TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Phil-Yannick Borkenhagen 223605IASM

MODULAR ARBITRATION FRAMEWORK FOR ROBOTIC REINFORCEMENT LEARNING WITH SIM-TO-REAL TRANSFER

Master's Thesis

Supervisor: Aleksei Tepljakov
Prof.

Co-supervisor: Vladimir Kuts
Prof.

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Phil-Yannick Borkenhagen 223605IASM

MODULAARNE ARBITRAAŽIRAAMISTIK STIIMULÕPPEKS ROBOOTIKAS KOOS SIMULATSIOONI REAALSUSEKS ÜLEVIIMISEGA

Magistritöö

Juhendaja: Aleksei Tepljakov
Prof.

Kaasjuhendaja: Vladimir Kuts
Prof.

Tallinn 2024

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Phil-Yannick Borkenhagen

16.12.2024

Abstract

The transition towards Industry 4.0 and 5.0 requires increasingly sophisticated robotic systems capable of handling complex, multi-step tasks while maintaining interpretability and adaptability. With Deep Reinforcement Learning (DRL) showing promises in robotic applications, challenges persist in enabling robots to execute complex tasks requiring higher-level decision-making. This thesis presents a novel modular arbitration framework to combine multiple pre-trained reinforcement learning agents to address these challenges in real-world robotic applications.

The framework utilizes actor-critic algorithms, specifically Proximal Policy Optimization (PPO), and uses a learning arbitration agent to select actions based on policy and value network outputs from pre-trained modules. The effectiveness of this approach is evaluated through a series of experiments in both simulated and real-world environments using a multi-step robotic task involving exploration, collision avoidance, and object detection.

Results demonstrate the framework's capability to combine specialized agents, but also show the limitations of only relying on the policy and value network outputs of pre-trained modules. This becomes especially apparent in changing environments, simulated or real-world, causing a shift in the value estimate ranges and thereby preventing a successful module selection.

This thesis is written in English and is 84 pages long, including 7 chapters, 23 figures and 17 tables.

Annotatsioon

Modulaarne arbitraažiraamistik stiimulõppeks robootikas koos simulatsiooni reaalsuseks üleviimisega

Üleminek tööstuse 4.0 ja 5.0 suunas nõuab üha keerukamaid robotisüsteeme, mis suudavad täita keerulisi, mitmeastmelisi ülesandeid, säilitades samal ajal tõlgendatavuse ja kohanemisvõime. Kuigi sügavtugevdusõppel (Deep Reinforcement Learning, DRL) on robotite rakendustes palju lubadusi, seisneb jätkuvalt väljakutse selles, kuidas võimaldada robotitel sooritada keerukaid ülesandeid, mis nõuavad kõrgema taseme otsustamist. Käesolev magistritöö esitleb uudset modulaarset arbitraažiraamistikku, mis ühendab mitu eelnevalt treenitud tugevdusõppe agenti, et lahendada neid väljakutseid reaalmaailma robotirakendustes.

Raamistik kasutab tegija-kriitik (actor-critic) algoritme, eelkõige Proksimaalset Poliitika Optimeerimist (Proximal Policy Optimization, PPO), ja õppivat arbitraažiagentti, et valida tegevusi eelnevalt treenitud moodulite poliitika- ja väärtusvõrkude väljundite põhjal. Selle lähenemise efektiivsust hinnatakse mitmete eksperimentide kaudu nii simuleeritud kui ka reaalmaailma keskkondades, kasutades mitmeastmelist robotülesannet, mis hõlmab uurimist, kokkupõrgete vältimist ja objektide tuvastamist.

Tulemused näitavad, et raamistik on võimeline kombineerima spetsialiseerunud agente, kuid toovad esile ka piirangud, mis tulenevad ainult eelnevalt treenitud moodulite poliitika- ja väärtusvõrkude väljunditele tuginemisest. See probleem muutub eriti ilmseks muutuvates keskkondades, olgu need siis simuleeritud või reaalmaailma omad, kus väärtushinnangute vahemikud võivad muutuda.

Lõputöö on kirjutatud [mis keeles] keeles ning sisaldab teksti 84 leheküljel, 7 peatükki, 23 joonist, 17 tabelit.

List of Abbreviations and Terms

CNN	Convolutional Neural Network.
CUDA	Compute Unified Device Architecture.
DQN	Deep Q-Network.
DRL	Deep Reinforcement Learning.
FNN	Feedforward Neural Network.
GAE	Generalized Advantage Estimate.
HRL	Hierarchical Reinforcement Learning.
KL	Kullback-Leibler.
LSTM	Long Short-Tem Memory.
MARL	Multi-Agent Reinforcement Learning.
MDP	Markov Decision Process.
MORL	Multi-Objective Reinforcement Learning.
MRL	Modular Reinforcement Learning.
OGM	Occupancy Grid Map.
PGM	Policy Gradient Method.
PPO	Proximal Policy Optimization.
RL	Reinforcement Learning.
RNN	Recurrent Neural Network.
RoI	Region of Interest.
ROS	Robot Operating System.
TD	Temporal Difference.
TRPO	Trust Region Policy Optimization.
URDF	Unified Robot Description Format.

Table of Contents

List of Figures

List of Tables

1. Introduction

Industrial processes are undergoing a fundamental transformation through the ongoing transition towards Industry 4.0 and 5.0. While Industry 4.0 focuses on automation and data exchange in manufacturing processes, Industry 5.0 emphasizes human-robot collaboration and flexibility. This shift requires advancements in robotic control and learning capabilities to enable systems that can adapt to dynamic environments and handle complex tasks. Future robots must be able to handle varying objectives and environments while being interpretable and maintainable. The ability to understand, modify, and extend robot behaviors is becoming as crucial as their capabilities, especially in safety-critical applications or human-robot collaboration scenarios [2].

One promising approach for achieving those capabilities is DRL, having demonstrated impressive potential in solving complex problems while processing high-dimensional input data. With the first notable milestones in simulations and game environments, DRL is successfully utilized for robotic tasks that cannot be handled effectively by classical control. This includes tasks like grasping, manipulation, and navigation. But challenges remain in enabling robots to handle complex, multi-step tasks that require a higher level of understanding and decision-making. For instance, robots in a modern warehouse might need to navigate dynamic obstacles, identify specific items, and perform multi-step manipulation sequences. Similarly, service robots in healthcare settings must execute complex tasks while safely interacting with humans [3].

There are multiple approaches for handling these challenges, including experience replay for improved sample efficiency, curriculum learning for guiding the learning process and leveraging prior knowledge using transfer learning. An especially promising direction is the decomposition of tasks into sub-tasks, as done in hierarchical and modular reinforcement learning. Hierarchical reinforcement learning addresses complex tasks by dividing them into a hierarchy of sub-tasks. Prominent frameworks are *Options* [4], *JointPPO* [5], *MaxQ* [6] and *Concepts* [7]. However, there are still notable areas for improvement, including the not straightforward reusability of the sub-agents and the requirements of the higher-level agents to obtain at least the same observation space as their sub-agents, resulting in high memory requirements and complex neural network architectures. In modular reinforcement learning, each of the resulting sub-tasks is trained independently and then re-combined using an arbitration agent to solve the original task. This approach reduces the complexity of the overall task and allows for a modular and

extendable implementation, making it especially promising for real-world robotics applications where flexibility and maintainability are crucial. Relevant work in this field includes *Q-decomposition* [8], *Arbi-Q Command Arbitration* [9] and *GRACIAS* [10] with some robotic-focused research [11].

Problem Statement and Research Questions

Despite the promising capabilities of modular reinforcement learning, available research still needs to be expanded. Most existing approaches primarily use value-based reinforcement learning algorithms with a limited focus on robotic applications and evaluation in real-world scenarios. This creates an interesting research gap for investigating the potential of actor-critic-based algorithms in the context of complex, multi-step robotic tasks.

To advance the field of modular reinforcement learning, we propose a novel modular framework designed to combine multiple pre-trained reinforcement learning agents to solve complex tasks in real-world robotic applications. The agents are trained using actor-critic algorithms, specifically the PPO algorithm. They are combined using an arbitration agent that selects actions based on the agents' policy and value network outputs. This approach aims to provide a modular, interpretable, and maintainable system that can be easily extended and adapted to new tasks.

This thesis will address the following key research questions:

1. How effectively can an arbitration agent, utilizing only policy and value network outputs and cross-evaluation of actions, combine multiple specialized reinforcement learning agents to solve complex tasks in robotic applications?
2. How does the performance of the combined system compare to that of the individual agents?
3. How successful can this approach be transferred to a real-world robotic system?

Hypothesis, Objectives, and Contributions

This thesis makes the following hypotheses:

- A modular arbitration framework using actor-critic agents and arbitration based on policy and value network outputs can effectively address complex tasks in robotic applications.
- Cross-evaluation of actions using the other modules' policy and value networks will improve the arbitration agent's decision-making.
- No modifications to the used algorithm are required to implement the proposed framework.

- The performance of the combined system will be comparable to that of the individual agents.
- This approach can be successfully transferred to a real-world robotic system.

The primary objective is to outline the modular arbitration framework using existing algorithms and connect it to the current literature. Based on this, the framework is applied to a multi-step robotic task. This includes the design of the required modules and the arbitration agent, whose capabilities are evaluated and compared in a simulated environment. Ultimately, the approach is transferred to a real-world robotic system and evaluated.

Contribution

This thesis contributes to the advancement of research in modular reinforcement learning approaches, especially in real-world robotic applications. It provides a novel framework using actor-critic algorithms for combining multiple pre-trained reinforcement learning agents, allowing to solve complex tasks in a modular, interpretable, and maintainable way. The framework is evaluated in a simulated environment and transferred to a real-world robotic system, providing insights into the effectiveness and challenges of the approach. Additionally, the complete codebase and instructions are provided to enable future research in this area.

Thesis Structure

The remainder of the thesis is structured as follows.

Chapter 2 presents the state-of-the-art in handling complex, multi-step tasks and connects our proposed framework to those existing approaches.

Chapter 3 covers the fundamental concepts and background knowledge required to understand the research topic. It introduces the basics of reinforcement learning, the used PPO algorithm, modular approaches, and sim-to-real transfer.

Chapter 4 introduces the proposed modular arbitration architecture in detail. This includes the design of the modules, the arbitration agent, the state inputs, the reward structure, and the system integration.

Chapter 5 describes the actual experiment within the simulation environment, the description of the complex, multi-step task, and the design and simulation results of each agent.

Chapter 6 presents the zero-shot transfer of the trained agents to the real-world *Hiwonder JetAuto* robot and compares the results to the simulation results.

In Chapter 7, the research questions are revisited, the key contributions and findings are summarized and analyzed, and an outlook for future research is given, followed by the conclusion of this thesis.

2. State-of-the-Art

The following literature review outlines the current state of research in the main areas of interest for this thesis. This includes an overview of reinforcement learning in robotics and a comparison and delimitation of our modular arbitration framework from other approaches for solving complex tasks, especially modular approaches.

2.1 Reinforcement Learning in Robotics

Reinforcement learning has its fundamentals in the development of Markov Decision Processes by Richard Bellman in the 1950s [12] and was first framed in a computational context in 1961 by Marvin Minsky in his paper *Steps Toward Artificial Intelligence* where he discussed the first concepts which would later evolve into reinforcement learning [13]. With the introduction of temporal difference learning [14] in 1988 and Q-learning [15] in 1992, the first agents could learn without explicit environment models and from delayed rewards. This was accompanied by one of the first robotics applications for learning control policies for robot motion in 1990 [16], showing the potential for learning robotic tasks through trial-and-error, which would have been difficult to program traditionally.

More complex applications of Reinforcement Learning (RL) in robotics were limited due to Q-Learning only supporting discrete action spaces and using a lookup table for the state-action mapping, preventing high-dimensional state spaces. This changed with the groundbreaking results of the famous *Human-level control through deep reinforcement learning* paper by Mnih et al., introducing Deep Q-Network (DQN) and innovations like experience replay, target networks, and integration of deep convolutional neural networks [17]. Those new methods made DRL stable and applicable, starting a surge in popularity and discoveries.

DRL can handle high-dimensional inputs and apply end-to-end learning, allowing to directly map the state inputs like sensor or camera data to the control action outputs, with automatic feature generation [18]. This allowed the application of various robotic tasks like grasping, manipulation, locomotion, navigation and control. Furthermore, with the advancements in algorithms and simulation environments, agents can be directly transferred from simulation to the real world, reducing the time and cost for training and testing in real-world environments [19].

Despite those achievements, DRL has some fundamental challenges for robotic applications. One is the low sample efficiency, requiring millions of interactions even for learning simple tasks, making the training in real-world environments non-feasible [20]. Additionally, deep neural networks come with interpretability and explainability issues, making it difficult to retrace agent decisions. Current approaches have problems solving complex, multi-step tasks and are not very good in applying learned skills on related tasks [3]. Agents still have problems in applying their learned skills in new situations or different environments, indicating a lack of generalization and robustness [21]. In high-dimensional state and action spaces, as in robotic applications, current algorithms struggle with more complex tasks [3].

2.2 Approaches for Learning Complex Task

Solving more complex, multi-step tasks in robotics requires alternative techniques. The first fundamental approaches for making complex tasks more manageable were multi-task learning [22], reward shaping [23], and curriculum learning [24]. More modern, robotic-focused approaches, were introduced by meta-learning [25], imitation learning [26], and the combination of multiple of those techniques [27].

Hierarchical Reinforcement Learning (HRL) made the beginnings of decomposing complex tasks by using hierarchies of subtasks with temporal abstraction [4, 28, 29], with successful applications in robotic manipulation tasks [7, 30] and more complex multi-step manipulation tasks [31]. Besides advantages like improved learning speed [32] and improved skill transfer for new tasks [33], there are still limitations regarding designing the hierarchy, task decomposition, training stability, communication overhead, and increased state spaces.

Another approach is Modular Reinforcement Learning (MRL), which has seen a surge in interest in recent years. It decomposes complex tasks concurrently into sub-tasks, allowing for better reusability, scalability, and interpretability [34]. Russell et al. used Q-decomposition, where each module has its Q-value function, and a central arbitrator sums all Q-values and selects the action with the highest value [8]. Sprague et al. used the RL algorithm *Sarsa(0)* for training each module and the arbitrator. The arbitrator uses the same state inputs as the modules for the selection [35]. This was similarly implemented using DQN [9] and applied in a robotic context using SplitDQN [11]. A weighted joint Q-value is used by Gupta et al. in their framework *GRACIAS*, with the weights being calculated at each timestep by the arbitrator based on the current environment state [10]. Fundamental challenges for modular approaches are designing an effective arbitration mechanism to combine different modules and handling the communication overhead between modules

and arbitrators [36].

The approach researched in this thesis uses the neural network representation of each module's policy and value network as input for the arbitrator, resulting in a smaller state space than by using the current environment state. This approach is similar to the one used by Simpkins et al. [9] and Huang et al. [11]. The main difference is that instead of a value function based algorithm, an actor-critic algorithm is used, allowing for different state inputs derived from the modules. By implementing the arbitrator as a learning agent, a more effective arbitration mechanism can be achieved, similar to the framework *GRACIAS* [10].

3. Background and Preliminaries

The following chapter introduces the components for implementing our proposed modular arbitration framework. First, the background of RL, DRL, and methods for solving complex tasks and hierarchical approaches are covered. This is followed by a more in-depth analysis of modular approaches with their task decomposition and arbitration. For evaluating our framework, the requirements for a successful sim-to-real transfer to a real-world application are discussed, as well as the technical frameworks and tools needed.

3.1 Reinforcement Learning

In RL, an agent learns to make decisions by interacting with their environment through trial-and-error, receiving rewards as feedback for performed actions. The agent aims to maximize the cumulative reward [37]. One of the main advantages of using RL compared to classical methods is that the agent can be trained without any instructions on how to solve the task but only by guiding its actions using a reward signal.
This section overviews the fundamental concepts of classical RL and creates the foundation for DRL.

3.1.1 Markov Decision Process

The trial-and-error interaction with the environment can be stated using a Markov Decision Process (MDP), a model for sequential decision-making with uncertain results. The MDP is defined by the tuple (S, A, T, R, γ), with S representing the set of states the agent can be in, A the set of actions the agent can take, P the transition probability function, R the immediate reward function and γ as a discount factor.

At each time step t, the agent is in a state $s_t \in S$, selects an action $a_t \in A$, and receives a reward $r_t \in R$. The environment then transitions to a new state s_{t+1} according to the transition probability function $P(s_{t+1}|s_t, a_t)$. The Markov property specifies that the new state depends only on the current state and the selected action, not on previous states. In other words, each state must contain all details of previous interactions that influence the next state [37].

The goal in RL is maximizing the expected reward G_t, which is defined by equation (3.1)

for continuous tasks, meaning tasks without a terminal state.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{3.1}$$

The discount factor γ can take values between 0 and 1 to focus more on future or immediate rewards. With $\gamma = 0$, the agent focuses on immediate rewards, while higher values emphasize future rewards. With $\gamma < 1$, the infinite sum converges to a finite value [37].

3.1.2 Policy and Value Functions

The policy π is a function that maps states to actions and is used to decide the following action a_t of the agent based on the current state of the environment s_t.

The value function estimates the expected discounted return for a particular state. Value functions can be divided into state-value functions $v_\pi(s)$, equation (3.2), estimating the expected return starting in the current state s and then following the policy π, and action-value functions $q_\pi(s, a)$, equation (3.3), estimating the expected return starting in the current state s, taking action a, and then following the policy π. State-value functions are mostly used with deterministic policies, and action-value functions with stochastic policies due to their consideration of the expected return of different actions [38].

$$\begin{aligned} v_\pi(s) &= E_\pi \left[G_t | S_t = s \right] \\ &= E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \end{aligned} \tag{3.2}$$

$$\begin{aligned} q_\pi(s, a) &= E_\pi \left[G_t | S_t = s | A_t = a \right] \\ &= E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_D onet = s, A_t = a \right]. \end{aligned} \tag{3.3}$$

By using the Bellman equation, $v_\pi(s)$ can be written recursively, equation (3.4). This sets the mathematical foundation for learning long-term values by allowing us to estimate future rewards recursively and the usage of temporal differences, which will be explained in Section 3.1.3 [38].

$$\begin{aligned} v_\pi(s) &= E_\pi \left[G_t | S_t = s \right] \\ &= E_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi(s')]. \end{aligned} \tag{3.4}$$

For maximizing the expected reward, an optimal policy π^* needs to be found that maximizes the cumulative reward as shown in equation (3.5).

$$\pi^*(s) = \arg\max_a Q^*(s, a). \tag{3.5}$$

This shows that the optimal policy is based on an optimal value function, defined by equation (3.6).

$$V^*(s) = \max_\pi \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_t + 1)]. \tag{3.6}$$

Policy Iteration

The fundamental approach for finding optimal policies and value functions is based on the following two key steps [38]:

1. Policy Evaluation: by applying the current policy, the state-value function is computed using equation (3.4). The resulting value is compared with values that could be obtained by selecting another action in the current state using the action-value function, equation (3.3).

2. Policy Improvement: based on the policy improvement theorem, if $q_\pi(s, \pi'(s)) \geq v_\pi(s)$ for all states s, then the new policy π' is at least as good as the original policy. The new, improved policy is obtained by selecting the action that maximizes the action-value function $q_\pi(s, a)$, equation (3.7).

$$\begin{aligned} \pi'(s) &\equiv \arg\max_a q_\pi(s, a) \\ &= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \arg\max_a \sum_{s', r} p(s', r|s, a)[r + \gamma v_\pi(s')]. \end{aligned} \tag{3.7}$$

Value Iteration

A computationally more efficient version is achieved by skipping the policy evaluation and directly updating the value function using value iteration; see equation (3.8) [38].

$$\begin{aligned} v_{k+1}(s) &\equiv \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r|s, a)[r + \gamma v_k(s')]. \end{aligned} \tag{3.8}$$

Various approaches to finding the optimal value function exist, such as dynamic programming, Monte Carlo methods, and their combination, Temporal Difference learning, which will be introduced in the next section.

3.1.3 Temporal Difference

Temporal Difference (TD) is a model-free RL algorithm, updating estimated values of states or state-action pairs based on the difference between predicted and observed reward and was introduced by Sutton in 1988 [14]. They combine Monte Carlo and dynamic programming methods, using observed transitions and rewards to update value estimates. The value estimate is updated at every time step, not needing a complete episode, using the observed reward R_{t+1} and the current value function estimate $V(S_t)$, as shown in equation (3.9) [39].

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{3.9}$$

Over time, estimations should get closer and closer to R_{t+1}, meaning the state estimations are getting more accurate. The difference between the estimated and updated value is called the TD error, see equation (3.10), which is used in various RL methods for learning the value function, while being less optimal for updating policies [38].

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{3.10}$$

3.2 Deep Reinforcement Learning

DRL builds upon RL but replaces the tabular function approximators for the policy and value functions with deep neural networks. Using neural networks allows for automatic feature engineering and end-to-end learning. This allows the handling of high-dimensional data and generalization over unseen states. Using a cost/loss function, the parameters of the neuronal network, weights, and biases are updated using backpropagation and gradient descent optimization methods [40].

3.2.1 Artificial Neural Networks

An artificial neural network is composed of neurons. Each neuron has n input connections and processes the inputs according to equation (3.11). The inputs are weighted and summed, the bias is added, and an activation function is applied.

$$a_j = f(\sum_i w_{ij} a_i + b_j), \tag{3.11}$$

a_j being the output of neuron j, w_{ij} the weight connecting neuron i from the previous layer to neuron j, a_i the output from neuron i in the previous layer, b_j the bias term for

neuron j and f the activation function for non-linearity.

For approximating a function, the weights w_{ij} and biases b_j are adjusted to minimize the difference between the predicted and wanted output; see equation (3.12). This is done by optimizing a loss function, for example, mean-square-error (3.13) [40].

$$Y_{pred} = f(X, \theta), \tag{3.12}$$

$$L(\theta) = \sum_i (Y_{pred} - Y_{true})^2. \tag{3.13}$$

For updating the weights, the gradient of the loss $\nabla L(\theta)$ is calculated and backpropagated through the network, meaning from end to start. Stochastic gradient descent is combined with batch learning, based on the equation (3.19) in the following section, for calculating how much the weights are updated.

The actual neural network uses certain fundamental building blocks connected feedforward. The main building blocks, which will also be used in this thesis, are feedforward, convolutional, and recurrent neural networks. Neural networks are used as function approximators by learning the parameter set θ, which contains the weights of all neurons inside the network.

Feedforward Neural Network
Feedforward Neural Network (FNN) consists of multiple neurons arranged in an input layer, N hidden layers, and an output layer. All neurons are fully connected to the next layer, and their output is calculated according to equation (3.11).
FNN can be used to approximate any function. For the usage in DRL, this includes the value function $V(s)$ for mapping states to value estimates, policy $\pi(a|s)$ by outputting action probabilities or values, and Q-function by mapping state-action pairs, to expected returns [40].

Convolutional Neural Network
Convolutional Neural Network (CNN) can be used for processing matrix-like data, especially images. In the context of robotics, this allows the processing of visual inputs, for example, from cameras. They are composed of two primary types of layers: convolutional and pooling.

The convolutional layer is the core building block that learns a square matrix of neurons, called filters, that are applied to the input matrix g in a convolution operation and

search for patterns, called localized features [40]. This is expressed mathematically by equation (3.14).

$$h[x,y] = f\dot{g}[x,y] = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} w[i,j]g[x+i,y+j]. \tag{3.14}$$

The advantage of convolutional layers is that they require a smaller number of weights than fully-connected layers, as the weights are shared across the input. This allows the network to learn spatial hierarchies of features.
Applying a pooling layer can reduce the data dimensions by applying non-linear downsampling. The most common pooling layer is the max-pooling layer, which uses a filter of size $[m \times m]$ and outputs the maximum value inside this filter. While this reduces the data size, it also loses information about exact feature positions [38].

Recurrent Neural Network
Using Recurrent Neural Network (RNN) allows the extraction of temporal features from a sequence of inputs. Using a hidden vector, resembling the memory, the network can remember previous inputs and update the states accordingly. This is especially useful for processing time-series data, for example, in the context of robotics for processing sensor data [40].

3.2.2 Stochastic Policy

Policies can be differentiated based on their action selection. Deterministic policies directly map a state to a single action, whereas stochastic policies assign probabilities to those actions and sample from their distribution. While the direct mapping by deterministic policies can be easier and more stable to learn, stochastic policies have the advantage of inherent exploration due to the randomness in the action selection and a better handling of noise. In the following, stochastic policies will be analyzed further, as they are usually the basis for policy gradient and actor-critic methods [41].

Stochastic policies assign probabilities to each action in the current state, for example Gaussian distributions in a continuous action space. The policy generates a mean and standard deviation and uses it to create a Gaussian distribution, see equation (3.15).

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}. \tag{3.15}$$

From this distribution an action is sampled using the reparameterization trick, which

describes a differentiable sampling approach, thereby allowing backpropagation during training [42]. The mean and standard deviation from the policy network is also used for calculating the log probability, see equation (3.16), which is needed for policy updates and training.

$$\log \pi(a|s) = -\frac{(a-\mu)^2}{2\sigma^2} - \log(\sigma) - \frac{1}{2}\log(2\pi) \tag{3.16}$$

The mean represents the most preferred action for the given state, while the standard deviation represents the uncertainty within the current state [41]. This will be used in Section 4 for designing our modular arbitration framework.

3.2.3 Policy gradient methods

Policy gradient methods, a subclass of policy-based methods, optimize a policy directly instead of learning a value function. The policy is represented as a parameterized function π_θ for this. The parameter set θ usually represents the neural network weights used to approximate the function. While the policy can be deterministic $a = \pi_\theta(s)$ or stochastic $a \sim \pi_\theta(a|s)$, the following explanation will focus on a stochastic policy like used in PPO which outputs a probability distribution of actions: $\pi_\theta(s) = \mathbb{P}[A|s;\theta]$.

The goal is to tune the policy parameters to sample actions that maximize the return more frequently. For evaluating the policy, we use the objective function $J(\theta)$, which measures the expected reward [43], see equation (3.17).

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)], \tag{3.17}$$

where $\tau = (s_0, a_0, ..., s_{T+1})$ represents a trajectory containing a sequence of states, actions and state transitions. The reward for this trajectory is calculated using the cumulative reward function $R(\tau)$. A trajectory differs from an episode because its last state must not be final [38].

The objective function can also be expressed in terms of the policy; see equation (3.18). By multiplying the total reward accumulated along a trajectory $R(\tau)$ with the probability of taking that trajectory under the current policy parameters θ, we get the weighted return for this specific trajectory. Summing all weighted returns results in the expected return, the average return of all trajectories weighted by their probabilities.

$$J(\theta) = \sum_\tau P(\tau;\theta)R(\tau). \tag{3.18}$$

The goal is to maximize the expected reward for the objective function, making it an optimization problem that can be optimized using gradient ascent. This is achieved by updating the θ parameter in the direction of steepest increase using the gradient of the objective function $\nabla_\theta J(\pi_\theta)$ multiplied with the learning rate α, see equation (3.19).

$$\theta_{k+1} = \theta_k + \alpha_k \nabla_\theta J(\pi_\theta)\big|_{\theta_k}. \tag{3.19}$$

For calculating the gradient, equation (3.18) is reformed into equation (3.20) and further simplified, allowing to express $P(\tau|\theta)$ as a product of probabilities of taking an action that lead to higher returns , see equation (3.21).

$$\nabla_\theta J(\theta) = \sum_\tau \nabla_\theta P(\tau|\theta) R(\tau) = \mathbb{E}_{\tau\sim\pi_\theta}[\nabla_\theta \log P(\tau|\theta) R(\tau)], \tag{3.20}$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right]. \tag{3.21}$$

The policy parameters can now be optimized by:

1. Sampling trajectories by following the current policy π_θ;
2. Computing the gradient of the objective function based on equation (3.21). By not using all but only the m sampled trajectories, the policy gradient is estimated using the sample mean in equation (3.22);

$$\nabla_\theta J(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau). \tag{3.22}$$

3. Updating policy parameters using equation (3.19).

Policy gradient methods are used in algorithms like Reinforce, Trust Region Policy Optimization (TRPO) and PPO. The advantages are that they can learn a stochastic policy that automatically implements an exploration and exploitation trade-off. By estimating the policy directly, the dimension is reduced. The function approximation using a neural network allows for the natural handling of continuous action spaces. Limitations are the high variance in gradient estimates, sample inefficiency, sensitivity to hyperparameters, and converging to local maximum instead of global optimum [44].

3.2.4 Actor-Critic Methods

Actor-critic methods combine the strengths of policy gradient and value-based methods by learning two function approximations with the actor representing the policy π_θ and the critic the action-value function $\hat{q}_w(s, a)$. The actor updates its policy network parameters using policy gradient methods as introduced before; see equation (3.19). In contrast, the critic updates its action-value network using TD learning, see equation (3.10) [41].

Based on the current state S_t, the actor generates the action A_t, and the critic uses this action with the current state for computing the action-value function, the Q-value $\hat{q}_w(s, a)$. After the action, A_t is executed, and the new state S_{t+1} and reward R_{t+1} are available, the actor can update its policy parameters θ using the Q-value, see equation (3.23). This is the same approach as in the previous section using policy gradient methods [37].

$$\theta = \theta + \alpha \nabla_\theta (\log \pi_\theta(s, a)) \hat{q}_w(s, a). \tag{3.23}$$

Using the updated policy π_θ, the actor generates the next action A_{t+1} which the critic uses for updating its value parameters w, see equation (3.24).

$$w = w + \beta \delta \nabla_w \hat{q}_w(s_t, a_t), \tag{3.24}$$

where β represents the learning rate, $\nabla_w \hat{q}_w(s_t, a_t)$ is the gradient for the value function and δ the TD error (see equation (3.10)). This shows that the actor updates policy parameters to increase the expected reward. At the same time, the critic provides feedback on the quality of the actions taken, which guides those policy updates directly [37].

Using an advantage function instead of TD error reduces the variance of policy gradient updates and allows to focus on actions that yield higher returns than average. The advantage function calculates how much better it is to take a specific action compared to the average, policy-based action at a given state by subtracting the value function from the Q-value function, see equation (3.25). By subtracting the value function as a baseline, the cumulative reward is smaller and, thereby, the gradients, resulting in more stable updates.

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t). \tag{3.25}$$

Instead of learning two neural networks for Q value and value function, which is very inefficient, the equation can be rephrased using the Bellman optimality equation, see equation (3.26), and rewritten as equation (3.27) [37].

$$Q(s_t, a_t) = \mathbb{E}[r_{t+1} + \gamma V(s_{t+1})]. \tag{3.26}$$

$$A(s_t, a_t) = r_{t+1} + V_{\pi_\theta}(s_{t+1}) - V_{\pi_\theta}(s_t). \tag{3.27}$$

Actor-critic methods have the advantage of better sample efficiency, resulting in less data requirements, and can be used for continuous and discrete action spaces. Limitations are the high variance in policy gradient updates, slow convergence, and difficulties in tuning hyperparameters [37].

3.3 Proximal Policy Optimization

The PPO algorithm was developed by Schulman et al. in 2017 and became one of the most used RL algorithms. It is an actor-critic method, approximating a policy function for decision-making, which maps states directly to actions [45]. It is often wrongly classified as a Policy Gradient Method (PGM), which is incorrect as it does not use the gradient directly but instead a surrogate objective [46]. The two core variants of PPO are PPO-Penalty and PPO-Clip. In the following, PPO-Clip is analyzed in more detail, as this is the variant used in this thesis.

Policy gradient methods have the limitation of unstable policy updates due to taking too large steps during improving the policy. This was addressed by TRPO by introducing a penalty in the form of the Kullback-Leibler divergence, which limits how much the policy can change in a single update [47], expressed by equation (3.28).

$$\max_\theta \mathbb{E}_t \left[A(s_t, a_t, \pi) \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} - \beta KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right], \tag{3.28}$$

where $\pi_{\theta_{old}}$ it the previous policy version and β the regularization parameter controlling the strength of penalty. While TRPO provided stability improvements, the practical usage was limited by the computationally inefficient use of second-order optimization, resulting in the development of PPO as an easier-to-implement alternative [45].

The PPO algorithm uses separate policy and value networks, with the value network computing the advantage function for policy updates and the policy network calculating the probability density function for deciding on the following action based on the current state. It uses a clipped surrogate objective function; see equation (3.29). This limits policy changes to a small range to avoid too large weight updates, making the policy more likely to converge to an optimal solution. More precisely, any advantage obtained is ignored if the probability of an action changes by a factor smaller than $1 - \epsilon$ or larger than $1 + \epsilon$ [45].

$$L^{clip}(\theta) = \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)], \tag{3.29}$$

where clip is constraining the policy, ϵ as clipping parameter and r_t as probability ratio,

see equation (3.30), L^{clip} as loss function and $\hat{A}_t$ the estimated advantage function, see equation (3.31).

Taking the minimum of the clipped and unclipped objective results in a lower bound, resulting in changes of the probability ratio r_t being ignored when the change improves the objective but included otherwise, see equation (3.30). The probability ratio compares the likeliness of the current policy taking an action to the old policy [45].

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}. \tag{3.30}$$

The advantage function, introduced in equation (3.25), is extended by using Generalized Advantage Estimate (GAE). In its basic form, the advantage function uses the TD error, which has a fixed number of steps looking ahead. Alternatively, it can use the Monte-Carlo method to calculate the estimation error till the end of the episode. GAE is a hybrid combination, using the parameter λ for adjusting the look ahead steps, see equation (3.31). With $\lambda = 0$, the advantage function represents the TD error, and with $\lambda = 1$, the Monte-Carlo method [48]. A typical value for PPO is $\lambda = 0.95$, giving a balance in considering only immediate rewards and all future rewards.

$$\hat{A}_t = \delta + (\gamma\lambda)\delta_{t+1} + ... + ... + (\gamma\lambda)^{T-t+1}\delta_{T-1} = \sum_{l=0}^{\infty}(\gamma\lambda)^l * \delta_{t+l}, \tag{3.31}$$

where δ_t as temporal difference error, see equation (3.10).

The objective function is extended by two additional terms: entropy and value loss. The entropy loss encourages exploration of the policy network by promoting policies that have more randomness in their actions. If a continuous action space is used, the entropy loss is not necessarily required because the actions are sampled from a Gaussian distribution, which is already encouraging exploration, see also Section 3.2.2. In case of a distinct action space, the *Shannon Entropy* is calculated and added to the loss function [49].

The value network is trained using the value loss function by minimizing the squared difference between the predicted value and the target value over multiple samples, see equation (3.32) [49].

$$L^{VF}(\theta) = \mathbb{E}_t[(V_\theta(s_t) - V_t^{target})^2]. \tag{3.32}$$

The complete objective function for PPO is shown in equation (3.33).

$$L^{TOTAL}(\theta) = \mathbb{E}[L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 S[\pi_\theta](s_t)], \tag{3.33}$$

where

- $L^{VF}(\theta)$: value-loss function: $(V_\theta(s_t) - V_t^{target})^2$
- $S[\pi_\theta](s_t)$: entropy bonus for encouraging exploration
- c_1, c_2: weighting factors

The main advantages are that PPO only requires a first-order optimization and can use standard stochastic gradient descent for the optimization. It can re-use the same data for multiple epochs while maintaining stability, similar to the training of neural networks, improving sample efficiency [49]. While PPO requires less hyperparameter tuning than other RL algorithms, finding good ones is still a significant challenge, especially if using it in more complex simulations where it takes a long time till the current hyperparameter can be evaluated.

The step-wise algorithm, as implemented by the reinforcement learning library used in this thesis, *skrl* [42], is described below.
For each iteration:

1. While the agent interacts with the environment using the current policy π collect the following data for the rollout memory:
   set of states s, actions a, rewards r, dones d, log probabilities $logp$ and values V from V_ϕ
2. Use the advantage function, equation (3.31), to estimate returns R and advantages A
3. If required, compute the entropy loss S
4. Compute policy loss using current and previous policy, equation (3.29)
5. Compute value loss, equation (3.32)
6. Optimize the total loss, equation (3.33)

3.4 Approaching Complex Tasks

Complex tasks in RL are defined by delayed rewards, sequential decision-making, long-term dependencies, and a hierarchical task structure, requiring sub-tasks to be completed to achieve the overall goal. Due to the delayed rewards, the agent must perform multiple actions before receiving feedback. In sequential decision-making, a series of actions is required in a specific order to achieve the goal. Long-term dependencies refer to the possible lasting impact of previous actions on the final outcome, requiring the agent to consider the long-term effects of the selected actions [50]. In the following, multiple approaches are presented for dealing with the challenges of complex tasks. Those approaches can be seen as fundamental blocks developed to expand the capabilities of DRL, allowing the integration into more sophisticated architectures or use as standalone solutions.

The first concept, multi-task learning, was introduced in 1997 by Caracuna, showing that learning similar tasks together can improve learning and generalization between tasks [22]. This is caused by the model having to find common features across multiple tasks, requiring a more evolved feature detection. While there is the risk of negative transfers between tasks or domination by one, this approach is still applied in state-of-the-art applications, with the newest approach being the development of neural network architectures that access certain areas of the neural network to share relevant features while maintaining task-specific specialization [51]. In robotics, Yang et al. successfully used multi-task learning to teach a robot multiple manipulation skills while preventing the risk of negative transfers inside the network by implementing a soft modularization technique. This uses a separate routing network that assigns certain parts of the base network to different tasks, thereby preventing gradient interference during learning from other tasks [52].

Reward shaping can be used to ensure a faster and more guided learning process. With rewards guiding the agents' learning, shaping it the right way helps them learn faster. This fine-tuning of the reward function does not change the optimal policy if the transformations are potential-based or positive linear transformations [23]. While incorrect reward shaping could lead to unintended behaviors, using the potential-based shaping suggested allows for a notable speed-up in learning. The guaranteed effectiveness makes reward shaping a very active research area with various variations. Using intrinsic motivation in hierarchical reinforcement learning, as done by Kulkarni et al., represents an alternative form of reward shaping for guiding the agents towards their sub-goals [30]. Inverse reward design, while primarily a theoretical approach, analyzes how to ensure that agents understand the implicit parts of the reward function, also when deployed in new environments, making it interesting for real-world application [53]. Fu et al. proposed an approach for deriving a reward function automatically from data by supplying ranked events and states, preventing poorly designed reward functions [54]. Using a dataset labeled as optimal allowed Wolf et al. to train a deep neural network to generate a shaped reward for a complex autonomous driving task, significantly improving the performance of the RL algorithm [55].

Curriculum learning can be applied to deal with complex, longer tasks. By progressively increasing the task complexity during training, an agent can start with learning fundamental object interactions and progress to more complex manipulations using the learned knowledge and skills, enabling it to learn tasks that would be too difficult to learn directly. The sample efficiency improves if the agent starts with more solvable tasks. This curriculum learning approach closely resembled human educational practices and was applied for training neural networks in a gradually increasing complexity [24]. Newer approaches apply automated curriculum learning for agents based on their learning process [56].

Meta-learning is a more fundamental approach for improving the learning process and thereby increasing the sample efficiency. It focuses on "learning to learn" and quickly adapting to and solving new tasks. This allows meta-RL agents to adapt to dynamic environments and new tasks quickly [25]. It positively influences various RL approaches and methods. By leveraging experiences from related tasks, the generalization improves, and learning becomes faster, resulting in an improved sample efficiency. Current research tries to create even more intelligent and adaptable agents by utilizing generative models for letting the meta-learner generate its training tasks [57]. Zou et al. combined meta-learning with reward shaping by training a neural network to generate a shaping reward even for new tasks, guiding and shortening the main agents learning time [58]. A very interesting publication is by Yu et al. and their *Meta-World*. This benchmark is for evaluating meta-learning algorithms on 50 robotic manipulation tasks to enable real meta-learning algorithms. Current approaches are trained on very similar tasks. This broader spectrum of tasks should support algorithms with even more generalization capabilities [3].

To shorten the long training times of models and also improve the generalization, transfer learning focuses on reusing knowledge learned in one task in a new task by transferring information about policies, value functions, state representations, or entire models. This allows for a speed-up in learning, better generalization, and lower data requirements. Transferring knowledge from a simulation to a real-world application reduces the requirements of collecting costly real data. In addition, transfer learning can be used to transfer knowledge from one task to a related task. Current research focuses on learning transferable policies in deep neural networks, transferring knowledge across tasks in a hierarchical structure, and transferring from simulation to the real world. In robotics, this is often applied for fine-tuning pre-trained visual features for new tasks [59].

The vast number of different approaches and research shows the high activity in the area of DRL and the growing focus on robotic applications. Those techniques are often even more effective if multiple are used together, provided that the correct methods for the specific task are combined [27]. Besides all advancements, those approaches still struggle with long-horizon, multi-stage problems. For this, techniques like hierarchical and modular reinforcement learning are more suited, as introduced in the following sections.

3.5 Hierarchical Approaches

HRL decomposes complex tasks into hierarchies of subtasks with multiple levels of temporal abstraction, as introduced by Sutton et al. with the *Options framework* [4]. Temporal abstraction describes policies that operate over multiple time steps using high-level controllers and low-level sub-policies, which allows handling long-horizon problems

with improved sample efficiency. Another fundamental hierarchy approach is *Feudal Networks*, introduced by Dayan et al. [28], which uses managers to set goals that workers execute. Parr et al. provided a framework for defining hierarchical policies, called *Hierarchical Abstract Machines* [29], and demonstrated their use for complex control tasks.

Options were applied to robotic manipulation tasks and showed improved performance and generalization compared to normal RL methods [30]. Further developments were made with *Concept Networks* by Aditya Gudimella et al., allowing for better re-usability by defining sub-agents as concepts that can be integrated into a higher-level agent in a modular way. Those concepts can be trained on a limited state space, resulting in more data-efficient learning [7].

Multiple practical implementations and research have shown several advantages of applying hierarchical approaches in robotics. Those are improved learning speed for manipulation tasks [32], skill transfer between robots [33], complex multi-step manipulation through hierarchical planning [31] or handling of complex quadrupedal locomotion by using high-level policies for path planning and low-level policies for motor control by combining hierarchical policies with modular architectures [60].

The main concepts in HRL are the hierarchy of tasks, temporal abstraction, hierarchical policies, and value functions. The hierarchical structure uses high-level tasks to represent abstract goals and low-level tasks for concrete actions or sequences of actions, forming a tree-like structure. Temporal abstraction allows learning at multiple time scales, with the high-level task being inactive while the low-level task executes its actions. This allows for a more efficient learning of long-horizon problems and reduces computational requirements. High- and low-level tasks have their own policies and value functions, which are trained separately and interact through the information flow between the hierarchy levels. The high-level policy selects the subtask, and the low-level policy executes the actions. The value functions are used for credit assignment and learning the value of the high-level tasks.

Some of the open challenges in HRL are the design of the hierarchy, which requires expert knowledge and, in current approaches, always a manual specification and task decomposition. For successful coordination, the information flow between the hierarchy levels has to be defined and balanced, together with the credit assignment across those levels. The training of the hierarchical policies can become unstable, and there are limitations in scaling the approaches to more complex or deeper hierarchies [60].

3.6 Modular Approaches

Similar to many fields, especially modern software engineering, a modular approach in reinforcement learning has advantages like reusability, easier scalability, and better interpretability. There are various approaches and frameworks for modular solutions with an increasing interest in recent years [34]. MRL decomposes complex tasks concurrently into a collection of sub-tasks with their own goals, distinguishing from HRL which decomposes tasks temporally. Each sub-task with its agent represents a specialized module and can be trained independently. Multiple modules can be combined using an arbitration function that evaluates the output of each module and selects an action, either from one of the modules or by generating a new one [9].

Multi-Objective Reinforcement Learning (MORL) differs from MRL as the focus is on optimizing multiple, potentially competing goals. The learning is done within a single policy, focusing on balancing objectives, not coordinating and integrating them. Multi-Agent Reinforcement Learning (MARL), while involving multiple agents, focuses on independent agents interacting with each other, with each agent technically running the same code.

One of the first modular approaches was proposed by Russel et al. in 2003 by using modules with their own Q-value function, which a central arbitrator combines by summing the Q-values. The sums are interpreted as the knowledge of each module, equation (3.34). The result is a joint Q-value with the arbitrator selecting the action with the maximum value [8].

$$Q_{joint}(s, a) = \sum Q_i(s_i, a). \tag{3.34}$$

Limitations are that the reward scales for each module must be compatible with the summation, and the modules need to be trained for similar goals; otherwise, the joint Q-function will select non-optimal actions.

Summing the Q-values represents a utility fusion approach, which is one of the main arbitration strategies. It is defined by applying a function to the individual utilities of pre-trained agents to solve a global problem; see equation (3.35).

$$Q^*(s, a) \approx (Q_1^*(s_1, a), ..., Q_n^*(s_n, a)). \tag{3.35}$$

This decomposition can lead to a suboptimal solution because it ignores interactions between subproblems. A suggested solution is calculating a correction term using a neural network, which is added to the output of the utility fusion to approximate a globally optimal solution [61].

A more advanced approach was introduced by Simpkins et al. in 2019 with the Arbi-Q command arbitration algorithm, which uses a DQN to learn which module to select under certain state conditions, allowing the usage of modules with different reward scales. The arbitration agent and modules can use state abstractions, meaning only a subset of the full state representation, and the modules can be trained independently [9]. Using DRL for training the modules and the arbitration agent is a similar approach as will be used in our proposed framework, with the main difference being the state inputs, class of RL algorithm, and the application of the framework on a robotic system. Those approaches show that the main required steps for solving a complex task with a modular method is decomposing the task into sub-tasks and designing an arbitration mechanism.

While there were some attempts for automated task decompositions, among others by applying the mathematical discipline category theory for a theoretical definition [62], all actual implementations still require domain expertise and human knowledge.

For selecting the action of the agent, the individual modules utilities need to be combined using an arbitrator. The main arbitration approaches are command arbitration, command fusion and utility fusion. In command arbitration, only one module's action is selected and executed. The selection can be based on a voting approach, where each module votes for an action and the action with the most votes is selected, on confidence-based selection, where the module with the highest confidence in its action is selected, or by the arbitrator selecting an action based on its state inputs. In command fusion, the arbitrator generates a new action by combining the proposed actions by the modules. The composite action is created by weighted averaging or neural networks. Using utility fusion, the value estimates for each action from the modules are combined, for example the Q-values, and the action maximizing the combined utility is selected. A last approach is learning-based arbitration, where the arbitrator learns to select an action based on the inputs from each module [61].

3.7 Sim-to-Real Transfer

Sim-to-real describes the transfer of RL agents from the simulated environment into the real world, making it essential for the application in real-world robotics. The following evaluates the challenges and requirements for a successful transfer and relevant research in this area.

Early attempts to transfer algorithms trained in a simulated environment to the real world failed due to computation and simulation limitations, variations in the real world, differences in the sensors, incomplete system identification, and unmodeled physical effects. This issue became known as the reality gap [63].

Initially, research focused on closing the reality gap by improving the simulation environment. While simulation environments are becoming more realistic, they still cannot capture all complexities of the real world, such as physics, lightning, sensor noise, and especially various dynamics. Modeling all factors would also result in a slow and overly complex simulation. Conveniently, Jakobi et al. showed in 1995 that just by adding noise to the simulation, a successful transfer of simulated controllers to the real world could be achieved [64]. This was further formalized by Tobin et al. with domain randomization, the usage of randomized rendering in the simulator [63]. Other work further decreased the reality gap by applying improved system identification in the simulation and more accurate models of the robot dynamics [65], with Kelchtermans et al. achieving a zero-shot sim-2-real transfer for visual navigation by training the agent in an empty simulated environment but applying randomized augmentations like different lightning and camera effects like noise and blur [66]. This confirms that a photo-realistic simulation is not necessary but variability in the collected data used to train the algorithms [63].

Visual domain randomization is a main factor for successful sim-to-real transfer, meaning randomizing colors, textures, lighting, and object positions. The randomization amount is ideally applied gradually over time to help the network learn [63]. Other aspects focus on domain randomization, which includes the visual domain but also changes physic properties and adds noise to the sensors. A more exact system identification or adding noise and random delays to the actuators can improve the transfer [65]. Due to the visual domain randomization, CNNs are forced to learn more robust and generalizable patterns by focusing on essential features, which improves the generalization capabilities and reduces the risk of overfitting.

While applying the above techniques, a zero-shot transfer, meaning no additional training on the target system is required, already shows their effectiveness, there are still various challenges in real-world robotic applications using only RL. To achieve a maximum reliable result, real-world data is still needed to fine-tune the agent. Collecting this data is expensive and time-consuming, especially regarding the amount of data needed with DRL algorithms and their low sample efficiency. Real-world environments are much more unpredictable than any simulation, especially with human interactions. Additionally, non-stationary dynamics like changing environments, hardware degradation, and variable physics like friction can further complicate the transfer. The usage of neural networks results in the decision process of the agent becoming a black box and difficult to interpret, making it very difficult to ensure the agent behaves responsibly and ethically or to accurately predict its behavior in certain situations [67].

Those limitations are why there is still no utilization of fully RL based robotic systems in

the real world. However, with the first agents used in applications like healthcare, news recommendations, and gaming, the first commercialized robotic systems should follow soon [68].

3.8 Simulation Environments and Frameworks

The following section describes the requirements for the simulation environment and frameworks needed to develop reinforcement learning-based agents for solving robotic tasks and transferring them to a real-world system. Additionally, the software selected for this thesis is justified.

As a first step, a model of the robot is required, which contains the basic structure, properties like joints, links and their relationships, and physical properties like inertia and mass. The model and other required objects like scenery or obstacles can be created using modeling tools like Blender or FreeCAD and are then usually translated into a Unified Robot Description Format (URDF) file, which allows loading it into the simulation environment.

The simulation environment is the platform where the agent is trained and tested. It must provide a physics simulation, rendering capabilities, and support for specific sensors like cameras or lidar. An important aspect is the performance of the simulation, the possibility of training multiple agents simultaneously, and support for GPU acceleration. A programming interface is required for executing a RL algorithm, and a certain community size is always helpful. Common platforms are PyBullet, Unity, Gazebo, and Isaac Sim, and Isaac Sim is used in this thesis. This choice is based on the close integration with Compute Unified Device Architecture (CUDA), NVIDIA's parallel computing platform using GPU acceleration, outperforming CPU-based implementations [69]. Additional factors were the availability of RL frameworks and tutorials and the NVIDIA Jetson Nano compute system used on the robot for the real-world application [70].

A framework is required to develop and run the RL algorithm inside the simulation environment. It must be able to communicate with the simulation environment for accessing data and manipulating objects and have features for running the training simulation with parallel execution and support for different RL libraries. This thesis uses Isaac Lab, an open-source framework specifically built on Isaac Sim (used to be called Orbit). It is implemented as a Python library, can easily access Isaac Sim, supports all important sensors, and supplies a wrapper for training with multiple agents in parallel [71].

The last building block for developing the agent is the RL library. The actual algorithms

like PPO are implemented here. Prominent choices are Stable Baselines, RL Games, RSL RL, and skrl. The choice of skrl was made because of the modular design of the algorithms, which enables a more custom implementation, especially for the neural network. Other libraries often focus on using configuration files for setting up the neural network and the training process, which is unsuitable for complex multi-stream networks that have to fuse different sensor data [42].

Figure 1 shows the interplay between simulation environment, RL framework and library. The framework initializes the simulation environment with its configuration. After this, at every time step the simulation environment send the current environment state to the framework which passes it on to the agent implemented using the RL library. The agent returns an action which is applied by the framework and the simulation environment is updated again. This repeats till a termination signal. For updating the policy and value functions, trajectories are stored as batches inside a memory created by the RL library. Those trajectories are then used for the learning.

For the real-world system, the Robot Operating System (ROS) framework is used. This is a middleware for communicating between robot components like sensors, actuators, and the control system. This choice is due to its wide usage in the robotics community and the already existing implementation for the robot used in this thesis [72].

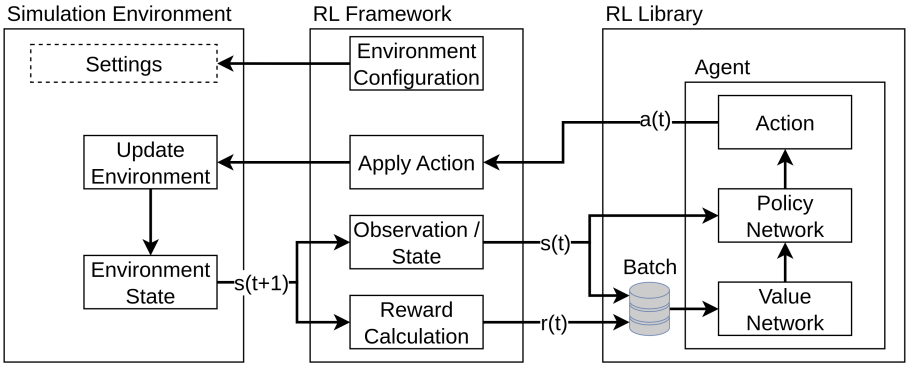


Figure 1. Connection between simulation environment, reinforcement learning framework and library

4. Modular Arbitration Framework

4.1 System Overview

The proposed modular arbitration framework combines multiple pre-trained actor-critic modules using a specialized arbitration agent. The arbitrator is a learning agent who uses each module's internal neural network representations — specifically the outputs of their value and policy networks — as state inputs. This allows a minimized state input compared to other modular or hierarchical approaches while fully utilizing the learned knowledge by each module.

The arbitration agent makes a discrete decision at each timestep, selecting which module's action to execute. This selection is based on the state inputs from each module's value and policy networks, allowing the selection of a different agent at each timestep without requiring an explicit termination state by the active agent. While the arbitration agent could be based on any reinforcement learning algorithm that supports a discrete action space, this implementation will utilize PPO due to its stability and wide use in robotic applications.

By using only the internal representations of each module, the arbitration agent can be trained without additional domain knowledge and without requiring access to each module's full environmental state or sensor inputs. This allows for a more general approach and easier integration of new modules. The arbitration agent is trained inside a simulation environment and can then be transferred to the real world. This arbitration framework builds upon other modular reinforcement learning approaches, but with a focus on utilizing the neural network representations of actor-critic algorithms.

For learning the selection strategy, the arbitration agent uses its own reward function, which allows it to consider certain conditions or constraints that are not directly related to the sub-tasks. This can be used to adapt the arbitration agent's strategy to specific scenarios, like preventing the selection of certain modules under specific conditions.

In the following, the actual implementation and functionality of this design is outlined.

4.2 Pre-Trained Modules Architecture

Overview

The modular architecture's foundation is specialized modules trained on one specific task. This task can be a sub-task from a multi-step, complex task or just a specific skill the agent should possess. Each module needs to be designed in a way that allows training the respective agent on its own to obtain optimized policies for each module. By focusing each module on only one well-defined task instead of multiple, the training is more efficient, interpretable, and a better performance can be obtained. For example, a possible complex task might be to navigate an unknown area, with one module responsible for exploring the area and another for avoiding obstacles. The separation into modules follows principles that are applied in other engineering areas, such as software engineering.

Sub-Task Composition

The actual sub-task composition is a manual step requiring domain knowledge. A starting point is to identify the complex task's main objectives and differentiate between the sensor inputs required. Each sub-task needs to be defined in a way that allows training the respective agent independently.

Architectural Requirements

Certain requirements must be met for the modules to be usable within the arbitration agent. With the value and policy networks of each module being used for generating the state inputs during arbitration, each agent has to be implemented and trained using an actor-critic method as introduced in section 3.2.4. Examples would be A2C, A3C, and PPO. The policy network generates actions based on the current state, with the probability density function allowing to compare those. The value network returns the expected cumulative reward, representing how promising this state appears to the module.

Because the value network is used as input for the arbitration agent, it can only use states that are also available during interference, just like the policy network. Usually, this is not required for actor-critic methods because the value network is only used during training, meaning additional values that are not accessible during interference can be supplied.

The value network should be designed as a state-action value network for optimal accuracy, as done in most practical implementations. This means it receives the last action or actions as part of its input states. Based on those inputs, the value function rates the agent's current state by returning the expected cumulative reward. By adding the last action as input, once this agent is used as a sub-agent in the arbitration approach, the value function of each agent can be used with the proposed actions of the other agent's policy network, generating

an additional input for the arbitration agent. This additional input can be interpreted as the expected cumulative reward by this agent's value network if executing the action of the other agent. As a practical example, supplying the action of the module, which explores the environment, to a collision avoidance module's value network, could show if the proposed action would result in a collision or is seen as acceptable by this module.

A similar cross-evaluation approach is implemented for the policy network for the actions of modules with the same action space by using the log probability density function of each other module, rating how likely this action would have been chosen by this module. This allows for an additional verification of the proposed actions and can prevent the execution of actions that are seen as risky by the other agents.

The modular approach allows for easy integration of new modules. Modules can share the same action space, allowing cross-evaluation, or generate different actions if required. This flexibility allows for a wide range of applications and use cases.

Training

By solving the sub-tasks independently in a modular way, each module can be developed and trained separately, reducing the complexity of the overall task. This reduces the risk of negative interference between the sub-tasks and allows for easier debugging and optimization of each module. Due to the faster training times of smaller agents, necessary adjustments for a successful simulation-to-real transfer can be made without the need to retrain each module.

It is essential to ensure that each module achieves robust and reliable performance in its specific task. This is crucial for the overall system as the arbitration agent depends on the reliability of each module.

4.3 Reinforcement Learning Based Arbitration Agent

The arbitration agent is used to combine the skills of pre-trained modules to solve a complex task. For this, it selects at each timestep which module's actions to execute, serving as the coordinator for the modules. The arbitration agent is implemented using a reinforcement learning algorithm that supports a discrete action space, for example, Q-learning, DQN or PPO.

4.3.1 State inputs and Action Generation

The state inputs for the arbitration agent are obtained from the policy and value networks of the specialized modules. Those are different evaluations of the probability density function of each policy network and the value estimates of each value network. For modules sharing an action space, each proposed action can be cross-evaluated by the policy and value network of the other agents.

Policy network
If the modules share an action space, the actions and log probability function of the policy networks are used to cross-evaluate (ce) the likelihood of executing the other agents actions. This gives a first measure how similar and thereby safe the respective actions are, see equation (4.1). For example, the collision avoidance module could evaluate the proposed actions of the exploration module to indicate actions that could result in a collision. As each action inside the action space receives their own probability rating, the minimum one is sufficient for the state input, keeping the input dimensions low for the arbitration agent.

$$\pi_{ij}^{ce} = \left\{ \min(\log \pi_i(a_j|s_i) \big| i, j = 1, ..., n, i \neq j) \right\}. \tag{4.1}$$

More suitable characteristics like the standard deviation of the log probabilities or a real certainty measure for the current action are not available outside of training for actor-critic policies.

In certain scenarios it is important that the actions do not change too abruptly if the module selection changes, for example in a robotic system. This can be ensured by adding the last executed action as state input to the policy network. By setting this action to the actual executed action by the arbitration agent, a stable transition between modules can be achieved.

Value network
The value network estimates represent the expected future rewards by each agent, rating the current state. A higher value should favor the selection of this agent because a high cumulative reward is expected by following its policy. Here, the progression of each agent's value estimate is important, with a change in the value function representing a different assessment of the current state.

Similar to the cross-evaluation of the proposed actions, the value network of each agent can be used with the proposed actions of the other agents. This allows us to get the estimated value of the proposed actions by other agents and thereby the difference to the initial value

estimate; see equation (4.2).

$$V_{ij}^{ce} = \left\{ V_i(s_t, a_j) - V_i(s_t, a_i) \middle| i, j = 1, ..., n, i \neq j \right\}. \tag{4.2}$$

Temporal context

Using the last N estimates as state inputs allows the arbitration agent to also consider temporal changes, resulting in the complete state input expressed in equation (4.3).

$$S_t = \{V(t-k), \pi_{ij}^{ce}(t-k), V_{ij}^{ce}(t-k) | k = 0, ..., N-1\}. \tag{4.3}$$

The value and policy network outputs can be combined in various ways, which is analyzed in-depth in Section 5. Using the value estimates, the cross-evaluated value estimates and the cross-evaluated minimum log probabilties as state inputs achieved the best performance in the conducted experiments. This results in a state space dimension according to equation (4.4), with n as number of modules.

$$dim(St) = N \cdot 6n. \tag{4.4}$$

An overview of the arbitration framework using two modules and the resulting state inputs is given in Figure 2. Only the modules receive observations from the environment, while the arbitration agent uses the outputs of their internal networks. This limited state space of the arbitration agent allows for a fast training and therefore easier integration of new modules.

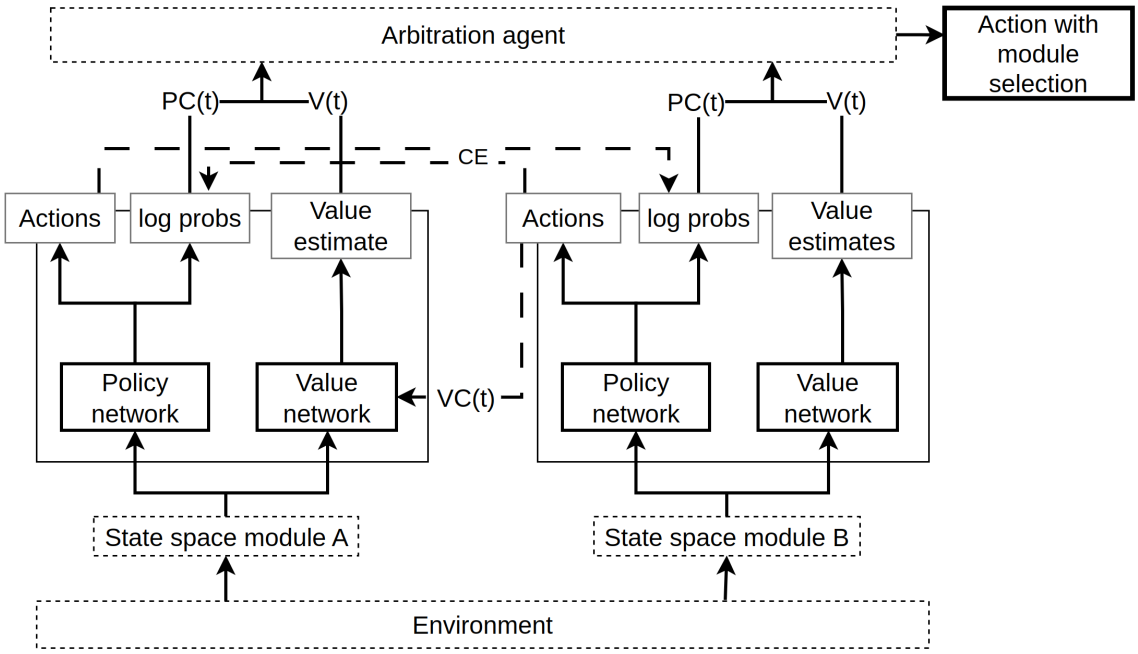


Figure 2. Concept of the proposed arbitration agent's architecture

Action Space

The arbitration agent's action space is discrete and consists of n actions corresponding to the number of modules. At each time step t, one module is selected and its actions are executed, without any modifications.

Reward Function

The reward function for the arbitration agent is crucial for training, as it shapes how the agent learns to coordinate between each module. While a sparse reward function would be the most straightforward approach and closes to an actual modular framework, it would result in long training times. Additionally, the learned strategy tends to always select the same module, making it not suitable for solving complex robotic problems [61].

A successful modular system requires a dense reward function, which defines the final objective and intermediate rewards to encourage the selection of different modules [9]. In some cases, the same reward functions as for the sub-agents can be used for the arbitration agent. However, this also risks the arbitration agent getting stuck in a non-optimal policy by only selecting the same agent. A more feasible approach is to write specific reward functions for the complex task. This also allows for the covering of specific scenarios like constraining excessive module switching or preventing the selection of certain modules under specific conditions.

Training Strategy

The required modules are integrated as function blocks to train the arbitration agent. Each module receives the required state inputs and produces the needed outputs for the arbitration agent. The modules themselves are no longer trained. Using the selected RL algorithm, state inputs, and reward functions, the arbitration agent learns under which conditions to select the action of which module. This is the same approach as with any other RL agent.

4.3.2 Design Considerations and Limitations

Certain considerations have to be taken into account when integrating the arbitration agent. These include:

- Normalization: Different modules will have different scales for their value and policy network outputs. It is always good practice to normalize the inputs to the arbitration agent to ensure a stable learning process. For the log odds, a normalization between 0 and 1 can easily be achieved by transforming them to a probability using the sigmoid function. The value estimates can be normalized by sampling the minimum

and maximum values before starting the learning process.

- Runtime execution: The update frequency of each module might be different. Those cases have to be handled individually, either by using the slowest sampling frequency or by re-using the previous observations for the slower modules.

While the functionality and deployability of our proposed arbitration framework is verified in the following sections in a simulated as well as a real-world environment, certain limitations remain and should be considered before applying this approach.
Primarily, the usage of actor-critic algorithms for the modules results in certain drawbacks. The policy network outputs do not provide a good measure for the certainty of the current action, which could be a valuable input for the arbitration agent. While using the already available value and policy networks allows a limited input dimension and utilization of the modules knowledge, under normal circumstances the value network would not be required during interference, resulting in a higher computational overhead. Other approaches utilizing state-value methods, or a module selection based on environment observations, might be more suitable.
Secondly, each module evaluates it's state space at every timestep, creating an additional computational overhead linear to the number of modules. While a single agent would have the same computational requirements, approaches where not all modules are active simultaneously, for example by using termination signals as done in hierarchical designs, could be more efficient.

5. Agent Design and Training

This chapter will implement the modular arbitration architecture as described in Chapter 4 for a compositional robotic task. Two specialized modules are designed and trained individually for their specific task in a simulation environment. The arbitration agent will solve the initial robotic task using the trained modules.

The fundamental task objective is to locate and approach a specific object, in this case, a red cube, in an unknown environment. This task is decomposed into two sub-tasks: an exploration module for exploring the environment and an object detection module for identifying and approaching the red cube. Each module is trained in the *Isaac Sim* simulation environment and evaluated based on key metrics like task success rates. Afterwards, the modules are combined using the modular arbitration approach, and the arbitration agent is trained. Finally, the performance of the individual modules is compared with the combined approach.

5.1 Task Decomposition Overview

5.1.1 System Requirements and Constraints

The robot's fundamental objective is to find and approach a specific object, in this case, a red cube, in an unknown environment. This objective can be decomposed into two sub-tasks with different sensor requirements and objectives.

First, the robot must systematically explore an unknown environment to locate the target cube while avoiding obstacles. Because the robot has no prior knowledge about the environment, systematic exploration is necessary while building a map representation. Solving this task requires a lidar and a depth camera for the state inputs.

Second, the robot must reliably detect the cube from different distances, angles, and lighting conditions. Once identified, it needs to drive towards it, maintaining visual contact and ideally handling a temporary loss of vision. This requires a specialized object detection module that uses RGB camera data to identify the cube and generate motion commands to drive toward it.

This task decomposition allows us to train and optimize each module individually while

ensuring an effective combination later using the modular arbitration architecture.

5.1.2 Shared Implementation Aspects

All agents share the following implementation aspects to ensure compatibility with the arbitration architecture.

Each agent is trained within the *Isaac Sim* simulation environment with *Isaac Lab* as training framework and *skrl* for the RL algorithm implementation. A simplified 4-wheeled model was generated for the robot, representing the Hiwonder JetAuto. The original JetAuto robot uses mecanum wheels, which caused issues when running the simulation using the GPU and its respective solver. All four wheels of the model are steerable, each using the same steering angle. This allows us to recreate most of the characteristics of mecanum wheels but not sideways driving.

Table 2 lists the used simulation parameters. The time interval between each simulation step is defined by *dt*. Decimation defines the ratio of simulation steps to control steps, meaning after how many simulation steps the RL algorithm receives a new observation. The rendering interval defines after how many simulation steps the visualizations are updated.

Table 2. Isaac Sim simulation parameter

Parameter	**Value**
dt	$1/60s$
decimation	4
rendering interval	1
device	CUDA

The training is done using PPO for all agents due to its proven robustness and successful application for robotic applications. The default algorithm configuration parameters and their connections to the fundamental equations are listed in Table 3. Changed parameters are specified in the respective sections. Each agent uses separate policy and value networks. State inputs for modules whose value network is used for action cross-evaluation are limited to environmental observations which are also accessible during interference.

Table 3. Default configuration parameters PPO by skrl

Parameter	Value	Explanation
rollouts	16	Number of trajectories gathered before update
learning epochs	8	Number of passes over collected trajectories for updating the policy and value networks
mini batches	2	Trajectories are divided into mini-batches for during learning, more memory efficient and possible more stable training
discount factor	0.99	Equals gamma γ
lambda	0.95	GAE parameter for advantage estimate, see equation (3.31)
learning rate	0.001	Step size for parameter updates θ, see equation (3.23)
grad norm clip	0.5	Limits size of gradient during policy updates
ratio clip	0.2	ϵ parameter for clipped surrogate objective, see equation (3.29)
value clip	0.2	Limits maximum change of estimated value
entropy loss scale	0.0	Entropy for increased exploration

Different domain randomizations, such as randomized lightning and random noise for the sensor inputs, are applied to the simulation to improve its transfer to the real-world robot.

Both modules use the same continuous action space with two values. The first represents the linear velocity, and the second the angular position of the four wheels. This allows the arbitration agent to use cross-evaluation for each action as additional input.

5.2 Object Detection and Approach Module

5.2.1 Design Requirements

The object detection and approach module must be able to identify the specified object in the environment and drive towards it. The main focus is a robust detection algorithm under varying distances, orientations, and lighting conditions. This was achieved by training a separate object detection network, see Section 5.2.2. A secondary goal is a smooth driving behavior. The technical implementation of the module is presented in Section 5.2.3 and the training and evaluation in Section 5.2.4.

Based on the task for this module and the supplied sensor data, the agent's expected learned strategy is to rotate into one direction at the starting position until the cube is detected and then drive towards it.

5.2.2 Fine-tuned Object Detection Network

Training a reliable end-to-end object detection network using RL was not feasible due to the need for a large amount of training data. This is connected to the known issues of sample inefficiency and sparse rewards. The feedback received for the agent is too sparse to supply meaningful updates for the object detection network. While training the object detection network together with the agent was successful if a bigger cube was used and no other objects or changed lightning conditions were present, a reliable detection under realistic conditions required a separate trained network.

The object detector is trained using supervised learning with labeled data generated within the simulation environment. This fine-tuned network classifies whether the cube is in the current image and supplies its position, which allows the RL agent to focus on learning the control policy instead of having to learn basic image recognition, which is especially troublesome with changing cube orientations, distances and changed lighting conditions.

For achieving an efficient and fast object detection network, the pre-trained object detection network *Faster R-CNN* [73] was used as a base model. This model excels at handling objects at different scales and positions due to using Region of Interest (RoI) pooling and region proposal networks. The first layer is a CNN for feature extraction, outputting feature maps. Those feature maps are analyzed by the region proposal network, which outputs a set of regions of interest, and the RoI pooling, which extracts the feature vectors from those regions. The object detection network then uses the extracted features to classify and regress the bounding boxes. For the backbone, the mobile phones optimized *MobileNetV3-Large FPN* was used, which allows for a faster computation compared to bigger models like *ResNet-50-FPN* [74].

The network was fine-tuned to accurately classify and detect the red cube by replacing the final classification layer with a new layer with the two output classes *no-cube* and *cube*. The fine-tuning was achieved using $1,000$ images, with and without a cube, generated within the simulation environment under varying lighting conditions and different distances and orientations. The images and cube positions were labeled using *Label Studio* [75]. A random color jitter was applied to each image during training to improve the stability (brightness $= 0.6$, contrast $= 0.4$, saturation $= 0.4$, hue $= 0.2$). See Table 4 for the used training parameter.

Table 4. Training parameter fine-tuned Faster R-CNN

Parameter	**Value**
Optimizer	type: SGD momentum: 0.9 weight decay: 0.0005
Learning rate	initial: 0.005 step scheduler: num_epochs//2
Epochs	25

The trained network has 5 neurons as output layer, one for the classification task with the confidence score and the other four for the normalized bounding box, specifying the x and y coordinates and width and height of the detection area.

5.2.3 Technical Implementation

The state inputs for the agent are the last six detection results of the object detection network and the last four executed actions. The last six detection results are used to ensure a more robust cube detection and a memory if specific actions or failed identifications resulted in losing the cube. Supplying the last four actions ensures that the actions by the agent are more steady, resulting in smoother driving behavior. Also, it ensures that the robot knows in which direction it is rotating when searching for the cube. Detection and action history are initially individually processed, both in their own two-layer feed-forward network. Their outputs are combined into an additional two-layer feed-forward network, producing the policy and value network outputs. Figure 3 displays the full network architecture which is used as separate networks for the policy and value functions.

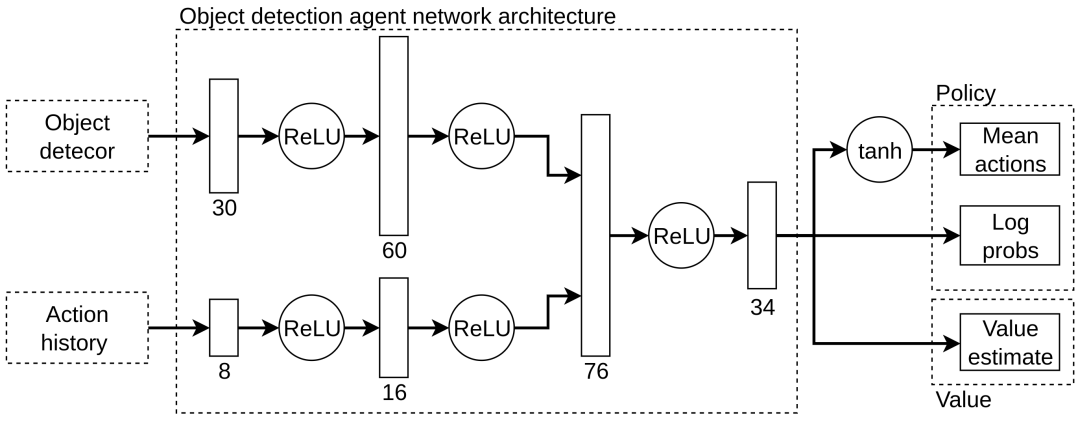


Figure 3. Neural network architecture object detection agent

The agent receives multiple rewards and penalties for guiding its behavior. The main reward is for reaching the cube, which is achieved when the distance to the cube is below $0.2\,\mathrm{m}$, equation (5.1). Additionally, it receives a dense reward or penalty based on the change of distance to the cube, equation (5.2). If the sum of the generated actions change by more than a specific threshold from the previous ones, a penalty is given proportional to this change, see equation (5.3). This ensures a smoother driving behavior. An additional reward or penalty is given for the driving direction to encourage the robot to rotate while driving forward when searching for the cube, equation (5.4). The total reward is the sum of all rewards and penalties, see equation (5.5).

$$R_{goal} = (d_{cube}(t) < 0.2) \cdot 3.0, \tag{5.1}$$

$$R_{dist} = (d_{cube}(t-1) - d_{cube}(t)) \cdot 2.0, \tag{5.2}$$

$$R_{action} = (|a(t) - a(t-1)| > 0.25) \cdot -0.015, \tag{5.3}$$

$$R_{direction} = a_{linear}(t) \cdot 0.04, \tag{5.4}$$

$$R_{total} = R_{goal} + R_{dist} + R_{action} + R_{direction}. \tag{5.5}$$

For the PPO algorithm the default parameters (see Table 5) were adjusted according to Table 5.

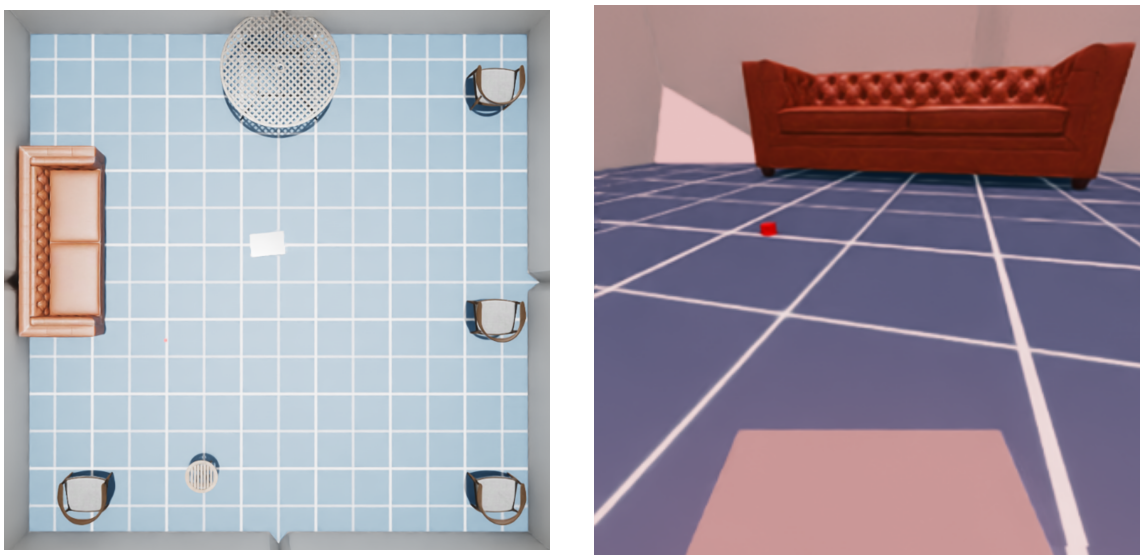
Table 5. Object detection module PPO training parameters

Parameter	**Value**
mini batches	8
rollouts	32
learning epochs	4
discount factor	0.95

5.2.4 Training and Evaluation

The object detection agent is trained using 32 parallel environments, with the robot spawning in the middle of the room with a random orientation and the goal cube with a randomized distance and orientation around it. Based on the agent's success rate, the cube's distance towards the robot is steadily increased, which is a usage of curriculum learning. Additionally, the lighting conditions are randomized to increase the robustness of the detection. The simulation environment is displayed in Figure 4, resembling a room with furniture at the walls. See Figure 4a for a top-down view and Figure 4b for the first-person view using the robot's camera. All objects are placed out of reach for the robot as they

only serve to increase the complexity for the object detection, not as obstacles.

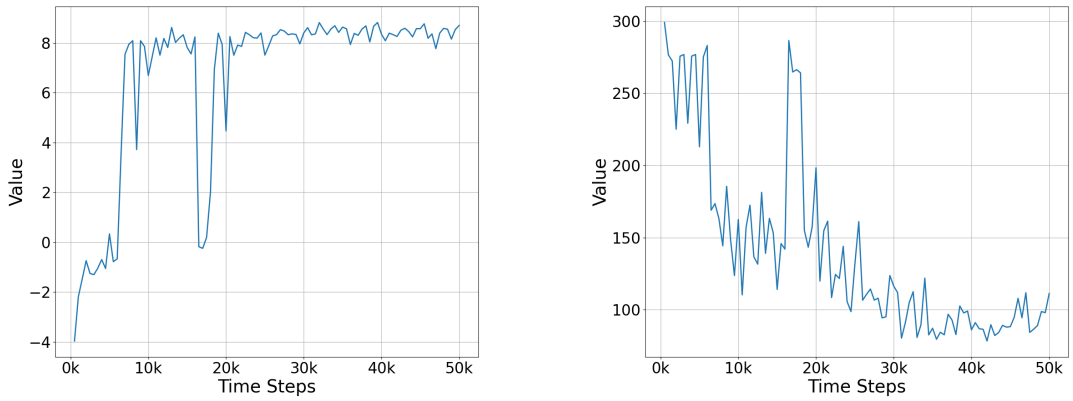


(a) Single simulation environment

(b) Robot's point of view

Figure 4. Object detection agent simulation environment

The tracked training process using *TensorBoard* is displayed in Figure 5. It can be seen that the agent successfully learned the task over time and achieved a stable performance after $\approx 30\ k$ time steps. This is indicated by the mean total reward per episode staying steady and the mean time steps per episode decreasing.



(a) Mean total reward per episode

(b) Mean time steps per episode

Figure 5. Object detection agent training progress TensorBoard

Evaluation
While initial training runs had issues with the agent just driving at the same spot in a circle, the agent successfully learned to find the cube after adjusting the rewards. The learned strategy corresponds to the expected behavior defined in the design requirements.

Table 6 contains the evaluation metrics of the trained agent. It has to be mentioned that the success rate is not the most informative value for this task alone, as the only way to fail is by reaching the time limit of $20\,\mathrm{s}$, which barely happens once the task is learned. Reducing the time limit more would just result in the random position of the cube deciding if the episode is a success or not. Nonetheless, the agent successfully learned the task and can now be used for the arbitration agent and transferred to the real robot. Because of the decimation of 4, only every 4 time steps a new observation is evaluated by the agent.

Table 6. Test results object detection module

Metric	**Value**
Total episodes	$1,000$
Total time steps	$381,456$
Total observations	$95,364$
Average observations per episode	95.36
Average time per episode	$6.36\,\mathrm{s}$
Average initial cube distance	$1.62\,\mathrm{m}$
Task success rate	$98.60\,\%$
Action deviation above 0.25	$6.81\,\%$
Action deviation above 0.5	$2.56\,\%$
Action deviation above 0.75	$2.24\,\%$

5.3 Environment Exploration Module

5.3.1 Design Requirements

The environment exploration module needs to learn to explore an unknown environment while avoiding obstacles, which requires a certain level of understanding about one's position and surroundings. For this, a lidar sensor and depth camera is used. The depth camera is used for obstacle avoidance, while the lidar data is used to generate an occupancy grid map to supply exploration awareness to the robot. The resulting action output should result in smooth and efficient motion planning.

Based on the task and available sensor data, the basic strategy the agent and, therefore,

the neural network should learn is as follows: Using the depth image, the steering output action should be adjusted so that the robot drives towards an area without obstacles. If the area ahead is a wall, the velocity action should be adjusted to stop the robot or drive backward. The lidar data will be turned into a map representation, allowing the agent to recognize unknown areas and adjust the steering output accordingly to drive toward them.

5.3.2 Technical Implementation

As state inputs for the exploration module, the lidar and depth camera sensor data is processed.

The lidar measurements are used to generate a robotic centered Occupancy Grid Map (OGM) of size 256×256. The resolution of the map is dependent on the selected room size. Using a square room with a length of $8\,\mathrm{m}$ results in a length of each grid cell of $l_{cell} = l_{room} * 2/N_{ogm} = 0.0625\ m$. The room size is multiplied by two because of the robotic-centered map; this ensures no loss of information when the robot approaches the walls of the room. Using a robotic-centered perspective ensures a consistent frame of reference and simplifies learning. The OGM cells are transformed into probabilities to normalize the values into the range $[0, 1]$ and used as state input for a CNN. The CNN should learn to extract features like rooms and open areas. The detailed architecture of the OGM CNN is shown in Figure 6, its integration in the overall network architecture in Figure 11.

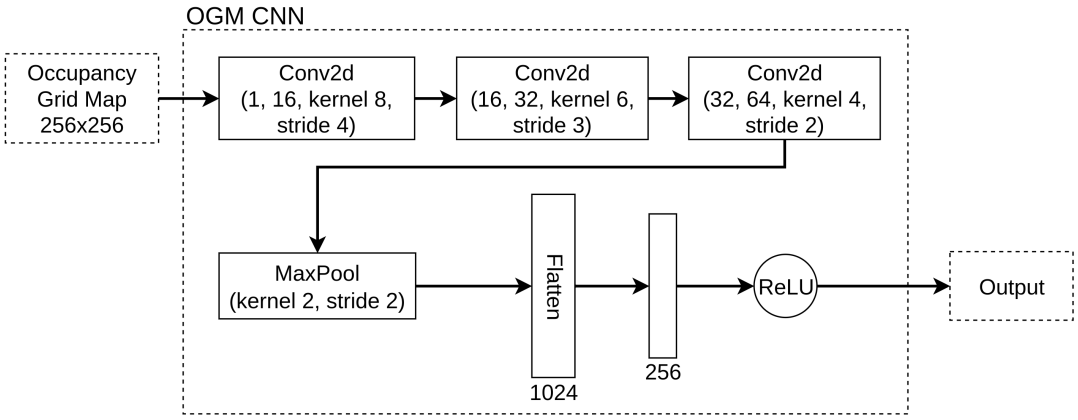


Figure 6. Occupancy grid map processing network architecture

A depth image from the RGB-D camera is used for obstacle avoidance, with an initial resolution of 320×240, and is transformed in multiple steps as depicted in Figure 10. For improving the robustness of the feature detection, noise and random black patches is added to the the initial depth image, Figure 10a. The black patches resemble capture

errors and reflections as found in the real world. For every fifth image, the count (2 to 6) size ($[5$ to $20] \times [30$ to $100]$ px), and location of patches are randomized, Figure 10b. The Gaussian noise is applied to the full image with an increasing standard deviation based on the distance measured, Figure 10c.

The image values are then clipped to a maximum depth of $3\,\mathrm{m}$ to improve the close-range perception and normalized to the range $[0, 1]$. Now, a horizontal slice of 320×16 is taken to greatly reduce the input size while maintaining the relevant obstacle information and used as input for the depth CNN. Figure 10d shows 15 sequential image slices, resembling the sequence length used by the following RNN. The network architecture of the CNN is shown in Figure 7. For the depth CNN residual blocks are used, first introduced with *ResNet*, which skip connections between layers to improve the gradient flow. This helps reduce the vanishing gradient problem, allows easier reuse of features, and allows the design of deeper networks [76]. The depth CNN output is then flattened into a $1D$ vector, processed by a fully connected layer, and analyzed by an Long Short-Tem Memory (LSTM) for temporal dependencies. Using an LSTM helped the agent not get stuck when oriented into a corner or taking a too-steep curve and colliding with the obstacle sideways. This network is integrated into the overall network architecture shown in Figure 8.

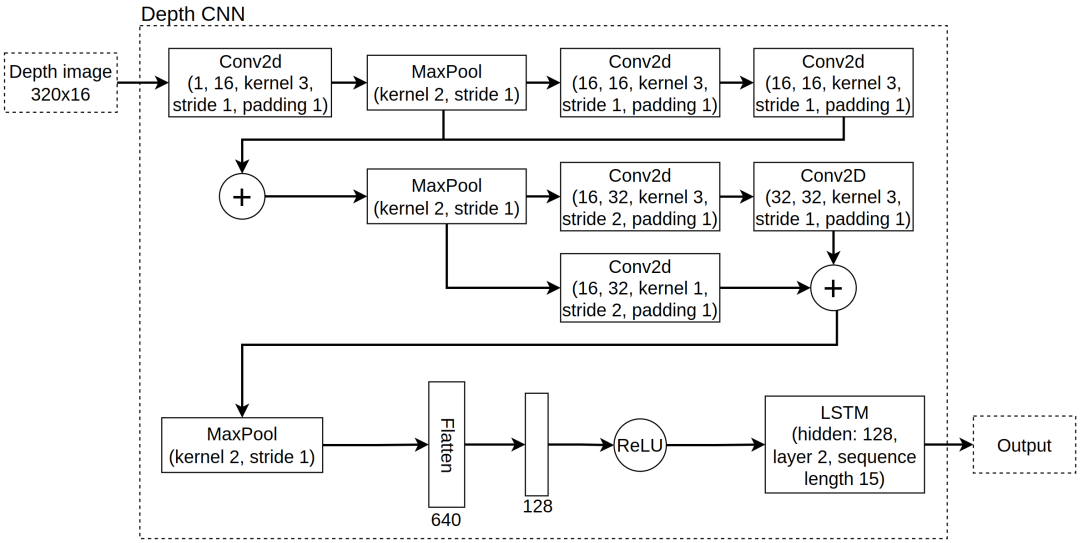


Figure 7. Depth image processing network architecture

As an additional component of the multi-stream network architecture, the last 8 actions and 5 robot orientations, normalized to $[0, 1]$, are used as inputs into a two-layer feed-forward network. The output of all three networks is concatenated and processed by a three-layer feed-forward network. The resulting output is used to select the robot's actions. The usage of multi-stream networks allows for efficient feature detection and processing of different sensor inputs. Figure 8 shows the final network architecture used by the exploration agent, implemented separately by the policy and value function.

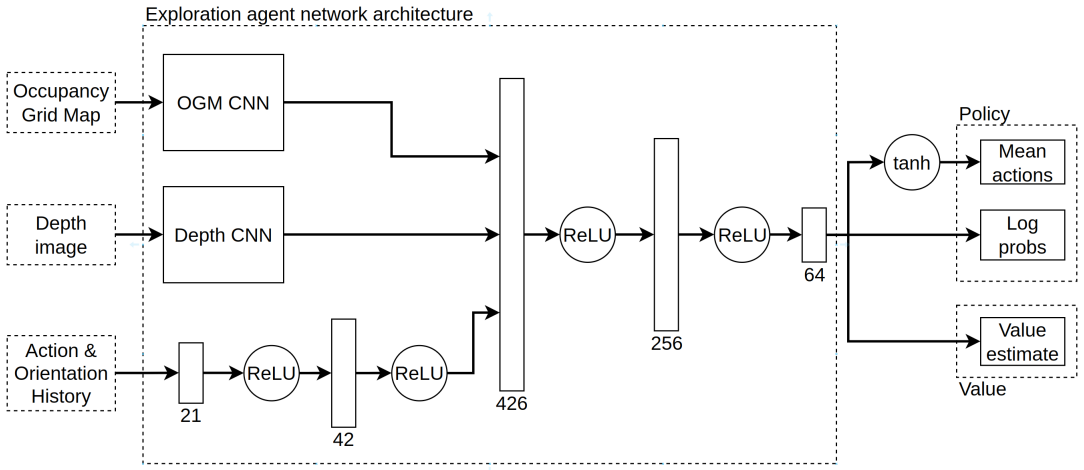


Figure 8. Exploration agent network architecture

For training the agent, multiple rewards and penalties are used to guide its behavior. The main reward is the exploration reward, which is calculated using the discovery percentage of the OGM map, equation (5.6), and the main penalty is if the robot collides with an obstacle, resulting in the episode being aborted, equation (5.7). To encourage smooth driving behavior, the agent receives a penalty if the current actions differ by more than a specific threshold from the previous ones, equation (5.8). For preventing strategies like just driving forward and backward or in circles, which prevents collisions and therefore a high penalty, one additional reward and one penalty are used. A reward is given for driving forward, equation (5.9), which also ensures that the robot correctly associates the depth images with obstacle avoidance. A penalty is given if the location over the last 50 time steps did not change enough, penalizing the driving in circles, see equation (5.10). The total reward is calculated using equation (5.11).

$$R_{explo} = [E(t) - E(t-1)] \cdot 0.25, \tag{5.6}$$

$$R_{coll} = -1.25, \tag{5.7}$$

$$R_{action} = (|a(t) - a(t-1)| > 0.25) \cdot -0.015, \tag{5.8}$$

$$R_{forward} = a_{velo}(t) \cdot 0.04, \tag{5.9}$$

$$R_{loc} = (p(t-50) - p(t) > 0.1) \cdot -0.1, \tag{5.10}$$

$$R_{total} = R_{explo} + R_{coll} + R_{action} + R_{forward}. \tag{5.11}$$

For the PPO algorithm the default parameters (see Table 5), were adjusted according to Table 7. The used learning rate scheduler *KLAdaptiveRL* adjusts the learning rate dynamically based on how much the policy is changing. For this it measures the Kullback-

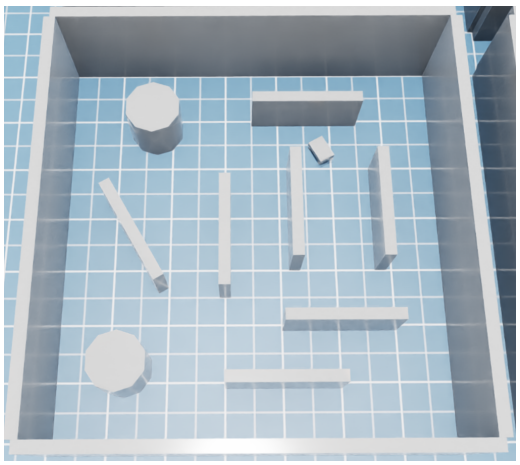
Leibler (KL) divergence between the last and current policy distribution. If the policy change is greater then the *kl threshold*, the learning rate is reduced by *lr factor*, resulting in more conservative updates.

Table 7. Exploration module PPO training parameters

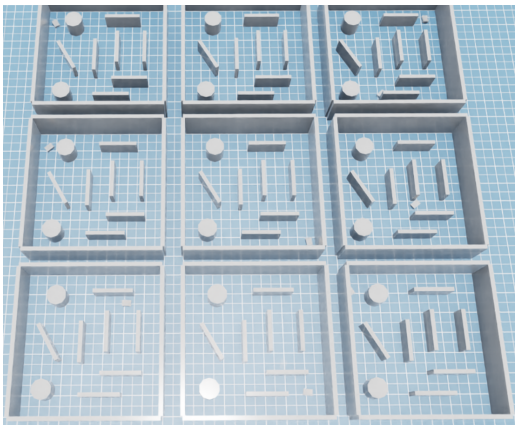
Parameter	**Value**
mini batches	32
rollouts	480
learning epochs	4
learning rate	$1e-4$
learning rate scheduler	*KLAdaptiveRL* kl threshold 0.008 lr factor 1.2

5.3.3 Training and Evaluation

The exploration agent is trained using 16 parallel environments. This limit is caused by Isaac Sim's replicator crashing if more than that are used. The simulation environment is a room containing multiple obstacles with different shapes and orientations. Figure 9a shows the simulation environment of a single agent, Figure 9b the arrangement of multiple ones. At the beginning of each episode, the robot spawns at a random location. For this simulation, no specific lightning or material is used.



(a) Single environment

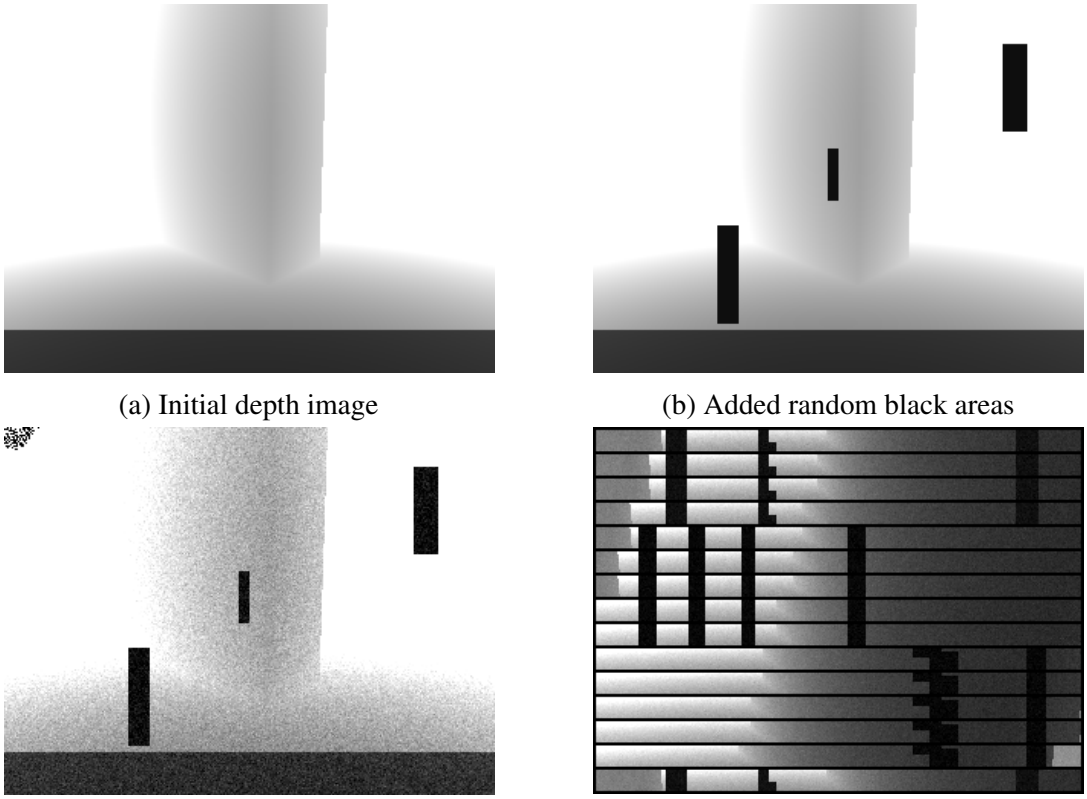


(b) Multiple environments for parallel computing

Figure 9. Exploration agent simulation environment

A sample for a depth image obtained during simulation and its transformation to the final slices used for the depth CNN input is shown in Figure 10. Figure 11 shows three OGM

maps over time, which are used as input for the OGM CNN. The maps show the exploration progress of the agent, with white representing explored cells, black occupied and gray unknown.



(a) Initial depth image

(b) Added random black areas

(c) Added random noise

(d) Series of 15 slices

Figure 10. Transformation process of initial depth image to slices for network input

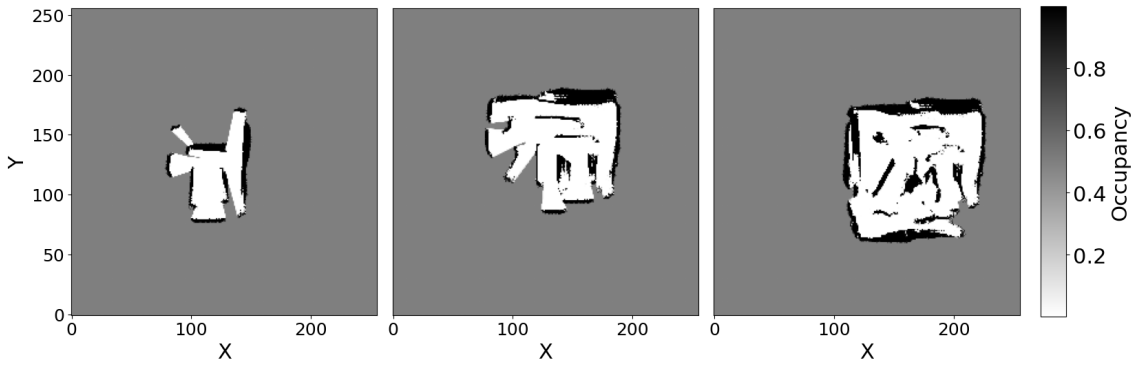
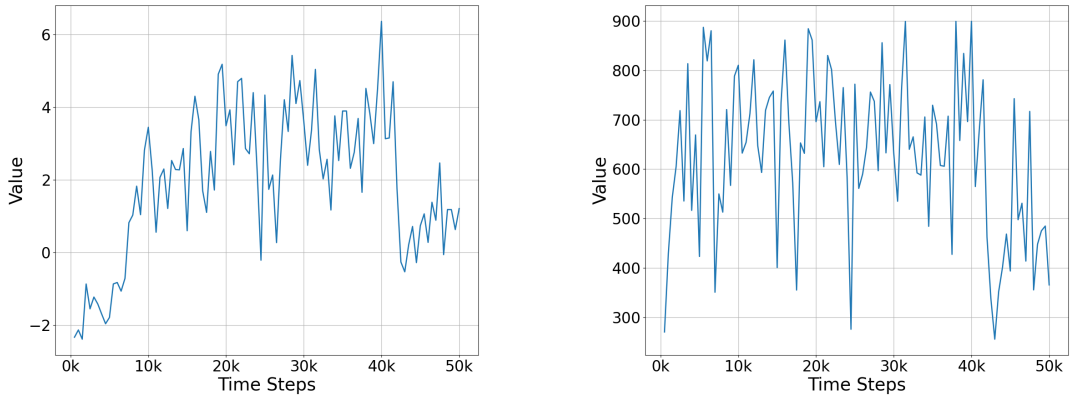


Figure 11. Exploration agent occupancy grid map over time

Figure 12 shows the tracked training progress using *TensorBoard* for $\approx 105\ k$ time steps. The mean total reward per episode is shown in Figure 12a, and the mean time steps per episode in Figure 12b. Based on the mean total reward, the best results were obtained with the learned progress at $40\,\mathrm{k}$ time steps, meaning this agent will be used for the following

evaluations. The mean reward does not stay steady at the maximum obtainable reward and the mean time steps per episode also show a lot of fluctuation. This indicates that the agent was maybe not able to learn an optimal policy and further adjustments to the state inputs, network architecture or learning parameters might give a better, more stable result.



(a) Mean total reward per episode

(b) Mean time steps per episode

Figure 12. Exploration agent training progress TensorBoard

Evaluation

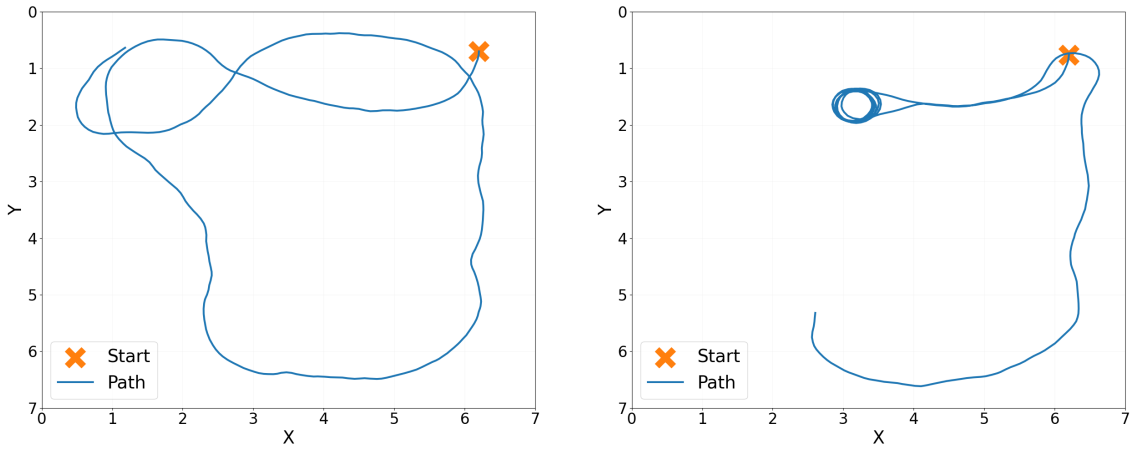
The agent learned successfully to drive around the map while avoiding obstacles. Table 8 shows the evaluation metrics of the exploration agent. The agent was able to explore on average $56\,\%$ of the map, with a maximum of $86.75\,\%$ and managed to finish $60.73\,\%$ of the episodes without colliding. While those results could be better, especially the percentage of episodes finished without colliding, it confirms the basic functionality of the agent.

Table 8. Test results exploration module

Metric	**Value**
Total episodes	$1,159$
Total time steps	$597,936$
Average time steps per episode	$520.9/600$
Average time per episode	$34.73\,\mathrm{s}$
Average discovery percentage	$56.9\,\%$
Average discovery percentage w/o colliding	$62.5\,\%$
Max discovery percentage	$84.7\,\%$
Count collisions	355
Task success rate	$69.37\,\%$
Average time step of collision	346.72

While the quantitative evaluation of the exploration agent gives first positive results, the average discovery percentage of only $56.9\,\%$ leaves room for improvement. One issue seems to be that the agent still collides too many times with obstacles, reflected by the task success rate of $69.37\,\%$. But even when just measuring episodes without collisions is the average discovery percentage only $62.5\,\%$, a difference of $\approx 22\,\%$ to the maximum achieved discovery percentage.

While the low success rate is caused by collisions with walls when the robot tries to turn around them, touching the wall with its inner side, the low discovery percentage could be due to the lidar data not being used effectively for guiding the robot towards undiscovered areas. Figure 13 shows exemplary the path traveled during two different episodes. The path in Figure 13a is a well chosen path with a high discovery rate, while in Figure 13b the robot got stuck driving in circles for quite a while. It remains questionable if the lidar data and, thereby, the OGM positively influenced the exploration success.



(a) Sample path robot A

(b) Sample path robot B

Figure 13. Exploration agent traveled path samples

5.4 Arbitration Agent Integration

5.4.1 Technical Implementation

The arbitration agent uses the internal neural network representations from the above-mentioned modules as state inputs. This can include the value estimates, policy network log probabilities and cross-evaluations of the other module's actions using the value or policy network. For the following training and evaluation, the last 3 values are used as state inputs for a temporal context, resulting in a state input dimension according to equation 4.4. Additionally, the last 5 agent selections are included in the state inputs. For evaluating the usefulness of the suggested state inputs, especially the log probabilities and

cross-evaluations, the following combinations of possible state inputs are tested:

- value estimates with value cross-evaluation: S_{vce}, see equation (5.12)
  with a dimension of $dim = 3 \times 4 = 8$
- value estimates with value cross-evaluation and minimum of cross-evaluated actions using log probs: S_{vmp}, see equation (5.13)
  with a dimension of $dim = 3 \times 6 = 18$
- value estimates with value cross-evaluation, cross-evaluated actions using log probs and ratio of policies: S_{vpr}, see equation (5.14)
  with a dimension of $dim = 3 \times 12 = 36$

$$S_{vce} = \left\{ V_0, V_1, V_{0,1}^{ce}, V_{1,0}^{ce} \right\}, \tag{5.12}$$

$$S_{vmp} = \left\{ V_0, V_1, V_{0,1}^{ce}, V_{1,0}^{ce}, \min(\pi_{0,1}^{ce}), \min(\pi_{1,0}^{ce}) \right\}, \tag{5.13}$$

$$S_{vpr} = \left\{ V_0, V_1, V_{0,1}^{ce}, V_{1,0}^{ce}, \pi_{0,1}^{ce}, \pi_{1,0}^{ce}, \frac{\pi_0}{\pi_1} \right\}. \tag{5.14}$$

Figure 14 shows the neural network architecture using a three-layer feed-forward network. The arbitration agent has a discrete action space with two possible actions representing the module selection. For this a categorical policy is used, which means the actions are sampled from a categorical, not Gaussian distribution.

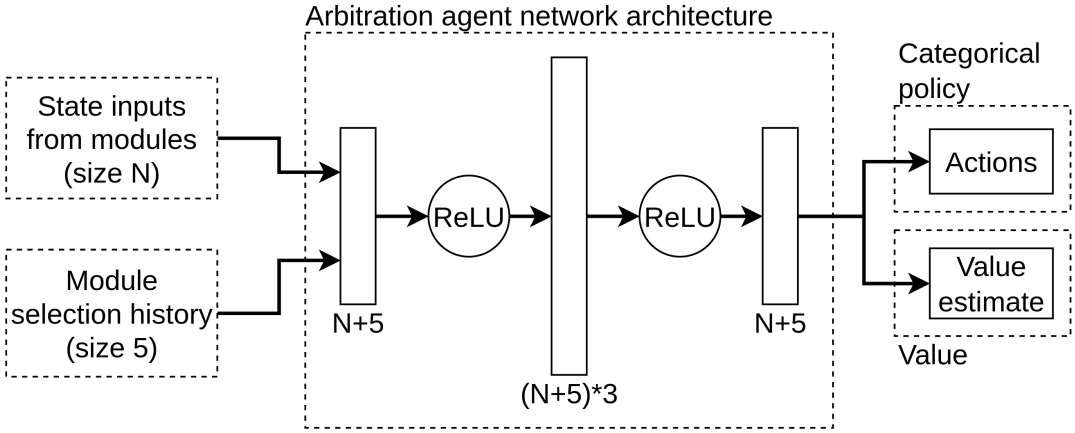


Figure 14. Neural network architecture arbitration agent

For the reward function, some of the reward functions of the individual modules are re-used for creating a dense reward function, which ensures that each module will be selected. This includes the exploration agent's exploration reward and collision penalty, equations (5.6) and (5.7). Additionally, the object detection agent's goal reached and distance change rewards are used, equations (5.1) and 5.2, with the restriction that the distance change

reward is only given if the cube is visible. To prevent excessive module swapping, a penalty is given for every changed module selection, equation (5.15). The total dense reward is calculated as shown in equation (5.16).

$$R_{switch} = (a_t \neq a_{t-1}) \cdot -0.01, \tag{5.15}$$

$$R_{dense} = R_{explo} + R_{coll} + R_{goal} + (R_{dist} \text{ if cube visible}) + R_{switch}. \tag{5.16}$$

The PPO algorithm is used with its default parameters, see Table 5.

5.4.2 Training Strategy

The arbitration agent is trained using 16 parallel environments, with the robot and cube spawning at random room locations. The pre-trained exploration and cube detection modules are used as state inputs with frozen weights. At each time step, both modules process the environment using their specific sensors and generate the state inputs needed for the arbitration agent. Based on those values, one of the modules is selected, the action is executed, and the simulation advances to the next simulation step.

In the following, the state inputs for the arbitration agent are analyzed in detail using one episode as sample. Figure 15 shows the distance to the cube and the classifier value for the cube detector. Those values are not supplied to the agent, but help to understand what is happening during this episode. In the beginning, no cube can be seen by the robot's camera up to the time step ≈ 420, indicated by the classifier value. Figure **??** shows the value estimates of the exploration and object detection agent. In can clearly be seen that, while no cube can be seen by the robot, the object detection agent's value estimate keeps the same value. Once a cube is detected this value rises. Using the policy networks log probability function for cross-evaluation allows to compare how similar the actions of each module are, see Figure 17. Here a higher value indicates that the action is safe to execute.

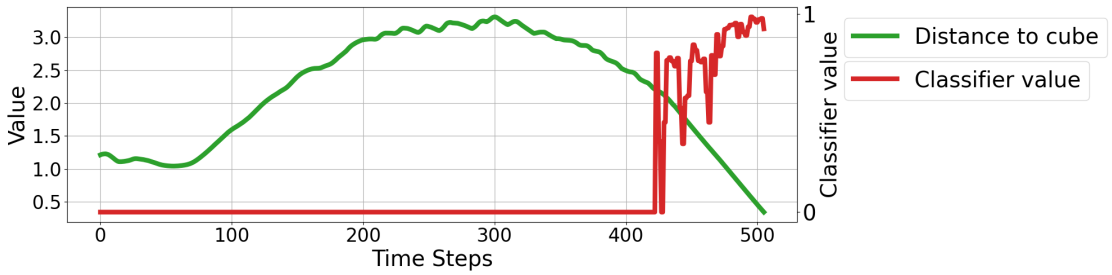


Figure 15. Arbitration agent sample episode cube distance and cube classifier

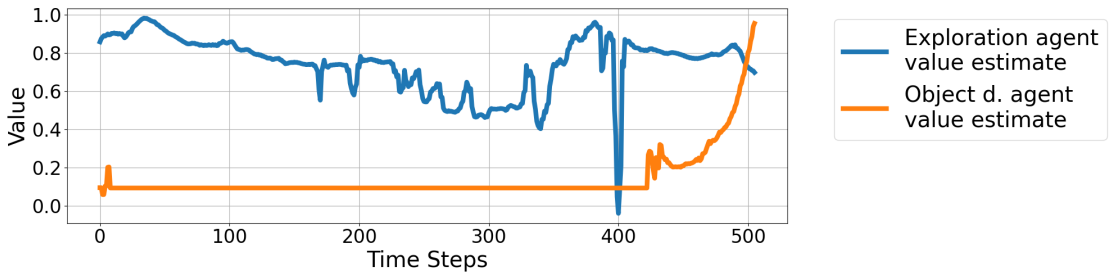


Figure 16. Arbitration agent sample episode value estimates

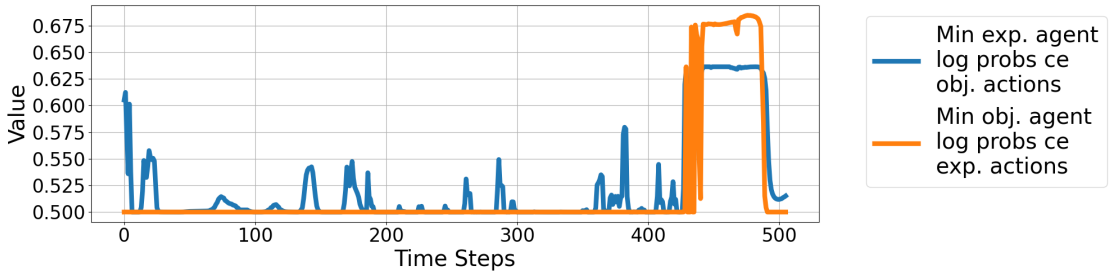


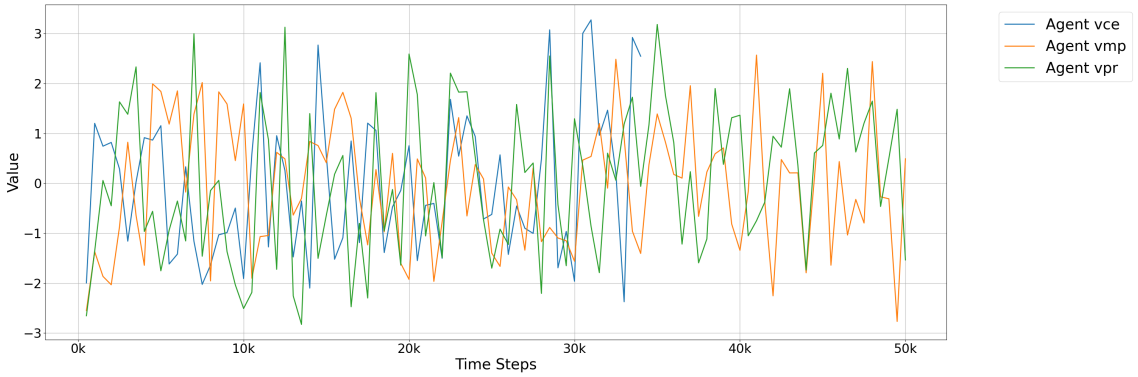
Figure 17. Arbitration agent sample episode cross-evaluation of actions using log probs

The tracked training process for all three state-input variations using *TensorBoard* is shown in Figure 18. The mean total reward per episode shows no real progress, indicating an issue with the learning.

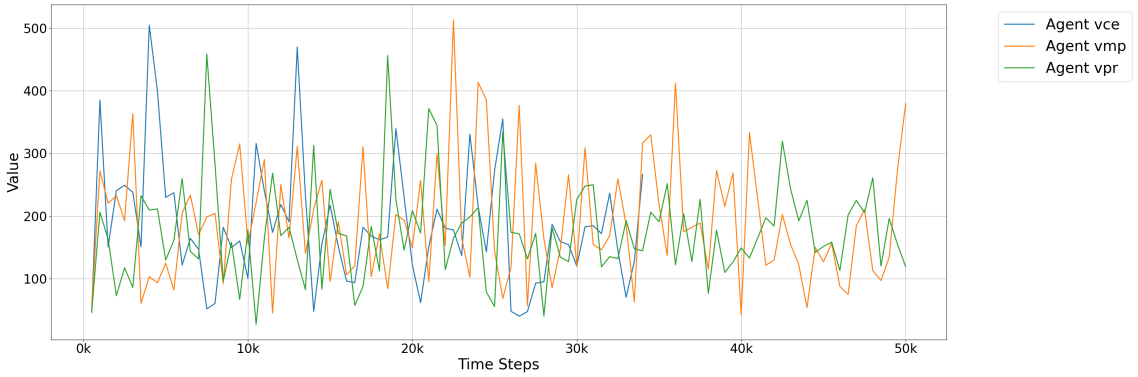
5.4.3 Performance Evaluation

For evaluating the performance of the arbitration agent and the different combinations of state inputs, each agent is tested in three different environments. Those are the object detection environment **cube**, the exploration agent environment **Exploration** and a mixed environment, containing obstacles and a cube, **multi**.

Inside the object detection environment, see also Figure 4a, the robot needs to find the cube as fast as possible. Obstacles are only placed near the walls and the walls itself. The test results are shown in Table 9. It can be seen that the agent *vce* and *vmp* have issues finding the cube. This can be attributed to the high usage of the exploration module, while the *vpr* agent has a very good success rate with a more balanced share of modules active. The challenge with this environment for the agents is that by only activating the exploration module in the beginning, the robot drives towards the room wall and then follows it, resulting in a time out before the cube becomes visible for the object detection module. While the *vpr* agent achieved an acceptable result with a task success rate of $78.80\,\%$ and an average of 155.3 time steps per episode, it is still notably below the performance of the



(a) Mean total reward per episode



(b) Mean time steps per episode

Figure 18. Arbitration agent training progress TensorBoard

object detection agent with its task success rate of $98.60\,\%$ and an average of 95.36 time steps per episode.

Table 9. Test results arbitration agent in cube only environment

Metric	**Agent vce**	**Agent vmp**	**Agent vpr**
Total episodes	376	450	769
Total time steps	117, 546	119, 107	119, 422
Average time steps per episode	312.6	264.6	155.3
Task success rate	$13.83\,\%$	$30.44\,\%$	$78.80\,\%$
Exploration module active	$95.36\,\%$	$92.1\,\%$	$47.5\,\%$
Object d. module active	$4.6\,\%$	$7.8\,\%$	$52.49\,\%$
Module changes	$6.36\,\%$	$2.3\,\%$	$35.8\,\%$

Running each agent in the exploration environment presents a different challenge. Here the optimal approach is to select the exploration agent all the time to achieve a maximum discovery percentage without colliding, see Table 10 for the test results. In this environment the agents *vce* and *vmp* were most successful, with task success rates of $70.12\,\%$ and $75.22\,\%$ respectively. The *vpr* agent seems to use the object detection module too much,

resulting in a worse performance of only $27.93\,\%$. Surprisingly, the task success rates and average discovery percentages of the first two agents are even better than the task success rate of the exploration agent, indicating that the occasional module change has a positive effect on exploration.

Table 10. Test results arbitration agent in exploration only environment

Metric	**Agent vce**	**Agent vmp**	**Agent vpr**
Total episodes	241	226	426
Total time steps	116, 379	115, 087	118, 423
Average time steps per episode	482.9	509.23	277, 9
Average discovery percentage	$59.25\,\%$	$58\,\%$	$36\,\%$
Average discovery percentage w/o colliding	$68.3\,\%$	$63.5\,\%$	$44.25\,\%$
Max discovery percentage	$89.3\,\%$	$89.1\,\%$	$67\,\%$
Task success rate	$70.12\,\%$	$75.22\,\%$	$27.93\,\%$
Exploration module active	$98.35\,\%$	$99.09\,\%$	$74.1\,\%$
Object d. module active	$1.6\,\%$	$0.9\,\%$	$25.9\,\%$
Module changes	$2.28\,\%$	$0.72\,\%$	$22.6\,\%$

The last environment for evaluation is the multi environment, representing the initial robotic task. Here, the agent has to explore its surrounding while avoiding obstacle, but can also find the red cube in random locations. The evaluation metrics are shown in Table 11. For this task the *vpr* agent was the most successful in finding the cube, while also achieving a satisfying average discovery percentage of $56.7\,\%$ and finishing $92.87\,\%$ of all episodes without colliding.

Table 11. Test results arbitration agent in mixed environment

Metric	**Agent vce**	**Agent vmp**	**Agent vpr**
Total episodes	431	1439	421
Total time steps	117, 718	118, 733	118877
Average time steps per episode	273.1	82.5	282.37
Average discovery percentage	$60.84\,\%$	$35.1\,\%$	$56.7\,\%$
Episodes finished without collision percentage	$58.24\,\%$	$30.30\,\%$	$92.87\,\%$
Cube found percentage	$28.54\,\%$	$30.37\,\%$	$67.93\,\%$
Exploration module active	$96.74\,\%$	$75.14\,\%$	$62.68\,\%$
Object d. module active	$3.2\,\%$	$24.86\,\%$	$37.31\,\%$
Module changes	$4.2\,\%$	$5.98\,\%$	$29.2\,\%$

An important observation considering the use of value estimates was made using the

arbitration agent in different environments. In every environment, the min and max value of the value estimates changed, see Table 12. The change is particularly high for the exploration module. This presents a notable challenge of using any value estimates as state inputs. While the value estimates can be normalized to a range of $[0, 1]$, as also done in this approach, this would prove challenging in a real-world application with no clear environment transitions to indicate that the bounds have to be recalculated.

Table 12. Module value estimate ranges in different environments

Module	**Value estimate**	**Environment**		
		Cube	**Exploration**	**Multi**
Object detection	Min	0.0	0.0	0.0
	Max	5.92	4.8	5.99
Exploration	Min	-2.4	-1.55	-0.76
	Max	1.17	1.13	1.02

6. Real-World Implementation and Evaluation

As a final validation of the modular agent approach, we conduct real-world experiments using the *Hiwonder JetAuto Pro* robot platform. Each sub-task module and the arbitration agent are transferred directly to the robot without additional training.

The experimental setup with the used hardware, environmental scene, and the system integration and deployment are described in Section 6.1. The results are analyzed in 6.2 by evaluating each module's performance and comparing it with the simulation results. As a conclusion, the results are discussed and key insights are worked out for possible improvements in 6.3.

6.1 Experimental Setup

6.1.1 Hardware Platform and System Architecture

The experiments were performed on the *JetAuto Pro* robot platform from *Hiwonder*, a four-wheeled mobile robot with mecanum wheels and various sensors, see Figure 19. This experiment uses the *SLAMTEC A1 Lidar*, the 3D RGB-D camera *Orbbec Astra*, and a real-time IMU. The system is powered by an *NVIDIA Jetson Nano*, which runs Ubuntu 18.04 and ROS Melodic for sensor communication and control [1]. The decision for the robot platform was based on the wide range of available sensors, the already existing ROS sensor integration, and the reasonable price.

6.1.2 Environmental Setup and System Integration

The experiments were conducted in a controlled indoor environment of 4×3.5 meters with furniture at each wall, including a hallway, see Figure 21. The trained agents were executed on an external computer as a ROS node, which subscribed to the required sensor topics and published the linear and angular velocity commands. The RGB-D images are scaled down to 320×240 pixels before being processed by each agent, the lidar scan is transformed from a point cloud to the OGM and the IMU data is transformed into euler coordinates.

Figure 19. JetAuto Pro robot from Hiwonder [1]

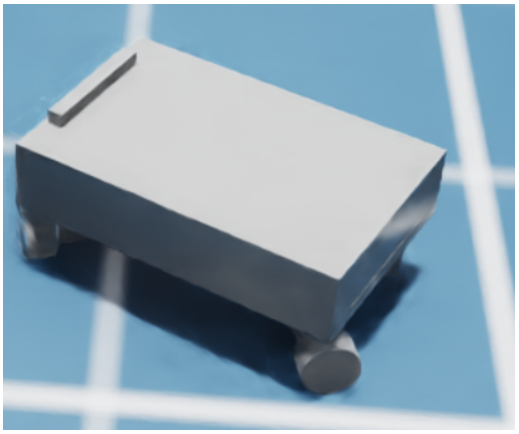


Figure 20. Robot model used in simulation

(a) Without obstacles

(b) With obstacles

Figure 21. Real-world test environments

6.2 Performance Evaluation and Comparison

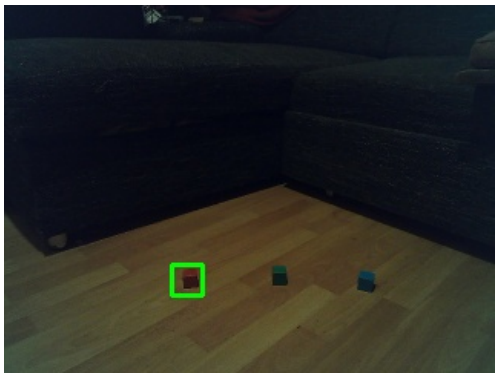
Each of the three agents is evaluated separately in the real-world environment by executing similar tasks as in the simulation. Where possible, results are compared to the simulation results to evaluate the sim-to-real transfer capabilities of the agents.

6.2.1 Object Detection Agent Sim-to-Real

First, the fine-tuned object detection network is tested under real-world conditions. Figure 22 shows the detection result for different scenarios. The prediction accuracy varies depending on the distance between camera and cube. With the minimum distance of $0.7\,\mathrm{m}$, resulting in the robot chassis being $25\,\mathrm{cm}$ distanced from the cube, the classification accuracy is around $63\,\%$. At $1.0\,\mathrm{m}$ distance, see Figure 22a, the classification accuracy increases to $77\,\%$ and reduces to $35\,\%$ at $1.5\,\mathrm{m}$ and $11\,\%$ at $2.0\,\mathrm{m}$, see Figure 22b. If multiple cubes with different colors are present, the network still correctly recognizes the red cube with a classification certainty of up to $87\,\%$, with the other two cubes having a value of $65\,\%$ and $61\,\%$ from left to right, see Figure 22c. Adding other red objects to the scene, see Figure 22d, results in the network still correctly identifying the red cube with a classification certainty of $80\,\%$ and no classifications for the other red objects. This shows that the network's main feature for the detection is the shape of the object, not so much the color of it, which becomes an issue if other objects but the cube are within the scene. For a more reliable detection, the network could be re-trained with more diverse objects in the scene, to ensure that the color has a stronger influence on the classification.

(a) Red cube with detection box, $1\,\mathrm{m}$ distance

(b) Red cube with detection box, $2\,\mathrm{m}$ distance



(c) Multiple cubes with detection box around red cube

(d) Multiple red objects with detection box around red cube

Figure 22. Object detection network detection results real-world

The second evaluation is done by running the object detection agent in the real-world

environment while measuring certain metrics. For this the agent is tested 20 times with a random orientation and the cube in a random location with a distance of $1.6\,\mathrm{m}$, which equals the average distance of the simulation test results. Every $15\,\mathrm{Hz}$ a new observation is analyzed by the agent, with the action generation taking on average $28\,\mathrm{ms}$. Table 13 shows the test results and the comparison to the simulation. If no other objects were mistakenly classified for the red cube, the agent always successfully detected and reached the cube. Only if other objects were mistaken for the cube the agent failed. This confirms the results from above that the object detection network needs to be more robust against other objects in the scene. While the average time per episode is three times higher than in the simulation, the real-world robot also drives four times slower than the simulation robot. The action change rate shows very similar results for the real-world and the simulation.

To summarize, the real-world transfer of the agent was successful. For a more robust detection, especially with more obstacles in the scenery, the object detection network needs to be improved.

Table 13. Real-world test results object detection agent

Metric	**Value**	**Compared to simulation**
Total episodes	20	$2\,\%$
Average initial cube distance	$1.6\,\mathrm{m}$	$100\,\%$
Average time per episode	$20.15\,\mathrm{s}$	$316\,\%$
Task success rate	$90\,\%$	$91.3\,\%$
Average processing time per observation	$28\,\mathrm{ms}$	-
Action difference above 0.25	$8\,\%$	$117\,\%$
Action difference above 0.5	$2.7\,\%$	$105\,\%$
Action difference above 0.75	$1.8\,\%$	$80\,\%$

6.2.2 Exploration Agent Sim-to-Real

The exploration agent was evaluated in the test environment with obstacles. Due to issues with the lidar and localization, only a depth image based obstacle avoidance was evaluated, limiting the comparability with the test results from the simulation. Basic comparisons can still be made, as the same reward functions were used for the training. Figure 23 shows two exemplary depth images from the test environment. It can be seen that those real-world depth images contain multiple black spots, which are areas where the depth sensor was not able to measure a distance. Those were replicated in the simulation by the random black areas added to the depth images, see Figure 10.

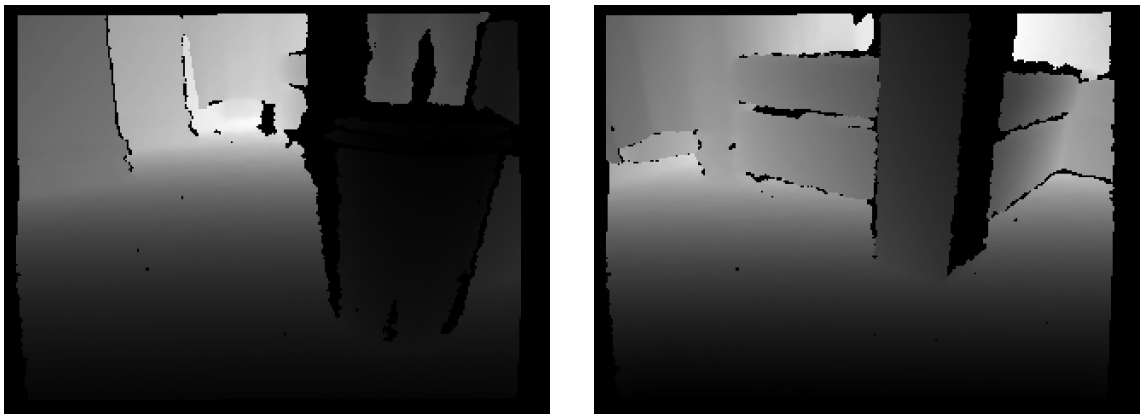


Figure 23. Real-world depth images

Table 14 shows the obtained test results, with the agent driving through the environment till colliding with an obstacle. The agent was mostly successful in identifying and avoiding obstacles, with an average time till collision of $61.7\,\mathrm{s}$. At the same time this only translates to a traveled distance of $5.5\,\mathrm{m}$ on average. This is mostly caused by an uncertain behavior of the agent, with lot of stopping and slow driving, further amplified by the slow processing time of $95\,\mathrm{ms}$. Additionally, due to only analyzing a sliced section of the depth image, the robot can only avoid obstacles of a certain height.

Table 14. Real-world test results exploration agent

Metric	**Value**
Total episodes	20
Average time till collision	$61.7\,\mathrm{s}$
Average distance traveled	$5.5\,\mathrm{m}$
Average processing time per observation	$95\,\mathrm{ms}$

6.2.3 Arbitration Agent Sim-to-Real

The arbitration agent is evaluated in three different real-world scenarios:

- Environment without obstacles but cube
- Environment with obstacles¸
- Environment with obstacles and cube

For the agent the *vpr* version is used, as it achieved the best results in the simulation. This means the value estimates with value cross-evaluation, cross-evaluated actions using log

probs and ratio of policies are used as state inputs, see equation (5.14). As already noted in the performance evaluation of the simulation, Section 5.4.3, the value estimate ranges differ a lot depending on the environment. This is an even bigger issue in the real-world scenario, with the exploration module value estimates now being in a range from $[-0.23, 0.61]$ and the object detection modules value estimates in a range from $[0.365, 1.272]$. In the simulations the value ranges were $[-2.4, 1.17]$ and $[0.0, 5.99]$ for the exploration and object detection module respectively, see Table 12. This required adjusting the min and max values for the normalization function of the arbitration agent.

Table 15 shows the real-world test results of the arbitration agent in a obstacle free environment with the cube in a random location. The task success rate was $50\,\%$, with the object detection module being reliable selected once the cube was visible. The module selection and activity share of the respective modules is not directly comparable to the simulated results, due to the usage of an exploration module without lidar. The much higher average processing time of $257\,\mathrm{ms}$ shows the increased computational requirement due to 2 value networks and 3 policy networks being calculated.

Table 15. Real-world test results arbitration agent cube environment

Metric	**Value**
Total episodes	10
Average initial cube distance	$1.6\,\mathrm{m}$
Average observations per episode	412
Average time per episode	$130\,\mathrm{s}$
Task success rate	$50\,\%$
Average processing time per observation	$257\,\mathrm{ms}$
Exploration module active	$80.1\,\%$
Object d. module active	$19.8\,\%$
Module changes	$21.3\,\%$

The results for the obstacle environment are shown in Table 16. It can be seen that the results are significantly worse than for the individual exploration agent. This is caused by the object detection module classifying parts of the objects as a cube, resulting in excessive module switching and early collisions.

Table 16. Real-world test results arbitration agent obstacle environment

Metric	**Value**
Total episodes	10
Average time till collision	23.3 s
Average distance traveled	2 m
Average processing time per observation	252 ms
Exploration module active	48.65 %
Object d. module active	51.34 %
Module changes	16.3 %

Similar results are observed in the multi environment, see Table 17. Due to too many wrong classifications by the object detection module, the robot barely moves and, if it does, collides quickly with an obstacle.

Table 17. Real-world test results arbitration agent multi environment

Metric	**Value**
Total episodes	10
Average time till collision	33.1 s
Cube found percentage	20 %
Average distance traveled	3 m
Average processing time per observation	246 ms

6.3 Discussion and Insights

6.3.1 Key Challenges and Solutions

The agents' designs and validation within the simulations, followed by the real-world transfer, exposed multiple challenges with each agent and the proposed arbitration framework.

While the object detection agent achieved a very good task success rate in the simulation and real-world, it showed clear limitations if other objects with a similar shape to the red cube were present in the environment. The solution for this would be further training of the fine-tuned object detection module. An interesting aspect would be to validate that changing the object detection module does not require retraining the object detection agent.

For the exploration agent, multiple challenges arise. The training never achieved the

same stable results as the object detection agent, indicating problems with the network architecture, the state inputs, or the algorithm parameters. A possible solution could be further modularizing this agent to allow a separate training of a lidar and depth image module, possibly by using frontiers and a way-point-based approach for the navigation. This would allow for easier error detection and targeted improvements. Furthermore, the real-world lidar data was much more noisy than expected, resulting in issues with the localization needed for generating the OGM.

The arbitration agent's main challenge is an effective approach for normalizing the value estimates within changing environments. The only applicable solution would be to use a running min-max scaler, resulting in additional challenges for correctly setting the window size. Additionally, the learning process did not arrive at a stable solution. While the non-optimal integrated modules partly caused this, it also indicates that the used state inputs are not sufficient to achieve an optimal policy.

6.3.2 Lessons Learned

The main lesson learned is that modularity is the most reliable and stable approach to reinforcement learning and neural networks. This applies to the proposed modular framework and to each agent separately. Individual verifiable modules allow for continuous development progress and subsequent goal-oriented improvements. The fine-tuned object detection network is a perfect example of this approach, which could also be applied to multiple other components.

Transferring the trained agent from the simulation to the real world generated mixed results. While the initial transfer was surprisingly simple, and it was easy to obtain the first results, refining those into a reliable system by adjusting the simulation parameters proved very challenging. This clearly shows the limitations of zero-shot transfers.

6.3.3 Recommendations for Future Implementations

While using already existing actor-critic algorithms for combining modules is an interesting and easily repeatable approach, it has clear limitations regarding the obtainable state inputs. For future implementations trying to use the internal neural network representations of the modules for the arbitration, some additional value for rating the action certainty of each module should be considered. This could be implemented by including the standard deviation as a learned parameter to the policy network.

Nonetheless, a more feasible approach for the arbitration, especially in a real-world scenario, is to use environment observations as state inputs. While this comes with disadvantages like increased state dimensions and a more difficult integration of new modules, the resulting agent should be more reliable. By relying on the other modules networks for the selection, any imprecision or errors in a module are passed on to the selection logic.

7. Conclusion and Future Work

7.1 Research Questions and Hypotheses Revisited

The research questions outlined in Section 1 are now revisited.

1. How effectively can an arbitration agent, utilizing only policy and value network outputs and cross-evaluation of actions, combine multiple specialized reinforcement learning agents to solve complex tasks in robotic applications?

Our research showed that the internal neural network representation, meaning the policy network log probabilities and value function estimates, provides mostly sufficient information to the arbitration agent for solving the original task. Based on those values, the agent was able to select the appropriate module in different situations. An important precondition is that the environment in which the arbitration agent was trained is similar to the test environment.

2. How does the performance of the combined system compare to that of the individual agents?

The comparison of the performance of the combined system was very dependent on the analyzed task and the test environment. While the object detection modules' performance was deprecated in the combined approach, the exploration agents' performance slightly increased.

3. How successful can this approach be transferred to a real-world robotic system?

While the basic transfer of the modules was successful, the specific arbitration framework using the module's value estimates as state inputs does not seem to be a reliable approach for real-world robotic systems. Due to the high variance in value estimates based on the environment and no practical solution to normalize those values in dynamic settings, a value estimate-based approach does not seem applicable to a real-world setting.

This results in the following conclusions for the hypotheses stated:

- A modular arbitration framework using actor-critic agents and arbitration based on

policy and value network outputs can effectively address complex tasks in robotic applications.
  - Yes, but only in a static environment. Any greater changes in the environment affecting the value estimates of the modules will degrade the performance.
- Cross-evaluation of actions using the other modules' policy and value networks will improve the arbitration agent's decision-making.
  - Yes, cross-evaluation was a useful additional input for the arbitration agent, improving its decision-making.
- No modifications to the used algorithm are required to implement the proposed framework.
  - Yes, the arbitration framework was implemented without any modifications to the PPO algorithm.
- The performance of the combined system will be comparable to that of the individual agents.
  - Inside the simulation, the performance was comparable to the individual agents. At the same time, it also became clear that the proposed arbitration mechanism will not result in an optimal solution but will always cause some unnecessary module changes.
- This approach can be successfully transferred to a real-world robotic system.
  - No, the value estimates deviate too much on a real-world system. Additionally, any uncertainties by the integrated modules are forwarded to the arbitration agent, making the decisions unstable.

To summarize, the basic arbitration framework approach works but is unsuitable for a real-world robotic system.

7.2 Critical Analysis

Apart from the arbitration framework's limited applicability to real-world robotic systems, the approach has some additional limitations and drawbacks. While the internal network representations were used to limit the state dimensions and thereby computational overhead, requiring the value network together with the policy network probably negates those advantages. Additionally, every module must evaluate the current state at each timestep, resulting in additional computational requirements. Certain hierarchical approaches wait for the active module to terminate before other modules are activated, reducing computational requirements.

The discrete action space only allows one module to be active at a time, and the internal network representations do not seem sufficient for an optimal module selection. Due to

the cross-evaluation of actions, the state space grows quadratically with the number of modules. Additionally, the complexity of the training will increase significantly with more modules.

Using a four-wheel steering robot model instead of the actual mecanum wheels used by the *Hiwonder JetAuto* resulted in a less smooth motion once transferred to the real world and did not allow for the full range of possible movements. All real-world tests of the agents only happened in a controlled test environment with no external influences.

As a last limitation of this research, no alternative approaches for solving the proposed task were analyzed, preventing any reliable comparison of our modular approach or an assessment of modular approaches in general.

7.3 Future Research Directions

There are several directions in which the research in this thesis can be improved. An interesting topic would be to extend the flat modular approach with a hierarchical component, which could allow for more efficient computations and even more reusability of elemental skills. The state inputs for the arbitration agent could be improved by adding an action certainty estimate from each module, resulting in a more reliable selection process. Another interesting research would be combining modules with different action spaces, with multiple active modules simultaneously.

7.4 Conclusion

This thesis presented a novel modular arbitration framework that allows decomposing complex tasks into sub-tasks and coordinating those modules to reach the original objective. All results were validated in simulation and the real world.

Our research shows the possibility of using an actor-critic-based modular design without requiring any new RL algorithms or modifications. While the benefits of a modular approach are evident for any step in the design of RL agents, using the policy and value networks as inputs for an arbitration mechanism showed its limitations.

Bibliography

[1] Hiwonder, "JetAuto Pro Robot." `https://www.hiwonder.com/products/jetauto-pro`. Accessed: 2024-11-28.

[2] S. Sundaram, M. Buss, and Y. Matsuoka, "Deep Reinforcement Learning for Industrial Robotics: A Review," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 6, pp. 455–481, 2023.

[3] T. Yu, D. Quillen, Z. He, R. Julian, K. Hausman, C. Finn, and S. Levine, "Meta-World: A Benchmark and Evaluation for Multi-Task and Meta Reinforcement Learning," in *Conference on Robot Learning*, pp. 1022–1033, PMLR, 2020.

[4] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning," *Artif. Intell.*, vol. 112, pp. 181–211, 1999.

[5] T. Wang, H. Zhang, H. Yu, Y. Liu, Y. Zheng, C. Fan, Z. Lan, G. Pan, Z. Zhang, *et al.*, "JointPPO: Diving Deeper into the Effectiveness of PPO in Multi-Agent Reinforcement Learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 17804–17818, 2022.

[6] T. G. Dietterich, "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition," *Journal of Artificial Intelligence Research*, vol. 13, pp. 227–303, 2000.

[7] P. Sharma, L. Pinto, and A. Gupta, "Deep Reinforcement Learning for Dexterous Manipulation with Concept Networks," in *Conference on Robot Learning*, pp. 83–92, PMLR, 2018.

[8] S. Russell and A. Zimdars, "Q-decomposition for reinforcement learning agents," *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 656–663, 2003.

[9] C. Simpkins and C. Isbell, "Composable Modular Reinforcement Learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4975–4982, Jul. 2019.

[10] V. Gupta, D. Anand, P. Paruchuri, and A. Kumar, "Action Selection for Composable Modular Deep Reinforcement Learning," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2021.

[11] C. Huang, Y.-Y. Su, and D. Pathak, "Total Singulation With Modular Reinforcement Learning," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 4117–4124, 2021.

[12] R. Bellman, "A Markovian Decision Process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957.

[13] M. Minsky, "Steps Toward Artificial Intelligence," *Proceedings of the IRE*, vol. 49, no. 1, pp. 8–30, 1961.

[14] R. S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988.

[15] C. J. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, 1992.

[16] V. Gullapalli, "A Learning Approach to Robotic Motion Control," in *Proceedings of the 1990 Conference on Systems, Man and Cybernetics*, pp. 48–53, IEEE, 1990.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[18] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.

[19] J. Kober, J. A. Bagnell, and J. Peters, "Deep Reinforcement Learning for Robotics: A Survey of Real-World Successes," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[20] B. Ibarz, J. Tan, C. Finn, D. Kalashnikov, A. Herzog, and S. Levine, "How to Train Your Robot with Deep Reinforcement Learning: 12 Challenges," *The International Journal of Robotics Research*, vol. 40, no. 4-5, pp. 409–433, 2021.

[21] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, M. Plappert, G. Powell, A. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang, "Solving Rubik's Cube with a Robot Hand," *arXiv preprint arXiv:1910.07113*, 2019.

[22] R. Caruana, "Multitask learning," *Machine Learning*, vol. 28, no. 1, pp. 41–75, 1997.

[23] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," *Proceedings of the sixteenth international conference on machine learning*, pp. 278–287, 1999.

[24] J. L. Elman, "Learning and development in neural networks: The importance of starting small," *Cognition*, vol. 48, no. 1, pp. 71–99, 1993.

[25] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *International Conference on Machine Learning*, pp. 1126–1135, PMLR, 2017.

[26] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation Learning," *ACM Computing Surveys (CSUR)*, vol. 50, pp. 1–35, 2017.

[27] A. Rajeswaran, V. Kumar, A. Gupta, G. Vezzani, J. Schulman, E. Todorov, and S. Levine, "Learning complex dexterous manipulation with deep reinforcement learning and demonstrations," in *Robotics: science and systems*, 2018.

[28] P. Dayan and G. E. Hinton, "Feudal Reinforcement Learning," in *Advances in neural information processing systems*, pp. 271–278, 1993.

[29] R. Parr and S. Russell, "Reinforcement learning with hierarchies of machines," in *Advances in neural information processing systems*, pp. 1043–1049, 1998.

[30] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. B. Tenenbaum, "Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation," *Advances in neural information processing systems*, vol. 29, 2016.

[31] B. Xue, Y. Wang, W. Xia, Z. Huang, and X. Chen, "Developmental Modular Reinforcement Learning," in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pp. 2349–2356, IEEE, 2020.

[32] O. Krömer, S. Niekum, and G. Konidaris, "A review of robot learning for manipulation: Challenges, representations, and algorithms," *arXiv preprint arXiv:1907.03108*, 2019.

[33] K. Hausman, J. T. Springenberg, Z. Wang, N. Heess, and M. Riedmiller, "Learning an embedding space for transferable robot skills," *arXiv preprint arXiv:1804.08632*, 2018.

[34] H. Sun and I. Guyon, *Modularity in Deep Learning: A Survey*, p. 561–595. Springer Nature Switzerland, 2023.

[35] N. Sprague and D. H. Ballard, "Multiple-goal reinforcement learning with modular sarsa(0)," in *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 2*, pp. 1073–1078, 2004.

[36] J. Andreas, D. Klein, and S. Levine, "Composable Modular Reinforcement Learning," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 156–165, JMLR. org, 2017.

[37] M. Ghasemi, A. H. Moosavi, I. Sorkhoh, A. Agrawal, F. Alzhouri, and D. Ebrahimi, "An Introduction to Reinforcement Learning: Fundamental Concepts and Practical Applications," *arXiv preprint arXiv:2408.07712*, 2024.

[38] R. Güldenring, "Applying Deep Reinforcement Learning in the Navigation of Mobile Robots in Static and Dynamic Environments," *Universität Hamburg*, 2019. Retrieved from https://tams.informatik.uni-hamburg.de/publications/2019/MSc_Ronja_Gueldenring.pdf.

[39] M. Gasic, "Temporal-difference methods." https://www.cs.hhu.de/fileadmin/redaktion/Fakultaeten/Mathematisch-Naturwissenschaftliche_Fakultaet/Informatik/Dialog_Systems_and_Machine_Learning/Lectures_RL/L3.pdf. Accessed: 2024-12-03.

[40] Y. Fenjiro and H. Benbrahim, "Deep reinforcement learning overview of the state of the art," *Journal of Automation, Mobile Robotics and Intelligent Systems*, vol. 12, no. 3, pp. 20–39, 2018.

[41] V. R. Konda and J. N. Tsitsiklis, "Actor-Critic Algorithms," in *Advances in Neural Information Processing Systems*, pp. 1008–1014, 2000.

[42] 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.

[43] E. Institute, "How do policy gradient methods optimize the policy, and what is the significance of the gradient of the expected reward with respect to the policy parameters?." https://eitca.org/artificial-intelligence/eitc-ai-arl-advanced-reinforcement-learning/deep-reinforcement-learning/policy-gradients-and-actor-critics/examination-review-policy-gradients-and-actor-critics/how-do-policy-gradient-methods-optimize-the-policy-and-what-is-the- 2024. Accessed: 2024-11-25.

[44] H. Face, "Deep Reinforcement learning Course." https://huggingface.co/learn/deep-rl-course/unit1/what-is-rl, 123. Accessed: 2024-10-15.

[45] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[46] OpenAI, "Part 3: Intro to Policy Optimization." `https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html`, 2020. Accessed: 2024-12-03.

[47] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, "Trust Region Policy Optimization," in *International conference on machine learning*, pp. 1889–1897, PMLR, 2015.

[48] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[49] D. Bick, "Towards Delivering a Coherent Self-Contained Explanation of Proximal Policy Optimization," Master's thesis, Artificial Intelligence, University of Groningen, 2021.

[50] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in Robotics: A Survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[51] J. Pfeiffer, S. Ruder, I. Vulić, and E. M. Ponti, "Modular Deep Learning," *arXiv preprint arXiv:2302.11529*, 2024.

[52] R. Yang, H. Xu, Y. Wu, and X. Wang, "Multi-Task Reinforcement Learning with Soft Modularization," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 3019–3030, 2020.

[53] D. Hadfield-Menell, A. Dragan, P. Abbeel, and S. Russell, "Inverse Reward Design," in *Advances in Neural Information Processing Systems*, pp. 6765–6774, 2017.

[54] J. Fu, A. Singh, D. Ghosh, L. Yang, and S. Levine, "Variational Inverse Control with Events: A General Framework for Data-Driven Reward Definition," in *Advances in Neural Information Processing Systems*, pp. 8538–8548, 2018.

[55] M. Wolf, M. Schmitt, D. Renner, and M. Bengel, "Deep Reinforcement Learning with Reward Shaping for Autonomous Driving," in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pp. 1–6, IEEE, 2020.

[56] R. Portelas, C. Colas, P. Fournier, O. Sigaud, and M. Chetouani, "TeachMyAgent: A Benchmark for Automatic Curriculum Learning in Deep RL," in *International Conference on Machine Learning*, pp. 7853–7862, PMLR, 2020.

[57] G. Jerfel, E. Grant, T. L. Griffiths, and P. Shafto, "Learning to Learn with Generative Models of Tasks," in *Advances in Neural Information Processing Systems*, vol. 34, pp. 20954–20966, 2021.

[58] H. Zou, T. Ren, D. Yan, H. Su, J. Wen, L. Weng, and H. Huang, "Reward Shaping via Meta-Learning," *arXiv preprint arXiv:1901.09330*, 2019.

[59] M. E. Taylor and P. Stone, "Transfer learning for reinforcement learning domains: A survey," *Journal of Machine Learning Research*, vol. 10, no. 7, pp. 1633–1685, 2009.

[60] S. K. Sharma, H. Abiri, B. Paudel, and D. Zhao, "Hierarchical and Modular Deep Reinforcement Learning: A Survey," *arXiv preprint arXiv:2302.12891*, 2023.

[61] M. Bouton, K. Julian, A. Nakhaei, K. Fujimura, and M. J. Kochenderfer, "Decomposition Methods with Deep Corrections for Reinforcement Learning," *arXiv preprint arXiv:1802.01772*, 2019.

[62] G. Bakirtzis, M. Savvas, R. Zhao, S. Chinchali, and U. Topcu, "Reduce, Reuse, Recycle: Categories for Compositional Reinforcement Learning," *arXiv preprint arXiv:2408.13376*, 2024.

[63] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World," *arXiv preprint arXiv:1703.06907*, 2017.

[64] N. Jakobi, P. Husbands, and I. Harvey, "Noise and the reality gap: The use of simulation in evolutionary robotics," *Advances in artificial life*, pp. 704–720, 1995.

[65] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-real: Learning agile locomotion for quadruped robots," in *Proceedings of Robotics: Science and Systems*, 2018.

[66] K. Kelchtermans and T. Tuytelaars, "RARA: Zero-shot Sim2Real Visual Navigation with Following Foreground Cues," *arXiv preprint arXiv:2201.02798*, 2022.

[67] G. Dulac-Arnold, D. Mankowitz, and T. Hester, "Challenges of Real-World Reinforcement Learning," *arXiv preprint arXiv:1904.12901*, 2019.

[68] D. Mwiti, "10 Real-Life Applications of Reinforcement Learning." `https://neptune.ai/blog/reinforcement-learning-applications`, 09 2024. Accessed: 2024-12-05.

[69] DATAmadness, "TensorFlow 2 - CPU vs GPU Performance Comparison." `https://datamadness.github.io/TensorFlow2-CPU-vs-GPU`, 10 2019. Accessed: 2024-12-07.

[70] NVIDIA, "Isaac Sim." https://developer.nvidia.com/isaac-sim, 2024. Version: 4.1.

[71] M. Mittal, C. Yu, Q. Yu, J. Liu, N. Rudin, D. Hoeller, J. L. Yuan, R. Singh, Y. Guo, H. Mazhar, A. Mandlekar, B. Babich, G. State, M. Hutter, and A. Garg, "Orbit: A Unified Simulation Framework for Interactive Robot Learning Environments," *IEEE Robotics and Automation Letters*, vol. 8, no. 6, pp. 3740–3747, 2023.

[72] Open Robotics, "ROS.org | Powering the world's robots." https://www.ros.org, 2024. Version: Noetic Ninjemys.

[73] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *arXiv preprint arXiv:1506.01497*, 2016.

[74] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, and H. Adam, "Searching for MobileNetV3," *arXiv preprint arXiv:1905.02244*, 2019.

[75] M. Tkachenko, M. Malyuk, A. Holmanyuk, and N. Liubimov, "Label Studio: Data labeling software," 2020-2024. Open source software available from https://github.com/HumanSignal/label-studio.

[76] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Phil-Yannick Borkenhagen

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Modular Arbitration Framework for Robotic Reinforcement Learning with Sim-to-Real Transfer", supervised by Aleksei Tepljakov and Vladimir Kuts
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

16.12.2024

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Code Repository

All code and instructions for the implementation of the individual modules, the modular arbitration framework and the experiments can be found at `https://github.com/PhixedAP/modular-ppo-sim2real`.