Master Thesis
for the degree
Master of Arts (M.A.)

THE USE OF MACHINE LEARNING ALGORITHMS IN THE PROCESS OF GAME DEVELOPMENT TO FIND BUGS AND UNWANTED GAME DESIGN FLAWS

21st June, 2021

Julius Hartmann
HAW Hamburg
Faculty of Design, Media and Information (DMI)
Department of Media Technology
Matriculation number: 2425905
julius.hartmann@haw-hamburg.de

under the guidance of

Prof. Dr. Sabine Schumann
and
Prof. Dr. Stephan Pareigis

Title

The use of machine learning algorithms in the process of game development to find bugs and unwanted game design flaws

Keywords

Artificial Intelligence, Machine Learning, Deep Reinforcement Learning, Proximal Policy Optimization (PPO), Behavioral Cloning (BC), Generative Adversarial Imitation Learning (GAIL), Curiosity, Game Development, Game Design, Quality Assurance

Abstract

The game development process involves many different disciplines, ranging from visual arts, sound design, game design, product management, frontend- and backend development and many more. All of which are contributing to a single project to create the best possible player experience. The player experience can easily be interrupted by bugs or imbalanced game design. For example if a strategy game has many types of buildings, to which one outperforms all others, it will be highly likely that players would use this advantage and adopt a strategy that may reduce the usage of the surrounding buildings. The game would become monotonous instead of diverse and exciting. This work covers exploratory research whether machine learning can be used during the process of game development by having an Artificial Intelligence agent testing the game. Previous studies from other researchers have shown that Reinforcement Learning (RL) agents are capable to play finished games on super human level. However, none of them are used to improve the games themselves as they treat the environment as unchangeable, but are still facing the problem that game environments often are not flawless. To improve the game itself, this work takes advantage of the reinforcement learning algorithms to unveil the bugs and game design errors in the environment. Within this work the novelty deep reinforcement learning algorithm Proximal Policy Optimization [1] will be applied to a complex mobile game in its production cycle. The machine learning agent is developed in parallel to the game. As a result, the trained agent shows positive indications that the development process can be supported by machine learning. For a conclusive evaluation the additional implementation effort needs to be taken into account right from the beginning.

Titel

Die Anwendung von Machine Learning Algorithmen im Entwicklungsprozess von Spielen zum Finden von Fehlern im Spieldesign oder der Implementierung.

Stichwörter

Künstliche Intelligenz, Maschinelles Lernen, Deep Reinforcement Learning, Proximal Policy Optimization (PPO), Behavioral Cloning (BC), Generative Adversarial Imitation Learning (GAIL), Curiosity, Spiele Entwicklung, Game Design, Qualitätssicherung

Zusammenfassung

In den Entwicklungsprozess von Video-Spielen sind viele verschiedene Disziplinen involviert. Dazu gehören Konzept-Artists, Game Designer, Produktmanager, Frontend- und Backendentwickler und viele mehr, die alle darauf hinarbeiten dem Spieler eine einzigartige Spielerfahrung zu ermöglichen. Fehler im Spiel oder eine unverhältnismäßige Spielschwierigkeit kann dabei den Spielfluss des Spielers schnell unterbrechen. Wenn beispielsweise in einem Strategiespiel mit verschiedenen Gebäuden eins davon besonders effizient ist, würde dies dazu führen, dass eine Strategie ohne dieses Gebäude nicht mehr in Betracht gezogen wird. In Folge wird die Diversität der Strategien auf ein Gebäude reduziert, wodurch das Spiel langweiliger wird. In dieser Arbeit wird erforscht wie Machine Learning Algorithmen im Entwicklungsprozess unterstützend eingesetzt werden können, um Video-Spiele auf die genannten Probleme zu testen. Andere wissenschaftliche Arbeiten im Bereich Machine Learning zeigen bereits, dass Agenten die mit Reinforcement Leanring trainiert wurden in der Lage sind Video-Spiele auf übermenschlichem Niveau zu spielen. Trotzdem wurden die Agenten bisher nur auf fertig entwickelten Spielen getestet mit dem Ziel die Künstliche Intelligenz zu verbessern und nicht das Spiel selbst, obwohl auch fertige Spiele meist noch Fehler aufweisen. Diese Arbeit zielt darauf ab die Vorteile der Reinforcment Learning Algorithmen zu nutzen um Fehler im Spieldesign und der Implementierung aufzudecken. Einer dem heutigen Standard entsprechender RL Algorithmus ist der Proximal Policy Optimization [1] Algorithmus. Dieser wird auf ein Handy-Spiel angewendet, welches sich in der Entwicklung befindet. Die Ergebnisse dieser Arbeit zeigen das mit genügend Implementierungsaufwand der trainierte Agent in der Lage ist Fehler im Spiel Design zu finden.

Die folgende textliche Ausarbeitung ist in englischer Sprache verfasst.

Contents

Page

List of Figures VI

List of Tables VI

List of Algorithms VII

Listings VII

Glossary IX

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Related Work 2
- 1.3 Scope 4
- 1.4 Structure 5

2 Foundations 7
- 2.1 The Game 7
- 2.2 Game Development Processes 10
  - 2.2.1 Embodiment of Machine Learning in the development process 11
  - 2.2.2 Game Design 11
  - 2.2.3 Quality Assistance (QA) 12
- 2.3 Machine Learning 13
  - 2.3.1 Artificial Neural Networks 13
  - 2.3.2 Backpropagation 14
  - 2.3.3 Supervised Learning 15
  - 2.3.4 Reinforcement Learning 15
  - 2.3.5 Curriculum Learning 19
  - 2.3.6 Curiosity 20
  - 2.3.7 Behavioral Cloning 21
  - 2.3.8 Generative Adversarial Imitation Learning 21
- 2.4 Unity Engine 21
  - 2.4.1 Unity Work Environment 22
  - 2.4.2 ML-Agents Toolkit 22

3 Implementation **27**
3.1 Application Preparation . . . 27
3.2 Software Architecture . . . 28
3.3 Actions . . . 32
3.4 Action Masking . . . 34
3.5 Observations . . . 34
3.6 Rewards . . . 35
3.7 Demo Actions . . . 37
3.8 Training Setup . . . 38

4 Evaluation **41**
4.1 Continuous Development . . . 41
4.2 Training Results . . . 41
4.2.1 Disabling Curiosity . . . 42
4.2.2 Curriculum Learning . . . 42
4.2.3 Stacked Observations . . . 44
4.2.4 Observed Agent Behavior . . . 45
4.2.5 Difficulties . . . 50

5 Conclusion **53**
5.1 Game Design Suggestions . . . 53
5.2 Quality Value . . . 53
5.3 Generality . . . 54
5.4 Training Time and Scaling . . . 55
5.5 Future Work . . . 55

List of Figures

1.1 Exploiting potential energy to “jump”. Image from J. Lehman et al. [19] 3

2.1 Gathering panel from a player perspective 8
2.2 Core loop of the game *Lost Survivors* 9
2.3 *Lost Survivors*: Island overview 10
2.4 Markov decision process - agent-environment interaction. Image from Sutton [35] 15
2.5 Plot of the surrogate function L^{CLIP} with the probability ratio in relation to the advantage function. Image from Schulman et al. [1] 18
2.6 Diagram of the curiosity module in Unity 20
2.7 Unity Editor window layout (color theme light) with additional markers (red) . . 23
2.8 Project dependencies between *Lost Survivors* and the ML-Agents Package . . . 24
2.9 Inspector view of the agent *GameObject* and its components in the Unity Editor 25

3.1 UML diagram of the implemented agent in *Lost Survivors* 31

4.1 Trainings with and without curiosity in comparison 42
4.2 All training runs with curriculum lessons 43
4.3 Reward progress on curriculum training runs 43
4.4 Curriculum learning with different amounts of stacked observations 44
4.5 Increasing episode length over time 45
4.6 Curriculum lessons training progress 46
4.7 Agent at training start 46
4.8 Trained agent at the beginning of its episode 47
4.9 Player feedback to expansions 48
4.10 Player feedback to expansions 48
4.11 Relative amount of actions used by the training agents 49
4.12 Anomaly on training starts and restarts 50
4.13 Unknown pauses in training progress 51
4.14 Cumulative reward by training steps 51

List of Tables

3.1 Agent action combinations 32
3.2 Agent actions 33
3.3 Automatic executed agent actions 33
3.4 Agent observations 35

4.1 Curiosity hyperparameters 42

List of Algorithms

1 RL training loop 16
2 Proximal Policy Optimization (PPO) 19
3 Clear action values without specifications 34
4 Curriculum reward function 36

Listings

3.1 First action check in the `DemoActionProvider` 37
3.2 Training configuration file 38

Glossary

.onnx A generalized file format to represent deep learning models. https://onnx.ai/, Accessed: 06.06.2021. 45

AI Artificial Intelligence. I, 2, 4, 13

API Application Programming Interface. 22, 23, 27, 28

AR Augmented Reality. 21

Atari A well known computer game company, producing video games and consoles since 1972. A very popular game from Atari is *Pong*. More info can be found on https://www.atari.com/, Accessed: 06.06.2021. 2, 22

bad smells Bad smells in code are surface indicators for deeper problems in the software architecture, which makes the code hard to maintain. A deeper explanation how to detect bad smells can be found in the work of Fontana et al.[2]. 53

BC Behavioral Cloning. 3, 4, 7, 13, 15, 21, 26, 36, 42, 45, 46, 55

CLI Command Line Interface. 24

CPI Costs Per Install. 11

GAIL Generative Adversarial Imitation Learning. 3, 4, 7, 13, 21, 26, 36, 42, 45, 46, 55

HAW Hamburg Hamburg University of Applied Sciences (ger. Hochschule für Angewandte Wissenschaften Hamburg). 27, 38

HTTP Hypertext Transfer Protocol. 22, 34

HUD Heads-Up Display. 27

hyperparameter Hyperparameters define the setup of machine learnings training algorithm i.e. the amount of layers in the neural network. 4, 16, 41, 42, 45

KL Kullback-Leiber divergence. 18

LTV Livetime Value. 11

ML Machine Learning. 1–4, 7, 13, 15, 23, 26, 53

model A model is the output of running a machine learning algorithm on training data. It contains a weighted ruleset to make predictions based on defined rules. 15, 22

namespace In the programming language C# namespaces help to organise classes in bigger projects. Namespaces are also used to make classes and attributes only accesible in a defined scope. 28

one-hot Binary representation of a value by a single high bit. Can be used to represent game items. For example, 001 = coconut, 010 = water, 100 = fish. 34

PPO Proximal Policy Optimization [1]. I, II, 2–4, 7, 13, 16–19, 22, 26, 42, 45, 46, 55

PyTorch An open source machine learning library written in python. It is based on Torch and provides tensor computation and deep neural networks. More info can be found on `https://github.com/pytorch`, Accessed: 06.06.2021. 22

QA Quality Assistance. 3, 4, 7, 55

retention In the context of mobile games retention defines how long players stay enganged with the game. This is often measured in consecutive login days. 11

RL Reinforcement Learning. I, II, VII, 2–5, 7, 13, 15, 16, 19–22, 26, 34, 56

RND Random Network Distillation. 26

SAC Soft Actor-Critic [3]. 22, 26, 42

SDK Software Development Kit. 23

technical debt Code written in a bad style, with the potential to slow down production in a later stage. Technical debt is often specific to the programming language. A common example is nesting of multiple if clauses instead of early returns. 12

TRPO Trust Region Policy Optimization. 18

UI User Interface. 21

UML Unified Modeling Language. 23, 28, 31

VR Virtual Reality. 21

xp experience points. 2, 7, 8, 19, 26, 32, 33, 35, 37, 43–47, 49, 53

YAML Yet Another Multicolumn Layout. `https://github.com/yamlcss/yaml`, Accessed: 06.06.2021. 24, 26

Chapter 1

Introduction

1.1 Motivation

In the development process of video games the synergy between many complex systems is crucial to form the final product. These are not only composed of technical systems, but also the game design and quality assurance. The goal is to create the best possible gaming experience, which is reached when players are kept in the so-called flow state [4]. In this state players have a pleasing gaming experience in such an immersive way that the ideal scenario may distract them from the reality to which they live in their day to day [5]. This state is often reached by fluctuation between higher and lower difficulties which are controlled by the game design. Game design is a broad discipline having effect on different aspects of a game.[1] For a detailed game analysis the method of L. Konzack [7] can be applied, which describes analyzes a game by the following seven aspects: hardware, program code, functionality, gameplay, meaning, referentiality and socio-culture.

During the game development phase the game designers are responsible for the aspect of gameplay while programmers are providing the program's code and thus the functionality. Quality of the code and functionality is additionally reviewed by the quality assurance [8]. There is no magic formula to craft the perfect game, but from past releases like *Cyberpunk 2077* [9] [10] and *Fallout 76* [11] it can be seen that bugs have a huge impact on the success of a game. In this work, Machine Learning (ML) will be used to improve the development cycle of a game. The hypothesis is that a ML agent can detect bugs and other flaws in the game which a developer or game designer wouldn't notice. In the best case scenario the ML algorithm will be trained to play the game on an expert level. Game designers or other developers could then set benchmarks for the algorithm, like a specific time it should take to reach a defined game state. After or during training, the agent should then report back to the developer if the state has been reached. The training process should also keep track of exceptions and other unwanted game states in which the player might get stuck. Analyzing the results would then show in which time frames the agent was able to reach its goal state and more importantly, which strategy the agent used to reach it. Game design can then decide if the observed agent behavior fits the planned game or if it has to be adjusted. As an example, in a simple race game the benchmark could be a certain amount of laps to finish. After training the agent, it may discover that it is faster to drive in reverse gear. This might be a legitimate tactic, but not something players would enjoy.

Another way to utilize ML would be to test different sets of game balancing data against each other. Game design could then define a common goal for the algorithms to train towards.

[1] A quick look on table of contents in *The Art of Game Design: A book of lenses* by Jesse Schell [6] can give an idea how complex game design is.

Therefore, after the training process game design can choose the balancing data which produced the best results. ML has the potential to improve the game development process in multiple aspects. The training process itself can be used to detect bugs in the form of exceptions, error messages or other defined conditions. It also produces a lot of random player data which can be analyzed to find exceptional game states, which can help with game design decisions. Lastly, the fully trained agent can be used to validate game concepts and to prove that there are no unforeseen exploits in the game.

1.2 Related Work

Maintaining a high quality standard is crucial to long living projects to keep them flexible to changes and new game features. To achieve high quality standards, a variety of tools and procedures are used. For example, unit tests, code reviews, automated frontend-, load-, build-tests or smoke testing by quality assurance, have become standards in today's software and game industry. In the work of Nayrolles et al. [12] Machine Learning has been successfully introduced to improve the game development process. Their *CLEVER* algorithm is able to detect risky commits with a precision of 79% (of all commits which would possibly introduce bugs to the code base) and to suggests possible fixes. After the code has been introduced into the repository, there are often no further automated checks before the game is tested manually. This is where the algorithm of this work comes into play. The machine learning agent can be used to analyze the game to create efficiencies before the manual testing takes place which is traditionally a high expenditure of time.

Research in Artificial Intelligence (AI) has already shown that ML algorithms are capable of playing nearly any computer game on a super human level. For example, OpenAI Five [13] was able to beat a professional e-sports-team in the highly complex video game *Dota2* [14]. In their work they used Proximal Policy Optimization [1] (PPO), which is a deep Reinforcement Learning (RL) algorithm. In order to achieve their extraordinary results OpenAI Five's ML training runs simulated hundreds of years of gameplay data. Another example is the AlphaStar [15] algorithm which also used RL with multi-agent learning on the game *StarCraft II* [16]. Their agent achieved the highest rank and was able to beat 99.8% of all active players. Remarkably is that AlphaStar algorithms perception is bound to the camera spectrum. The algorithm itself has to control the camera movement to observe and interact with the game. Therefore the agent is initially trained with imitation learning to learn from 971,000 replays of expert players. To improve the performance this was further combined with RL in respect to reward the agent on winning the played games.

Most AI agents are run on finished games. Even in these scenarios some agents were able to find bugs and other exploits to get the best possible scores depending on their reward functions. In the work of P. Chrabaszcz et al. [17] an unknown bug was found in a long existing Atari game. The original goal of their work was to show that Evolution Strategies are competitive to classic RL algorithms. As a benchmark they used a set of eight Atari games, which are part of the OpenAI Gym [18]. A part of their results showed that their algorithm was able to perform a specific sequence of movements in the game *Q*Bert*, which led to a state in which the game grants the player a huge amount of points. Instead of previous highscores around 25,000 points, in 22 of 30 runs, their algorithm achieved close to a million points due to exploiting the in-game bug.

The work from J. Lehman et al. [19] shows that a seemingly simple task like "jump as high as you can" can lead to surprising results when running the program. The AI agent's task was to build its own creatures out of blocks connected by joints. For the training process the AI agent

was rewarded for distance between the initial lowest body part and the ground. In this case the agent decided to build a creature with an identifiably tower-like limb. Instead of jumping the agent made it actively fall, which also resulted in an airborne state for a short period of time (see Figure 1.1).

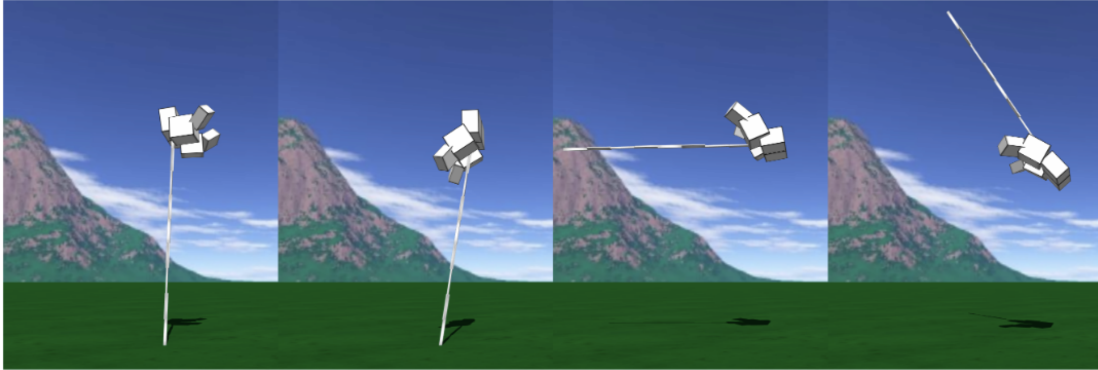


Figure 1.1: Exploiting potential energy to "jump". Image from J. Lehman et al. [19]
The blocks are connected with joints which can be moved. To achieve a high distance between the tower-like limb and the ground, the creature leaps itself head-forward to the ground. The creature is a result of the work from Krcah [20] which was represented in the work of J. Lehman et al. [19].

The problem lies in the abstraction of human understanding of the original goal translated into a reward function. Here, the reward was defined to just get the highest distance to the ground. These findings are often seen as mistakes and are fixed by redefining the reward functions. The AI safety researcher Krakovna at al. [21] shows that the ingenuity of RL algorithms' to solve problems in an unexpected way is a common issue. More than 60 examples can be found in the associated list [22] to the work of Krakovna. In reverse, this also means that a good RL algorithm is able to effectively find specification issues in the game environment given that the reward function reflects the players' goal by design. Defining the right reward function is crucial, but in the context of games the goal is already an abstract objective like a highscore or earning experience points (xp). This can be understood by the ML algorithm in the same way, and no transformation into an abstract reward function is needed. In this work, the goal is not to create an agent which can play the game perfectly, but to see which unexpected ways or bugs the agent utilizes to optimize its defined reward.

Another key player in today's research field is the Unity engine. In the work from Juliani et al. [23] Unity shows great results as highly compatible simulator for machine learning. Unity provides a variety of open source machine learning algorithms which can be easily integrated into the Unity engine (see section 2.4). This also includes the PPO algorithm which was used by OpenAI Five. In their own *Pyramids*[24] environment Unity shows that training results can further be improved with additional ML algorithms Behavioral Cloning (BC), Generative Adversarial Imitation Learning (GAIL) and their curiosity module (a detailed introduction on these can be found in section 2.3). Based on its own tools, Unity also offers the machine learning service GameTune[25]. GameTune uses machine learning to optimize game design parameters on a player basis. The goal is to improve the players' experience to increase retention and monetization by providing individual game design parameters for each player. As an example, an experienced player could only get a brief introduction to the controls, while a less experienced player gets a full tutorial on all game mechanics. To achieve its results, GameTune requires a minimum of 30,000 daily active users to support the training process.

For games in development or newly released games it is nearly impossible to acquire this amount of players. This is why this work is focusing the training process on data which can be acquired and evaluated already during development. Even though it will not be possible to predict or create human player behavior, it will be possible to unfold game design flaws and bugs which will retrospectively enhance the player experience as well.

1.3 Scope

The research goal of this thesis is to find a solution to reduce or mitigate the workload of Quality Assistance (QA) and game design by implementing a deep RL algorithm to play a game by itself to reveal and discover game design flaws and bugs. This is especially useful during development, where a lot of changes are applied to the game. Utilizing machine learning as an additional resource to assist the game design and the QA department can speed up the development process, as it would reduce the need for manual gameplay testing. It has to be proven whether the effort of implementation such algorithms produces a significant additional value.

The approach in this work is to implement a machine learning algorithm to play a complex strategic city building simulation game. This work will be created with the support of InnoGames GmbH [26] by providing the complete code base of a city-building game similar to *Township* [27]. The game is called *Lost Survivors* [28]. The objective of this work is to answer the following hypotheses:

- H1 The trained agent on *Lost Survivors* always finds an optimal way to play the game in regard to its reward function.
- H2 Machine learning processes help the game design department to optimize balancing data.
- H3 Machine learning processes help the QA department to find bugs.
- H4 Machine learning can be generalized and added to any game project.
- H5 Once the machine learning is implemented it is easily maintainable and further training can be performed.
- H6 Machine learning helps in the process of game development.

The first two hypotheses will be answered having a trained AI agent playing the game. Observing the behavior and collecting additional data about the quantity and sequence of the output actions will help to analyze the strategies the agent has developed in the training process. During the training process and the implementation of the agent upcoming bugs will answer the question if the QA can benefit from the training results. In future prospects the reward function can be adjusted by the game designer to support training for different player types. In complex games there are often multiple goals a player can achieve to address different motivational drivers of different player types [29]. It will be evaluated if possible balancing suggestions can be extracted by either giving the ML algorithm the possibility to take the balancing parameters as input or by training multiple agents with a range of balancing data. For the hypotheses H4 and H5 each implementation step of this work will be evaluated. In focus are comparability between different games and the adjustments which would be needed to get the ML algorithm working on other games. With the results of the previous evaluation this work will then come to the conclusion if ML is a useful tool to assist in game development processes.

Having AI solving problems or playing various kinds of video games has been accomplished numerous times in the previous years. One of the biggest difficulties in the field of AI is to have one training algorithm being able to solve different games without having to be trained again, however this will not be discussed in this work. Machine Learning is often seen as a black-box with many different parameters to adjust. In order to accomplish the best possible results in this work, the deep RL algorithm PPO is used as its implementation aims for ease of use in parameter tuning [30]. Because the game involves mainly single player actions, this work will focus on training a single-agent, which will be used in combination with Behavioral Cloning, Generative Adversarial Imitation Learning and curriculum learning.

The provided game *Lost Survivors* is in development and has currently the status of a minimum viable product of a feature complete game. During the current development cycles the game's balancing data is constantly changing which makes it a perfect candidate for this research question. Before starting with a highly complex system, though, this work will first cover a basic implementation of the RL algorithm on a demo project. Therefore, the *Hummingbird* example by Adam Kelly [31] will be used. The *Hummingbird* example project is implemented with the Unity engine (see section 2.4) and uses the ML-Agents plugin (see subsection 2.4.2) to integrate RL algorithms. This makes it great for *Lost Survivors*, as it is also programmed with the Unity engine. The working RL implementation will then be transferred into a basic version of the *Lost Survivors* game. In the implementation of the ML agent's interface, special features like time-based events, decorations, the chat system etc. will be excluded as they are not essential for the agent to play the game. This phase will also serve as reference in chapter 4 to evaluate the effort needed to transfer the ML algorithms to different games. To avoid an additional layer of complexity, the interface between the ML algorithm will send most of the input actions as direct requests to the game backend. In *Lost Survivors* most of the logic is implemented and tested in the game's backend system which run on external servers. The frontend client only handles visuals and readability for the player. A visual frontend client will not be needed to train the agent, but can be added later for easier interpretation of the results. During the transition from the *Hummingbird* demo project to the game, the aspect of generalization is always in consideration. As soon as the first version of the ML algorithm is able to train on the game, the learning algorithm will be tested with different hyperparameters and examined on its performance. For further evaluation different reward functions will be compared with each other to gain better understanding of how the reward function can represent player motivations. Based on the final results it will be evaluated if the game design proves as correctly implemented or if the agent found exploits to reach its goal. From that, it will be possible to create proposals for game design changes which need to be integrated.

1.4 Structure

For a better understanding of the game, the following chapter 2 will first cover the game mechanics in section 2.1. For the later analysis of how ML can improve the game development process section 2.2 will illustrate a typical game development cycle with the focus on game design (subsection 2.2.2) and Quality Assistance (subsection 2.2.3). The concepts of the Machine Learning and deep Reinforcement Learning algorithms used in this work will be explained in (section 2.3). These will be applied within the Unity engine, which will be introduced in section 2.4, followed by the implementation in chapter 3. The results of the algorithm will be presented in chapter 4 which will be followed by the conclusion and future work in chapter 5.

Chapter 2

Foundations

This chapter introduces all the concepts which impact the results of this work. It starts by giving an overview of the game *Lost Survivors* which is used as an example for this work's research question. The next section 2.2 gives an overview of the game development process to create a clear picture where the ML algorithms fits in. The game development process gives a general idea of the industry standards in mobile game development, followed by a more detailed description of Quality Assistance and game design processes. Further the section Machine Learning explains deep Reinforcement Learning, specifically with the focus on Proximal Policy Optimization [1]. Expanding concepts which can improve the training results of RL are explained right after. This includes curriculum learning, curiosity, Behavioral Cloning and Generative Adversarial Imitation Learning. The final section of this chapter introduces the Unity game engine with its ML-Agents plugin. This section covers how the ML algorithms from the previous section are integrated into the *Lost Survivors* game, which is also implemented with Unity.

2.1 The Game

The game provided by InnoGames is called *Lost Survivors*. At the time this work was created the game has been in development, but has the feature set of a minimum viable product. The game falls into the category of city building simulation games in which resource management is one of the main mechanics. The theme of the game is a group of survivors waking up on an abandoned tropical island after a plane crash. From here on the main character Amelia introduces the player to the main tasks as camp leader. The player sees the game environment from a top-down perspective and controls it by touch and swipe gestures (the target platform are mobile devices). The goal is clear and simple: survive as many days as possible. Over the process of the game players have to build and upgrade their home island, which they do by ordering the survivors to gather resources (see Figure 2.1), craft product, build buildings, expand the territory, send survivors on expeditions and many more activities.

Most of these activities are additionally rewarded with experience points (xp). Each experience point brings the player closer to reach the next level, which is in this game's setting equivalent to the next day. There are no activities which can make you loose experience. The game is designed to be played in multiple short sessions between 10 and 30 minutes. Therefore, most activities, once started, can take minutes if not hours before the task is completed. The time for the tasks increases the further the player progresses in the game. Building a level-one building might only take five minutes but a building on level ten can already take two hours to be completed. To make the player enjoy coming back to the game, every session should feel rewarding.

Figure 2.1: Gathering panel from a player perspective
In the current state four gatherers are idle, while the two gatherers on the right have finished their production of resin. The productions will add the resin to the storage and grand a small xp reward.

By game design there is always something meaningful the player can do to gain further progress. As shown in Figure 2.2, an example session could look like this: Collect the resource productions which have been finished from the last session → Use some collected resources to send out a parrot at the task board → Send a survivor on an expedition with the resources you collected → Finish a building construction → Send gatherers to collect more resources → As you now have to wait again until your survivor comes back from the expedition, you can end the session and collect the rewards in the next one.

The game is targeted to casual players who according to Bartle [29] fit into the demographic of the player types achievers and socializers. The player groups are usually not clearly distinct from each other and a game can include multiple motivational drivers to widen the range of players. By breaking down the core loop of *Lost Survivors* it can by analyzed which player types are supported the most. The main motivation of the achiever is to finish everything the game has to offer. In the game *Lost Survivors* this translates to completing every quest, task, building, upgrades etc. Achievers will find their challenges from early on in the game as most of the collection and completion tasks get introduced to the game during the first session. In the second and third session the player gets familiarized to the expeditions. These enable the player to explore new unseen areas and spark the sense of discovery. Shortly after the player unlocks the main social feature: the guilds. These are especially designed to enable socializers to communicate with each other, which is their main motivational driver. In later stages the guilds enable competition between players and guilds. Important to mention is that the social motivation is an intrinsic player motivation[32] as it is not directly rewarded by the game itself (there are very few exceptions, which will be left aside in this work for the sake of simplicity).

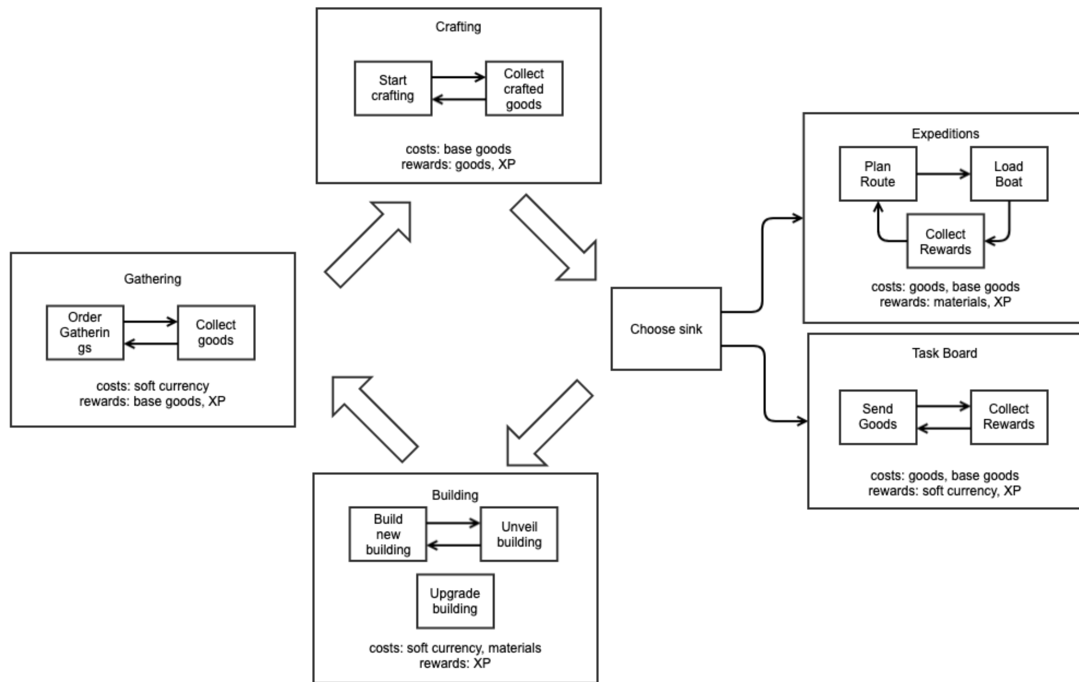


Figure 2.2: Core loop of the game *Lost Survivors*

The loop pictures all main activities the player cycles though while playing the game.

Another example for intrinsic motivation can be seen in Figure 2.3, where the player has built a lot of decorative objects. Decoration objects can be bought for soft or hard currency[1] and only serves the visual purpose. In contrast, achievement driven motivation is an extrinsic motivation as most of the actions like building buildings, collecting new survivors etc. are directly rewarded with experience points. From Figure 2.2 it is apparent that most of the game actions in *Lost Survivors* are supporting the achievers' player type and has the common extrinsic reward of experience points. As this reward is deterministic and measurable, this will later be the basis for the reward function of the machine learning algorithm (see chapter 3).

[1] While soft currency can be earned inside the game (see Figure 2.2), hard currency has to be purchased with real world money.

Figure 2.3: *Lost Survivors*: Island overview

The screenshot shows the island of a high level player which values decorative objects. In the image center paths, fences, bushes and sculptures can be seen. All of these objects are optional buildings which do not have any effect on the gameplay itself. The production buildings in the top right indicate finished productions with the hovering icons. In the bottom left idling gatherers are waiting to be sent out for collecting goods.

2.2 Game Development Processes

It takes a lot of time and effort to create great games. While mobile games can be developed in a few months, most of the publicly known games took three to five years to release and if successful the development continues in the form of support and extension updates. The development process can be summarized in the following phases: pre-production, production and post-production. Each of these phases can be split up in several process steps, in turn, the focus of this work lies on the production phase. In this phase the setting, concept and often also a prototype have been approved, and the development team is fully staffed. Typical positions are Game designer, Quality Assistance, Front- and Backend Developer, UI/UX designer, 2D/3D Artists and Product Manager.

Game development is an agile development process typically separated by milestones and sprints. Each sprint is two to four weeks long and filled with tasks to work towards the milestone's goal. Sprints allow better planning due to smaller tasks which make the goal more tangible. They also help to foresee if the development is still on track. The first milestone is to implement the core loop of the game. As soon the core loop is done, the internal play testing is conducted by game design and quality assistance. The next step is to evaluate the current state of the game in which game design may change balancing data. Game design and product management will then plan which mechanics need readjustment and which features will be implemented in the next development cycle. This loop continues, and the code base grows quickly as most of the software architecture still has to be created.

The goal of the production phase is to release the first playable version on the market. The development focus then switches to the enhancement of player retention. In *Lost Survivors* the goal was defined by a benchmark of players consecutively playing to the third day. The first version had to include a tutorial to guide the player into the game with three days of additionally playable content. If the app successfully launches the game production continues with the post-production. In the mobile game industry success is measured by data like, Costs Per Install (CPI), retention rate and user Livetime Value (LTV). In the post-production the existing game features will be maintained, and additional content will be produced in the form of in-game events.

2.2.1 Embodiment of Machine Learning in the development process

Game design and Quality Assistance are the main parties benefiting from the machine learning process presented in this work. Because there are no players yet, the game evaluation from the above-mentioned process heavily relies on manual play testing. This has the disadvantage of being time-consuming and dependent on subjective perception. At the end, experience of previous projects helps Game Designers and Quality Assistance to make the right decisions.

In this work, machine learning is used as an additional process step to create artificial player data. By evaluating the agent data and the strategies of the finished trained algorithm, design decisions and balancing changes can be supported. In the best case also unseen balancing problems will become apparent by the agents' actions. During the training process itself, the algorithm can log out bugs and game states in which the player is stuck. During the training, the agent will encounter a huge variety of game states (if not all), which otherwise would have been covered by manual smoke testing. This gives Quality Assistance the possibility to focus on other tasks which are not covered by the training process, for example graphical bugs.

2.2.2 Game Design

A game designer's job is to come up with the game idea and concept. The game designer defines the rules, theme, concepts, dialogs and many more game related content, from the attack damage of a weapon item to the mythology of a mighty hero. There are many disciplines in game design to master which is why this role is often split up to multiple people. This work will especially focus on the discipline of game balancing.

Balancing Balancing is the part of game design dealing with all static numbers defining the algorithmic functions like increase of player experience per defeated monster or loss of health on falling. The complexity increases with the amount of game mechanics. Each newly introduced feature can increase the possible outcomes exponentially, depending on how many core game mechanics it affects. For example, when introducing an extremely powerful weapon to a first-person shooter game. This can cause an imbalance that could reduce the variety in considerable play stiles. This decreases the game's fun-factor because every player would use the same weapon and there would be less room for surprise [4]. The goal is to balance the game's content in a way that the player keeps a high motivation and has fun playing the game. From another perspective the balancing data also influences how the player is approaching the game. With balancing it is possible to enable, disable or add certain requirements to actions. Changing the balancing data will influence the player's actions and thus the outcome of the game score. Let's take a look at the balancing data to build a building in *Lost Survivors*. To build a building you have to fulfill a certain amount of requirements. These requirements include player level, construction space, resources, other buildings and other survivors. Depending on the unfulfilled requirements, the player will

judge the action to build a building as easy or difficult. This not only depends on the amount of different requirements, but also on the time and effort it takes the player to complete these. With a higher difficulty the player will expect also better rewards (in comparison to what the player has encountered when finishing building previously). If all requirements are met the player is allowed to place the building on the island. This starts a construction process which, on the highest level building, can take up to 57 hours. After the construction is finished, its reward (in the form of experience points) can be collected and the building gets unlocked. The direct reward of experience points is not decisive if the player will feel motivated to build a certain building. Unlocked buildings can give the player the ability to produce more goods, finish a quest or might be needed as a requirement to build or upgrade other buildings. A player might also feel rewarded if the constructed building has a beautiful outer appearance, especially if the building can be shown off to other players to gain prestige. Once more it can be seen that different player types (mentioned in section 2.1) perceive rewards differently, however the direct reward can already be a good indicator.

In the algorithm of this work, the direct reward will also be part of the algorithm's reward function. Here the reward is rather used to measure game progress than player motivation. From the agent's perspective it is not possible to measure fun or motivation on game variables which is why the goal of the implemented algorithm is to actively assist game design. The game designer has to chose which group of players and play-styles should are preferable. Based on that the reward function needs to be adjusted.

2.2.3 Quality Assistance (QA)

Quality Assistance takes a slightly different approach to ensure the software quality than the usual Quality Assurance strategy. The following paragraph is the guiding principle which is used at InnoGames.

> The whole team is responsible for the quality of the product. Every team member who contributes to the product is responsible for the entire feature development life cycle. QA is responsible for assisting the whole team in providing high quality.
>
> (InnoGames GmbH [26])

The main difference to the usual Quality Assurance approach is that not only QA is responsible for the quality of the end product but the developers themselves. This means that during feature development programmers themself have to make sure that the feature meets the quality standards. Meanwhile, QA assist the developers with automated tests, smoke testing (cross-feature), raise awareness on feature edge cases, providing an overview of the product quality and other tools to make the development cycle as efficient as possible.

Quality In game development, quality can be defined as the following. The current state of the game (when working with a versioning tool this is equal to the master branch) should always be stable and as bug free as possible. All new implemented features should not break any existing content. New content behaves exactly the way it was designed. The technical debt of the software code should not be increased.

Bugs in the code base can immediately destroy the immersion of a game. For example when the player glitches through a wall and therefore skips big parts of the level by accident. But bugs must not necessarily be noticed by the user to alternate the gaming experience. As an example, an enemy which never spawns in the game might not be noticed as a bug, but the player might

feel underwhelmed as the game is now too easy. Finding bugs includes a lot of manual game testing which is a time-consuming process. Even though QA tries to track down as many bugs as possible, the majority of bugs are much more likely to be found by players who haven't been part of the development process. This is often due to the fact that developers are biased to test the game in the way it was designed. Players on the other hand side come with different previous gaming experiences and expectations. Using external testers during the development process can help but is not enough to reflect the amount of thousands of players. This is where the training process of this works algorithm can be tied into the Quality Assistance. The Machine Learning algorithm has no previous gaming experience and is able to paralyze its training runs which are comparable to smoke testing. Therefore, it is expected that a wide range of play tests is covered during the training process.

2.3 Machine Learning

Machine Learning is part of the scientific field of Artificial Intelligence. Its goal is to find solutions for problems which are too complex or time-consuming to be solved by heuristic programs. Machine Learning can be traced back to 1961 when Minsky published his work *Steps toward artificial intelligence* [33] in which he developed universal problem solvers for his fundamental credit assignment problem. General principles of ML include that the algorithm is capable to improve through experience, while adapting the fundamental theoretical laws of the learning environment. According to Tom Mitchell ML can be defined as followed:

> Each machine learning problem can be precisely defined as the problem of improving some measure of performance P when executing some task T, through some type of training experience E.
>
> (Tom Mitchell [34])

In example, for this work the Task T is to learn which input action maps to the best possible experience points outcome. The Performance metric P is the reward the agent collects during its training run. The Unity environment itself reflects the experience E as it is feeding the algorithm with its observation. Until today, various concepts of ML have been applied to many applications and have become a standard in today's search engines, image recognition, robotics and many more.

The present work will focus on the deep Reinforcement Learning algorithm PPO which will be explained in section 2.3.4. To get a better understanding on the underlying concepts this chapter is further separated in respect of the used RL algorithms. It starts with an explanation on Artificial Neural Networks (ANNs) (subsection 2.3.1). The next subsection 2.3.2 covers the Backpropagation optimization process for ANNs via stochastic gradient descent. The use case of optimizing ANNs is first explained in Supervised Learning (subsection 2.3.3) before getting into more details on Reinforcement Learning in subsection 2.3.4. Afterwards, further concepts of curriculum learning, curiosity BC and GAIL will be discussed, which are used in combination with PPO.

2.3.1 Artificial Neural Networks

In reinforcement learning the acting algorithm called agent is represented by a system of artificial neurons, in other words a neural network. Similar to real neurons, each neuron can have multiple weighted inputs which fire signals depending on its activation function. By adjusting the weights though a learning process, the system is optimized to output the desired actions. A neural

network contains multiple layers of neurons, which includes an input layer which represents the game state, an output layer which represents the possible actions, and optionally multiple hidden layers in between. When the reinforcement learning network contains hidden layers it is referred to as deep reinforcement learning.

The policy network implements the agent and takes the game state as input and predicts an action with the best possible result as output (the actual representation of actions, game states and rewards will be further discussed in subsection 2.3.4 and subsection 2.4.2). Hereby the input i is processed though a defined amount of hidden layers[2] to produce a probability (by using a stochastic policy) for each output action. Each neuron j has a weighted influence on the predicted output[3].

2.3.2 Backpropagation

To optimize a policy network backpropagation is used. The goal is to adjust the neurons weight w_{ij} over the training process, in a way that they predict the correct outcome. Backpropagation is based on the mean square error and uses gradient descent to minimize it. In practice, the game state is fed to the policy network, which will produce a vector of logarithmic probabilities for each agent action. The agent would then choose the action o_i by the highest probability which can then be compared to the optimal outcome t_i as seen in Equation 2.1:

$$E = \frac{1}{2}\sum_{i=1}^{n}(t_i - o_i)^2 \tag{2.1}$$

The goal is to minimize the error E which is calculated as the sum of network states n and the deviation between the wanted action t_i and the output action o_i. $\frac{1}{2}$ is used to simplify the mathematical derivation.

With the known weights and the fact that the activation function φ is non-linear and differentiable, the output o_j of each neuron j can be defined as the following Equation 2.2. Where net_j is the neurons j input which is equal to the sum of all neuron outputs from the previous layers k with its respective weights w_{jk}.

$$o_j = \varphi(net_j) = \varphi(\sum_{k=1}^{n} w_{jk}o_k) \tag{2.2}$$

Retrospectively this allows to calculate the inputs and output for each neuron and enables to adjust the weights. The gradient of the mean squared error is calculated by the partial derivative of Equation 2.1 in respect to w_{jk}. By factorizing it with a chosen learn rate $-\eta$ the following Equation 2.3 can be used to calculate the weight change Δw_{ij}.

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial net_j} \tag{2.3}$$

To optimize the policy network and minimize the error E the result of Δw_{ij} is added to each neuron weight w_{ij}. By repeating this process over and over again the network will converge to the next local minimum. To achieve best results it is important to choose the right learning rate. A too small learning rate can lead to a very slow training process or end up in a local minimum. A too big learning rate can overshoot the local minimum and end up with a greater error than before.

[2] four hidden layers were used in this work's training

[3] at the start of a learning process neuron weights usually get initialized with a randomized value between 0 and 1

2.3.3 Supervised Learning

Reinforcement Learning algorithms are similar to supervised learning algorithms. For a better overview on the reinforcement learning algorithms used in this work, this section will give a brief introduction to supervised learning.

The objective of supervised learning is to train a model to accurately yield the correct output based on a set of training data. In this case the supervised learning algorithm will also be represented by a policy network. The policy network is then fed with previously generated training data. The returned output action would then be compared with the labeled training data which provides the correct action. The correct action would get a gradient of 1 while the others will get 0. With the correct action and the output action, the mean squared error can be calculated with Equation 2.1, and the policy gets updated by backpropagation. A problem with supervised learning is that a large set of training data has to be created before the agent can start its training. As the learning process by definition tries to imitate the actions from the training data, it cannot become a better player as the existing training data. This process can also be referred as Behavioral Cloning which will be further described in subsection 2.3.7.

2.3.4 Reinforcement Learning

One approach of Machine Learning algorithms is trying to imitate the human learning process by learning from it's past experiences. In reinforcement learning, the agent acts within its environment over time while trying to achieve a goal by performing actions which influence the state of the environment. The interaction with the environment happens in a defined amount of steps in which the agent observes, acts and collects rewards for its actions. A visualized representation of the agent interaction is shown in the agent-environment interface from Sutton and Barto [35] in Figure 2.4. His Markov decision process displays the cycle of agent actions alternating the game state to optimize the perceived reward.

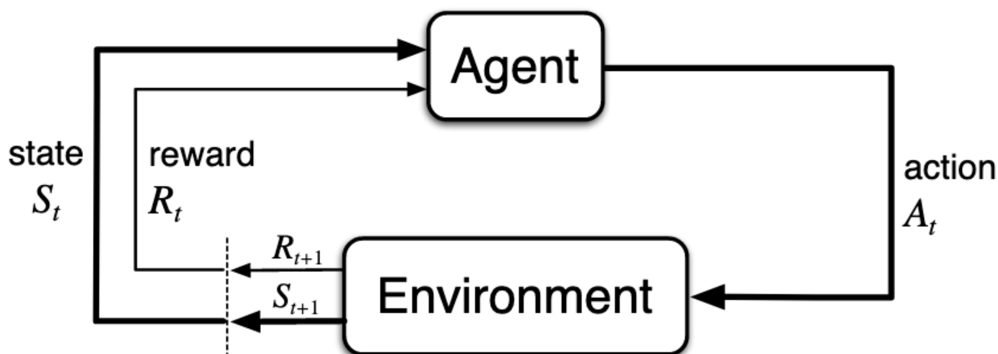


Figure 2.4: Markov decision process - agent-environment interaction. Image from Sutton [35]

At any given time t the environment is in a state $S_t \in S$. With the current state S_t the agent can take an action $A_t \in A$ to get to the next state S_{t+1}. With the transition from state S_t to state S_{t+1} an immediate reward R_t is given. The immediate reward is based on the chosen action: $R_a(s_t, s_{t+1})$. (Sutton and Barto [35])

Each action to transitions into the next state S_{t+1} has a stochastic probability to be chosen by the agent, which is defined in the agents policy π. Formally, the policy can be defined as Equation 2.4, which returns the probability distribution of actions given a state.

$$\pi(a \mid s) \doteq P(A_t = a \mid S_t = s), \quad (2.4)$$
$$\pi : \mathcal{A} \times \mathcal{S} \rightarrow [0, 1].$$

The big difference from the above described procedure in supervised learning (see subsection 2.3.3) is that in Reinforcement Learning no pre-labeled data is given. Instead, the agent generates its own data during the training process by processing the policy in the environment (as described above in Figure 2.4). To optimize the policy network, the reward given by the environment can be used. The procedure works the following:

Algorithm 1 RL training loop

1: **loop**
2: **for** *batchSize* **do**
3: Run and record the policy network
4: Label the output actions according to the gained rewards (rewards can be negative or positive)
5: **end for**
6:
7: Update the policy network by backpropagation
8: **end loop**

Where *batchSize* is a fixed amount of steps defined by the hyperparameters.

As the algorithm is optimizing for a reward function, which is not limited to a static data set, the performance of the RL algorithm is also not limited and able to become better than human players as well. A downside of the so far described RL algorithms is that the states the agent encounters during the training are highly dependent on the initialization of the policy network and hyperparameters like the learning rate or steps. Another factor is that the agent has to experience positive and negative rewards many times to optimize its policy network accordingly. There are different variances of RL algorithms which target these issues to improve the learning process for certain objectives. One of todays state-of-the-art algorithms is Proximal Policy Optimization [1] (PPO), which is an on-policy RL algorithm.

There are two ways to update the policy network from the collected experiences, which are on-policy and off-policy RL. In on-policy algorithms the agent collects its experience data by interacting with the environment. The experiences consists of the previously mentioned states, actions, following states and rewards. The experiences in on-policy RL are then used to update the agents behavior policy. After that the previously collected experiences are then deleted and the updated policy is used to collect the next batch of experiences. In contrast, off-policy algorithms have a separate policy to chose the action. The experiences collected by the behavior are stored in a data buffer. With the data buffer, off-policy algorithms are able to learn from experiences collected by previous policy's. Because PPO is a on-policy RL the following text will focus on that. A more detailed explaination on on- and off-policy RL can be found in the book by Sutton [35].

Proximal Policy Optimization

PPO is an on-policy learning algorithm and is optimizing its policy network directly from a batch of collected experiences. During training the algorithm is alternating between updating the policy network and collecting a new batch of experiences. The objective function for natural policy gradient methods, can be defined as the following policy gradient loss in Equation 2.5.

$$L^{PG}(\theta) = \hat{E}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.5}$$

In this function the logarithmic output of the policy network $\log \pi_\theta(a_t|s_t)$ (which was covered in subsection 2.3.4) is multiplied with the advantage function $\hat{A}_t$ (Equation 2.6). The advantage function itself estimates the relative value of the selected action. The advantage function covers two terms: The discounted sum of rewards (Equation 2.7) subtracted by the value function (Equation 2.8).

$$\hat{A}_t = G_t - V(s_t) \tag{2.6}$$

Return The Return G_t is the sum of all immediate rewards the agent collects during its training episode: $R_{t+1}, \gamma R_{t+2}, \ldots, \gamma R_T$. Any reward R_{t+1} at a given time step $t < T$, gets multiplied by a discount factor $0 \leq \gamma \leq 1$ to favor direct rewards rather than future rewards. Thus picking a gamma closer to zero leads to myopic behavior while a gamma close to one can encourage including distanced rewards as well[4].

$$G_t = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \tag{2.7}$$

Note that the advantage function is evaluated after the policy network, and thus the return G_t can be exactly calculated.

Value function The value function $V(s_t)$ is an estimate for the discounted sum of rewards based on the current environment state s. The value function is represented by another neural network and is also updated by the experiences the agent collects in the environment.

$$V(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \tag{2.8}$$

Where $\mathbb{E}_\pi$ is the expected value relative to the policy π given the calculated return G_t and the current state S_t.

Combining the Return and the Value function, by subtracting the estimated return from the actual return, results in the Advantage function (Equation 2.6). If the actual return is bigger than the estimated return value the advantage function will yield a positive value, which means that the selected action was better than expected. When multiplying the policy network result $\log \pi_\theta(a_t|s_t)$ with the advantage function $\hat{A}_t$, the objective function in Equation 2.5 will yield a positive gradient. When the selected action performs worse, the result of the advantage function will become negative and thus the gradient will be negative too. Now the policy network just has to be optimized by the resulting gradient to increase the rewards and leading to a better agent behavior. Note that as the value function is represented by a neural network, it is expected that the estimated value is not always correct and outputs noisy values. This can lead to an

[4] gamma is usually define between 0.9 and 0.99

irrecoverable state of the policy network, especially when starting the training with initially randomized weights on the neural networks. Therefore, Schulman et al. [36] makes use of the Kullback-Leiber divergence (KL) in his Trust Region Policy Optimization (TRPO) algorithm. This way, it is made sure of that the optimized policy network does not diverge too much from the previous optimization step. Schulman et al. optimizes the computational heavy KL further in Proximal Policy Optimization [1] where the KL constraint is included into the optimization objective:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{2.9}$$

Let $r_t(\theta)\hat{A}_t$ denote the normal policy gradients objective in the form of $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$. Where θ_{old} describes the policy parameters before the last update. By dividing the updated policy with the old policy, this results in the probability $r_t(\theta)$ which indicates how much more likely a selected action is in the new policy network. By multiplying it with the advantage function $\hat{A}$ we then know if the policy update will have a positive or negative effect.

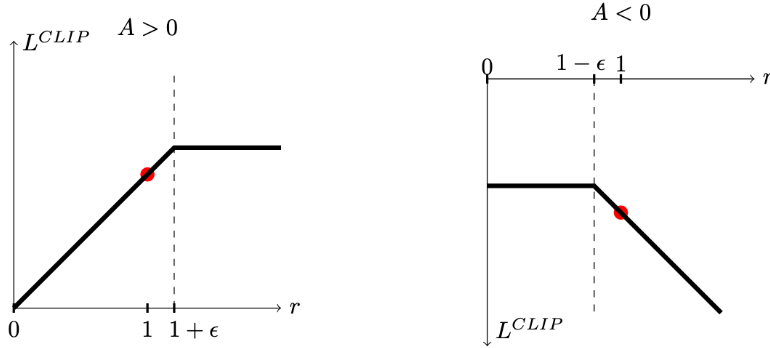


Figure 2.5: Plot of the surrogate function L^{CLIP} with the probability ratio in relation to the advantage function. Image from Schulman et al. [1]

The plot shows a single time step for the optimization objective function L^{CLIP}. On the left side the behavior of the probability ratio for positive advantages is shown, while on the right side the trend for negative advantages can be seen. The red circle on each graph indicates the starting point $r = 1$ for the optimization. The dotted lines indicate where the clipping of Equation 2.9 takes affect.

As $r_t(\theta_{old}) = 1$, a value of $r > 1$ indicates that the selected action has become more likely while $r < 1$ makes actions less likely. In Figure 2.5 it can be seen that the policy gradient objective is clipped between $1-\epsilon$ and $1+\epsilon$ [5]. This assures that the policy update is happening conservatively. For example, the selected action has a higher probability in the new policy, and it performs better than expected, the advantage function then yields a positive value $\hat{A} > 0$, for which the objective functions gradient is limited to $1+\epsilon$. The same goes for negative values, when the selected action has a higher probability in the new policy but results in a worse than expected outcome. Where $\hat{A} < 0$ the gradient is limited to $1-\epsilon$, so that the policy update makes the selected action less likely but is not overdoing the update. In the special case a selected action has become more likely ($r_t(\theta) > 1$) but the advantage function indicates that the action performs worse ($\hat{A} < 0$) the unclipped policy gradient objective will return a lower value which will be favored by the minimization operator.

[5] a value of $\epsilon = 0.2$ was used in this work

The pseudocode for PPO by Schulman et al. [1] can be seen in the following algorithm:

Algorithm 2 Proximal Policy Optimization (PPO)

1: **for** *iteration* $= 1, 2, \ldots$ **do**
2:     **for** *actor* $= 1, 2, \ldots, N$ **do**
3:         Run policy $\pi_{\theta_{old}}$ in environment for T time steps
4:         Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
5:     **end for**
6:     Optimize surrogate L wrt. θ, with K epochs and minibatch size $M \leq NT$
7:     $\theta_{old} \leftarrow \theta$
8: **end for**

Where in each iteration a number of N actors are collecting T timesteps of data in parallel. The policy $\pi_{\theta_{old}}$ is run on all timesteps NT. With the sampled data, several epochs K of stochastic gradient descent are performed to optimize surrogate L with respect to θ.

2.3.5 Curriculum Learning

One way to further improve the training process of PPO and other machine learning processes is curriculum learning. This has been first introduced by Elman [37] and Rohde and Plaut [38], which has been followed up by Bengio et al. [39]. While the work of Rohde and Plaut was unsuccessful to improve training speed, Bengio et al. work shows that the learning agent can converge faster to a global minimum, when the presented training task raises in difficulty over time. The idea behind this is that humans and animals are learning faster when they are confronted with tasks in a meaningful order. This gives the possibility to learn general concepts first before optimizing them on more complex training data. Therefore, the training is separated in lessons, in which each lesson has its own setup. During training the agent then progresses sequentially though the lessons by archiving the lessons goal. To enable curriculum learning in combination with RL, the environment is adjusted during the learning process. In order to do this, the environment needs to be parameterized, so it can then get adjusted with each lesson of the curriculum learning. There are multiple possibilities to set up the curriculum learning process. One example is to increase the task complexity by adjusting the environment itself. For example, if the agent learns to drive a car along a track, it could start with wide angled curves which get tighter over time. Other ways are to adjust the reward function or the observations' inputs. Note that the goal of curriculum learning is not to generalize the agent to be able to solve different tasks, but rather to guide the agent though the training process to its final assignment.

In the Unity application, the time at which the lesson proceeds to the next one can either be a fixed timestep or a defined reward which has to be achieved over multiple consecutive lessons. For the *Lost Survivors* project over 20 curriculum lessons were used which linearly increase the amount of xp the agent has to reach to gain a reward.

2.3.6 Curiosity

When observing children in an unknown area it can be observed that they will start expanding their horizon by constantly pushing themselves out of their comfort zone. With no promise of reward in return, their curiosity drives their intrinsic motivation to learn and explore the world [32]. The concept of curiosity can be translated into the RL learning process as well, by rewarding the agent to explore its environment. One approach has been implemented by Pathak et al. [40], which is called *Curiosity-driven Exploration by Self-supervised Prediction*. Its strength lies especially in spars reward environments. In their work they showed that learning by intrinsic reward only is also possible. A further study [41] showed that one of the main challenges is to design the intrinsic reward function in a way that the agent will not be trapped by external factors, which they called the noisy-TV problem. A TV, which was placed in the agents environment, falsely stimulated new states when the agent was standing in front of it.

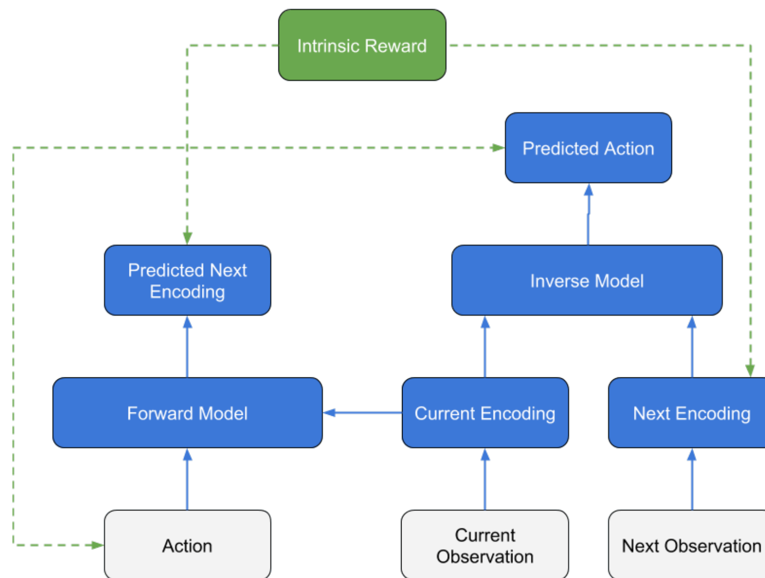


Figure 2.6: Diagram of the curiosity module in Unity

The inputs actions and observations are represented by the white boxes and fed into the two network models. The output of the models are represented by the blue boxes on the very top which will then be compared with the corresponding feature linked by the green dotted line. Figure created by Unity[42].

The curiosity module is implemented by two neural networks which are focused on the agent's actions and observations. Hereby the goal is to ignore observations which do not have any effect on the agent and cannot be modified by the agent's actions. The two networks are set up with an inverse model and a forward model which can be seen in Figure 2.6. The inverse model tries to predict the action the agent has taken from the current observation and the next observation. The current and next observation are encoded into a single feature vector and fed to the input state for the model. On the opposite side, the forward model takes the current observation and action and predicts the next encoded observation. The intrinsic reward is then created by comparing outputs of the two models. A bigger difference represents a higher factor of surprise and thus results in a higher reward.

2.3.7 Behavioral Cloning

Behavioral Cloning has proven to be an effective tool to learn a task by imitating demonstrated actions, and has been successfully applied to flying drones [43], self-driving cars [44] and steering robots [45]. Training a neural network from scratch can be difficult. Especially having a larger action space with over 10 output actions can create a huge search space to find the correct output action for any state. To give the reinforcement learning process a head start, it is possible to pretrain the neural net with recorded actions from experts. The recorded action- and observation space have to be equal to the used policy network. With simple supervised learning (which is also described in subsection 2.3.3) it is then possible to train the policy network with the provided expert data as it can be used as labeled data. The result will be an agent which learned basic behavioral patterns based on the repeating observations and actions given by the expert. As the agent only learned from existing data without necessarily exploring the complete action space, the agent can at maximum become as good as the expert data. Morales and Sammut [46] have also showed that using the pretrained policy network as a starting point for deep reinforcement learning can be beneficial to speed up the overall training process.

2.3.8 Generative Adversarial Imitation Learning

To increase sample efficiency in imitation learning algorithms like BC, Ho and Ermon [47] propose the Generative Adversarial Imitation Learning (GAIL) algorithm. Similar to BC the agent learns from expert demonstrations, but instead of mimicking the demonstrations directly GAIL uses a second neural network as discriminator. This has the benefit that the agent is not limited by the amount of steps provided in the demonstrations. The discriminator provides the agent a reward, based on the estimate of how close the performed action and observations are to the demonstrated actions. It is also possible to combine the agent's reward with extrinsic rewards from the environment. While the agent is optimized by the reward, the discriminator is trained to better separate between demonstrations and agent behavior. As the discriminator is not very well-trained at the beginning it easy for the agent to get rewards from the discriminator without precisely imitating the demonstrated actions. As the discriminator gets better over time, the agent has to become better at mimicking the demonstrated actions to obtain its rewards. To combine GAIL with RL, GAIL is used prior to RL. GAIL also has to be limited by a fixed amount of steps to not achieve results which are not completely biased by the demonstrations. Another pitfall can be the combination of positive extrinsic rewards with the GAIL learning process, as it introduces a survival bonus [48]. The survival bonus affects environments where the goal has to be reached as quickly as possible. In those environments, it can happen that the agent will avoid its goal, as it collects more rewards by executing actions close to the demonstrations without ending the episode.

2.4 Unity Engine

The Unity engine [49] is a game engine allowing cross-platform development for various fields of applications. Since the first release in June 2005 Unity established themselves as a professional development environment for games, automotive, film, architecture, brands, gambling and educational technologies. They enable publishing for 25 different platforms including windows, macOS, Linux, game consoles, WebGL, VR, AR and mobile operating systems. Many common tools and systems which are needed to create a game environment come already shipped within the Unity Engine. Depending on the needs of the application this includes systems for physics, animations, rendering, UI and many more. Unity is highly modular and additional functionality

for the engine itself can be added via C# based scripts, which are often combined in packages. Unity packages can be found in the Unity Asset Store [6] which also provides community created packages, or in the inherent package manager [7]. Packages provided by the package manager are developed by Unity Technologies which also includes the (open-source) ML-Agents package which is used to connect the learning environment with the python API of the ML-Agents toolkit which is explained in subsection 2.4.2. As stated by Juliani et al.[23] the Unity engine is a great fit for Machine Learning applications. In their work, Unity (in combination with the ML-Agents Toolkit) is evaluated against other simulators in the field of machine learning. Unity is standing out as it is providing every aspect which is needed to create a variety of learning environments. With Unity's built-in sensors, physics and network components it is possible to recreate simple 2D environments like the Atari games, as well as simulating 3D real world environments or multiplayer applications.

2.4.1 Unity Work Environment

Developing a Unity application is done with the Unity Editor. The editor consists out of multiple windows which allow for basic drag and drop functionalities to edit and arrange game assets, navigate within the virtual three-dimensional space and playing the application in its current state (see Figure 2.7). Every project file which is supported by Unity is an Asset. Assets are for example C#-scripts, 3D meshes, audio files, images, scenes or prefabs. One of Unity's main Assets are scenes. Scenes are Unity specific data structures which contain a hierarchical arrangement of *GameObject*. At the end the application will have a scene as its entry point, but it is possible to display multiple scenes simultaneously. Similar to C# Unity follows an object orientated programming approach, with the *GameObject* as its main component. A *GameObject* can hold several components which will give the *GameObject* its functionality and representation.

The *Lost Survivors* application is programmed with the Unity engine, which communicates with a java based server via Hypertext Transfer Protocol (HTTP) requests. The game is developed for the mobile platforms Android and iOS but thanks to Unity's cross-platform functionality it is also possible to build a Linux application of the game.

2.4.2 ML-Agents Toolkit

The ML-Agents Toolkit is an open source project and mainly developed by Unity Technologies. The goal is to provide a general platform to utilize machine learning in virtual environments. In comparison to real world training data, virtual environments have the advantage of scalable training processes by running in parallel or with increased time speed. The output of training the machine learning algorithms is saved in a neural network model. Models can be integrated in the Unity application and can operate on any platform without any further dependencies. The toolkit provides cutting edge machine learning algorithms, which are ready to use without requiring any additional python programming. It is used in combination with Unity and is built on top of the PyTorch library. The core of the toolkit are its deep Reinforcement Learning algorithms, Proximal Policy Optimization [1] and Soft Actor-Critic [3]. It is also possible to attach own training algorithms using the provided python API and the gym wrapper. The toolkit enables to transform any Unity project into a learning environment to connect it with machine learning algorithms. The environment will feed the algorithms with observation data which is generated from its current state. The algorithms will then output actions which have to be processed within the environment to alternate its state.

[6] https://assetstore.unity.com/, Accessed: 06.06.2021
[7] https://docs.unity3d.com/Manual/upm-ui.html, Accessed: 06.06.2021

Figure 2.7: Unity Editor window layout (color theme light) with additional markers (red)

The window with marker 1 shows the hierarchy of the current selected scene. The second window (2) is for navigating the projects directories and Assets. In this layout the bottom panel (3) contains four window tabs from which the *Console* is selected. The console displays all logs, exceptions and warnings. In the center (4) the currently running simulation is rendered in the *Game* view. In the right panel (5) the *Inspector* tab is selected, which shows the in window 2 selected prefab asset named "MlAgent".

The toolkit consists of the following five main components. Figure 2.8 shows a UML diagram of the main components with their dependencies to the *Lost Survivors* project.

Learning environment Any Unity scene can be transformed into a learning environment by adding one or more implementations of the so-called agent. The agent can be added as a *GameObject* component and defines the behavior within the training environment. There can be multiple behaviors within one training environment. All agents of one behavior type are managed by its academy. The academies are connected to the external communicator. Agent and academy are part of the ML-Agents Unity SDK.

External Communicator This is also part of the ML-Agents Unity SDK. It connects the learning environment with the python API. This includes the academies but also the possibility to send customized data from the environment outside of the agent behaviors.

Python Low-Level API This is part of the *mlagents_ envs* Python package and is not part of the Unity application itself. The python API is the key point between the learning environment and the machine learning algorithms. It hands all data from the learning environment to the ML algorithms and controls the academies during the training process. The API is also the entry point for own implementations of ML algorithms.

Python Trainers Is part of the *mlagents* Python package with dependency on the *mlagents_ envs* package. It contains all the predefined training algorithms linked to the Low-Level API. It is used to start the training process with its training configuration file [8] and additional CLI options for the Unity application.

Gym Wrapper Is optional and has its own python package called *gym-unity*. The gym wrapper links to the gym toolkit provided by OpenAI and can be used to compare different reinforcement learning algorithms.

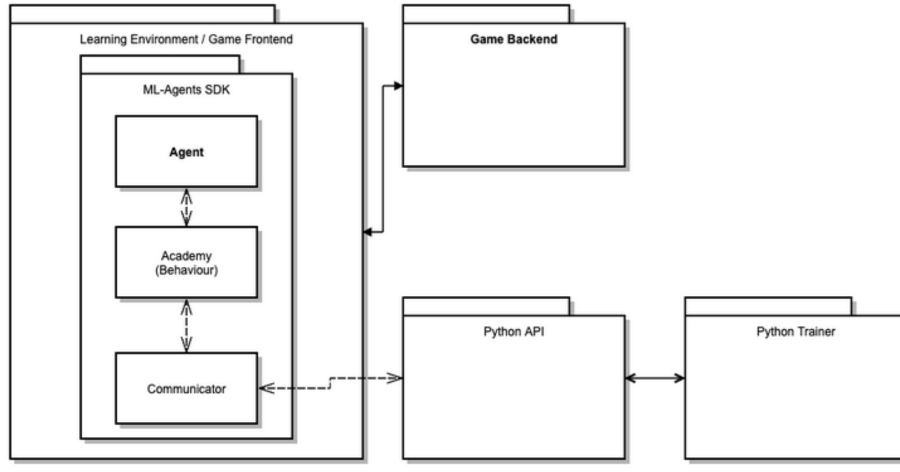


Figure 2.8: Project dependencies between *Lost Survivors* and the ML-Agents Package

To run the ML-Agents toolkit the prerequisite of Unity 2018.4 or later and Python 3.6.1 or higher are needed[9]. With the requirements in place only two packages need to be installed. One is the Unity C# SDK which can be installed with the Unity package manager and the *mlagents* python package. With the installed SDK, an agent can be created by adding an implementation of the agent component to a *GameObject*. The *GameObject* then needs to be included in the scene asset which should be used as the training environment. With at least one agent added to the scene, the training can be started by running the application in the editor or as a standalone built application. On startup the agent academy will be instantiated which will establish the connection to the agents and the low-level API. The agent will send its environment observations to the trainers via the low-level API and executes all actions it receives.

Agent actions

The action space $\mathcal{A}$ (see subsection 2.3.4) is transmitted as a vector $\vec{a}_n$ of floating point numbers with specified value ranges. Values can be either discrete or continuous. In a continuous action-space the value ranges between -1 and 1 ($\in \mathbb{R}$). While discrete action space values are equal to the naturals numbers ($\mathbb{N}$). On receiving the action vector, the agent translates each value (also called branch) into an individual action. An example for continuous actions would be to

[8] A YAML based file which defines the training algorithms and hyperparameters to be used during training
[9] for this work Unity 2019.4.15f0-2019.20f1 and Python 3.7.9 were used together with the ML-Agents version 1.8.1-preview (Release 14)

combine three branches into a three-dimensional direction vector $\vec{a}_{1...3}$ to move a character in an 3D environment. A fourth branch could be used additionally to define the character's movement speed. In discrete action space, each value represents an action. Multiple branches are only needed to enable simultaneous actions or to specify action behavior. In the game *Super Mario* [50] for example, only two branches ($\vec{a}_{1...2}$) would be needed to play the game. The first branch would define the move direction. To be able to jump simultaneously, a second branch needs to be defined. Another benefit of the discrete action space is that it is possible to filter invalid actions by masking them. For example, if the character stands beneath a stone and would not be able to jump, the jump action can be masked for the next step. This way the jump action will not be taken into account in the next policy update. Unity implementation of action masking is based on the work of Vinyals et al. [51]. To mask an action the branch and value have to be specified. Multiple actions of the same branch can be masked, but at least one action per branch has to be available.

Agent observations

Observations represent the game state for the agent and should include all necessary information for the agent to choose the best possible action. The best possible action is measured by the reward the agent gets for its actions. An observation can range from images of the current environment to simple truth values. Observations are transmitted to the network as a float vector of observations. The vector has a fixed size which is defined in the behavior parameters component, which can be seen in Figure 2.9.

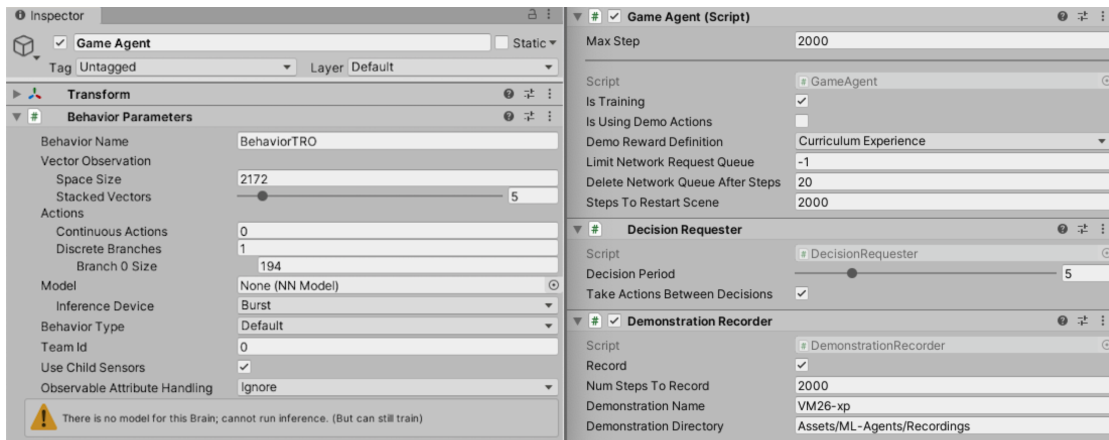


Figure 2.9: Inspector view of the agent *GameObject* and its components in the Unity Editor On the left side the behavior parameters are defined. The behavior name is matching with the training configuration of the python trainer. The sizes of the observation and action vectors are also defined here. The behavior type decides if the agent is training, if it is controlled by an existing model, or heuristic actions. On the right side the agent component is displayed. The *Max Step* parameter defines the length of an episode. The rest of the agent parameters are custom game specific parameters. The *Decision Requester* configures the interval in which new action decisions are requested from the academy. The *Demonstration Recorder* is used to record the agent actions to be used in future training runs.

Agent rewards

The most important part when designing the agent is its reward function. To successfully train an agent it is crucial to have a well-defined reward function. Rewards defined in the Unity environment are counted as extrinsic rewards and are transmitted to the ML algorithm. Reward signals can be positive or negative. A positive reward to reinforce good behavior and negative rewards when the agent shows unwanted behavior. For an optimal training, the agent receives a reward signal after each action step. In environments where it takes multiple steps to reach the goal, it is not always possible to define a reward after each step. Depending on how sparse or dense the reward signals in the environment are, it can be favorable to choose a different RL algorithm. According to the Unity documentation PPO achieves the better results in dense reward environments, while in sparse reward environments SAC is often more beneficial. For further information on the PPO algorithm used in this work, see section 2.3.

ML-Agents setup

For the additional training algorithms GAIL and BC a recording of agent actions, observations and rewards is needed. To generate demo behavior the heuristic method of the Agent has to be implemented. Instead of the communicator controlling the agent, the heuristic method is then used to control the agent.[10] In this method action vector values can be defined by manual user input or rule-based, which will later be used to include predefined behavior (see section 3.7). When controlling the agent with Heuristic behavior the Unity *Demonstration Recorder*-component can record all actions, observations and rewards and saves them into a *.demo* asset.

All hyperparameter settings for the machine learning algorithms are defined in a training configuration YAML file. The file contains the behavior configurations for all agents in the environment. The configuration file is not part of the environment itself and needs to be passed to the python trainers when starting the training. The configurations also include which algorithms are used for the training (an example can be seen in section 3.8). Unity provides state-of-the-art deep Reinforcement Learning algorithms PPO and SAC but also methods like BC, GAIL, curiosity, parameter randomization, curriculum learning and Random Network Distillation to shape the agent's behavior. Because multiple agents with different behaviors can exist within the same environment, different training algorithms can be used in parallel. As *Lost Survivors* has a very dense reward environment when using xp as decisive, PPO is used for the training. This algorithm is combined with BC and GAIL to give the agent a head start and increase the cumulative reward in the first training steps. More details will be explained in section 2.3.

The training process can be monitored via TensorBoard. TensorBoard is a tool from the open source library TensorFlow[11] and helps to visualize ML training processes. The python trainer is saving statistical data about the policy, learning loss functions and the environment. This allows for more detailed analysis on whether the training was successful or what parameters can be improved. It is also possible to implement custom statistical data in the environment for further analyses. For *Lost Survivors* the additional data was recorded for all executed actions and if an agent was reaching its policy goal or was terminated earlier.

[10] Is also possible to control the agent with an existing model of a trained neural network, which can be useful when the trained behavior should be included into the application.

[11] TensorFlow is a open-source framework optimized for machine learning processes. `https://www.tensorflow.org/`, Accessed: 28.065.2021

Chapter 3

Implementation

This chapter contains a detailed overview on the implementation of previous explained machine learning algorithms (see section 2.3) into *Lost Survivors*. As *Lost Survivors* is also implemented with Unity, this part will cover the integration of the ML-Agents package (see subsection 2.4.2). To transform the game into the agent's learning environment the existing game architecture needs to be modified, which will be covered in the first section. The next section pictures the software architecture before getting into implementation details on the agent actions, observations and rewards.

3.1 Application Preparation

Usually the *Lost Survivors* game is run on Android and iOS mobile devices, but training an agent requires a lot more computational power than a mobile device could offer. To be able to utilize the high computational power of the HAW Hamburg server, it was needed to prepare the *Lost Survivors* game for a Linux operating system. To create a Linux version only a few mobile specific code library's like the Apple Store payment connection, notification services and Google Play Store API had to be removed from the code base. Because the ML-Agents package came with its own dependency to the Googles protocol buffers, the already existing one was replaced by it. To decrease the application loading time, additional aesthetic content was removed. This included mainly assets for the Heads-Up Display (HUD), popups and other user interfaces, which were not needed for the agent to interact with its environment. All removed code was not affecting any gameplay modules. This enables running the *Lost Survivors* application on external servers. The technical specifications of the server can be found in section 3.8. As the game usually runs as a single instance per operating system there had to be made one adjustment to an external code package which was responsible for storing player data. To avoid using the same player data for all agent instances, the stored data was instanced with a random ID, which made the stored data unique.

The game *Lost Survivors* is separated in a frontend and backend part. The backend is responsible to calculate and maintain the complete game logic, while the frontend is responsible to display the game state and capturing players actions. For this work an additional backend server was set up which is located at InnoGames and holds all player data. To improve the training speed, the balancing data on the backend was speed up with a multiplier of 2000 to avoid long waiting times between the actions. The player state is loaded from a preset which can be setup and configured by game design or QA. The preset data is stored in the backend and cointains information like player level, items, soft currency, etc. For the training of this work the preset

was set up to start right after the tutorial. The tutorial is an introduction to the game with a linear sequence of actions. During the tutorial the player is very limited in its choices, so that it is less useful to train the agent in this state.

Another part which had to be adjusted in the frontend was an included prediction system. For some requests the frontend predicts the backend response instead of waiting for the actual backend response message. Because both systems are running with an up-scaled speed, the prediction system was turned off to avoid desynchronization between backend and frontend. It still can happen that the frontend is not synchronized with the backend when for example the resources have been spent on an action, but have not been removed from the inventory due to network delay. If the agent then tries to spend the resources twice, the backend will throw an error. To stabilize the training process, backend errors are ignored in the frontend to avoid reloading the game during the training. The downside of this is that for some actions the returned reward might not be transmitted from the backend in the same agent step which could lead to the agent optimizing for the wrong action.

3.2 Software Architecture

This part will cover the C# side of the implementation. To create the training environment, a new namespace inside the *Lost Survivors* project has been created. Figure 3.1 shows an Unified Modeling Language (UML) diagram to picture the class dependencies and how they are integrated into the project. The following sections will describe the functionalities of the classes seen in the UML diagram. They are sorted by the execution time of the associated class when starting the application.

SceneSetup To enable the ML-Agents package in *Lost Survivors* an additional setup scene is interposed. The scene is used as the entry point for the Unity application and contains the `SceneSetup` class. The `SceneSetup` has three main responsibilities. It will load the game, instantiate the training, and it will reset the environment when an agent episode has ended. On the first scene startup a `DummyAgent` *GameObject* is instantiated before loading the game. Creating the agent first is necessary to avoid running into timeouts with the communicator during long loading times. After the agent is created, the application starts loading the game assets and connects itself to the backend server to load the player state. The connection is establishes by authenticating with a token.

MainContextInstaller The main context installer is loaded together with the game assets. By loading the game, the `MainContextInstaller` will be executed, which will also install all bindings for the agent and its components in the dependency injection container (`DIContainer`). Because agents will be destroyed and re-instantiated multiple times during the training, the `SceneSetup` will hold and persist the `DIContainer` to inject the agents dependencies during instantiation. When the game is done loading, the `GameAgent` is instantiated and the `AgentAction`, `AgentRewards`, `AgentObservations`, `AgentActionMaskBuilder` and the `AdditiveStatsRecorder` get injected.

MLContextInstaller Installs all ML-Agent related dependencies. It is triggered by the `MainContextInstaller`.

DummyAgent The `DummyAgent` component is an empty implementation of the ML-Agents agent interface. This way the agent academy and the connection to the low-level API is already established on the application startup before loading the game.

GameAgent The `GameAgent` implements the Agent interface from Unity's ML-Agents package. The Game agent is initialized with the parameters from the training configuration file. The agent can be in different states and either receives actions from the academy or by heuristics.

Agent The Agent interface is part of the ML-Agent package and contains five methods to be implemented by the inheritor. `Initialize()` is called at the beginning of the Agents training and should prepare the agent to act in the environment. `Heuristics(float[] actionsOut)` is used to control the agent within the environment instead of the academy. `CollectDiscreteActionMasks(DiscreteActionMasker actionMasker)` masks all actions which cannot be used in the next step. `OnActionReceived(float[] vectorAction)` is called to execute an action. The float array is one dimensional and represents the action as a float value. `CollectObservations(VectorSensor sensor)` collects the game state S and sends it back to the python trainers.

Academy The academy is the link between the communicator (not displayed in the diagram) and the agent. It controls the agent with the actions sent by the python trainers. It also provides the environment parameters from the training configuration file to set up the agents behavior and reward function.

DemoActionProvider For the recording of agent actions, the `DemoActionProvider` is used to provide the heuristic actions. The `DemoActionProvider` determines a single action value which is based on simple rules regarding current game state.

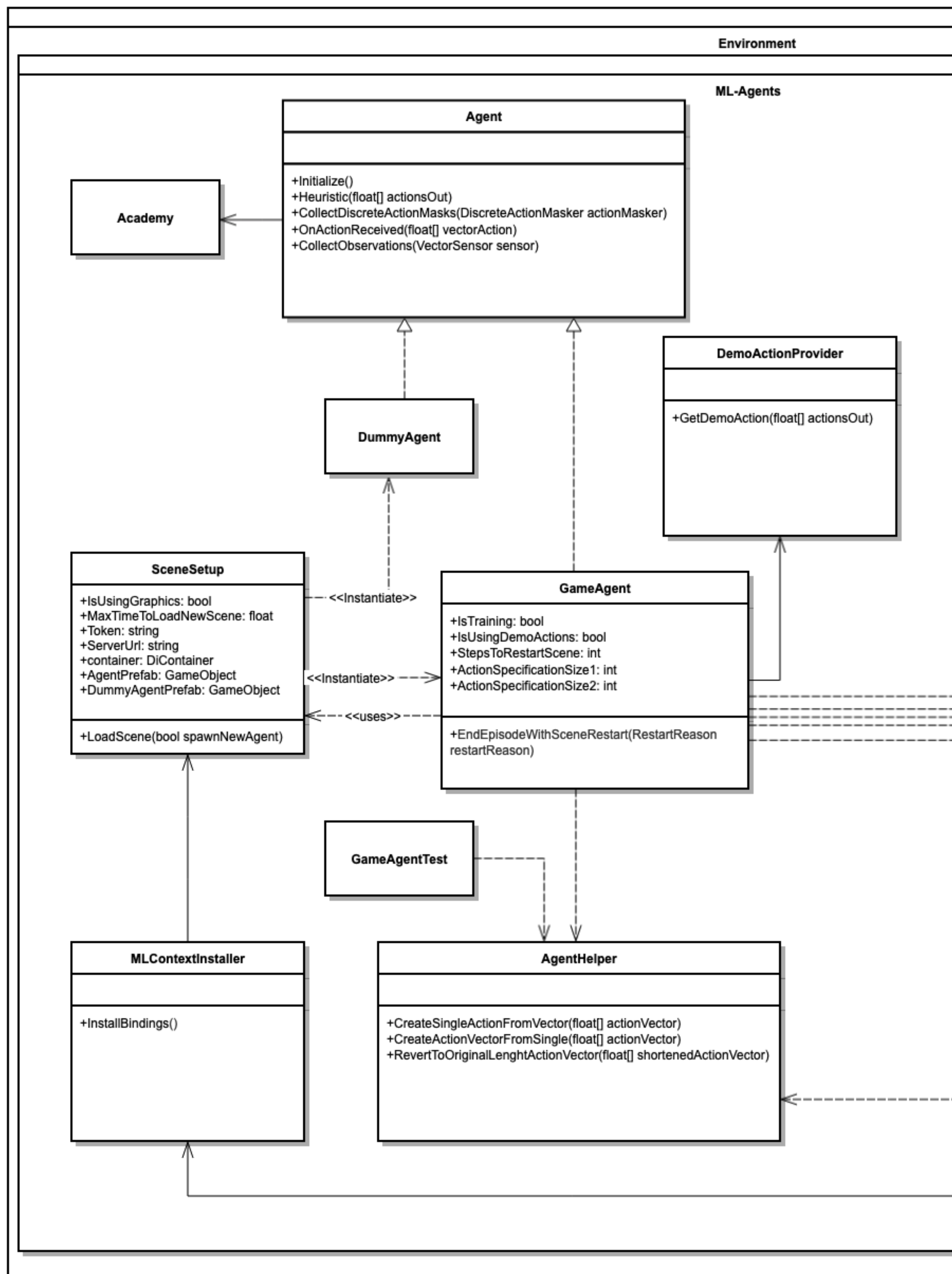
AgentHelper The single action value the `GameAgent` receives gets translated to the associated action by the `AgentHelper` before it is executed by the `AgentActions` class. It will transform the action value into a three-dimensional action vector containing the action, the first action specification $S1$ and the second action specification $S2$ (see section 3.3).

AgentActions The `AgentActions` class holds all game services which are necessary to trigger a certain action. Services typically construct the server request which triggers the action execution in the backend. The composite of the agent actions can be found in section 3.3

GameServices and DataProvider On the frontend side the game state is persisted via value objects which can be accessed via the `DataProviders`. `GameServices` allow to manipulate the game state, which is done mainly through sending server requests. `GameServices` are also responsible to handle server responses. An example value object is the `BuildingVO` which contains the game definition for a specific building. The `IBuildingDataProvider` offers methods like `CanSellBuilding(BuildingVO buildingVO)` which returns a boolean value to ask if a building can be sold. Then the `IBuildingService` can be used to call the `SendSellBuildingRequest(BuildingVO buildingVO)` method, which will send a request to the server to sell the building. Note that the services do not check for themselves if the request is valid.

AgentRewards After each executed action, the `AgentRewards` class sets a reward. Depending on how the reward function is set up via the training configuration file, the `AgentRewards` will end the agents' episode for reaching the curriculum goal or not reaching a reward at all for a defined amount of agent steps. Because some rewards are based on the game state, `AgentRewards` is using `AgentObservations` to define the rewards.

AgentObservations The main purpose of the `AgentObservations` class is to extract the current game state in the form of attributes which get encoded in an observation vector. Its observations are also used to create the action mask and define the agents rewards. The observations are created by using existing game services and data providers.


Environment
ML-Agents
Agent
+Initialize()
+Heuristic(float[] actionsOut)
+CollectDiscreteActionMasks(DiscreteActionMasker actionMasker)
+OnActionReceived(float[] vectorAction)
+CollectObservations(VectorSensor sensor)
Academy
DemoActionProvider
+GetDemoAction(float[] actionsOut)
DummyAgent
SceneSetup
+IsUsingGraphics: bool
+MaxTimeToLoadNewScene: float
+Token: string
+ServerUrl: string
+container: DiContainer
+AgentPrefab: GameObject
+DummyAgentPrefab: GameObject
+LoadScene(bool spawnNewAgent)
<<Instantiate>>
<<Instantiate>>
<<uses>>
GameAgent
+IsTraining: bool
+IsUsingDemoActions: bool
+StepsToRestartScene: int
+ActionSpecificationSize1: int
+ActionSpecificationSize2: int
+EndEpisodeWithSceneRestart(RestartReason restartReason)
GameAgentTest
MLContextInstaller
+InstallBindings()
AgentHelper
+CreateSingleActionFromVector(float[] actionVector)
+CreateActionVectorFromSingle(float[] actionVector)
+RevertToOriginalLenghtActionVector(float[] shortenedActionVector)

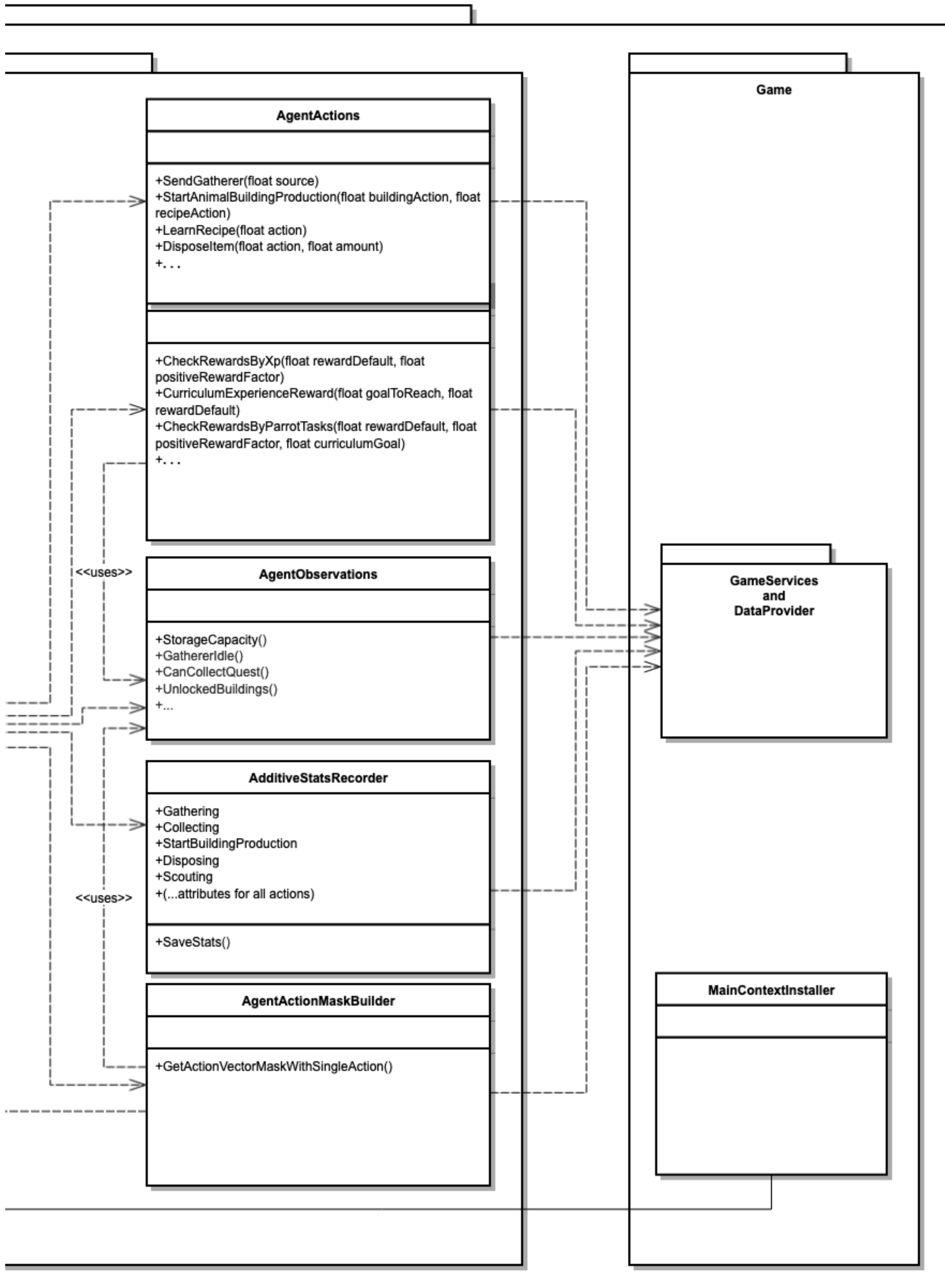


Figure 3.1: UML diagram of the implemented agent in *Lost Survivors*

The figure shows the conceptional UML class diagram of the agent within the game environment. The implementation of the agent interface (from the ML-Agents package subsection 2.4.2) lies inside of the *Lost Survivors* code-base. When starting the application the `MainContextInstaller` executes the `MLContextInstaller` which sets the `DIContainer` (dependency injection container) in the `SceneSetup`. With the `DIContainer` the `SceneSetup` is ready to instantiate the `GameAgent` with all its required dependencies to the game services. For better separation of responsibilities the `GameAgent` uses separate interfaces for actions, rewards, masking, observations and statistical records.

AgentActionMaskBuilder Because not all action are available in every state, the `AgentActionMaskBuilder` uses the `AgentObservations` to create an action mask. The GameAgent uses the action mask to filter for possible actions before executing the next step.

GameAgentTest Unit test class to assure the correct transformation for actions in form of a single float value into a three-dimensional array and vice versa.

AdditiveStatsRecorder All executed actions get recorded by the `AdditiveStatsRecorder` for the statistical evaluation of the `GameAgent`. The recorded actions will be summed up over one episode before they get transmitted to the python trainers. At the end of the training the results can be visualized with the help of TensorBoard.

3.3 Actions

For *Lost Survivors* no simultaneous actions are required. Even though it would be possible to start different productions at the same time, it has no significant impact to start actions sequentially. Most actions in *Lost Survivors* are time bound in order of minutes, while the agent acts in a time frame of milliseconds. In order to reduce complexity, this work assumes it is not a big difference (if any difference at all) to start an action some milliseconds delayed because of the sequential order. In fact, it might even be beneficial because every action is based on new observations. To further reduce learning complexity, the agent is running only on a subset of all possible actions in the game. The selected actions the agent is able to perform are carefully chosen in a way that the agent is able to complete the full core loop cycle of Figure 2.2. Hereby the actions are separated in actions which can be controlled by the python API and actions which are executed by the agent automatically as soon as they become available (with consideration of their requirements, see Table 3.3). All automated actions create advantages for the agent in items, experience points, soft currency and unlocking more possibilities. Except for “Send parrot Task” they also have in common that they do not have any costs on execution (like soft currency or items). Because of the comparably great advantage to use "Send Parrot Tasks" it was decided to use this action automatically. Automatic actions are independent of the training algorithm and executed in parallel to the selected training actions.

The Agent of this work is able to perform the following actions. Table 3.1 and Table 3.2 are showing actions which the agent can actively choose from based on its observations. Table 3.1 shows actions which include predefined logic based on the current game state. And lastly Table 3.3 includes all actions which are automatically executed as soon as they are possible.

Vector index	Action	Requirements	Specifications	Costs	Rewards
12	Smart dispose	-	-	-	-
13	Smart gathering	-	-	-	-
14	Get parrot task resource	-	-	-	-

Table 3.1: Agent action combinations

Vector index	Action	Requirements	Specifications	Costs	Rewards
1	Do nothing	-	-	-	-
2	Send gatherers	Gatherers idle Scouted locations	Gathering location index	-	-
3	Dispose items	Items in storage	Item index to pick from storage	Items	-
4	Scout locations	Unlocked by level	Index of locations to scout	Soft currency	Resource location
5	Building production (Crafting)	Recipe unlocked Building unlocked Needed resources	Building index Recipe index	Base goods	-
6	Unlock recipes	Unlock by experience points	Recipe index to learn	Soft currency	Recipe
7	Animal production	Building unlocked Needed resources	Building index Recipe index	Base goods	-
8	Start build buildings	Building unlocked	Building index	Soft currency	-
9	Upgrade buildings	Building unlocked	Building index	Soft currency Goods	Experience points
10	Start building island expansion	Unlocked by level	-	Soft currency	-
11	Cultivation production	Building unlocked Needed resources	Building index	Base goods	-

Table 3.2: Agent actions

Action	Requirements	Specifications	Costs	Rewards
Collect from gatherers	Gatherers production ready Storage space	-	-	Items Experience points
Collect building production	Production finished	-	-	Item Experience points
Collect from cultivation buildings	Production started	-	-	Items Experience points
Collect From animals	Animal-Survivor unlocked Habitat build Animal fed	-	-	Items
Send parrot task	Parrot available Resources in storage	-	Goods Base goods	Soft currency Experience points
Unlock parrot slot	Enough parrots sent	-	-	Parrot task slot
Start quests lines	Unlocked by survivors Game progress	-	-	New quests
Collect quests	Quests task fulfilled	-	-	Soft currency
Finish build buildings	-	-	-	Experience points
Finish expansions	-	-	-	Experience points
Collect chests	-	-	-	Goods Materials Experience points

Table 3.3: Automatic executed agent actions

Some listed actions need to be specified to make sense in the game context. For example, if the agent wants to build a building it has to specify which building to build. For some actions like the building productions even two specifications are needed to define the production building and the affiliated recipe. The action specifications $S1$ and $S2$ account for all actions. To strike a good balance between a small action space and to cover most of the game actions (especially on the lower player levels, where most of the training will happen) the action specifications are defined as $S1 = 9$ and $S2 = 5$. For example, with this setup it is possible to perform 5 different actions on each of the first 9 production buildings. With 14 actions in total this produces an action vector of $\vec{a} = \begin{pmatrix} 14 & 9 & 5 \end{pmatrix}$. As it is not needed to execute these actions in parallel, the vector $\vec{a}_{1...3}$ can be transformed into the vector $\vec{a}'_{n=1}$ with the length of one. With Equation 3.1 a unique value for each specified action can be calculated.

$$\vec{a}'_1 = (\vec{a}_1 * S1 * S2) + (\vec{a}_2 * S2) + \vec{a}_3 \tag{3.1}$$

With the defined specification sizes $S1 = 9$ and $S2 = 5$, Equation 3.1 results in 680 possible actions. Because not every action needs both specifiers, the action space can be further reduced by eliminating all unnecessary action values. With Algorithm 3 all unused action values are removed which leaves 194 possible actions to choose from for the machine learning algorithm.

Algorithm 3 Clear action values without specifications

```
1:  for action = 0, 1, ... 13 do
2:      if specification of action is 0 then
3:          remove S1 * S2 − 1 = 44 action values starting at: action * S1 * S2 + 1
4:      end if
5:      if specification of action is 1 then
6:          for s = 0, 1, ... 9 do
7:              remove S2 − 1 = 4 action values starting at: action * S1 * S2 + s * S2 + 1
8:          end for
9:      end if
10: end for
```

For all 14 actions each action has a default value range of 45 actions, if the action has two specifications. When there is only one specification (checked in line 5) the range can be reduced to 9 action values in total. This is done by removing each action value which is associated to the second specification. In the case that the action has no specification at all (checked in line 2) only one action value is needed. The other 44 values associated to the action specifications can be removed.

3.4 Action Masking

Every action the agent requests is first checked to meet all requirements before it is translated into an HTTP request to the backend. If the agent tries to execute an action without the requirements it will not be executed. To avoid having the agent learn these rules, state dependent action masking is applied. The masking used in this thesis works similar to the *AlphaZero* algorithm published by DeepMind from Silver et al. [52]. In *AlphaZero* a Reinforcement Learning algorithm was trained to play the games chess, shogi and go with the specialty that the agent did not know which game it was playing. The only domain knowledge the algorithm had was based on the game rules. Illegal moves were masked out from the action representations by setting their probabilities to zero, and renormalizing the probabilities for remaining moves. The masking in *Lost Survivors* is based on the action requirements. For actions with specifications, each action variation is checked against its requirements. In the most extreme case this could lead to the agent being able to perform only the "Do nothing" action, which should by game design never happen.

3.5 Observations

As this work does not aim to imitate human player behavior by clicking screen elements, no visual observations are included. The game state is represented by observations deriving in majority from Boolean-values. In addition, different states of items, buildings and recipes are transmitted in the one-hot fashion. The complete list of observations made for the *Lost Survivors* game can be found in Table 3.4.

Observation Description	Observation vector size
Missing experience points to the next level	1
Current owned soft currency	1
Item storage capacity	1
Gatherers with idle state	1
Gatherers with a production ready to collect	1
Gatherers in production	1
Can any new gather-spot be scouted	1
Can collect from any production building	1
Can collect from any animal building	1
Can collect from any cultivation building	1
Can complete a parrot task	1
Can collect any quest	1
Can start any quest	1
Can build any new building	1
Can finish any building construction	1
Can upgrade any building	1
Can start building any island expansion	1
Can finish any island expansion	1
Can learn any new recipe	1
Production building recipes ready to be produced	$9 * 5 = 45$
Animal building recipes ready to be produced	$9 * 5 = 45$
Cultivation building recipes ready to be produced	$9 * 5 = 45$
Items in storage	350
Items missing for productions	350
Items missing for Parrot tasks	350
Buildings built on the island	237
Buildings without active production	237
Buildings ready to be built	237
Learned recipes	257
	Total: 2172

Table 3.4: Agent observations

Because of the fixed observation size, observations for items, buildings or recipes are created with a list of all possible items, even if the agent only reaches a fraction during its training states. To enable the agent to take actions based on previous observations, the observation vector can be stacked multiple times. For example, in the *Lost Survivors* game it can be useful to know if some productions were already running in previous steps, so the agent can make sure to have enough space in its storage. For this work it turned out beneficial for the results to stack the observation vector five times, creating a total observation vector size of 10860.

3.6 Rewards

In this work the different reward functions were used to evaluate different behavior. The reward functions focused on different aspects of the game to see if the agent creates different strategies to accomplish its goal. This as well has the benefit to focus the agents' behavior on a specific game aspect. With different reward functions it is possible to reevaluate the game if the game designer changes parts of the game design data during the development process. It can also be used to train the agent on a specific play style to mimic different player types. A general component to measure game progress are experience points (xp). Players' progress in the game by collecting xp for their actions. With a certain amount of xp the player reaches the next player level which unlocks more game features. As shown in Figure 2.2, every main activity is rewarded with xp. Therefore, xp are used to create a dense reward signal.

When designing a reward function it can be challenging to balance rewards (or punishments) against each other. One learning from the *Hummingbird* example [31] was to keep it rather simple instead of having a mix of small rewards for different actions. It is also recommended keeping the total reward in a range from -1 to 1. Another learning from the *Hummingbird* example is that too much negative rewards can lead to a paralyzed agent. With too many negative rewards, the agent will try to do nothing, to not get more punishment. The reward function used in this work is defined as the following Algorithm 4:

Algorithm 4 Curriculum reward function

```
1:  if no xp gained then
2:      reward = −1/steps
3:  end if
4:
5:  if no xp gained for x steps then
6:      reward = −1
7:      start a new episode return
8:  end if
9:
10: if curriculum goal xp reached then
11:     reward = 1
12:     start a new episode return
13: end if
14:
15: reward = (gained xp since the last step)^2 ∗ totalfactor
```

When the agent does not gain any xp in the current step, the agent will be punished. The punishment is separated into two states. As it is not possible to collect xp in each single step the agent gets just a very small negative reward of $\frac{1}{steps}$ where in most of the training's $steps = 2000$. However, the agent should always collect at least some xp within the last x steps. If the agent fails to collect any xp within the defined range x the current training episode will be stopped. For most trainings' $x = 500$. A positive episode end could be reached by the agent if the current curriculum goal was reached. The goal is defined as a fixed amount of xp which raises in each lesson. If the goal was not reached, but the agent was still collecting new xp, a few xp will be granted to nudge the agent into the right direction. The gained xp are counted as squared to value bigger quantity of xp even more. Higher quantities of xp ($\sim 50 - 200$) are given for completing complex tasks like buildings and upgrades, while lower xp ($\sim 1 - 10$) are given out for collecting single productions and base resources. To make sure the reward does not become bigger than 1 it was factorized by $totalfactor = 0.0002$.

3.7 Demo Actions

As mentioned in section 2.3 it is possible to provide the agent with demo actions to boost the training speed via BC and GAIL. Therefore, the `DemoActionProvider` has been implemented. In combination with the `GameAgent` the `DemoActionProvider` acts as a player following a predefined rule set. For each action the `DemoActionProvider` checks the action conditions in a sequential order. The first action becoming available will then be returned and executed by the agent. The rule set is determent by the order in which the `DemoActionProvider` checks the action conditions. An example action check can be seen in Listing 3.1.

```
1   float[] actionsOut = new float[3];
2
3   // Default: Do nothing
4   actionsOut[0] = (int) ActionsEnum.DoNothing; // Action
5   actionsOut[1] = 0; // Specification 1
6   actionsOut[2] = 0; // Specification 2
7
8   // Scout if possible
9   List<int> countUnlockableGatherSpots =
10    agentObservations.GetUnlockableGatherSpotsByIndex();
11  if (countUnlockableGatherSpots.Count > 0)
12  {
13    actionsOut[0] = (int) ActionsEnum.Scouting;
14    return actionsOut;
15  }
```

Listing 3.1: First action check in the `DemoActionProvider`

In line 1 the action vector $\vec{a}$ is defined as `actionsOut`. As default the vector is initialized with the “Do Nothing” action. The first action which is checked in the demo action provider is if scouting of new gatherer spots is possible. The condition for this action to be available is that gatherer spots are unlocked by player level and that the player has the required amount of soft currency. Both of which is checked in line 9 - 10 via the `AgentObservations`. If a new gatherer spot is available, the action $\vec{a}_1$ is adjusted and returned to the `GameAgent`. In this case the demo action will always scout the first possible gather spot. In the later training the `GameAgent` will be able to specify in $\vec{a}_2$ which of the unlockable gathers spots will be scouted. For scouting $\vec{a}_3$ is not needed. The strategy used for the training depends on the used reward function. For xp depended rewards the conditions were checked in the following order:

Scouting → BuildingBuilding → UnlockingRecipe → CollectingChest → CollectingQuest → FinishingExpansions → FinishingBuilding → StartingQuest → StartExpansionConstruction → StartingBuildingUpgrade → SmartGathering → SendingParrot → SmartDispose (if no storage space left) → GetTaskResource → StartingBuildingProduction → StartingAnimalProduction → StartCultivationProduction → CollectCultivationProduction → CollectingAnimalProduction → CollectingBuildingProduction → Collecting → DoNothing

The sequence is complementary to the designed gameplay. It is not guaranteed to be the optimal way but gives a good direction for the later training. The sequence above follows two main incentives. First, use up as many resources and prepare for further productions. If no further spending of resources is possible, check if there is enough storage space to collect goods and finished productions. Note that all above automated actions listed in Table 3.3 have no effect on the agent training and are just listed for completion.

3.8 Training Setup

Machine learning requires a lot of computational power to come to results in a timely manner. For the training process of this work the hardware was split up on two servers. One server was running the games' backend and was located at InnoGames. Because of legal and security reasons, the game logic and balancing data was not allowed to leave the InnoGames network. The training process itself was running on the university servers of the HAW Hamburg. The university server runs with 18 CPUs and 256 GB RAM. The server is managed via docker and is running the operating system is Linux. Since the application runs the training in batch mode without graphics, no significant GPU power is needed.

The *Lost Survivors* game has a single entry point for one player per application. Additionally, all game internal services and data providers are designed to persist data of one player. Because of this it is impossible to log in with multiple players within one application. This takes away one of the main training optimization possibilities. In contrast, in the *Hummingbird* project and other examples from the Unity ML-Agents package, the training scene contains multiple environments (often 20 or more). To still train multiple agents simultaneously, the server has to launch multiple application instances, which connect to the same python trainer. As each application allocates memory, this creates unusable memory space. The server was able to conveniently run up to 18 simultaneous applications.

The python trainer also runs on the university server and is initialized with the hyperparameters from the training configuration file. The configuration file which achieved one of the best results is displayed in Listing 3.2.

```
behaviors:
  Behavior_LostSurvivors:
    trainer_type: ppo
    hyperparameters:
      batch_size: 128
      buffer_size: 500000
      learning_rate: 0.0003
      learning_rate_schedule: constant
      beta: 0.001
      epsilon: 0.2
      lambd: 0.99
      num_epoch: 10
    network_settings:
      vis_encoder_type: simple
      normalize: true
      hidden_units: 128 # higher = relation between observations and actions is
    more complex. Typical range: 32 - 512
      num_layers: 4
      memory:
        sequence_length: 64
        memory_size: 256
    max_steps: 5.0e7
    time_horizon: 2000
    summary_freq: 5000
    keep_checkpoints: 5
    checkpoint_interval: 50000
    threaded: true
    init_path: null
    reward_signals: # Environment rewards
      extrinsic:
        strength: 1.0
        gamma: 0.99
      gail:
        strength: 0.05
```

```
34         gamma: 0.99
35         encoding_size: 128
36         demo_path: VM26_xp.demo
37         learning_rate: 0.0005
38         use_actions: true
39         use_vail: false
40     behavioral_cloning:
41         demo_path: VM26_xp.demo
42         strength: 0.5
43         steps: 150000
44 environment_parameters:
45   min_steps_to_reach_reward: 500 # Restart is forced when the agent is not getting
       a reward for the amount of steps defined.
46   reward_update_steps: 1 # Defines how many steps the agent does before checking
      his rewards. Can be used to fake sparse rewards.
47   reward_default: 0
48   reward_factor: 0.0002
49   reward_function_specification: 1 # Defines the reward function used by the agent
50   reward_to_reach:
51     curriculum:
52         - name: Lesson0
53           completion_criteria:
54             measure: reward
55             behavior: BehaviorTRO
56             signal_smoothing: true
57             min_lesson_length: 100 # Average reward on XX episodes
58             threshold: 0.8 # Reward to reach
59           value: 300 # Xp goal value for environemnt
60         - name: Lesson1
61           completion_criteria:
62             measure: reward
63             behavior: BehaviorTRO
64             signal_smoothing: true
65             min_lesson_length: 100
66             threshold: 0.8
67             require_reset: true
68           value: 400
69         - name: Lesson2
70 #Lesson2, Lesson3, ...
```

Listing 3.2: Training configuration file

Chapter 4

Evaluation

Before getting into the results of the agent training, a short review of developing the agent in parallel to a running production is given. The following section then covers the training results which are structured in chronological order to the development. Hereby, all significant intermediate steps and results are evaluated in detail. The last section includes which difficulties occurred during implementation and also during the training of the agent.

4.1 Continuous Development

To prove that ML is also working in a continuous development process, in the implementation of this work, the game was continuously updated with the master branch. Over this time more than 10,000 commits have been added by 25 team members. The merges into this work's implementation were done at the end of the sprints or milestone goals. Not all merges were conflict free. Most of the conflicts accrued in the services or data providers which had been previously modified to enable the agent to train (see section 3.1). Some conflicts would have been evitable with this work being included into the master branch. For example renamed getter methods in the services which were also accessed by the game agent. However, as this work was exploratory research, it was not planned to be included into the project. Speeding up the durations via the balancing data also led to conflicts when merging changes in balancing. Unfortunately, the game architecture at this point was not able to decrease productions times differently, although time boosters were already planned into future production cycles.

4.2 Training Results

In total, over 30 training runs with different sets of hyperparameters and reward functions have been conducted. Similar to the game, the training runs were also executed continuously with the implementation of the `GameAgent`. The training runs had between 0.3 and 11.5 million training steps with a training time between two hours up to eight days. Noticeable, the run with the longest duration is not equals to the run with the most training steps. This partially relates to the fact that the complexity of the game agent increased during the implementation process. Not all actions and observations were available in the first training runs. With each change in the agents observation or action space the parameters were reevaluated and eventually changed for future runs. Resulting from that, the one training run which achieved the best performance as

an agent will be compared on different aspects. In the following text the main findings will be marked down with corresponding explanations.

With the knowledge of previous research the first training runs started with the combination of BC, GAIL and PPO. Due to the dens reward environment, PPO was chosen over the SAC as deep reinforcement learning algorithm. Another reason for taking PPO was the better compatibility with GAIL.

4.2.1 Disabling Curiosity

The first big change in the training setup was the removal of the curiosity module for future training runs. With the curiosity module enabled, the reward showed too high fluctuations, which can be seen in Figure 4.1. The red line indicates the reward with the settings from the final training run, while the others are executed with curiosity enabled. Even though the strength of the curiosity module was only 0.08 while the extrinsic reward was set to 1 it had a strong effect on the training. Limiting the learning rate had no significant effect. It was also tested to limit the rewards in the total value, which led to the same strong fluctuations however in a smaller scale. Therefore, the curiosity module was considered to be of no avail. The curiosity module has been tested with the hyperparameters shown in Table 4.1.

gamma γ	**strength**	**encoding size**	**learning rate η**
0.99	0.08	64	3.0e-03
0.99	0.08	64	3.0e-05
0.99	0.08	128	3.0e-04

Table 4.1: Curiosity hyperparameters
The different training configuration sets with which the curiosity module was tested.

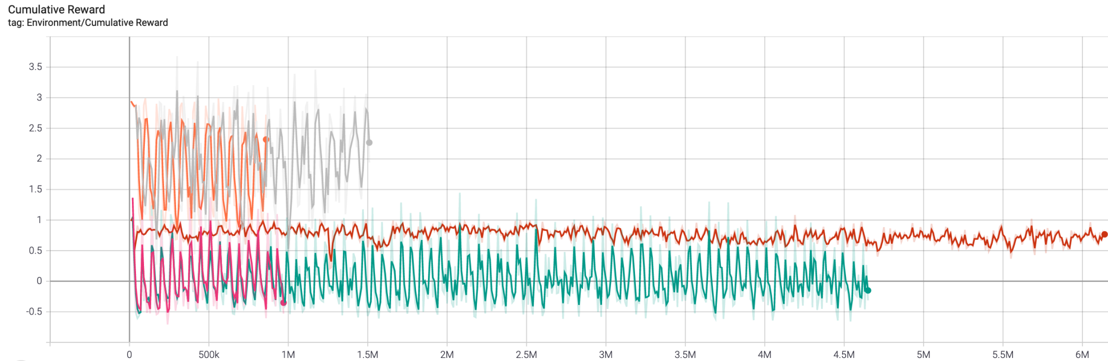


Figure 4.1: Trainings with and without curiosity in comparison

The graphs colored in gray, orange, magenta and green have the curiosity module enabled. All of these graphs show a high fluctuation in rewards. The red graph has not used the curiosity module which lead to a much more stable reward.

4.2.2 Curriculum Learning

Introducing curriculum learning to the training process has quickly shown to be helpful. An overview of all successful curriculum training runs can be seen in Figure 4.2. Before curriculum

learning was introduced, the agent would receive its rewards solely based on the gained xp. Previously the reward function was equal to Algorithm 4 without lines 10 - 13 and a different total factor. The idea to introduce curriculum learning was to create a clear goal for the agent instead of an endless episode of rewards. With the curriculum reward function, episodes are terminated with the maximum reward of one as soon as the curriculum goal is reached. This way the optimization goal is clearly defined. Low amounts of xp collection is avoided as the goal is to achieve a high amount of xp with the least amount of steps. By increasing the goal in each lesson it is still possible to optimize for a maximum of possible experience points. The reward is expected to be roughly maintained at the same level, however, with drops at the beginning of new lessons, due to the agent which has not yet learned to achieved its next goal. The respective training runs can be seen in Figure 4.2 (lessons) and Figure 4.3 (rewards). It is important to note that the agent might use different strategies in its lessons. As an example: the agent might pass a lesson with lower xp goals by just collecting water, but when the lessons increase, collecting only water will not bring enough reward to reach the goal. As a result, the agent will override its weights in favor to the next goal and collecting water will become less important. Therefore, the agent will adapt its learnings to strive only for the highest possible amount of xp. The lessons in between only serves the agent as guidance and gives the agent a better initial state for the next lesson. If the optimal strategy is far apart from the initially strategies chosen by the agent, the risk remains that the initial lessons may influence the training in a wrong direction and could result in a local minimum.

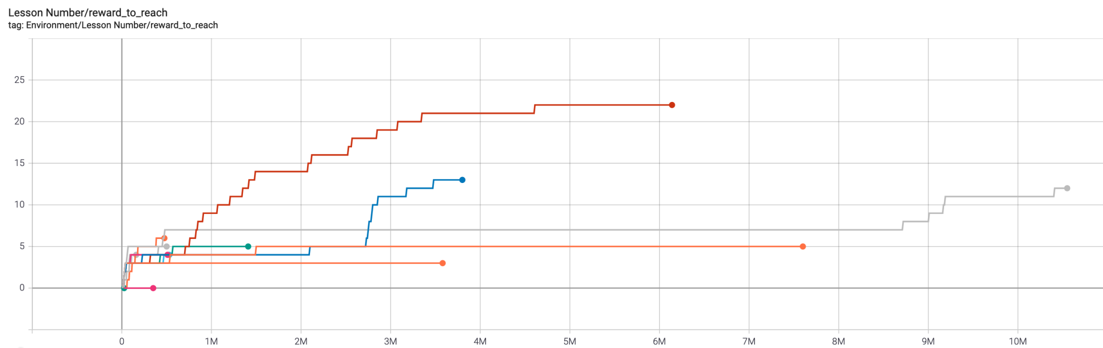


Figure 4.2: All training runs with curriculum lessons

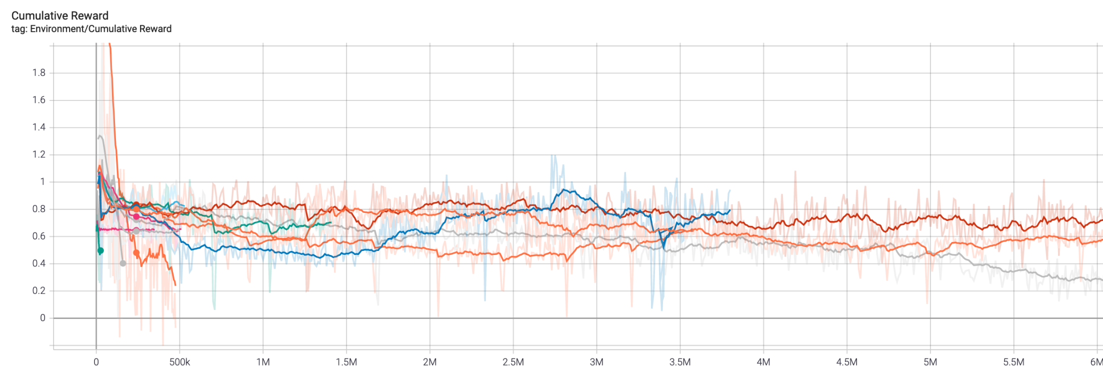


Figure 4.3: Reward progress on curriculum training runs
The shown graphs' colors are matching with the associated graphs' in Figure 4.2.

4.2.3 Stacked Observations

The next significant improvement was found when stacked observation vectors were introduced. This change was made on the agent implementation. Hence, the input for the neural networks increased, which makes previous trained models incompatible. Compatible versions can become important if a continuous training in future versions is planned, as they need to have the same observation and action space. In Figure 4.4 the red graph with 5x stacked observations significantly outperforms the other two training runs with 1x and 20x stacked observations. As explained in section 3.5 it is beneficial for the agent to know previous states to plan its next actions accordingly. As an example, gatherers are always cycling though the same three states. At first, they are *idle*, when being sent out they change to *in production*, and after that they are in *production ready*. All of these states are preserved in the observations. On the other hand side the training run with 20x stacked observations performed significantly worse. This shows that far distanced observations do not have a decisive impact on the game. The additional computational power needed to optimize the policy network for 20x stacked observations seems to outweigh the benefit of past observations. The reason that all three training runs managed to overcome the third lesson, was due the BC and the satisfactory performance of the random actions obtaining the needed amount of xp.

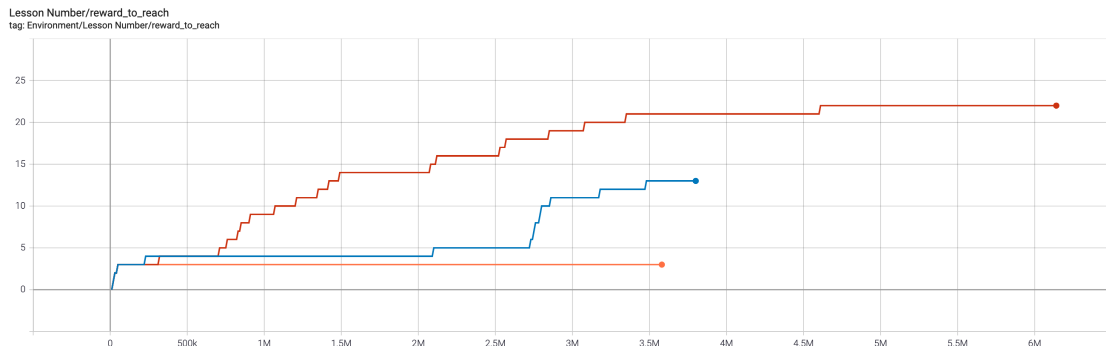


Figure 4.4: Curriculum learning with different amounts of stacked observations

The blue graph shows the training with a single observation space, the red graph has 5x stacked observations and the agent, indicated by the orange graph, was trained with 20x stacked observations.

In correlation to the higher lessons reached by the agent, the episode length increases (see Figure 4.5). The increasing episode length indicates that the agent starts to learn that not collecting xp leads to negative rewards and termination. It also shows that lessons with higher xp goals require the agent to do more training steps due to complex action sequences to receive more xp. With the training setup (see Listing 3.2) the agent can do 500 steps without reward before the training episode gets terminated with a negative reward. The lesson is terminated with a positive reward of one when the goal is reached.

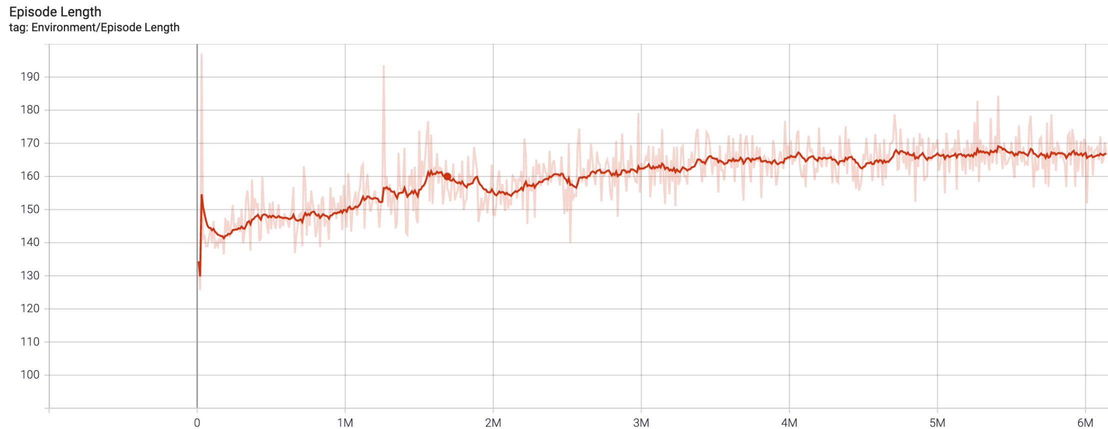


Figure 4.5: Increasing episode length over time

Each data point shows the mean episode length of 10,000 training steps. The training has a total of amount of 6.14 million steps.

4.2.4 Observed Agent Behavior

The most successful training progress was achieved with a combination of curriculum learning, Behavioral Cloning, Generative Adversarial Imitation Learning and Proximal Policy Optimization [1]. The hyperparameters for this training are shown in Listing 3.2. Including the curiosity module, has shown itself as less reliably to improve the overall reward. With the curiosity module in place the results showed a big variance in the cumulative reward, performing worse than without. Over all training steps the agents reward was defined by the current curriculum lesson which started at a value of 300 xp which the agent had to reach. The curriculum lesson would advance if the agent managed to continuously achieve a mean reward of 0.8 with at least 100 episodes. Each episode is equal to 2000 steps. The results of the curriculum learning progress can be seen in Figure 4.6. The behavioral cloning was limited to the first 150,000 steps, in which the agent tried to imitate the 2000 steps which were provided by the `DemoActionProvider`. From there on the training was continued with GAIL in combination with PPO.

After the agent finished its training, the python trainer exports the neural network model into a .onnx file. It is also possible to save checkpoints of the model during the training to compare how the agent evolves over time. Inside the Unity Editor, the agent can then be set up to use the trained model for its actions. By enabling the graphics again and setting the game speed back to one, it is then possible to analyze the agents' behavior. As expected due to the random instancing of the neural network, the agent starts with random actions at the beginning of the training. As most of the actions lead to receiving xp the agent is able to complete the first curriculum lessons quickly. At this point the agent is still in its BC training steps. Watching the agent, it can be observed that the majority of its soft currency is spent on scouting locations and construct new buildings, which can be seen in Figure 4.7. Another peculiarity is that the agent quickly stockpiles items in its storage. As soon as the storage is filled the agent has only a limited set of actions because it cannot collect any goods or other productions from its gatherers. The untrained agent stagnates from here on as it only disposes items occasionally, and directly fills the free space again with random goods. To improve, the agent should collect goods which can be spent on task or productions. After completing the training, the filled up storage is less of a concern. Most of the time, the agent is capable of keeping the storage space at roughly 50%. The agent achieves this by disposing items earlier and completing tasks.

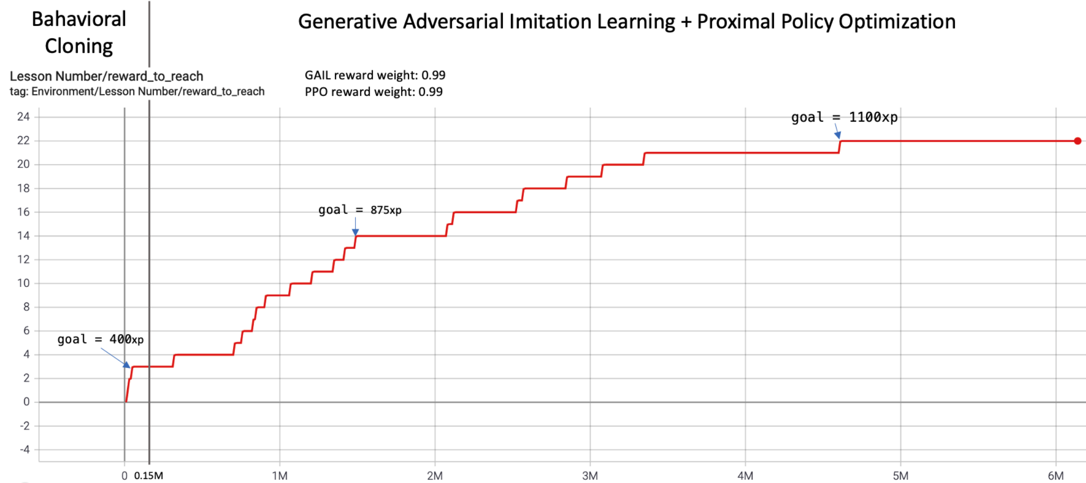


Figure 4.6: Curriculum lessons training progress
The graph shows that the curriculum goal of 400 xp is reached within the first 150 thousand steps of BC. As a reference further curriculum goals are marked for lesson 14 and 22. The intrinsic rewards of GAIL and the extrinsic reward of PPO had the same weight factor of 0.99. The highest curriculum lesson reached was 22 after 4.61 million steps. The next lessons goal would have been 1150 xp.

Figure 4.7: Agent at training start

At the beginning of the training the agent spends its soft currency randomly, which results in a mix of scouted gathering locations, buildings and expansions. In this image most buildings are still under construction marked with a progress bar. The buildings with yellow flags are ready to be unveiled. In the center of the image the loot feedback for fish productions is shown. The blue stars are indicating the xp received for the productions.

Figure 4.8: Trained agent at the beginning of its episode

Screenshot from a training agent. Expansions constructions are marked by red arrows and circles. The storage is filled with 36 out of 70 item slots. The agent also completed a task (top center) from which it received 25 xp, one hard currency and 25 soft currency (center).

Figure 4.8 shows an agent after 6 million training steps. The screenshot was taken after the first few steps of a training episode. A noticeable difference to the previous screenshot in Figure 4.7 is that the agent is building fewer buildings, but instead a lot of island expansions. Unlike buildings, island expansions do not unlock any further recipes or productions. Their designed purpose is to create more space on the island to enable the construction of more buildings and or decorations. The expansions are paid with soft currency and are rewarded with some xp in return. As the agent is continuously following the strategy of building expansions, this gives the hint that their return in xp could be too high. Game design confirms this hypothesis because expansions are not designed to be a primary xp source. In addition to the result of the agent, two internal game testers also reported that they feel that the expansion are too cheap (see Figure 4.9 and Figure 4.10). The manual tests were running over three days before coming to their results. The statement from Figure 4.10 also shows why too cheap expansions are disadvantageous for the playing experience. In conclusion, the balancing for expansions was adjusted in the following sprint. Although, soft currency costs for building expansions were reduced by 25% the xp share was reduced even more by 33%.

Although a retraining on the new balancing data has been executed, the agent was not developing a new strategy. With the reduced cost and xp reward, it still built as many expansions as possible. This could be due to the fact of the agent's starting preset, in which the expansions were still comparably cheap due to the low level. The price for each expansion is raising with each build, and a second resource is needed on higher levels. However, in the current training scenario the agent would not get to that point, as the episodes are restarted earlier. Besides this, the starting game state had a clear focus on the island activities which matched the action space of the agent very well. With this, the suggestion to increase the expansion cost still remains.

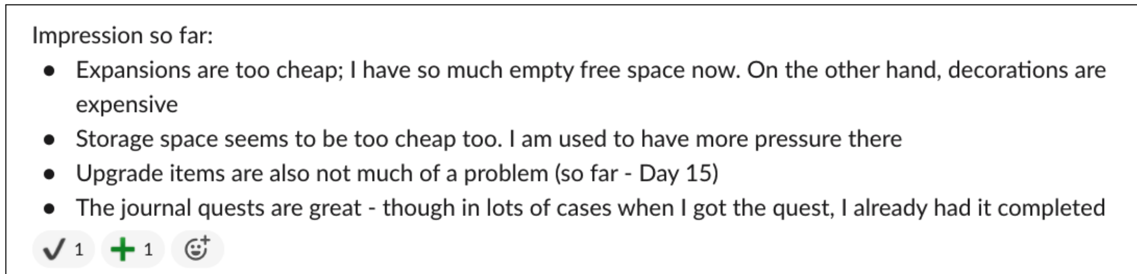


Figure 4.9: Player feedback to expansions

Along other balancing related feedback, a tester noticed that the expansions are comparably cheap. The green "+"-emoji at the bottom indicates that other testers agree with his feedback. The check mark indicates that changes regarding the feedback will be made.

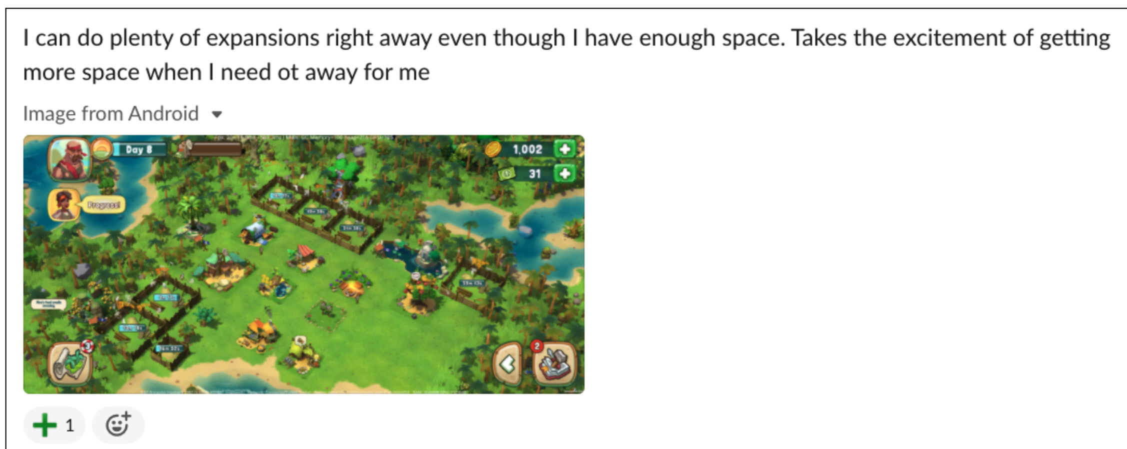


Figure 4.10: Player feedback to expansions

Another tester mentioning the cheap expansions together with a screenshot of his current game state. On the screenshot a lot of expansions are currently in the building process despite only a few buildings have been built on the island.

The executed actions were recorded by the `AdditiveStatsRecorder` which accumulated all actions during an episode. The results can be seen in Figure 4.11, but have to be observed with care as the graphs are showing the mean amount per episode. In general, the graphs should be analyzed in relation to each other and not in absolute values.

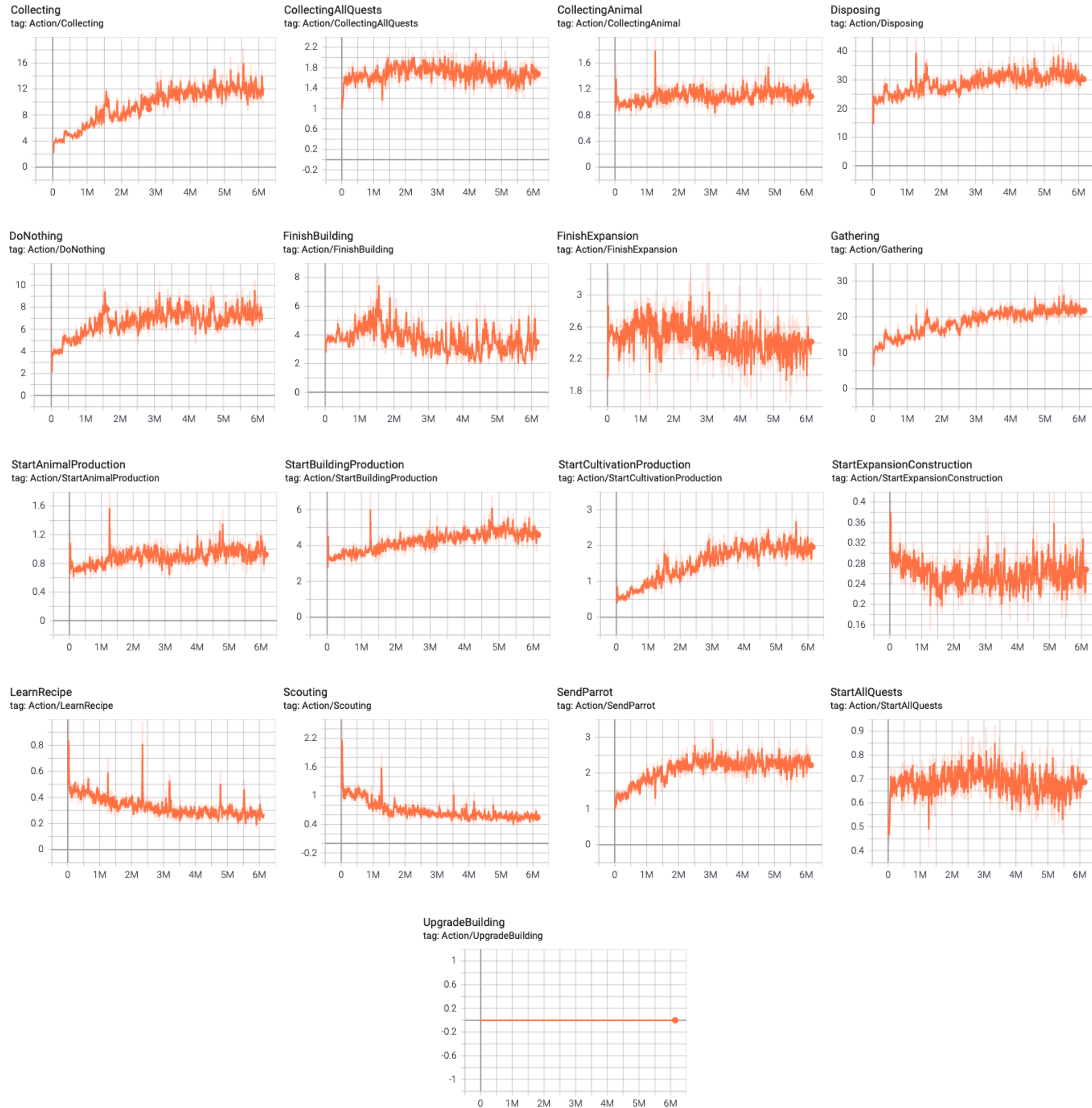


Figure 4.11: Relative amount of actions used by the training agents

Because the episode length is increasing (see Figure 4.5), it is expected that the total amount of actions is increasing as well. Still, it can be seen that the actions scouting and learn recipes are decreasing. Both actions cost soft currency and had no direct xp reward in return, but instead unlocked productions of new materials which were needed to upgrade buildings. In this particular run upgrade buildings had never been used because the resource conditions were never met. It shows that this agent did not advance enough to include this long term goal in its strategy as in

most cases other actions returned enough rewards. It is insured that this was not due to a bug as other training runs successfully upgraded buildings.

4.2.5 Difficulties

The `GameAgent` and its `AgentObservations` helper class relies heavily on preexisting data providers. During implementation, it became an issue that incorrect observation data led to wrongly masked actions. This often became visible through error messages by the backend responses or by the `AdditiveStatsRecorder` not receiving action recordings. As the goal was to find bugs with the help of ML this is partially a success, with the disadvantage to be forced to fix it. Another time issue expense during the implementation of this work was continuous crashes during the training run. With the agent restarting multiple times and exploring millions of possible states, it happened that the backend responses sometimes came later than expected. This presumably led to the agent not being able to complete the scene loading within time, which resulted in the communicator stopping the agent. Unfortunately the ML-Agents python trainers are not restarting a stopped agent and the complete training run will be terminated. The terminated training run is stopped gracefully and the so far trained policy network and the logs are saved. For these cases, a workaround was implemented which automatically starts the training via the *–resume* bash command. However, the restart of runs resulted in some cases in a reward anomaly at the beginning of the episodes (see Figure 4.12).

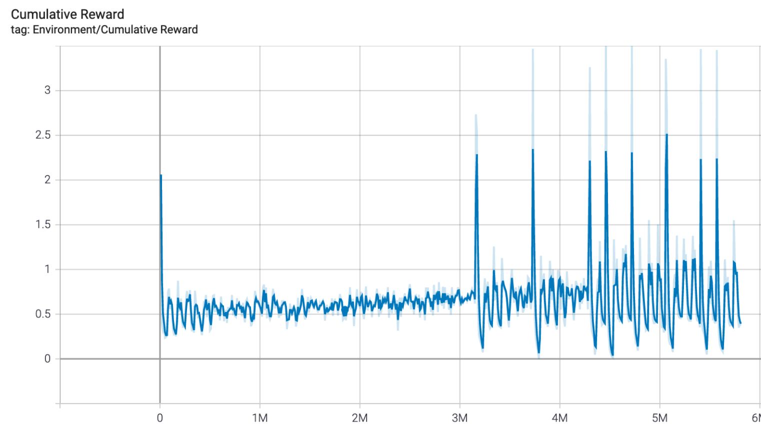


Figure 4.12: Anomaly on training starts and restarts

The training run was resumed multiple times. On each restart the reward spikes noticeably.

An additional time expense with unknown origin is that the training process has time gaps between the training steps. In Figure 4.13 gaps of multiple hours can be seen in between the agent steps evaluation. The gaps are also visible on the backend server which does not receive any requests in that time. The time gaps seem to be related to the *Lost Survivors* environment, as the *Hummingbird* example which was trained on a different machine, but with the same ML-Agents package version did not show these time gaps. By analyzing the timestamps, it is also ruled out that the pauses are related to the previously mentioned restarts. However, the time gaps seem not to affect the training results, as there is no missing data when analyzing on a step scale, as it can be seen in Figure 4.14. It is reasonable to presume that the network buffer of 500,000 steps is reached in that time, and the network optimization is calculated, even though the amount of pauses (16) does not entirely fit to the amount of trained steps ($\frac{6{,}000{,}000}{500{,}000} = 12$).

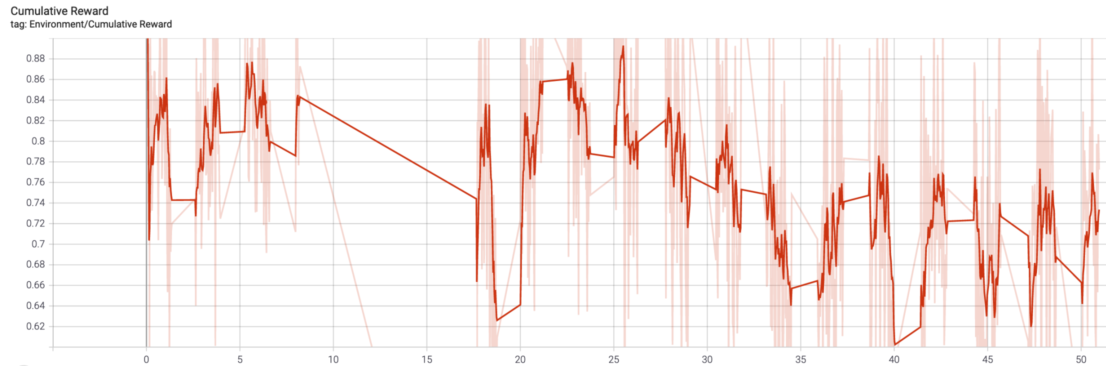


Figure 4.13: Unknown pauses in training progress

Cumulative reward in relation to training time in hours. The horizontal sections of the graph indicate pauses in which the training agents' did not execute any training steps. With 16 pauses in total most of them had a duration of approximately one hour. The graph was smoothed by a factor of 0.9. The original data points are displayed transparently in the background.

Figure 4.14: Cumulative reward by training steps

Analyzing the cumulative reward by training steps show no indication for training interruptions. The graph was smoothed by a factor of 0.9. The original data points are displayed transparently in the background.

Chapter 5

Conclusion

In this chapter concludes which values the use of machine learning algorithms brings to the development process. The hypothesis formulated in chapter 1, are discussed in their own sections. The sections are separated into the topics: *Game Design Suggestions*, *Quality Value*, *Generality* and *Training Time and Scaling*. All sections together answer the overall question how ML algorithms help to find bugs and unwanted game design flaws. Possible improvements and tools which can arise from this work are discussed in the very last section 5.5.

5.1 Game Design Suggestions

H1 The trained agent on *Lost Survivors* always finds an optimal way to play the game in regard to its reward function.

H2 Machine learning processes help the game design department to optimize balancing data.

For this work it is a great achievement that the agent was able to develop a strategy which aligned with the known flaws in the balancing data. The game designer and the testers, which reported that the expansions are too expensive, had no previous knowledge about the training results. With this the reliability on the agent strategies can be partially confirmed. On the other side, with the above discussed results it is fair to say that the agent was not able to find the most optimal way to play *Lost Survivors*. Rather then strategizing the collected goods, the agent gathered them seemingly just to gain some xp to avoid the episode termination. Because of the sub optimal gameplay of the agent, the strategy of the agent should be verified by a game designer. Still, it can be profitable to use a machine learning agent to save valuable time in manual testing.

5.2 Quality Value

H3 Machine learning processes help the QA department to find bugs.

As mentioned in chapter 5 the quality of the game code is crucial for the agent to work. The process of implementing an agent into the game not only uncovered unknown bugs but also showed a lot of bad smells, which could lead to technical difficulties in future development. One example is the current impossibility of displaying other players cities, which is based on the same issue to not being able to instantiate multiple playing agents in one application.

Once the agent is implemented, it produces a lot of player data which can be analyzed on multiple layers. To find the optimal policy, the agent is experiencing many different game states while training. This can be especially useful for migrating new game versions. Once a game is live, it is regularly updated. As the game logic is separated in front- and backend, the backend gets updated first to enable the migration from old game clients to the new ones. Even with automated tests it is nearly impossible to cover all game states. This is where the agents training process can help to test old and new clients against the backend migration. One way is to scan the agents' player logs for exceptions and other anomalies. Because the agent is usually training for more than a day, the log level has to be chosen carefully to not create multiple gigabyte of log files, which are tedious to check.

Another positive side effect was the `DemoActionProvider`. The player presets which are used to cheat a player to a certain game state are created by hand, and do not represent real player actions. As the checks can be easily modified, it is an additional tool to create or setup new test cases. It can also be used to try out strategies or use it as a benchmark for the agent.

Lastly, the custom designed statistic like the agent actions can also indicate if something is not working as expected. But not all bugs can be detected within the agents created data. One of the most important parts of the player experience are visuals and sounds. As this agent implementation was using the game internal services, sound and rendering were disabled to increase training speed.

5.3 Generality

H4 Machine learning can be generalized and added to any game project.

H5 Once the machine learning is implemented it is easily maintainable and further training can be performed.

The *Hummingbird* example and *Lost Survivors* had two things in common. Both games have a well-defined set of rules and a clear goal to strive for. Additionally, they have a limited simulated space with digital inputs. Both of these facts are in favor of deep reinforcement learning algorithms, as the environment is already given though the game itself and the reward function can be defined by the goal of the game. With both games developed with the Unity engine, the ML-Agents plugin was the ideal to transform the game into the learning environment. The setup is easy to implement and the same for any game. Even the training parameters have a default setup ready to go. Most of the implementation efforts need to be done within the environment to enable the agent actions, observations and the action masking. Still, there are some factors which are not feasible for every gaming environment and should be considered before starting the implementation of the agent. The models inputs and outputs are defined by the observations and agent actions. When training a model, the inputs and outputs have fixed connections in the neural network. It is possible to use a trained neural network as the initial state for deep reinforcement learning, but it is not possible to subsequently add or remove nodes. As an example in *Lost Survivors* the production actions were specific to a certain building. If during development a new building would be introduced, the model would not be compatible anymore and learned behavior would have to be retrained to introduce a new output action for the new building. With the drawback of longer training times, placeholders actions and observations can be introduced. The placeholder action would then be masked until they are introduced to the game. This however can be hard to predict as the complexity of a game in development constantly grows.

5.4 Training Time and Scaling

H6 Machine learning helps in the process of game development.

It is hard to say if the presented agent will find an optimal way with longer training times without changing the reward function or other training parameter. Even if the agent would eventually find an optimal policy in further training steps, it would not be feasible for a typical development process. As mentioned in section 2.2 the development cycle is separated into sprints which at *Lost Survivors* are two weeks long. For the results presented in this work the training took around eight hours to complete two million training steps. In comparison to the *Hummingbird* example runs with twenty agents in one scene, which reaches four million steps in only two hours. Assuming the agent is training overnight, together with the training setup and evaluation of the results, it takes two working days to complete the training. In practice this would limit the game designers to apply their balancing changes within the first sprint week to still be able to evaluate the results before the end of the sprint. On QA side it is not necessary to have an optimized agent as the training process itself can be monitored.

Latency is not only a problem during loading the game, but has also an effect on the training itself. With the latency between the backend server and the training agent, it is not possible to reward the agent for its actions immediately. Instead, the agent has to guess which of the previous chained commands led to its current result. However, as the value function (see section 2.3.4) is also a neural network and gets better over time, the agent should still be able to correlate the actions. Optimizing or a complete removal of the latency could still have significant effect on the training speed. One possible way would be to run the backend locally on the training machine. This was not possible for this work due to code confidential reasons. Another solution could be to fake all backend responses. This would be similar to the prediction system, which was removed to stabilize the training, but without the backend trying to re-sync the game state afterwards. An even faster possibility would be to train the agent on the backend itself using its request endpoints only. The downside of this approach is that it would require a lot of additional game state checks beforehand to assure the selected action is available. It would also isolate the agent further from the current development process. In the current state most of these checks are evaluated by the frontend services in the first place.

5.5 Future Work

Within the scope of this work a fully functional machine learning agent was implemented and trained with PPO, BC, GAIL and curriculum learning. Without being feature complete, the agent was able to detect basic playing strategies. With this result it can be said that machine learning can be a helpful tool when the infrastructure of the game allows for ease of implementation and convenient scaling. The agent was able to perform 194 possible actions based on its current game state represented by 10860 observations. One of this works' strengths is the amount of player data which is created by the agents training. With the huge amount of different game states, the testing process can be strengthened. The next steps from QA perspective would be to make the agent feature complete to explore even more states. This will be especially useful for cooperative game features where the agent can be influenced by other players. Here multiple agents could be trained using self-play, where the agent plays against its previous versions or with the currently developed MA-POCA (Multi-Agent POsthumous Credit Assignment) algorithm[1].

[1] MA-POCA was released at the end of the implementation for this work. While this work was written, a research paper has not yet been published, but more information can be found on `https://blog.unity.com/technology/ml-agents-v20-release-now-supports-training-complex-cooperative-behaviors`, Accessed: 06.06.2021.

From a game design perspective it is of interest to focus future training on simpler and more specific tasks to gain more knowledge about specific game features. The goal is to get closer to the optimal playing agent. Depending on the examined feature, some agent actions and observations could be removed to optimize training times. As most features rely on other actions, the agent would need to be combined with a cheat system or simple rule based actions, similar to the `DemoActionProvider`.

To further build on this work, a second neural network could be used to train with different balancing data. The game designer then just has to define a game state to achieve with certain conditions on the way. In terms of Reinforcement Learning, the conditions and the goal state then have to be translated into a reward function which will reward the agent for reaching the correct state or punish and reset the agent if it disregards the conditions. This would allow for better comparability between the agents.

Article References

[1] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[2] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. "Automatic detection of bad smells in code: An experimental assessment." In: *J. Object Technol.* 11.2 (2012), pp. 5–1.

[3] Tuomas Haarnoja et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 1861–1870.

[4] Noah Falstein. "The Flow Channel". In: *Game Developer Magazine: https://ubm-twvideo01.s3.amazonaws.com/o1/vault/GD_Mag_Archives/Game.Developer.2004.05.pdf* (2004).

[5] Mihaly Csikszentmihalyi. "Play and intrinsic rewards". In: *Flow and the foundations of positive psychology*. Springer, 2014, pp. 135–153.

[6] Jesse Schell. *The Art of Game Design: A book of lenses*. CRC press, 2008.

[7] Lars Konzack. "Computer Game Criticism: A Method for Computer Game Analysis." In: *CGDC Conf.* 2002.

[8] Daniel Galin. *Software quality assurance: from theory to implementation*. Pearson education, 2004.

[10] Mark Ehren. *Sony stoppt Videospiel Cyberpunk 2077*. https://www.tagesschau.de/wirtschaft/sony-cyberpunk-2077-101.html. Accessed: 06.06.2021. 2020.

[12] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. "CLEVER: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 153–164.

[13] OpenAI. *OpenAI Five*. https://blog.openai.com/openai-five/. Accessed: 06.06.2021. 2018.

[15] The AlphaStar team. *AlphaStar: Grandmaster level in StarCraft II using multi-agent reinforcement learning*. https://deepmind.com/blog/article/AlphaStar-Grandmaster-level-in-StarCraft-II-using-multi-agent-reinforcement-learning. Accessed: 06.06.2021. 2019.

[17] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. "Back to basics: Benchmarking canonical evolution strategies for playing atari". In: *arXiv preprint arXiv:1802.08842* (2018).

[18] Greg Brockman et al. "Openai gym". In: *arXiv preprint arXiv:1606.01540* (2016).

[19] Joel Lehman et al. "The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities". In: *Artificial Life* 26.2 (2020), pp. 274–306.

[20] Peter Krčah. "Towards efficient evolutionary design of autonomous robots". In: *International Conference on Evolvable Systems*. Springer. 2008, pp. 153–164.

[21] Victoria Krakovna et al. "Specification gaming: the flip side of AI ingenuity". In: *DeepMind Blog* (2020).

[22] unknown. *Specification gaming examples in AI - master list : Sheet1*. https://docs.google.com/spreadsheets/d/e/2PACX-1vRPiprOaC3HsCf5Tuum8bRfzYUiKLRqJmbOoC-32JorNdfyTiRRsR7Ea5eWtvsWzuxo8bjOxCG84dAg/pubhtml. Accessed: 06.06.2021. 2020.

[23] Arthur Juliani et al. "Unity: A general platform for intelligent agents". In: *arXiv preprint arXiv:1809.02627* (2018).

[24] Unity Technologies. *Example Learning Environments, Pyramids*. https://github.com/Unity-Technologies/ml-agents/blob/release_1/docs/Learning-Environment-Examples.md#pyramids. Accessed: 06.06.2021. 2020.

[25] Unity Technologies. *GameTune*. https://unity.com/products/gametune. Accessed: 06.06.2021. 2020.

[26] InnoGames GmbH. *InnoGames*. https://www.innogames.com/de/. Accessed: 06.06.2021. 2003.

[29] RA Bartle, Clubs Hearts, and Spades Diamonds. "Players who suit MUDs, 1996". In: *Saatavissa: http://mud.co.uk/richard/hcds.htm* (1996).

[30] John Schulman et al. *Proximal Policy Optimization*. https://openai.com/blog/openai-baselines-ppo/. Accessed: 06.06.2021. 2017.

[31] Adam Kelly. *ML-Agents: Hummingbirds*. https://learn.unity.com/course/ml-agents-hummingbirds. Accessed: 06.06.2021. 2020.

[32] Richard M Ryan and Edward L Deci. "Intrinsic and extrinsic motivations: Classic definitions and new directions". In: *Contemporary educational psychology* 25.1 (2000), pp. 54–67.

[33] Marvin Minsky. "Steps toward artificial intelligence". In: *Proceedings of the IRE* 49.1 (1961), pp. 8–30.

[34] Tom M Mitchell et al. "Machine learning". In: (1997).

[35] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[36] John Schulman et al. "Trust region policy optimization". In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897.

[37] Jeffrey L Elman. "Learning and development in neural networks: The importance of starting small". In: *Cognition* 48.1 (1993), pp. 71–99.

[38] Douglas LT Rohde and David C Plaut. "Language acquisition in the absence of explicit negative evidence: How important is starting small?" In: *Cognition* 72.1 (1999), pp. 67–109.

[39] Yoshua Bengio et al. "Curriculum learning". In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 41–48.

[40] Deepak Pathak et al. "Curiosity-driven exploration by self-supervised prediction". In: *International Conference on Machine Learning*. PMLR. 2017, pp. 2778–2787.

[41] Yuri Burda et al. "Large-Scale Study of Curiosity-Driven Learning". In: *ICLR*. 2019.

[42] Arthur Juliani. *Solving sparse-reward tasks with Curiosity*. https://blogs.unity3d.com/2018/06/26/solving-sparse-reward-tasks-with-curiosity/. Accessed: 06.06.2021. 2020.

[43] Alessandro Giusti et al. “A machine learning approach to visual perception of forest trails for mobile robots”. In: *IEEE Robotics and Automation Letters* 1.2 (2015), pp. 661–667.

[44] Mariusz Bojarski et al. “End to end learning for self-driving cars”. In: *arXiv preprint arXiv:1604.07316* (2016).

[45] Scott Niekum et al. “Learning grounded finite-state representations from unstructured demonstrations”. In: *The International Journal of Robotics Research* 34.2 (2015), pp. 131–157.

[46] Eduardo Morales and Claude Sammut. “Learning to fly by combining reinforcement learning with behavioural cloning”. In: Jan. 2004. DOI: 10.1145/1015330.1015384.

[47] Jonathan Ho and Stefano Ermon. “Generative adversarial imitation learning”. In: *arXiv preprint arXiv:1606.03476* (2016).

[48] Ilya Kostrikov et al. “Discriminator-actor-critic: Addressing sample inefficiency and reward bias in adversarial imitation learning”. In: *arXiv preprint arXiv:1809.02925* (2018).

[49] Unity Technologies. *Unity Real-Time Development Platform*. Microsoft Windows, Mac OS, Linux, https://unity.com/. Accessed: 06.06.2021. 2020.

[51] Oriol Vinyals et al. “Starcraft ii: A new challenge for reinforcement learning”. In: *arXiv preprint arXiv:1708.04782* (2017).

[52] David Silver et al. “Mastering chess and shogi by self-play with a general reinforcement learning algorithm”. In: *arXiv preprint arXiv:1712.01815* (2017).

Game References

[9] CD Projekt RED. *Cyberpunk 2077*. Microsoft Windows, PlayStation 4, Xbox One, GeForce Now, Google Stadia. 2020.

[11] Bethesda Game Studios. *Fallout 76*. Microsoft Windows, PlayStation 4, Xbox One. 2018.

[14] Valve Corporation. *Dota 2*. Windows, macOS, Linux. 2013.

[16] Blizzard Entertainment. *StarCraft II*. Windows, macOS. 2010.

[27] Playrix. *Township*. iOS, Android. 2012.

[28] InnoGames GmbH. *Lost Survivors*. iOS, Android. 2021.

[50] Nintendo. *Super Mario Bros.* Famicom, Nintendo Entertainment System. 1985.

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,
Name: Hartmann
Vorname: Julius Malte
dass ich die vorliegende Masterarbeit mit dem Thema:

The use of machine learning algorithms in the process of game development to find bugs and unwanted game design flaws

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Hamburg, 21.06.2021

Ort, Datum   Unterschrift