FH University of Applied Sciences
TECHNIKUM WIEN

MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program AI-Engineering

Unity ML Agents vs. Godot RL Agents Comparing the performance of two reinforcement learning frameworks

By: Florentin Luca Rieger, BSc

Student Number: 2110585030

Supervisors: FH-Prof. Dipl.-Ing. Alexander Nimmervoll
Markus Petz, MSc

Vienna, September 24, 2023

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, September 24, 2023 Signature

Kurzfassung

In den letzten Jahren sind viele Reinforcement Learning Tools entstanden. Von Libraries, die es Software Endtwickler:innen und ML Forscher:innen ermöglichen, bereits implementierte RL Algorithmen zu verwenden, bis zu ganzheitlichen Frameworks, die es ermöglichen, Simulationen und Reinforcement Learning Probleme zu kreieren, Agenten zu trainieren und den Trainingsprozess zu visualisieren. Diese Arbeit vergleicht zwei bestehende Reinforcement Learning Frameworks: Unity ML Agents Toolkit und Godot RL Agents in Bezug auf deren Lern- und Rechenlaufzeit-Performance, um Forscher:innen und Spiele Entwickler:innen eine Hilfestellung zu liefern, welches Tool am besten ihre Anforderungen erfüllt.

Schlagworte: AI, Reinforcement Learning, Deep Learning, Game Engine, Performance

Abstract

Over the last years, a lot of different reinforcement learning tools have been created. From libraries, which provide software developers and ML researchers with out-of-the-box RL algorithms, to full-blown frameworks, which allow creating simulations and reinforcement learning problems, training agents and visualizing the training process. This work compares two of those reinforcement learning frameworks: Unity ML Agents Toolkit and Godot RL Agents in terms of their learning performance and computational runtime performance to give researchers and game developers some insights on which tool might fit their needs best.

Keywords: AI, Reinforcement Learning, Deep Learning, Game Engine, Performance

Acknowledgements

Thank you, Nina for giving me the courage to start this study in the first place. Thank you, Marek, Judith, Maria, Marlene, Franci & Hannah for letting me occupy your coffee shop during the creation of this work. Big thanks to Edward Beeching and Ivan Dodic for helping me out with questions and issues related to Godot RL Agents. Thank you, Aleksei Petrenko for supporting me with open questions regarding Sample Factory and for giving me the essential hint about using wall time over time-steps as stopping condition. Thank you, Alexander Nimmervoll for all the valuable feedback during the creation of this work. Thank you, Stefan for proofreading!! Thank you to my family for always being there! <3

Contents

1 Introduction

1.1 Background

Artificial intelligence (AI), especially in the field of machine learning (ML), has made huge advances in the last 10 years. From large language models (LLMs) [1], self-driving cars [2], prediction of protein structures in the field of medicine and biology [3], to high performing agents in computer games [4], artificial intelligence has had major impacts in almost every aspect of life. The introduction of deep artificial neural network (ANN) and their combination with other machine learning algorithms has made endeavours possible which have been thought of as impossible previously [5].

One of those advancements is deep reinforcement learning (DRL) [6], which is a combination of reinforcement learning (RL) algorithms and deep artificial neural networks. These DRL algorithms are self-learning, sequential decision-making systems and are most commonly known for performing tasks in the fields of robotics, self-driving cars, healthcare, finance, advertisement, large language models [7] and video games [8].

1.2 Motivation

RL algorithms are self-learning systems which learn from experience. Those experiences and observations are collected by the system while interacting with the environment [6]. Compared to supervised learning algorithms, which need a high amount of pre-collected, cleaned data to excel, reinforcement learning algorithms generate their own training data from interacting with the environment and can directly learn from those observations. This makes reinforcement learning algorithms especially suitable in fields and tasks where there is not much initial training data [9] or where the environment and data are constantly changing and sequential decision-making is needed.

For example in the field of robotics, DRL is deployed, so robots autonomously learn the skills needed to move or complete specific tasks in the most efficient way [10]. Video games are often used for benchmarking DRL algorithms, as modern video games provide complex environments, which only a few algorithms, like deep reinforcement learning, seem to master well. Success stories like DeepMind's Alpha Zero, an AI which taught itself to play the board games

Go, chess, and Shogi without any human knowledge and managed to beat a world champion program in each [8], or Deep Mind's AlphaStar, which has been the first AI to reach the top league of StartCraft 2 [11], showcase what deep reinforcement learning is already capable of.

Furthermore, video games or simulations are used to train RL agents in a virtual environment. Training in a simulation is commonly preferred as opposed to training an agent directly in the real world, as gathering training data and making errors in a virtual environment is much faster and less costly than doing so in a real-world setting [12]. After initial training, the pre-trained agent can then be deployed in the real world and continue learning in the actual intended environment [13].

Apart from benchmarking and training DRL algorithms in video games and simulations, those algorithms can also be beneficial for video games themselves. A trained DRL agent can be added as a gameplay mechanic, e.g. in the form of intelligent enemies or non-playable characters cooperating with the player. Intelligent agents have the potential to increase the immersiveness of video games even further and therefore improve the overall quality of the game.

Last but not least, trained DRL agents can be used for automated gameplay testing. Video game testing is a hard to automatize task, due to the sheer complexity and endless possibilities of video games and their environments. A trained DRL agent can make it possible to automate certain tasks of the video game's quality assurance (QA) process [14]. For example, pre-trained DRL agents could be part of the continuous integration process and be used for automated regression testing. This process would save the video game's QA team from a lot of redundant testing, making it possible for them to focus on newer features while feeling confident that old features and behaviours have not been broken. Furthermore, DRL agents might improve the overall quality of the video game by revealing additional exploits or bugs, which would have gone unnoticed otherwise. Finally, an agent might play the game a bit differently than a human tester and vice versa. Together they can catch more bugs and make the game more robust in the end.

1.3 Problem

Over the last few years, many tools, libraries and frameworks surfaced, which make it easier and faster to simulate, train and benchmark reinforcement learning agents. There are game engines like Unity [1], Godot [2], or Unreal Engine [3], which make it possible to create simulations and virtual environments for reinforcement learning problems. The game engines Unity and Godot

[1] https://unity.com/
[2] https://godotengine.org/
[3] https://www.unrealengine.com

offer the frameworks Unity ML-Agents Toolkit [15] and Godot RL Agents [16] respectively, which provide reinforcement learning algorithms, data visualization tools and a python API to train and evaluate RL agents directly in the engine or in an executable build of the environment. OpenAI Gym [17] provides a standard API for communication between reinforcement learning algorithms and the training environment allowing for a more streamlined developer experience and making it possible to easily compare the algorithms. Reinforcement learning libraries like Stable Baselines3 [18], Ray RLlib [19], Sample Factory [20] and CleanRL [21] provide their own open source implementations of popular reinforcement learning algorithms and utilize computational performance optimisations like parallelism. Instead of developing those algorithms from scratch, these libraries and their associated algorithms can be readily employed for training RL agents.

While having many options and possibilities to train RL agents is advantageous, it also makes choosing and picking the right framework for the right problem harder. One of the reasons is that even if those frameworks implement the same algorithms, the actual learning performance of the agents might differ a lot and may be more or less robust to changes in the hyperparameter configuration, environment or reward functions. This might be due to differences in the detailed implementation of the algorithms or due to the default configuration of the algorithm's hyperparameters. Furthermore, the computational runtime performance of those frameworks can vary significantly. While one framework might achieve a result in 20 minutes, another framework may achieve a comparable result in only 4 minutes. Those differences are dependent on the implementation details and how well those libraries are utilizing computational performance optimizations and the underlying hardware. Working with high-performance optimized algorithms not only allows for faster training, iteration and the possibility to test more variations of hyperparameters or environment setups, but it also allows persons with weaker hardware to train machine learning models, making the field more inclusive and accessible for a broader audience of developers across the world. Furthermore, high-performing algorithms and their corresponding reduced wall-clock runtimes, at times also need less energy and therefore less electricity for the same results [22], making them relevant from an environmental point of view, where reducing energy and the corresponding carbon footprint are important goals.

1.4 Research Question

Considering the points mentioned above, this work aims to answer the question about the differences between some of those frameworks in terms of learning and computational runtime performance. More specifically, this work focuses on the frameworks Unity ML-Agents Toolkit and Godot RL Agents rather than on individual reinforcement learning libraries as both frameworks offer an all-in-one solution for reinforcement learning problems. There is also a reinforce-

ment learning framework for Unreal Engine called Learning Agents [4], but it seems to offer, at the time of this writing, fewer possibilities compared to Unity ML-Agents and Godot RL Agents and is therefore not taken further into account. Both Unity ML-Agents and Godot RL Agents allow users to create a complex simulation environment for a reinforcement learning problem in their corresponding game engines, to directly train the agent in the created environment by offering a python API for communicating between the environment and the provided reinforcement learning algorithms and to visualize the results in data visualization tools, making it easy to quickly quantify and evaluate the agent's performance. Furthermore, Unity ML Agents and Godot RL Agents provide the opportunity to deploy the trained agent in an actual game thereby expanding the tools' usefulness beyond research and algorithmic benchmarking by also making it possible to integrate the trained agent as actual game content or as an additional tool for automated testing.

The resulting research question is formulated as: "How do the reinforcement learning frameworks Unity ML-Agents Toolkit and Godot RL Agents differ in terms of learning performance and computational runtime performance?"

1.5 Methods Overview

In order to facilitate a meaningful comparison between the two different reinforcement learning frameworks, it is essential to ensure that certain conditions are the same for both agents and environments. First off, it is important that the agents are being trained on the same hardware to be able to compare the computational runtime performance of the frameworks. Furthermore, it is essential that the agents are trained in comparable environments and that the Markov decision process (MDP), including the defined rewards, possible observations and actions of the agent, are the same in both reinforcement learning problems to be able to grant a relevant comparison regarding the learning performance of the agents.

In the interest of achieving these goals, a small identical game/simulation is built in Unity and Godot. The Unity ML agent and Godot RL agent are then being trained in their respective environments on the same hardware and the data of learning and runtime performance is being collected during training. While Unity ML-Agents offers their custom reinforcement learning algorithms out of the box, Godot RL Agents offers the option to choose between four different RL backends (Stable Baselines3, Ray RLlib, Sample Factory and CleanRL). This work will compare the performance of Unity ML Agents with the performance of Godot RL Agents and its four different RL backends, resulting in a total comparison of five different implementations of reinforcement learning algorithms.

[4] https://dev.epicgames.com/community/learning/tutorials/8OWY/unreal-engine-learning-agents-introduction

The RL algorithm which will be used in both frameworks and the respective RL libraries is Proximal Policy Optimization (PPO) [23], one of the most popular and widely used RL algorithms. The frameworks will be compared under different settings. Initially, the frameworks will be compared using their default hyperparameter configuration. Subsequently, the frameworks will be compared with a shared configuration of hyperparameters, which aims to be as similar as possible. Each trial for each framework will be run 5 times and averaged over the 5 runs with a fixed set of seeds, as was done in previous work on RL evaluation [24], to make sure that potential differences are not just due to randomness. The data will then be visualized and compared in the form of graphs and the results and findings will be discussed.

1.6 Expected Results

The author expects that there are substantial differences between the performance of Unity ML Agents and Godot RL Agents and their implementations of reinforcement learning algorithms with their default settings of hyperparameters. Furthermore, the author suspects that these differences will get smaller the more similar these settings become. Finally, it is to be expected that the runtime performance of the different algorithms might vary significantly even when the same configuration of hyperparameters is chosen.

2 Reinforcement Learning

The goal of this work is to compare two reinforcement learning frameworks with respect to their learning and runtime performance. The following chapter describes the definition of reinforcement learning and its individual components and algorithms. It builds up a fundamental understanding of the main concepts of reinforcement learning and finishes with a description of PPO, one of the most popular RL algorithms nowadays, which is also used in the experiments of this work. Finally, the used RL frameworks, Unity ML Agents and Godot RL Agents and its respective reinforcement learning libraries are getting concisely described.

2.1 Definition

Fundamentally, reinforcement learning is the science of decision-making [25]. As such, it is part of a lot of different scientific fields, namely: Computer Science, Engineering, Mathematics, Neuroscience, Psychology and Economics. Within the realm of Computer Science, reinforcement learning falls under the broader category of machine learning, which can be subdivided

into three primary branches: Supervised Learning, Unsupervised Learning, and Reinforcement Learning. Reinforcement learning is different from supervised learning in that it lacks a supervisor but relies solely on a reward signal which might be delayed and not instantaneous. Furthermore, the collected data is not i.i.d (independent and identically distributed) but dependent on time and by the actions of the agent. In the following sections, the different aspects and components of reinforcement learning will be described in greater detail.

2.1.1 The Reinforcement Learning Problem

The reinforcement learning problem consists of a RL agent, which tries to maximize its expected rewards (the most reward it can obtain from the environment) by optimizing its sequential decision-making by following a learned suboptimal or optimal policy. In order to accomplish this, the reinforcement learning problem has to be formulated as a MDP [6].

2.2 Markov Decision Process

A Markov decision process (MDP) [6] is a model and a way to formally describe an environment for reinforcement learning problems. A MDP consists of an agent and an environment in which the agent takes an action A_t at each time-step t and observes the next state S_t and reward R_t, generated from the environment and its dynamics. The agent's objective is to maximize the accumulated reward. The following sections describe the MDP and its individual components more formally.

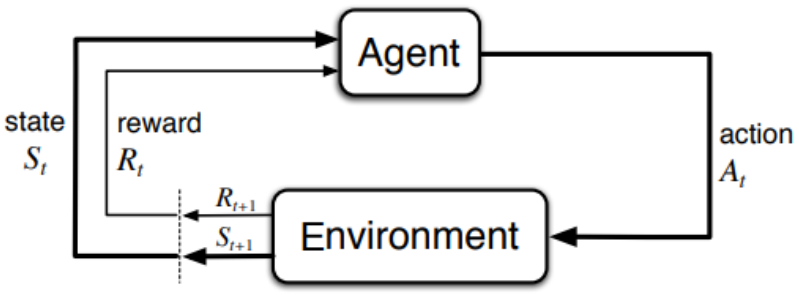

Figure 1: Markov Decision Process (MDP) [6]

2.2.1 Markov State

The Markov state or information state is a requirement to formulate a problem as Markov decision process. It is defined by the Markov property which defines the possibility to derive a future state solely from the current state, without the need to remember all other previous states [6].

In other words, the Markov state contains all the needed information from history in the present state.

$$\mathbb{P}(S_{t+1}|S_t) = \mathbb{P}(S_{t+1}|S_t, S_0, \ldots, S_t) \tag{1}$$

The Markov property provides the opportunity to neglect the history and only store the current state, thereby making the whole problem more memory efficient and easier to compute.

2.2.2 Markov Process

A Markov process, also known as Markov chain, is a sequence of random states with Markov property. The Markov process can be described as a tuple (S, P), where S is a set of states and P is a state transition probability matrix described as:

$$P_{ss'} = \mathbb{P}(S_{t+1} = s'|S_t = s) \tag{2}$$

A Markov process is already capable of describing the dynamics of an environment. For instance, it could describe that starting from state 0 with probability x, the agent will end up in state 1 and with probability y it will end up in state 2.

2.2.3 Markov Reward Process

By adding two more parameters to the equation, the reward function R and the discount factor $\gamma \in [0, 1]$, the Markov process becomes a Markov reward process (S, P, R, γ). The reward function R_s adds a value judgement about each state and the discount factor γ influences how much emphasis lies on rewards in the future.

$$R_s = \mathbb{E}(R_{t+1}|S_t = s) \tag{3}$$

The agent aims to maximize the return G_t, which is the total discounted reward from time-step t, by utilizing the state value function $v(s)$ 5, which itself reports the expected return starting from state s.

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4}$$

$$v(s) = \mathbb{E}(G_t|S_t = s) \tag{5}$$

The value function $v(s)$ of the Markov reward process can be recursively decomposed as a Bellman equation in the following form:

$$
\begin{aligned}
v(s) &= \mathbb{E}(G_t|S_t = s) \\
&= \mathbb{E}(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots |S_t = s) \\
&= \mathbb{E}(R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \ldots)|S_t = s) \\
&= \mathbb{E}(R_{t+1} + \gamma G_{t+1}|S_t = s) \\
&= \mathbb{E}(R_{t+1} + \gamma v(S_{t+1})|S_t = s)
\end{aligned} \tag{6}
$$

2.2.4 Markov Decision Process

The Markov decision process is a Markov reward process, but with the additional complexity of decisions or so-called actions (S, A, P, R, γ), where A is a set of actions/decisions. The state transition probability matrix gets extended to a state-action transition probability matrix $P^a_{ss'}$

$$
P^a_{ss'} = \mathbb{P}(S_{t+1} = s'|S_t = s, A_t = a) \tag{7}
$$

and the reward function R^a_s also takes the additional complexity of the action parameter into account.

$$
R^a_s = \mathbb{E}(R_{t+1}|S_t = s, A_t = a) \tag{8}
$$

2.3 Policies & Value Functions

Policies and value functions are fundamental concepts of reinforcement learning algorithms as they enable the application of evaluation and control to the reinforcement learning problem. In the following sections, policies and value functions are described in greater detail.

2.3.1 Policies

A policy π is a distribution over actions a given state s.

$$
\pi(a|s) = \mathbb{P}(A_t = a|S_t = s) \tag{9}
$$

In other words, a policy is a mapping of states and actions. A policy $\pi(a|s)$ defines the probabilities of the agent choosing a specific action a when being in state s. The stochastic property of a policy allows to balance the exploration-exploitation trade-off, further described in section 2.6.1. With the provided MDP, an agent has the capability to maximize the expected future reward under a policy π.

2.3.2 Value Functions

Value functions enable to evaluate the quality of states and actions. The two most common value functions are the state value function $v_\pi(s)$ and the action value function $q_\pi(s, a)$, which are getting described in the following sections.

State Value Function

The state value function $v_\pi(s)$ of an MDP is the expected return (sum of rewards) starting from state s, and then following policy π.

$$v_\pi(s) = \mathbb{E}_\pi(G_t|S_t = s) \tag{10}$$

The state value function can be decomposed into the following Bellman equation:

$$v_\pi(s) = \mathbb{E}_\pi(R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s) \tag{11}$$

Action Value Function

The action value function $q_\pi(s, a)$ is the expected return starting from state s, taking action a, and from there following policy π.

$$q_\pi(s, a) = \mathbb{E}_\pi(G_t|S_t = s, A_t = a) \tag{12}$$

Like the state value function, the action value function can be decomposed as a Bellman equation:

$$q_\pi(s, a) = \mathbb{E}_\pi(R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a) \tag{13}$$

2.3.3 Bellman Expectation Equation

The state value function $v_\pi(s)$ and the action value function $q_\pi(s, a)$ can be represented as a Bellman expectation equation in order to effectively solve them. $v_\pi(s)$ can be expressed in the following form to describe how it relates to $q_\pi(s, a)$

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) q_\pi(s, a) \tag{14}$$

Additionally, the action value function $q_\pi(s, a)$ can be formulated in the following manner to describe its relationship to the state value function $v_\pi(s)$.

$$q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \tag{15}$$

Under that definition, both functions can be further extended to the following form:

$$v_\pi(s) = \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')) \tag{16}$$

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s',a') \tag{17}$$

This description presents a system that can be directly solved with a linear equation.

2.3.4 Optimal Value Function

The agent's ultimate objective is to maximize the sum of expected rewards. In order to achieve this, it needs to utilize the optimal state value function $v_*(s)$ and/or the optimal action value function $q_*(s,a)$.

The optimal state value function $v_*(s)$ is the maximum value function overall policies, or put differently, it is the state value function, which reports the maximum reward that can be extracted from the system, starting from state s.

$$v_*(s) = \max_\pi v_\pi(s) \tag{18}$$

The optimal action value function $q_*(s,a)$ is the maximum action value function overall policies, or in other words, it is the action value function, which reports the maximum reward that can be retrieved from the system, starting from state s and taking action a.

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{19}$$

The optimal action value function $q_*(s,a)$ offers the agent at least one optimal policy, thereby telling the agent everything it needs to know to behave optimally in the system and maximize the expected return.

2.3.5 Optimal Policy

For any MDP there exists at least one optimal policy, which is better or equal to all other policies $\pi_* \geq \pi, \forall \pi$. A policy is better than another policy $\pi \geq \pi'$, if its state value function is better than the state value function of the other policy for all states $v_\pi(s) \geq v_{\pi'}(s), \forall s$. As a result, value functions are an essential component for finding the optimal policy.

Furthermore, the following holds true for optimal policies:

- All optimal policies achieve the optimal state value function, $v_{\pi_*}(s) = v_*(s)$
- All optimal policies achieve the optimal action value function, $q_{\pi_*}(s,a) = q_*(s,a)$

Given the optimal action value function $q_*(s,a)$, the optimal policy can be derived by taking the $argmax$ over $q_*(s,a)$.

$$\pi_*(a,s) = \begin{cases} 1 & \arg\max_{a \in A} q_*(s,a) \\ 0 & otherwise \end{cases}$$

2.3.6 Bellman Optimality Equation

In order to efficiently obtain q_* and the optimal policy π_*, the Bellman optimality equation is needed. It is expressed in the following form for the optimal state value function v_* and the optimal action value function q_*:

$$\begin{aligned} v_*(s) &= \max_a q_*(s,a) \\ &= \max_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \end{aligned} \tag{20}$$

$$\begin{aligned} q_*(s,a) &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \\ &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s',a') \end{aligned} \tag{21}$$

The Bellman optimality equation describes a system, which can actually solve the MDP by computing the optimal action value function and thereby the optimal policy. It is, however, a non-linear equation (because of the introduced max), which cannot be solved directly anymore. Nonetheless, it can be solved using iterative solution methods, like Value Iteration, Policy Iteration, Q-Learning and SARSA, which will be described in greater detail in the following sections.

2.4 Dynamic Programming

Dynamic programming within the context of reinforcement learning serves the purpose of planning, which can be understood as prediction and control [6]. Prediction is responsible for evaluating a given policy and control focuses on finding the optimal policy and maximising the expected return. Considering the dynamics and rewards of the environment are already known, prediction can use algorithms like iterative policy evaluation and control can utilize algorithms like policy iteration or value iteration. The subsequent sections delve more into the intricacies of these concepts and algorithms.

2.4.1 Policy Evaluation

For an agent to learn, it is essential to know how well the current policy behaves compared to other policies. Policy evaluation is a method which reports these metrics via the state value

function [6]. Policy evaluation is being achieved by applying the Bellman expectation equation iteratively, via a one-step look ahead for every state. Policy evaluation will in the end converge to the true state value function.

$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s')) \tag{22}$$

An interesting property of this process is that even though only a specific policy gets evaluated, policy evaluation can be used to find a better policy which directly leads to the concept of policy iteration.

2.4.2 Policy Iteration

Policy iteration [6] is part of the control division of planning and aims to find the optimal policy for a given reinforcement learning problem. Policy iteration does this in two steps:

1. Policy iteration evaluates the policy π via policy evaluation which returns a state value function $v_\pi(s)$ for the given policy π.

$$v_\pi(s) = \mathbb{E}_\pi(R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s) \tag{23}$$

2. Policy iteration then improves the policy π' by acting greedily on the retrieved state value function v_π.

$$\pi' = greedy(v_\pi) \tag{24}$$

   Where acting greedily means taking the $argmax$ over the current value function $q_\pi(s, a)$.

$$\pi'(s) = \arg\max_{a \in A} q_\pi(s, a) \tag{25}$$

These two steps, consisting of evaluation and improvement, are usually done in an iterative fashion in which a policy gets evaluated and a better policy gets derived from that evaluation. This process is then iteratively repeated until the optimal policy π_* is found.

2.4.3 Modified Policy Iteration

It can take a lot of steps until policy iteration convergence, even though the optimal policy might have been already found. A modified version of policy iteration takes this problem into account by adding a stopping condition. There are multiple options for stopping conditions. The stopping condition can be a quantity ϵ, which adds some bounds to the convergence and the algorithm stops once it is within those bounds. Another stopping condition may be that the algorithm stops after running k steps of iterative policy evaluation. A special case of the last stopping condition is $k = 1$, which stops already after one step of policy evaluation. This method is called value iteration [6] and will be described in greater detail in the next section.

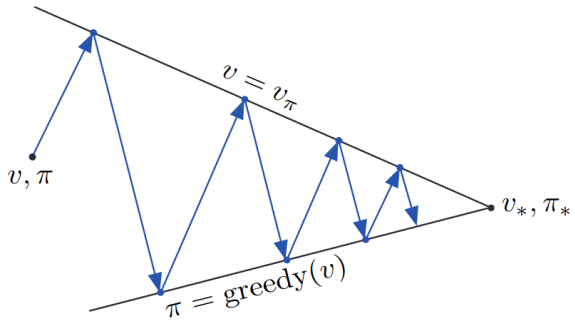

Figure 2: Policy Iteration Converges To The Optimal Policy & Optimal Value Function [6]

2.4.4 Value Iteration

Like in policy iteration, value iteration also aims to find an optimal policy π_* [6]. Unlike policy iteration, value iteration uses the Bellman optimality equation rather than the Bellman expectation equation. It does this also in an iterative manner but updates the value function every step. Unlike policy iteration, value iteration uses no explicit policy but rather improves the value function directly in the value space.

$$v_{k+1}(s) = \max_{a \in A}(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s')) \tag{26}$$

2.5 Model Free Prediction

So far the described methods assumed the MDP is completely known. In case the environment dynamics, transitions and rewards are not or only partially known, a different approach has to be chosen to solve a MDP. Model free methods allow for solving those problems by sampling experience. The only requirement is that every state gets visited often enough, so that the estimated value function approaches the correct value function for that particular state under a given policy (law of large numbers). There are two main ideas to address the concept of model free prediction: Monte Carlo (MC) learning and temporal difference (TD) learning.

2.5.1 Monte Carlo Learning

Monte Carlo learning [6] only works for episodic tasks, which means tasks that have a clear terminal state. In MC learning the agent collects an entire set of experiences $S_1, A_1, R_2, \ldots, S_k$ under a policy π, until the end of an episode. The goal is to learn the value function V_π under policy π. MC learning utilizes the empirical mean return from the sampled experiences instead of the expected return for policy evaluation: $V_\pi = \mathbb{E}_\pi(G_t|S_t = s)$. There are two options on how

MC learning calculates the empirical mean: first visit Monte Carlo policy evaluation and every visit Monte Carlo policy evaluation.

First Visit Monte Carlo Policy Evaluation

In first visit Monte Carlo policy evaluation, a counter $N(s)$ gets incremented the first time the state s gets visited during an episode. A total return $S(s)$ for state s is calculated over all episodes and the value function $V(s)$ is estimated by the mean return: $V(s) = S(s)/N(s)$.

Every Visit Monte Carlo Policy Evaluation

In every visit Monte Carlo policy evaluation, a counter $N(s)$ gets incremented every time the state s gets visited during an episode. The rest of the algorithm is the same to the first visit approach: $V(s) = S(s)/N(s)$.

Incremental Monte Carlo Updates

Utilizing the concept of incremental mean $\mu_k = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$, the MC learning algorithm can be rewritten in the form of incremental Monte Carlo updates:

$$V(S_t) = V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)) \tag{27}$$

For non-stationary problems it can be rewritten with a running mean:

$$V(S_t) = V(S_t) + \alpha(G_t - V(S_t)) \tag{28}$$

2.5.2 Temporal Difference Learning

Like MC learning, temporal difference (TD) learning is also model free, learns directly from episodes of experience and its goal is to estimate the value function $V(S_t)$ under given policy π. Unlike MC Learning, TD learns from incomplete episodes by bootstrapping and estimating the expected return. As a result, TD learning does not need to complete the whole episode to update its estimates and is also suitable for incomplete experience sequences and continuous, non-terminating environments [6]. The upcoming sections illustrate different variants of TD learning.

TD(0)

The simplest form of temporal difference learning is $TD(0)$, in which the agent only looks one step forward and estimates from that point onward the rest of the return:

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{29}$$

where $R_{t+1} + \gamma V(S_{t+1})$ is the TD target and $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is the TD error.

TD(n)

$TD(n)$ consists of the idea that the agent can take a predefined number of look-ahead steps n. On one side of the spectrum, $TD(0)$ is a special case of $TD(n)$ in which $n = 1$. On the other side of the spectrum, where $n = \infty$, $TD(n)$ is MC learning. The general form of $TD(n)$ is:

$$\begin{aligned} G_t^{(n)} &= R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) \\ V(S_t) &= V(S_t) + \alpha(G_t^{(n)} - V(S_t)) \end{aligned} \tag{30}$$

TD(λ)

The chosen number of steps n highly affects the quality of results of $TD(n)$ (as can be seen in Figure 3). $TD(\lambda)$ takes this into account by introducing the constant λ (a quantity between 0 and 1) which incorporates multiple different step sizes, thereby making the algorithm more robust.

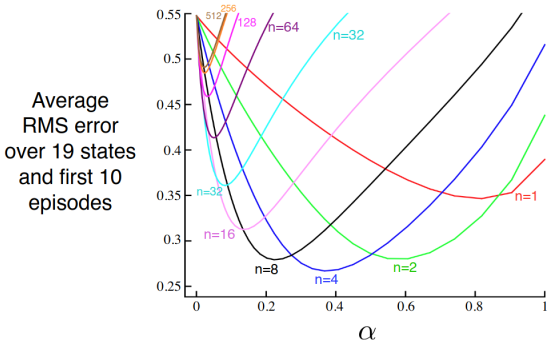

Figure 3: RMS Error of $TD(n)$ with different step sizes n [6]

$$\begin{aligned} G_t^{\lambda} &= (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \\ V(S_t) &= V(S_t) + \alpha(G_t^{\lambda} - V(S_t)) \end{aligned} \tag{31}$$

2.5.3 MC Learning vs. TD Learning

Monte Carlo learning has high variance and zero bias as it always updates its value function from complete episodes. MC learning comes with good convergence properties, even under

function approximation, is not very sensitive to initial values and is easy to understand and use.

Temporal difference learning has low variance and some bias. $TD(0)$ converges to the true value function but not always under function approximation and it is more sensitive to the initial values. Apart from that, it is usually more efficient than MC.

$TD(\lambda)$ offers a way to combine MC learning and TD learning into one algorithm.

2.6 Model Free Control

While model free prediction is about evaluating a value function for an unknown MDP, model free control is about optimising a value function for an unknown MDP. Model free control builds upon the general principles of policy iteration. Since the entire MDP is not known (or is too complex to describe) in model free control, the action value function Q needs to be utilized for policy improvement, as Q does not need to know the environment dynamics [6]. As the complete MDP is unknown and the experiences are collected via samples, it is vital to find a good exploration-exploitation trade-off, so the policy won't get stuck in a local optima.

2.6.1 Exploration-Exploitation Trade Off

In model free control, the MDP is not or only partially known and it may occur that the agent only finds a sub-optimal solution to the problem, because it hasn't experienced enough states to be able to evaluate which policy performs best. In order to prevent this from happening, it has to be made sure that the agent experiences all or enough states to be able to find the optimal policy. This is achieved by balancing the exploration-exploitation trade-off, which makes sure that the agent explores enough to experience sufficient states, while also continuing to exploit the learned behaviour to improve the policy and accumulate more rewards. One of the simplest and most effective ideas is ϵ-greedy exploration [6].

ϵ-Greedy Exploration

In ϵ-greedy exploration, a constant quantity ϵ, which lies between 0 and 1, is introduced. ϵ controls the likelihood with which the agent exploits or explores. With likelihood $1-\epsilon$ the agent will exploit and act greedily and with likelihood ϵ the agent will explore and choose a random action.

$$\pi_*(a|s) = \begin{cases} 1-\epsilon & a^* = \underset{a \in A}{\arg\max}\, Q(s,a) \\ \epsilon & \text{random action} \end{cases}$$

In order to make sure the algorithm will eventually converge to an optimal policy, ϵ has to decay towards zero after some time. One way of achieving this decaying property is via *Greedy in the Limit with Infinite Exploration (GLIE)*, which reduces ϵ to zero at $\epsilon_k = \frac{1}{k}$, where k is the kth episode.

2.6.2 GLIE Monte Carlo Control

In order to achieve model free control, a model free prediction algorithm, like Monte Carlo learning, has to be used to evaluate a value function under policy π. One example is the GLIE Monte Carlo control algorithm. The GLIE MC control algorithm samples k episodes under policy π: $\{S_1, A_1, R_2, \ldots, S_T\}$. For each experienced state S_t and action A_t in the episode, a counter $N(S_t, A_t)$ gets increased and the value function $Q(S_t, A_t)$ gets evaluated.

$$\begin{aligned} N(S_t, A_t) &= N(S_t, A_t) + 1 \\ Q(S_t, A_t) &= Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t)) \end{aligned} \tag{32}$$

The policy is then improved upon the new action value function via ϵ-greedy exploration, where $\epsilon = \frac{1}{k}$.

2.6.3 SARSA

Monte Carlo learning has high variance and works on complete episodes. Temporal difference learning however offers lower variance and can be applied online and on incomplete sequences. This results in an improvement of model free control, where TD is applied in the evaluation step instead of MC. This specific algorithm is called SARSA [6] and is one of the most well-known basic RL algorithms. In contrast to MC control, which updates its value function every episode, SARSA updates its value function every time-step. This makes sure that the policy improves more rapidly. Furthermore, SARSA is an on-policy algorithm, which means it improves the policy it is actually following.

$$Q(S, A) = Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A)) \tag{33}$$

2.6.4 Q-Learning

While SARSA is an on-policy algorithm, Q-Learning [6] is an off-policy algorithm. Off-policy learning means learning a policy π while following a different policy μ. There are multiple situations where off-policy learning is preferred over on-policy learning. For instance when the agent should learn multiple policies while following only one or when the agent should learn from observing other agents.

Algorithm 1 SARSA Algorithm

Parameters: step size α, greedy ϵ
Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for each episode **do**
    Initialize s
    Choose action a from state s using an exploration policy derived from Q (e.g., ϵ-greedy)
    while s is not terminal state **do**
        Take action A, observe reward r and next state s'
        Choose next action a' using the same exploration policy
        $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$
        Update state $s \leftarrow s'$ and action $a \leftarrow a'$
    end while
end for

The goal of Q-Learning is to learn the action values $Q(S, A)$ by choosing the next action A, using the behaviour policy μ, but also considering an alternative action a', following the target policy π, which is a greedy policy. The action value function $Q(S, A)$ is then updated towards the value of the alternative action a'.

$$Q(S, A) = Q(S, A) + \alpha(R + \gamma \max_{a'} Q(S', a') - Q(S, A)) \tag{34}$$

2.7 Value Function Approximation

The described methods so far work with table look ups, where all the different values of value functions are getting stored in a table or a matrix. Applied to real world problems, this becomes infeasible very quickly, as real environments tend to have very large or even infinite state and/or action spaces. Even board games like GO have a state space of 10^{170}, which are more states than atoms in the known universe [26] and a robot operating in the real world operates in an infinite, continuous state and/or action space. Those large state and action spaces make lookup tables impractical, as on the one hand there are too many states/actions to fit into memory and on the other hand it would be too slow to learn the value of each state individually. Subsequently, another method needs to be chosen to make reinforcement learning scale and work in real world problems, namely function approximation. The goal of function approximation in RL is to estimate the true value function for each state or state-action pair.

$$\begin{aligned} \hat{v}(s, w) &\approx v_\pi(s) \\ \hat{q}(s, a, w) &\approx q_\pi(s, a) \end{aligned} \tag{35}$$

where the parameter w is a vector of weights. The function approximator can take many forms e.g.: a linear function approximator or an artificial neural network. The parameter w

gets updated using MC or TD learning to fit the function approximation $\hat{v}(s, w)$ in a compact form to the true value function $v_\pi(s)$. Function approximation allows to generalize from seen states to unseen states and therefore needs less memory and is faster during training. In the following sections, the individual parts and workings of value function approximation are getting described more closely.

2.7.1 Stochastic Gradient Descent

In order for value function approximation to work, it needs to fit the approximate value function to the true value function. One way of achieving this is via stochastic gradient descent (SGD), which minimizes the error of the objective function $J(w)$ between approximation and the true value function, by updating the weights w into the direction of the gradient $\nabla_w \hat{q}(S, A, w)$ with step-size α. For simplicity's sake, we only consider the action value function $\hat{q}(S, A, w)$, but the same principles apply to the state value function $\hat{v}(S, w)$.

$$J(w) = (q_\pi(S, A) - \hat{q}(S, A, w))^2 \tag{36}$$

$$\begin{aligned} \Delta w &= \alpha \nabla_w J(w) \\ &= \alpha(q_\pi(S, A) - \hat{q}(S, A, w)) \nabla_w \hat{q}(S, A, w) \end{aligned} \tag{37}$$

2.7.2 Linear Value Function Approximation

In linear value function approximation, the estimated value function $\hat{q}(S, A, w)$ can be represented as a linear combination of features $x(S, A)$ and weights w. $x(S, A)$ is a feature vector which describes the state and action space in a more compact form and can be found through various ways like coarse coding, tile coding or neural networks [6]. The aspect of finding the right feature representation will not receive further scrutiny, as delving deeper into this matter would exceed the scope of this work. The update rule Δw via SGD becomes relatively straightforward in linear value function approximation.

$$\hat{q}(S, A, w) = x(S, A)^T w \tag{38}$$

$$\Delta w = \alpha(q_\pi(S, A) - \hat{q}(S, A, w)) x(S, A) \tag{39}$$

2.7.3 Incremental Methods

Since the true value function $q_\pi(S, A)$ is not initially known in reinforcement learning problems but needs to be learned from experience, the stochastic gradient descent algorithm has to be

updated to use this learned value function. This can be either done via MC Learning and the return G_t

$$\Delta w = \alpha(G_t - \hat{q}(S_t, A_t, w))\nabla_w \hat{q}(S_t, A_t, w) \tag{40}$$

, via TD Learning, for example via TD(0) and the TD target $R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, w)$ or via other known algorithms.

$$\Delta w = \alpha(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}w) - \hat{q}(S_t, A_t, w))\nabla_w \hat{q}(S_t, A_t, w) \tag{41}$$

Incremental prediction algorithms update the weights in the direction of the gradient at every step. Those methods sometimes do not converge to the global optimum when used with value function approximation. Batch methods address some of those issues.

2.7.4 Batch Methods

Batch methods make reinforcement learning via value function approximation more stable by adding an experience replay buffer D. By using an experience replay buffer D, the algorithm becomes more sample efficient. The experience replay buffer is created from the agent's experience and consists of subsequent $(state, value)$ pairs.

$$D = \{(s_1, v_1^\pi), (s_2, v_2^\pi), \ldots, (s_T, v_T^\pi)\} \tag{42}$$

The objective function $LS(w)$ to minimise the error between the value function approximation and the true value function is the least squares method.

$$LS(w) = \sum_{t=1}^{T}(v_t^\pi - \hat{v}(s_t, w))^2 \tag{43}$$

In order to converge to the least squares solution $LS(w)$, the experience replay buffer is repeatedly sampled, and stochastic gradient descent updates are performed on those samples.

$$\begin{gathered}(s, v^\pi) \sim D \\ \Delta w = \alpha(v_\pi - \hat{v}(s, w))\nabla_w \hat{v}(s, w)\end{gathered} \tag{44}$$

In the end, this becomes similar to the supervised learning process, in which the training data would be the observed states and values D.

2.7.5 Deep Q-Learning

Deep Q-Learning takes the idea of batch methods even further by applying a fixed Q target. Deep Q-Learning (DQN) [27] is one of the most well-known reinforcement learning algorithms. It is specifically prominent for performing on a superhuman level over several different Atari

games and learning just from screen pixels, using a convolutional neural network (CNN) as a value function approximator. Deep Q-Learning collects experiences according to an ϵ-greedy policy and stores the transitions $(s_t, a_t, r_{t+1}, s_{t+1})$ in a replay buffer D. It then samples a random mini-batch of transitions from the replay memory. Furthermore, it utilizes two different sets of weight vectors: one fixed for the q-target-network w^- and another for the q-network w to stabilize training even further. It then optimises the mean squared error (MSE) between the q-network and q-target-network via stochastic gradient descent.

$$L_i(w_i) = \mathbb{E}_{s,a,r,s' \sim D}((r + \gamma \max_{a'} Q(s', a', w_i^-) - Q(s, a, w_i))^2) \tag{45}$$

After some number of steps the fixed q-target-network gets updated with the values of the q-network and training continues under the new fixed q-target-network.

2.8 Policy Gradient Methods

Compared to the previous value-based methods, which achieve policy improvements via value functions, policy gradient methods optimize the policy directly by parameterizing the policy π. The goal is to maximize the total return by modifying the parameter θ via gradient ascent. Formally, the policy π is a probability distribution over actions a that is conditioned by the state s and parameters θ:

$$\pi_\theta(s, a) = \mathbb{P}(a|s, \theta) \tag{46}$$

Policy gradient methods have a few advantages over value-based methods as they have better convergence properties, are effective in high-dimensional or continuous action spaces and can learn stochastic policies, which is especially useful in state spaces with state aliasing (states which are not unique). On the downside, policy gradient methods typically converge slower and rather to a local than a global optimum.

2.8.1 Policy Objective Functions

The goal of policy gradient methods is to find a policy which maximizes the total reward by adjusting parameter θ. In order to achieve this, there needs to be a policy objective function which is able to measure the quality of a policy π_θ. In episodic environments, an objective function could measure how much reward it would accumulate starting from state s,

$$J_1(\theta) = R_{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}(r) \tag{47}$$

while in continuous environments, the average reward or the average reward per time-step could be used as a valid objective function.

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) R^{\pi_\theta}(s) \tag{48}$$

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) R_s^a \tag{49}$$

where $d^{\pi_\theta}(s)$ is a stationary distribution of a Markov chain for π_θ.

2.8.2 Monte Carlo Policy Gradient

Monte Carlo policy gradient allows the computation of the policy gradient, which needs to be ascended to maximize the expected rewards. The expectations of the policy gradient can be described via likelihood ratios:

$$\begin{aligned} \nabla_\theta \pi_\theta(s,a) &= \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} \\ &= \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) \end{aligned} \tag{50}$$

where $\nabla_\theta \log \pi_\theta(s,a)$ is the score function, which indicates in which direction the policy needs to be changed to achieve more or less of a certain quantity.

One-step MDP

In an one-step MDP (a MDP that terminates after one step/decision and gives an immediate reward for that step), the objective function $J(\theta)$ takes the following form:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta}(r) \\ &= \sum_{s \in S} \sum_{a \in A} \pi_\theta(s,a) R_{s,a} \end{aligned} \tag{51}$$

and the policy gradient of that objective function is represented as:

$$\begin{aligned} \nabla_\theta J(\theta) &= \sum_{s \in S} \sum_{a \in A} \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a) R_{s,a} \\ &= \mathbb{E}_{\pi_\theta}(\nabla_\theta \log \pi_\theta(s,a) r) \end{aligned} \tag{52}$$

if r is a positive reward, the score function $\nabla_\theta \log \pi_\theta(s,a)$ indicates which direction to move in, to achieve more of that reward and if r is negative, the score function moves into the opposite direction to achieve less negative rewards.

Policy Gradient Theorem

In order to solve a multi-step MDP, the immediate reward r needs to be replaced by a value function $Q_\pi(sa)$:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}(\nabla_\theta \log \pi_\theta(s,a) Q_{\pi_\theta}(s,a)) \tag{53}$$

REINFORCE

The REINFORCE [6] algorithm is a Monte Carlo policy gradient algorithm which updates its parameter θ via stochastic gradient ascent, using the policy gradient theorem and the sum of the rewards G_t as an unbiased sample of the value function $Q_{\pi_\theta}(s_t, a_t)$

$$\Delta_{\theta_t} = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) G_t \tag{54}$$

2.8.3 Actor Critic Policy Gradient

Monte Carlo gradient methods learn very smoothly as the policy is always a little bit improved, but they tend to be very slow and have high variance. Actor critic methods aim to solve this via the introduction of the critic [25]. Instead of the total sum of rewards G_t, actor critic methods use a value function approximation $Q_w(s, a)$, which is estimated by the critic via policy evaluation. Actor critic methods deploy two different function approximators, one for the value function and one for the policy. The critic updates the action value function parameter w and the actor updates the policy parameter θ in the direction suggested by the critic. Conceptually, the actor is responsible for acting in the world, while the critic gives information on the quality of those actions and into which direction to improve. The approximate policy gradient of the general actor critic algorithm has the following form:

$$\Delta_\theta = \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a) \tag{55}$$

Advantage Estimate

An additional way to reduce variance is to introduce a baseline $B(s)$. A preferred baseline is the state value function $V_{\pi_\theta}(s)$ [28]. This baseline gets subtracted from the action value function, resulting in an advantage estimate A_{π_θ}. The advantage estimate gives information on how much more reward than usual can be gained, when taking a certain action a.

$$A_{\pi_\theta} = Q_{\pi_\theta}(s, a) - V_{\pi_\theta}(s) \tag{56}$$

$$\Delta_\theta J(\theta) = \mathbb{E}_{\pi_\theta}(\nabla_\theta \log \pi_\theta(s, a) A_{\pi_\theta}) \tag{57}$$

Instead of estimating $Q_w(s, a)$ and $V_v(s)$ separately and introducing another function approximation, there is a more elegant way, which is approximating $V_v(s)$ alone and using the TD-error δ_v for the advantage estimate.

$$\delta_v = r + \gamma V_v(s') - V_v(s) \tag{58}$$

$$\Delta_\theta J(\theta) = \mathbb{E}_{\pi_\theta}(\nabla_\theta \log \pi_\theta(s, a) \delta_{\pi_\theta}) \tag{59}$$

2.8.4 TRPO

Trust Region Policy Optimization (TRPO) [28] is another policy gradient algorithm, which improves actor critic methods even further by adding the surrogate loss as an objective function and step sizing within a trust region.

Surrogate Loss

The surrogate loss as an objective function allows to reuse samples of an old policy to improve a current policy, making the algorithm more data efficient. This can be achieved via a statistical method called importance sampling, which allows estimating a distribution by taking samples of another distribution. The surrogate loss in policy gradient methods takes the following form:

$$J(\theta) = \mathbb{E}_{\pi_{\theta_{old}}} \frac{\pi_\theta(s,a)}{\pi_{\theta_{old}}(s,a)} A_{\pi_{\theta_{old}}}(s,a) \tag{60}$$

Step Sizing

The step size in policy gradient methods is critical as too big step sizes result in bad policies and it is very hard for the algorithms to recover from that. Whereas very small step sizes take a lot of time to train.

Trust Region

TRPO solves this issue by adding a trust region and turning the problem into a constraint optimization problem, in which it optimizes for the surrogate loss, but makes sure that the new policy does not differ too much from the old policy by constraining the Kullback-Leibler (KL) divergence to be not greater than ϵ. KL divergence describes the distance between two distributions.

$$\begin{aligned} \text{maximize: } & \mathbb{E}_{\pi_{\theta_{old}}} \frac{\pi_\theta(s,a)}{\pi_{\theta_{old}}(s,a)} A_{\pi_{\theta_{old}}}(s,a) \\ \text{constraint: } & \mathbb{E}_{\pi_{\theta_{old}}} KL(\pi_{\theta_{old}}(s,a), \pi_\theta(s,a)) \leq \epsilon \end{aligned} \tag{61}$$

2.8.5 PPO

Proximal Policy Optimization (PPO) [23] is an improvement of TRPO. It aims to utilize first order approximations instead of higher order approximations to be compatible with common deep learning frameworks like Tensorflow [29] or PyTorch [30] and therefore be easier and practical to apply. Furthermore, PPO is more general, has better sample complexity and is one of the most used RL algorithms nowadays because of its ease of use, robustness and good results.

Version 1

PPO version 1 achieves the mentioned improvements by moving the KL divergence constraint with a weighting factor β into the objective function:

$$J(\theta) = \max_{\theta}(\mathbb{E}_{\pi_{\theta_{old}}} \frac{\pi_\theta(s,a)}{\pi_{\theta_{old}}(s,a)} A_{\pi_{\theta_{old}}}(s,a)) - \beta(\mathbb{E}_{\pi_{\theta_{old}}} KL(\pi_{\theta_{old}}(s,a), \pi_\theta(s,a)) - \epsilon) \tag{62}$$

This turns the constrained optimization problem into an unconstrained optimization problem.

Version 2

PPO version 2 improves this even further by changing the way the trust region is being calculated. Instead of the KL divergence, it uses clipping to make sure that θ can only contribute a certain amount in optimizing the objective.

$$J(\theta) = \mathbb{E}_t \min(\frac{\pi_\theta(s_t,a_t)}{\pi_{\theta_{old}}(s_t,a_t)} A_t, clip(\frac{\pi_\theta(s_t,a_t)}{\pi_{\theta_{old}}(s_t,a_t)}, 1-\epsilon, 1+\epsilon) A_t) \tag{63}$$

PPO is also the algorithm which gets used in Unity ML Agents and Godot RL Agents and in the experiments of this work.

2.9 Reinforcement Learning Frameworks

There are multiple reinforcement learning frameworks which provide RL algorithms and environments, but only a few allow the creation of complex and diverse 2D and 3D reinforcement learning environments with realistic physics simulation. Unity ML Agents and Godot RL Agents tackle this issue by providing a general-purpose engine to easily create challenging environments and train agents in those environments. Being able to easily create new complex reinforcement learning environments is essential for the invention and progress of new reinforcement learning algorithms [15] as a lot of new RL algorithms already tend to perform on a superhuman level on previous RL benchmarking environments and new, more complex environments are able to diversify the picture again. Unity ML Agents and Godot RL Agents, both, target AI researchers and game developers. AI researchers are able to benchmark new algorithms in custom, shareable, complex environments, whereas game developers have the option to train and integrate RL agents as game assets or to automate parts of the QA process via trained RL agents. Both frameworks provide a training API to communicate via a Transmission Control Protocol (TCP) connection between the environment and the reinforcement learning algorithms. In the following sections Unity ML Agents and Godot RL Agents are getting described in a concise manner.

2.9.1 Unity ML Agents

Unity ML Agents [15] is an open source reinforcement learning framework by Unity for the proprietary game engine Unity. It allows to create reinforcement learning environments and simulations in the game engine Unity and to train reinforcement learning agents via a python API. Additionally, it provides its own out-of-the-box implementations of popular reinforcement learning algorithms like PPO, Soft Actor Critic (SAC) [31] and self-play. Apart from training agents via classical reinforcement learning Unity ML Agents also offers the possibility to train agents via imitation learning and neuroevolution.

2.9.2 Godot RL Agents

Godot RL Agents [16] is inspired by Unity ML Agents and is an open source reinforcement learning framework by Edward Beeching for the open source game engine Godot. Like Unity ML Agents, it allows to create simulations for reinforcement learning problems inside a game engine and to train RL agents via a python interface. Unlike Unity ML Agents, Godot RL Agents offers the possibility to use four different reinforcement learning libraries, each with their own implementation of popular RL algorithms.

2.10 Reinforcement Learning Libraries

Godot RL Agents offers the choice between four different open source RL libraries: Ray RLlib (RLlib), Sample Factory (SF), Stable Baselines3 (SB3) and CleanRL. Those libraries all provide their own implementations of high quality, well tested RL algorithms. In the following sections, the four RL libraries are getting briefly described.

2.10.1 Ray RLlib

Ray RLlib [19] is an open source reinforcement learning library and part of the Ray unified compute framework. Godot RL Agents allows to configure Ray RLlib hyperparameters via yaml files, similar to the config yaml files in Unity ML Agents.

2.10.2 Sample Factory

Sample Factory [20] is an open source RL library by Aleksei Petrenko and claims to be one of the fastest and most efficient RL libraries [1]. Unlike Ray RLlib, which covers implementations of various RL algorithms, Sample Factory focuses on implementations of synchronous and asynchronous policy gradient algorithms (PPO). Sample Factory supports at the time of this

[1] https://www.samplefactory.dev/

writing only Linux, which is the main reason why the training of the agents for this work was done on Linux, rather than on any other platform.

2.10.3 Stable Baselines3

Stable Baselines3 [18] is a newer, reworked, open source implementation of Open AI's baselines reinforcement learning algorithms. It offers several different RL algorithms (PPO, SAC, DQN, A2C, ...), RL examples and integrations of different ML utilities.

2.10.4 CleanRL

CleanRL [21] offers clean and simple single-file implementations of online reinforcement learning algorithms (DQN, SAC, PPO, ...). The single-file solutions enable rapid modifications to the algorithms. Like the other libraries, it also offers tracking capabilities via Tensorboard, allowing one to visualize the training process.

2.11 Inference & Heuristic

Unity ML Agents and Godot RL Agents allow for inference and heuristic control inside the respective game engines. The following sections describe the meaning of inference and heuristic in the traditional sense and in the context of the two frameworks.

2.11.1 Inference

Inference is the process of "predicting values of variables given some other variables" [32]. While training in machine learning is the process, in which the model gets trained on previously collected data via ML algorithms to make accurate predictions, inference is the process where the pre-trained model gets utilized in production to make predictions on live data. In the context of Unity ML-Agents and Godot RL Agents inference is the process of using the pre-trained reinforcement learning model to act live in the simulation. Inference in this situation allows for using a trained agent as an asset in the actual game or simulation, e.g. as a non-player character (NPC) which has learned how to perform certain tasks in the environment. While the training process takes a long time, ranging from minutes to weeks and sometimes even months, depending on the complexity of the reinforcement learning problem, inference is relatively quick and is, therefore, more suitable for real-time applications like games or simulations [9]. The main reason why training takes a lot longer than inference is that the agent has to explore the environment a lot, as it does not yet know what the best actions are and therefore has to learn them. However, at inference time, the agent will always choose the action which maximizes its

likelihood to increase the expected reward. Additionally, the process of training deep neural networks during reinforcement learning also takes a considerable amount of time, as it is not only forward propagating the data, but it is also back propagating to minimize the loss and update its weights and biases. At inference time, the neural network just takes the observed data as an input, forward propagates it and outputs a prediction, making it a less computationally intensive task and thereby faster. Unity ML-Agents and Godot RL Agents allow for inference by providing the trained model as a `.onnx` file to the agent. In Godot RL Agents this currently only works for models trained with the SB3 backend.

2.11.2 Heuristic

Judea Pearl describes heuristics as "criteria, methods, or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal" [33]. In Unity ML-Agents and Godot RL-Agents heuristic is a way to control the agent's behaviour other than by the trained reinforcement learning model but for example via mouse and keyboard or gamepad. This is especially useful for debugging purposes. A game designer or reinforcement learning problem designer can test if the agent actually behaves the way it is expected to behave when receiving new input data. By transforming key and button presses to input data and feeding it into the API, which is responsible for the character movement, the designer can simulate how the agent would behave if it receives those kind of inputs from the reinforcement learning algorithm. Additionally, it is a way for the designer to check if the environment behaves as expected when the playable character collects a reward or reaches a terminal state. Making sure that the simulation works as intended is important for the success of a reinforcement learning experiment. This is especially true for such a time-consuming process as training RL agents, in which the iterative process is much slower compared to traditional software engineering.

3 Experiment

The goal of this work is to compare the RL frameworks Unity ML Agents and Godot RL Agents with respect to their learning and runtime performance. As briefly mentioned in the introduction, a few things have to be assured to be able to meaningfully compare the two reinforcement learning frameworks. The agents have to be trained on the same hardware to be able to compare the computational runtime performance of the frameworks. Additionally, it is important that the agents are trained in similar environments and that the MDP, including the defined rewards, observations and actions of the agent, are the same in both reinforcement learning problems

to be able to derive a meaningful comparison about the learning performance of the agents. In the following sections, the technical setup, the simulation environment, the design of the reinforcement learning problem and the MDP are described in greater detail. Moving forward, the experiment, its details and configurations are illustrated more extensively. Finally, a brief overview of the technical challenges will be provided, along with an explanation of how they were addressed.

3.1 Technical Setup

The agents were trained on a Lenovo ThinkBook 16p Gen 2. The notebook contains an AMD® Ryzen 9 5900HX (8 cores, 16 threads) central processing units (CPUs), 32 gigabyte (GB) random access memory (RAM) and a 1 terabyte (TB) solid-state-drive (SSD). Additionally, it features two different graphics processing unit (GPU), a Radeon Graphics (8 cores) and a NVIDIA GeForce RTX 3060 (6GB video random access memory (VRAM)). The training happens on the operating system Ubuntu 22.04.2 LTS 64-bit, as Linux is the only operating system (OS) which supports all features of the different reinforcement learning backends provided by Godot RL Agents, making it the only choice to compare all the different RL backends of Godot RL Agents with Unity ML-Agents. The computer is plugged in during training to ensure that the training can utilize the hardware fully. Furthermore, every trial is run sequentially and not in parallel to ensure that each trial gets the whole available computational power. The training process utilizes the GPU (NVIDIA GeForce RTX 3060), as it is the fastest option for training an agent, due to the capabilities of a GPU to perform tasks in a highly parallel manner [34]. NVIDIA's compute unified device architecture (CUDA) [1] provides an application programming interface (API) which allows the training application to harness the computational power of the graphics card.

3.1.1 CUDA

CUDA is an API by NVIDIA which allows developers to utilize the GPU to perform certain tasks in a highly parallel manner. Unity ML-Agents and Godot RL-Agents both allow to utilize the CPU or the GPU (as long as the computer has a supported GPU) for training. Training on the GPU is usually much faster than training on the CPU, due to parallelism. More specifically Unity ML-Agents uses the deep learning library PyTorch [30] and Godot RL-Agents utilizes either PyTorch or TensorFlow [29], depending on the reinforcement learning backend, which in turn offer to utilize the CPU or GPU. Both PyTorch and TensorFlow are deep learning libraries, which provide a wide range of deep learning algorithms, like deep neural networks and have an emphasis on computational performance by utilizing the possibilities of the underlying hardware.

[1] https://developer.nvidia.com/cuda-toolkit

3.2 Experimental Setup

The reinforcement learning experiment comprises several aspects. In the following sections, the simulation environment and the reinforcement learning problem will be described in greater detail, followed by the MDP and its corresponding observations, actions and rewards. Continuing from there, the detailed configurations and changes in the configuration are getting explained. As a final step, the technical challenges, which occurred during the creation and conduction of the experiment, will be revealed.

3.2.1 Simulation Environment

Overview

The simulation environment is a small 3D platformer game, in which the playable character (the avatar) has to jump from platform to platform and collect coins. Every time the playable character collects a coin, a new platform with a new coin gets spawned at a certain distance and in a random direction. The playable character has to jump on the newly spawned platform and collect the new coin. If the playable character falls off a platform or misses a platform during a jump, the avatar resets to the start position and the amount of coins is reset to zero. The overall goal of the game is to collect as many coins as possible in one run without falling off a platform. The simulation is built twice, one time in Unity and another time in Godot. During the creation of the simulation and its environment, a lot of effort has been taken into account to make the simulations as similar as possible, to be able to meaningfully compare the two different reinforcement learning frameworks. This does not only apply from a conceptual point of view but also from a technical point of view. In the upcoming section, the technical details will be depicted in greater detail.

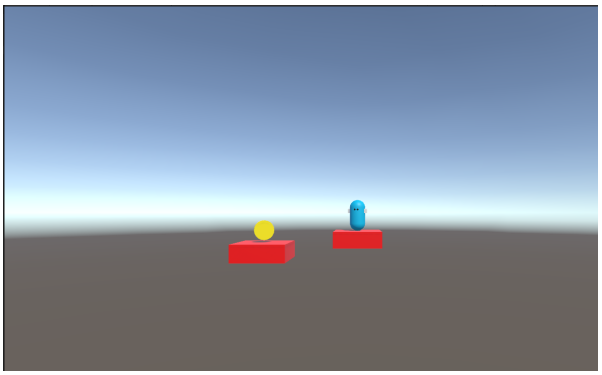

Figure 4: Environment Simulation in Unity

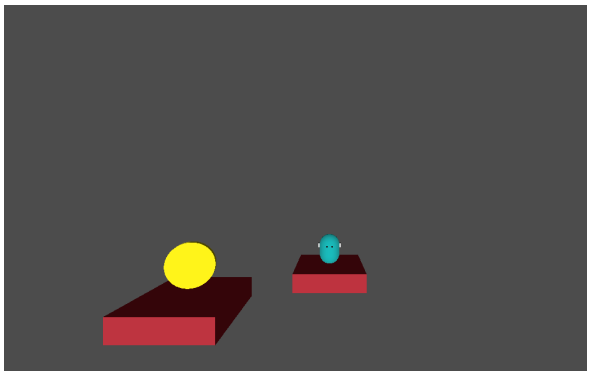

Figure 5: Environment Simulation in Godot

Training Environments

The training can take place either directly inside the game engine or inside an exported build of the simulation. Both variations have certain advantages and disadvantages.

Training inside the game engine allows one to inspect the values of the environment and the agent during training which can be useful for debugging purposes. Furthermore, it is possible to iterate more quickly, as there is no waiting for the build process to complete, but training can be directly started via the play button in the corresponding game engine. On the other hand, training in the game engine is slower, as it does not allow to spawn multiple instances of the same simulation and run the training in parallel.

Training inside an exported build however allows to spawn multiple instances of the same simulation and thereby speeding up the training process. Additionally, it is also possible to share the builds with other researchers more easily, without needing the other party to install the game engine to run the experiment. Last but not least, exported builds offer researchers the possibility to deploy their reinforcement learning problem to the cloud, run the experiment on better hardware than their local machine and scale up the training process. As the research question of this work aims to measure the performance of the two reinforcement learning frameworks, the experiments are run on exported builds rather than directly inside of the game engine.

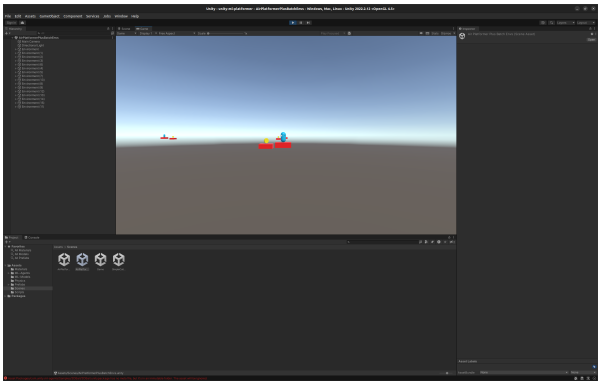

Figure 6: Training inside the Unity Engine

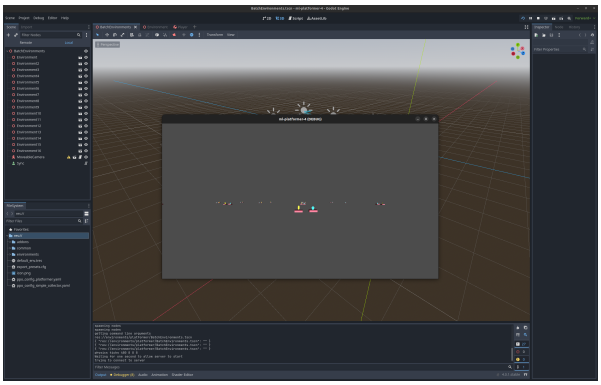

Figure 7: Training inside the Godot Engine

Multiple Environments

For the purpose of speeding up the training, 16 simulation environments and playable characters are getting spawned in one single instance of the reinforcement learning application. This makes it possible for the agents to train and learn in parallel, while at the same time, all of the 16 agents contribute their learning experience to the training of the artificial neural network. This process increases the overall training speed, as more data, in the form of observations, actions and rewards can be generated by multiple environments and the artificial neural network has to

wait less for the next batch of training data, being able to train in parallel to the execution of the simulation.

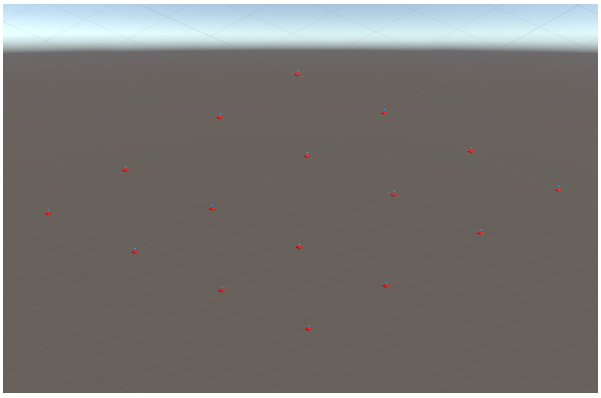

Figure 8: Multiple Environments in Unity

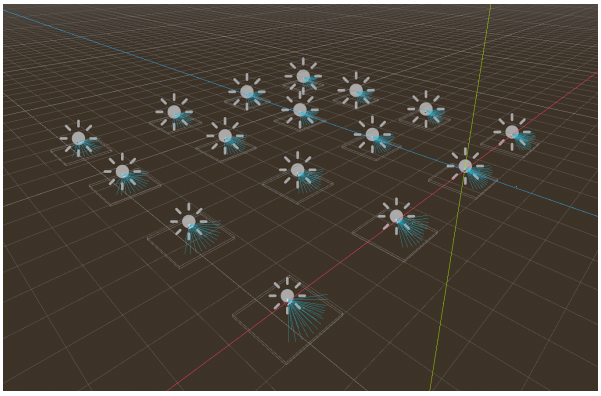

Figure 9: Multiple Environments in Godot

Multiple Instances

Apart from multiple environments in a single instance, it is also possible in Unity ML-Agents and in all of the RL backends of Godot RL Agents to spawn multiple instances of the reinforcement learning application and carry out the training of these instances in parallel. In the trials of the experiment, 8 instances with 16 environments each, were spawned, resulting in 128 (8 * 16) parallel environments. This practice allows for additional speed-up during training.

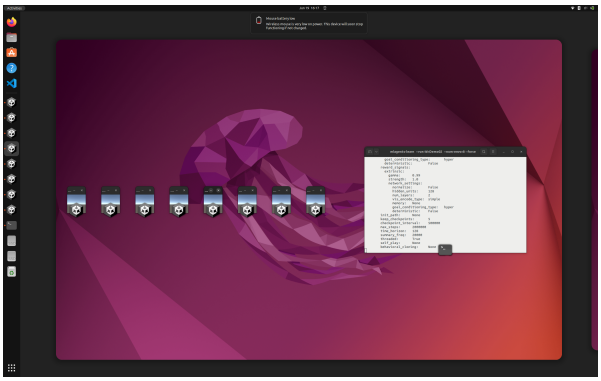

Figure 10: Multiple Instances in Unity

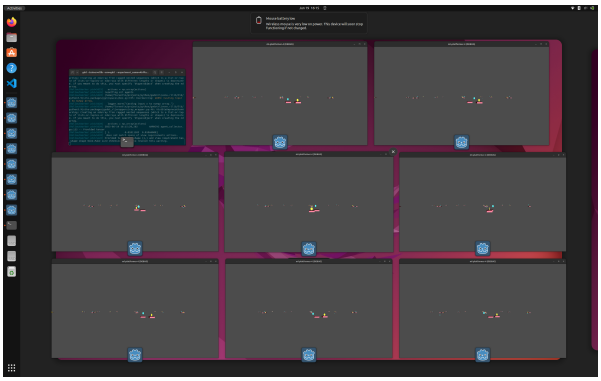

Figure 11: Multiple Instances in Godot

3.2.2 Technical Details

From a technical point of view, considerable efforts have been invested in making the simulations in Unity and Godot as similar as possible. The following sections describe the technical details of the environment and the playable character. If not stated otherwise, the description applies to both implementations of the simulation in Unity and Godot.

Environment

The environment consists of a playable character, two platforms and one coin. Once the playable character falls beneath a certain predefined threshold, the episode ends and the environment gets reset. The environment reset involves setting the reward to zero and placing the playable character back to its original position. Every time the playable character collects a coin by colliding with the coin's trigger area, the oldest platform gets destroyed and a new platform with a new coin gets spawned. The spawning logic makes sure that there are always only two platforms and one coin in the environment at any time in the simulation. The new platform is consistently generated in a random direction and positioned at a specific spawn distance from the playable character's position. The platform spawn distance is a parameter which can be adjusted in the game engine.

Playable Character

The logic of the playable character consists of two main components. One component is responsible for providing an API to control the character (moving left, right, forwards, backwards and jump) and receiving observed states from the character (e.g. is the playable character currently touching the floor?). The other component is responsible for communicating with the python API and the reinforcement learning algorithms. It is transforming and sending perceived observations, actions and rewards from the simulation to the python API and translating received data from the python API into actions. The translated actions are then fed back to the character control API, making the character move accordingly. This separation of concerns allows for making changes easily in one part of the logic without affecting the other part too much. For example, it would be relatively simple to rewrite part of the character control logic without having to touch the component which is responsible for communicating with the python API and the reinforcement learning process and vice versa. Furthermore, it allows to easily switch between inference and heuristic control.

The character's movement is implemented via the engine's corresponding physics system. The playable character has a Rigidbody, a component in both game engines, which can be influenced by physical forces. If the character were to move forward, a force in this direction is applied to its Rigidbody and if the character jumps, a force in the y-direction is applied to its Rigidbody. Furthermore, the playable character's Rigidbody is also affected by gravity and is responsible for making the character fall down, if there is no platform underneath it. Having a similar movement system, with equal values for gravity, movement and jump speed is essential for a meaningful comparison and was implemented accordingly.

The logic to check if the playable character is grounded (standing on a platform) is implemented via a function, which is casting a single ray, into the vertical direction underneath the

character. If the ray hits a collidable object inside a specific predefined small range, the character is grounded and the function returns true. If the playable character is not grounded, but in the air, the function returns false. This check makes sure that the playable character is only allowed to jump if there is solid ground underneath it, preventing it from flying infinitely high into the air. Additionally, there is an implementation of a slope limit check, making sure that the playable character cannot jump from walls or angles above 45 degrees.

Finally, the playable character is able to recognize if there is something in front of it via a raycast sensor (Godot RL Agents) or ray perception sensor (Unity ML-Agents). A raycast sensor/ray perception sensor in the context of the two frameworks is a set of ray casts that fire into a predefined direction at a given length. The game designer can influence the number of rays, the length of the rays, the maximum degree of the sensor and which objects the rays should be able to detect. The number of rays and angle of the sensor are responsible for the character's "field of view", defining how much the character can "see" horizontally and vertically. In the Godot environment, the agent has 10 rays per width and 2 rays per height, resulting in 20 rays for perceiving its surroundings. In the Unity environment, the agent has 11 rays per width, as Unity ML-Agents always adds the defined amount plus one more ray, and 2 rays per height, resulting in a total amount of 22 rays. The angle of the ray perception sensor is 100 degrees wide, making it possible for the agent to perceive the environment not only directly straight in front of it, but also 50 degrees to the left and 50 degrees to the right of it. The length of the rays defines how far the character can "see". It is quite crucial to find a decent length for the rays to avoid the character from observing things not relevant to it (because they might be too far away), but at the same time, making sure that the length of rays is sufficient enough, so the character can observe all relevant information.

Another tool to control what the agent can perceive is tags in Unity ML Agents and masks in Godot RL Agents. It allows the designer to tag or mask certain game objects as being perceivable by the sensor or not. In the case of the reinforcement learning experiment, it is made sure that only the platforms and coins are observable by the sensor. If a ray hits a perceivable object, the ray returns a normalized floating point number between 0 and 1, indicating the distance between the playable character and the perceived object. If the value is close to 1, the observed object is far away (at the end of the ray) and if the value is close to 0, the perceived object is very close to the playable character. If the ray does not hit any perceivable object it returns 0. Unity ML-Agents allows for visualizing successful ray hits inside the Unity Game Engine, by marking the hitting ray red, as can be seen in Figure 12.

Randomness

Randomness is an important factor in machine learning as it has the potential to influence the results of training drastically and, if not taken into account properly, makes it hard to repro-

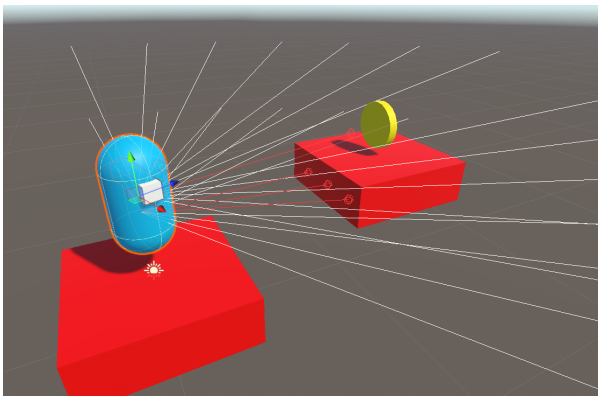

Figure 12: Ray Perception Sensor in Unity

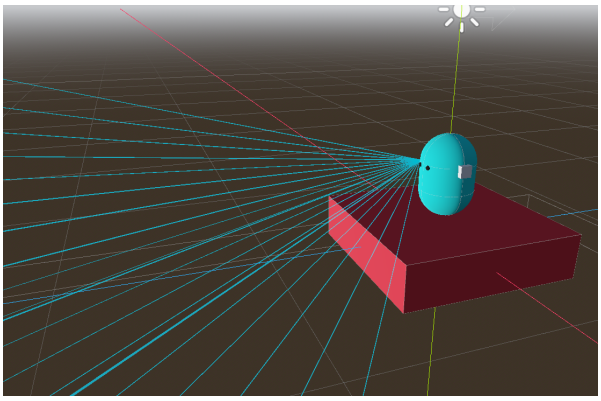

Figure 13: Raycast Sensor in Godot

duce certain outcomes [24]. The environment utilizes randomness for the platform spawning behaviour. Every time a new platform spawns, it gets placed in a new random position. This makes sure that the agent experiences a lot of different states which should prevent the agent from over-fitting but rather to generalize well on the problem [35]. The simulation consists in total of 16 environments that all spawn platforms randomly, which increases the range of experienced states even further.

In order to offer reproducible results, the random number generator of the Unity and Godot simulation got seeded with a fixed seed (42), to ensure that the platforms always get spawned in the same pseudo-random sequence. This allows for reproducible randomness over multiple runs. While getting reproducible random results is an important issue, it is still crucial that the 16 environments within the simulation have different random seeds to guarantee a broad diversity in the experienced states. Whereas Unity handles this out of the box by just providing a random seed, the random number generator of Godot has to be fed with an additional parameter, the unique scene ID, to ensure different random states across the different environments.

Listing 1 Setting the random seed in Godot

```
func _ready():
    rng = RandomNumberGenerator.new()
    rng.seed = 42 + self.get_instance_id()
```

3.3 MDP

Formulating a problem as Markov decision process is an essential requirement for using reinforcement learning algorithms [6]. In the following sections, the properties of the MDP, the

agent, the environment, the states and observations, the actions, and the rewards are described in greater detail.

3.3.1 Agent

The reinforcement learning agent controls the playable character. The agent observes the internal states of the playable character and the environment and tries to maximize its expected rewards by taking optimal actions. In order to achieve this, the agent has to learn the optimal policy. The agent manages to learn this via the right amount of exploration and exploitation during training.

3.3.2 Environment

The environment is the place with which the agent interacts via the playable character. In the depicted reinforcement learning experiment, the environment consists of 2 platforms and a coin. The environment is also responsible for providing the agent with rewards and penalties under certain conditions. The states, actions and rewards defined in the experiment will be described more extensively in the upcoming sections.

3.3.3 States and Observations

The states and observations in the experiment are described by the internal states of the playable character and its relationship to the environment. The states are in a mixed space of continuous and discrete states. The following is a list of the observable states:

Sensor Observations: This state informs the agent of what is in front of the playable character. The observations are getting returned by a set of rays, casting into the direction in front of the playable character at a predefined angle. The values are an array of normalized floating point numbers in the range between 0 and 1.

Is Grounded: This state informs the agent if the playable character is touching the floor (true) or if it is currently in the air (false). The value is a boolean and can be either 0 (false) or 1 (true).

Goal Distance: This state informs the agent about the distance between the playable character and the next goal platform (the platform with the next coin on it). The value is a normalized floating point number in the range between 0 and 1.

Goal Direction: This state provides the agent with information about the relative position of the goal platform to the playable character's position. The value is a normalized 3-dimensional

floating point vector in the range between 0 and 1. The final value gets represented, together with the goal distance, as a 4-dimensional vector.

Listing 2 Goal Distance + Goal Direction represented as a 4-dimensional vector

```
Vec4(GoalDistance, GoalDirection.x, GoalDirection.y, GoalDirection.z)
```

3.3.4 Actions

The agent is able to interact with the environment by performing a set of predefined actions. These actions are in a mixed space of continuous and discrete actions and are defined as follows:

Move: The move action allows the agent to move the playable character either forward or backward. The move action is a continuous action. If the value of the move action is negative, the playable character will move backward and if the value is positive, the character will move forward. When the value of the move action is 0, the character will not move at all. The value of the move action gets clamped between -1 and 1, preventing the agent from moving the playable character too fast and thereby breaking the simulation.

Rotate: The rotate action allows the agent to rotate the playable character either left or right. The rotate action is a continuous action. If the value of the rotate action is negative, the playable character will rotate left and if it is positive, it will rotate right. When the value of the rotate action is 0, the character will not rotate at all. Like the move action, the value of the rotate action gets clamped between -1 and 1, preventing the agent from rotating the playable character too fast.

Jump: The jump action allows the agent to make the playable character jump. The jump action is a discrete action and its value can be either 0 or 1. If the value of the jump action is 1 the playable character will jump, provided the character is currently touching the ground. When the value of the jump action is 0, the character will not jump.

3.3.5 Rewards

The reward is the property of a MDP, which gives the most control over what an agent will eventually learn and achieve. Designing the right reward system is essential for being able to solve a reinforcement learning problem according to the designer's wishes and can be a challenging task. The reward is a number the agent receives in certain situations or states, predefined by the designer of the reinforcement learning problem. The reward can be positive to reinforce specific behaviour or negative to penalize behaviour. The experiment in this work

uses positive and negative rewards. The following list describes the individual rewards more thoroughly.

Pick Up Coin: The agent always gets a reward of 100 when it picks up a coin. As jumping from platform to platform and picking up coins without falling down is the ultimate goal of the reinforcement learning experiment, this is the reward with the highest value.

Moving Into The Right Direction: The agent gets a reward when it is moving in the right direction of the next goal platform. This reward is calculated by comparing the smallest experienced distance of the playable character to the goal platform (the current best distance), with the character's distance to the goal platform of the current frame. If the new distance is smaller, the difference between the old best distance and the new distance gets added as a reward. The best distance then gets updated with the new distance value. Every time a new platform gets spawned, the best distance gets reset to a predefined value to keep the values correctly updated. This reward pattern adds additional information for the agent, telling it in which direction it should be generally moving to achieve its goal.

Listing 3 Calculating The Reward For The Goal Distance

```
    private float ShapingReward() {
        float reward = 0f;
        float goalDistance = 0f;

        if (goalPlatform) {
            goalDistance = Vector3.Distance(
                transform.position,
                goalPlatform.transform.position
            );

            if (goalDistance < bestGoalDistance) {
                reward += bestGoalDistance - goalDistance;
                bestGoalDistance = goalDistance;
            }
        }

        return reward;
    }
```

Time: In each time-step of the game's physics loop, the agent receives a negative reward of -0.01. By default, the fixed physics update loop of Unity runs at 50 frames per second (FPS) and in Godot at 60 FPS. In order to be able to make a reliable comparison between

the two simulations, the physics loop of the Unity simulation was increased to 60 FPS. The penalty of -0.01 at every time-step urges the agent to jump as quickly as possible from platform to platform, as the agent tries to maximize its reward and taking more time would decrease its final reward.

3.4 Configurations

The experiment will be conducted with different configurations, which aim to test the performance of the reinforcement learning frameworks. The following section describes these distinct configurations.

3.4.1 Default Hyperparameters

The experiment will run in the MDP described in section 3.3 with the default configuration of the two frameworks' hyperparameters. Those default settings of the hyperparameters may vary a lot depending on Unity ML-Agents and Godot RL Agents and its respective reinforcement learning backends (Ray's RLlib, Sample Factory, Stable Baselines 3 and CleanRL). This experiment aims to figure out how well the agents perform under these default hyperparameter configurations.

3.4.2 Similar Hyperparameters

As a next step, the experiment will run again in the same MDP, but this time with a shared configuration of hyperparameters, which aims to be as similar as possible. The experiment intends to compare the different frameworks and their respective RL backends in a more "fair" way, as theoretically, the performance of the frameworks should be the same, but might still differ depending on the frameworks' implementation of the corresponding RL algorithms.

3.5 Challenges

During the implementation of the reinforcement learning experiment in both game engines, a few challenges arose. In the subsequent sections, these challenges are elaborated upon in greater detail.

3.5.1 Making The Simulations Comparable

Having similar simulations built in both game engines was one of the basic conditions to be able to derive any meaningful and fair comparison between the two frameworks. This condition

was also one of the main sources of challenges experienced during the creation of this work. It was not only important that the agent and the environment would behave in the same way, but also that they were implemented as closely as possible. For instance, the playable character in both frameworks utilizes a physics-based movement system and in both engines, the agent is controlling the playable character via a similar interface. Furthermore, it was crucial for a fair comparison that the playable characters would move with the same speed and jump approximately the same height, as otherwise, a playable character with more speed or jump power would have a clear advantage of reaching the next platform. Another essential aspect was making sure that the sizes of the playable character and the environment were around the same in both simulations. This was especially important as the distance between the playable character and the next goal platform is also used as part of the reward system. If the maximum distance between the character and a platform would be in one simulation around 20 and in another simulation only around 5 units, then the agent with the higher maximum distance would be able to accumulate 4 times more rewards for the reward function, making the comparison unfair. Additionally, having a similar frame rate for the physics system is yet another crucial aspect for maintaining a substantial comparison between the frameworks, as the agent gets a negative reward of -0.01 for every frame passed within the physics loop and a higher frame rate means a higher total loss for each second. For that reason, the physics system's frame rate of Unity was updated from 50 FPS to 60 FPS to match the frame rate of Godot. Finally, it is possible to set a speedup in Unity ML Agents and Godot RL Agents, which controls how fast the simulation runs and an action-repeat parameter, which influences for how many frames an action will be repeated. Speedup and action-repeat were set to 8 in both Unity ML Agents and Godot RL Agents to allow for a relevant comparison.

3.5.2 Finding Similar Hyperparameters

Another set of challenges in making the frameworks and the corresponding reinforcement learning backends comparable was to find a similar, shared configuration of hyperparameters. The difficulty originated from the fact that every framework has its own naming conventions for its hyperparameters. These hyperparameter names were not always straightforward to derive and therefore the documentation and source code of the libraries had to be checked to find out the meaning of the different names. For example, the number of steps to take per agent before adding it to the experience buffer is called `time_horizon` in Unity ML-Agents, `rollout_fragment_length` in Ray RLlib, `num-steps` in CleanRL and `n_steps` in Stable Baselines3. Furthermore, it is not possible to set every hyperparameter in every framework and library explicitly. For instance, Unity ML-Agents and Ray RLlib have an explicit hyperparameter for the buffer size (`buffer_size` and `train_batch_size`) but in Stable Baselines3, CleanRL and Sample Factory, this hyperparameter cannot be set directly but is rather a combination of other parameters (`n_envs * n_steps` in SB3 and CleanRL and

Table 1: Names of hyperparameters in Unity ML-Agents & Godot RL Agents

Unity ML	Godot[RLlib]	Godot[SF]	Godot[SB3]	Godot[CleanRL]
max_steps	timesteps_total	train_for_env_steps	total_timesteps	total-timesteps
learning_rate	lr	learning_rate	learning_rate	learning-rate
learning_rate_schedule	lr_schedule	lr_schedule	lr_schedule	anneal-lr
epsilon	clip_param	ppo_clip_ratio	clip_range	clip-coef
buffer_size	train_batch_size	num_batches_per_epoch * batch_size	n_envs * n_steps	num-envs * num-steps
batch_size	sgd_minibatch_size	batch_size	batch_size	(num-envs * num-steps) / num-minibatches
gamma	gamma	gamma	gamma	gamma
lambd	lambda	gae_lambda	gae_lambda	gae-lambda
num_epoch	num_sgd_iter	num_epochs	n_epochs	update-epochs
time_horizon	horizon	rollout	n_steps	num-steps
beta	entropy_coeff	exploration_loss_coeff	ent_coef	ent-coef

`num_batches_per_epoch * batch_size` in SF). This makes it harder to choose just any set of hyperparameters as SB3, CleanRL and SF are more restrictive about them which has to be taken into account when trying to find a working shared hyperparameter configuration. Table 1 displays the various hyperparameters and their different names in the corresponding frameworks and RL backends.

3.5.3 Finding A High Performing Set Of Hyperparameters

As stated in [24] finding the correct set of hyperparameters is essential for reinforcement learning and machine learning in general. One challenge was to find a set of hyperparameters which achieves a high reward in all frameworks. The author used two approaches to find a high-performing set of hyperparameters. On the one hand, the hyperparameters of the framework which performed best with the default configuration were tried and applied to every other framework. On the other hand, the author used Optuna [36], an open source hyperparameter optimization framework, to automate the search of hyperparameters in SB3 and from there applied the best found set of hyperparameters to the other frameworks. Optuna allows to automate the search for well-performing hyperparamters by specifying a set of different hyperparameters and a range of values which should be searched for within those hyperparameters. Additionally, the user is able to specify the number of iterations and for how long the framework should search the specified hyperparameter space. Optuna then tries different combinations of hyperparameters, tracks how much reward the agent accumulated for each configuration and prints the best-performing configuration at the end of the search process. During the whole process, it is essential to reduce randomness and have deterministic runs, which are only affected by the hyperparameter configuration, to be able to derive meaningful results.

3.5.4 Getting Unity ML Agents and Godot RL Agents To Run

Another challenge was getting Unity ML Agents and Godot RL Agents and its respective RL backends to install and run. Due to the nature of python and its package ecosystem and the utilization of hardware specifics to gain computational performance, the installation process is not always straightforward. For instance, the python version had to be downgraded to work with PyTorch and the correct CUDA torch version had to be installed to utilize the underlying NVIDIA graphic card. For Unity ML Agents, the author had to update its `setup.py` to allow a newer PyTorch version. As Godot RL Agents supports different RL backends and those backends sometimes have conflicting dependencies, the author had to create 2 separate virtual environments (`python -m venv venv.rllib` and `python -m venv venv.sb3.cleanrl.sf`), one for Ray's RLlib and another for Stable Baselines3, CleanRL and Sample Factory. Furthermore, not every RL backend of Godot RL Agents is supported on Windows. In order to utilize and compare every RL library, the author switched the training of the RL agents from Windows to Ubuntu (Linux).

3.5.5 Designing The Reward System

Designing the right reward system is a crucial component for the success of any reinforcement learning problem. The way a reward system is designed decides what an agent will learn in the end to maximize its expected reward and if not properly engineered, the agent might learn something, which was not intended by the designer. For instance, as an initial approach, the author of this work designed a system, in which the agent would get a negative reward of -1 every time it would fall off a platform. The reasoning behind this design decision was that it would incentivize the agent to be more careful and precise when jumping to another platform. However, this reward design actually led the agent to not move at all, as it found this to be the policy that maximizes its expected reward, which is clearly a behaviour that is misaligned with the intended goal of collecting as many coins as possible without falling off a platform.

3.5.6 Upgrading Technology

Last but not least, another challenge occurred with the Release of Godot 4.0. While the initial simulation was written for Godot 3.5, the whole simulation was rewritten from scratch with the release of Godot 4.0. This was necessary to utilize the latest features of Godot RL Agents, which continued to improve but was mainly targeting those improvements for Godot 4.0 and not for Godot 3.5.

This illustrates an additional challenge in the process, namely upgrading to newer versions of Godot RL Agents while co-developing the framework. New updates from time to time involved updates for the RL backends, which might introduce dependency conflicts between the back-

ends mentioned in section 3.5.4. Since the author was also actively developing, contributing and testing new changes to Godot RL Agents, this was a very time-consuming process, which often necessitated the implementation of fixes.

4 Evaluation

As stated in [24], it is crucial to have standardized, reproducible and meaningful reporting methods for the evaluation of deep reinforcement learning algorithms, so the significant progress of those algorithms, made in solving challenging problems, is able to continue. Furthermore, reproducibility should help to reduce misinterpretation and minimize wasted efforts originating from reproducing those results. In the following sections, the methods, which were followed while conducting the experiments, are getting described and the resulting outcomes will be presented as graphs, to be able to easily compare and visualize the differences between the two reinforcement learning frameworks in terms of learning and runtime performance.

4.1 Methods

The ultimate goal of this work is to compare the two frameworks Unity ML Agents and Godot RL Agents regarding their learning and runtime performance. Since Godot RL Agents allows to choose between four different RL backends: Ray RLlib, Sample Factory, Stable Baseline3 and CleanRL, this essentially becomes a comparison between 5 different frameworks.

According to [24], there are extrinsic factors and intrinsic factors which are affecting reproducibility. Extrinsic factors are components like hyperparameters and the underlying codebase. Whereas intrinsic factors are elements like random seeds or environment properties. To handle the extrinsic factors, a trial for each framework (for shortness sake, in the graphs referred to as Unity, Godot[RLLib], Godot[SF], Godot[SB3] and Godot[CleanRL]) will be executed with 2 different configurations of hyperparameters: the frameworks' default configuration and a configuration with shared hyperparameters. As in [24], each trial consists of 5 consecutive runs to ensure that results are not just due to randomness. Furthermore, each run will be executed for 22 minutes, as this is the duration, where most of the learning seemed to have already happened for the frameworks under the default configuration, as can be seen by the flattening of the curve in Figure 16.

In order to tackle the intrinsic factors, each run of a trial will run with a different seed. The seed ranges from 0 to 4, where the first run uses seed 0 and the last run utilizes seed 4. Each

trial utilizes the same set of seeds. The simulation of the environment itself uses a fixed seed for its random number generator to guarantee reproducibility in the otherwise stochastic environment, as described in section 3.2.2.

The 5 runs of a trial are then getting averaged and the learning performance, the runtime performance and the corresponding confidence interval will be plotted as a graph to be able to easily compare the differences between the frameworks' performance. The exact hyperparameter configurations, scripts and trial run commands, the collected data and the Unity and Godot simulation executables are provided in the form of GitHub links in the appendices A and B. This is done to allow other researchers to easily reproduce the results presented in this work. Furthermore, the source code of the simulations, written for Unity and Godot, is also open source and publicly available on GitHub to allow for an external, in-depth inspection of the comparability of the two environments.

As mentioned above, the trials were run under different configurations of hyperparameters. In the following sections, the different hyperparameter configurations of the frameworks are getting described in greater detail.

4.1.1 Default Configuration

Unity ML Agents and Godot RL Agents and its respective RL backends provide a default configuration of hyperparameters, so users and researchers can easily get started with training deep RL agents without the need to specifically set every single hyperparameter. Some of these default hyperparameters vary a lot between the different frameworks and thereby potentially affect the initial learning and runtime performance of the agents. In this part of the experiment, the frameworks will be compared on how well they perform under their default configuration on the given reinforcement problem.

4.1.2 Shared Configuration

As a next step, a certain set of hyperparameters is being adjusted to be the same across all the different frameworks. This is done to be able to compare the performance of the different frameworks in a potentially fair manner. The hyperparameters, which are adjusted for the shared configuration and their corresponding values, are:

Table 2: Shared Config: Adjusted hyperparameters

Buffer Size	Mini Batch Size	Steps	Epochs	Learning Rate	Beta	Gamma	Epsilon
16384	128	128	3	0.0003	0.005	0.99	0.2

Other adjustable hyperparameters are not taken further into account, as they are not available in every framework or exceed the scope of this work. Furthermore, the hyperparameters which seemed to have the most impact during hyperparameter search were buffer size, mini batch size and number of steps.

4.2 Results

In the following section, the results, which were found while conducting the experiments, are getting presented and visualized and the frameworks' differences in runtime and learning performance are getting revealed.

4.2.1 Default Configuration

This section visualizes the trials under the default hyperparameter configuration. Table 3 displays the mean reward and 95% confidence interval for each framework after around 2, 12 and 22 minutes of training. The graphs display the mean reward accumulated by each framework during 22 minutes of training. Figure 14 presents all 5 individual runs for each framework, while Figure 15 and 16 showcases and compares the averaged results of all the different frameworks. Since one run of Unity ML Agents did not learn the optimal policy but got stuck in a local optimum, the confidence interval is very high as can be seen in Figure 16. This high confidence interval overlaps with all the other learning curves, making the graph harder to read and compare. For that reason, Figure 15 was introduced, which presents the performance of the different frameworks without any confidence interval, making the graph easier to compare.

Table 3: Default Config: Mean reward + 95% confidence interval after around 2, 12 & 22 minutes of training

Duration	Godot[CleanRL]	Godot[RLlib]	Godot[SB3]	Godot[SF]	Unity ML
2m	584 (54, 1115)	22 (11, 33)	190 (85, 295)	2946 (1668, 4224)	64 (-22, 150))
12m	2474 (569, 4379)	27 (14, 41)	3090 (978, 5202)	8827 (5739, 11914)	8510 (-1519, 18539)
22m	3527 (-1048, 8103)	32 (0, 65)	3558 (1402, 5714)	10383 (9661, 11106)	9975 (-1854, 21803)

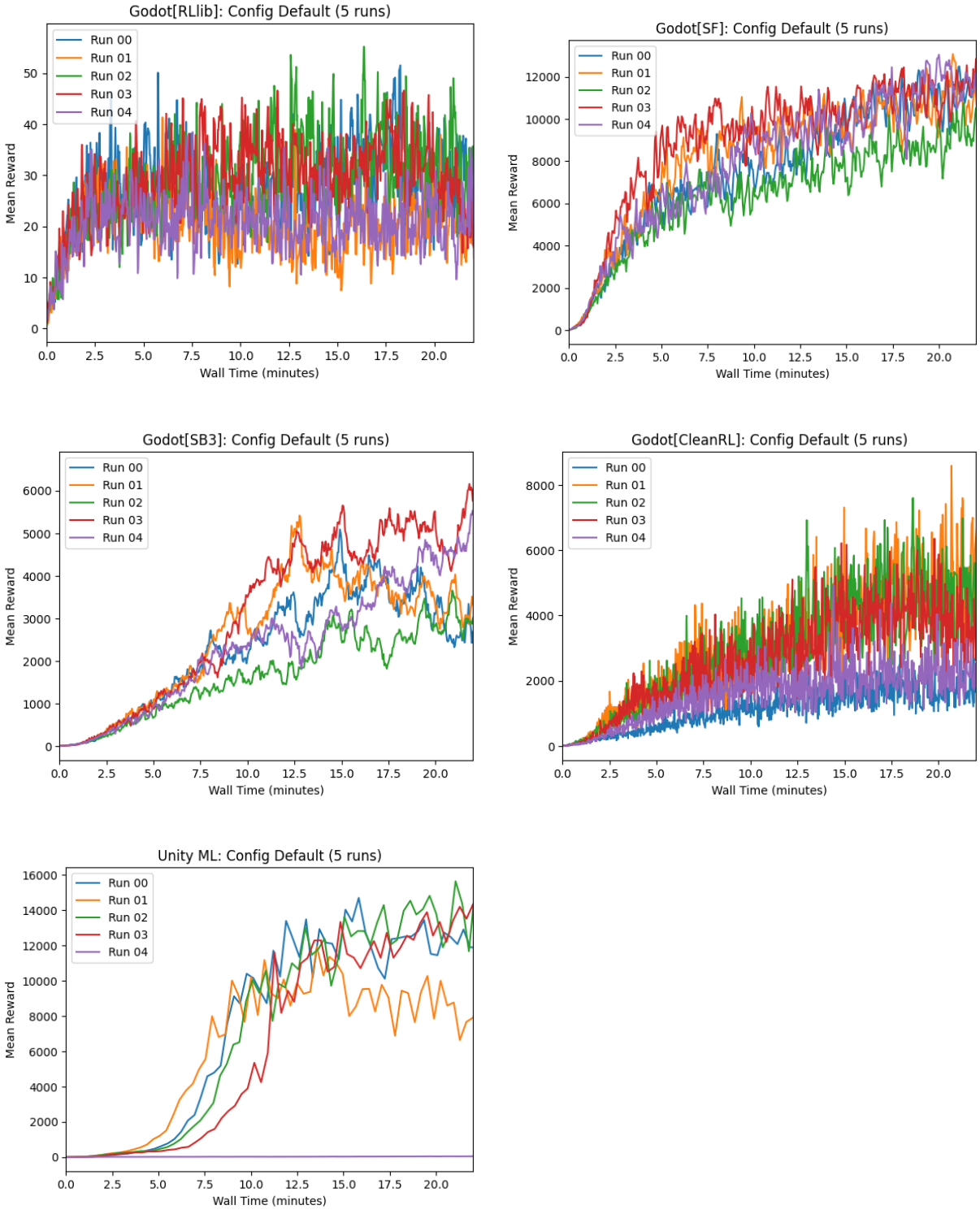

Figure 14: Individual Runs | Default Config

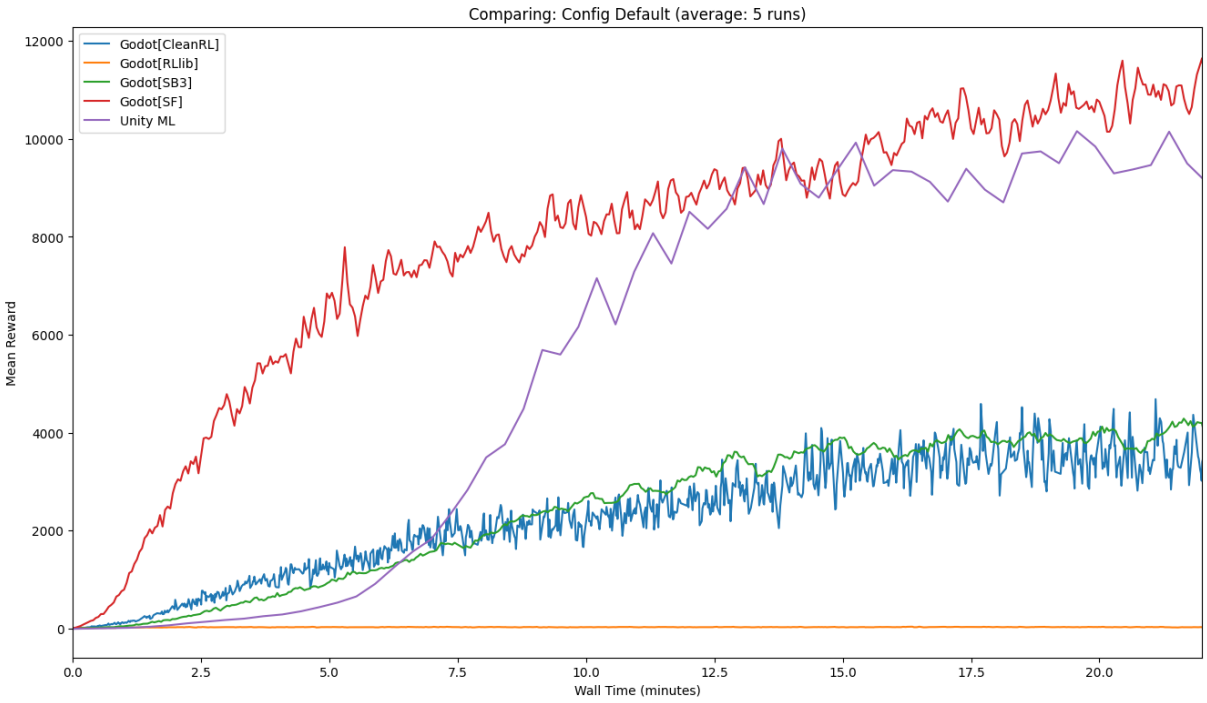

Figure 15: Comparing Frameworks | Default Config | Without Confidence Intervals

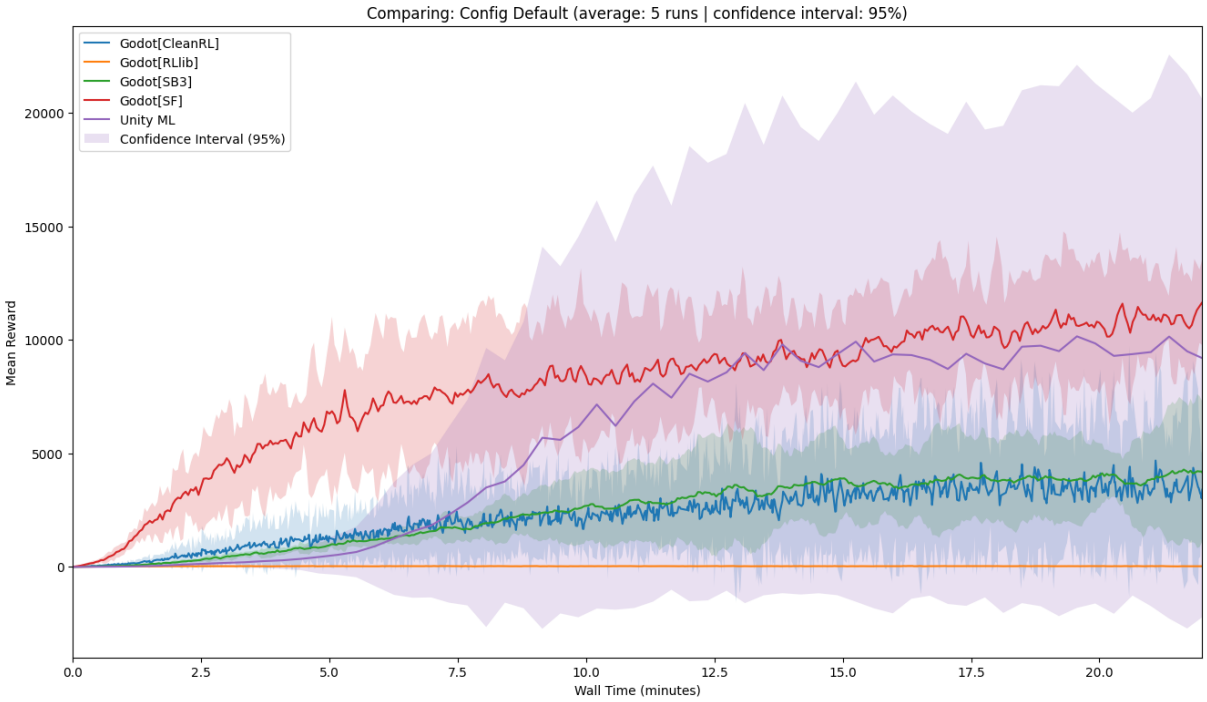

Figure 16: Comparing Frameworks | Default Config | With 95% Confidence Intervals

4.2.2 Shared Configuration

This section illustrates the results of the experiments conducted using the same hyperparameter configuration. Table 4 displays the mean reward and 95% confidence interval for each framework after around 2, 12 and 22 minutes of training. The graphs display the mean reward accumulated by each framework in 22 minutes of training. Figure 17 presents all 5 individual runs for each framework, while Figure 18 compares the averaged results of all the different frameworks without confidence intervals and Figure 19 displays it with 95% confidence intervals.

Table 4: Shared Config: Mean reward + 95% confidence interval after around 2, 12 & 22 minutes of training

Duration	Godot[CleanRL]	Godot[RLlib]	Godot[SB3]	Godot[SF]	Unity ML
2m	114 (15, 214)	5 (1, 10)	142 (94, 190)	169 (103, 235)	35 (7, 63)
12m	1787 (492, 3081)	126 (26, 226)	3846 (1977, 5715)	2675 (1476, 3875)	10042 (5224, 14859)
22m	3353 (1765, 4941)	1163 (308, 2017)	6007 (4561, 7453)	3935 (1675, 6195)	15488 (14289, 16687)

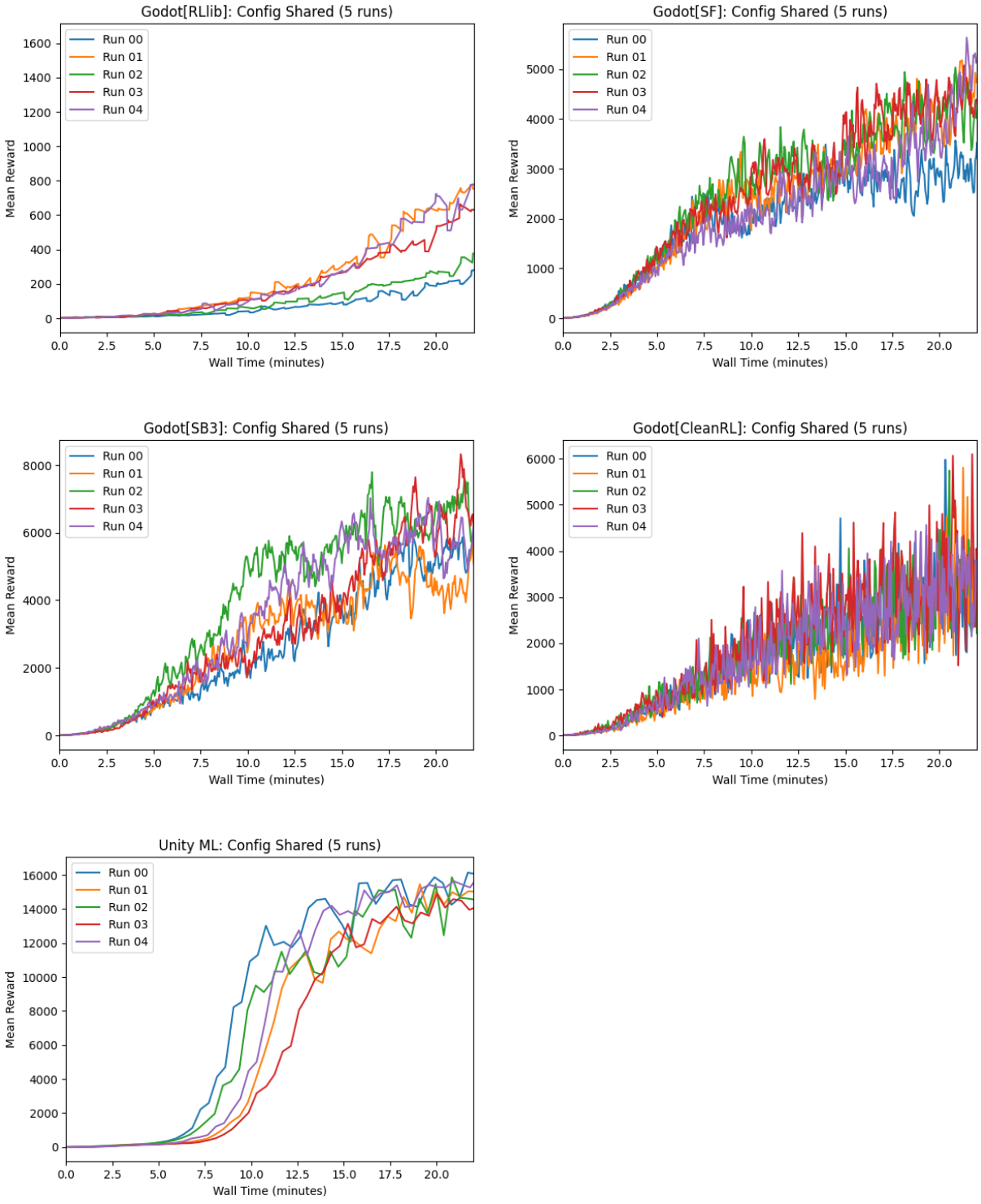

Figure 17: Individual Runs | Shared Config

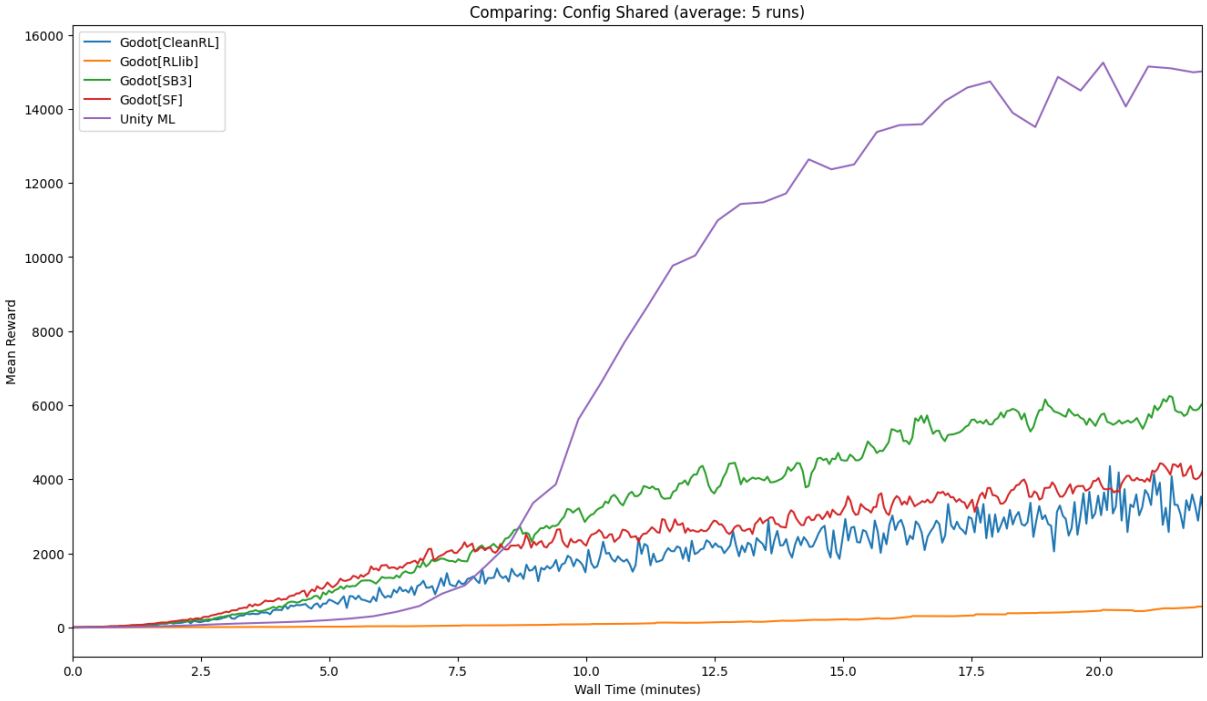

Figure 18: Comparing Frameworks | Shared Config | Without Confidence Intervals

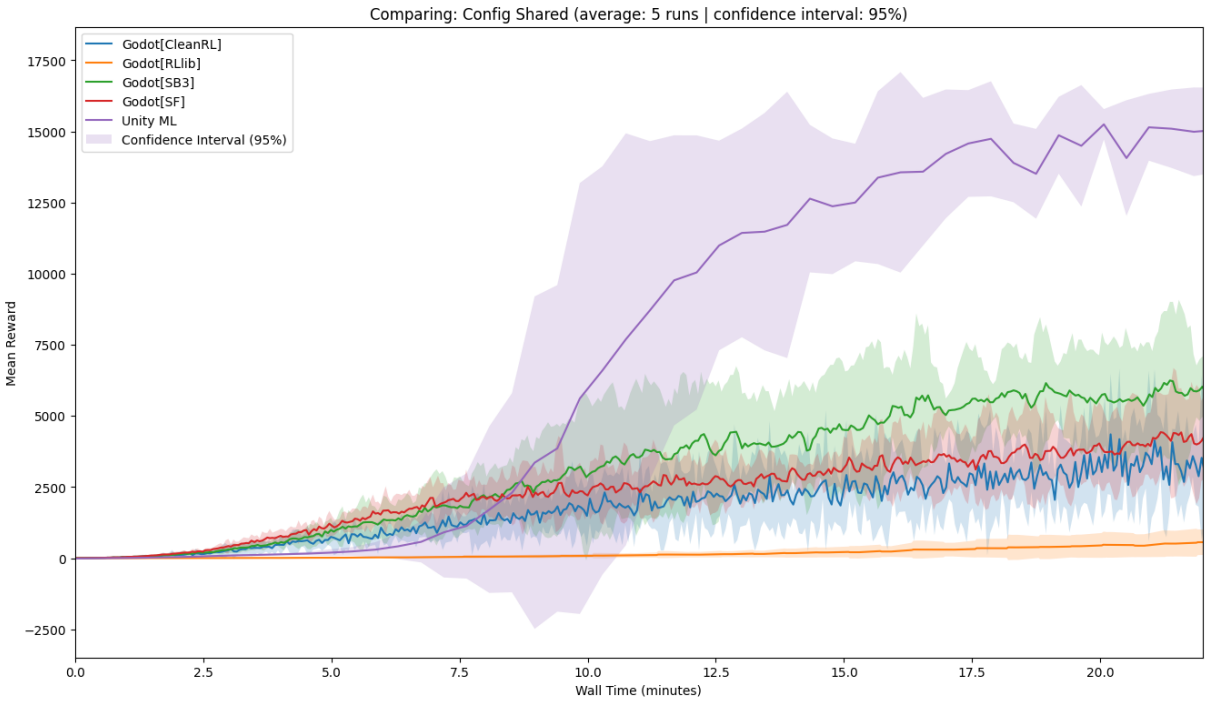

Figure 19: Comparing Frameworks | Shared Config | With 95% Confidence Intervals

4.2.3 Additional Findings

This section presents additional findings which occurred during the conduction of the trials. Figure 20 compares the time-steps of the frameworks after 22 minutes of training and Figure 21 and Table 5 display the mean reward for each individual framework after 1.000.000 time-steps of training.

Table 5: Default Config: Mean reward + 95% confidence interval after 1.000.000 time-steps

Godot[CleanRL]	Godot[RLlib]	Godot[SB3]	Godot[SF]	Unity ML
896 (-158, 1950)	26 (18, 34)	2777 (1052, 4502)	1131 (743, 1519)	1823 (-1349, 4994)

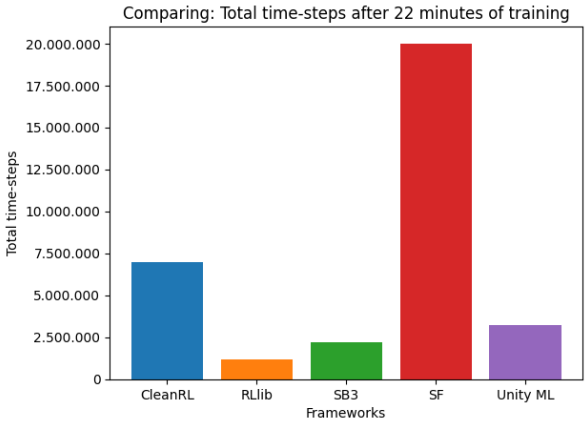

Figure 20: Comparing total time-steps

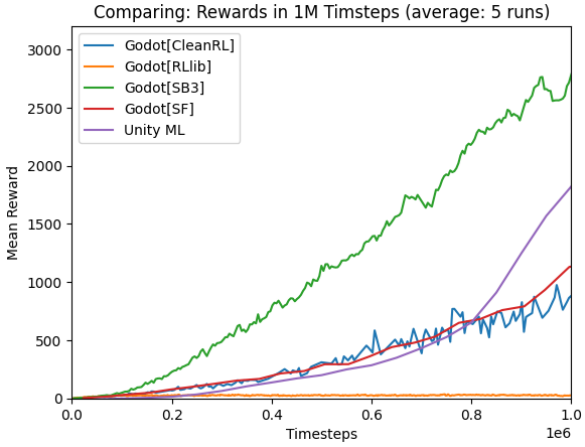

Figure 21: Comparing rewards after 1.000.000 time-steps

5 Discussion

In the following sections the results, which were presented in 4.2, are getting discussed. The framework differences in runtime and learning performance under the default and shared hyperparameter configuration are getting discussed and interesting parts are getting highlighted. Furthermore, additional findings, which have been found during the experiments, will be presented.

5.1 Default Configuration

Running the trials under the frameworks' default hyperparameter configuration revealed that Unity ML Agents and Godot RL Agents utilizing Sample Factory as RL backend (Godot RL Agents[SF]) performed the best among all other frameworks, as can be seen in Figure 16. Both Unity ML Agents and Godot RL Agents[SF] achieve a mean reward of around 10.000 after running 22 minutes. It is notable that Godot RL Agents[SF] manages to learn a lot faster than Unity ML Agents and all the other frameworks. After around 2 minutes, it already managed to gather a mean reward of 3.000 while Unity ML Agents is, at that time, at a mean reward of 64, making Godot RL Agents[SF], at that point in time, 46 times faster than Unity ML Agents. After around 14 minutes, Unity ML Agents and Godot RL Agents[SF] achieve the same accumulated reward of around 10.000.

It is noteworthy that even though some individual runs of Unity ML Agents achieved slightly higher rewards (the best run was around 14.200, as shown in Figure 14) than Godot RL Agents[SF] (the best run was around 11.700, as can be seen in Figure 14) - one run of Unity ML Agents (Run 04) did not manage to learn the optimal policy but rather got stuck in a local optimum. When reviewing the learned behaviour of Run 04 via inference in the Unity game engine, it was revealed that the agent has not learned to use its sensors properly, but would rather jump backwards into roughly the direction of the next platform, "hoping" that it will land. This might indicate that the default configuration of Unity ML Agents for this specific reinforcement learning problem could be more sensible to factors like random weight initialization, but it cannot be said for sure that this sub-optimal behaviour was not due to a bug in the created simulation because none of the Godot frameworks generate such extreme differences between the runs. This failed run is also responsible for the high confidence interval for the Unity ML Agents trial, indicating that more runs are necessary to confidently measure the real performance of Unity ML Agents under the default hyperparameter settings.

It is also worth mentioning that Godot RL Agents[RLlib] under the default hyperparameter configuration does not learn the intended behaviour. It accumulates in 22 minutes a mean reward of 30, indicating that the configuration of hyperparameters needs some adaptation to achieve better or comparable results. Godot RL Agents[CleanRL] and Godot RL Agents[SB3] manage a mean reward of 3.500, placing them at the same place, in terms of learning performance under the default configuration. Furthermore, the trend of the graphs, namely the flattening of the curves, in Figure 16 suggests that training has stabilized for all frameworks and that they achieved their maximum expected return in this default configuration of hyperparameters. This however can only be answered definitely with longer runs of the trials.

5.2 Shared Configuration

The results of the trials under the shared hyperparameter configuration reveal some further aspects. On the one hand, the similar hyperparameters did not achieve similar performance across all frameworks. Unity ML Agents performs really well in this configuration, achieving an accumulated reward of around 15.000 and thereby increasing its mean reward by 5.000, compared to the default configuration. It is also notable that during the runs with shared configuration, Unity ML Agents did not get stuck in a local optimum but always learned an optimal policy. Additional runs would be needed to see if this is by chance or if the new configuration generally helps the agent learn better than the default configuration.

While Unity ML Agents learning performance increased, the shared configuration had the opposite effect on Godot RL Agents[SF]. During the trials with the shared configuration Godot RL Agents [SF] only achieved an average reward of 4.000, thereby decreasing its mean reward by 6.000, compared to the trial with the default configuration. This seems to suggest that Unity ML Agents is much more robust to changes in the hyperparameters than Godot RL Agents[SF], but further experiments would be needed.

Finally, the learning performance of all the other Godot RL Agents frameworks improved slightly or stayed the same under the shared configuration. Godot RL Agents[SB3] received an accumulated reward of 6.000, making it perform better than Godot RL Agents[SF] in this configuration of hyperparameters. Godot RL Agent[CleanRL] achieved a mean reward of around 3.300, which is similar to the run under the default configuration. Even though Godot RL Agents[RLlib] performed relatively bad in this configuration compared to the other frameworks, it still managed to achieve an accumulated return of around 1.163, making it perform 36 times better than during the default configuration. Differently from the default configuration, there still seems to be some learning going on after 22 minutes for most of the frameworks, as can be seen in the graphs of Figure 19.

5.3 Additional Findings

A noteworthy discovery while conducting the experiments is that the frameworks completed significantly different numbers of total time-steps for the same amount of time, as can be seen in Figure 20. As an example: while it took Godot RL Agents[RLlib] around 7 minutes to complete 1.000.000 time-steps, it took Godot RL Agents[SF] only 80 seconds. This is relevant, in that it shows that the default stopping condition for such experiments, which is often a certain amount of total time-steps, does not allow for a fair comparison as can be seen in Figure 21. Looking at Figure 21, Godot RL Agents[SB3] seems to perform much better than most of the other frameworks, even though it took Godot RL Agents[SB3] around 10 minutes to reach

1.000.000 time-steps, while it took Godot RL Agents[SF] only 80 seconds. Looking at Figure 16, it becomes apparent that Godot RL Agents[SF] has reached a mean return of 8.000 after 10 minutes of training, making it actually perform better than Godot RL Agents[SB3] under the default configuration. The duration a machine learning algorithm needs to train to achieve the desired results is relevant for multiple reasons. On the one hand, a faster machine learning algorithm allows for more iterations thereby offering the possibility to search the hyperparameter space quicker and finding an optimal configuration much faster. On the other hand, most serious machine learning experiments are executed on server clusters in the cloud, which involve payments for every hour the server runs. A faster machine learning algorithm can allow to reduce those costs drastically. Last but not least, from an environmental view, faster algorithms at times also need less energy and electricity, as they can achieve similar results in way less time, thereby decreasing the potential carbon footprint. Because of all these mentioned reasons, training time is usually a metric which is more important for researchers, who apply machine learning algorithms, than the total time-steps. This is why the relative wall time became one of the main metrics and stopping condition for the trials of this work.

6 Conclusion

During the evaluation of the trials, a few important findings and differences in the reinforcement learning frameworks became apparent. On the one hand, Unity ML Agents and Godot RL Agents[SF] achieved similar high results, while the results of Godot RL Agents[SB3] and Godot RL Agents[CLeanRL] were also high, but still lower overall. Godo RL Agents[RLlib] achieved the least rewards over all trials. While Godot RL Agents[SF] proved to be the fastest of all the frameworks, at some point 20 times faster, it also seemed to be more sensible to hyperparameter changes, varying in learning performance drastically, while Unity ML Agents seemed to be more robust to those changes. Furthermore, one run of the Unity ML Agents default configuration trial only reached a local optimum, not learning the intended behaviour. This did not happen for any of the Godot RL Agents frameworks, once a good hyperparameter configuration was found. It cannot be said for sure if the learned sub-optimal behaviour is due to Unity ML Agents or a bug inside of the simulation.

Furthermore, it became evident that a shared and similar hyperparameter configuration does not guarantee a similar learning performance. While the mean learning performance of Unity ML Agents became better, the mean learning performance of Godot RL Agents[SF] became a lot worse. This indicates that individual frameworks need custom hyperparamter configurations, specifically fitted to them, even if they use the same algorithms under the hood. This is

probably due to differences in implementation details or potentially due to differences in other default hyperparameters, which were not adjusted during the experiment.

Finally, it also became apparent during the conduction of the experiments that the default stopping condition of reinforcement learning experiments, the total number of time-steps, does not guarantee a fair and meaningful comparison, as frameworks differ a lot in the finished time-steps per second, and that the relative wall time allows for a more meaningful comparison.

Considering the research question: "How do the reinforcement learning frameworks Unity ML-Agents Toolkit and Godot RL Agents differ in terms of learning performance and computational runtime performance", it can be said that Unity ML Agents and Godot RL Agents achieve similar results in terms of learning performance when utilizing the Godot RL Agents' backend Sample Factory. In terms of runtime performance, however, Godot RL Agents[SF] achieves higher results much faster than any of the other frameworks, especially in the first minutes of training. On the other hand, Unity ML Agents seems to be more robust and performing well under different hyperparameter configurations whereas Godot RL Agents[SF] is more sensible to those changes. At the same time, Unity ML Agents does not always achieve to learn an optimal policy under the default configuration, making it more unstable in terms of individual runs. The other RL backends of Godot RL Agents achieve decent results regarding learning performance but not near as high as Unity ML Agents and Godot RL Agents[SF]. Godot RL Agents[RLlib] was the RL backend which performed worst under the hyperparameter configurations of the trials.

While the experiments of the different frameworks revealed big differences in the total accumulated reward, the important factor, in the end, is how well the agent actually behaves in the simulation or in the real world. When evaluating those results within the simulation and actually watching the agents solve the problem, it becomes apparent that the agents perform really well regardless if they accumulate a mean reward of 10.0000 or 5.000 and therefore most of the frameworks are suitable for achieving the desired goal in this reinforcement problem. However, in terms of quick iterations, short wall clock runtime and reduced costs, Godot RL Agents[SF] is the clear winner among those frameworks, as it was up to 46 times faster in the first minutes of the experiment. Unity ML Agents on the other hand worked considerably well under various hyperparameter configurations, indicating its robustness.

6.1 Future Work

Even though a lot of care was taken into account to guarantee comparability between Unity ML Agents and Godot RL Agents and its respective reinforcement learning backends, there are still some open points which can be addressed in future research. On the one hand, a

more thorough search of the hyperparameter space for each individual framework, resulting in a custom-fitted hyperparameter configuration, might reveal even more comparable results in terms of total return. On the other hand, regarding the shared hyperparamter configuration, further experiments could be run, where every possible hyperparameter is adjusted to be the same, to confidently prove or disprove that a shared configuration does not reach similar results. Furthermore, it would be interesting to conduct more experiments in different environments to see if the frameworks still achieve the same results in terms of their current ranking or if significant differences arise. Last but not least, more trials could be run with longer duration to see if training continues to improve and to get to the core of the high variance of the Unity ML Agents trials under the default configuration.

6.2 Competing Interests Statement

This thesis compares Unity ML Agents with Godot RL Agents. The author of this thesis has contributed to the development of Godot RL Agents.

Bibliography

[1] OpenAI. *GPT-4 Technical Report*. Mar. 27, 2023. arXiv: 2303.08774[cs]. URL: http://arxiv.org/abs/2303.08774 (visited on 06/18/2023).

[2] Mariusz Bojarski et al. *End to End Learning for Self-Driving Cars*. Apr. 25, 2016. DOI: 10.48550/arXiv.1604.07316. arXiv: 1604.07316[cs]. URL: http://arxiv.org/abs/1604.07316 (visited on 06/23/2023).

[3] John Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (Aug. 2021). Number: 7873 Publisher: Nature Publishing Group, pp. 583–589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2. URL: https://www.nature.com/articles/s41586-021-03819-2 (visited on 06/16/2023).

[4] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Dec. 19, 2013. arXiv: 1312.5602[cs]. URL: http://arxiv.org/abs/1312.5602 (visited on 09/06/2023).

[5] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550.7676 (Oct. 2017). Number: 7676 Publisher: Nature Publishing Group, pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270. URL: https://www.nature.com/articles/nature24270 (visited on 01/08/2023).

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Second edition. Adaptive computation and machine learning series. Cambridge, Massachusetts: The MIT Press, 2018. 526 pp. ISBN: 978-0-262-03924-6.

[7] Yiheng Liu et al. *Summary of ChatGPT/GPT-4 Research and Perspective Towards the Future of Large Language Models*. May 10, 2023. arXiv: 2304.01852[cs]. URL: http://arxiv.org/abs/2304.01852 (visited on 06/18/2023).

[8] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (Dec. 7, 2018). Publisher: American Association for the Advancement of Science, pp. 1140–1144. DOI: 10.1126/science.aar6404. URL: https://www.science.org/doi/10.1126/science.aar6404 (visited on 10/25/2022).

[9] Ahmed Khalifa et al. *PCGRL: Procedural Content Generation via Reinforcement Learning*. Aug. 12, 2020. DOI: 10.48550/arXiv.2001.09212. arXiv: 2001.09212[cs,stat]. URL: http://arxiv.org/abs/2001.09212 (visited on 10/25/2022).

[10] Jens Kober, J. Andrew Bagnell, and Jan Peters. “Reinforcement learning in robotics: A survey”. In: *The International Journal of Robotics Research* 32.11 (Sept. 1, 2013). Publisher: SAGE Publications Ltd STM, pp. 1238–1274. ISSN: 0278-3649. DOI: 10.1177/0278364913495721. URL: https://doi.org/10.1177/0278364913495721 (visited on 06/23/2023).

[11] Oriol Vinyals et al. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. In: *Nature* 575.7782 (Nov. 2019). Number: 7782 Publisher: Nature Publishing Group, pp. 350–354. ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. URL: https://www.nature.com/articles/s41586-019-1724-z (visited on 10/25/2022).

[12] Błażej Osiński et al. “Simulation-Based Reinforcement Learning for Real-World Autonomous Driving”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020 IEEE International Conference on Robotics and Automation (ICRA). ISSN: 2577-087X. May 2020, pp. 6411–6418. DOI: 10.1109/ICRA40945.2020.9196730.

[13] OpenAI et al. *Solving Rubik's Cube with a Robot Hand*. Oct. 15, 2019. arXiv: 1910.07113[cs,stat]. URL: http://arxiv.org/abs/1910.07113 (visited on 06/14/2023).

[14] Linus Gisslen et al. “Adversarial Reinforcement Learning for Procedural Content Generation”. In: *2021 IEEE Conference on Games (CoG)*. 2021 IEEE Conference on Games (CoG). Copenhagen, Denmark: IEEE, Aug. 17, 2021, pp. 1–8. ISBN: 978-1-66543-886-5. DOI: 10.1109/CoG52621.2021.9619053. URL: https://ieeexplore.ieee.org/document/9619053/ (visited on 10/25/2022).

[15] Arthur Juliani et al. *Unity: A General Platform for Intelligent Agents*. May 6, 2020. DOI: 10.48550/arXiv.1809.02627. arXiv: 1809.02627[cs,stat]. URL: http://arxiv.org/abs/1809.02627 (visited on 10/25/2022).

[16] Edward Beeching et al. *Godot Reinforcement Learning Agents*. version: 1. Dec. 7, 2021. DOI: 10.48550/arXiv.2112.03636. arXiv: 2112.03636[cs]. URL: http://arxiv.org/abs/2112.03636 (visited on 10/25/2022).

[17] Greg Brockman et al. *OpenAI Gym*. June 5, 2016. DOI: 10.48550/arXiv.1606.01540. arXiv: 1606.01540[cs]. URL: http://arxiv.org/abs/1606.01540 (visited on 06/23/2023).

[18] Antonin Raffin et al. “Stable-Baselines3: Reliable Reinforcement Learning Implementations”. In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. ISSN: 1533-7928. URL: http://jmlr.org/papers/v22/20-1364.html (visited on 06/16/2023).

[19] Eric Liang et al. *RLlib: Abstractions for Distributed Reinforcement Learning*. June 28, 2018. arXiv: 1712.09381[cs]. URL: http://arxiv.org/abs/1712.09381 (visited on 06/16/2023).

[20] Aleksei Petrenko et al. *Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning*. June 22, 2020. arXiv: 2006.11751[cs,stat]. URL: http://arxiv.org/abs/2006.11751 (visited on 06/16/2023).

[21] Shengyi Huang et al. *CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms*. Nov. 16, 2021. arXiv: 2111.08819[cs]. URL: http://arxiv.org/abs/2111.08819 (visited on 06/16/2023).

[22] Rui Pereira et al. "Ranking programming languages by energy efficiency". In: *Science of Computer Programming* 205 (May 2021), p. 102609. ISSN: 01676423. DOI: 10.1016/j.scico.2021.102609. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642321000022 (visited on 08/24/2023).

[23] John Schulman et al. *Proximal Policy Optimization Algorithms*. Aug. 28, 2017. arXiv: 1707.06347[cs]. URL: http://arxiv.org/abs/1707.06347 (visited on 06/17/2023).

[24] Peter Henderson et al. *Deep Reinforcement Learning that Matters*. Jan. 29, 2019. DOI: 10.48550/arXiv.1709.06560. arXiv: 1709.06560[cs,stat]. URL: http://arxiv.org/abs/1709.06560 (visited on 07/09/2023).

[25] David Silver. *Lectures on Reinforcement Learning*. 2015. URL: https://www.davidsilver.uk/teaching/.

[26] Melvin M. Vopson. "Estimation of the information contained in the visible matter of the universe". In: *AIP Advances* 11.10 (Oct. 19, 2021), p. 105317. ISSN: 2158-3226. DOI: 10.1063/5.0064475. URL: https://doi.org/10.1063/5.0064475 (visited on 08/22/2023).

[27] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015). Number: 7540 Publisher: Nature Publishing Group, pp. 529–533. ISSN: 1476-4687. DOI: 10.1038/nature14236. URL: https://www.nature.com/articles/nature14236 (visited on 08/17/2023).

[28] John Schulman et al. *Trust Region Policy Optimization*. Apr. 20, 2017. arXiv: 1502.05477[cs]. URL: http://arxiv.org/abs/1502.05477 (visited on 08/19/2023).

[29] Martín Abadi et al. *TensorFlow: A system for large-scale machine learning*. May 31, 2016. arXiv: 1605.08695[cs]. URL: http://arxiv.org/abs/1605.08695 (visited on 08/22/2023).

[30] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Dec. 3, 2019. arXiv: 1912.01703[cs,stat]. URL: http://arxiv.org/abs/1912.01703 (visited on 08/22/2023).

[31] Tuomas Haarnoja et al. *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. Aug. 8, 2018. arXiv: 1801.01290[cs,stat]. URL: http://arxiv.org/abs/1801.01290 (visited on 06/17/2023).

[32] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[33] J. Pearl. "Heuristics: Intelligent search strategies for computer problem solving". In: (Jan. 1, 1984). Publisher: Addison-Wesley Pub. Co., Inc.,Reading, MA. URL: https://www.osti.gov/biblio/5127296 (visited on 06/20/2023).

[34] Mohit Pandey et al. “The transformational role of GPU computing and deep learning in drug discovery”. In: *Nature Machine Intelligence* 4.3 (Mar. 2022). Number: 3 Publisher: Nature Publishing Group, pp. 211–221. ISSN: 2522-5839. DOI: 10.1038/s42256-022-00463-x. URL: https://www.nature.com/articles/s42256-022-00463-x (visited on 06/18/2023).

[35] Sebastian Risi and Julian Togelius. “Increasing generality in machine learning through procedural content generation”. In: *Nature Machine Intelligence* 2.8 (Aug. 2020). Number: 8 Publisher: Nature Publishing Group, pp. 428–436. ISSN: 2522-5839. DOI: 10.1038/s42256-020-0208-z. URL: https://www.nature.com/articles/s42256-020-0208-z (visited on 10/25/2022).

[36] Takuya Akiba et al. *Optuna: A Next-generation Hyperparameter Optimization Framework*. July 25, 2019. arXiv: 1907.10902[cs,stat]. URL: http://arxiv.org/abs/1907.10902 (visited on 09/06/2023).

List of Figures

List of Tables

List of source codes

List of Abbreviations

AI artificial intelligence

API application programming interface

ANN artificial neural network

CNN convolutional neural network

CPU central processing unit

CUDA compute unified device architecture

DRL deep reinforcement learning

FPS frames per second

GB gigabyte

GLIE Greedy in the Limit with Infinite Exploration

GPU graphics processing unit

KL Kullback-Leibler

LLM large language model

MC Monte Carlo

MDP Markov decision process

ML machine learning

MSE mean squared error

NPC non-player character

OS operating system

PPO Proximal Policy Optimization

QA quality assurance

RL reinforcement learning

RLlib Ray RLlib

RAM random access memory

SAC Soft Actor Critic

SB3 Stable Baselines3

SGD stochastic gradient descent

SF Sample Factory

SSD solid-state-drive

TB terabyte

TCP Transmission Control Protocol

TD temporal difference

TRPO Trust Region Policy Optimization

VRAM video random access memory

A Source Code & Builds

Appendix A includes the links to the source code of the simulations, written in Unity (1) and Godot (2). Furthermore it also contains the builds and individual hyperparameter configurations for the trials in Unity ML Agents (3) and Godot RL Agents (4).

1. https://github.com/visuallization/unity-ml-platformer
2. https://github.com/visuallization/godot-4-ml-platformer
3. https://github.com/visuallization/unity_ml_platformer_experiments
4. https://github.com/visuallization/godot_4_ml_platformer_experiments

B Data & Graphs

Appendix B contains the link to the collected data of the experiments and the source code of the plot visualization, written in python.

1. https://github.com/visuallization/rl_master_thesis_evaluation