maps are processed through a convolutional encoder consisting of four convolutional layers with max-pooling. Orientation data is passed through a fully connected layer and merged with the elevation feature map, which is subsequently processed by a final convolutional layer. The resulting feature representation

is passed through an LSTM layer, followed by two fully connected layers that produce the navigation action and state-value estimate, as illustrated in Figure 2.10. This type of shallow network architecture ensures a lower computational cost, improving viability in real-time use on resource constrained platforms.

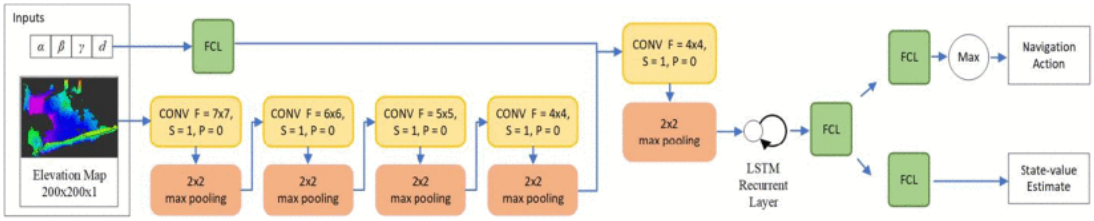

Figure 2.10: Illustration of the NN architecture used by Hu et al. Taken from [39]

Expanding on lightweight encoders, Zhang [36] proposes a system for point-goal navigation in rough 3D terrain that integrates depth images, elevation maps, and 3D orientation as input modalities. Elevation and depth inputs are processed independently through separate convolutional encoders, each consisting of four convolutional and four pooling layers. The elevation map features are fused element-wise with orientation features, which are first passed through a fully connected layer and reshaped. The resulting feature map is further processed by a shared convolutional layer before being merged with the depth image features. The combined representation is passed through an LSTM recurrent layer to capture temporal context, followed by fully connected actor and critic heads for policy learning. Despite the multi-branch design, the network remains relatively shallow, making it computationally efficient and suitable for onboard real-time deployment in resource-constrained field robots.

Yu et al. [33] present a multi-modal policy architecture designed for autonomous rover navigation, incorporating depth images, LiDAR point clouds, and rover/target state inputs. The network follows a three-branch design: depth images are processed via a 2D CNN with three convolutional layers and two fully connected layers, LiDAR point cloud data is passed through a CNN followed by two fully connected layers, and rover state vectors are fed directly into a fully connected layer. The resulting feature vectors from all branches are concatenated and processed through three additional fully connected layers to produce a distribution over actions using a softmax output. Feng et al. [34] similarly propose a sensor fusion architecture for safe rover navigation in deformable and uncertain terrain. The system processes depth image sequences and RGB frames through separate CNN encoders, each followed by attention modules. The depth stream incorporates a temporal dimension and passes features through an LSTM to capture short-term motion cues. In parallel, wheel-terrain interaction data and rover locomotion states are encoded using an MLP. All feature amps are concatenated and passed to the SAC policy optimization module. Furthermore, features are passed on to the WTIP to estimate slip.

In a more vision-centric system, Tang et al. [40] present Geo-Nav, combining monocular

RGB input with self-supervised depth predictions. A ResNet18 backbone, pre-trained on ImageNet, encodes RGB features, while a secondary encoder processes depth maps. These are fused through a custom Cross-Modality Pyramid Fusion (CPF) module. This architecture improves navigation in texture-poor or visually ambiguous scenarios, and real-time performance is demonstrated in simulation. While not RL-based, the system offers insights into fusion strategies applicable to DRL pipelines handling RGB-D data. Finally, Egan and Göktogan [38] propose a compact CNN for terrain classification trained on RGB imagery from the Mars Analog MAAS Lab. The output is combined with geometric obstacle detection to form traversability maps. The architecture is optimized for real-time operation on rover-class hardware, and although not RL-based, it addresses planetary navigation from a perception and autonomy standpoint, offering relevant insight into low-complexity, mission-suitable architectures. The main architectural characteristics of the reviewed systems are summarized in Table 2.2.

Table 2.2: Summary of NN architectures employed for navigation and visual processing.

Authors	**NN Type**	**Custom or Existing**	**Pre-trained**	**Size/Notes**	**Citation**
Hu et al.	CNN + LSTM	Custom	No	Lightweight encoder with four CNN + one FC + LSTM layers	[39]
Zhang	Dual CNN + LSTM + FC	Custom	No	Independent elevation/depth branches; shallow design for real-time use	[36]
Yu et al.	CNN + MLP	Custom	No	Three-branch structure: depth, LiDAR, and rover state input	[33]
Feng et al.	CNN + MLP + LSTM + Attention	Custom	No	Multi-stream encoder with temporal depth stream; attention and slip estimation	[34]
Tang et al.	ResNet18 + Fusion	Existing + Custom	Yes	ResNet18 backbone + custom cross-modality fusion; RGB-D processing	[40]
Egan and					

Continued on next page

Authors	NN Type	Custom or Existing	Pre-trained	Size/Notes	Citation
Göktogan	CNN	Custom	No	Compact CNN for RGB terrain classification; optimized for rover hardware	[38]

Across these approaches, the design of neural network architectures is closely aligned with the structure and complexity of the input modalities. Lightweight CNNs are often favored for RGB-D processing in resource-constrained settings, while dual-branch and fusion-based designs support multi-modal integration where required. These architectural choices reflect the practical challenges of processing high-dimensional data within RL pipelines intended for real-world or planetary deployment.

2.3.2 Summary

This section has reviewed relevant RL approaches and architectural strategies for autonomous navigation in rough and planetary terrain. Among the studies surveyed, PPO appears as the most widely adopted RL algorithm, selected in the majority of cases for its practical balance between implementation simplicity, sample efficiency, and stable convergence. Based on this, PPO is a viable option for implementation of this solution. From an architectural standpoint, convolutional encoders are the most common method for processing high-dimensional visual inputs, particularly RGB and depth inputs. Several systems utilize ResNet-based backbones to encode RGB-D or RGB-derived observations, either trained from scratch or initialized from pre-trained weights. Although these models vary in depth, ResNet18 appears frequently due to its balance between feature extraction capacity and computational efficiency. In this thesis, a pre-trained ResNet18 encoder is adopted to reduce training time, benefit from general-purpose visual features, and align with methods that demonstrated strong performance in simulation-based navigation tasks. Together PPO and ResNet18 form a stable foundation for processing RGB-D input in a RL framework.

3 Problem Formulation

Based on the preliminary research on relevant literature and the established challenges related to the issue, objectives can be established for the final proposed solution. Due to the inherent ambiguity of the problem in this thesis, instead of requirements, objectives are formed for the proposed solution. Thus, this chapter presents the final problem statement and the derived objectives based on the insights gained through the previous research.

Final problem statement
Building on the analysis of the theoretical foundation and implementation of the teacher-student framework, as well as insights from alternative approaches aimed at mitigating the sim-to-real gap, the following problem statement is formulated:

> *"How does employing heightmap observations for the teacher policy and RGB-D input for the student policy within a DAgger-based teacher-student framework influence learning efficiency, simulated performance, and sim-to-real transferability in reinforcement learning for mobile robot navigation?"*

3.1 Project Objectives

Based on the research question and prior work, the following objectives are defined. Each is further divided into sub-objectives to provide a clearer structure for development and evaluation. The models introduced are based on available hardware platforms at Aalborg University.

1. **Integrate existing mobile robots into the RLRoverLab framework.**
   - 1.a Integrate the RobuROC4, Summit XL, and Leo Rover as new assets.
   - 1.b Calibrate each asset's physical properties, including kinematic constraints.
   - 1.c Implement a Skidsteering controller for the new assets.
   - 1.d Validate asset integrity to ensure realistic behaviors in simulation.
2. **Train one or more new assets with the existing heightmap approach using PPO.**
   - 2.a Further validate the integrity of the models by training with the existing heightmap approach in RLRoverLab.
   - 2.b Verify and benchmark performance metrics to ensure effective navigation in existing environments within the RLRoverLab simulation.
3. **Implement and integrate RGB-D observations in existing RLRoverLab architecture.**

3.a Extend the RLRoverLab simulation environment by integrating RGB camera-based observations into the RL training pipeline.

3.b Utilize the choosen ResNet18 model to process images semantically for obstacle avoidance and terrain type recognition.

4. **Train and test one or more agents based on new RGB-D sensor observations.**

4.a Train one or more RL agents using the RGB-D adapted framework.

4.b Train one or more RL agents using noisy RGB-D inputs based on the adapted framework for comparisson.

4.c Evaluate agents through the simulated scenarios, comparing their performance on tasks, including obstacle avoidance and terrain navigation.

5. **Train one or more agents with an adapted teacher-student framework.**

5.a Extend the RLRoverLab simulation environment to support DAgger-based teacher-student training, based on the learning by cheating method proposed by Chen et al. [12]

5.b Train one or more agents using the heightmap-trained agents as a teacher and the student having noisy RGB-D sensor inputs.

5.c Evaluate the trained agents in different tasks, such as obstacle avoidance and pointgoal navigation, and compare with agents trained solely on RGB-D inputs and heightmap inputs.

6. **Evaluate the robustness and real-world transferability of RL agents trained using the teacher-student approach.**

6.a Evaluate the performance of trained policy or policies through deployment on their corresponding physical rover platform(s).

6.b Test the navigation and obstacle avoidance capabilities of the policy in a set environment without prior knowledge.

6.c Based on this experiment, propose adjustments to improve the performance of simulation-trained RL agents to practical lunar missions.

4 Implementation

Building upon the objectives and system design outlined in the previous chapters, this chapter presents the practical implementations in the RL framework RLRoverLab. The implementation presented in this Chapter mainly builds on top of the existing RLRoverLab suite. The focus points covered here is the modeling and integration of new models including development of a new control scheme and tuning of parameters, implementation of the RGB-D input modality into the framework and development of teacher-student framework with sensor modality separation. The following sections cover these focus points sequentially. The final implementation can be found at: https://github.com/Blueguardian/RLRoverLab_fork.

4.1 Model implementation

Although RLRoverLab provides a few preconfigured robotic models, these represent only a limited subset of potential agents. For the purpose of this thesis and in regards to Objective 1a, four additional robotic models were integrated into the framework. These include two commercially available platforms; The Leo Rover by Fictionlab and the RB-Summit by Robotnik. In addition the RobuROC4 by Robosoft, which is no longer in production is included in both a full-featured and a simplified representation, the latter being derived from a prior project [41].

The newer models were obtained in Universal Robot Description Format (URDF) from their respective Github repositories (Leo rover and RB-Summit) and imported into Isaac Sim using the built-in URDF importer. This tool converts URDF files into Universal Scene Description (USD) assets, which are compatible with the RLRoverLab pipeline. For the RobuROC4, no public URDF is available, thus a new model was created, supplemented by the simplified version. All four models are shown in Figure 4.1.

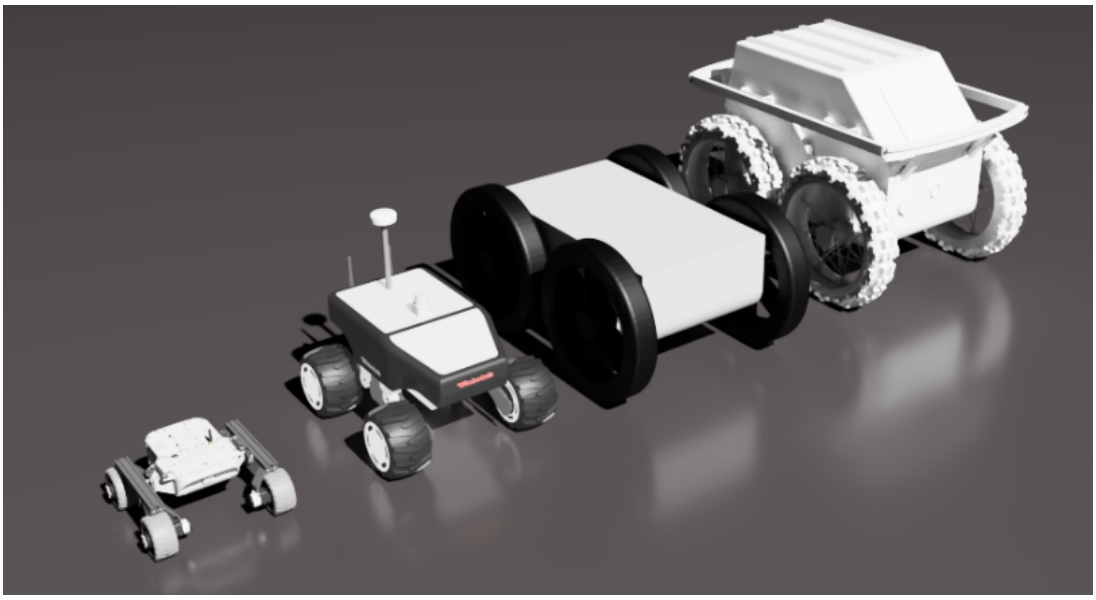

Figure 4.1: Depiction of the used models. From the left; Leo rover, RB Summit XL, RobuROC4 simplified, RobuROC4.

Before import, all URDF files are modified to remove unsupported or unnecessary elements, such as gazebo-specific tags or redundant collision definitions. The URDF import utility in Isaac Sim, depicted in Figure 4.2 is further used in choosing a drive type and configuration along with other parameters. For all models, the following import settings are applied: `moveable base` is enabled, the joint configurations are set to use `natural frequency`, drive types are defined as force drives and all joints are configured for velocity control.

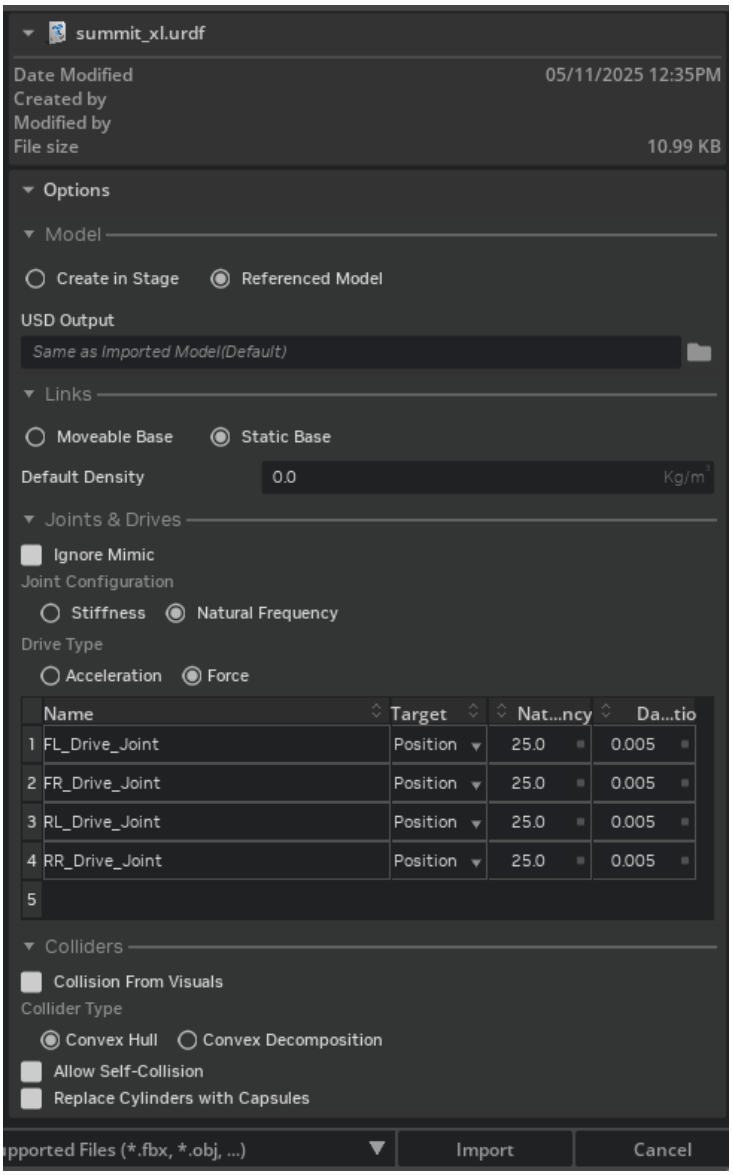

Figure 4.2: Depiction of the URDF import user interface.

Since the new models differ kinematically from those already present in RLRoverLab, being four-wheel skid-steered platforms, a new control scheme is required. The following section introduces the skid-steering control structure developed for this purpose.

4.1.1 Controlling the new models

To support the control of these models within RLRoverLab as required by Objective 1c, a new `ActionTerm` is implemented. It builds upon the joint and drive detection logic from the existing Ackermann Steering ActionTerm, but introduces new parameters and control, specific to skid-steering

In real-world applications, the motion of skid-steered robots depends heavily on terrain friction.

Since Isaac Sim does not expose floor or terrain friction coefficients directly to the agent, and learning them through training is effectively more robust, the control is simplified to a differential drive form. This approach uses linear and angular velocity commands from the policy to compute individual wheel velocity as follows:

$$\omega_L = \frac{v - \frac{b}{2} \cdot \omega}{r}, \quad \omega_R = \frac{v + \frac{b}{2} \cdot \omega}{r} \tag{4.1}$$

where:

- ω_L and ω_R are the angular velocities of the left and right wheels, respectively.
- v is the input linear velocity of the robot's center.
- ω is the input angular velocity.
- b is the track width, i.e. the distance between the left and right wheels.
- r is the radius of the wheels.

The corresponding Python implementation used in RLRoverLab is shown in Snippet 4.1.1

```python
def skid_steer_simple(vx, omega, cfg, device):
    track_width = cfg.track_width
    wheel_r = cfg.wheel_radius
    vel_left = (vx - omega * track_width / 2) / wheel_r
    vel_right = (vx + omega * track_width / 2) / wheel_r
    wheel_vel = torch.stack([vel_left, vel_left, vel_right, vel_right], dim=1)
    return wheel_vel
```

Snippet 4.1.1: Snippet of the Skidsteering control

The SkidSteering controller requires the wheel radius and track width, all of which are specified per model, including the existing; scale and offset that are applied to each action. These parameters are presented in the following section.

4.1.2 Implementation in RLRoverlab

The original implementation of RLRoverLab requires manual configuration and Gymnasium registration for new assets. To streamline this process and promote a more modular and centralized architecture, an automated integration system is introduced.

This system uses Python's `type` metaclass in conjunction with a set of YAML configuration files, `robot_default`, `sensors_default`, `terrain_default` and `training_default`. A python script reads and parses these files and extracts parameters for each asset. these are then

used to define a `ArticulationCfg` object, which holds the USD path and other model-specific settings described in more detail in subsection 4.1.3.

The configuration data is compiled into a dictionary and passed to the metaclass `type` to dynamically generate parts of environment classes. A fragment of this implementation is shown in Snippet 4.1.2

```
    def _generate_cfgs(self):
        ...
        robot_cfg = self._load_yaml(folder / "configs" / "robot_default.yaml")
        training_cfg = self._load_yaml(folder / "configs" / "training_default.yaml")
        # Generate or fetch class name
        class_name = self._class_name(folder, robot_cfg)

        attributes = {
            "__doc__": f"Configuration for {folder.name} rover environement.",
            "__post_init__": post_init_gen(robot_cfg,
             ↪  self._articulation_cfg(robot_cfg), self._action_cfg(robot_cfg))
        }

        cls = type(class_name, (self.parent_class, self.__class__), attributes)
        cls = configclass(cls)
        env_cfgs[class_name] = cls

        ...
        return env_cfgs
```

Snippet 4.1.2: Part of the implementation of the configuration generator for assets in RLRoverLab

To register the generated environments with Gymnasium, a custom registar class is used. It maps each environment class to a Gymnasium ID based on folder names or task names defined in the configuration. This functionality is illustrated in Snippet 4.1.3.

```
class GymEnvRegistrar:
...
    def register_envs(self):
            env_id = f"{env_folder.name}-v0" if not "task_name" in learning_config
            ↪  else learning_config.get("task_name")+"-v0"

            gym.register(
                id=env_id,
                entry_point='rover_envs.envs.navigation.entrypoints:RoverEnv',
                disable_env_checker=True,
                kwargs={
                    "env_cfg_entry_point": env_config_class,
                    "best_model_path": Path(env_folder,
                    ↪  "policies/best_agent.pt").absolute().as_posix(),
                    "get_agent_fn": get_agent,
                    "skrl_cfgs": skrl_configs,
                    },
                )
        ...
```

Snippet 4.1.3: Part of the implementation of the gymnasium reigstration class, which registers each new configuration into gymnasium.

To reduce the risk of missing configuration files when adding new models, a directory watchdog monitors the asset directory. When a new folder is added, it automatically copies in the required configuration files, if they are not already present. The method utilized in this functionality is showcased in Snippet 4.1.4

```
def on_created(self, event):
    if not event.is_directory:
        return

    config_path = Path(event.src_path) / "configs"
    if not config_path.exists():
        try:
            shutil.copytree(self._CONFIGS_DIR, config_path, dirs_exist_ok=True)
            os.chown(config_path, self._CONFIG_DIR_STAT.st_uid,
             ↪  self._CONFIG_DIR_STAT.st_gid)
            for file in config_path.rglob("*"):
                os.chown(file, self._CONFIG_FILES_STAT.st_uid,
                 ↪  self._CONFIG_FILES_STAT.st_gid)
        except Exception as e:
            ...

    self.paths_dirs.append(event.src_path)
```

Snippet 4.1.4: Part of the implementation of the directory watchdog that ensures default configuration files are copied when a new asset is added.

From here, each model can then be tuned with the parameters found through testing directly in Isaac Sim. These are presented in the following section.

4.1.3 Model parameter tuning

To ensure appropriate physical behavior during simulation and to accomplish Objective 1b, the drive parameters of each model are tuned manually in the Isaac Sim user interface. Parameters such as velocity limits, effort limits, damping, and stiffness are iteratively and empirically adjusted until satisfactory performance is observed. The desired performance includes the capability to execute turn-in-place while maintaining robust traversal across sloped terrain. Additionally, motion dynamics should be smooth and adequately damped to minimize body oscillations and prevent instability or toppling. Once validated, these values are inserted directly into the configuration files used by RLRoverLab for each model. These values are presented below for each model, alongside the SkidSteering control parameter values.

Table 4.1: Leo Rover – Drive and Control Parameters

Drive Parameters		Control Parameters	
Effort limit	20	Scale	[3.0, 3.0]
Velocity limit	10	Offset	[0.01, 0.0]
Damping	9000.0	Track width	0.359 m
Stiffness	5.0	Wheel radius	0.065 m

Table 4.2: RB Summit XL – Drive and Control Parameters

Drive Parameters		Control Parameters	
Effort limit	100	Scale	[3.0, 4.0]
Velocity limit	12	Offset	[0.01, 0.0]
Damping	9000.0	Track width	0.470 m
Stiffness	15.0	Wheel radius	0.1175 m

Table 4.3: RobuROC4 – Drive and Control Parameters

Drive Parameters		Control Parameters	
Effort limit	800	Scale	[0.6, 1.0]
Velocity limit	6	Offset	[0.01, 0.01]
Damping	7000.0	Track width	0.690 m
Stiffness	5.0	Wheel radius	0.280 m

Table 4.4: RobuROC4 (Simplified) – Drive and Control Parameters

Drive Parameters		Control Parameters	
Effort limit	800	Scale	[0.6, 0.9]
Velocity limit	10	Offset	[0.01, 0.0]
Damping	6500.0	Track width	0.690 m
Stiffness	5.0	Wheel radius	0.280 m

The testing and evaluation of the implemented models in their current form are presented in section 5.2. In alignment with the focus of this thesis and the objectives outlined in section 3.1, RGB-D functionality with ResNet18-based feature extraction is required. The following section presents the implementation of this functionality within the RLRoverLab framework.

4.2 Introducing RGB-D as sensor input

To support RGB-D input for the newly added assets in accordance with Objectives 3a and 3b, a simulated RGB-D camera model is integrated into the agent's sensor configuration. In parallel, a new policy model, incorporating a ResNet18-based encoder, is implemented as part of the existing policy collection to handle the high-dimensional visual input. The following sections detail the implementation and configuration of both components.

4.2.1 Camera integration

The camera model used during training and simulation is implemented as an instance of Isaac Lab's *Tiled_Camera*, which is optimized for manager-based environments such as the one used in RLRoverLab [42]. To reflect realistic deployment, the camera is front-mounted with a wide field of view, ensuring obstacle visibility during navigation. Its configuration is shown in Snippet 4.2.1.

```
tiled_camera: TiledCameraCfg = TiledCameraCfg(
        prim_path="{ENV_REGEX_NS}/.*/Main_Body/Camera1",
        depth_clipping_behavior='max',
        data_types=["rgb", "depth"],
        width=100, height=100,
        spawn=sim_utils.PinholeCameraCfg(
            focal_length=18.00, focus_distance=400.0,
            horizontal_aperture=32.000, clipping_range=(0.1, 10.0)
        ),
        offset=TiledCameraCfg.OffsetCfg(
            pos=(0.55, 0.0, 0.6),
            rot=(0.5, 0.5, -0.5, -0.5),
            convention="parent"
        )
    )
```

Snippet 4.2.1: Implementation of the camera model in the `RoverSceneCfg`

The camera parameters are selected with consideration to both memory usage and visual information, based on empirical testing for multi-agent training scalability. While the *Tiled_Camera* supports a range of rendering outputs, including direct image segmentation, only the *rgb* and *depth* modalities are used, for realism. An example of the output from the raw camera data can be seen in Figures 4.3 and 4.4.

Figure 4.3: An example of the raw RGB output of the camera model

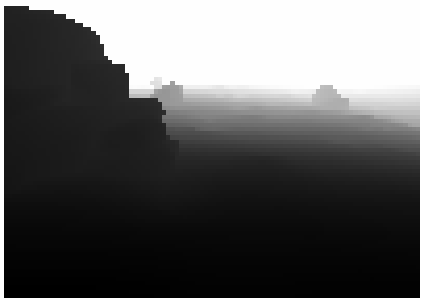

Figure 4.4: An example of the raw depth output of the camera model

As seen, these outputs retain sufficient information for spatial awareness, despite the lower resolution. However, because simulation-based sensors are deterministic and lack measurement noise by default, a set of sensor noise models is integrated: a Gaussian noise model for RGB data and a Redwood-inspired noise model for depth, to simulate noise as seen in real-world

sensors. While Isaac Lab provides a Gaussian noise model, it is aimed for normalized images, thus a separate class is implemented to account for RGB *uint8* format, with the same base functionality. For depth a a separate model is implemented to simulate depth sensor noise, inspired by the characteristics of the Redwood dataset, the introduced model is defined as follows. Let $\hat{d}$ denote the noisy depth value, d' the jittered depth, $\eta \sim \mathcal{N}(0,1)$ a standard normal variable, and s, α, and m denote the scale factor, exponential rate, and noise multiplier, respectively. Then:

$$\hat{d} = d' + m \cdot s \cdot \left(e^{\alpha d'} - 1\right) \cdot \eta \tag{4.2}$$

Jittering refers to the spatial perturbation of the sampling location in the image. Rather than using the original pixel at (x,y), a nearby pixel within a ± 2 window is randomly selected to simulate spatial uncertainty. The neighboring pixel coordinates are clamped to the image size to account for edge-cases. Jittering is introduced to simulate calibration errors, depth discontinuities and temporal aliasing. The parameters for both the Gaussian and Redwood-inspired models are defined in Tables 4.5 and 4.6 respectively.

Table 4.5: Gaussian Noise Model Parameters (RGB)

Parameter	**Value**
Mean	5.0
Standard deviation	10.0
Operation	add

Table 4.6: Redwood-Inspired Noise Model Parameters (Depth)

Parameter	**Value**
Scale	0.1
Exponent	0.3
Max depth	10.0 m
Noise multiplier	1.0
Operation	add

The mean and standard deviation parameters for the Gaussian noise model are chosen to also simulate mild oversaturation effects due to lighting conditions. The parameters used in depth noise modeling are chosen to simulate a similar curve as observed by Teichman et al. [43] for depth dependant noise. The resulting curve from the depth noise parameters yields a depth noise standard deviation curve as seen in Figure 4.5

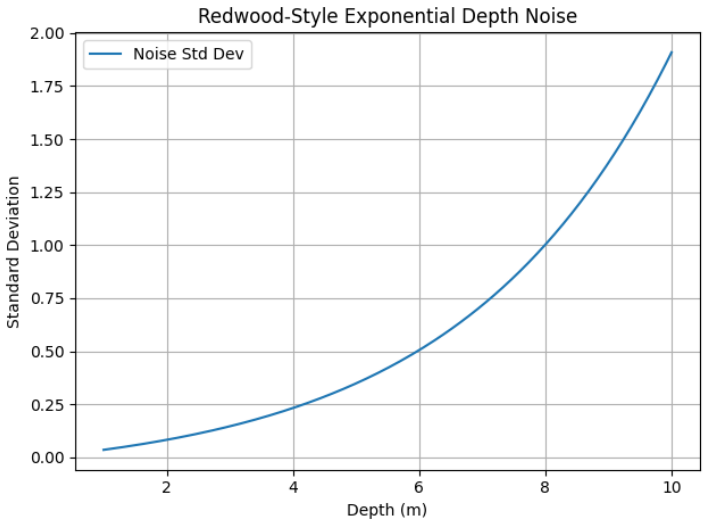

Figure 4.5: Exponential noise standard deviation curve resulting from the parameters used in the depth model, a sample is randomly picked from this and added to the rendered depth from the camera model.

The resulting images after applying this noise, analog to those seen as raw outputs of the RGB and depth, can be seen in Figures 4.6 and 4.7.

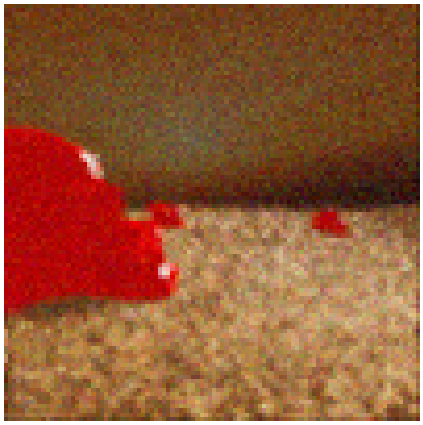

Figure 4.6: An example of the noisy RGB output of the camera model as a result of added noise

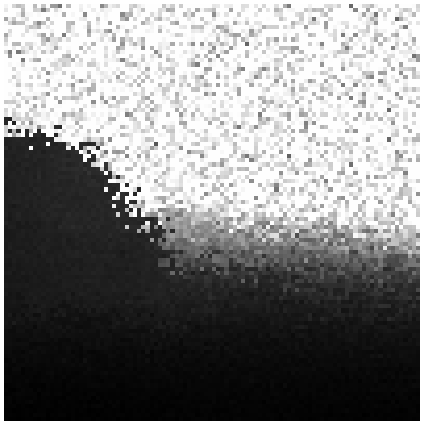

Figure 4.7: An example of the noisy depth output of the camera model as a result of added noise

To handle the outputs of the camera, an observation processing function is implemented for both depth and RGB modalities. This function retrieves and preprocesses the rendered data during simulation. Both inputs are converted to channel-first (BCHW) format to ensure compatibility with neural network processing, and RGB data is converted to a tensor if initially provided as a NumPy array. The implementation can be seen in Snippet 4.2.2.

```python
def image(env, sensor_cfg, data_type):
    sensor = env.scene.sensors.get(sensor_cfg.name)
    output = sensor.data.output.get(data_type)
    if data_type == "rgb":
        try:
            if not isinstance(output, torch.Tensor):
                output = torch.from_numpy(output)
            output = output.to(torch.float32)
            output = output.permute(0, 3, 1, 2)  # BCHW
            return output
        except Exception as e:
            print(f"[ERROR RGB] Failed to process RGB output: {e}")
            return torch.zeros((1, 3, 100, 100), dtype=torch.float32,
             ↪  device=env.device)
    elif data_type == "depth":
        try:
            depth = sensor.data.output["depth"]
            depth = depth.permute(0, 3, 1, 2)
            return depth
        except Exception as e:
            print(f"[ERROR] Failed to handle depth output: {e}")
            return torch.zeros((1, 1, 100, 100), dtype=torch.float32,
             ↪  device=env.device)
    else:
        print(f"[ERROR] Unknown data_type: {data_type}")
        return torch.zeros((1, 3, 100, 100), dtype=torch.float32, device=env.device)
```

Snippet 4.2.2: Implementation of the observation term function to retrieve data from the camera.

The data is subsequently passed through the observation manager to the policy model. To interpret this high-dimensional visual input, a policy model is implemented using the required ResNet18 network in an encoder for the RGB modality.

4.2.2 ResNet18 Policy Model

To ensure architectural parity with the approach used by Mortensen et al. [11], a comparable policy model is implemented within the existing framework. While the original model from Mortensen et al. is based on a bimodal input structure with two separate heightmaps providing spase and dense spatial data, the implementation in RLRoverLab instead uses a single encoder processing a single dense heightmap. This serves as the baseline implementation, shown in Figure 4.8.

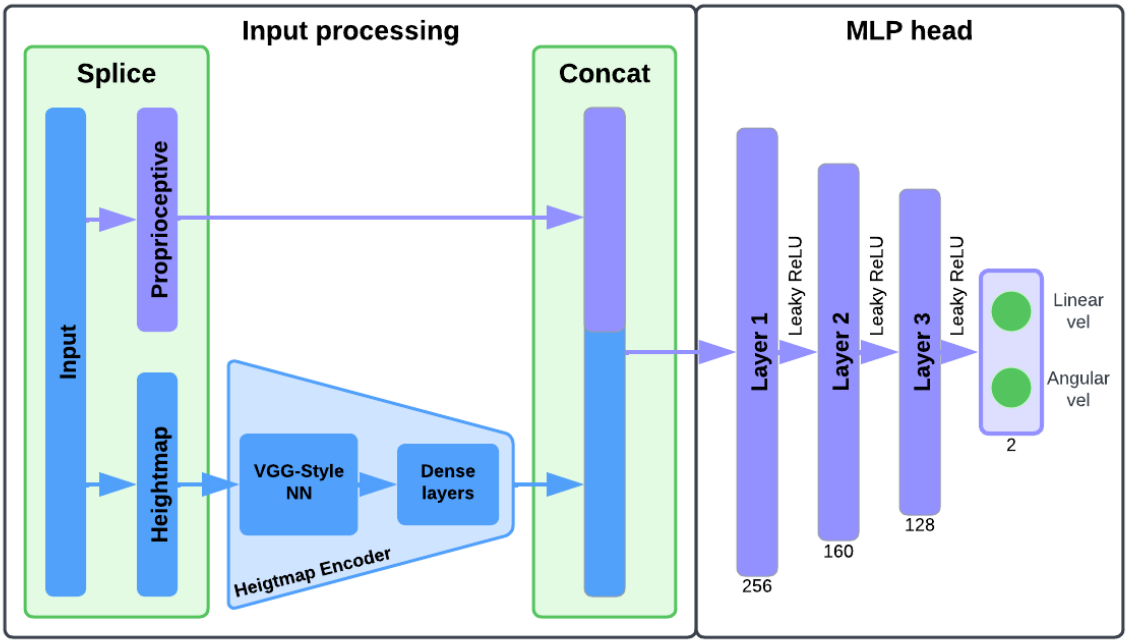

Figure 4.8: An illustration of the heightmap-based policy model. The observation input is divided into proprioceptive data and a heightmap image. The heightmap is processed using a VGG-style encoder consisting of convolutional and fully connected layers. The resulting feature vector is concatenated with the proprioceptive input and passed through an MLP head, which outputs linear and angular velocity commands.

For RGB-D input, a similar architecture is employed but extended to include two encoders; one for RGB input, using ResNet18 with pre-trained ImageNet-V1 weights, and utilizing the existing heigtmap encoder for depth. The architecture is illustrated in Figure 4.9.

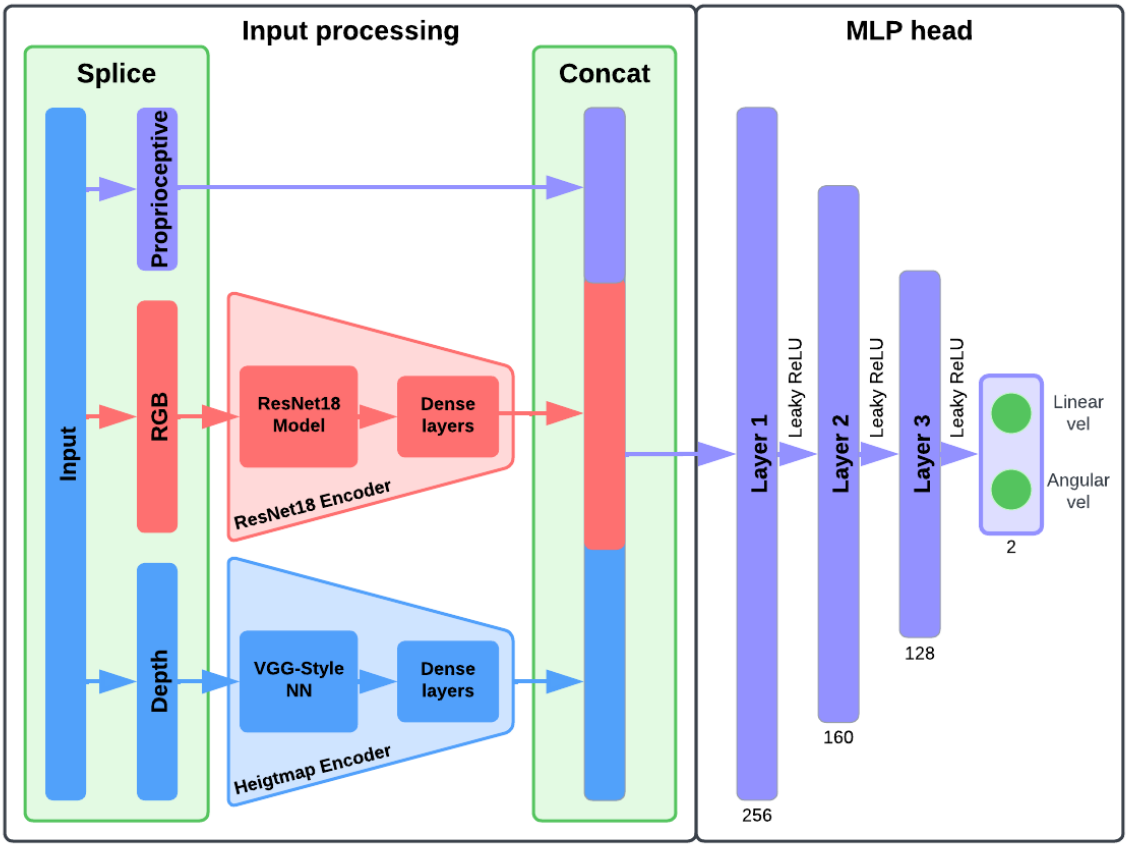

Figure 4.9: An illustration of the RGB-D-based policy model. The observation input is divided into proprioceptive data, a depth image, and an RGB image. The depth input is processed using the same encoder as the heightmap, while the RGB input is processed using a ResNet18 encoder followed by similarly structured dense layers. The resulting feature vectors are concatenated with the proprioceptive input and passed through an MLP head, which outputs linear and angular velocity commands.

As illustrated, the policy model integrates both RGB and depth encoders, with the resulting feature vectors concatenated alongside proprioceptive input and passed through an MLP head. The MLP structure is retained from the original heightmap-based policy model to maintain architectural comparability. For the encoders, the heightmap encoder is reused for depth encoding due to the similarity in spatial input, while the RGB encoder utilizes a ResNet18 model from the *Pytorch* library [44]. The ResNet18 model uses pre-trained weights from ImageNet-V1, eleminating the need for initial perception training. Furthermore, the model is frozen to simplify training. The neural network used in depth encoding employs a leaky Rectified Linear Unit (leaky-ReLU) as activation function as in the original implementation. Similarly the RGB encoder utilizes leaky-ReLU for activation in the fully connected layers. The implementation of this RGB encoder can be seen in Snippet 4.2.3.

```python
class ResnetEncoder(nn.Module):
    def __init__(self, in_channels, encoder_features=[80, 60],
    ↪  encoder_activation="leaky_relu"):
        super().__init__()
        weights = ResNet18_Weights.DEFAULT
        self.resnet = resnet18(weights=weights, progress=True).eval()
        self.transform = weights.transforms(antialias=True)
        self.flatten = nn.Flatten()
        for param in self.resnet.parameters():
            param.requires_grad = False
        self.output_encoder = nn.ModuleList()
        in_features = 1000
        for feature in encoder_features:
            self.output_encoder.append(nn.Linear(in_features, feature))
        self.output_encoder.append(get_activation(encoder_activation))
            in_features = feature
        self.out_features = encoder_features[-1]
    def forward(self, x):
        x_transformed = torch.stack([self.transform(x[i]) for i in
        ↪  range(x.shape[0])], dim=0)
        with torch.no_grad():
            x = self.resnet(x_transformed)
        x = self.flatten(x)
        for layer in self.output_encoder:
            x = layer(x)
        return x
```

Snippet 4.2.3: Implementation of the RGB encoder for the policy model.

To accommodate the multiple modalities introduced by the RGB-D input, the observation space of the environment is restructured from a flattened format to a dictionary-based format using Gymnasium's Dict space. This change allows the policy model to access and process RGB, depth, and proprioceptive data independently, without requiring manual slicing of observation tensors. Each modality is retrieved using distinct observation terms defined in the observation manager, enabling improved clarity in data handling. With the RGB-D observation pipeline fully integrated into the simulation framework, the next step involves leveraging this setup within an imitation learning paradigm. An imitation framework, based on the approach used by Chen et al. [12], is implemented to enable student policies using RGB-D input to learn from expert demonstrations derived from heightmap-based policies.

4.3 Teacher-student framework implementation

In accordance with Objective 5, the simulation pipeline is extended to support a teacher-student training paradigm. This setup is inspired by the framework proposed by Chen et al. [12], in which a privileged agent with access to ground-truth information trains a vision-based agent through imitation. This pipeline is implemented as a separate package which intertwines with the original implementation through the `train.py` script. The approach used in this implementation decomposes the learning process into two stages: first, training an expert agent, in this case on heightmap observations, and secondly, using this expert to supervise a student agent trained on high-dimensional, noisy RGB-D input, by labeling the student's observations with corrective actions. A general conceptual illustration of the teacher-student framework can be seen in Figure 4.10.

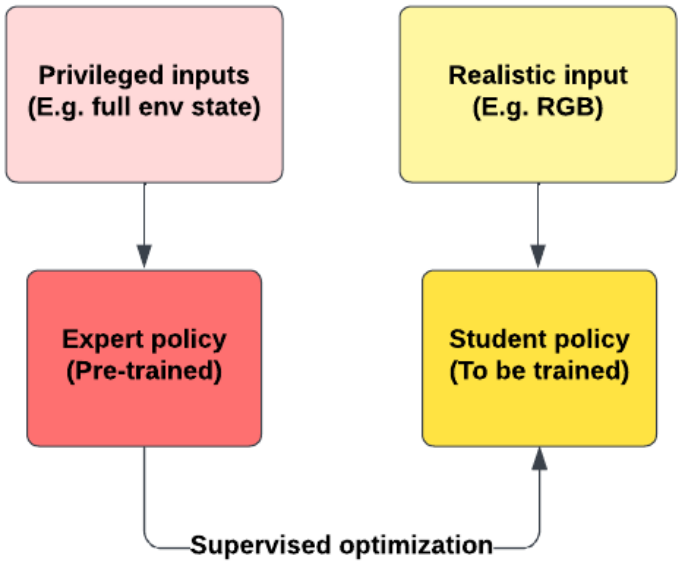

Figure 4.10: Conceptual illustration of the teacher-student framework. The teacher policy is pre-trained on privileged observations and is used to supervise the optimization of the student policy, which is trained on realistic inputs such as noisy RGB data.

To implement this imitation-based supervision approach, a DAgger algorithm is employed using the Imitation library, based on Stable Baselines3 (SB3), [45], [46]. For context, DAgger improves upon regular Behavior Cloning (BC) by addressing the sensitivity to distributional shift. Whereas BC relies on a fixed or static dataset of expert demonstrations collected in isolation, DAgger allows the student to act in the environment while querying the expert for correct actions. These interactions are aggregated into an expanding dataset, alloing the student to adapt to an expanding variety of states, resulting in more robust behavior transfer. The `Imitation` library is selected for its built-in `SimpleDAggerTrainer` implementation, thereby eleminating the need to design and implement a custom DAgger pipeline.

The following sections describe the integration of this framework into RLRoverLab, including wrapping the environment to support separate student and expert observations, setting up the expert policy model and configuring the training pipeline.

4.3.1 Wrapping the Gymnasium environment

To facilitate integration with the Imitation framework, it is beneficial to adapt the environment structure from RLRoverLab to retain the observations obtained by the expert policy. This includes adopting its Gymnasium-based interface and the mechanisms for action execution in simulation. To support this integration, a set of wrappers are implemented to bridge the expectations of the `Imitation` framework with the RLRoverLab infrastructure. The `SimpleDAggerTrainer` requires the environment to follow SB3 conventions. To this end, the RLRoverLab environment is wrapped using a series of lightweight modules that enforce finite action bounds, flatten observation dictionaries into SB3-compatible vectors, and perform the necessary conversions between NumPy arrays and Torch tensors. A shared base environment is split into two views, one for the expert and one for the student, allowing both policies to interact with a common simulation context while receiving appropriately processed observations. The implementation of this wrapping and splitting is shown in Snippet 4.3.1.

```
def make_shared_vecenv(task_name, cfg, video=False):
    base = gym.make(task_name, cfg=cfg, viewport=video)
    base = FiniteActionBox(base)
    base = NumpyToTorchAction(base, device="cuda:0")

    def _build_expert():
        env = FlattenPolicyObs(base, TEACHER_KEYS + STUDENT_KEYS)
        env = TorchTensorToNumpy(env)
        return env

    def _build_student():
        env = FlattenPolicyObs(base, TEACHER_KEYS + STUDENT_KEYS)
        env = StudentIdentity(env)
        return env

    vec_expert  = DummyVecEnv([_build_expert])
    vec_student = DummyVecEnv([_build_student])
```

Snippet 4.3.1: Implementation of the teacher-student environement splitting.

The wrapped environment serves to bridge differences in data representation between the Imitation framework and the Torch-based RLRoverLab simulation. While the simulation expects Torch tensors for actions and outputs Torch.tensor observations, the Imitation framework operates through SB3-style interfaces, where environments and policies communicate using NumPy

arrays. To ensure compatibility and efficient use of system memory, particularly since transitions are stored in NumPy format in system memory during DAgger training, the environment performs bi-directional conversion at its interfaces. Observations are flattened to satisfy the input format of SB3 policies, and actions are bounded and converted to Torch before being passed to the simulator. A single environment instance is used to serve both student and expert views, maintaining consistent simulation state while preserving their distinct observation modalities: the student operates on noisy RGB-D input, while the expert is queried using clean proprioceptive and heightmap data.

While `SimpleDAggerTrainer` expects a vectorized environment, the vectorization approach used in RLRoverLab via Isaac Lab's `SkrlVecEnvWrapper` is not directly compatible. SkrlVecEnvWrapper enables vectorization by spawning multiple agents within a single simulation instance, and sharing a single terrain asset. In contrast, the Imitation framework requires vectorization through Gymnasium or SB3 utilities, such as `DummyVecEnv`, which replicate the entire environment configuration for each instance. This results in multiple agents being instantiated across individual terrain instances, significantly increasing memory usage and initialization overhead due to redundant terrain loading. Thus, to satisfy the Imitation framework's interface requirements without compromising memory constraints or simulation fidelity, the environment is wrapped using SB3's `DummyVecEnv`, which emulates a vectorized interface while preserving single-instance execution.

The student and action environment views are wrapped such that they accommodate the Imitation framework requirements, while the simulation environment remains within the RLRoverLab framework. With the Isaac Lab-based Gymnasium environment adapted to support the SB3-based Imitation framework, the next step is to address compatibility at the policy level. As the expert policies are trained within RLRoverLab using `skrl`, they do not natively conform to SB3's policy interface. To make them callable within the Imitation framework, an additional adaptation is required to expose them through an SB3-compatible wrapper.

4.3.2 Porting expert policy to SB3

Since the expert policy is trained using `skrl` within the RLRoverLab framework, it does not conform to the SB3 policy interface expected by the Imitation framework. To enable compatibility with `SimpleDAggerTrainer`, the expert policy is wrapped in a custom subclass of SB3's `BasePolicy`, allowing it to be queried using the standard `.predict` method. The wrapper instantiates the original `GaussianNeuralNetworkConv` model and loads the pretrained weights from the skrl training pipeline. To ensure the expert policy receives the correct input representation, a mapping from the flattened observation vector to the expert's expected input is constructed using index slices defined during environment preprocessing. Only teacher-specific observation terms are extracted and assembled into a clean input vector before being passed to the policy network. This allows the model to operate exactly as it did during training, while pre-

senting a compatible interface to the Imitation library. The initialization of the expert network can be seen in Snippet 4.3.2.

```
    def _predict(self, obs: np.ndarray, deterministic: bool = True):
        if obs.ndim == 1:
            obs = obs[None, :]
        flat = torch.as_tensor(obs, device=self.device)
        flat = flat.index_select(-1, self.teacher_indices.to(flat.device))
        clean = torch.cat([flat[..., :5], flat[..., 5:]], dim=-1)
        assert clean.shape[-1] == 10_206
        with torch.no_grad():
            mean, log_std, _ = self.net({"states": clean})

            if deterministic:
                act = mean
            else:
                std = log_std.exp()
                eps = torch.randn_like(mean)
                act = torch.tanh(mean + eps * std)
        return act
```

Snippet 4.3.2: Implementation of the instantiation of the RLRoverLab `GaussianNeuralNetworkConv` policy network in the SB3-based expert policy.

To satisfy the interface constraints of the BC context in which the expert is used the `_predict` method is implemented to convert incoming observations to Torch tensors, extract the relevant indices, and forward them through the model. Depending on whether deterministic or stochastic actions are requested, the wrapper either returns the network mean or samples from the modeled distribution. The resulting actions are returned in NumPy format to conform with SB3 expectations. This implementation is shown in Snippet 4.3.3.

```
self.net = GaussianNeuralNetworkConv(
            observation_space = spaces.Box(-np.inf, np.inf, shape=(10206,),
            ↪  dtype=np.float32),
            action_space      = spaces.Box(low=-1, high=1, shape=(2,),
            ↪  dtype=np.float32),
            device            = self.device,
            mlp_input_size    = 5,
            encoder_input_size= 10201,
            encoder_layers    = [8, 16, 32, 64],
            encoder_activation= "leaky_relu",
            mlp_layers=[256, 160, 128],
            mlp_activation="leaky_relu",
        ).to(self.device)
```

Snippet 4.3.3: Implementation of the override of `_predict` method from BasePolicy.

These slices originate from the flattening logic in the `FlattenPolicyObs` wrapper, which stores a mapping of observation terms to their vector indices during environment preprocessing. By preserving the original skrl architecture and weights, this approach allows seamless expert policy reuse in the Imitation framework, avoiding the need for retraining or architectural translation. With both the environment and the expert policy now fully wrapped and aligned with SB3 and Imitation framework expectations, the final step is to configure the `SimpleDAggerTrainer` to integrate these components into a functional training loop.

4.3.3 Configuring the DAgger trainer

To configure the `SimpleDAggerTrainer`, each of its required components must be instantiated with appropriate parameters. A custom student policy is implemented directly in SB3 to ensure compatability with the Imitation framework and to facilitate integration with the surrounding infrastructure. The design deliberately aligns with the skrl-based RGB-D policy used in RL-RoverLab, supporting comparability and enabling a more direct conversion back to skrl format for evaluation, such as with Weights and Biases (Wandb) [47].

Central to this compatability is a custom feature extractor, `SB3_GaussianNeuralNetworkConvResNet`, implemented as a subclass of `BaseFeaturesExtractor`. This module replicates the encoder structure from RLRoverLab's policy, ensuring architectural consistency in the extracted feature representation across frameworks. The remaining fully connected layers responsible for generating policy and value outputs are defined via the `net_arch` parameter at policy instantiation. The complete policy structure used by SB3 can be seen in 4.11.

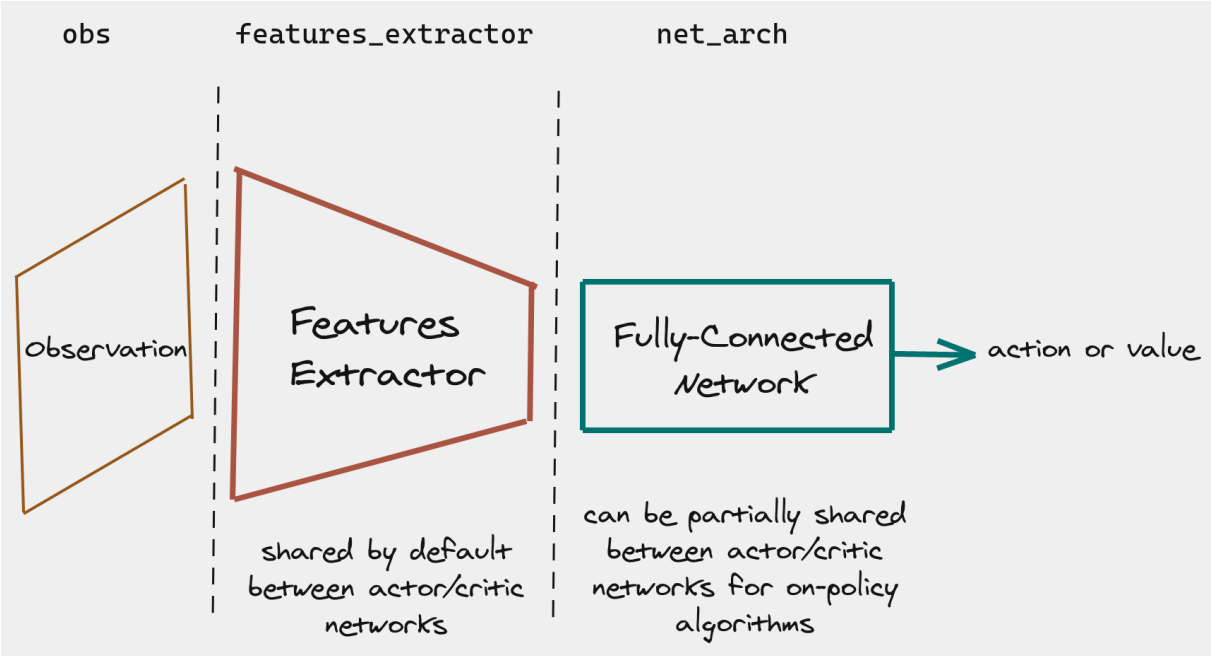

Figure 4.11: Illstration of the policy structure used by SB3, the observations (obs) is passed through a policy network (features_extractor), which then passes the output to a fully connected network (net_arch) tailored to output to the correct action shape. Adapted from [48].

This reflects the SB3's actor-critic design, where shared feature representations are passed to both policy and value heads. In this implementation, the actor and critic share the same feature extractor and multilayer perceptron (MLP), consistent with RLRoverLab's policy structure. The configuration of the student policy is shown in Snippet 4.3.4.

```
student_alg = PPO(
        policy            = "MultiInputPolicy",
        env               = vec_student,
        device            = device,
        verbose           = 1,
        policy_kwargs     = dict(
            features_extractor_class = SB3_GaussianNeuralNetworkConvResNet,
            features_extractor_kwargs = dict(
                encoder_layers     = [8, 16, 32, 64],
                encoder_activation = ("leaky_relu", "leaky_relu"),
                # Mapping info (generated by FlattenPolicyObs)
                key_slices         = vec_student.key_slices,
                key_shapes         = vec_student.key_shapes,
            ),
            net_arch = [256, 160, 128],
            activation_fn = torch.nn.LeakyReLU
        ),
    )
```

Snippet 4.3.4: Implementation of the initialization of the student policy as a PPO agent in SB3 format.

The MLP configuration reproduces the structure and activation functions used in the original skrl RGB-D model, preserving model depth. With the student policy fully defined, it is wrapped in a `BC` trainer, a required component by the `SimpleDAggerTrainer`. The `BC` trainer is responsible for updating the student using expert-labeled transitions collected during rollouts, tereby enabling imitation learning in a supervised manner. Once both the expert and student components are configured, the `SimpleDAggerTrainer` is instantiated, as show in Snippet 4.3.5.

```
dagger = SimpleDAggerTrainer(
        venv          = vec_expert,
        expert_policy = expert_policy,
        bc_trainer    = bc_trainer,
        scratch_dir   = logdir,
        rng           = np.random.default_rng(0),
        beta_schedule = lambda _: 1.0,
    )
```

Snippet 4.3.5: Initialization of the `SimpleDAggerTrainer` object.

Here, the expert policy and environment are passed directly, while the student policy is embedded via the `bc_trainer`. The `beta_schedule` is set to a constant value of 1, ensuring the expert is always queried for supervision during training. Finally, a minimal conversion script was im-

plemented to enable evaluation of the SB3-based DAgger policy within the SKRL framework. With all components of the training pipeline configured and integrated, an additional consideration is made to ensure the environment setup remains isolated. This separation maintains a clear boundary between the teacher-student pipeline and the original RLRoverLab configuration, preventing unnecessary coupling between the two.

4.3.4 Ensuring environment detachment

To maintain a clear separation between the teacher-student framework and the original RLRoverLab implementation, the environment configuration is reproduced within the Imitation framework, minimizing dependencies and potential inconsistencies. With the exception of selected imports from the other framework and the RLRoverLab policy collection, as well as the integration point within the central `train.py` script, the two systems remain distinct.

Within the Imitation framework, similarly to the implementation mentioned in subsection 4.1.2, the environment configuration is generated dynamically for each model using a structured configuration pipeline. This process is driven by three YAML files `agent_config`, `environment_config`, and `learning_config`, which define the respective components of the simulation setup. These configurations are parsed and composed into a unified configuration object used to instantiate the environment, allowing centralized and transparent control over model-specific settings. With all components integrated and the environment configuration encapsulated, the framework is now prepared for systematic evaluation. The next chapter presents the testing procedures and analyses aligned with the objectives outlined in section 3.1.

5 Testing and evaluation

To assess the viability and effectiveness of the proposed method, the evaluations defined in section 3.1 are conducted using the integrated models. These evaluations enable an analysis of performance in both simulation and real-world deployment. To ensure reproducibility, all experimental configurations are documented in detail, here including the parameters used. The following sections describe the test rig, the individual testing scenarios and the evaluations of these. The tests are described sequentially in accordance with the objectives.

5.1 Testing Framework

All simulations and training procedures presented in this thesis are conducted on a single machine to ensure comparability between experiments and support reproducibility. All training is performed inside the Docker container provided by RLRoverLab. The testing rig is equipped with the following hardware specifications:

- **Operating System**: Ubuntu 22.04.5 LTS (x86_64)
- **Kernel Version**: 6.8.0-58-generic
- **CPU**: Intel Core i9-12900KF (12th Gen, 24 cores)
- **GPU**: NVIDIA GeForce RTX 3090 Ti 24 GB VRAM
- **System Memory**: 64 GB DDR5 RAM

This hardware configuration allows for high-fidelity parallel training of reinforcement learning policies and efficient processing of high-dimensional sensory data. In all experiments, PPO is used as the core learning algorithm, consistent with the standard configuration in RLRoverLab. To ensure stable learning performance across rover models, the original hyperparameters from RLRoverLab are used consistently throughout training and evaluation. These are summarized in Table 5.1.

Table 5.1: Hyperparameters used in all testing scenarios.

Optimization		Training Configuration	
`learning_rate`	$1.0e-4$	`rollouts` (n_steps)	60
`discount_factor` (γ)	0.99	`learning_epochs` (n_epochs)	4
`lambda` (λ)	0.95	`mini_batches`	60
`ratio_clip`	0.2	`grad_norm_clip`	0.5
`value_clip`	0.2	`kl_threshold`	0.008
`entropy_loss_scale`	0.0	`value_loss_scale`	1.0

In addition to a fixed training configuration, all policies share the same reward function. This reward formulation is inherited from RLRoverLab and designed to encourage safe and efficient navigation. It consists of seven components, including three terminal rewards that are applied upon episode completion. The overall reward is computed as follows:

$$r_{total} = r_{dist} + r_{osc} + r_{direc} + r_{head} + r_{rel\_ori} + \mathbf{1}_{done} \cdot (r_{success} + r_{far} + r_{collision}) \tag{5.1}$$

where $\mathbf{1}_{done}$ indicates episode termination, triggering one of the terminal rewards depending on the termination condition. Each reward term is scaled by its associated weight prior to aggregation, as summarized in Table 5.2.

Table 5.2: Reward Components with Definitions, Weights, and Variables

Reward	**Definition**	**Weight**	**Where**
Distance to Target r_{dist}	$\frac{1}{1+0.11\cdot\|\mathbf{d}\|^2}\cdot\frac{1}{T_{\max}}$	5.0	$\mathbf{d}$: distance to target, $T_{\max}$: max episode length
Reached Target ($r_{success}$)	$\begin{cases} 2\cdot\frac{T_{\text{rem}}}{T_{\max}}, & \|\mathbf{d}\|_2 < \tau \wedge \|\theta\| < 0.1 \\ 0, & \text{otherwise} \end{cases}$	5.0	T_{rem}: timesteps left, τ: distance threshold, θ: Relative goal orientation error
Far From Target r_{far}	$\begin{cases} 1.0, & \|\mathbf{d}\|_2 > \tau \\ 0, & \text{otherwise} \end{cases}$	-2.0	$\|\mathbf{d}\|_2$: Euclidean distance to target, τ: distance threshold
Relative goal orientation r_{rel_ori}	$\frac{1}{(1+\|\mathbf{d}\|_2)(1+\|\theta\|)}\cdot\frac{1}{T_{\max}}$	5.0	$\|\mathbf{d}\|_2$: Euclidean distance to goal, θ: angle to goal
Angle to target r_{direc}	$\begin{cases} \frac{\|\theta\|}{T_{\max}}, & \|\theta\| > 2.0 \\ 0, & \text{otherwise} \end{cases}$	-1.5	θ: Relative goal angle error
Oscillation Penalty r_{osc}	$\frac{[(\Delta v)^2+(\Delta\omega)^2]^2}{T_{\max}}, \Delta > 0.05$	-0.05	Δv: linear diff, $\Delta\omega$: angular diff between steps
Heading constraint r_{head}	$\frac{0.4\cdot\max(-v,0)}{T_{\max}}$	-0.5	v: linear velocity
Collision Penalty $r_{collision}$	$\begin{cases} 1.0, & \|\mathbf{F}\| > \tau \\ 0, & \text{otherwise} \end{cases}$	-3.0	$\|\mathbf{F}\|$: net contact force, τ: collision force threshold

This reward structure is designed to balance goal-seeking behavior with stability and safety. Rewards such as **Distance to Target**, **Reached Target**, and **Relative Goal Orientation** guide the agent toward the goal while promoting alignment in both position and heading. Conversely, penalties like **Far from Target**, **Angle to Target**, **Heading Constraint**, and **Oscillation** discourage inefficient, unstable, or non-purposeful behavior. Finally, the **Collision** penalty en-

forces safety by penalizing contact with obstacles. These components collectively shape the reward signal to balance goal-directed motion, stability, and safety during learning. To evaluate performance under realistic conditions, all policies are trained for 300,000 iterations on a Mars-analog terrain provided by the RLRoverLab framework. This training horizon was chosen empirically based on the time required to complete a full training run and to support stable learning, particularly when using high-dimensional observation spaces such as RGB-D input. The terrain, shown in Figure 5.1, spans a large simulated area and includes varied elevation, slope irregularities, and scattered obstacles which presents sufficient complexity to evaluate navigation performance under varied terrain conditions.

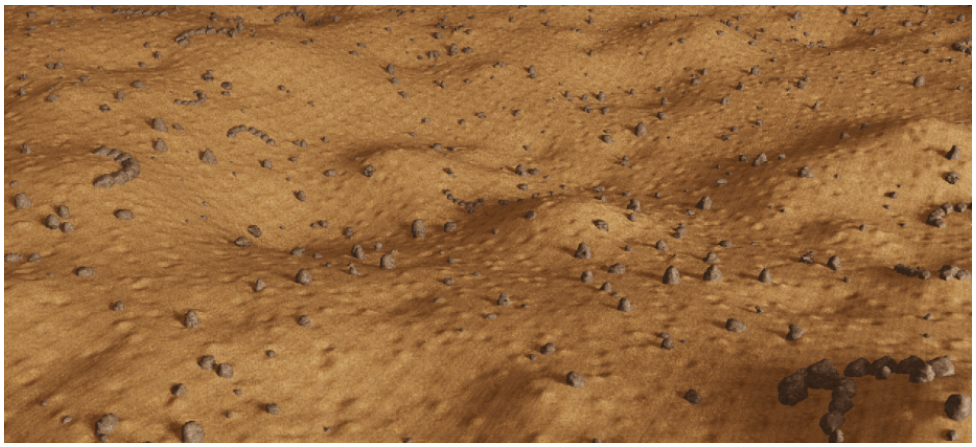

Figure 5.1: Depiction of the simulated mars terrain utilized in training.

The use of a fixed training regime, unified reward structure, and shared terrain environment ensures consistency and comparability across all experimental scenarios presented in this thesis. The following sections presents the results in the sequential order of the objectives presented in section 3.1.

5.2 Introduction of New Hardware Platforms

To address objectives 1d, 2a and 2b, a test is conducted to evaluate the viability of the newly added models and analyze their behavioral performance. For this purpose, the existing RLRoverLab policy model utilizing the heightmap input, depicted in Figure 5.2, is employed to verify the suitability of the models in training using the drive and control parameters defined in 4.1.

Figure 5.2: Visualization of the heightmap sensor modality in simulation. A 5×5 m area with 0.05 m resolution is projected onto the terrain and transformed into the robot's frame for local perception and processing.

For each model, a policy is trained with 512 agents for 300,000 timesteps to ensure training robustness. The resulting mean reward is compared to that of Mortensen [14], as showcased in Figure 5.3 under 7×7 `Resolution 5cm, CNN: Type 2`. Although the resolution differs slightly from that used in RLRoverLab, this configuration represents the most appropriate point of comparison available. In addition, each policy is evaluated through an analysis of the training performance metrics, specifically target reaching consistency, collision frequency and action smoothness, to ensure alignment with expected real-world performance. Lastly, the functional performance of each policy is evaluated through visual inspection of the model policy in simulation.

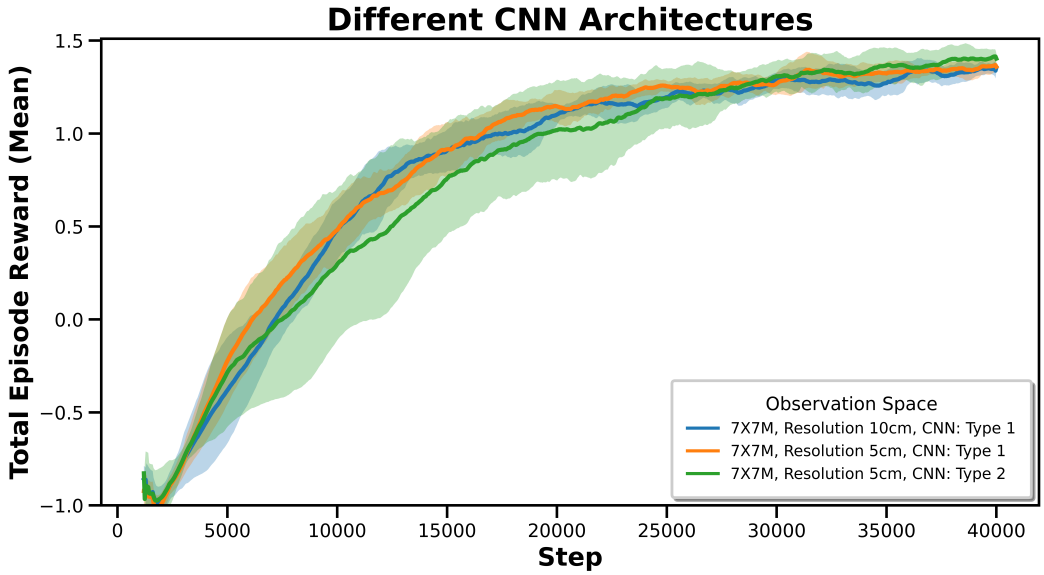

Figure 5.3: Rewards attained by Mortensen in [14].

The training of each policy is conducted in the RLRoverLab framework, with no changes, such that it matches with the setup used in training the AAUMarsRover model. The following section presents the results of this testing and the behavioral analysis of each model.

5.2.1 Results and analysis

The mean training results for each model are presented in Figure 5.4 for the entire training horizon. From this a trend can be extracted to suggest the viability of each of the models according to Objective 1d. As can be seen both the Summit and Leo Rover, perform similarly to the results presented from Mortensen, albeit delayed, they appear stable. The two RobuROC4 models perform slightly worse and plateau early, which may reflect limitations in their parameter configurations or underlying model dynamics. Building on these overall performance trends, the subsequent analysis focuses on specific behavioral outcomes, beginning with goal-reaching consistency.

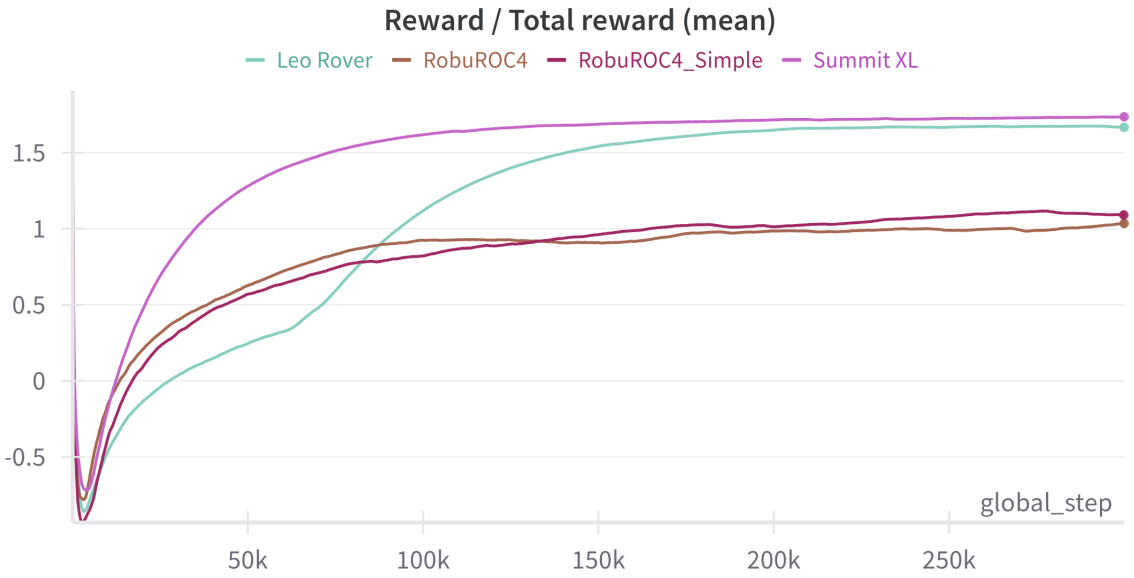

Figure 5.4: Mean total reward over 300,000 training steps for the RB Summit, RobuROC4, simplified RobuROC4, and Leo Rover models, all trained with heightmap inputs. The graph shows how each model's reward develops over time for comparison.

To support the following analysis, the videos below capture the behavior of fixed policies during post-training evaluation and do not reflect ongoing learning.

- **Summit video**: https://youtu.be/an286Ncc9uI
- **Leo Rover video**: https://youtu.be/eUEGS_PmYsQ
- **RobuROC4 video**: https://youtu.be/rqOJoziLt8w
- **RobuROC4 Simple video**: https://youtu.be/rqOJoziLt8w

In general, the behavior of all models indicates a stable trajectory toward goal attainment, as shown in Figure 5.6. While the Summit outperforms the other models, its resulting behavior is not entirely optimal; it has learned to exploit backward motion to achieve an early alignment with the target orientation, thereby exhibiting a preference for reverse velocity. This behavior can be seen it its associated video. Although the heightmap input provides visual information in the area surrounding the agent, such environmental awareness cannot be guaranteed in extraterrestrial contexts, therefore, this behavior is not considered desirable. However, it does exhibit strong performance in executing turn-in-place maneuvers, which contributes to its ability to align with the target orientation efficiently. Similarly, the Leo Rover performs well overall, but also exhibits unintended behavior, particularly during turning maneuvers as can be seen in the associated video. Its rocker joints respond excessively to external forces and friction, resulting in unrealistic dynamics, including tipping over and exaggerated motion showcased in Figure 5.5.

Figure 5.5: Depiction of the exaggerated movement seen during turning maneauvers for the Leo Rover, taken in Isaac Sim. As can be seen the rocker joints reach the limits in opposite directions while performing turning maneauvers.

Although the Simple RobuROC4 model performs worse in terms of overall performance, it still demonstrates consistent behavior. However, it shares the Summit's unintended preference for reverse velocity. Additionally, it struggles with turning on sloped or uneven terrain, leading to increased time required for goal alignment. Finally, the more complex RobuROC4 model also exhibits difficulties with turning and, overall, performs similarly to the simpler variant. However, it does not display the unintended behaviors observed in the other models, and therefore aligns more closely with the intended, realistic behavior.

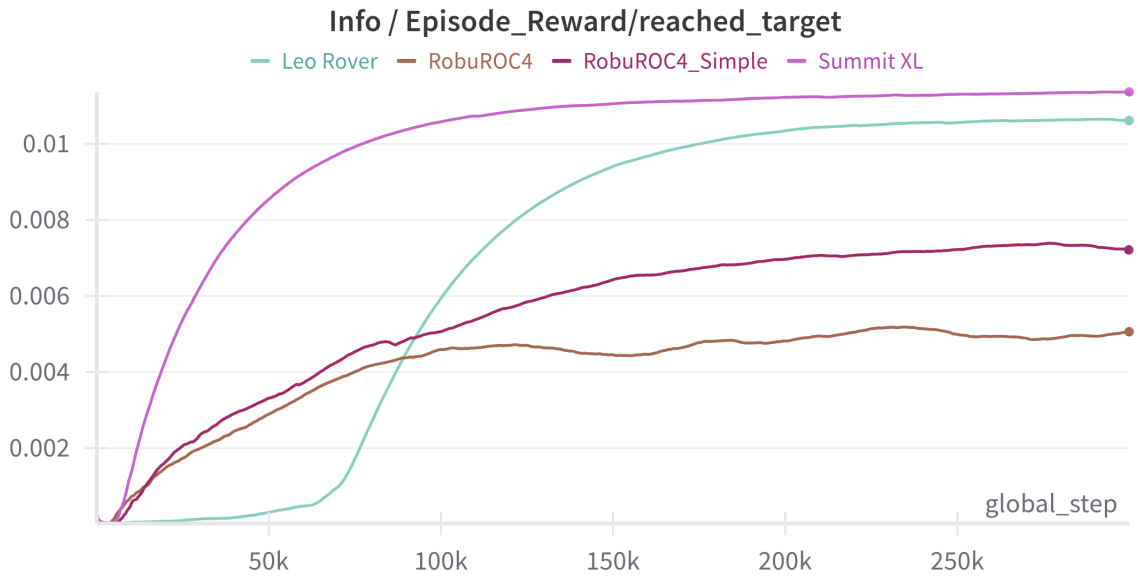

Figure 5.6: Target-reaching success over 300,000 training steps for the RB Summit, RobuROC4, simplified RobuROC4, and Leo Rover models. The graph indicates how consistently each model reaches the goal throughout training through the size and consistency of the reward.

While goal completion is important, safety remains a primary concern in extraterrestrial missions. In this context, safety is defined as the avoidance of collisions with obstacles. As shown in Figure 5.7, all models exhibit a near-zero collision penalty, indicating reliable obstacle avoidance throughout training. Minor collisions do occur, which is to be expected, as the inherent characteristics of skid-steering, particularly during turning on sloped or uneven terrain, can lead to slipping and occasional contact with obstacles. This is especially evident in the RobuROC4 models, where turning often requires slight back-and-forth motion, increasing the likelihood of such incidents.

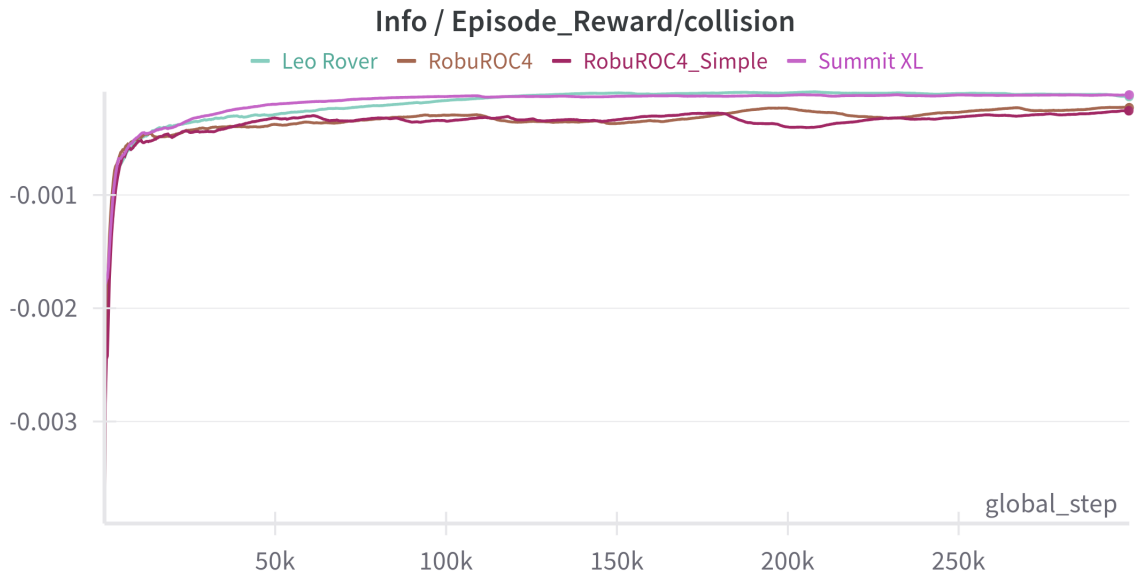

Figure 5.7: Collision penalty over 300,000 training steps for the RB Summit, RobuROC4, simplified RobuROC4, and Leo Rover models. The graph reflects how often each agent collides with obstacles in the environment through the size and consistency of the reward.

Another critical performance metric is the smoothness of actions, as sudden or oscillatory movements can pose risks in extraterrestrial environments. Most models converge toward minimal action changes, resulting in smoother trajectories and motion. However, the RobuROC4 models exhibit an early plateau and fail to fully minimize oscillations, as shown in Figure 5.8. This is likely attributable to their bulky, compact design, which complicates turning, particularly on inclined surfaces, and leads to the characteristic back-and-forth motion observed during maneuvers, further contributing to persistent oscillatory behavior.

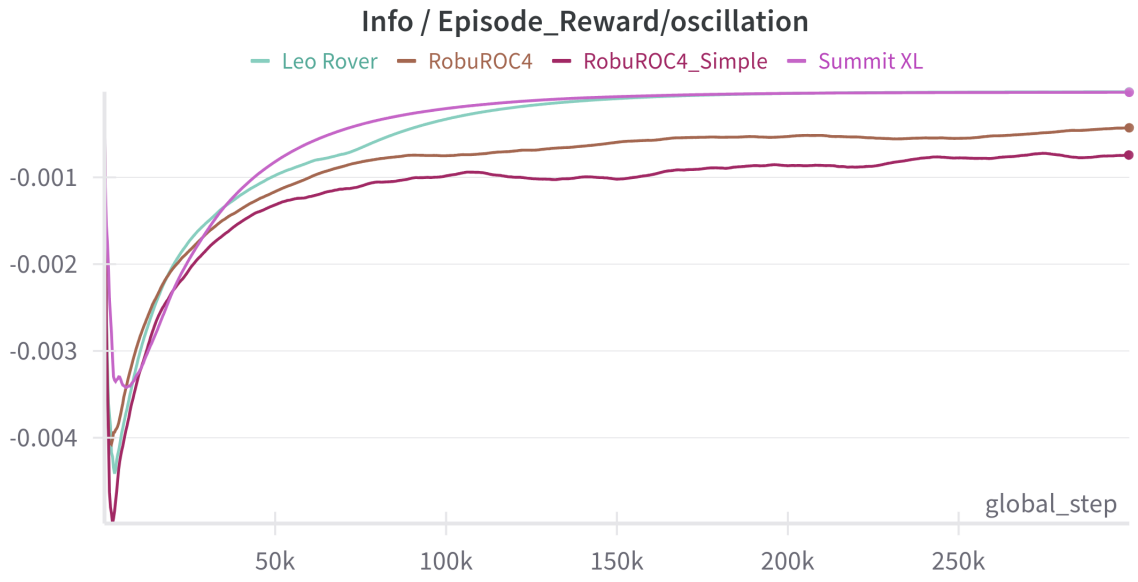

Figure 5.8: Penalty for large action changes over 300,000 training steps for the RB Summit, RobuROC4, simplified RobuROC4, and Leo Rover models. The graph reflects how often and to what extent each agent exhibits erratic or unstable movements during training.

Overall, the models demonstrate reasonable performance, with most negative outcomes primarily attributed to the limitations of the skid-steering control scheme, particularly in relation to the high-friction terrain. While the Summit and Leo Rover achieved strong performance metrics, their behavior during evaluation was suboptimal. The simplified RobuROC4 model exhibited similar behavioral issues, along with lower performance metrics. Although the more complex RobuROC4 model performed below average in overall performance, it was the only one to demonstrate realistic and consistent behavior. Although the other models may perform differently under future testing conditions, the RobuROC4 model is selected for continued evaluation due to its realistic and consistent behavioral characteristics, despite its below-average performance metric. Thus, with the initial evaluation of the models completed, the following tests the viability and efficiency of training policies with direct RGB-D input in both clean and noisy formats.

5.3 RGB-D input modality

Building on the previous evaluation, this section investigates the feasibility of using RGB-D data as a standalone input modality, analogous to the previously tested heightmap, for training agents in navigation tasks. In line with Objective 4, the aim is to assess whether RGB-D input enables effective learning and whether this holds under the presence of realistic sensor noise.

The evaluation focuses on two main aspects: the agent's ability to learn navigation tasks using RGB-D input, from a front-facing simulated camera, under both clean and noisy conditions, and

a comparison of training and behavioral performance relative to the heightmap-based approach under equivalent training conditions. For this test, the RGB-D encoder, presented in subsection 4.2.2, is used as the policy model, with each policy trained for 300,000 timesteps. Due to GPU memory limitations, the number of parallel agents is reduced to 256. Lastly, to prevent interference during evaluation, agent visibility is disabled, ensuring that the simulated cameras do not capture other agents in the environment. The results and behavioral comparisons are presented in the following section.

5.3.1 Results and analysis

The mean training performance for each model is shown in Figure 5.9. Evidently, neither policy achieves a mean reward above zero throughout training. This consistently low performance suggests that learning directly from RGB-D input is significantly more difficult than from the heightmap representation used previously. Several potential causes may contribute to this outcome, including previously identified issues and the complexity or redundancy of RGB-D features, which can hinder effective optimization. Despite the lack of convergence, the following analysis examines specific behavioral outcomes to further assess policy performance, beginning with goal-reaching consistency.

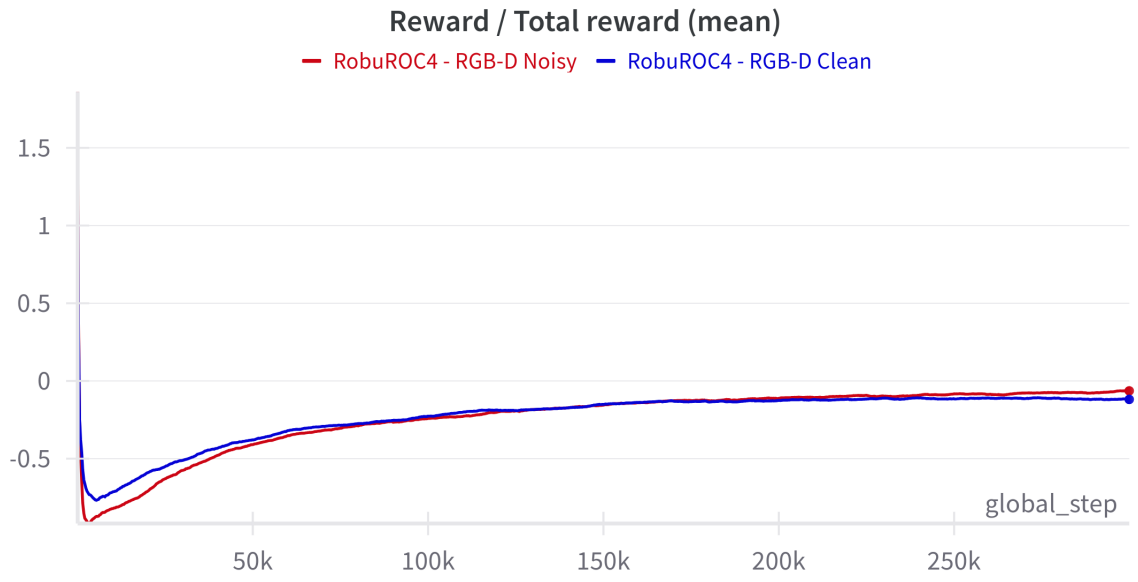

Figure 5.9: Mean total reward over 300,000 training steps for the RobuROC4 model using clean and noisy RGB-D inputs. The graph shows how the reward evolves over time for each input modality.

As in the previous test, the following two videos support the behavioral analysis by providing a visual reference of policy behavior after training, illustrating agent performance without the influence of continued learning.

- **RobuROC4; Clean RGB-D input**: https://youtu.be/55M80YyUHac

- **RobuROC4; Noisy RGB-D input**: https://youtu.be/KamLWRok-j4

Overall, neither policy consistently exhibits goal-reaching behavior, as reflected in Figure 5.10. Both policies tend to remain near their initial positions while performing turning maneuvers in place, a behavior also visible in the video recordings. This may be the result of a number of factors, including tuning the hyperparameters for this purpose, optimizing the ResNet18 encoder and tuning the model parameters to enhance the turning capabilities of the RobuROC4 model.

Figure 5.10: Target-reaching success over 300,000 training steps for the RobuROC4 model using clean and noisy RGB-D inputs. The graph shows how the reward associated with goal-reaching evolves over time for each input modality.

As a consequence of the agents not moving away from the initial position, they receive very few penalties for collisions, as illustrated in Figure 5.11. This is likely due to them rarely coming into contact with obstacles with the exhibited behavior. The collisions that do occur are likely due occasional cases, such as when turning on sloped terrain, where slippage may lead to unintended contact.

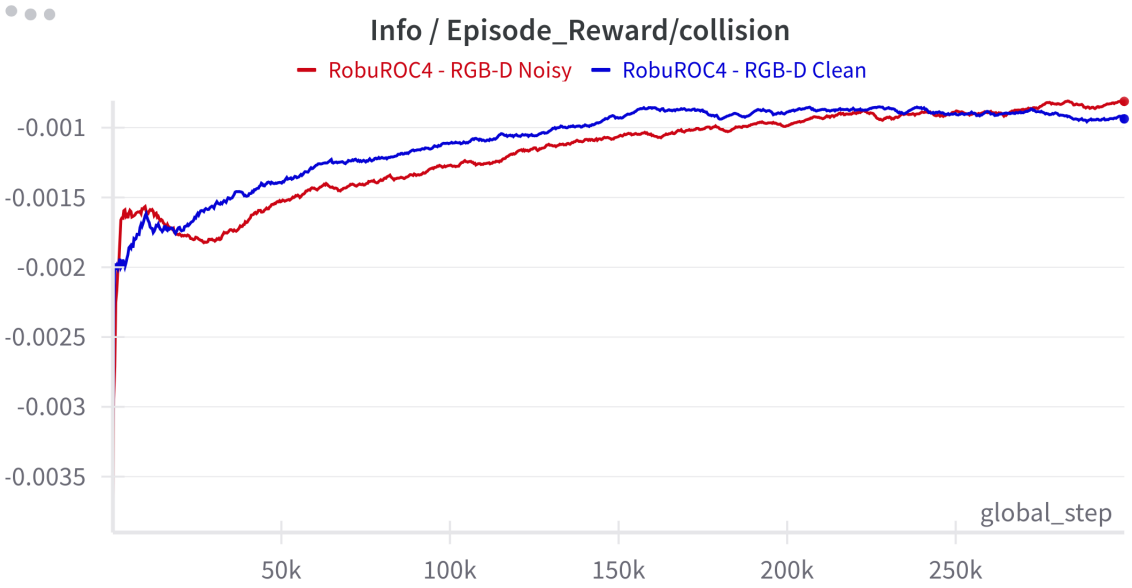

Figure 5.11: Collision penalty over 300,000 training steps for the RobuROC4 model using clean and noisy RGB-D inputs. The graph shows how collision-related penalties evolve over time for each input modality.

Similarly, with regard to oscillatory behavior, the agents learn to execute turning maneuvers smoothly, thereby maintaining a low penalty for large changes in actions. However, in the video of Noisy RGB-D trained policy, the agent act more erratically. Thus this penalty still applies pressure in regards to not performing excessive and erratic movements.

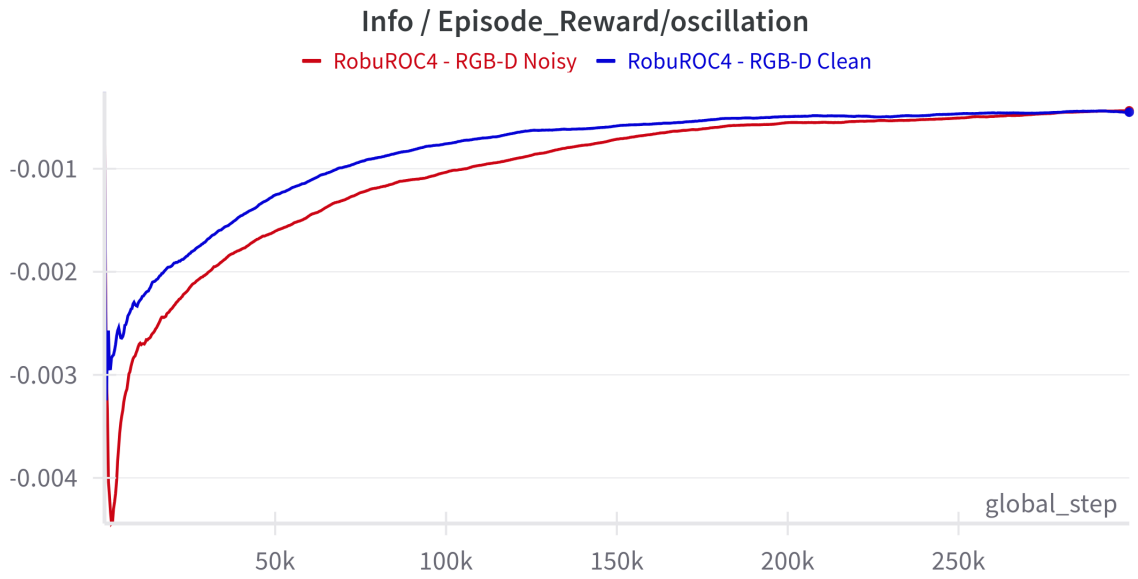

Figure 5.12: Penalty for large action changes over 300,000 training steps for the RobuROC4 model using clean and noisy RGB-D inputs. The graph reflects how often and to what extent each policy exhibits erratic or unstable movements during training.

In conclusion, neither of the policies learned behaviors that consistently resulted in successful task completion or demonstrated safety awareness. The results suggest that while using RGB-D as a standalone input modality for reinforcement learning is not necessarily infeasible, it may require further tuning of training parameters or policy models to achieve stable and consistent convergence toward desired behaviors. Despite the limitations observed in this test, the results offer useful insights for refining the training process. With the feasibility of training directly on RGB-D input explored, the following test investigates the viability of using a DAgger-based teacher-student approach can improve training efficiency and stability and better convergence.

5.4 Teacher-student training

Building on the previous tests, this section evaluates the viability of a teacher-student training approach, where a policy trained with privileged input, specifically a noise-free heightmap input, is used to supervise the learning of a student policy that relies on noisy RGB-D input from the same front-mounted camera as in the previous test. In accordance with Objective 5, the focus is on assessing how effectively the behavior learned by the teacher can be transferred to the student, despite the difference in sensor input and the added challenge of input noise.

For this purpose the implemented Imitation DAgger framework is employed to train a student policy using the trained RobuROC4 policy from the first test in section 5.2. All configurations of the environment are retained and the policy network utilizes the implemented `SB3_GaussianNeuralNetworkConvResNet` policy network for the student policy, which re-

sembles the original skrl policy model for RGB-D inputs. The teacher keeps the original policy model from the RLRoverLab framework. The student is trained for a minimum of 10,000 timesteps, with the BC framework training for at least two epochs. This is due to memory constraints and limitations within the implementation of the DAgger trainer. After training, the resulting student policy is converted to skrl format and evaluated evaluated deterministically for 50,000 steps using 64 parallel agents, to establish a consistent performance baseline. The following section presents and analyzes the results.

5.4.1 Results

The output of the DAgger training is shown in Figures 5.13 and 5.14, reflecting logs from the SB3-based training process. While mean episodic rewards decline over time, the probability of selecting the teacher's action (`prob_true_act`) steadily increases. This suggests that the student is gradually aligning with the teacher's behavior at an action level, but fails to generalize this behavior effectively during full rollouts.

batch_size	32
bc/	
batch	0
ent_loss	-0.00284
entropy	2.84
epoch	0
l2_loss	0
l2_norm	7.53e+03
loss	218
neglogp	218
prob_true_act	1.49e-16
samples_so_far	32
rollout/	
return_max	0.126
return_mean	-0.256
return_min	-0.406
return_std	0.206

Figure 5.13: Initial output of rollout training from the DAgger training.

batch_size	32
bc/	
batch	500
ent_loss	-0.00308
entropy	3.08
epoch	1
l2_loss	0
l2_norm	7.69e+03
loss	36.9
neglogp	36.9
prob_true_act	1.87e-15
samples_so_far	16032
rollout/	
return_max	-0.556
return_mean	-1.06
return_min	-1.83
return_std	0.442

Figure 5.14: Final output of rollout training from the DAgger training.

Evaluation in skrl supports the trend seen during training. As shown in Figure 5.15, the resulting policy maintains a negative mean reward throughout evaluation. Given the short training duration and the nature of the dataset-based supervision, the policy does not demonstrate meaningful improvement over time. Despite the lack of meaningful improvement, the following analysis examines the policies' behavioral outcomes, again starting with goal-reaching.

Figure 5.15: Mean total reward over 50,000 evaluation steps for the RobuROC4 model trained using DAgger. The graph shows how the reward evolves over time for each input modality.

To support the behavioral analysis, a video of post-training behavior is provided: https://youtu.be/i6-hTXRRqZM. As seen in the recording, the agent primarily performs localized rotations and minor exploratory motion, without forming consistent trajectories toward the goal. This is reflected in Figure 5.16, where the goal-reaching success remains effectively zero throughout evaluation. The initial spike is attributed to agents spawning directly on the goal, a common and expected outcome in randomized placements.

Figure 5.16: Target-reaching success over 50,000 evaluation steps for the RobuROC4 model trained using DAgger. The graph shows how the reward associated with goal-reaching evolves over time for each input modality.

As a consequence of this behavior, the agent receives slightly higher penalties for collisions than earlier tests, as shown in Figure 5.17. While it remains localized in the same area, it tends to explore more than the RGB-D tests. This results in significantly more collisions and a higher penalty.

Figure 5.17: Collision penalty over 50,000 evaluation steps for the RobuROC4 model trained using DAgger. The graph shows how collision-related penalties evolve over time for each input modality.

Contrarily, for the oscillation penalty, the agent appears to have adopted the teacher's smooth action trajectory, resulting in consistently low penalties for oscillatory behavior, as shown in Figure 5.18.

Figure 5.18: Penalty for large action changes over 50,000 evaluation steps for the RobuROC4 model trained using DAgger. The graph reflects how often and to what extent each policy exhibits erratic or unstable movements during training.

In summary, while the student policy trained using DAgger shows increasing alignment with the teacher at the action-selection level, this does not translate into functional behavior during evaluation. The agent remains mostly localized, fails to pursue goals, and exhibits minimal interaction with the environment. However, the retention of smooth control patterns from the teacher indicates that some aspects of behavior are preserved. Due to the outcome of this, there is no academic reason for testing it on a physical platform, therefore the testing of Objective 6 is excluded. Thus, these findings conclude the testing phase and lay the groundwork for the subsequent discussion, which reflects on the broader implications, limitations, and interpretability of the results.

6 Discussion

The results presented in Chapter 5 collectively inform the viability of transferring behavior between sensor modalities through a teacher-student framework. Central to this exploration is the question of whether a policy trained with privileged input, in this case a noise-free heightmap, can effectively supervise the learning of a policy relying on noisier, more realistic RGB-D input. To contextualize this final test, preceding experiments focused on validating the simulation pipeline, tuning robot models, and assessing the difficulty of training policies directly from RGB-D data. These tests were designed as a sequential process, gradually building towards the evaluation of cross-modality policy transfer under realistic and constrained conditions.

The first set of tests concluded that none of the evaluated robot models achieved satisfactory results across both task and behavioral metrics. While the Summit and Leo Rover models achieve significantly better task performance than the RobuROC4 models, their in-simulation behavior does not align with expectations for safety-oriented navigation, particularly in the case of the Leo Rover, whose behavior appears unrealistic during turning maneuvers. In contrast, the RobuROC4 models performs significantly worse on performance metrics. However, the more complex RobuROC4 model demonstrates better general behavioral characteristics, albeit still lacking in task performance. These shortcomings likely stem from suboptimal parameter tuning, particularly related to damping, stiffness, and the action scaling applied to each model's control output. These parameters directly influence how effectively a model can respond to terrain friction and execute precise movements. Additionally, RobuROC4 models exhibited specific control issues, such as an inability to turn in place effectively, which may be linked to misconfigured physical parameters like *track_width* and *wheel_radius*, regardless of being based on the physical platforms. In contrast, the other tested models, such as Summit and Leo Rover, did not show the same difficulty with in-place rotation. Despite these limitations, RobuROC4 was selected for the remaining tests, as it demonstrated forward-oriented movement and relatively stable behavior, even though it did not meet the thresholds for success in terms of total mean reward.

The second set of test examined whether RGB-D input, combined with proprioceptive data, could serve as a viable standalone modality for reinforcement learning. In contrast to heightmap-based training, policies trained using RGB-D struggled to achieve neither meaningful task performance nor significant behavioral traits. This highlighted the difficulty of extracting spatially coherent policies from high-dimensional, limited field-of-view input. Both clean and noisy RGB-D variants were tested, with the noisy input exhibiting marginal improvements, compared to the clean input, in several evaluation metrics, likely due to the use of a ResNet18 backbone pretrained on ImageNet, which may generalize better to realistic noise patterns than to idealized

simulated data. The subsequent test therefore focused on policy transfer using a teacher-student framework, where a model trained with privileged heightmap input supervises learning under RGB-D observations.

The DAgger-based teacher-student experiment aimed to improve policy learning under RGB-D observations by leveraging supervision from a heightmap-trained expert. In this setup, the expert policy remained in a supervisory role and only processed the privileged inputs that it was originally trained on. The student, in turn, received corrective action labels directly from the expert, preserving the Learning by Cheating formulation in which the student learns exclusively from privileged supervision without direct access to privileged observations. Despite this architecture, the resulting student policy failed to generalize effectively. While action-level imitation improved over time, as reflected by the increasing *prob_true_act* metric, the learned behavior remained largely meaningless and directionless, with the agent showing no goal-seeking behavior during evaluation. These shortcomings are primarily attributed to practical constraints in the current implementation rather than conceptual limitations of the DAgger approach itself. The implementation restricted training to a single agent and a small number of student-environment interactions, due to the high memory usage of the Imitation framework, which loads the entire dataset into system memory. Additionally, behavioral cloning was limited to two epochs per iteration, further constraining the student's exposure to the collected supervision data. As a result, the student policy was trained on a relatively small and narrow dataset, reducing its ability to generalize across the full range of observed states.

A more scalable implementation, fully integrated into the skrl-based vectorized training pipeline of RLRoverLab, would allow parallel agent training with broader environmental coverage. This would likely yield a more diverse and representative dataset, improving the robustness of the learned policy. Two key limitations in the current setup hinder this potential. First, incompatibility between the *SkrlVecEnvWrapper* and the vectorization strategy expected by the Imitation framework restricted training to a single agent through *DummyVecEnv*. This reduced the diversity of encountered states during rollouts. Second, the Imitation framework stores the entire dataset in system memory, which constrains the overall training duration and volume of usable data.

Addressing these constraints would significantly enhance the practicality of DAgger in this setting. Enabling data streaming from disk would lift memory restrictions, allowing for extended training runs and greater dataset variety. Additionally, incorporating on-policy expert querying during rollouts with episodic transition saving would increase the range of state distributions covered during training. Together, these improvements point toward a custom skrl-based DAgger trainer as a more scalable and effective foundation for learning robust policies under high-dimensional, noisy sensor input.

7 Conclusion

The results presented in this thesis demonstrate the successful integration of four new robotic models into the RLRoverLab framework. Each model completed a full training cycle, allowing for direct performance comparisons across existing and new models. While the models varied in their ability to meet task-related objectives, all exhibited distinct and interpretable behaviors. Some models achieved higher scores in performance metrics, while others showed more consistent and safety-aligned movement patterns, particularly in terms of directional preference and reduced erratic motion. Similarly, training with RGB-D as a standalone input modality proved substantially more difficult. The combination of high-dimensional visual input and limited spatial coverage posed a significant challenge for policy optimization. These difficulties underscore the constraints of learning directly from RGB-D input under the tested conditions. Consequently, methods that incorporate structured supervision, such as imitation learning from privileged inputs, become increasingly important when working with constrained sensory representations.

In this context, the DAgger-based teacher-student framework was used to explore whether policies trained with privileged heightmap input could guide the learning of policies operating solely under noisy RGB-D observations. While the student policy failed to generalize to effective goal-directed behavior, the experiment confirmed the conceptual soundness of the framework. The architectural separation between the teacher and student was maintained throughout, ensuring that learning relied strictly on behavioral supervision without access to the teacher's privileged observations. This outcome is primarily attributed to limitations in the implementation of the current Imitation-based setup. Training was constrained by single-agent rollouts, static datasets stored entirely in memory, and limited training epochs per iteration. These factors severely restricted the diversity and representativeness of training data, impeding the student's ability to learn generalizable behavior.

These findings suggest that transferring behavior between separate sensor input modalities remains conceptually viable, assuming sufficient support from the training infrastructure. The experiments illustrate that the teacher-student framework can be implemented in settings with mismatched observation spaces, even if the resulting student performance was limited. This establishes a foundation for future work aimed at refining the implementation and scaling the training process to better support imitation learning under constrained sensor conditions.

7.1 Future work

To fully realize the potential of DAgger-based imitation learning within realistic robotic training environments, future work should prioritize the development of a custom DAgger trainer tai-

lored to the RLRoverLab and skrl frameworks. The current reliance on the Imitation framework imposes architectural and performance constraints that limit training scale, agent parallelism, and dataset diversity. A custom implementation would enable tighter integration with RLRoverLab's vectorized training pipeline, allowing simultaneous training of multiple agents in shared simulation contexts. This would significantly increase the range of explored states during rollouts, improving the broadness of the collected data.

Additional, the solution should incorporate memory-safe or memory-aware management of transition buffers. The current in-memory dataset handling restricts training duration and scalability, particularly with high-dimensional input such as RGB-D. A revised pipeline should support efficient on-disk storage and sequential and selective streaming to accommodate large transition datasets without exhausting system resources and forming an early bias. These improvements could support longer training runs, larger agent populations, and richer training distributions.

Bibliography

[1] NASA, Artemis iii - nasa, Dec. 2024.
URL: https://www.nasa.gov/mission/artemis-iii/, Retrieved: 15-02-2025.

[2] NASA, Nasa's lunar exploration program overview, NASA Publication, Sep. 2020.
URL: https://www.nasa.gov/wp-content/uploads/2020/12/artemis_plan-20200921.pdf?emrc=f43185.

[3] NASA, Artemis i, Mar. 2024.
URL: https://www.nasa.gov/mission/artemis-i/, (Retrieved: 22-02-2024).

[4] NASA, Nasa's plan for sustained lunar exploration and development, NASA Publication, Aug. 2020.
URL: https://www.nasa.gov/wp-content/uploads/2020/08/a_sustained_lunar_presence_nspc_report4220final.pdf?emrc=5aa8ef.

[5] T. E. S. Agency, Terrae novae 2030+ strategy roadmap, ESA Publication, Unclassified, 2022.
URL: https://esamultimedia.esa.int/docs/HRE/Terrae_Novae_2030+strategy_roadmap.pdf.

[6] NASA, Nasa technology roadmaps ta 4: Robotics and autonomous systems, NASA Publication, 2015.
URL: https://www.nasa.gov/wp-content/uploads/2016/08/2015_nasa_technology_roadmaps_ta_4_robotics_and_autonomous_systems_final.pdf.

[7] V. Wiberg, E. Wallin, T. Nordfjell, and M. Servin, Control of rough terrain vehicles using deep reinforcement learning, *IEEE Robotics and Automation Letters*, vol. 7, no. 1, pp. 390–397, 2022. DOI: 10.1109/LRA.2021.3126904.

[8] G. Kulathunga, A reinforcement learning based path planning approach in 3d environment, *Procedia Computer Science*, vol. 212, pp. 152–160, 2022, 11th International Young Scientist Conference on Computational Science, ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2022.10.217.
URL: https://www.sciencedirect.com/science/article/pii/S1877050922016891.

[9] T. Sakai and T. Nagai, Explainable autonomous robots: A survey and perspective, *Advanced Robotics*, vol. 36, no. 5-6, pp. 219–238, 2022. DOI: 10.1080/01691864.2022.2029720. eprint: https://doi.org/10.1080/01691864.2022.2029720.
URL: https://doi.org/10.1080/01691864.2022.2029720.

[10] W. Zhao, J. P. Queralta, and T. Westerlund, Sim-to-real transfer in deep reinforcement learning for robotics: A survey, in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2020, pp. 737–744. DOI: 10.1109/SSCI47803.2020.9308468.

[11] A. B. Mortensen, E. T. Pedersen, L. V. Benedicto, L. Burg, M. R. Madsen, and S. Bøgh, Teacher-student reinforcement learning for mapless navigation using a planetary space rover, 2023.
URL: https://arxiv.org/abs/2309.12807.

[12] D. Chen, B. Zhou, V. Koltun, and P. Krähenbühl, Learning by cheating, in *Proceedings of the Conference on Robot Learning*, L. P. Kaelbling, D. Kragic, and K. Sugiura, Eds., ser. Proceedings of Machine Learning Research, vol. 100, PMLR, 2020, pp. 66–75.
URL: https://proceedings.mlr.press/v100/chen20a.html.

[13] A. B. Mortensen and S. Bøgh, Rlroverlab: An advanced reinforcement learning suite for planetary rover simulation and training, in *2024 International Conference on Space Robotics (iSpaRo)*, 2024, pp. 273–277. DOI: 10.1109/iSpaRo60631.2024.10687686.

[14] A. B. Mortensen, From simulation to space: Advancing planetary space robotics with machine learning, M.S. thesis, Aalborg University, 2024.

[15] A. Ellery, Rover vision—fundamentals, in *Planetary Rovers: Robotic Exploration of the Solar System*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 199–262, ISBN: 978-3-642-03259-2. DOI: 10.1007/978-3-642-03259-2_6.
URL: https://doi.org/10.1007/978-3-642-03259-2_6.

[16] A. B. Andersen and M. W. Jørgensen, Reinforcement learning for robotic rock grasp learning in off-earth space environments, 2022.
URL: https://projekter.aau.dk/projekter/files/473992583/Paper___Reinforcement_Learning_for_Robotic_Rock_Grasp_Learning_in_Off_Earth_Space_Environments.pdf.

[17] A. Orsula, S. Bøgh, M. Olivares-Mendez, and C. Martinez, Learning to grasp on the moon from 3d octree observations with deep reinforcement learning, in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022, pp. 4112–4119. DOI: 10.1109/IROS47612.2022.9981661.

[18] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, Second. The MIT Press, 2018.
URL: http://incompleteideas.net/book/the-book-2nd.html.

[19] S. Levine, A. Kumar, G. Tucker, and J. Fu, Offline reinforcement learning: Tutorial, review, and perspectives on open problems, 2020. arXiv: 2005.01643 [cs.LG].
URL: https://arxiv.org/abs/2005.01643.

[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, Proximal policy optimization algorithms, 2017. arXiv: 1707.06347 [cs.LG].
URL: https://arxiv.org/abs/1707.06347.

[21] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, Trust region policy optimization, 2017. arXiv: 1502.05477 [cs.LG].
URL: https://arxiv.org/abs/1502.05477.

[22] M. M. Rahman and Y. Xue, Robust policy optimization in deep reinforcement learning, 2022. arXiv: 2212.07536 [cs.LG].
URL: https://arxiv.org/abs/2212.07536.

[23] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018. arXiv: 1801.01290 [cs.LG].
URL: https://arxiv.org/abs/1801.01290.

[24] S. Fujimoto, H. van Hoof, and D. Meger, Addressing function approximation error in actor-critic methods, 2018. arXiv: 1802.09477 [cs.AI].
URL: https://arxiv.org/abs/1802.09477.

[25] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, Carla: An open urban driving simulator, in *Conference on robot learning*, PMLR, 2017, pp. 1–16.

[26] S. Ross, G. J. Gordon, and J. A. Bagnell, A reduction of imitation learning and structured prediction to no-regret online learning, 2011. arXiv: 1011.0686 [cs.LG].
URL: https://arxiv.org/abs/1011.0686.

[27] N. Corporation, Nvidia isaac sim, 2025.
URL: https://developer.nvidia.com/isaac/sim, (Retrieved: 08-04-2025).

[28] A. Serrano-Muñoz, D. Chrysostomou, S. Bøgh, and N. Arana-Arexolaleiba, Skrl: Modular and flexible library for reinforcement learning, *Journal of Machine Learning Research*, vol. 24, no. 254, pp. 1–9, 2023.
URL: http://jmlr.org/papers/v24/23-0112.html.

[29] M. Voellmy and M. Ehrhardt, Exomy: A low cost 3d printed rover, Oct. 2020.

[30] G. Brockman, V. Cheung, L. Pettersson, et al., Openai gym, cite arxiv:1606.01540, 2016.
URL: http://arxiv.org/abs/1606.01540.

[31] V. Murali, A. Ramananda, S. Pandit, and N. H, Design and development of four-wheel steering for all terrain vehicle (a.t.v), vol. 7, pp. 1660–1668, Dec. 2020.

[32] E. Byte, Study of ackerman's steering gear mechanism, 2023.
URL: https://www.engineeringbyte.com/study-of-ackermans-steering-gear-mechanism, (Retrieved: 04-05-2025).

[33] X. Yu, P. Wang, and Z. Zhang, Learning-based end-to-end path planning for lunar rovers with safety constraints, *Sensors*, vol. 21, no. 3, p. 796, 2021. DOI: 10.3390/s21030796.

[34] W. Feng, L. Ding, R. Zhou, et al., Learning-based end-to-end navigation for planetary rovers considering non-geometric hazards, *IEEE Robotics and Automation Letters*, vol. 8, no. 7, pp. 4084–4091, 2023. DOI: 10.1109/LRA.2023.3281261.

[35] B.-J. Park and H.-J. Chung, Deep reinforcement learning-based failure-safe motion planning for a 4-wheeled 2-steering lunar rover, *Aerospace*, vol. 10, no. 3, p. 219, 2023. DOI: 10.3390/aerospace10030219.

[36] K. Zhang, Autonomous mobile robot navigation in 3d rough terrain using deep reinforcement learning, Master's Thesis, University of Toronto, 2020.

[37] R. Partsey, E. Wijmans, N. Yokoyama, O. Dobosevych, D. Batra, and O. Maksymets, Is mapping necessary for realistic pointgoal navigation?, in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 17 232–17 241.

[38] R. Egan and A. H. Göktogan, Deep learning based terrain classification for traversability analysis, path planning and control of a mars rover, in *Australasian Conference on Robotics and Automation (ACRA-2021), Melbourne, Australia*, 2021.

[39] H. Hu, K. Zhang, A. H. Tan, M. Ruan, C. Agia, and G. Nejat, A sim-to-real pipeline for deep reinforcement learning for autonomous robot navigation in cluttered rough terrain, *IEEE Robotics and Automation Letters*, vol. 6, no. 4, pp. 6569–6576, 2021.

[40] T. Tang, H. Du, X. Yu, and Y. Yang, Monocular camera-based point-goal navigation by learning depth channel and cross-modality pyramid fusion, in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, 2022, pp. 5422–5430.

[41] T. S. Sørensen, J. W. Wagner, and C. Z. G. Hostens, System and tool design for lunar regolith manipulation, Aalborg Universitet, Project Report, 2024.

[42] T. I. L. P. Developers, Camera, 2025.
URL: https://isaac-sim.github.io/IsaacLab/main/source/overview/core-concepts/sensors/camera.html, (Retrieved: 22-05-2025).

[43] A. Teichman, S. Miller, and S. Thrun, Unsupervised intrinsic calibration of depth sensors via slam. In *Robotics: Science and systems*, Citeseer, vol. 248, 2013, p. 3.

[44] J. Ansel, E. Yang, H. He, et al., PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation, in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, ACM, Apr. 2024. DOI: 10.1145/3620665.3640366.
URL: https://docs.pytorch.org/assets/pytorch2-2.pdf.

[45] A. Gleave, M. Taufeeque, J. Rocamonde, et al., Imitation: Clean imitation learning implementations, arXiv:2211.11972v1 [cs.LG], 2022. arXiv: 2211.11972 [cs.LG].
URL: https://arxiv.org/abs/2211.11972.

[46] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, Stable-baselines3: Reliable reinforcement learning implementations, *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.
URL: http://jmlr.org/papers/v22/20-1364.html.

[47] L. Biewald, Experiment tracking with weights and biases, Software available from wandb.com, 2020.
URL: https://www.wandb.com/.

[48] S. Baselines3, Policy networks, 2025.
URL: https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html, (Retrieved: 31-05-2025).