INVERSE REINFORCEMENT LEARNING AND AFFORDANCES

A master thesis about Imitation Learning in the context of Affordances

JAN-PHILIPP SCHRAMM

MATRICULATION NUMBER: 4516495

1. REVIEWER: PROF. DR. GABRIEL ZACHMANN
2. REVIEWER: DR. FELIX PUTZE

Computer Graphics and Virtual Reality Research Lab
Faculty 03: Mathematics/Computer Science
University of Bremen

December 2024

DECLARATION

Nachname: Schramm Matrikelnr.: 4516495
Vorname.: Jan-Philipp

A) Eigenständigkeitserklärung
Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet, dazu zählen auch KI-basierte Anwendungen oder Werkzeuge. Die Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht. Die elektronische Fassung der Arbeit stimmt mit der gedruckten Version überein. Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

☒ Ich habe KI-basierte Anwendungen und/oder Werkzeuge genutzt und diese im Anhang "Nutzung KI basierte Anwendungen" dokumentiert.

B) Erklärung zur Veröffentlichung von Bachelor- oder Masterarbeiten
Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten. Archiviert werden:

1. Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Abschlussarbeiten
2. Bachelorarbeiten des jeweils ersten und letzten Bachelorabschlusses pro Studienfach und Jahr

☒ Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☐ Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☐ Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

C) Einverständniserklärung zur Überprüfung der elektronischen Fassung der Bachelorarbeit / Masterarbeit durch Plagiatssoftware
Eingereichte Arbeiten können nach § 18 des Allgemeinen Teil der Bachelor- bzw. der Master- prüfungsordnungen der Universität Bremen mit qualifizierter Software auf Plagiatsvorwürfe untersucht werden.
Zum Zweck der Überprüfung auf Plagiate erfolgt das Hochladen auf den Server der von der Universität Bremen aktuell genutzten Plagiatssoftware.

- [X] Ich bin damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum oben genannten Zweck dauerhaft auf dem externen Server der aktuell von der Universität Bremen genutzten Plagiatssoftware, in einer institutionseigenen Bibliothek (Zugriff nur durch die Universität Bremen), gespeichert wird.

- [ ] Ich bin nicht damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum o.g. Zweck dauerhaft auf dem externen Server der aktuell von der Universität Bremen genutzten Plagiatssoftware, in einer institutionseigenen Bibliothek (Zugriff nur durch die Universität Bremen), gespeichert wird.

Mit meiner Unterschrift versichere ich, dass ich die obenstehenden Erklärungen gelesen und verstanden habe und bestätige die Richtigkeit der gemachten Angaben.

Bremen, den 19.12.2024	
Ort, Datum	Unterschrift

ABSTRACT

Manually creating a reward function can be difficult. Instead, alternative techniques known as imitation learning can bypass this step. This thesis investigates the effectiveness of imitation learning in learning reward functions or policies to imitate experts behavior for affordances within a virtual environment. Various categories in imitation learning leverage a small set of expert demonstrations as the sole input for imitation; however, this introduces problems with sparse rewards and generalization. Techniques to counter that are task rewards and hyperparameter optimization. This research uses these techniques and imitation learning algorithms that operate in continuous space and model-free environments. An implementation in Unreal Engine, combined with a VR headset for trajectory recording and Python for training, is the base for evaluation. While the algorithms did not execute the task reliably despite using task reward, using hyperparameter optimization in some cases surpassed manually tuned hyperparameters.

ZUSAMMENFASSUNG

Die manuelle Erstellung einer Belohnungsfunktion kann schwierig sein. Stattdessen kann dieser Schritt durch alternative Techniken, die als Imitationslernen bekannt sind, umgangen werden. Diese Arbeit untersucht die Effektivität des Imitationslernens beim Erlernen von Belohnungsfunktionen oder Policies, um das Verhalten von Experten für Affordances in einer virtuellen Umgebung zu imitieren. Verschiedene Kategorien des Imitationslernens nutzen einen kleine Menge an Expertenvorführungen als einzigen Input für die Imitation. Dies führt jedoch zu Problemen mit spärlichen Belohnungsfunktionen und Generalisierung. Techniken, die dem entgegenwirken, sind Aufgabenfunktionen und Hyperparameter-Optimierung. Diese Arbeit verwendet diese Techniken und Imitationsalgorithmen, die im kontinuierlichen Raum arbeiten und modellfrei sind. Eine Implementierung in der Unreal Engine, kombiniert mit einem VR-Headset für die Aufzeichnung von Trajektorien und Python für das Training, bildet die Grundlage für die Bewertung. Während die Algorithmen die Aufgabe trotz der Verwendung der Aufgabenfunktion nicht zuverlässig ausführten, übertrafen sie mit der Hyperparameter-Optimierung in einigen Fällen manuell eingestellte Hyperparameter.

When a configuration is reached for which the action is undetermined, a random choice for the missing data is made and the appropriate entry is made in the description, tentatively, and is applied. When a pain stimulus occurs all tentative entries are cancelled, and when a pleasure stimulus occurs they are all made permanent.

— Alan Turing [65]

ACKNOWLEDGMENTS

I would like to thank Prof. Dr. Gabriel Zachmann, who advised me on my approach. I'm also highly grateful to my supervisor, Hermann Meißenhelter, who always stood by me with help and advice and provided valuable guidance throughout my work.

I sincerely thank all the people who helped me make the existing software for a newer version of Unreal Engine work, and those who provided valuable feedback on my approach. Your support and insights have been invaluable to me.

Lastly, I would like to mention my family, especially my brother, who helped me with the VR hardware.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

AGI Artificial general intelligence
AIL Adversarial imitation learning
AIRL Adversarial inverse reinforcement learning
BC Behavioral cloning
GAIL Generative adversarial imitation learning
GAN Generative adversarial networks
IL Imitation learning
IRL Inverse reinforcement learning
MDP Markov decision process

PPO Proximal policy optimization

RL Reinforcement learning

TCP Transmission control protocol

UDP User datagram protocol

UE Unreal Engine

VR Virtual reality

GLOSSARY

Adversarial inverse reinforcement learning
AIL algorithm that infers both the reward function and a policy based on the demonstrations [24].

Adversarial imitation learning
An approach to recover the reward function or policy based on a set of demonstrations, but this with an algorithm that uses GAN to infer said products [69].

Affordance
At its core, this refers to the capability an object offers. For example, a chair affords to sit on, or a bottle opener affords to open a bottle.

Artificial general intelligence
Algorithm that can reason or learn any skills and tasks that a human being can perform [53].

Behavioral cloning
BC is an IL algorithm that directly maps states to available actions. Most commonly, only using the expert's demonstrations [67].

Blueprint
Blueprints allow for visual scripting of gameplay elements within an interface inside the Unreal Engine Editor [21].

Creabots
A master project that, among other things, evaluates the suitability and limitations of virtual environments implemented in different physics engines for simulating affordances and training RL algorithms on them [12].

Demonstration
A list of tuples representing the change of states over action done by an expert in a certain environment (recording is done, e.g., with VR).

Dense reward
Contrary to sparse rewards, this function always gives new information to better reinforce behavior [22].

Generative adversarial networks

A combination of two CNNs called generator and discriminator that compete against each other to create realistic images or voices, for example.

Generative adversarial imitation learning
Similar to AIRL, this is also an AIL algorithm but only offers the capability to infer the policy [33].

Gymnasium
A standard interface implementation for RL environments that also offers standardized environments for comparison of algorithms [64].

Imitation
Python module containing a collection of different imitation learning algorithms [28].

Imitation learning
Imitation learning is an umbrella term for the category of algorithms that try to imitate the behavior of an expert solely by their demonstrations or a policy [69].

Inverse reinforcement learning
An algorithmic approach to infer the reward function just by expert demonstrations for later RL training [5].

Learning Agents
A plug-in for Unreal Engine to train machine learning agents in an UE environment to train via RL or IL algorithms [9].

Markov decision process
A stochastic control problem is typically defined as a 4-tuple containing states, actions, transitions, and a reward function.

MindMaker
Unreal Engine plugin to enable UE to function as a Gymnasium environment for machine learning algorithms. [41].

MongoDB
MongoDB is a database that is required for the use of USemLog [47].

MuJoCo
Physics engine used in different areas, including machine learning [63].

Omniboard
A web dashboard to display sacred experiments with all information. Also, it can create plots of selected metrics [66].

Optuna
API to search a pre-defined hyperparameter space for the best set of hyperparameters for an algorithm in a defined number of runs [3].

Policy
Also known as π, is a function mapping a state $s \in S$ to either a single action $a \in A$ or a probability distribution over the

set of actions A. In other words, π is the (trained) ruleset that determines the following action to take on the current state s.

Proximal policy optimization
RL algorithm that limits the amount of change in each training step by using, e.g., a clip range [58].

Reinforcement learning
In simple terms, reinforcement learning is an approach to infer the policy for a task by interacting with a given environment. These interactions are evaluated by a reward function (also known as a cost function) and considered for further policy improvements.

Reward function
A mapping of state-action to a scalar value, which evaluates the behavior and gives feedback to reinforce a particular behavior.

Sacred
An open-source Python framework that provides functionalities to manage configurations, reproduce results, and more [31].

Sparse reward
A reward function that gives sporadically positive feedback. Most of the state-action combinations do not return more information than the combinations before [22].

Stable-Baselines3
Package containing different RL algorithms using PyTorch [52].

Task reward
A function that splits a task into different stages to evaluate these individual pieces with a constant reward [72].

TCP-Unreal
Unreal Engine TCP socket wrapper [25].

Trajectory
A synonym for demonstration.

Transmission control protocol
Connection-oriented internet protocol.

Unreal Engine
A game engine developed by Epic Games [20].

USemLog
Semantic logger used in this project to record the trajectories [55].

User datagram protocol
Connectionless internet protocol for data communication.

Virtual reality
Makes a virtual environment tangible with the help of technical aids such as a VR Headset.

Part I

CONTEXT AND FOUNDATIONS

The introduction, research context, problem statement, objectives, and essential theoretical or technical foundations.

1 INTRODUCTION

Humans usually do everyday activities like cleaning, cooking, or putting things away. Electronic helpers like robots could help us reduce the time we spend a day on such things. However, programming said robots proved difficult, especially in more complex situations [39]. Thus, training them with, e.g., reinforcement learning (RL) is used to create their policy to follow without explicitly programming every particular case in an algorithm. Instead, a reward function gives an agent feedback, which the algorithm wants to maximize to learn a specific behavior [34].

In recent decades, much research has focused on RL and how to improve it [6]. Creabots, a project from the University of Bremen, tried to incorporate RL with affordances (e.g., covering a pot) and examined its limitations and the capabilities of virtual environments in Unreal Engine (UE) compared to data in the real world [12]. The definition of affordances varies depending on the scientific field. Nevertheless, in this case, it just means the relationship between actions, objects, and their effects [48]. In other words, what is the capability of an object? For example, a bottle opener can open a bottle. Busse et al. [11] concluded in Creabots that a virtual environment is feasible as input for learning algorithms, even though the data might require some processing. However, more importantly, learning the policy for affordances with RL proved more challenging than expected. One of the reasons might be the reward function. Hence, they proposed a taxonomy that splits the learning process into multiple components. Abbeel et al. [1] even state that manually defining a reward function requires much manual tweaking, which adds a barrier to RL's success.

These problems raise the question of how to improve the RL approach to make it easier for a user to reach a satisfying policy. The group of imitation learning (IL) algorithms is a counterpart to RL to simplify the process. This field involves inverse reinforcement learning (IRL), where only data recorded from an expert is used instead of a specially defined reward function. This procedure derives a reward function from demonstrations and the desired policy through RL [5]. Interestingly, some authors like Silver et al. [59] even argue that agent reward maximization can contribute to a solution of artificial general intelligence (AGI). This means that IRL may lay the groundwork for AGI. Furthermore, IL also includes behavioral cloning (BC) and adversarial imitation learning (AIL). One thing all these categories have in common is the goal to imitate the behavior given an input, which mainly consists of a set of trajectories. However, the algorithms in the different categories choose different ways to achieve this goal. For example, BC and some of the algorithms from AIL, such as generative

adversarial imitation learning (GAIL), deal directly with learning the behavior, while other algorithms from the IRL category take a detour via a reward function [69]. Researchers raised the category of AIL due to issues like the computational complexity of IRL [33].

Based on the previously mentioned challenges of the reward functions in Creabots, this thesis tries to combine existing research in the field of IL to learn from expert demonstrations while querying all information needed from a virtual reality (VR) environment. Requirements that the algorithms should include are: works in continuous space, does not require any feature specification, is model-free, and has a stationary reward that does not change over time. However, recording trajectories is time-consuming, which means a user can only do a limited number of recordings, which might not produce a feasible policy due to problems with generalization. To be more precise, the intent of the demonstration does not get extrapolated to initial states not included in the input set of the algorithm [5]. To counter this problem, an approach known as task reward (a sparse reward function) can stabilize and accelerate the training [72]. Furthermore, the choice of hyperparameters can significantly enhance the performance of RL algorithms, yet scientists often overlook it [19]. Combined with task reward, this should improve the standard approach of using an arbitrary IL algorithm.

To evaluate this approach, task rewards and hyperparameter optimization with Optuna will be added to classical AIL algorithms. So, in the end, a comparison between the classical approach and the extension will be made. The implementation realized in Unreal Engine, including the training space and a virtual environment using a VR headset for recording the trajectories, is introduced in the later chapters, as well as an external Python script focusing on the actual algorithmic part of cloning the behavior. Although some successes occur, training the algorithms requires significant time.

In the next chapter, the text provides background information, including all the terms mentioned in the introduction. Related works follow this background. The methodology section explains the design choices and the implementation process. Finally, the evaluation compares the different approaches and concludes with suggestions for improvements.

The summary of the objectives of this thesis:

- Integration of UE for IL algorithms and recording of trajectories in VR.
- Testing the approach of task reward to tackle generalization problems with AIL algorithms like AIRL and GAIL and comparing them.
- Using a hyperparameter optimization method for the individual environments and IL algorithms, contrary to manually tuned parameters.

2

BACKGROUND

This chapter introduces the essential RL concepts, explaining important terminology and foundational principles. It will also discuss some of the limitations associated with this approach. Finally, this chapter will briefly cover IL and its importance to this work.

2.1 DEFINITIONS

Before delving deeper into the different fields of IL, it is essential to define some crucial terms to lay the groundwork for understanding how those concepts work.

2.1.1 *Affordances*

Gibson [26] introduced the term affordances in 1966 in his work as a substitute for values. To clarify, he says, these can be derived from the constant properties of an object and ultimately mean what an object offers to an observer. In later work, he underpins this with the example of a surface. It can be "walk-on-able," "sink-into-able," and more depending on the physical properties relative to the creature using it and how they perceive it (e.g., water bugs can stand on water while other animals cannot) [27].

Researchers have conducted extensive studies on the definition of affordances in various fields like psychology, neuroscience, and robotics [4, 38]. This thesis will focus on affordances as the relationship between objects and the actions that can be performed on them to produce an effect [48].

2.1.2 *Markov Decision Process*

Markov decision processes (MDPs) are frameworks that are, among other things, used as the base for the definition of RL [68]. It includes everything to model a sequential decision-making process and to use this model to find an optimal policy. While the formal definition varies in the literature [34, 50, 53, 68], its essence remains consistent. In this context, this thesis aligns closely with the equation definitions proposed by Hu [34] with discrete time steps $t \in \{0, 1, 2, ...\}$ and where upper case variables at a time step (e.g., A_t) represent random variables and lower case variables specify the result of these random variables. A MDP is defined by a tuple $M = \{S, A, P, R\}$ [34]. The definition can also include the discount factor δ, making it $M =$

$\{S, A, P, \delta, R\}$ [53]. Each component of the tuple, including δ, will be defined as follows [34, 53]:

- ***S*:** The set of all possible states for this particular MDP. Consider a simple 2×2 grid with one entity (the agent) as an example. The position of this entity defines the states in this case as $S = \{0, 1, 2, 3\}$.
- ***A*:** The set of all defined actions. In the previous example, this would be discrete movement actions (up, down, left, right).
- ***P*:** The function P is a transition mapping defined as $P(s'|s, a) = P[S_{t+1} = s'|S_t = s, A_t = a]$ which gives the transition probability to a state in s' based on a given state $s \in S$ and action $a \in A$. So, to say, it is the way how the MDP will react to an action.
- ***δ*:** Is the discount factor, where $\delta \in [0, 1]$ and weights future rewards. Introduced to eliminate the problem of infinite reward if a task has no termination goal.
- ***R*:** Another mapping $R(a, s) = \mathbb{E}[R_t|S_t = s, A_t = a]$, that gives a scalar value as an evaluation (or reward) to the given state s and action a. R actively guides a learning agent toward correct behavior by providing rewards and punishments.

Furthermore, in addition to the current tuple definition of the MDP, the problem statement still misses some functions for measuring the expected return and returning the best action according to a state, which is crucial to solving the MDP:

POLICY: Still missing is something to optimize the MDP. This is the policy π, which is a function taking states as input and mapping it to an action or the probability distribution of the set of actions. Solving an MDP is, in other words, the search for the optimal policy π_*, which maximizes the reward or state value for each action taken. π, a function either discrete or stochastic, where the sum of all actions adds to 1, is in equation (2.1) defined as follows [34]:

$$\pi(a|s) = P[A_t = a|S_t = s], \text{ for all } s \in S, a \in A \tag{2.1}$$

RETURN: Equation (2.2) measures the total reward of a given sequence [34]:

$$G_t = R_t + \delta R_{t+1} + \delta^2 R_{t+2} + ... \tag{2.2}$$

It accumulates the reward given a start reward and all following rewards. The discount factor δ allows emphasizing immediate rewards or incorporating future rewards. The larger the δ, the more it factors future rewards into the sum G_t. A value of 0 considers only the immediate reward. It also prevents issues with infinite sequences, meaning tasks without a specified end.

STATE VALUE FUNCTION: The return value of a sequence alone is not meaningful enough to improve the policy π. Since it only represents the value that happened in this particular observation and not the value of a state as a whole. To help an algorithm find the best policy, the concept includes the state value function $V_\pi(s)$ defined as [34]:

$$V_\pi(s) = \mathbb{E}[G_t|S_t = s], \text{ for all } s \in S, \tag{2.3}$$

Equation (2.3) defines the expected return of an state s when incoperating all possible return values G_t when following π. Solved is this function with iterative methods that converge to the optimum.

Equation (2.4) forms the Bellman equation for the state value function, which takes only the immediate reward and discount value [34]:

Notice: The Bellman equation for a value function helps define the update operation for value function in algorithms and is the base for many reinforcement learning algorithms [34].

$$V_\pi(s) = \sum_{a\in A} \pi(a|s) \left[R(s,a) + \delta \sum_{s'\in S} P(s'|s,a) V_\pi(s') \right], \text{ for all } s \in S, \tag{2.4}$$

STATE-ACTION VALUE FUNCTION: This function is similar to $V_\pi(s)$, but it includes an action for the expected return of a policy π. The equation (2.5) provides the definition [34]:

$$Q_\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a], \text{ for all } s \in S, a \in A \tag{2.5}$$

Making this a higher resolution version of V_π. This function also has the excellent property that the state value function is the same as all state-action values for a state weighted by the probability of taking such action in a state.

Equation (2.6) defines the Bellman equation for the state-action value function [34]:

$$Q_\pi(s,a) = R(s,a) + \delta \sum_{s'\in S} P(s'|s,a) V_\pi(s'), \text{ for all } s \in S, a \in A \tag{2.6}$$

MDPs have their optimal value functions $Q_*(s,a)$ and $V_*(s)$ linked to the optimal policy π_*. Theoretically, there might be multiple optimal policies. With the optimal state-action value function, the MDP can be simply solved by defining π_* as [34]:

$$\pi_*(a|s) = \begin{cases} 1, & \text{if } a = \arg\max_{a\in A} Q_*(s,a) \\ 0, & \text{otherwise} \end{cases} \tag{2.7}$$

Approaches to solving the MDP include dynamic programming and Monte Carlo algorithms, which iteratively improve the value functions to generate π_*. Those algorithms are also commonly used in RL algorithms.

2.1.3 *Reward Function*

Section 2.1.2 introduced the term reward function as a simple mapping from a state-action tuple to a scalar reward value. However, there is a difference in the way a reward function responds. For example, the reward function can provide more or less information at each step of the action taken. Two different reward concepts can define the amount of information and are commonly mentioned in literature [22, 57]:

SPARSE REWARD: As the name suggests, a sparse reward function has its information sparsely distributed. One extreme example is when the reward function only grants a reward upon reaching the goal. This example means that only one state can reinforce the behavior of an agent, which leads to numerous training steps [61]. When implementing a sparse reward function, this function can also rely on a specific condition, where meeting these conditions leads to a specific reward upon transitioning to the next state [22]. However, only some states give reward information, while most do not.

DENSE REWARD: Contrary to sparse reward, a dense reward function (also called intermediate rewards [57] or shaped reward [36]) includes more information and emits different rewards frequently [22]. Unlike sparse rewards, this will make the learning process significantly easier [57]. For example, imagine an environment where the goal is to move in a specific direction. With this reward function, the learning agent would gather a positive reward at each step. The reward function could incorporate a movement vector to accomplish such a dense reward function.

2.1.4 *Reinforcement Learning*

RL is briefly introduced here, with a few key terms essential for understanding IRL in the following sections. RL is a subcategory of machine learning with the addition of an agent that can navigate in a defined environment freely [68]. Given a reward function that gets maximized, it answers which action is the best for an environment. The rules of such an environment might be unknown to the learning agents. In other words, a learned policy π gets gradually improved over sequences of action taken by the agent in the trial-and-error learning procedure. Agent and environment constantly exchange actions from the agent and, in addition, observation and reward from the environment until the learned policy converges to π_*. In this context, MDP (as seen in section 2.1.2) models this reinforcement learning problem of training π. However, as already said, a learning algorithm might not have access to variables of the MDP (e.g., P), so the agent

has to learn through trial and error to learn about the MDP to solve it [57].

The following explanations outline some critical differences in the properties of reinforcement learning algorithms:

ONLINE VS. OFFLINE: According to Winder [68], an online algorithm directly accesses the environment and updates its policy using data from observations returned by the environment, discarding the data once the algorithm utilized it. In contrast, as Winder noted, offline algorithms learn from logged interactions, preventing real-time environment computation. However, the available data limits the algorithm's exploration capabilities, as it cannot interact with the environment in real-time.

ON-POLICY VS. OFF-POLICY: Defines how an algorithm updates their policy during training. An on-policy algorithm will constantly improve one policy, while off-policy algorithms have two policies [34]. One keeps track of the output of the environment and updates their policy according to good actions, while simultaneously, the other generates the trajectory, which represents the output of the environment [34].

MODEL-FREE VS. MODEL-BASED: Russell et al. [57] note that an algorithm is model-free if the agent does not know about the transition function P and will not learn an estimation of it. Instead, for example, it leverages interaction with the environment to develop a value function for learning. This is in contrast to model-based algorithms. They define it as algorithms that use a given P, or even learn P during runtime, for decision-making.

VALUE-BASED VS. POLICY-BASED: Algorithms vary in their use of value functions. Some might even do training without a value function, which introduces the policy-based algorithms. This approach optimizes a policy represented by parameters directly in training without a value function [34]. On the other hand, value-based means using such a value function to learn a policy [34].

The RL algorithm used by most of the IL algorithms in the later chapters uses proximal policy optimization (PPO) as the standard to solve the forward problem. PPO itself is a model-free, on-policy, online algorithm that optimizes its policy policy-based (from actor-critic-strategy). Schulman et al. [58] proposed this learning algorithm. It tries to optimize the policy each step as much as possible but with limits to not cause any harm to the already achieved performance. They presented two methods to optimize the policy: one with a clipping

function and another that introduces a penalty for Kullback-Leibler divergence.

2.1.5 *Trajectory*

A trajectory (also called demonstration) in the context of RL and IL in general describes the history of state-action pairs taken by either a policy π or an expert. Of course, in this thesis context, a dedicated expert creates these trajectories. Taking the definition from Arora et al. [5], the set of trajectories is D with trajectories defined as $< (s_0, a_0), ..., (s_j, a_j) >$ and denoted as τ.

2.2 LIMITATIONS IN REINFORCEMENT LEARNING

Often, when discussing RL and consequently IL, the focus is on highlighting successes while pushing problems into the background. For that reason, it is essential to know about the limitations of this research field in order to have a realistic approach in this area. Consequently, Irpan [36] as well as Hu [34] criticized some general approaches or views on RL in his book. This thesis will also address their concerns in the following sections and mention where IL could offer valuable support.

2.2.1 *Time Complexity*

The time complexity of RL relies on the algorithm's sample efficiency and the training environment's performance. A physical training environment typically faces limitations due to the real-time operation of robotic arms. Similarly, the computer hardware powering such an algorithm limits a physics engine in virtual space. An environment like UE is not developed with machine learning in mind, even though they now offer some plug-ins for machine learning [9]. Resulting in the number of samples outputted limited by the number of frames per second during training.

So, in the end, reducing the number of needed samples (increasing the sample efficiency) for the training of RL is an essential step for reducing the time complexity. However, RL needs many samples to achieve the desired results, especially in environments with large state spaces [34]. For example, Hessel et al. [32] compare different DQN architectures and a combination with impressive results. However, in the end, the number of samples needed for human-level performance exceeds 10 million.

Hu [34] mentions some techniques, such as off-policy learning or function approximation, that can reduce the number of samples needed for policy learning.

2.2.2 *Reward Function*

Using a reward function is essential in RL, whether self-defined or provided by the training environment, to generate π. Towers et al. [64] already offer with Gymnasium a standardized environment for research to compare algorithms under the same conditions. Their package offers environments and reward functions with engines such as MuJoCo, Box2D, and more. However, what if one wants to use a self-defined training environment with its own reward function? This requires reward engineering to create the proper function, which helps the agent learn the correct policy. Irpan [36] names a few examples of how difficult it can be to write such a reward function and the agent learns an entirely wrong behavior. This problem is where IL might come in handy since it does not require the definition of a reward function; more on that topic in section 2.3.

2.2.3 *Local Optima and Generalization*

Another problem arising from RL is the exploration-exploitation dilemma. It is essential to balance these two phases, as taking too little time in the exploration phase can lead to learning nothing, while starting the exploitation phase too early can reinforce incorrect behaviors in the policy, thus leading to a local optimum [36].

Generalization is another problem that might come with agents needing to interact with more situations. Those agents have not learned enough to react to new situations from past learned scenarios [34].

2.3 INTRODUCTION TO IMITATION LEARNING

In RL, a significant challenge is the necessity of defining a reward function, where a user might have difficulties (as discussed in section 2.2.2). IL offers a potential solution to this issue by concentrating on replicating an expert's behavior. It does this through demonstrations instead of a manually defined reward functions, allowing us to generate a policy or derive the needed reward function. Multiple surveys [5, 69, 71] reflect the situation on IL or IRL specifically. The purpose of IL is to extract knowledge from an expert (human or artificial agent) based on demonstrations, allowing for the modeling of behavior in the same or a similar environment to that of the demonstrations [71]. IL comprises two categories, namely IRL and BC. Zheng et al. [71] and Zare et al. [69] define both BC and IRL, but they also add a third category, an adversarial approach. The next section will briefly overview each category, highlighting their features and including relevant algorithms used in this thesis.

Notice: *Even tho Imitation from Observation (IfO) is another form of IL, this thesis will neglect IfO due to its focus on learning with raw video material rather than state-action sequences as input [69, 71].*

2.3.1 *Behavioral Cloning*

Behavioral cloning (BC) (also just called Behavior Cloning) is the simplest IL approach that maps a state $s \in S$ to an action $a \in A$, thus creating a function $a = \pi(s)$ [67]. Wang et al. [67] further explain that the creation of π solely takes (offline) expert's trajectories instead of interacting with an environment like AIL or IRL. Thus making this approach a supervised learning algorithm. Conversely, this also means the performance is significantly faster than the other IL approaches. This mapping function, also known as policy, uses a negative log-likelihood loss function to induce π [71].

However, there are two main problems with this approach. First, the policy generated by BC is sensitive to changes; moreover, inducing this function has a low success rate below 50% [74]. Of course, there are solutions to some problems in the field of BC, like model-based BC, to help with different environment dynamics, but these come at the cost of greater time complexity [71].

An example of an algorithm that falls under the BC category is known as *"DAGGER"* [56].

2.3.2 *Inverse Reinforcement Learning*

As discussed, BC is the simplest way to imitate the behavior by using recorded data to induce the policy. IRL, on the other hand, tries to solve IL by finding the reward function for RL iterativly instead of learning a policy directly [71]. In classical RL, the reward function is seen as given (see section 2.1.2), but what if there is no reward function or it is too difficult to define? For example, defining the reward for autonomous cars can be difficult due to rules, the behavior of other cars, the weather, and more. This approach instead tries to infer a reward function from the demonstrations by an expert. Which, in hindsight, is the key for RL to imitate the behavior. The benefit of IRL contrary to BC is that you can use the reward function under different conditions (e.g., less or more available actions) and reproduce a similar behavior, which is better than just copying the actions [5]. Zheng et al. [71] point out that BC can perform better and take less time than IRL when there are abundant trajectories and an accurate controller. However, depending on the state space, this might require an unrealistic amount of demonstrations.

2.3.3 *Definition*

Aurora et al. [5] formally define IRL as the inverted RL problem. Assume a MDP without a reward function R_E dependent on an expert E. D as defined in section 2.1.5 is the set of all recorded trajectories.

They then present that the reward function $\hat{R}_E$ should be determined based on either the expert policy or the collected demonstration set.

Listing 2.1: The IRL algorithm as a template modeled after Aora et al. [5].

```
input: D
ω = randomly initialized reward parameters
while learned and expert behavior differ significantly:
        R̂_E = initialize R̂_E using ω
        π = learn policy with current R̂_E under ω
        ω = update ω by reducing the difference between expert's
            behavior and learned policy
output: R̂_E
```

The goal is to learn the reward parameters ω that define the reward function. In typical algorithms, a weighted combination of features approximates the underlying reward function [69]. For example, features could be the visitation count of states under the experts demonstrations. A general IRL algorithm based on Arora et al. [5] follows four steps until a stopping criterion is met (see listing 2.1). The previous definitions provide a set of trajectories D as input and give a reward function $\hat{R}_E$ as output. The first step will initialize the reward function with the initial values or, if already learned, the new features. Next, the algorithm will do RL, also called solving the forward problem, with the current reward function, then update the parameters ω to minimize the difference between the expert's and learned behavior. Lastly, this process continues until the algorithms meet a stopping criterion.

2.3.3.1 *Problems*

This section specifies only problems relevant to a virtual environment. For example, the problem of security concerns with IRL, which control agents in the real world (e.g., a car), is left out. While Aurora et al. [5] pointed out a few more challenges to overcome, like the accuracy of prior knowledge and missing states in observations. The focus will lie on some of the mentioned problems by Aurora et al. [5] and Zare et al. [69]. They define these as:

AMBIGUITY: There are several possible reward functions for the demonstrations generated by the expert. Since it is not feasible to include all possible trajectories in the smaller, finite set the algorithm uses as input, the algorithm must work within these constraints. This results in multiple reward functions, each capable of creating policies that can match the input demonstrations.

COMPLEXITY: IRL needs to solve an MDP in multiple iterations to gain the reward function. While this has polynomial complexity, one key element, the state space, suffers from the curse of dimensionality.

With each new dimension, the state space grows exponentially. This leads to poorer sample efficiency. Furthermore, with the increasing state size, the number of demonstrations also needs to grow (also called sample complexity). Otherwise, not enough coverage of space might lead to sparse rewards.

GENERALIZABILITY: The problem of extrapolating unknown states and actions from the observation. This means that the agent can also cope with unknown environments. The challenge is inferring a reward function and using as little data as possible without creating a too-large approximation error.

2.3.3.2 *Algorithms*

While the classical approach involves linear reward functions, more modern approaches utilize neural networks for non-linear reward functions [5, 69]. However, the goal remains: to learn the underlying experts reward function. And thus, the optimal policy. When helping to counter the ambiguity problem, Aurora et al. [5] outlined four categories of IRL. In contrast, Zare et al. [69] roughly divide them into three:

MAXIMUM MARGIN: A method that should help counter the ambiguity problem. Maximum margin aims to derive a reward function that represents the optimal policy more accurately than other competing policies, achieving this by a defined margin.

MAXIMUM ENTROPY: Another approach to solving the ambiguity problems, but without the bias introduced by the maximum margin method in the reward function. As the name suggests, this method maximizes the entropy, assuming that the reward function with the highest entropy has the least commitments.

BAYESIAN LEARNING: The pairs of states and actions in a trajectory are used as evidence to facilitate a Bayesian update of a prior distribution over potential reward functions.

Various categories exist for solving IRL, and while those are important, a more crucial question for this thesis is whether an algorithm is compatible with continuous space. As an explanation for the need for continuous space, think of robotics. In robotics, a robot arm is controlled by continuously adjusting the joints. However, the environment given by the algorithm in which the arm operates is often discrete. If we try to convert the continuous arm movement into discrete steps, we will introduce inaccuracies in the control. Therefore, we need algorithms that can operate in continuous space, which is often high-dimensional. Due to the goal to learn affordances accurately,

the need for these algorithms arises. Thus, the following paragraph presents commonly used algorithms in continuous space and one in discrete space.

Since many discrete algorithms exist, this thesis will only present the one used in the implementation. When looking at maximum entropy methods, maximum entropy IRL (MaxEntIRL) by Ziebart et al. [73] is most commonly associated with that category. In itself, it is a convex non-linear optimization. However, MaxEntIRL, like many algorithms in IRL, only works in discrete state-action spaces, so it does not fit the use case of affordances in a continuous space. Therefore, the following paragraphs mention better-suited approaches, although most solutions focused on solving IRL in discrete space [5]. These approaches mainly expand the maximum entropy (deep) IRL algorithm:

CONTINUOUS MAXIMUM ENTROPY DEEP INVERSE REINFORCEMENT LEARNING: Chen et al. [13] developed a new algorithm for continuous action and state space. They also included a "hot start" mechanism to improve the algorithm's convergence further.

GUIDED COST LEARNING: Finn et al. [23] introduced a sample-based approximation of the MaxEntIRL algorithm to learn the reward function for a continuous MDP.

MAXIMUM ENTROPY INVERSE REINFORCEMENT LEARNING WITH PATH INTEGRALS: Another approach that lets an IRL algorithm work in a continuous space proposed by Aghasadeghi et al. [2] uses Path Integrals. To reduce the complexity of learning RL during IRL training, such methods aim for local optimality of trajectories [5].

2.3.4 *Adversarial Approach*

AIL is another approach that addresses the problem of IL. While both IRL and AIL are pretty similar in the way they work, AIL uses generative adversarial networks (GANs). As Ho et al. [33] pointed out in their implementation of an adversarial approach, normal IRL is computationally complex due to the requirement of running RL in the inner loop. With their implementation, they achieved an algorithm that was pretty sample-efficient and related to the expert's input. Furthermore, later introduced in this chapter, common algorithms are also continuous for states and actions, making them a perfect fit for implementing this thesis [24, 33].

2.3.4.1 *Definition of Generative Adversarial Networks (GANs)*

Goodfellow et al. [29] introduced GANs as a framework of two adversarial networks playing a min-max game with a value function defined in equation (2.8):

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

(2.8)

A generative model, G, tries to generate an output that a discriminative model D would see as actual data samples. Contrary to that is D, which tries to discriminate between the fake output generated by G and the actual input data samples. A trained model, D, would give a value close to 1 to actual data and a value close to 0 to generated data. So equation (2.8) uses the expected return based on the actual data x of the discriminator and adds this with the expected return of the evaluated output of the generator based on a random input noise z. Notice that the discriminated value of G gets subtracted by 1 because otherwise, the discriminator would always output one to maximize D. This kind of adversarial play improves both models until the discriminator can not distinguish between both sample sets.

2.3.4.2 *Algorithms*

Two prominent algorithms stand out to mimic expert behavior through structured adversarial interactions. Each algorithm differs in its approach to solving the IL problem, contributing to advancements in this area of research:

GAIL: A well-known IL approach that utilizes GANs as their base to imitate expert's behavior is GAIL. Ho et al. [33] question the need for an intermediate step to infer a reward function and thus introduced this concept of a model-free algorithm that directly learns the policy. They argue that calculating the reward function adds to unnecessary computational complexity. Due to its structure, it can also run in large continuous spaces. The algorithm uses the standard binary neural net classifier as a discriminator and updates their policy with TRPO (PPO is used instead of TRPO in the Imitation package). Like in the standard GAN structure, the algorithm alternates between the training of the binary classifier D and G.

AIRL: The adversarial inverse reinforcement learning (AIRL) approach introduced by Fu et al. [24] is the improvement of already existing IRL algorithms based on the adversarial approach to infer the reward function. Contrary to GAIL, the discriminator now gives a probability instead of only binary values. Furthermore, it will be optimized with the parameters of the reward function, thus giving a reward function for generator training. Another advantage besides the existing training in continuous space is that generated reward functions are robust to changes in the environment dynamic. So instead of classic IRL or GAIL, which fail with high variability environments, AIRL proves to generalize well.

3

RELATED WORK

Chapter 2, presented a brief overview of the field of IL, along with some standard algorithms. Since this thesis examines two categories of IL, reviewing projects related to affordances, VR, and game engines is essential. Doing so will help illustrate how these areas contribute to the broader landscape of IL and provide context for the following work.

In terms of classical IRL, Lindner et al. [43] developed an algorithm that does not need the dynamics of the environment to be known or access to a generative model. The algorithm actively explores the environment and policy of the expert to gather the reward function. However, their approach is, at the moment, only compatible with discrete space.

Djeumou et al. [14] introduced an approach that questions the observation of many IRL algorithms in complete environments. To tackle this, they utilized partially observable MDPs and improved sample efficiency by incorporating side information into the learning algorithm. They achieved behavior similar to that of an expert. They even demonstrated final results in a continuous Unity 3D environment. However, a challenge with this approach is its reliance on an environmental model.

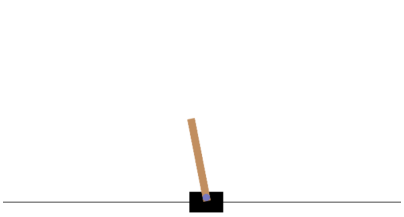
Zhu et al. [72] introduced a combination of RL and IL to develop a learning algorithm that performs better than both algorithms alone. Their primary motivation is to help learning with multi-stage manipulation tasks and to reduce the number of samples needed while learning. To combine individual algorithms, they introduced a task reward. Combined with the normal reward function gained through an IL algorithm, this leads to learning acceleration and increased stability. Since their training phase was in a physics simulation, they could also leverage form from the state information of the simulation to improve learning further. An imitation approach using Path Integrals introduced by Kalakrishnan et al. [40] also has an algorithm for feature selection for the final IRL algorithm. It samples the right features from a list typically for robot arms. They evaluated their approach with inverse kinematics and optimal motion planning.

Since the global reward function learned by IL can produce suboptimal results due to noise and errors in the data collection, Liu et al. [44] introduced an algorithm known as CSIRL. Their approach divides a task dynamically into several local reward functions, leading to improved outcomes compared to earlier methods.

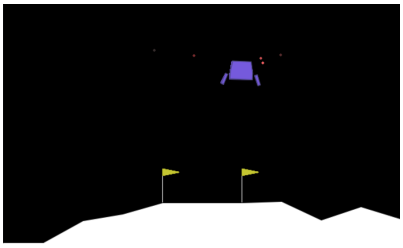
One concept explicitly using affordances in the implementation is from Lopes et al. [45], which uses an affordance model. They relate known high-level actions, such as grasping and tapping, to their effects and present a dynamics model for learning. The agent then employs these effects to allow a Bayesian IRL algorithm to learn by imitating the behavior observed in a demonstration. This approach enables the agent to learn complex tasks.

It is essential to highlight approaches incorporating VR in behavior learning projects. Zhang et al. [70] identified issues with the traditional methods of generating demonstrations, often requiring costly teleoperation or using external force on the robot. To address this, they suggested using a straightforward VR setup with a controller to collect data. They then used the collected data to evaluate the effectiveness of IL through a basic BC algorithm.

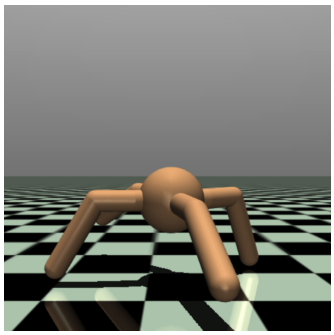
Dyrstad et al. [18] introduced a different approach to transferring virtually generated data into the real world. They aimed to use recordings from VR, employing domain randomization, to create a synthetic training set for imitation learning in a 3D Convolutional Neural Network. Their method enabled a robot to learn how to grasp fish, achieving a success rate of 74%. Similarly, Hu et al. [35] developed a two-stage IRL system to train a robot to grasp living objects with a success rate of 90%.



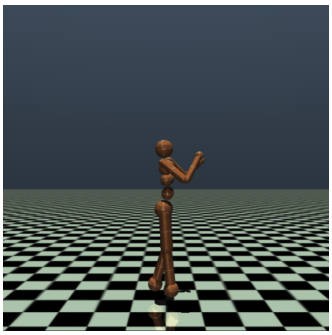
(a) Classical Control: Cart Pole [16]



(b) Box2D: Lunar Lander [15]



(c) MuJoCo: Ant [17]



(d) MuJoCo: Humanoid [17]

Figure 3.1: Standard Gymnasium environments used to compare various learning algorithms [64].

Regarding frameworks that provide standardized environments for RL, they facilitate benchmarking and allow for more effective comparisons of improvements across various algorithms in this field. A prominent example is OpenAI Gym [10] along with its successor, Gymnasium [64]. These frameworks have become foundational tools for comparison in numerous scientific papers [13, 28, 33, 58]. Both platforms leverage various physics engines, enabling the testing of diverse properties such as 2D and 3D simulations and creating custom environments. As illustrated in figure 3.1, different standard environments are available within Gymnasium, like MuJoCo and Box2D [64]. Additionally, a variant of OpenAI Gym has been explicitly developed for multi-agent RL [62]. Another framework, Orbit, is based on Isaac Sim and offers a variety of features, including benchmark tasks [46].

Nevertheless, these are just tools for a standardized comparison between algorithms, but software implementations also already offer some UE and IRL elements. Namely, MindMaker and Learning Agents plug-in. Learning Agents is a machine learning plug-in for agents used in the Unreal Engine environment to train via RL or IL algorithms [9]. However, this implementation only offers simple BC capabilities. MindMaker on the other side is also a UE plug-in that uses a UE environment as a Gymnasium environment and offers some experimental implementation of Imitation [41].

Since development started with typical discrete IRL learning algorithms, which both existing software extensions did not support, they found no use in the initial plan. Therefore, this thesis planned a new self-implemented plug-in. Nonetheless, at the time of writing the thesis, Learning Agents only supported vanilla BC and PPO as machine learning algorithms, which means that only the simplest IL algorithm is supported. On the other hand, MindMaker has experimental support for Imitation but is not in development anymore. This means the already self-implemented plug-in would be extended to a continuous space and used instead.

Part II

METHODOLOGY

The system architecture, algorithm development, software implementation.

4 APPROACH

This chapter lists the basic ideas for implementing the IL process and the UE extension, as well as the techniques that should lead to an improvement. Furthermore, it provides a list of package details and explains the preference for specific communication methods. Initially, the plan was to use discrete IRL algorithms, but the focus shifted due to their limitations in continuous space. Although the intention was for the software to run exclusively on Windows, software constraints require execution of VR on Windows and conducting training on Linux. More information on that is in the following chapter.

4.1 PROBLEM DEFINITION

Due to the difficulties with defining reward functions for RL, the goal is to instead use IL algorithms in a virtual environment. Everything from recording demonstrations to training should occur in this environment with the continuous algorithms GAIL and AIRL due to their advantages over other algorithms. However, since IL algorithms have problems generalizing well, this work also focuses on adding auxiliary help functions to the standard algorithms to improve this. Furthermore, choosing the correct hyperparameters is crucial for a project's success, which is why this approach includes an automated method to determine the correct values.

4.2 AFFORDANCES

The selection of affordances for testing the implementation should align with everyday tasks found in a kitchen environment. In training, each agent will execute an affordance task represented as an individual environment, including a hand and a few objects in the UE with a 60 fps simulation speed. An agent has direct control over a 7-Dof hand. Such a hand in a kitchen environment offers various tasks, such as having an agent use a knife to cut an object or stir content in a pot. However, the chosen task must be simple enough for implementation to minimize glitches from the physics engine and keep the cost of creating the environment manageable compared to the overall system. To later test the capability of task rewards, the task needs easily differentiable stages such as picking up, standing still, and more. These requirements lead to objects offering affordances in the following three environments:

COVERING: A task where the agent uses an object to cover the container. Image boiling water and wanting to cover the pot to accelerate the procedure. In this case, an agent needs a lid to cover the pot. Moreover, the hand will move towards the lid to pick it up and place it on the pot.

INSERT: Packing objects into a container is also something that often needs to be done in a kitchen. Such a scene would require two objects, in this case, a pot and a spatula. The hand is then supposed to put the spatula into the pot to finish this task. This task is similar to covering but adds the object rotation requirement into the task.

STACKING: An agent is supposed to stack objects. However, after some trial and error with the physics engine, it was found that the initially planned plates proved to be too unstable, which is why thicker objects such as blocks had to be used in this task. In the final design, a hand will stack three cubes to form a tower.

Other tasks in the initial planning phase included flipping an object like an ingredient in a pan with a spatula. Again, this could have led to physical instability due to these objects' inherently flat collision boxes. Another task that would have been interesting, for example, is pouring liquid. Again, physics constraints could have been problematic, even more so for recording the whole state space. In the end, to keep the task reasonably realistic, quite a few balls would have been needed to simulate a liquid adding more objects to the state space that behave chaotically and would lead to the agent not learning the right reward function or policy.

4.3 UNREAL ENGINE AND USEMLOG

The choice for a physics engine fell on UE due to existing knowledge of UE in the CGVR research lab and other projects related to this work, such as RobCoG [54], whose objects and scenes offer an entry to the environment and USemLog [55] to capture the trajectories. It logs the position and rotation of previously marked objects in a selected database. External software can then write an entry for further processing to a JSON file. Furthermore, several projects also utilize UE for RL [9, 41], leading to the decision to favor this game engine as the simulation and recording environment. However, a question remains: What recording technique should be used? Two different systems were available for use: first, OptiTrack [49] and second, VR. OptiTrack itself is a solution that tracks objects in the real world. However, using such a technique would require having objects used in the training simulation to be available in the real world and setting up a second utterly independent environment to resemble the virtual environment.

Hence, it would take time to use such a technique. Furthermore, objects need to have some markers for the recording, which can lead to only specific poses of the object being useful since a marker could hinder other movement. So the decision fell quickly to ordinary VR with a head-mounted display and a controller to simulate input from the hand.

4.4 DISCRETE IDEA

It is worth mentioning that implementing a discrete algorithm was also considered for the entire project; however, this approach proved too imprecise for a robotic environment that operates with continuous values and is significantly affected by the curse of dimensionality. A brief implementation using MaxEntIRL, featuring communication between the learning algorithm and the training environment with only some discrete actions, led to this conclusion. As a result, while the next chapter also discusses the implementation in a discrete format, it will not be adequately evaluated.

4.5 TASK REWARD

As mentioned in chapter 3, one idea, introduced by Zhu et al. [72], to counter the effect of terrible convergence to multi-stage tasks and to reduce the number of needed samples to generalize, is the usage of task reward. Such a task reward is just a sparse reward defining critical steps of the task. It aims to guide an imitation learning algorithm, in this case GAIL, in the right direction to accelerate training and to help overcome local maxima. The usage of both task reward and reward function learned through IL is named hybrid reward defined in equation (4.1) [72]:

$$R(S_t, A_t) = \lambda R_{IL}(S_t, A_t) + (1 - \lambda) R_{TASK}(S_t, A_t), \quad \lambda \in [0, 1] \tag{4.1}$$

Equation (4.1) adds in addition to the two reward functions a variable λ, which is a factor to balance the influence of both functions to the final result. So if λ is 1, only the learned reward function influences the final result, while a lower value also incorporates the task reward.

The pseudo-code in listing 4.1 below shows an example of a procedure of a task reward, which the implementation will, in contrast to previous work, also use with AIRL. First of all, the input slightly varies from the input in the equation (4.1). This means the algorithm only needs the current and following states for calculation. Said states enable the calculation of relevant information like movement toward objects and distance between objects. That qualifies if-statements to check if said values are within a specific range to set the reward function into returning a specific reward. There are four main stages: hand movement towards the lid, holding the lid, moving it towards the pot,

and finally, placing the object on top. This sparse reward should guide the learning algorithm in the right direction.

Listing 4.1: The pseudo-code of the task reward for covering.

```
input: state, next_state # in other words, the initial state and
    subsequent state
movement_hand_to_lid, movement_lid_to_pot, next_lid_to_pot,
    next_lid_to_hand = initialize these values using state and
    next_state. All these variables are scalar values according to
    the length of the calculated vector.
# Zero reward as a baseline
reward = initialize reward with 0.0
# Stage 1: Movement of the hand towards the lid
if movement_hand_to_lid exceeds a certain value:
        reward = 0.5
# Stage 2: Hand holds lid
if: next_lid_to_hand is small enough and hand grasps
        reward = 1.0
# Stage 3: Movement towards pot while holding the lid
if movement_lid_to_pot exceeds a certain value while holding in
    next_state:
        reward = 1.5
# Stage 4: The object is on the pot and not grabbed
if next_lid_to_pot is small enough:
        reward = 2.5
if next_state is still grabbing while next_lid_to_pot:
        reward -= 0.5
output: reward
```

Since the learning algorithm requires separate task reward for each affordance and listing these would go beyond the scope of this section, appendix A.1 list all other task rewards for insert and stacking.

4.6 EXISTING MACHINE LEARNING SOFTWARE

Since there is already some work done in the field of IL, there is also some learning software available. Execution of these works in an external process, which will need an independent way to communicate to the learning environment implemented in UE:

- **Imitation:** A collection of different IL algorithms built on Stable Baselines 3. Amongst GAIL, AIRL also contains BC as Pytorch implementations [28].
- **Gymnasium:** A standard interface implementation for RL environments [64].
- **irl-maxent:** An implementation for maximum entropy IRL [51].
- **Stable-Baselines3:** Package containing a multitude of RL algorithms using PyTorch [52].

4.7 OPTUNA

In their research, Eimer et al. [19] highlighted the importance of hyperparameter selection. They concluded that the wrong selection of hyperparameters can lead to the failure of the whole algorithm, even when they initially seem insignificant. This problem is also why this thesis uses a hyperparameter optimization tool called Optuna. Optuna allows for easy tuning parameter selection and the definition of a sampling set for the tuning parameter. This means a user can, e.g., define lists with predefined values for a parameter and even ranges from which to sample. Since this algorithm does not just use sweeps or grid search but instead uses techniques like early pruning or efficient search space exploration, it decreases the time needed. While running, the framework tunes all specified hyperparameters provided. However, things such as optimizing the network structure of the PPO algorithm used to train GAIL and AIRL are left out, as this would increase the dimensionality of the parameter size and further worsen the training time. In the final implementation, to optimize the set of hyperparameters, the output of task reward defines the metric to optimize.

4.8 INTER-PROCESS COMMUNICATION

Before writing about the intended way of implementing the extensions for UE and Python, the following paragraphs discuss the pros and cons of different kinds of communication between processes since there are multiple ways to enable communication. In the literature are four common concepts for communication [8]:

SOCKETS: Sockets allow communication over the network with a transport layer protocol independent of the operating system. This approach includes two different protocols for usage, named user datagram protocol (UDP) or transmission control protocol (TCP). While TCP is more reliable, sending messages in the correct order, UDP has a more negligible overhead. This approach needs identification in the form of an IP address and a port number.

PIPES: Special one-directional communication channels allow one process to write while another reads. These channels, known as anonymous pipes, require closely related processes to create. Once all related processes terminate, this channel closes. Another type, named pipes, facilitates communication between processes that are not closely related. Named pipes enable bidirectional communication and remain active even when no processes use them.

MESSAGE QUEUES: Communication uses linked lists, which can exceed the process's lifetime. An application can process the information sent via the FIFO principle since the order of messages is chronological. The system does access coordination, and messages get a message type to help differentiate the messages.

SHARED MEMORY: A memory-based communication. Multiple processes can access a shared segment of memory, enabling communication between processes. While currently written data is not accessible, there is still the need to coordinate access operations to counter race conditions. Either a system call or a reboot can remove a shared memory segment.

In the end, due to already existing software like TCP-Unreal and a straightforward package available in Python, the primary focus for the implementation lies on socket communication.

4.9 IMPLEMENTATION DESIGN

The following subsections explain the basic idea of creating the software. Some schematics further illustrate this approach.

4.9.1 *Pseudo-Code*

Listing 4.2: Pseudo-code that describes the procedure of the planned algorithm. For clarity, only the relevant parameters are provided as input.

```
1  input: hyperparameters, hyperparameter_optimization
2  def imitation_learning(hyperparameters):
3                  while not stop_criterion:
4                          π = update the policy with current
                               reward_function
5                          reward_function = use samples from π to
                               update reward
6                          reward_function = modify learned
                               reward_function by including
                               task_reward blend accoriding to
                               balancing_value in hyperparameters
7                  return π
8
9  if hyperparameter_optimization is active:
10         optimizer = optuna
11         π = optimizer.optimize(imitation_learning)
12         optimizer.save_parameters()
13 else:
14         π = imitation_learning(hyperparameters)
15 output: π
```

4.9.2 *Processes*

Two processes must work together to realize the procedure mentioned in section 4.9.1. First of all, the Unreal Engine to simulate the environment and a Python instance to learn from the simulation:

UNREAL ENGINE: The goal for UE is to provide a simulation environment that enables training of the IRL algorithms and facilitates trajectory recording. During training, a predefined number of agents operate in parallel, controlled by the forward part of the learning procedure. Each agent Actor gets a specified client instance connected to the Python server to manage this UE Actor for the Python server. A manager class organizes the learning parameters influencing the environment, allowing direct configuration within the UE editor. Data exchange between the final learning algorithm and the training environment occurs over TCP-Unreal, chosen for simplicity. Recording occurs in the same environment as training. However, instead of the learning algorithm controlling the agent, a controller paired with a head-mounted display (as shown in figure 4.1) serves as the input method. To be more precise, the HP Reverb G2 serves as the input device.

Figure 4.1: Picture showing the head-mounted display and controller (HP Reverb G2).

PYTHON: Since most learning algorithms are already available outside of UE, an external process running in a Python instance will train the algorithms. Using the Python instance as the server for TCP communication makes the most sense because this way, UE can offer all agents an independent client connection. The Imitation package employs Gymnasium environments, providing a standardized interface for interacting with various learning environments. Consequently,

the IL programming requires a Gymnasium environment wrapper for the UE environment to facilitate communication.

4.9.3 *Interaction*

In the following, figure 4.2 and figure 4.3 show the interaction of each process to visualize better the association between UE and Python.

IMITATION LEARNING: Figure 4.2 specifies the information exchange between both processes when using the implementation for IL. Suppose the user wants to start the IL training process. In that case, they should configure the environment with all necessary parameters (like training steps, type of environment, number of parallel agents, etc.) before finally starting the training process. After being executed by the user, the program will independently start a sub-process in a new terminal window, which will run the Python script. After the script starts running, a new *IRLServer* will start. The server will then create new *IRLClientSocket*s, while *UE* instance will try to create its own *UEClients* to connect to the client sockets. Before connecting to the Python implementation, each *UEClient* will create its own *UESimulationEnvironment*, representing all the objects and a server-side controlled hand. After all *UEClients* finally have a connection to the Python instance, the IL process will start executing the *IRLTraining*. This means a modified version of the *ImitationScript* by Imitation starts with all necessary sacred experiment parameters. The scripts then will start multiple *UEGymnasiumWrapper* so that the learning algorithms in Imitation can interact with the environment in UE until reaching a predefined number of maximum steps. This means the *UEGymnasiumWrapper* sends a request to the *IRLClientSocket* implementation, which then starts communicating with the *UEClient*. Each *UEClient* instance gives the information to its assigned environment. After the script succeeds, the learning algorithm will return a policy. The Python script can also return a reward function depending on the algorithm.

HYPERPARAMETER TUNING: The interaction is more or less the same as described in section 4.9.3, with the difference being that *IRLTraining* will start a modified *OptunaScript*. This will lead to multiple imitation learning instances (*ImitationScript*) training on different sets of hyperparameters in parallel. Optuna chooses the policy and, more importantly, the set of hyperparameters according to the best mean reward returned during the procedure. Everything else stays the same; the system relies on one wrapped Unreal Engine environment for all imitation instances to reduce hardware usage during training. The slight change in the procedure visualizes the figure 4.3.

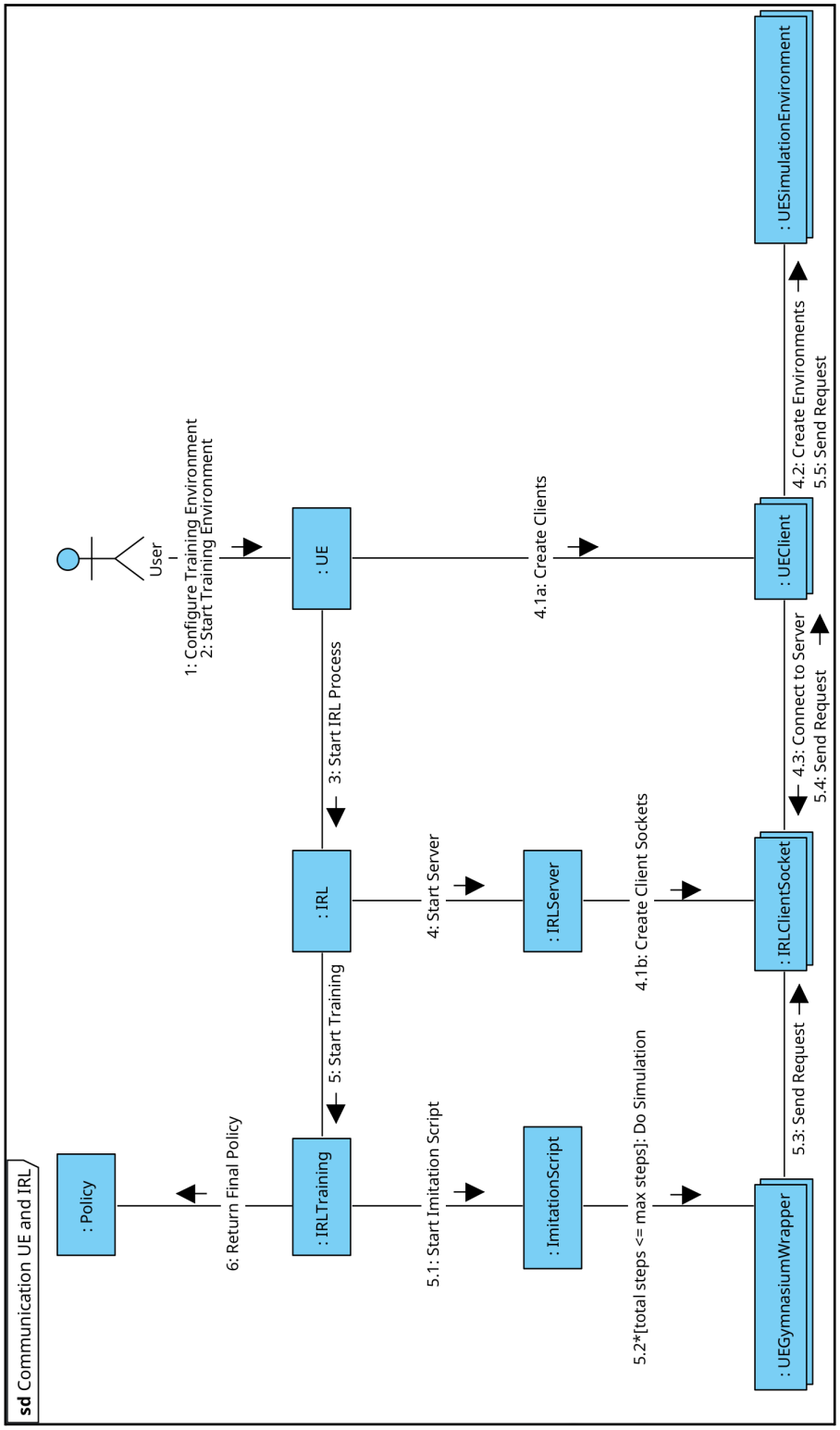


Figure 4.2: Communication between Python and Unreal Engine for IL.

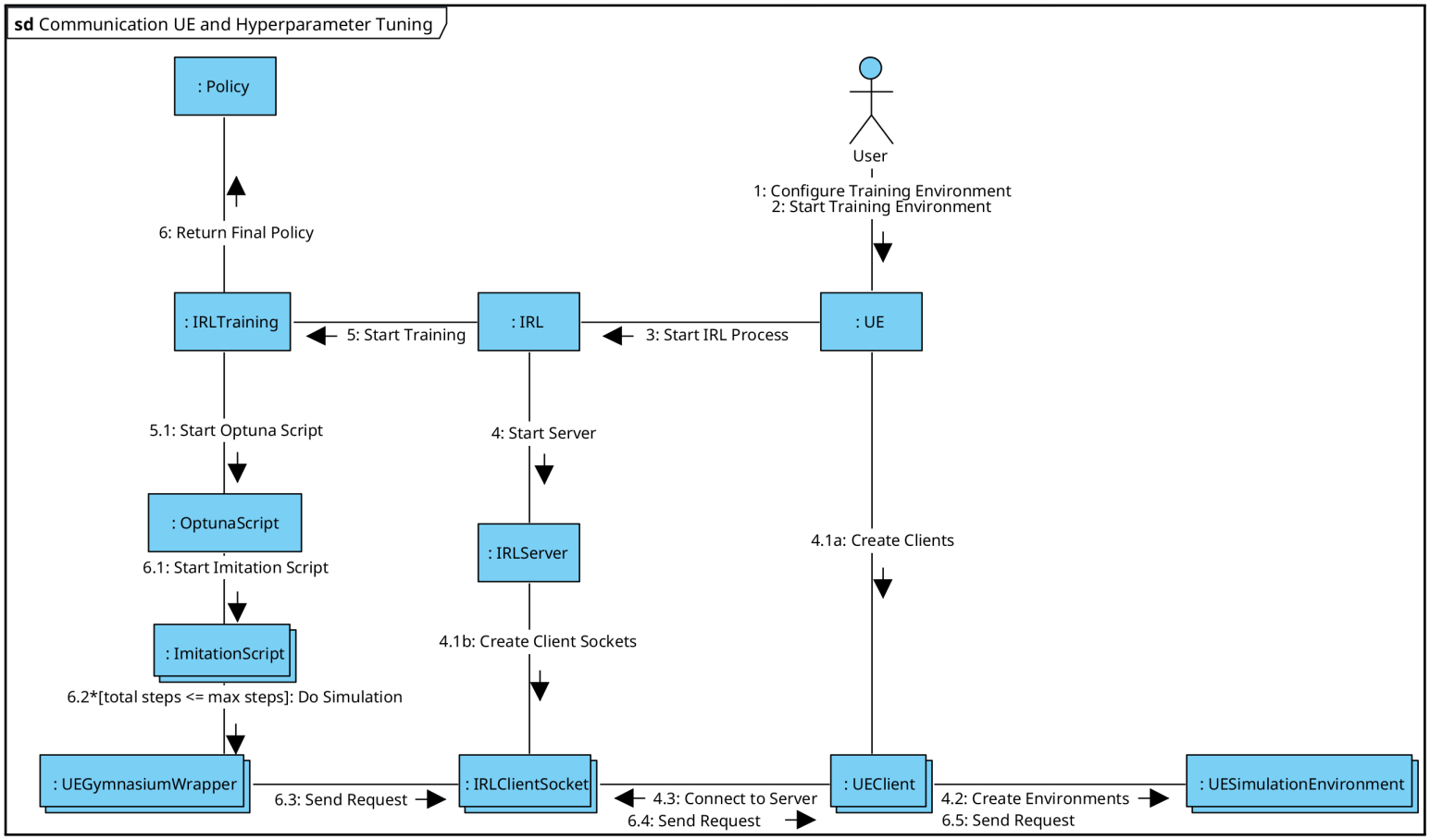


Figure 4.3: Communication between Python and Unreal Engine for hyperparameter tuning.

5 IMPLEMENTATION

This chapter explains the implementation of the fully working environment to evaluate the performance of different IL algorithms, including the auxiliary components for learning affordances. UE 5.3.2 forms the base for the simulation environment, and the available Python packages offer AIL implementations. Communication between both processes realizes the training process.

5.1 UNREAL ENGINE

UE offers the capability of using VR for trajectory recording and the opportunity to upgrade to OptiTrack. The project mainly utilizes C++ code, but some elements also use blueprints. Implementation-wise, UE works as the main instance to control all parameters for training and launches the Python training process. Though communication-wise, UE will act as the client.

5.1.1 *Client Implementation*

As explained in section 4.8, Python and UE communicate over TCP/IP sockets to configure the setup quickly. UE contains client instances to whom the server can communicate. Available as *UClientComponent*, the client component for UE uses the TCP-Unreal package to handle the communication. It also contains all necessary methods, from connection status to sending messages and callbacks, to get the received information from the server and notify other registered objects within UE if a message occurs. A client has precisely one communication stream to the Python server, meaning for each learning instance running, UE will create a new *UClientComponent*.

In early programming, responses to actions could become very large; hence, the implementation also partitions messages into smaller packages and merges them again. Also, messages have a specific end marker that signals the end of a message since there is no option to control low-level protocol information. The system encodes and decodes every message as a JSON object because UE, among other things, offers conversion techniques to structs for more uncomplicated handling. Disconnecting is done with a simple "c" as a signal.

5.1.2 *Executing Python*

The system should automatically start the Python script to enable the learning process whenever the user presses the play button within UE. To make this possible, a collection of functions checking for system-wide installation of Python, a virtual environment containing all packages, and executing the main Python script are available. UE calls all functions at every start. If, for example, the virtual environment containing all packages for Python is missing, it will get automatically generated due to these functions. If the virtual environment exists, a new window with the learning process running appears.

5.1.3 *Action and State Space*

Internally, the action space (*TActionSpace*) is a template class of numerous helpful methods linked to the usage of action spaces. However, more importantly, since it is a template, it is available for both discrete and continuous implementation. The bounds of the action space are statically defined to match the natural movement of an expert, which means the system uses the upper-value bounds captured from the expert trajectories. This hinders a learning algorithm from taking unrealistic steps, which also improves learning by narrowing the possible actions. Another remake is that grasping with a hand is a discrete action. Moreover, since the used algorithms cannot handle discrete and continuous actions simultaneously, this value appears as a continuous action within the action space—a certain threshold within UE then maps it into a binary value.

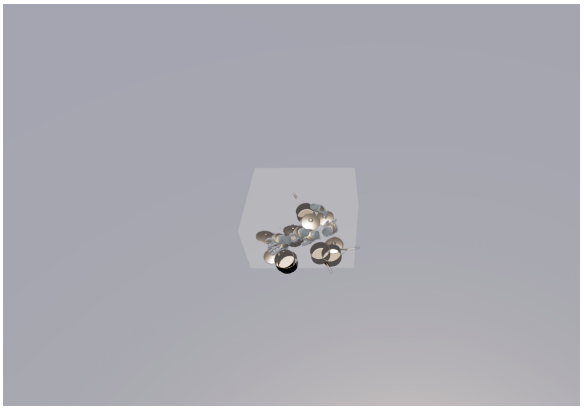
(a) Without visualization (Continuous). (b) With visualization (Discrete).

Figure 5.1: Discrete visualization of the state space using the *AGridActorVisualizer* in Unreal Engine.

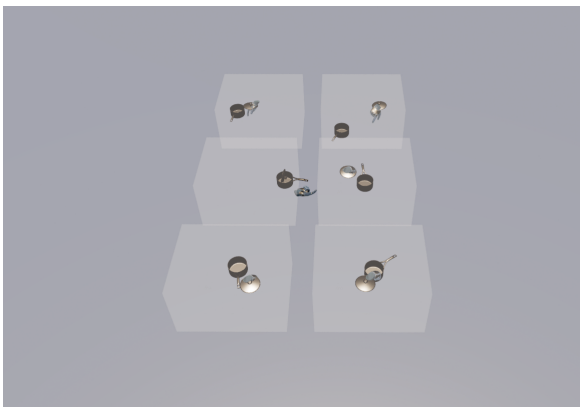
Contrary to the statically defined action space within UE is the state space. Here, the user defines the values of the state space and can visualize it as seen in figure 5.1. It is possible to choose between either discrete or continuous state space. Moreover, defining the state space involves placing and scaling a particular Actor called *AStateSpaceBounds*. Choosing discrete space requires changing the number of states at each dimension. The continuous state space, however, is only

defined by *AStateSpaceBounds* (Actor seen in figure 5.1a). Being the state space, it also offers some functionality for internal checking of correct object placement within the state space, besides pure visualization. However, the definition of the state space bounds is independent of discrete or continuous properties. This is where UE internally will use the abstract class *AStateSpaceBase* to offer functions that fit these properties. To be more precise, the two inheriting classes are for continuous and discrete space. These classes execute resets and steps on a specified set of objects within the space. The UE physics engine bound the simulation speed within the state space to its tick rate. Which in this case is 60 fps. This means UE will execute all actions of the Python script within UE's *Tick()* method to modify physics.

Since it is for standard training not necessary to have a good visualization, all clients learning a policy share the same position. However, this setup confuses and is not visually pleasing for the visual evaluation of the results. Therefore, the UE project offers better visualization of the learning agents running in parallel through an Actor called *AGridActorVisualizer*. This Actor helps to visualize and differentiate multiple learning agents by representing them in a grid, as seen in figure 5.2.



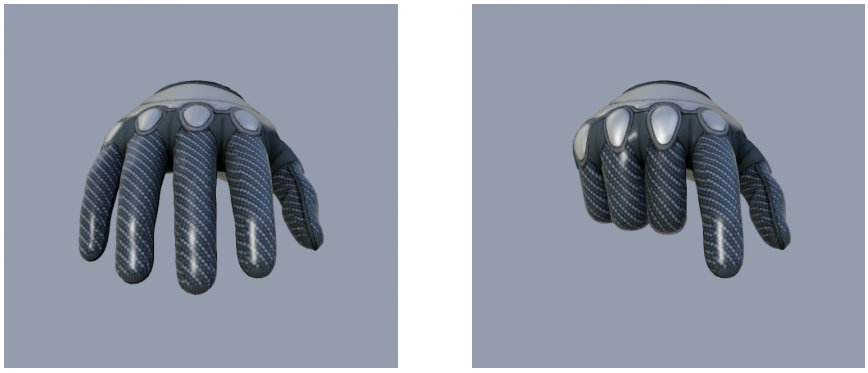
(a) Without visualization.



(b) With visualization.

Figure 5.2: The impact of *AGridActorVisualizer* in Unreal Engine.

To the environment closely linked belongs an *AActionController* executing all actions from the learning algorithm of the Python script within the UE environment. Of course, this is available in both discrete and continuous forms to enable the usage of both action types. An action controller is visualized as a hand mesh within UE and uses an *UAnimInstance* to help visualize the grabbing animation (see figure 5.3) of the hand while training/testing within the C++ code.



(a) Grasp value is set to 0.

(b) Grasp value is set to 1.

Figure 5.3: Images showing the effect of using the grasp value in *ABP_MannequinsIRL* from the C++ implementation.

To influence the hand's starting position within the state space, another Actor named *AHandStartPosition* places the hand at the desired starting location in the scene. Furthermore, this class randomizes the starting pose of the hand using a specified seed.

5.1.4 *Available Functions for Python*

A dedicated *AMessageHandler* interprets data received through *UClient-Components* within its implementation to pre-process and categorize the messages sent by the server. Therefore, the implementation offers two delegate types with one or two parameters. This construct allows registered functions within UE to receive information based on whether the message contains extra information or if the call alone is sufficient. The implementation created six class members from these delegates, each representing a callable function by the server. These functions are:

- **TransitionProbability:** Generates the dynamics of a discrete environment. This function is only needed for algorithms depending on a model of the environment to operate. However, neither GAIL nor AIRL needs this function.
- **VisualizeReward:** Another function for the discrete setting. This will visualize the expected return of the reward function at

each state with the colors red as a low reward and green as a high reward. Of course, if the expected return is between the minimum and maximum value, the color between both gets interpolated.

- **MetaInformation:** The function that shares all hyperparameters configurable within UE for the learning algorithm, as well as the size of action and state space. Usually, it only gets called on the first client instance, since the server only needs this information once at the startup.
- **ExecuteStep:** Executes a step within the UE environment, resulting in a hand Actor representing the expert moving within the defined state space. A vector containing float values represents an action. Those values encode a movement vector, the rotation, and how the grasp value changed.
- **ExecuteReset:** Resets the environment according to values specified in the Python script and additionally randomizes the hand if specified with a user-defined seed. Usually, training the learning algorithm does not require randomization, which is why it should stay deactivated.
- **ExecuteChangeSeed:** Changes the seed generating the initial poses of all objects in the scene. It allows the Python script to change the seed within UE. Useful when trying to generate multiple trajectories starting with the same seed without restarting UE all the time.

As mentioned in section 5.1.1, messages for communication between both processes are JSON objects. These offered a simple transformation of structs defining the action and state space to JSON objects. Thus, reading and writing information within the UE project was easy. After UE finishes a requested action from the Python script, it converts the data back into a JSON object to return it to the server.

5.1.5 *User Interface and Scenes*

The UE project offers many settings, including modifying some hyperparameters of the learning algorithms. Figure 5.4 shows the UE editor open with the implementation. Selected within this window is an Actor responsible for further configuration of this project. The Actor's name is *AEnvManager*, and it controls the currently used game mode, which can either be IRL or VR. The naming IRL is a remnant of the discrete project phase. At the moment, it means IL in general. Thus, it also has to offer all the necessary parameters to set either of them correctly. Within a scene are different environments containing different objects for a task. The system will use either of these environments for the different modes. Which of these modes is another

option available in the Environment manager (see appendix A.2.3.1 for a detailed explanation of configuring this manager).

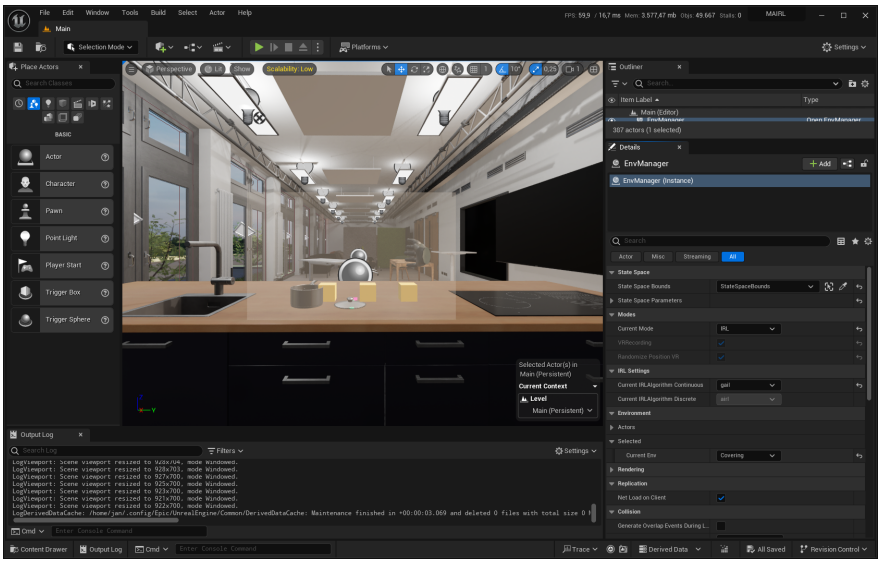
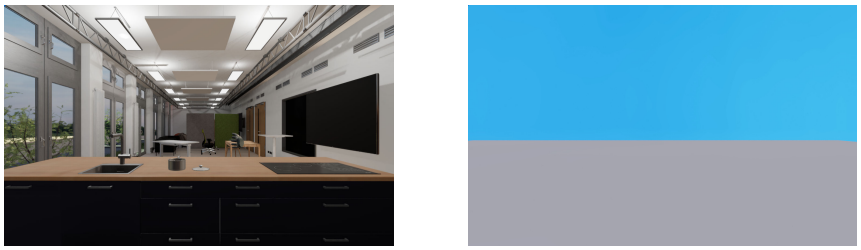


Figure 5.4: Unreal Engine Editor with *EnvManager*.

Internally, the project uses some *Enums* like *EGameModes* to create entries for the drop-down menus inside the *AEnvManager*. Furthermore, two scenes are available for the user to choose as training scenery. Firstly, a standard kitchen environment, but also an environment with no other elements besides a plane and the training objects to reduce hardware demand in case of overloading the graphics card. Figure 5.5 also shows this difference in these scenes.



(a) Kitchen: The standard environment used for the project presentation.

(b) Plane: Without additional objects to reduce stress on hardware.

Figure 5.5: Images showing the different scenes that are available for user selection.

5.1.6 *Training Environments*

Implementing the environments followed the description of section 4.2. Alongside the three primary environments seen in figure 5.6, there is also a dummy environment seen in figure 5.6d. This plain setup includes only the hand as the agent, with no other objects present.

(a) Covering: Place a lid on top of the pot.

(b) Insert: Spatula needs to be placed inside a pot with rotation.

(c) Stacking: Three cubes for the task.

(d) Dummy: Toy problem for testing the whole setup.

Figure 5.6: All environments implemented in Unreal Engine with the initial pose of all objects. One common element is the hand, which represents the learning agent.

This dummy environment tests whether the algorithm can learn basic policies without additional items in the room and without relying on perfectly tuned hyperparameters. The training set of the dummy environment consists of simple forward movements, so the agent could at least learn this essential capacity.

5.1.7 *Virtual Reality*

For VR, the standard *VRPawn* provided by UE was sufficient. However, since USemLog only provides the capability of recording object positions and not the capability to record all actions taken by an VR controller without crashing, the implementation adds another small component to the *VRPawn*. This extension, named *UActionMonitorComponent*, aims only to record the right hand since this is the only input needed for the later learning phase. *UActionMonitorComponent* gets called by the motion controller blueprint, which transmits the currently used action name and the type of action. A log file gets automatically extended by appending actions from the action monitor. Another problem is that USemLog does not track the skeletal mesh component of the hand used by the *VRPawn*. Hence, in the implementation, another Actor named *AVRPawnHandTrackball* reflects all hand movements. Tracking of this Actor works because it uses an *AStaticMeshActor*, which the software can track.

5.1.8 *Reading and Writing Data*

Of course, the system also needs to write critical information for later data handling outside UE. A prominent example is the recording of trajectories. The system will automatically create a structure containing all the files generated during the recording to mitigate user error. This user error could otherwise happen since USemLog initially stores all information inside a database. Implementing this feature requires special tools to handle the database information and also tweaking of USemLog's default values. By default, the Actor *SLWorldStateLogger* of USemLog uses randomly generated names for collections and databases, which cannot be changed in code when added and configured in the Editor. This creates a problem because a script gathering the information does not know which collection it belongs to. The solution for creating easily distinguishable trajectories within a database is to create the *SLWorldStateLogger* in C++ and to set the correct information at creation. Another solution would be to override the same database every time, though this is inconvenient when directly interacting with the database. The final implementation can then call the external application *mongoexport* with all required parameters specifying the database and more to save the trajectories in the folder structure.

5.2 PYTHON

Due to many already existing packages like Imitation, Stable-Baselines3, and other IL algorithms implemented in Python, the project uses Python to implement the learning part. This section will explain some bits of the implementation.

5.2.1 *Server Implementation*

Python serves as the server; therefore, it must also implement all necessary functions to manage and create independent client socket communications to UE. The Python implementation is the server because UE offers the capability to train multiple learning instances in parallel, where each instance gets independent instructions from the learning algorithm within Python. Creating an individual channel for each instance makes the most sense from a design point of view. Using the Python package *socket* enables the TCP/IP communication for the script. However, two classes divide the actual implementation of communication. First, the server class *IRLServer* searches for a specified number of clients and thus constantly listens for communication requests from clients created by the UE process. Each new connection between the Python process and the UE process gets represented as a unique socket object. The second class *IRLClientInstance* requires the socket created by the *IRLServer* for its creation. When created, *IRLClientInstance* listens for incoming messages from an UE client. This only happens when UE gets a request beforehand. A client method gets called by using a name represented as a string and some additional parameters if needed. The server manages and calls individual client methods to perform actions within the UE environment using their respective client numbers.

5.2.2 *Task Reward*

Task reward, introduced in section 4.5, addresses the problem that the number of recorded trajectories might not be enough to generate a policy that generalizes to the expert's intent. Since Python controls the complete learning process for all IL algorithms, this is also the process to apply the concept of task rewards. Implementing this concept requires switching to a custom discriminator. The default for training is the *BasicShapedRewardNet*, which also serves as the base for the implementation. It extends the reward network to a user-specified task reward fitting to the current learning environment and a balancing value to weight the output of both trained reward and task reward.

A specific Python file then defines three different task reward functions according to section 4.5 and appendix A.1—one for each UE environment. A task reward evaluates a tuple consisting of the current

state, an action, the next state, and whether the agent reached the final stage. The last entry is entirely irrelevant, though the implementation of the reward function should match the implementation of the reward network. Depending on these values, the function generates a reward. A hyperparameter configuration file (the file looks like this: *./Hyperparameters/UEGymEnv-*.json*) then contains the correct task reward to a training environment.

5.2.3 *Reading Trajectories*

Since trajectories recorded in UE are saved in an external database and then exported as a JSON file, some pre-processing is required to transform this information to an internally usable object for learning. This introduces the *TrajectoryReader*, which is a class whose only purpose is to create usable trajectories from a JSON file created by USemLog. It works for both continuous and discrete cases. Also, since a trajectory file does not include the action taken during a state transition, the *TrajectoryReader* has to add these recorded actions from another file in order to turn the incomplete trajectory into a fully-fledged one. The end product is an object containing *numpy* arrays of vectors representing the states (also called observation) and actions. Additionally, the reader applies smoothing to the final grab values in both the actions and states to reduce the abrupt transition between grabbing and not grabbing, making it easier for the algorithm to train.

5.2.4 *Environments, Action and State Space*

Two implementations, offering various functions for processing requested or sent data by the learning algorithms, serve as an interface for simplified communication with UE. The reason behind multiple interfaces is that the needed functionality differs depending on the learning algorithm. Initially, due to the discrete approach, the plan was to implement only an algorithm that operates in discrete space. This algorithm required functionality for processing the dynamics of an UE environment and more, leading to the design of an explicit implementation named *UEMaxEntEnv* for this algorithm.

However, using the AIL algorithms made this unnecessary. Also, Gymnasium's *Env* is the interface, which the continuous learning algorithm already uses. Hence, the project integrated a new implementation using Gymnasium's *Env* class called *UEGymEnv* into the project. It offers either discrete or continuous training. Moreover, since not normalizing observations and actions could cause potential performance loss with the learning algorithm, an extra class named *UESpaces* keeps track of the normalized and denormalized action and state space boundaries [7]. The class normalizes the actions symmetrically between -1 and 1 and the states between 0 and 1.

However, there is another problem, where the internally used method of Imitation to create parallel running environments, which uses *fork_server* from the package *multiprocessing*, causes initializing parameters to vanish. This means the project's implementation relies on a file created to give each instance the needed hyperparameters. The file contains information about the state and action space of the environment.

5.2.5 *Learning Algorithms*

To start the learning process, a server thread initializes the communication between Python and UE. Once this is successful, the abstract class *AlgoBase* gathers all necessary information by calling *MetaInformation* in UE to set up the right learning environment. The information gathered from UE decides the next step. If the parameters specify only to show the expert's trajectories, an option not meant for training, then the normal learning process will not start. A user could verify that all recordings are flawless with this option. If UE did not send this parameter, then this abstract class calls a method to start learning implemented by one of the inheriting classes:

- **AlgoMaxEnt:** A simple extension of the base class that implements the training method using UE's dynamics to calculate the rewards with the MaxEntIRL algorithm. After finishing the calculations, the method returns those values and visualizes them in the UE environment.
- **AlgoGym:** Another abstract class for learning algorithms that use the Gymnasium's *Env* interface. It offers threads that handle the communication between individual Gymnasium environment instances and their UE client counterparts. Furthermore, this class also creates the external file needed for the parallel running Gymnasium environments. Since the format of a trajectory differs when using the continuous learning algorithm, the implementation also offers recreation of this trajectory representation for the Imitation script.
- **AlgoAIRL:** This class manages all hyperparameters for AIRL and uses the usual components of Imitation, instead of relying on sacred and the Imitation script. It was created before *AlgoImitationAdversarial* but had some problems, hence the switch to Imitation's script.
- **AlgoImitationAdversarial:** To reduce the source of errors, this implementation uses Imitation's learning script. This change also means that sacred, which conveniently manages all runs and hyperparameters, is now included for each execution of an experiment. If required, the user can view logs containing

metrics logged from sacred during training with an external software called Omniboard. Also, this class offers the option to display a trained policy and automatically optimizes a set of hyperparameters using Optuna.

During development, some experimental implementations occurred. For example, the *PPOContinuousIRL* served as an experimental modification of Stable-Baselines3's PPO implementation, adjusting the outputted grab value within an action to either 0.0 or 1.0. This change aimed to test whether discretizing the values could lead to better performance, though it introduced too much discretizing error.

5.2.6 *Hyperparameter Tuning*

Hyperparameter tuning utilizes a modified version of Imitation's hyperparameter tuning script. At its core, it uses a version of Optuna. For optimization, Optuna requires the user to specify a search space for the exact hyperparameters during training and the sample area of those parameters. For instance, the parameter *batch_size* gets a list of possible options to choose from while optimizing. These settings can also include the number of runs and their duration. The setup of Optuna in this project ensures that only one run gets executed at a time, guaranteeing that the learned hyperparameters remain valid for training the final runs with more time steps. Otherwise, the number of parallel clients will decrease if the implementation allows the optimization of multiple parallel runs. The reason is that some hyperparameters depend on a specific number of parallel-running clients. Since the script is an additional intermediate step executed on external processes, implementing hyperparameter optimization introduces more inter-process communication. To create such inter-process communication between Optuna's optimization processes and the process containing the interface to communicate with UE, the implementation again uses the Python package *multiprocessing*. This way, another process holds all the queues that manage requests and responses, making them accessible via an interface in each other process. Ultimately, the Optuna script automatically samples the hyperparameters and tests them individually in runs in UE, outputting the hyperparameter set with the best values alongside the final policy in the results.

5.3 TESTING

Testing the communication between the Python script and UE worked flawlessly and was straightforward. First, testing reused the recordings already done by the expert. The implementation read these trajectories, transforming them into actions, which then got sent to the UE. During execution, all objects within the simulation of UE moved precisely

like when the expert had moved them; thus, the communication worked. Also, this was a method to test whether the recordings worked flawlessly besides viewing the files directly.

Another manual verification ensured that the task rewards functions as intended, providing a higher reward upon reaching the next stage. In this case, testing required turning off some components of the whole algorithm. One component is generating action by the learning script to prevent interference while moving the hand. The corresponding environment for a task reward had to run in UE, with the player controller detached, which allows manual movement of Actors using UE's scene manipulation tools. Adding a print statement within the Python script outputs the current return from the task reward during this process. Testing with the standard debugger was no option due to how often the debugger would stop at a breakpoint. The verification process involved executing each state manually within the UE Editor. This approach enabled the tuning of the task reward to correct values, and also facilitated improvements to specific parameters that could have resulted in a faulty implementation.

To ensure the entire program effectively learns a policy with UE, the previously described toy problem in section 5.1.6 served as the *Dummy* environment. This setup allowed learning basic movement, demonstrating that the implementation could also work with other environments, although it necessitated identifying the appropriate set of hyperparameters.

5.4 USAGE

This section briefly explains how to use the software project. Look for a more in-depth explanation in appendix A.2. Also, read the *README* file of the project for installation.

5.4.1 *Unreal Engine*

The user can apply most configurations within the UE. Specifically, an actor in the standard scene called *EnvManager* enables configuration options. It allows switching the game mode from recording to IRL, allowing you to choose a specific IL algorithm. Among the game modes, IRL is essential because it allows for communication with Python and, thus, training. The user can customize various training settings, such as using discrete state and action space, visualizing the state space, adjusting time steps, determining the client count, optimizing hyperparameters, and more. The system saves recordings generated from VR in the directory *./IRL_Trajectories* and learned policies in the directory *./learning/output*. To better view the recordings, access Omniboard at `http://localhost:9000` while the docker container is running.

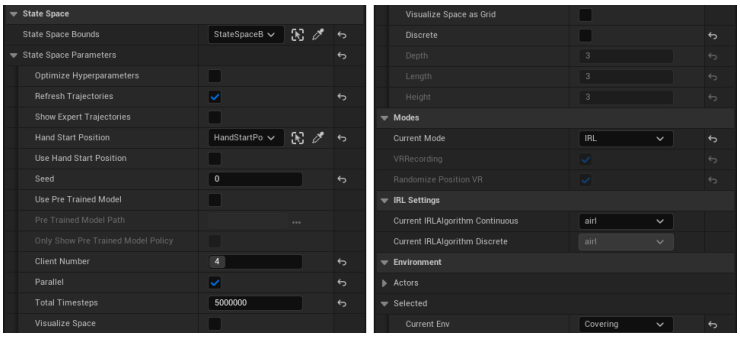


Figure 5.7: Complete view of EnvManager in Unreal Engine.

5.4.2 *Configuration Files*

Since UE only has a few hyperparameters, such as time steps, external files are necessary to configure the hyperparameters of the learning algorithm further. To be more precise, the *./Hyperparameters/*.json* files for ordinary IL and for Optuna hyperparameter optimization, some configurations in the *./learning/irl/algo_imitation_adversarial.py* have to change.

5.5 UNSOLVED PROBLEMS

There are still a few minor issues in the programming. For example, the UE hangs if you stop executing UE before the Python server establishes a connection. Furthermore, the server sometimes fails to start if the user tries to relaunch it too quickly after terminating a previous instance. Moreover, probably the biggest issue is a problem directly linked to the UE implementation, which increases the memory usage of the application slowly in the span of a couple of days. This memory leak causes the hyperparameter optimization procedure to use more RAM than required since optimizing parameters requires a few days.

Part III

EVALUATION AND CONCLUSION

The analysis and discussion of results, a summary of findings, and recommendations for future work.

6 EVALUATION

This chapter covers the evaluation of the system from recording of trajectories to using it in various learning algorithm configurations to see which performs the best. A test set consisting of 12 trajectories independent of the training set of 100 trajectories serves as a baseline for comparison.

6.1 DATA COLLECTION

Gathering the data for evaluation required two systems for recording and training. Table 6.1 shows the hardware specification and the applications of these systems. As an explanation, the data collection required two systems since training took longer than initially expected. Two systems ensured parallel tuning of hyperparameters with Optuna and starting with IL on manually tuned hyperparameters. Furthermore, due to the intensive workload of simulating multiple virtual environments and calculating policy improvements, further load on a system would have increased training times, making comparisons between configurations less fair. One last remark is the higher requirement of system memory for Optuna due to the problem stated in section 5.5, which system *System 2* offers. The following sections gloss over the procedure and settings to acquire the data.

Hardware		
	System 1: VR + IL	**System 2:** Optuna
CPU	Ryzen 7 3700X	Xeon Gold 6128
GPU	Radeon RX 5700 XT	Tesla V100
RAM	32 GB	48 GB

Table 6.1: System specification for both trajectory recording and imitation learning as well as the system specification for the hyperparameter tuning.

6.1.1 *Recording*

An expert performed the motion needed to record the trajectories using a head-mounted display and a motion controller to create a training and test dataset. The training dataset consists of 100 recordings with different initial objects and hand placement to ensure better generalization of the expert's intent from the learning algorithms. Though the virtual environment places the object, the expert must

manually ensure different starting poses for the motion controller. This approach is necessary because automatically varying the rotation poses might be irritating and fail to represent how the expert holds the controller accurately in the real world. Consequently, the recording will start after a brief delay rather than immediately. The expert needed to do the same procedure for the test set consisting of 12 trajectories used for later evaluation. Contrary to other approaches common in RL, where only a predefined reward function linked to an environment serves as the metric to evaluate the performance of a learning algorithm, this thesis additionally uses the mean return of the expert according to the corresponding task reward as a baseline for comparison—more on that topic in section 6.2.

Gathering the demonstrations for the algorithms took around two and a half hours per affordance, including the time to review and retake some of the trajectories. This review was a pre-processing step to ensure the collected data had no errors. Sometimes UE did not stop the recording, which generated too long trajectories, and sometimes USemLog induced faulty trajectories by starting the recording too early. This behavior induced readjustment of the controller's position to appear in the final results, which caused jumps in the trajectory.

6.1.2 *Training and Runtime*

The experiment compares the performance of the implemented AIL algorithms with different configurations using UE. Both algorithms utilize the same set of manually tuned hyperparameters across all environments, with varying balancing values λ that affect the blending of the task reward. Furthermore, to evaluate whether automatically tuned hyperparameters enhance performance, the hyperparameters optimized by Optuna undergo the same procedure. However, instead of one set of hyperparameters for all configurations, the automatically optimized algorithms get their own set depending on the environment and learning algorithm. Appendix A.3 lists all hyperparameters used for training.

The final evaluation relies on the environment, as shown in section 5.1.6. During training, the system randomly samples each object's initial pose within the environment according to the initial pose within the training set. Each environment uses a maximum of $5,000,000$ training steps to learn a policy with four agents running in parallel. *System 2* could not handle more agents, which reflects the reason for choosing these values among time constraints. The evaluation uses three balancing values, 0.7, 0.5, and 0.3, to gain a broad insight into their effect. For each fifth checkpoint, the system saves a policy reflecting the current learning progress of the learning procedure for later evaluation.

Due to differing hyperparameters, the number of rounds that equals the number of checkpoints varies. The number of rounds relies on how many steps the generative network takes before it switches to the

discriminator. Hyperparameters also influence the number of discriminator updates each round. All these factors of the hyperparameters also cause algorithms to take more or less time despite having a similar number of training steps. Final results showed training time between six and eight hours per environment on *System 1*, leading to around ten training days across all settings, taking breaks for the system into account.

As mentioned, Optuna optimizes each environment and AIL combination. However, since AIL training alone takes more than six hours and *System 2* has slower hardware, the training parameters are significantly lower for hyperparameter optimization. Instead of using 300 runs with different configurations for evaluation, the system only relies on 60 runs. The same is true for the steps used in each run. The number is $1,300,000$ instead of five million training steps in AIL training. With these Optuna parameters, optimizing the hyperparameters for each environment and learning algorithm combination took $5-6$ days each. Requiring around a month for the final results of hyperparameter optimization. Another remark that influenced training and setting up an optimization run in *System 2* was the connection through remote desktop software. This connection caused the system to slow down when the UE window appeared with a large rendering surface on the screen. Reducing the window, which simultaneously reduces the number of pixels needed for rendering, significantly decreased the training by a few days.

6.2 RESULT

Gathering the evaluation trajectories required additional scripts to independently evaluate each trajectories's returns and record the trajectories generated from a policy. As previously mentioned, the evaluation uses a test set to evaluate whether the different configurations had any effect. The results of the IL training consist of multiple checkpoints representing the training process at a particular stage. To evaluate the improvement at each stage, all policies get the initial pose according to the test set, from which they can generate trajectories—using the mean return over all trajectories according to the corresponding task reward function results in the mean return for a checkpoint. The mean return of the expert shown in table 6.2 then serves as a baseline to evaluate how well the algorithm performed at each stage.

Configuration	Covering	Insert	Stacking
Expert	522.66	410.25	578.54

Table 6.2: Mean return of each environment when taking the expert's trajectories.

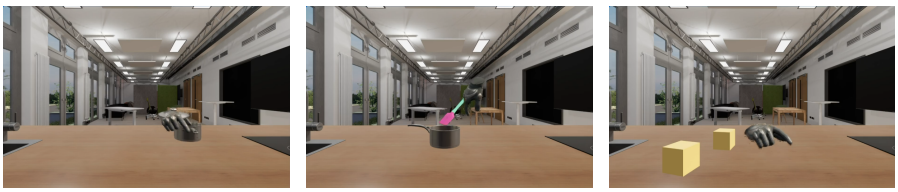
However, the small test set introduces noise to the outputted values, which requires some post-processing. Therefore, the evaluation

uses moving averages to reduce short-term changes within the policy training and emphasize global progress. Configured is a window size of five, and of course, the length of the actual dataset shrinks by two checkpoints at the beginning and ending to eliminate the boundary effect due to the convolution. Also, for evaluation, plots do not contain outliers to increase readability further. Furthermore, some plots in the later section seem to perform worse over time. This is not the case, since a random policy sometimes archives a close-end position of the objects at the beginning, which results in a high reward. The following section will go into more detail.

Before listing all results, figure 6.1 visualizes the best policies for each environment in four stages for a better understanding. The following pages introduce plots containing mean returns for all configurations and the corresponding minimum and maximum values for each checkpoint in transparent form.

(a) 1/4 Covering (b) 1/4 Inser (c) 1/4 Stacking

(d) 2/4 Covering (e) 2/4 Insert (f) 2/4 Stacking



(g) 3/4 Covering (h) 3/4 Insert (i) 3/4 Stacking

(j) 4/4 Covering (k) 4/4 Insert (l) 4/4 Stacking

Figure 6.1: Each column represents the best trajectory gathered from the environment's final best performing policies. Four stages visualize the progress and show success in some cases. The following configurations produced these results: covering manually GAIL $\lambda = 0.7$, insert automatically GAIL $\lambda = 0.5$, stacking automatically GAIL $\lambda = 0.5$.

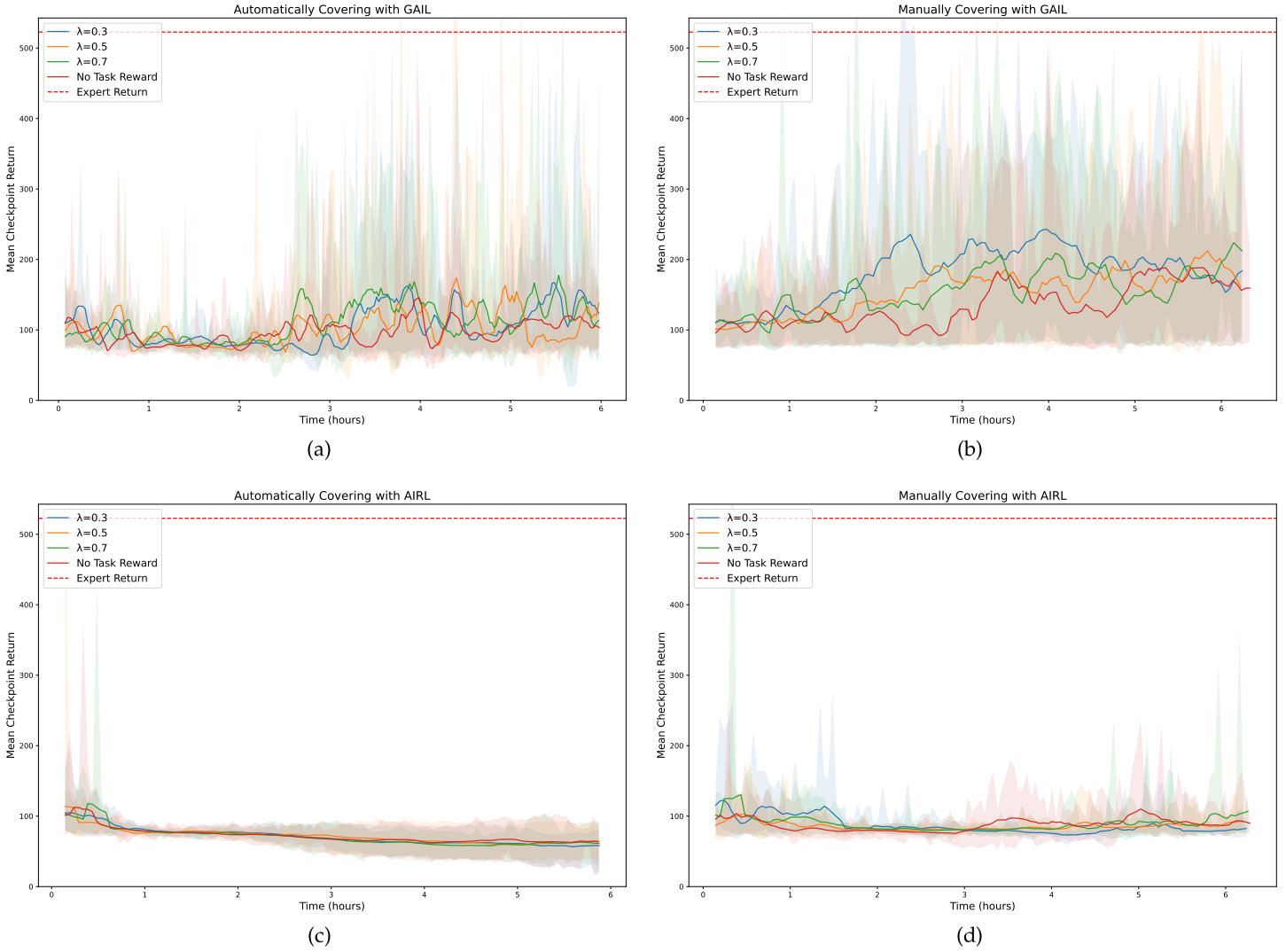


Figure 6.2: **Results of covering:** Both configurations with AIRL seemingly reach a local optimum early during training, while GAIL show more promising results. The manually tuned results learn faster and get higher results at this task. None of these results reach the expert's mean return.

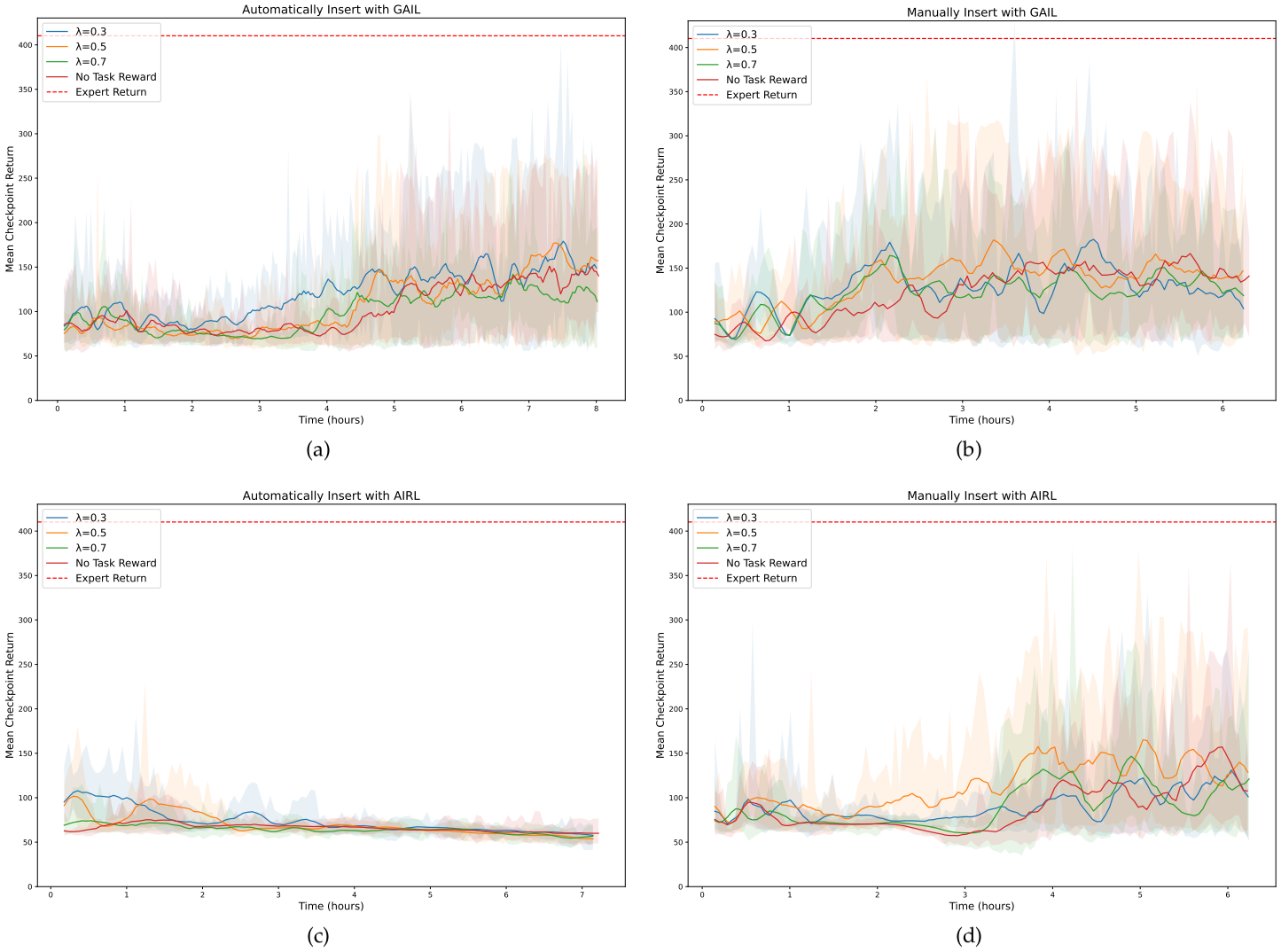


Figure 6.3: **Results of insert:** Only the automatically induced hyperparameter set seems to struggle using AIRL. All other configurations start to learn the behavior of the expert. While not by a considerable margin, the automatically induced hyperparameter set for GAIL is the best-performing configuration. However, none of these results reach the expert's mean return.

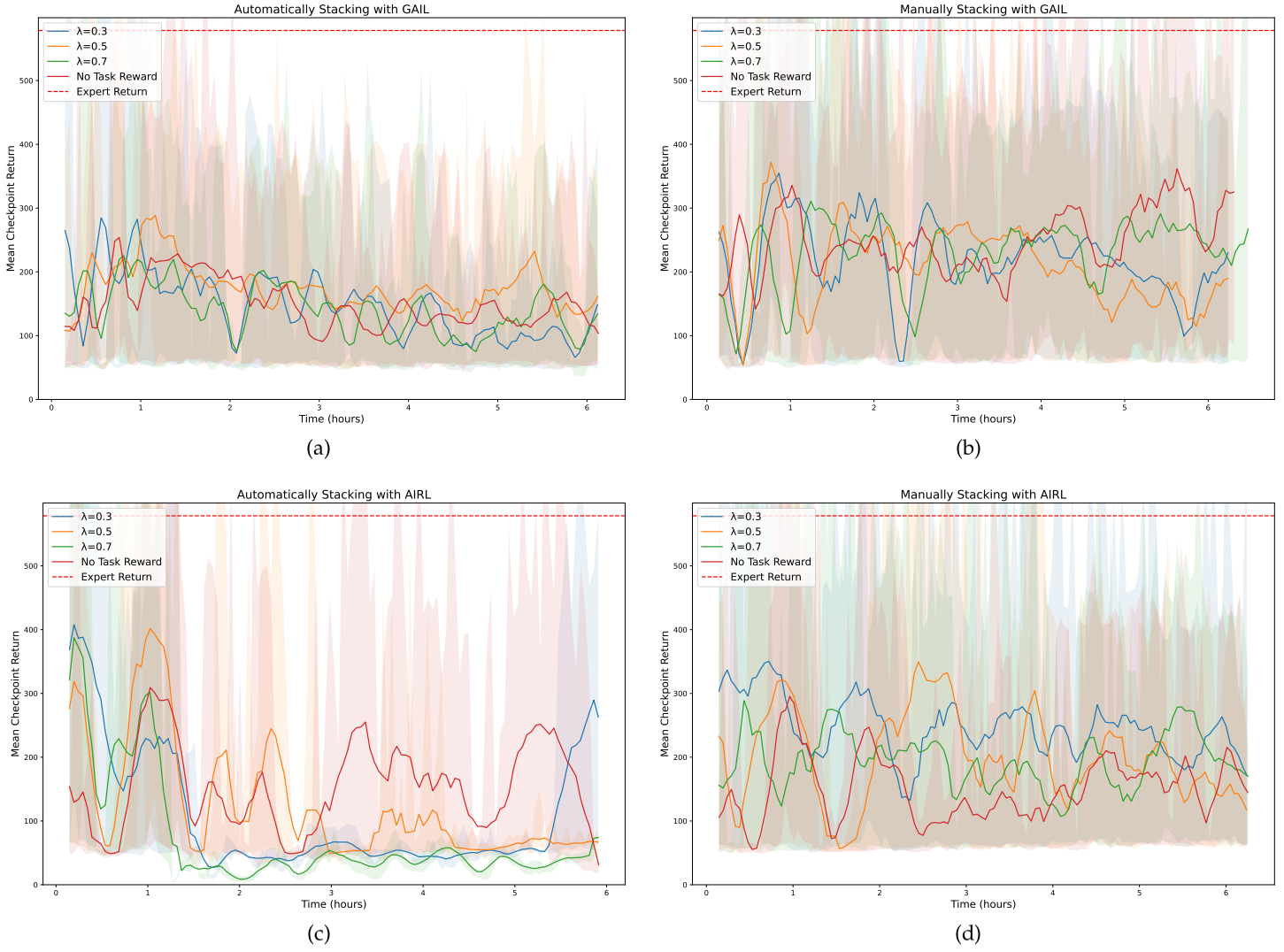


Figure 6.4: **Results of stacking:** While training stabilizes the agent's movement within the simulation, all configurations struggle to imitate the behavior just once. The plots also reflect this situation, where none of the configurations increase the mean return over time.

6.2.1 *Covering*

Figure 6.2 shows all important plots. The environment covering with AIRL achieved similar results regardless of the configuration (see figure 6.2c and figure 6.2d). Interestingly, the results stay the same or worsen over time. However, it only seems like that because of the definition of task reward. In its first stage, it rewards the movement towards the lid. Since this behavior gets reinforced in the underlying PPO algorithm early on without further exploration to other stages, there is no rise in the reward over time. However, the runs using manually tuned hyperparameters still have more variance within each checkpoint, making the mean value over time less consistent than with the automatically tuned hyperparameters. This is caused by a, in comparison to the automatically tuned set, more chaotic agent movement of the policy in the simulation. The other parameters achieve a uniform movement towards the lid without further movement, so no further reward is issued. While hand movement stabilized in both cases, none of the AIRL consistently tried to pick up the lid.

However, the results of GAIL are more interesting. Where both AIRL algorithms never achieved to pick up the lid, GAIL is consistent in doing so. Even placing the lid close to the top of the pot is possible, but it is an exception rather than the rule. To be more precise, often, only the manually tuned achieves this. Figure 6.2b reflects this observation with a higher mean reward, showing the manually tuned hyperparameter set as the best performing policy in covering. Compared to the other GAIL configuration, this is likely due to the higher learning rate and entropy coefficient impacting the exploration.

Overall, the usage of task reward impacted the performance and learning speed of the policy learning process only by a small margin. Moreover, none of the policies learned by all the different configurations succeeded or had a big lead in performance, making a final statement about the effectiveness difficult.

6.2.2 *Insert*

Even though both covering and insert share a similar setup, figure 6.3 shows more deviating results for insert than expected. However, the configuration using automatically tuned hyperparameters with the learning algorithm AIRL stays mostly the same (see figure 6.3c). It achieves constant movement towards the spatula but never picks it up. One deterioration to covering, however, is that the movements induced by the policy are no longer as smooth, and training took seven instead of six hours. The more interesting observation with AIRL is that the manually tuned hyperparameter reached the next phase of picking the spatula up despite having the same hyperparameter set as covering and the same number of objects within the environment. A plausible explanation could be that the way the expert picks up the object in the

trajectory recording with a slight rotation gives the AIRL approach a better hint to differentiate between learned and real trajectory, leading PPO to reach the next stage. Still, the expert only picks up the object but never moves towards the pot.

Again, both GAIL algorithms accomplish the first stage of picking the object up. However, when visually evaluating the output of the automatically tuned hyperparameter policy, this policy seems to reach a state close to the end state of the expert more often. Regardless, figure 6.3a and figure 6.3b do not reflect that observation. Here, both hyperparameter configurations perform similarly, except that the automatically tuned set needed around eight hours instead of six, and learning how to pick up the object was slower than the manually tuned hyperparameter set. A lower learning rate is probably responsible for the slower learning speed, and a smaller batch size and more discriminator updates lead to more extended training.

Overall, using task reward did not significantly affect learning. However, the automatically trained hyperparameter set for GAIL, including the task reward with $\lambda = 0.3$, seemed to accelerate the learning but did not improve the final result. Furthermore, none of the configurations lead to a policy, which could consistently solve the task.

6.2.3 *Stacking*

As shown in figure 6.4, stacking results deliver no real success. When looking into more detail, AIRL with automatically induced hyperparameters generate trajectories, where the agent spins into one of the state spaces corners while constantly trying to grab something. This behavior also leads to overall low rewards with occasionally higher rewards due to objects that get grabbed and thrown away, which increases the accumulated reward for a trajectory (see figure 6.4c). The manually tuned hyperparameter set for AIRL at least stabilizes the movement and also sometimes leads to the direction of one cube, which adds the noise in figure 6.4d.

In this case, GAIL acts no different. Although, the automatically tuned results stabilize way better than the manually tuned results (see figure 6.4a and figure 6.4b). More interesting, though, is that the overall mean return of the manually tuned results looks way higher; however, its policy spins the agent around. This observation means that even though the hand grabs objects within the environment and throws them near other objects, which generates higher rewards, the seemingly lower rewarded automatically tuned hyperparameter set generalizes better to the expert's intent due to hand movement resembling the expert.

However, none of the algorithms presented with the configuration solved the task in the end. In this case, the problem is probably the

design of this environment, which makes it harder for the algorithms to learn since objects can be picked up in an arbitrary order.

6.3 DISCUSSION

The findings indicate that GAIL has demonstrated consistently more pleasing performance. However, the overall results of all IL configurations did not meet the mean return of the expert or the final stage for successful task execution. One reason for stacking, in particular, might be the overall design of the task, where the expert had no differentiation between objects, which enabled arbitrary object placement—making the whole process harder than the other tasks.

While including the task reward can sometimes accelerate the training in the initial stages, it did not have a meaningful impact on the final goal of having a stable policy that generalizes well. Furthermore, the introduction of hyperparameter optimization, while not helping with a policies always fulfilling the task, still got hyperparameters that enabled a policy to learn the task or at least parts of the task similar to the manually defined hyperparameters. In the case of the GAIL algorithm in stacking, it even surpasses the manually tuned parameters. Still, this is an impressive result, considering that manually finding usable hyperparameters required several runs during the implementation phase and more to achieve adequate results. In contrast, some of the manually defined hyperparameters in the initial phase of this project only got slightly beyond a policy similar to that of a random one. Visually speaking, these hyperparameters only caused movement to another point in space but did not enable proper agent rotation from the expert.

One drawback of hyperparameter optimization is that producing a non-optimal hyperparameter set takes several days. Considering that, with a more significant run number, e.g., 300, and a higher step number close to the one used in actual IL training, it can be hypothesized that this could result in a set of hyperparameters that outperforms the manual whenever the manually tuned hyperparameter set is not the optimal set. However, with the current implementation, getting just one set of hyperparameters would take weeks, which is not feasible. Future efforts should focus on reducing the complexity of the simulation environment to facilitate parallel training, which would enhance the efficiency of the optimization process. Additionally, it is crucial to consider the variability in hyperparameters based on distinct environments to achieve more robust performance.

Another limitation is probably the lack of a larger number of training samples, which likely led to the early termination of configurations that hindered the reinforcement of the desired behaviors. Overall, greater sample sizes and more extensive testing could lead to the discovery of superior hyperparameters that outperform manual tuning.

7 CONCLUSION

In conclusion, this section summarizes the work completed, outlines the key results obtained, and suggests areas for further improvement.

7.1 SUMMARY

This thesis presented a fully functional system to capture expert demonstrations for behavior imitation in Unreal Engine 5.3.2. To collect the necessary trajectories for the learning algorithms, a VR setup, including a head-mounted display and controller, was utilized. The expert had three distinct environments tailored to the affordances typically in a kitchen environment: one for stacking items, another for covering items, and a final one for placing items inside a container. Two established algorithms from the AIL category (a sub-group of IL) served as the foundation for learning evaluation. The first being AIRL and the second GAIL. Given the challenges related to the generalization of IL methods, the implementation also relied on the concept of task reward as an auxiliary function. Furthermore, the significance of hyperparameter optimization was emphasized, both during the implementation of this project and in existing research [19]. Consequently, the system used an automatic hyperparameter optimization technique known as Optuna to achieve enhanced results.

The final evaluation shows that while GAIL performs better, none of the IL configurations reliably solved these tasks. For the stacking environment, this is probably due to the unknown object order for the final construct. Ultimately, for all environments, the lack of sufficient training samples likely constrained the effectiveness of configurations, emphasizing the importance of future work. Although incorporating task reward can accelerate initial training, it did not contribute to developing a stable and generalizable policy. Hyperparameter optimization sometimes yielded marginal improvements but was limited by time constraints, again indicating the need for a more efficient approach, such as simplifying the simulation environment for parallel training. This implies that Unreal Engine might not be the best choice for IL. Even though it works, the training is, at least for this implementation, time-consuming. However, in the end, despite the time limit and the optimized hyperparameters not being optimal, the results were still slightly better in some cases compared to the baseline algorithms with manually tuned hyperparameters.

7.2 FUTURE WORK

Using 100 trajectories is undoubtedly a lot and also takes much time to record; future work should focus on using a reduced set for training. The lifting order of the cubes in the stacking environment was initially not considered. Even though this would not be given in a realistic environment, it also hinders the learning algorithm's ability to learn this task effectively, so future research should color these cubes to make them distinguishable and the position when finally stacked together consistent.

However, the biggest problem was the speed in the final training process, so changing the engine might be a good idea, whether for training or recording the trajectories. For example, MuJoCo could work as a recording and training engine, as it provides at least essential support for VR [42]. Other projects even extend VR support of the MuJoCo engine [60].

Also, a system that utilizes explicit knowledge about action can help, like Lopes et al. [45], who use demonstration interpretation to map expert information to affordances, thereby creating an affordance network. This approach could be modified to create automatic task rewards. Doing so aligns the usage of task reward more closely with classical IL approaches, which rely solely on demonstrations and not an additional user-specified function. Additionally, this method allows for using a static metric for hyperparameter optimization with Optuna.

Part IV

APPENDIX

APPENDIX

A.1 TASK REWARDS

This section contains the remainder task rewards and a few brief additional explanations.

A.1.1 *Insert*

The task reward for the insert environment shares similarities with the previously defined task reward for covering in listing 4.1. This is because both functions have similar stages and a comparable number of objects in each environment. The primary difference lies in the requirement for distance, which needs to be much closer in this context. Look at listing A.1 below to understand this task reward.

Listing A.1: The pseudo-code of the task reward for insert.

```
input: state, next_state # in other words, the initial state and
    subsequent state
movement_hand_to_spatula, movement_spatula_to_pot,
    next_spatula_to_pot, next_spatula_to_hand = initialize these
    values using state and next_state. All these variables are
    scalar values according to the length of the calculated
    vector.
# Zero reward as a baseline
reward = initialize reward with 0.0
# Stage 1: Movement of the hand towards the spatula
if movement_hand_to_spatula exceeds a certain value:
        reward = 0.5
# Stage 2: Hand holds spatula
if: next_spatula_to_hand is small enough and hand grasps
        reward = 1.0
# Stage 3: Movement towards pot while holding a spatula
if movement_spatula_to_pot exceeds a certain value while holding in
     next_state:
        reward = 1.5
# Stage 4: The object is in a pot and not grabbed
if next_spatula_to_pot is small enough:
        reward = 2.5
if next_state is still grabbing while next_spatula_to_pot:
        reward -= 0.5
output: reward
```

A.1.2 *Stacking*

As a side note, the following task reward uses more objects; hence, stages are quite different from the already existing task reward. Look at listing A.2 below to see how the stacking gets supported.

Listing A.2: The pseudo-code of the task reward for stacking.

```
input: state, next_state # in other words, the initial state and
    subsequent state
movement_hand_to_blocks_vector, movement_blocks_to_blocks_vector,
    next_distance_blocks_to_blocks_vector = initialize these values
    using state and next_state. All these variables are scalar
    values according to the length of the calculated vector.
# Zero reward as a baseline
reward = initialize reward with 0.0
# Stage 1: When the hand movement to one box is registered
if one value in movement_hand_to_blocks_vector exceeds a certain
    value:
        reward += 0.25
# Stage 2: If blocks get closer while holding
if movement_blocks_to_blocks_vector exceeds a certain value while
    holding in next_state:
        reward += 0.125
# Stage 3: If two blocks are close together
if the value between two blocks in
    next_distance_blocks_to_blocks_vector is small enough:
        reward += 0.5
# Stage 4: If three blocks are close together
if the value between three blocks in
    next_distance_blocks_to_blocks_vector is small enough:
        reward += 0.5
# Stage 5: If blocks get stacked and are close together
if next_state one block is two times the normal z position:
        reward += 1.0
if next_state one block is three times the normal z position:
reward += 1.0
        output: reward
```

A.2 IMPLEMENTATION USAGE

This section explains the usage of the featured software. Keep in mind that *./README.md* in the root directory of the project contains the installation instructions and all the additional software required. Topics are both the recording of trajectories and training the IL algorithms on said trajectories. However, a section will first explain some general options affecting both topics.

A.2.1 *General*

This section outlines configuration options that apply to all modes available in the software within the *EnvManager* Actor. So, these settings affect the usage no matter the selected project mode. Furthermore, the implementation refers to IRL for all IL learning algorithms.

A.2.1.1 *State Space Parameters*

While the system applies most of the options in this category only to IRL mode, it also offers some helpful options while doing VR recordings. Implementation-wise, the system visually guides the user to move only their hand and objects within the borders of the state space. Of course, objects are also physically held within the state space, but colliding with the borders leads to unnatural movement trajectories. This is avoidable when using this option as visual guidance. In this case, it is *Visualize Space*, which, as the name suggests, visualizes the state space as a translucent white box and is the option to activate such visual guidance. The option appears in figure A.1.

Figure A.1: Shows how to visualize the state space in *State Space/State Space Parameters*.

A.2.1.2 *Modes*

Another option for independent use is the menu entry to change the project mode. The user can switch between IRL or VR to use the desired mode through this setting. As a side note, this option only takes effect when set before running UE. It has no effect when switching while running UE. See figure A.2 to see which drop-down menu to use.

Figure A.2: Shows how to switch the current project mode in *Modes*.

A.2.1.3 *Environment*

The section *Environment* offers the ability to set the objects for all available environments. But more importantly for the user of this project, this section also gives the user the ability to select the current environment used for the next run. Figure A.3 shows the option, which

Note: *All environments are available in discrete or continuous space.*

is marked as the only option that should be used by the user in red, except if object modification of the environments is wanted.

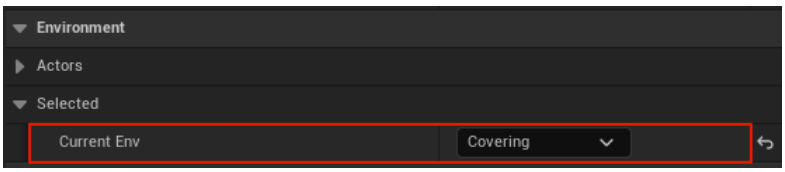


Figure A.3: Shows the options for the environment section within *EnvManager*.

The following list shows the difference of all available environment options within the drop-down menu seen in figure A.3 except for *None* since it is the default empty environment.

- **Covering Env:** A pot and a lid are available for interaction in this environment. Ultimately, the lid should cover the pot's upper part, representing the task.
- **Insert Env:** A pot and a spatula are given to place the spatula inside the pot.
- **Stacking Env:** Here, the environment has three cubes, which should be stacked, forming a little tower.
- **Dummy:** This environment is just a toy problem to test the functionality of the algorithms. A simple grid world includes a single agent in a discrete space. The agent has to reach a specific state to finish. In continuous space, the goal is to learn a movement in one direction.

A.2.2 *VR recording*

The VR mode only generates trajectories and offers a few configuration options. This section will explain how to properly use the VR mode. Keep in mind that this will work best on Windows.

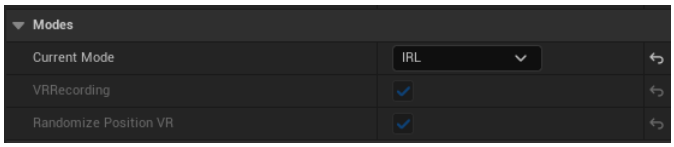
A.2.2.1 *Configuration*

Important: *Make sure you started MongoDB Compass and connect to a MongoDB deployment with the host "localhost:27017" to ensure USemLog to write into the database!*

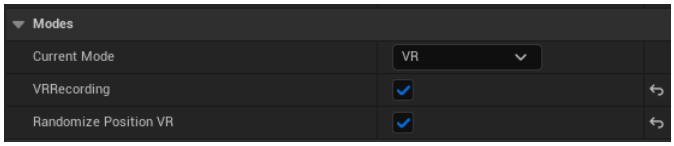
When using VR to generate new trajectories, you have to start all additional software not directly started by UE like your VR setup and database. Like switching to IRL, VR is another option available in the *Modes* section within the *EnvManager*. Nevertheless, as opposed to IRL, VR only offers two additional configuration options. figure A.4 shows the configuration of the VR mode. The following paragraphs explain the configuration for the VR mode:

VRRECORDING: Enables recording trajectories within the VR mode. Interacting with objects in the environment is possible without enabling this option. However, the system will not record or write any information to the hard drive.

RANDOMIZE POSITION VR: Enables randomizing the start position of all objects within a selected environment.



(a) VR mode not selected.



(b) VR mode selected.

Figure A.4: Showing VR mode options when selected and when not.

A.2.2.2 *Recording*

When the user sets the mode to VR, activates *VRRecording*, and configures everything else to the set requirements, recording will start by hitting UE's *Play* button. Recommended is to use a keyboard near the VR setup to start and stop runs with their respective shortcuts. You should only start moving when a green text with *Start: n*, where *n* is the number of runs, appears in the upper left corner (see also figure A.5). Otherwise, the recording of the demonstration might miss some parts. The system assigns the run number based on the count of trajectory directories within the respective environment directory. Also, remember that the system only records the right VR controller. Thus, interacting with the left one will result in an incorrect trajectory. A run only stops when the simulation stops.

Note: *Shortcut for starting is Alt + P and stopping the run is Escape.*

Figure A.5: Message on screen outlined in red appears when recording in VR is ready. The number represents the ID of the current trajectory.

A.2.2.3 *Result*

Figure A.6 shows the directory structure of the recorded trajectories. Intuitively, each parent directory ending with *_ENV* contains runs

linked to this environment. Each recorded trajectory is within a directory named after the ID number of the recording. You might as well see those parent directories with *.srl* added. These directories contain three files holding processed trajectories from the normal directory, building a dataset for later usage while IL training.

Important: *If you delete a recorded run, you have to rename all files to accommodate the number of directories for the recording! Also, delete the database, since entries are only needed after a recording ends for copying. Otherwise, it can cause problems for data creation!*

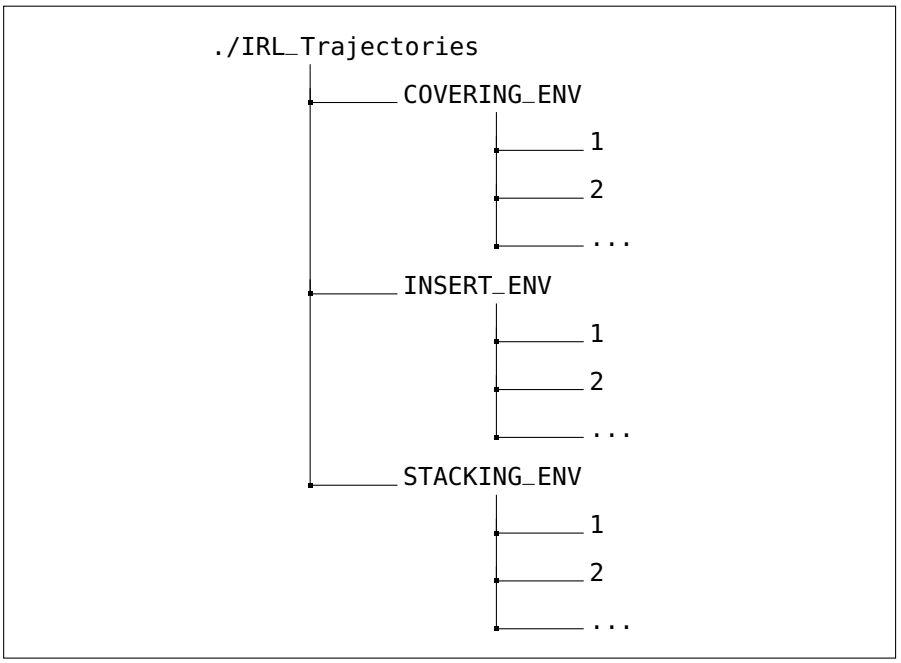


Figure A.6: Structure of the directories generated by the VR recording.

A.2.3 *IRL training*

This section focuses on configuring and running of the IL training. Using Linux as the operating system is important because the software packages required for this training only achieve optimal performance under that system. Otherwise, expect performance impacts, as it is not possible to use parallelization.

A.2.3.1 *Configuration*

Refer to appendix A.2.1.2 for switching to IRL mode and running the docker container. The user can make additional adjustments within UE for IL training. The category *State Spaces* offers many of the settings that influence the training process (see figure A.7).

Important: *When running the IL training process, you have to run the docker container within the ./docker directory to start a MongoDB instance to record training information. Otherwise, training might crash at the beginning.*

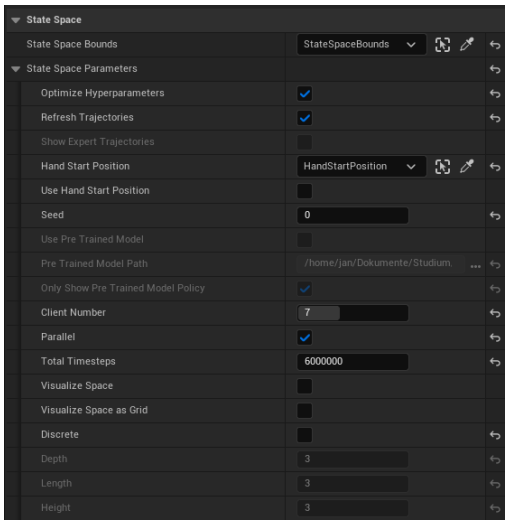


Figure A.7: Shows the configuration menu for IRL mode.

The list below explains some of the configuration opportunities for the user:

- **Optimize Hyperparameters:** Uses a modified version of Imitation's *tuning.py* script, which uses Optuna to find the best set of hyperparameters for the specified learning algorithm and environment. The system saves all results within the user's home directory under *~/ray_results*. Training defaults to 60 runs with a total of 1,300,000 steps. If the user requires other values, they have to change some values in the configurations within "mairl_gail" and "mairl_airl" in the Python script *./learning/irl/algo_imitation_adversarial.py*. This option can require a lot of RAM (at least 64 GB); otherwise, the application might crash and lose all progress. If attempting this option with less memory, adjusting the number of runs, as previously explained, is necessary.
- **Refresh Trajectories:** The Python implementation requires some pre-processing for the trajectories. This also means that if some changes to the recordings in *./IRL_Trajectories* happen, the system only applies them for training when this setting is activated. However, this option should always be selected to prevent user errors since pre-processing only takes a few seconds.
- **Show Expert Trajectories:** This shows the recorded trajectories recorded in VR for the currently selected environment. This can help identify errors within a recording. The parallel running Python script window will also show the number of the currently running trajectory.
- **Hand Start Position:** You can set a *HandStartPosition* Actor with this setting. As the name suggests, this Actor allows to specify the default starting position of the hand object while training.
- **Use Hand Start Position:** To activate the previously mentioned default starting position for the training hand, set this setting.

- **Seed:** The *Seed* option randomizes the starting rotation and position of the *Hand* at each reset if *Use Hand Start Position* is not selected. Selecting 0 as the seed means the algorithm will not randomize the initial pose. Of course, all other options will lead to consistently randomized poses at each run. This option is for IRL mode use.
- **User Pre Trained Model:** If you want to use a model from a previous training session, you have to activate this setting. The user must specify *Pre Trained Model Path* before activating this option.
- **Pre Trained Model Path:** Defines the path to the pre-trained model directory. Please ensure to select only the directory, not the model file.
- **Only Show Pre-Trained Model Policy:** If you wish to review your results with an already trained model, you must activate this setting. UE will not train during the next run and only show actions taken according to the trained policy of the selected model.
- **Client Number:** Defines the number of training instances created in UE. Test different values to determine the optimal settings for the system. It could be that training ends up being slower with a higher value.

Important: *If you notice that UE freezes during training, you have to decrease this number.*

- **Parallel:** If you have not activated this setting, the system will sequentially execute all instances created within UE. Otherwise, the system executes these actions in parallel, drastically increasing the speed if the processor is powerful enough. Deactivate this only if you run on a Windows computer since the Python implementation only supports *Parallel* on Linux.
- **Total Timesteps:** Defines the amount of steps taken for a run in UE.
- **Visualize Space:** This setting activates the visualization of the state space where the environment objects can operate. The system visualizes the space in a transparent white box.
- **Visualize Space as Grid:** This setting is only necessary if you want to view the results of the learning algorithm since this will create a grid representation of all agents to help differentiate between all learning spaces. Figure 5.2 shows the effect of this setting.
- **Discrete:** Activates the discrete training mode. This will also switch the available learning algorithm within the category *IRL Settings* (see figure A.8).

- **Depth:** Defines the depth of the discrete space.
- **Length:** Defines the length of the discrete space.
- **Height:** Defines the height of the discrete space.

As previously explained, there are two different state spaces within which a learning algorithm can operate. To choose the right one, the category *IRL Settings* offers a choice between two options. Depending on whether the user selected *Discrete* in *State Space*, the system greys one option out. Figure A.8 shows the option.

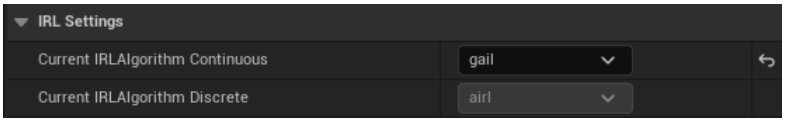


Figure A.8: Shows the configuration menu for IRL training.

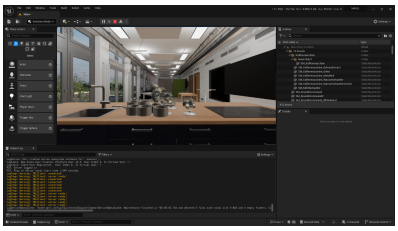
Since each IRL training run uses sacred to log and use externally defined hyperparameters in a run, you could even do more fine-tuning outside UE by changing some parameters in one of the hyperparameter files corresponding to the available learning environments (e.g., in the project directory *./Hyperparameters/UEGymEnv-COVERING.json* to modify hyperparameters for the environment *Covering*). Remember that the system will override the *total_timesteps* parameter due to the settings in UE. In general, the user can see the files within the *Hyperparameter* directory as a template used in the implementation to create and add more parameters for the final training script. The default set of hyperparameters, either automatically or manually tuned, are listed in appendix A.3. If desired, the user can review the last used hyperparameters by checking *./learning/mairl.json*. Documentation for each parameter is available in the Imitation documentation or the Stable-Baselines3 documentation for the RL algorithms whose parameters are in the *rl* category of each hyperparameter file.

Important: *Look for the PPO algorithm and their corresponding parameters within the Stable-Baselines3 documentation.*

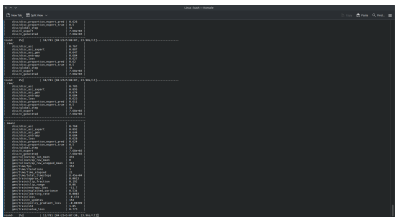
A.2.3.2 *Training*

Refer to appendix A.2.3.1 to configure the IRL mode in UE. Also, please follow the side notes with critical remarks to achieve a smooth experience without errors. When the user sets all settings correctly, such as training hyperparameters and displaying trajectories, they can click on the UE's *Play* button. Depending on the system and the training settings, training might last a few hours up to several weeks.

After hitting play, a second window will appear showing the current training status. UE's sole purpose from now on is to do and show the physics simulation. Figure A.9 shows both windows. While training, figure A.9b estimates the remaining training time. When performing hyperparameter tuning, estimate the total time by multiplying the expected time for an individual run by the number of runs.



(a) UE's environment window for physics calculation.



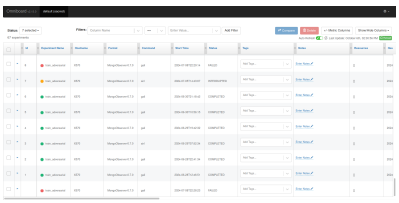
(b) Python's output window to report the current training status.

Figure A.9: After hitting UE's Play button, the training will start with two windows, as seen above.

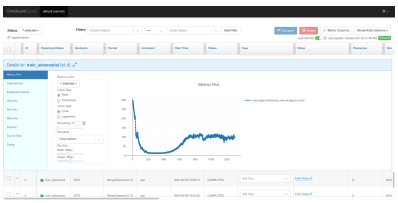
A.2.3.3 *Result*

The results and representation vary depending on the previous configuration, namely regular training and hyperparameter tuning.

IRL: The system loggs training results under their respective run ID directory in the *cout.txt* file. Find these directories in the project's directory *./learning/output/sacred/tune_adversarial*. Additionally, Omniboard allows analyzing all runs and creating plots for selected metrics. Access Omniboard in your web browser by clicking or copying the following link: `http://localhost:9000`. Also, see figure A.10 to view the interface and how to plot the metrics.



(a) Omniboard's experiment list.



(b) Viewing individual metric plots from experiment runs.

Figure A.10: Viewing sacred experiments is done with Omniboard and indicates how the run went.

HYPERPARAMETER TUNING: Other than IL training, the system will not create a log file of the algorithm's output because of long sessions and coincidentally large output files. Furthermore, using Omniboard as an analysis tool to review all runs is also impossible. As per the explanation of *Optimize Hyperparameters* in appendix A.2.3.1, the system will save all optimization results in *~/ray_results*. If the Optuna optimization succeeded, you find the best set of parameters in this directory.

A.3 HYPERPARAMETERS

Read the columns of all tables from left to right and correspond to the hierarchy in the JSON file. For example, the first entry in table A.1 means that the parameter *demo batch size* in the category *algorithm kwargs* has a value of 7680. If an entry divides the values with a comma, training uses different values for this attribute in different runs. A dash is a placeholder for the default value.

A.3.1 *Manually Tuned*

The tuned parameters were collected by testing various combinations of hyperparameters and referencing suitable parameters from the Imitation project for MuJoCo environments.

Hyperparameters for all environments			
algorithm kwargs	demo batch size		7680
	gen replay buffer capacity		7680
	n disc updates per round		8
reward	net kwargs	balancing value	1.0, 0.7, 0.5, 0.3
rl	batch size		7680
	rl kwargs	batch size	192
		clip range	0.08
		ent coef	0.022
		gamma	0.97
		learning rate	0.0003
		n epochs	14
		vf coef	0.1

Table A.1: A table showing all relevant hyperparameters used in the hyperparameter config file, manually tuned for IL training. These values are used in all environments.

A.3.2 *Automatically Tuned*

The manually tuned hyperparameters apply universally to all environments and learning algorithms. In contrast, the automatically tuned hyperparameters undergo individual training for each environment and learning algorithm combination. This approach should ensure the best hyperparameter set without manual interference. The following tables show the Optuna results. The results have been rounded to the sixth decimal place (if needed) to make them easier to read.

A.3.2.1 *GAIL*

Hyperparameters for Covering (GAIL)			
algorithm kwargs	demo batch size		32
	gen replay buffer capacity		-
	n disc updates per round		4
reward	net kwargs	balancing value	1.0, 0.7, 0.5, 0.3
rl	batch size		4096
	rl kwargs	batch size	128
		clip range	0.08
		ent coef	9.158107e-07
		gamma	0.996499
		learning rate	0.000429
		n epochs	5
		vf coef	0.00882

Table A.2: Automatically tuned hyperparameters for the GAIL covering environment.

Hyperparameters for Insert (GAIL)			
algorithm kwargs	demo batch size		512
	gen replay buffer capacity		-
	n disc updates per round		8
reward	net kwargs	balancing value	1.0, 0.7, 0.5, 0.3
rl	batch size		4096
	rl kwargs	batch size	16
		clip range	0.02
		ent coef	7.330827e-07
		gamma	0.989083
		learning rate	7.644344-05
		n epochs	12
		vf coef	0.137331

Table A.3: Automatically tuned hyperparameters for the GAIL insert environment.

Hyperparameters for Stacking (GAIL)			
algorithm kwargs	demo batch size		128
	gen replay buffer capacity		-
	n disc updates per round		8
reward	net kwargs	balancing value	1.0, 0.7, 0.5, 0.3
rl	batch size		8192
	rl kwargs	batch size	64
		clip range	0.2
		ent coef	1.137963e-06
		gamma	0.962156
		learning rate	5.346024e-05
		n epochs	7
		vf coef	0.171239

Table A.4: Automatically tuned hyperparameters for the GAIL stacking environment.

A.3.2.2 *AIRL*

Hyperparameters for Covering (AIRL)			
algorithm kwargs	demo batch size		128
	gen replay buffer capacity		-
	n disc updates per round		16
reward	net kwargs	balancing value	1.0, 0.7, 0.5, 0.3
rl	batch size		8192
	rl kwargs	batch size	512
		clip range	0.08
		ent coef	2.321705e-05
		gamma	0.976245
		learning rate	9.434924e-05
		n epochs	9
		vf coef	0.142388

Table A.5: Automatically tuned hyperparameters for the AIRL covering environment.

Hyperparameters for Insert (AIRL)			
algorithm kwargs	demo batch size		512
	gen replay buffer capacity		-
	n disc updates per round		4
reward	net kwargs	balancing value	1.0, 0.7, 0.5, 0.3
rl	batch size		8192
	rl kwargs	batch size	32
		clip range	0.04
		ent coef	9.415599e-05
		gamma	0.999833
		learning rate	4.664156e-05
		n epochs	15
		vf coef	0.197963

Table A.6: Automatically tuned hyperparameters for the AIRL insert environment.

Hyperparameters for Stacking (AIRL)			
algorithm kwargs	demo batch size		32
	gen replay buffer capacity		-
	n disc updates per round		4
reward	net kwargs	balancing value	1.0, 0.7, 0.5, 0.3
rl	batch size		8192
	rl kwargs	batch size	512
		clip range	0.2
		ent coef	1.996332e-05
		gamma	0.964847
		learning rate	0.008879
		n epochs	15
		vf coef	0.055766

Table A.7: Automatically tuned hyperparameters for the AIRL stacking environment.

A.4 NUTZUNG KI BASIERTE ANWENDUNGEN

Diese Abschlussarbeit hat sowohl Grammarly [30] zur Korrektur des geschriebenen Textes verwendet als auch für das Schreiben der Evaluations-Skripte zum Teil auf JetBrains AI Assistant [37] zurückgegriffen.

BIBLIOGRAPHY

[1] Pieter Abbeel and Andrew Y. Ng. "Apprenticeship learning via inverse reinforcement learning." In: *Proceedings of the Twenty-First International Conference on Machine Learning*. ICML '04. Banff, Alberta, Canada: Association for Computing Machinery, 2004, p. 1. ISBN: 1581138385. DOI: 10.1145/1015330.1015430.

[2] Navid Aghasadeghi and Timothy Bretl. "Maximum entropy inverse reinforcement learning in continuous state spaces with path integrals." In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. San Francisco, CA, USA: IEEE, Sept. 2011, pp. 1561–1566. DOI: 10.1109/IROS.2011.6094679.

[3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. *Optuna: A Next-generation Hyperparameter Optimization Framework*. 2019. arXiv: 1907.10902 [cs.LG].

[4] Paola Ardón, Èric Pairet, Katrin S. Lohan, Subramanian Ramamoorthy, and Ronald P. A. Petrick. *Affordances in Robotic Tasks – A Survey*. 2020. arXiv: 2004.07400 [cs.RO].

[5] Saurabh Arora and Prashant Doshi. "A survey of inverse reinforcement learning: Challenges, methods and progress." In: *Artificial Intelligence* 297 (2021), p. 103500. ISSN: 0004-3702. DOI: 10.1016/j.artint.2021.103500.

[6] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. "Deep Reinforcement Learning: A Brief Survey." In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38. DOI: 10.1109/MSP.2017.2743240.

[7] Stable Baselines3. *Reinforcement Learning Tips and Tricks*. Stable Baselines3. URL: https://stable-baselines3.readthedocs.io/en/master/guide/rl_tips.html#tips-and-tricks-when-creating-a-custom-environment (visited on 12/11/2024).

[8] Christian Baun. "Interprocess Communication / Interprozesskommunikation." In: *Operating Systems / Betriebssysteme : Bilingual Edition: English – German / Zweisprachige Ausgabe: Englisch – Deutsch*. Wiesbaden: Springer Fachmedien Wiesbaden, 2023, pp. 195–274. ISBN: 978-3-658-42230-1. DOI: 10.1007/978-3-658-42230-1_9.

[9] Mulcahy Brendan. *Learning Agents Introduction*. Unreal Engine. Mar. 31, 2023. URL: https://dev.epicgames.com/community/learning/tutorials/8OWY/unreal-engine-learning-agents-introduction (visited on 08/19/2024).

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG].

[11] Antonia L. Busse et al. *Approach*. CreaBots. URL: https://cgvr.cs.uni-bremen.de/teaching/studentprojects/creabots/approach.html (visited on 12/03/2024).

[12] Antonia L. Busse et al. *Home*. CreaBots. URL: https://cgvr.cs.uni-bremen.de/teaching/studentprojects/creabots/ (visited on 10/17/2024).

[13] Xiliang Chen, Lei Cao, Zongben Xu, Jun Lai, and Chenxi Li. "A study of Continuous maximum entropy Deep inverse Reinforcement learning." In: *Mathematical Problems in Engineering* 2019 (Apr. 2019), pp. 1–8. DOI: 10.1155/2019/4834516.

[14] Franck Djeumou, Christian Ellis, Murat Cubuktepe, Craig Lennon, and Ufuk Topcu. *Task-Guided IRL in POMDPs that Scales*. 2022. arXiv: 2301.01219 [cs.LG].

[15] Gymnasium Documentation. *Box2D*. Farama Foundation. URL: https://gymnasium.farama.org/environments/box2d/ (visited on 11/19/2024).

[16] Gymnasium Documentation. *Classic Control*. Farama Foundation. URL: https://gymnasium.farama.org/environments/classic_control/ (visited on 11/19/2024).

[17] Gymnasium Documentation. *MuJoCo*. Farama Foundation. URL: https://gymnasium.farama.org/environments/mujoco/ (visited on 11/19/2024).

[18] Jonatan S. Dyrstad, Elling Ruud Øye, Annette Stahl, and John Reidar Mathiassen. "Teaching a Robot to Grasp Real Fish by Imitation Learning from a Human Supervisor in Virtual Reality." In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Madrid, Spain: IEEE Press, 2018, 7185–7192. DOI: 10.1109/IROS.2018.8593954.

[19] Theresa Eimer, Marius Lindauer, and Roberta Raileanu. *Hyperparameters in Reinforcement Learning and How To Tune Them*. 2023. arXiv: 2306.01324 [cs.LG].

[20] Inc. Epic Games. *The most powerful real-time 3D creation tool*. Unreal Engine. URL: https://www.unrealengine.com/en-US (visited on 10/17/2024).

[21] Inc. Epic Games. *Unreal Engine Terminology*. Unreal Engine. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/unreal-engine-terminology?application_version=5.3 (visited on 10/17/2024).

[22] Jonas Eschmann. "Reward function design in reinforcement learning." In: *Reinforcement Learning Algorithms: Analysis and Applications*. Ed. by Boris Belousov, Hany Abdulsamad, Pascal Klink, Simone Parisi, and Jan Peters. Cham: Springer International Publishing, Jan. 1, 2021, pp. 25–33. ISBN: 978-3-030-41188-6. DOI: 10.1007/978-3-030-41188-6_3.

[23] Chelsea Finn, Sergey Levine, and Pieter Abbeel. *Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization*. 2016. arXiv: 1603.00448 [cs.LG].

[24] Justin Fu, Katie Luo, and Sergey Levine. *Learning Robust Rewards with Adversarial Inverse Reinforcement Learning*. 2018. arXiv: 1710.11248 [cs.LG].

[25] Getnamo. *TCP-Unreal: Convenience TCP wrapper for Unreal Engine*. GitHub. URL: https://github.com/getnamo/TCP-Unreal (visited on 10/16/2024).

[26] James J. Gibson. *The Senses Considered as Perceptual Systems*. Houghton Mifflin, 1966.

[27] James J. Gibson. *The ecological approach to visual Perception*. New York: Psychology Press, Nov. 2014. ISBN: 9781315740218. DOI: 10.4324/9781315740218.

[28] Adam Gleave, Mohammad Taufeeque, Juan Rocamonde, Erik Jenner, Steven H. Wang, Sam Toyer, Maximilian Ernestus, Nora Belrose, Scott Emmons, and Stuart Russell. *imitation: Clean Imitation Learning Implementations*. arXiv:2211.11972v1 [cs.LG]. 2022. arXiv: 2211.11972 [cs.LG].

[29] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML].

[30] Grammarly. *Grammarly: Free AI Writing Assistance*. Grammarly Inc. URL: https://www.grammarly.com/ (visited on 12/15/2024).

[31] Klaus Greff, Aaron Klein, Martin Chovanec, Frank Hutter, and Jürgen Schmidhuber. "The Sacred Infrastructure for Computational Research." In: *Proceedings of the 16th Python in Science Conference*. Ed. by Katy Huff, David Lippa, Dillon Niederhut, and M Pacer. 2017, pp. 49 –56. DOI: 10.25080/shinma-7f4c6e7-008.

[32] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. *Rainbow: Combining Improvements in Deep Reinforcement Learning*. 2017. arXiv: 1710.02298 [cs.AI].

[33] Jonathan Ho and Stefano Ermon. *Generative Adversarial Imitation Learning*. 2016. arXiv: 1606.03476 [cs.LG].

[34] Michael Hu. *The art of reinforcement learning*. Berkeley, CA: Apress, Jan. 1, 2023. ISBN: 978-1-4842-9606-6. DOI: 10.1007/978-1-4842-9606-6.

[35] Zhe Hu, Yu Zheng, and Jia Pan. "Grasping Living Objects With Adversarial Behaviors Using Inverse Reinforcement Learning." In: *Trans. Rob.* 39.2 (Apr. 2023), 1151–1163. ISSN: 1552-3098. DOI: 10.1109/TRO.2022.3226108.

[36] Alex Irpan. *Deep Reinforcement Learning Doesn't Work Yet*. https://www.alexirpan.com/2018/02/14/rl-hard.html. 2018.

[37] JETBRAINS. *JetBrains AI*. JetBrains s.r.o. URL: https://www.jetbrains.com/ai/ (visited on 12/15/2024).

[38] Lorenzo Jamone, Emre Ugur, Angelo Cangelosi, Luciano Fadiga, Alexandre Bernardino, Justus Piater, and José Santos-Victor. "Affordances in Psychology, Neuroscience, and Robotics: A Survey." In: *IEEE Transactions on Cognitive and Developmental Systems* 10.1 (2018), pp. 4–25. DOI: 10.1109/TCDS.2016.2594134.

[39] Matthew Johnson, Jeffrey Bradshaw, Paul J. Feltovich, Renia Jeffers, Hyuckchul Jung, and Andrzej Uszok. "A semantically rich policy based approach to robot control." In: *ICINCO-RA*. INSTICC Press, Jan. 2006, pp. 318–325.

[40] Mrinal Kalakrishnan, Peter Pastor, Ludovic Righetti, and Stefan Schaal. "Learning objective functions for manipulation." In: *2013 IEEE International Conference on Robotics and Automation*. Karlsruhe, Germany: IEEE, 2013, pp. 1331–1336. DOI: 10.1109/ICRA.2013.6630743.

[41] Krumiaa. *MindMaker: MindMaker UE4 Machine Learning Toolkit*. GitHub. URL: https://github.com/krumiaa/MindMaker (visited on 08/19/2024).

[42] DeepMind Technologies Limited. *Visualization - MUJOCO Documentation*. MuJoCo. URL: https://mujoco.readthedocs.io/en/stable/programming/visualization.html (visited on 11/29/2024).

[43] David Lindner, Andreas Krause, and Giorgia Ramponi. *Active Exploration for Inverse Reinforcement Learning*. 2023. arXiv: 2207.08645 [cs.LG].

[44] Shunyu Liu, Yunpeng Qing, Shuqi Xu, Hongyan Wu, Jiangtao Zhang, Jingyuan Cong, Tianhao Chen, Yunfu Liu, and Mingli Song. *Curricular Subgoals for Inverse Reinforcement Learning*. 2023. arXiv: 2306.08232 [cs.LG].

[45] Manuel Lopes, Francisco S. Melo, and Luis Montesano. "Affordance-based imitation learning in robots." In: *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*. San Diego, CA, USA: IEEE, 2007, pp. 1015–1021. DOI: 10.1109/IROS.2007.4399517.

[46] Mayank Mittal et al. "Orbit: A Unified Simulation Framework for Interactive Robot Learning Environments." In: *IEEE Robotics and Automation Letters* 8.6 (June 2023), 3740–3747. ISSN: 2377-3774. DOI: 10.1109/lra.2023.3270034.

[47] Mongodb. *mongo: The MongoDB Database*. GitHub. URL: https://github.com/mongodb/mongo (visited on 10/16/2024).

[48] Luis Montesano, Manuel Lopes, Alexandre Bernardino, and Jose Santos-Victor. "Affordances, development and imitation." In: *2007 IEEE 6th International Conference on Development and Learning*. London, UK: IEEE, 2007, pp. 270–275. DOI: 10.1109/DEVLRN.2007.4354054.

[49] NaturalPoint. *Motion capture systems*. OptiTrack. URL: https://www.optitrack.com/ (visited on 11/27/2024).

[50] Martin L. Puterman. *Markov Decision processes*. Hoboken, New Jersey: John Wiley & Sons, Apr. 15, 1994. DOI: 10.1002/9780470316887.

[51] Qzed. *irl-maxent: Maximum Entropy and Maximum Causal Entropy Inverse Reinforcement Learning Implementation in Python*. GitHub. URL: https://github.com/qzed/irl-maxent (visited on 11/27/2024).

[52] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. "Stable-Baselines3: Reliable Reinforcement Learning Implementations." In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[53] Rafael Ris-Ala. *Fundamentals of Reinforcement Learning*. Cham: Springer Nature Switzerland, 2023. ISBN: 978-3-031-37345-9. DOI: 10.1007/978-3-031-37345-9.

[54] Robcog-Iai. *RobCoG: Robot Commonsense Games*. GitHub. URL: https://github.com/robcog-iai/RobCoG (visited on 11/27/2024).

[55] Robcog-Iai. *USemLog: Semantic logger plugin for Unreal Engine*. GitHub. URL: https://github.com/robcog-iai/USemLog (visited on 10/16/2024).

[56] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. *A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning*. 2011. arXiv: 1011.0686 [cs.LG].

[57] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, Global Edition*. Pearson Deutschland, 2021, p. 1168. ISBN: 9781292401133.

[58] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[59] David Silver, Satinder Singh, Doina Precup, and Richard S. Sutton. "Reward is enough." In: *Artificial Intelligence* 299 (2021), p. 103535. ISSN: 0004-3702. DOI: 10.1016/j.artint.2021.103535.

[60] SkytAsul. *DeformableSimulation: A research project on linking a haptic device, a VR headset and a physics simulation to simulate real-time touch of a deformable object.* GitHub. URL: https://github.com/SkytAsul/DeformableSimulation (visited on 11/29/2024).

[61] Christopher Stanton and Jeff Clune. *Deep Curiosity Search: Intra-Life Exploration Can Improve Performance on Challenging Deep Reinforcement Learning Problems*. 2018. arXiv: 1806.00553 [cs.AI].

[62] J. K. Terry et al. *PettingZoo: Gym for Multi-Agent Reinforcement Learning*. 2021. arXiv: 2009.14471 [cs.LG].

[63] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control." In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[64] Mark Towers et al. *Gymnasium: A Standard Interface for Reinforcement Learning Environments*. 2024. arXiv: 2407.17032 [cs.LG].

[65] Alan Turing. "Intelligent Machinery (1948)." In: *The Essential Turing*. Oxford University Press, Sept. 2004. ISBN: 9780198250791. DOI: 10.1093/oso/9780198250791.003.0016.

[66] Vivekratnavel. *omniboard: Web-based dashboard for Sacred*. GitHub. URL: https://github.com/vivekratnavel/omniboard (visited on 10/16/2024).

[67] Han Wang, Youfang Lin, Sheng Han, and Kai Lv. "Offline Reinforcement Learning with Diffusion-Based Behavior Cloning Term." In: *Knowledge Science, Engineering and Management: 16th International Conference, KSEM 2023, Guangzhou, China, August 16–18, 2023, Proceedings, Part IV*. Guangzhou, China: Springer Nature Switzerland, 2023, 267–278. ISBN: 978-3-031-40291-3. DOI: 10.1007/978-3-031-40292-0_22.

[68] Phil Winder. *Reinforcement learning*. O'Reilly. Nov. 2020. URL: https://learning.oreilly.com/library/view/reinforcement-learning/9781492072386/ (visited on 10/27/2024).

[69] Maryam Zare, Parham M. Kebria, Abbas Khosravi, and Saeid Nahavandi. *A Survey of Imitation Learning: Algorithms, Recent Developments, and Challenges*. 2023. arXiv: 2309.02473 [cs.LG].

[70] Tianhao Zhang, Zoe McCarthy, Owen Jow, Dennis Lee, Xi Chen, Ken Goldberg, and Pieter Abbeel. "Deep Imitation Learning for Complex Manipulation Tasks from Virtual Reality Teleoperation." In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. Brisbane, Australia: IEEE Press, 2018, 1–8. DOI: 10.1109/ICRA.2018.8461249.

[71] Boyuan Zheng, Sunny Verma, Jianlong Zhou, Ivor Tsang, and Fang Chen. *Imitation Learning: Progress, Taxonomies and Challenges*. 2022. arXiv: 2106.12177 [cs.LG].

[72] Yuke Zhu et al. *Reinforcement and Imitation Learning for Diverse Visuomotor Skills*. 2018. arXiv: 1802.09564 [cs.RO].

[73] Brian D. Ziebart, Andrew L. Maas, J. Andrew Bagnell, and Anind K. Dey. "Maximum entropy inverse reinforcement learning." In: *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*. AAAI'08. Chicago, Illinois: AAAI Press, 2008, 1433–1438. ISBN: 9781577353683.

[74] Dorian Šuc and Ivan Bratko. "Problem Decomposition for Behavioural Cloning." In: *Machine Learning: ECML 2000*. Ed. by Ramon López de Mántarasand and Enric Plaza. Berlin, Heidelberg: Springer Berlin Heidelberg, Jan. 1, 2000, pp. 382–391. ISBN: 978-3-540-45164-8. DOI: 10.1007/3-540-45164-1_39.