



# Reinforcement Learning in Unity

Field of studies:	Computer-Science
Author:	Werthemann Sebastian
Supervisor:	Prof. Dr. Graf Erik
Experts:	Dr. Flueckiger Federico
Date:	10.06.2020

# Contents

1	Introduction	3
1.1	Motivation and Goals	3
1.2	Project Status after the Project 2 Module	3
1.3	Project Overview	4
1.3.1	Baseline Establishment	4
1.3.2	Agent and Training Optimisation	5
1.3.3	Game Mode Exploration	5
1.4	Overview Reinforcement Learning	5
2	Getting Started with Reinforcement Learning in Unity	6
2.1	Reinforcement Learning with ML-Agents	6
2.2	ML-Agents Setup	6
2.3	ML-Agents Demos	6
2.4	Integration of Reinforcement Learning into the Game Prototype	7
2.5	First Training with the Bachelor Thesis Reinforcement Learning Project	8
3	Understanding the Machine Learning Environment	9
3.1	Committing a Training	9
3.1.1	Machine Learning Configuration	10
3.1.2	Executing a Training Run	11
3.2	Components within Unity	12
3.2.1	Behaviour Parameters	12
3.2.2	Decision Requester	14
3.2.3	The Agent Scripts	15
4	The Game Environment	19
4.1	The Arena	19
4.2	The Characters	20
4.3	The Gym	21
5	Training and Improving the AI	22
5.1	RunBot	22
5.2	Attacking a Stale Target	23
5.3	Self-Play	23
5.4	Improving the Reward Distribution	27
6	Game Mode Exploration	28
6.1	Training N Versus N Agents	28
6.2	Agent Variations	30
6.3	Obstacles	32
6.4	Spawn Zones	35
6.5	Healer Agent	36
7	Results	38
8	Excuse: Open AI Five	39
9	Conclusion	40
10	Sources	41
10.1	Web sources	41
10.2	Literature	42
10.3	Tutorials	42
10.4	Figures	42
10.5	Images	43
11	Declaration of Originality	45

# 1 Introduction

## 1.1 Motivation and Goals

Having chosen the data engineering specialisation mainly with the machine learning parts of the modules in mind, being quite keen on learning more about artificial intelligence in general and having always dreamt about developing a video game formed the motivation for this work.

The primary goal of the bachelor thesis was to apply machine learning to the video game prototype established during the project 2 module, by replacing the previously scripted behaviour of computer-controlled entities with behaviour determined through artificial intelligence.

During this report, a distinction between two types of computer-controlled entities will be made:

- Bots are controlled by a bot behaviour script, which logically tells them what actions to take given a certain situation.
- Agents are controlled by an agent behaviour script that enables artificial intelligence to take control either during training or gameplay.

Work on the agents would include tackling various machine learning questions and issues, such as bias, hyperparameter balancing and improving the AI's robustness by, for example, exposing it to various training environments either during training or gameplay to see how it handles situations it has not been trained for specifically.

A further goal was to find out about certain limits, either given by the possibilities of the training algorithms, the training environment, computing power and/or time.

## 1.2 Project Status after the Project 2 Module

The result of the project 2 module was a game prototype – a simple environment with a few entities that could face off against each other.

The player was able to choose one of three characters, comprising of a healer and two damage dealers, while the remaining two were controlled by bots whose behaviour was defined in a corresponding script. The player was then able to switch between these characters while playing.

Aside from that, there were three types of enemy characters the player could fight against. These bots were logically controlled by a rather simple script as well.

The primary focus of the project was to get started with game development in unity and defining the game's mechanics, such as interaction of characters with the environment and among each other. This was achieved through a set of abilities that had to be created and customized by playing around with particle systems and various features unity offers such as ray casts, overlap circles, colliders, triggers, animators and renderers.

### 1.3 Project Overview

At the start of the project, it was hard to estimate how difficult it would be to integrate AI into the existing project and how long it would take to do so. Hence an initial decision was made to be rather conservative about adding too many goals and not include additional extensions to the game. These could have included the expansion of the game world, the addition of a storyline or further game mechanics.

The scope of the thesis was thus set to only comprise of the addition and continuous improvement of the AI elements, which in turn allowed for a clear focus on the matter.

Overall project planning was time-boxed based on the official time frame provided by the bachelor thesis regulations at BFH. To mitigate the challenge of estimating the difficulty of implementing the AI integration, and the uncertainty of training success, the project plan was based on the following sequential phases and their respective milestones.

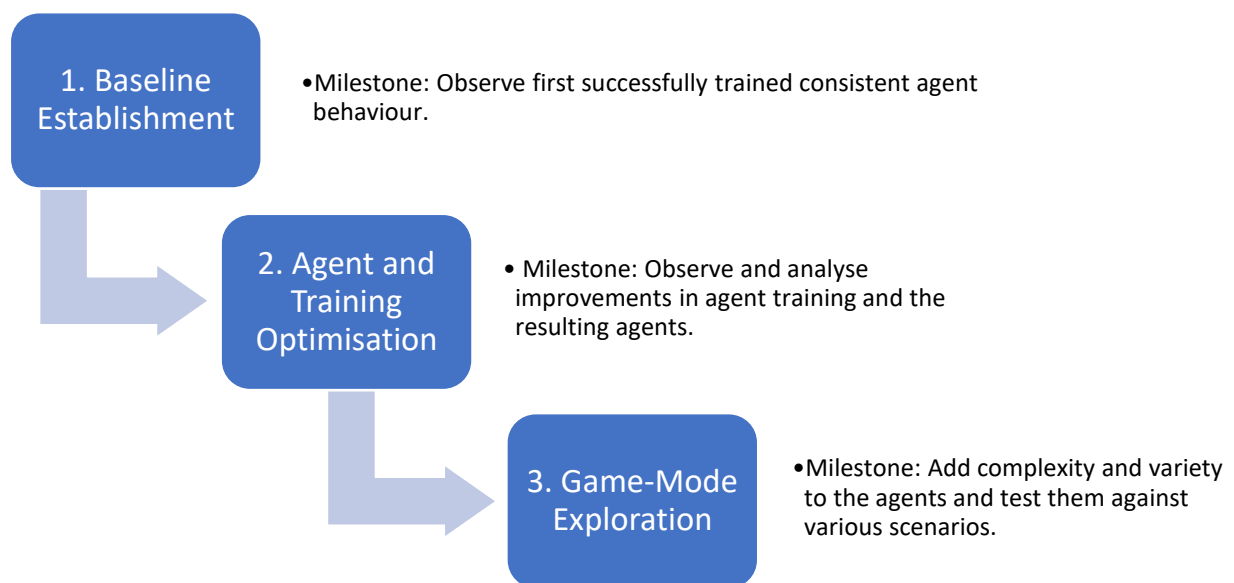


Figure 1: Project Overview

Subsequently an overview of the different phases will be given.

#### 1.3.1 Baseline Establishment

The goal of the initial phase consisted of the following main tasks:

- Simplification of the existing environment
- Integration of a way to train AI through the ML-Agents library
- Launching a first training inside the Bachelor Thesis Reinforcement Learning environment

The first task's definition of done consists of the ability to start training an agent in an arena. As a strategy to reduce risk and potentially speed up the conclusion of this first milestone, it was decided to simplify the initial training setup as much as possible. Section 2 provides an overview of the necessary steps taken for task 1.

Sections 2, 3, and 4 provide an overview of the necessary steps that were taken to achieve the goal of task number 2. In addition, these sections provide detailed information about the project setup and the approach taken for the integration and training.

### 1.3.2 Agent and Training Optimisation

The optimisation of the training consisted of the following steps:

- Getting a first agent to complete a simple task
- Enabling the agent to replace a bot, meaning that it may move around and use a set of assigned abilities to attack
- Training the agents against an enemy that fights back
- Improving the agents by working on the training hyperparameters and the agent script to adapt its rewards, reset mechanism and observation and action space
- Increasing training efficiency by having multiple arena instances
- Improving the reward system to allow for rewards outside the agent script

The evaluation was initially done qualitatively. As for quantitative evaluation, the training time, steps taken, and game mode related metrics were considered.

Section five provides details regarding the optimisation process.

### 1.3.3 Game Mode Exploration

Upon completion of the previous two, the last milestone focused on applying the earned knowledge and established training environment to increase the complexity of the game and implementing a set of agents to it, while evaluating the capabilities of those agents.

Within the context of this work, the following aspects were targeted:

- Having more than two agents in one arena (n versus n)
- Adding obstacles in the form of trees
- Letting the agents spawn in predefined spawn-zones
- Implementing a variety of agents with different roles

Section six reports on the work and observations of this third and final phase.

## 1.4 Overview Reinforcement Learning

In reinforcement learning, a type of machine learning, an AI must find out, which of the actions, available to it on a step by step basis, result in the maximum reward.

Rewards may be either positive or negative (penalties) and are awarded to the agent, controlled by the AI, when its actions cause it or the environment to reach certain states defined by the designer.

For the AI, learning how to optimise reward-gain is a process of trial and error. The agent's policy, randomised or set to default values at start, is updated – like a weight vector – to allow for a better prediction of what actions are most likely to be rewarded in the near future. To do so, it is given a certain state of the environment in the form of an observation vector.

The algorithm primarily used to update the agents' policies during this thesis is called Proximal Policy Optimisation algorithm (PPO). [1]

Deciding how to structure the reward-system is a crucial part; while having only a few rewards that are directly connected to the goals of the environment gives the AI the freedom to reach them on its own terms, the sequence of actions required to get there may be too complicated to be discoverable without having unlimited time. With this in mind, it is also possible to define rewards that help guide the decision-making process.

While they may be needed for an agent to reach a certain goal, they may also limit its exploration process, meaning that an agent is less likely to go down a path with no short-term rewards, or penalties even, in order to reach a higher reward in the end. The agent might hence learn to go down a path that its human designer considers best for it, leaving out the opportunity to find a better path that was not thought of. [2]

## 2 Getting Started with Reinforcement Learning in Unity

### 2.1 Reinforcement Learning with ML-Agents

A few years ago, a friend introduced me to a simple project of his, which used TensorFlow to train an AI to control a bird in the flappy bird game.

The game's objective is to navigate a bird in a way that it does not crash into any pipes that, aside from a single opening, connect the ceiling with the floor. (Picture below)

Hitting one of those pipes would lead to a game over, or, in the case of the AI training to the demise of one of the many bird instances simulated in the game simultaneously. [3]

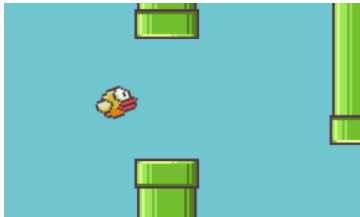


Image 1: Flappy Bird

The question of how to get started with applying machine learning to the project, was hence quite quickly answered by looking for a way to integrate TensorFlow into Unity – and then learning about a project called ML-Agents, which was designed to do just that.

The following sub-sections provide an overview of the required steps for setting up ML-Agents, and for getting started based on the available demos.

### 2.2 ML-Agents Setup

The first step taken after stumbling upon Unity's ML-Agents repository, was to install it on the machine that would be used during the work on the thesis.

With ML-Agents, this turned out to be quite challenging – it took quite some time to get all the dependencies figured out and to get started with the available demo-package.

While the unity version chosen for the project (2019.3.0b3) was not an issue, the versions of TensorFlow, Python and ML-Agents itself however caused quite some complications until finding a stable combination evolving around the latest version of ML-Agents (0.14.0), TensorFlow 2.0.1 and a slightly older version of Python (3.7). [4] [5]

### 2.3 ML-Agents Demos

Once a working environment was established, it became possible to launch a demo project contained in the ML-Agents folder drawn from the GitHub repository, by adding it to the Unity Hub, which is Unity's project manager, coming with a new installation of Unity. [6]

The project itself contained quite a few demo scenes showing off various features of the ML-Agents library. These scenes all contained pre-trained brains in the form of ".nn" files. The term "Brain" serves as another way to refer to the one used in machine learning: "model".

These decision models which are the main result of the trainings, would take over control of whatever agent they had been assigned to. – It was, however, more interesting to take a few projects and train the brains from scratch, by running a simple shell command and then hitting the play button in the unity environment. [7] [8] [9]

Another demo could be added by following a tutorial on the ML-Agents repository: The RollerBall. It had an agent control a sphere by moving it around a small platform to find a reward box, resetting whenever it fell off the platform. [10]

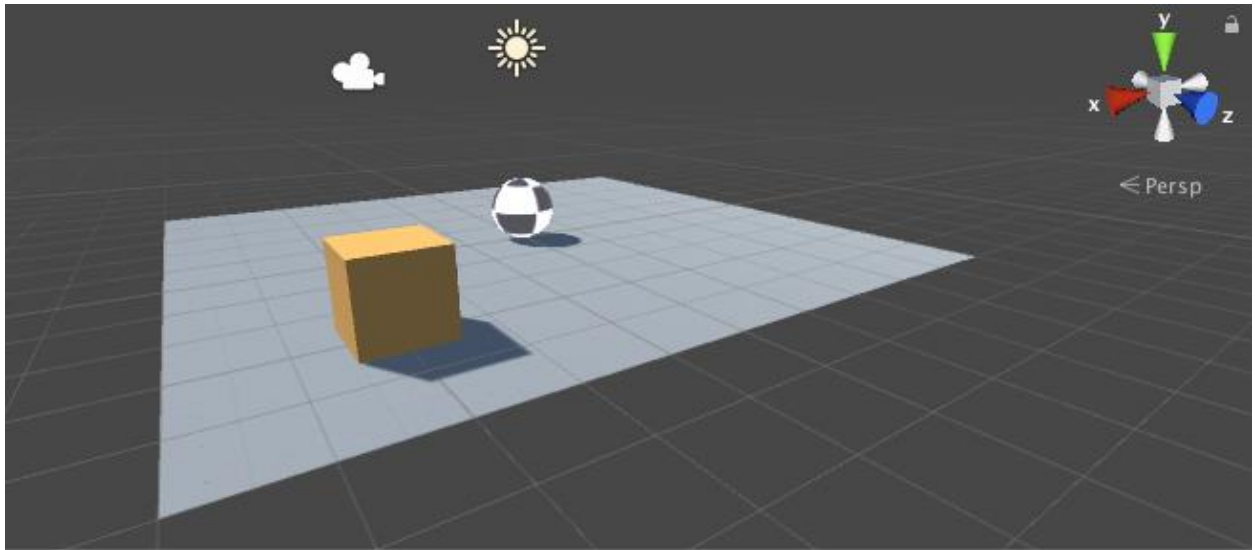


Image 2: Roller Ball

The image above shows the RollerBall – the box serves as the goal, resetting whenever the ball, which is controlled by the AI, hits it. The agent, controlling the ball object, simply observes its current movement speed and the three-dimensional coordinates of both itself and the box. Based on those observations, the brain decides whether it should apply positive or negative force horizontally and or vertically.

Most of the demos were quite simple, further, to conduct a training, one could define certain values in the `trainer_config.yaml` file, which had already been taken care of for the demos. Aside from the default values, a set of values came predefined for each specific behaviour, such as the RollerBall behaviour (more on that in section three), which actually made use of the default values. Because of this, seeing most demos came with more or less optimised values, the trainings were quite fast, and the agents learned quite quickly. Nevertheless, it was quite interesting to play around with the various demos, especially because it was a way to learn about how everything fits together. [9]

## 2.4 Integration of Reinforcement Learning into the Game Prototype

The first step towards the integration of machine learning into the project, was to simplify the game prototype wherever possible. – Aside from a simple bot that was kept to eventually let the AI train against, all other bots were removed from the game environment. Further, most abilities and prefabs were taken out and the camera was set to be static on an arena, rather than following a player. In addition, a lot of logic revolving around determining whether a character is controlled by the player or not or to tell the bots who they should fight against, was no longer required.

Aside from that, the process was quite straight forward – after importing the ML-Agents package, it was possible to have a look at the RollerBall example and add the required scripts to the bot left in the project environment, replacing the BotBehaviour script attached to it with the first agent script; the PlayerAgent script.

## **2.5 First Training with the Bachelor Thesis Reinforcement Learning Project**

To get everything working, it was important to start out with a very simple agent that would basically do the same as the agent seen in the RollerBall example – an agent that would, instead of rolling, move to a target location and reset with a positive reward when finding it.

Instead of resetting when falling of the platform, the so called RunBot would reset when walking too far away from the goal, triggering a negative reward – this was necessary for it not to be stuck running against a wall for too long. More detail on the components in play and the RunBot will be given in chapters three, four and five.



### 3 Understanding the Machine Learning Environment

The following section aims at elaborating the machine learning environment, the components it consists of, the role of these components, how they may be adapted and work together and how changes to them affect the trainings and their resulting models.

Further, this section also includes knowledge earned over the term working on the thesis project to elaborate on important aspects of the training such as action and observation vector sizes, training time and computing power.

#### Unity

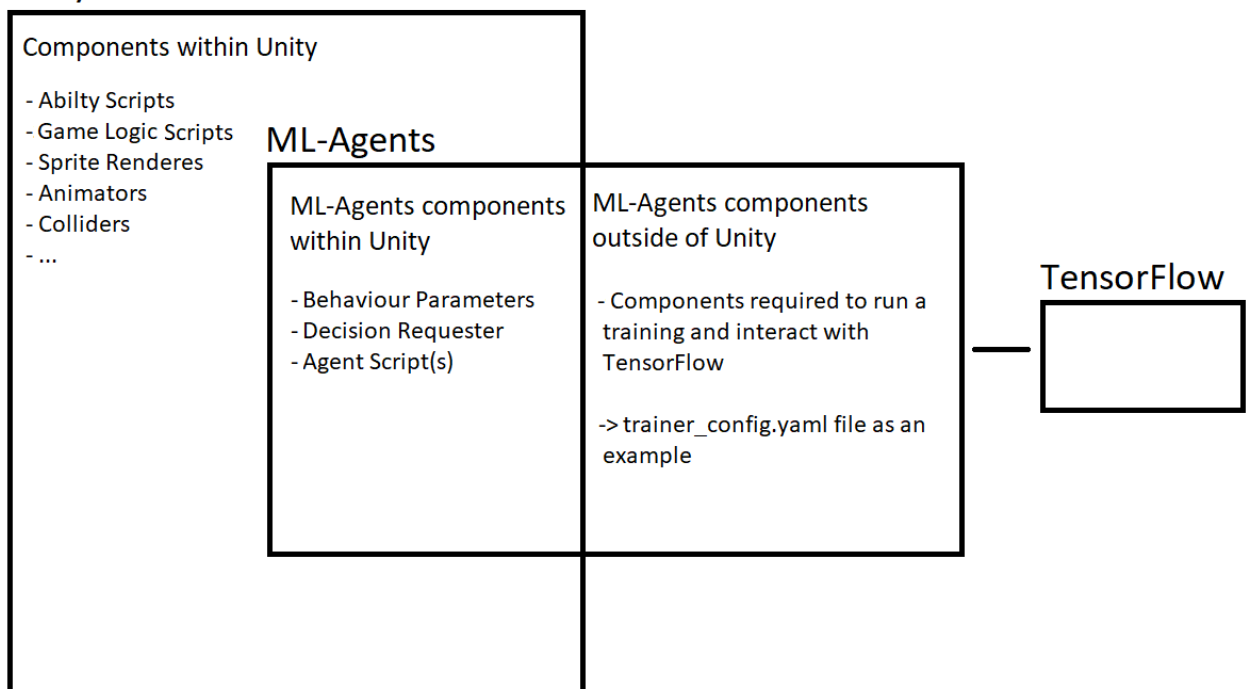
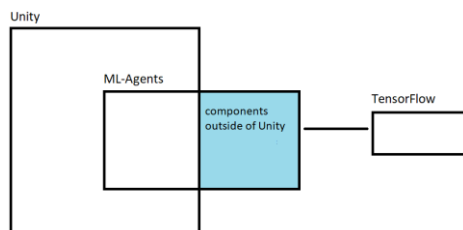


Figure 2: Components Overview

The figure above shows a rough overview of the components covered in sections three and four and where they are to be placed relative to the Unity environment.

While Unity serves as the environment to train the artificial intelligence in and TensorFlow as the toolkit used to train it with, ML-Agents may be seen as the connection between the two. [11] [4]

#### 3.1 Committing a Training



The actual execution of a machine learning training consists of the following two main steps:

1. The configuration of the training via the trainer\_config.yaml file
2. The execution of the training via command line

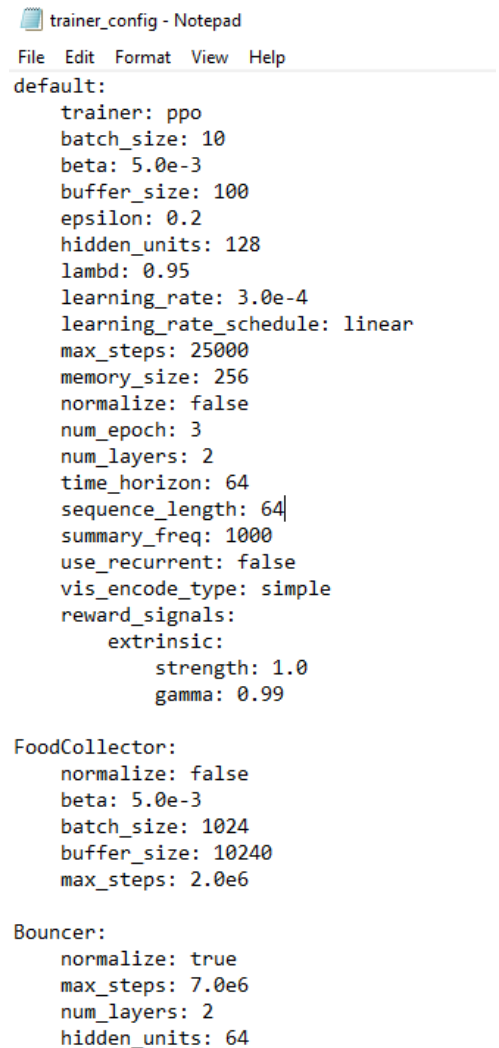
As the graphic above shows, these steps are executed within the part of ML-Agents outside of Unity.

### 3.1.1 Machine Learning Configuration

The trainer configuration file, which has to be specified in the training command enables the specification of certain values, such as hyperparameters like the learning rate, batch and buffer sizes to be used during the training, training length defined by the `max_steps` parameter and other parameters such as the strength (weight) of the reward signals the agent will receive while performing actions.

The picture to the right is a screenshot of the top of the default `trainer_config` file found in the ML-Agents package drawn from the repository. It depicts the default settings to be used for all behaviours, whose name does not appear in the file, such as “FoodCollector” or “Bouncer”. [8] [12] [13]

Additional entries not seen in the picture are possible, such as algorithm specific values, or values that enable self-play, which will be covered in the chapter describing the various trainings covered during the time spent on the project. [13] The web source cited here shows the entire list of roughly fifty available parameters.



```
trainer_config - Notepad
File Edit Format View Help
default:
  trainer: ppo
  batch_size: 10
  beta: 5.0e-3
  buffer_size: 100
  epsilon: 0.2
  hidden_units: 128
  lambda: 0.95
  learning_rate: 3.0e-4
  learning_rate_schedule: linear
  max_steps: 25000
  memory_size: 256
  normalize: false
  num_epoch: 3
  num_layers: 2
  time_horizon: 64
  sequence_length: 64
  summary_freq: 1000
  use_recurrent: false
  vis_encode_type: simple
  reward_signals:
    extrinsic:
      strength: 1.0
      gamma: 0.99

FoodCollector:
  normalize: false
  beta: 5.0e-3
  batch_size: 1024
  buffer_size: 10240
  max_steps: 2.0e6

Bouncer:
  normalize: true
  max_steps: 7.0e6
  num_layers: 2
  hidden_units: 64
```

Image 3: Trainer configuration file

### 3.1.2 Executing a Training Run

To run a training on windows, the following command may be entered in the command prompt:  
< mlagents-learn "path to trainer\_config.yaml"--run-id="name of training" --train >

This, if everything goes well, will show the unity logo as shown in the image below and initialise all elements necessary for the training. [12] The training then starts as soon as the play button in the Unity editor is pressed.



```
Command Prompt - mlagents-learn C:\Users\Shadowlink\Desktop\ml-agents-latest_release\config\trainer_config.yaml --run-id=cancel --train
Microsoft Windows [Version 10.0.18363.836]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Shadowlink>mlagents-learn C:\Users\Shadowlink\Desktop\ml-agents-latest_release\config\trainer_config.yaml --run-id=cancel
--train
WARNING:tensorflow:From c:\users\shadowlink\appdata\local\programs\python\python37\lib\site-packages\tensorflow_core\python\compat\
v2_compat.py:65: disable_resource_variables (from tensorflow.python.ops.variable_scope) is deprecated and will be removed in a futu
re version.
Instructions for updating:
non-resource variables are not supported in the long term

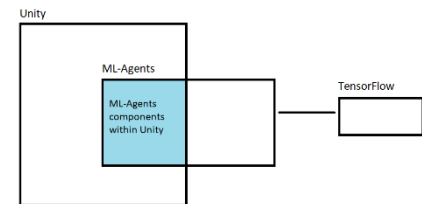
Version information:
ml-agents: 0.14.0,
ml-agents-envs: 0.14.0,
Communicator API: API-14,
TensorFlow: 2.0.1
WARNING:tensorflow:From c:\users\shadowlink\appdata\local\programs\python\python37\lib\site-packages\tensorflow_core\python\compat\
v2_compat.py:65: disable_resource_variables (from tensorflow.python.ops.variable_scope) is deprecated and will be removed in a futu
re version.
Instructions for updating:
non-resource variables are not supported in the long term
INFO:mlagents_envs:listening on port 5004. Start training by pressing the Play button in the Unity Editor.
```

Image 4: ML-Agents training command

## 3.2 Components within Unity

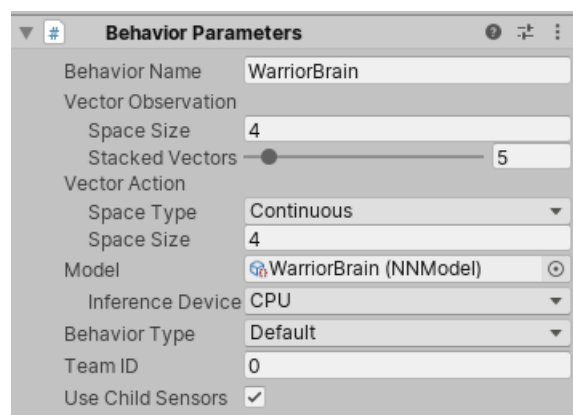
In the following sections, the components within Unity that are involved in the machine learning process are introduced.

The components in unity all have to be attached to a so called GameObject, an empty container for various scripts such as a MovementController, HealthController, FireBolt or ChargeAttack and further components such as a rigid body, colliders, sprite renderers, animators etc.



The ones directly required for the machine learning using ML-Agents, explained in this section, are the BehaviorParameters, DecisionRequester and a user-defined Agent script such as WarriorAgent. To add one of these components, it is possible to hit a button in the inspector window after clicking on a game object in the hierarchy. [14]

### 3.2.1 Behaviour Parameters



The image on the left shows the configurable values of the BehaviourParameters script as it shows up in the Unity editor. The script itself belongs to the ML-Agents package, meaning that it was not required to be implemented.

The right configuration is quite straight forward, but nonetheless very important seeing it contains crucial values such as the action and observation vector size.

Models that result from a training have to be added to the model-field in order for them to be able to take control of the agents.

Image 5: Behaviour Parameters in the Unity inspector

The following subsections explain the different values seen in this picture and aim to elaborate why their configuration is crucial.

#### 3.2.1.1 Behaviour Name

As a small note, the people behind ML-Agents refer to “behaviour” with the American-English term “behavior”, whereas generally in Unity, the British-English term “behaviour” is used – such as with almost all Unity scripts that extend from a class called MonoBehaviour - Seeing that the two terms do not serve any distinguishing purpose, references to the term shall be consistently taken with the British variant.

The first entry, the behaviour name, is, aside from naming purposes, mainly important to be able to distinguish what values from the trainer\_config should be loaded. The output of the training then generates a model.nn file for each different behaviour used during training. Later on, multiple agents were trained simultaneously, while identical agents during self-play for example, all contribute to the same model. Letting two warriors train versus two mages, would hence result in one WarriorBrain.nn and one MagicianBrain.nn file within the output folder named after the training name defined when starting the training through the cmd. [15]

### 3.2.1.2 Observation and Action Vector

The observation and action vectors are what the training algorithm works with, they correspond to what the AI, the agent, is able to perceive and what actions it is able to perform. While the observation vector values may be any value – a coordinate in the form of (x,y,z) would have to be split into three observation vector entries – the action vector always returns a value between -1 and 1, which may be used to define a certain action in the agent script.

Each step, which corresponds to each frame, the vector is collected and based on past rewards, the action most probably leading to an increased reward is taken. This can basically be imagined as a sample with a number of features corresponding to the observation vector size. The algorithm returns a prediction (the action vector) based on its policy, which can be understood as a weight vector of other machine learning tasks. [15]

### 3.2.1.3 Vector Sizes

For a brain to be compatible with an agent, both vector sizes must match the ones defined in the inspector, else the vectors will either be clipped or too large, resulting in an unused entry in the vector.

This is even more important if a training has been completed and the resulting model is used to control the agents – if the vector size changes, after updating the agent script for example, it becomes very risky to use the model.

If, for example, a model was trained with an observation and action vector size of four and one of the four observations is then deemed obsolete and removed, resulting in a size of three, the model, trained with four entries will then be missing one observation that was required to generate a certain output in the form of an action.

If the sources that generate the input value for the observation and action vector change, the agent will stop working as intended certainly – it will behave randomly. To give an example, this would happen (even with unchanged vector sizes) if observation[0] corresponded to the enemy's x axis position was changed to be the value of its remaining health.

Given that the actual sources that generate the input value for the observation and action vector remain the same and if the missing observation is the last input, the agent may still work almost properly, but if, for example the second observation is removed, the third observation will be perceived as the second, the fourth as the third and the input required for the fourth sensor will be missing, leading again to "random" behaviour. Increasing the observation size to five, will only result in vector clipping, however, which does not cause a big issue.

As for the action vector, a certain set of observations will trigger a certain action each step, so if for example, the model decides at point x in time that action[3] should be set to 0.5, corresponding in a boolean yes to shooting a fire bolt at the closest enemy, it is quite easily imaginable what would happen if the vector size changes – again, if the last value is removed, the action the model has decided for may still be done properly, but if the action order changes, instead of shooting that fire bolt, the agent may as well walk positively with a strength of 0.5 in direction y, which, seeing the model has already been trained, will certainly lead to a very strange behaviour.

These circumstances make it, unless specifically planned to do so, unwise to reuse a trained model after changing vector sizes or anything corresponding to the meaning of the vector values in the Agent's behaviour script.

The vector sizes also directly contribute to the complexity of the agent. The action vector size, of course, means that the agent is able to perform a larger variety of actions, but it might also make it harder for the agent to find a beneficial action in the larger set of possible actions. The observation vector too leads to higher complexity, a larger set of possible observation, which each require an optimal resulting action to be found for.

While in theory larger vector sizes mean that the agents resulting a training become more aware and more capable, it might become impossible to train them long enough to achieve a level where they are able to perform accordingly, at some point even for vastly bigger computing power that was available for this project.

#### 3.2.1.4 Stacked Vectors

Aside from the actual vector size, it is possible to stack observations, meaning that not only the past one observation leads to a calculation of the optimal action, but, for example the past five. With a vector size of four and a stacking of five, an observation vector (not a matrix, hence stacking) of 20 results.

#### 3.2.1.5 Behaviour Type

There are three choices here; default, heuristic only and interference only

Per default, a model will be trained when launched through a training command and the model will be used to control the agent during regular play.

Interference only means that, even if started through a train command, no new model will be trained and the existing one (specified under Model) will be used to control the agent – this can be used to train a new model against an existing one.

Heuristic will let the user control the model through a heuristic function defined in the agent script – this is mainly for testing, but also allows the user to play through the agent script.

While this is appreciated for testing purposes, it is generally better to dedicate a separate script for to control things manually.

#### 3.2.2 Decision Requester

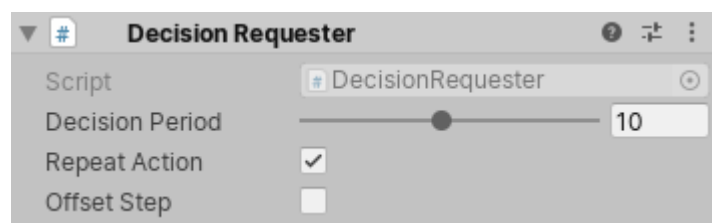


Image 6: Decision Requester in the Unity inspector

The decision requester allows for less configuration – the decision period specifies how many steps have to pass till a decision is made. With a decision period of ten and a stacked vector observation of five, the agent takes an action each ten steps based on the observations collected in the past five seen from the decision moment.

If there was a counterpart to the observation vector stacking, this would be it for the action vector – the difference is that the action amount is at its maximum if the decision period is left at the default value of one, meaning that increasing this value means that there will be less actions taken per steps. With this parameter, it is not only possible to “slow down” agents, which may be wanted, seeing a human counterpart would decide way fewer times than once per frame, but also to reduce the complexity of the training and computing power per time.

The repeat action toggle defines whether the agent should repeat the last action each step until a new action is chosen – if untoggled, the agent commits no action unless each nth step where n is the decision period.

The offset step toggles whether the agent should start with a random offset vector – this did not seem to generate much impact with the conducted trainings. [15]

### 3.2.3 The Agent Scripts

The agent scripts all inherit from the Agent class, a ML-Agents parent class that contains certain functions such as AddReward, Done and TotalSteps. The Agent class in turn inherits from the MonoBehaviour class, the standard class to define how GameObjects behave under certain circumstances.

Aside from the Start method, which may be added to every MonoBehaviour class, which is mainly used to load, assign or trigger certain events that are to happen when the game is started, the inheriting agent mainly has three important methods:

#### 3.2.3.1 Agent Resetting

First of all, it is important to be able to reset agents when they reach crucial user defined points, or the “Done” function is called. Aside from user defined actions such as the one seen below, where the agent’s position is reset to a spawn point, health is restored and some values are given to scripts keeping track of important statistics, the method also resets the current step count of the agent.

```
public override void AgentReset()
{
    //currentsteps at this point is zero, triggered at max steps or when Done();
    ResetPosition(transform);
    GetComponent<HealthController>().Max();
    GetComponent<CharacterStats>().Reset();
    GetComponent<CharacterStats>().TotalSteps(maxStep);
    arena.GetComponent<ArenaBehaviour>().deathcount++;
    int a = arena.GetComponent<ArenaBehaviour>().deathcount;
    // if (a % 5 == 0) arena.GetComponent<ArenaBehaviour>().UpdateTrees(); //trees randomized every 5th death
}
```

Image 7: AgentReset Method

For the character agents seen in the project, the AgentReset method is called through the Done function whenever the agent dies and/or directly when the MaxStep parameter set in the inspector is reached.

The picture on the left shows the assignable values in the unity editor of a warrior’s WarriorAgent script which serves as an example child of the BasicAgent class, which in turn inherits from the Agent class. Unlike the decision requester and the behaviour parameters scripts, the agent scrips have to be designed specifically for each use case.

Aside from values required for the game mechanics, such as the agent’s health, the arena prefab which defines to which of the many arena instances the warrior belongs and the Team ID, which is required to enable SelfPlay and determine which characters are on what team, the editor shows one important variable, the Max Step count.

This variable is important seeing it defines the number of steps an agent may take until it automatically resets during training – this exists among other reasons so that an agent which would be stuck in an unwanted state is able to reset and try again with an updated policy. [15]

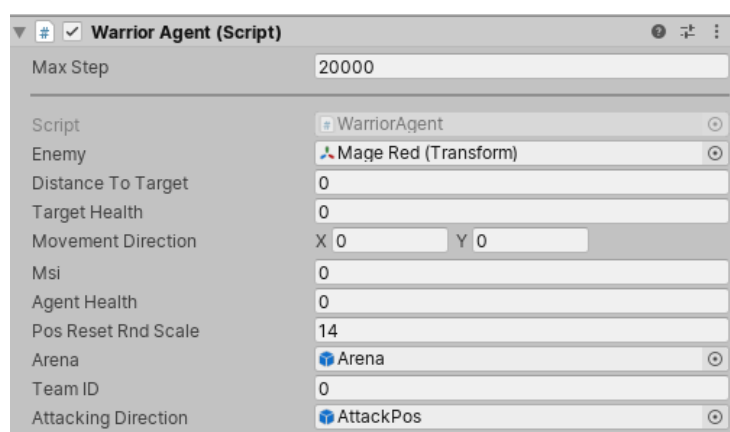


Image 8: Agent Parameters in the Unity inspector

### 3.2.3.2 Observations

Together with the method responsible for the agent's actions, the method defining what kind of values the agent is able to monitor is probably one of the most important aspects to work on when trying to improve the agent's learning behaviour.

```
public override void CollectObservations()
{
    enemy = arena.GetComponent<ArenaBehaviour>().ClosestEnemy(transform, enemy); //get closest enemy inside arena
    AddVectorObs(enemy.localPosition.x);
    AddVectorObs(enemy.localPosition.y);
    AddVectorObs(transform.localPosition.x);
    AddVectorObs(transform.localPosition.y);
}
```

Image 9: CollectObservations Method of an Agent script

The CollectObservations method is called each step and generates a vector of a size depending on the amount of values added to it through AddVectorObs.

As seen above, the vector size generated is four – it is also possible to add more complex values to the vector such as a coordinate, which can be three-dimensional as well. If instead of the two single values, the enemy's local position as a whole would be added, the z value would be included as well, resulting in a vector of size three without any other observations. If the same was done for the transform's local position, the observation vector's size would be six, which for a 2D game would expand the observation space needlessly. [15]

The transform corresponds to the GameObject the script is attached to, meaning it's the agent's own local position. The LocalPosition differs from the global position by the difference in origin, which with the local position is the one of its parent-object. In the case of the agents, this is the arena prefab instance, allowing for easy distinguishing between the various agents regarding which arena they belong to, whom they interact with and where they should respawn. [16]



### 3.2.3.3 Agent Actions

AgentAction is a bit more complicated than CollectObservations, but it basically revolves around the float array vectorAction, which will be of the size defined in the BehaviourParameters as seen earlier. Like already mentioned, the algorithm returns a float value between -1 and 1 for each position in the action vector on each decision step, which unless configured otherwise, is each step.

From that point on, it is the game developers job to make use of the algorithms output, by defining ways to translate the values into actions happening in the game environment, which then usually alter the observation vector the algorithm receives on the next step – trough a change of the agent's position for example. [15]

```
//action Vector
//References
public override void AgentAction(float[] vectorAction)
{
    distanceToTarget = Vector2.Distance(this.transform.position, enemy.position);
    if (GetStepCount() > 0 && GetStepCount() % 1000 == 0) GetComponent<CharacterStats>().DpSteps(GetStepCount());

    // Actions -> unity documentation: By default the output from our provided PPO algorithm pre-clamps the values of vectorAction into the [-1, 1]
    Vector2 movementAction = Vector2.zero;
    movementAction.x = vectorAction[0];
    movementAction.y = vectorAction[1];

    _a = vectorAction[2] >= 0.5f ? true : false;
    if (_a)
    {
        GetComponent<MultiSlash>().Attack();
    }
    _e = vectorAction[3] >= 0.5f ? true : false;
    if (_e)
    {
        GetComponent<ChargeAttack>().Charge(enemy);
    }

    movementDirection = new Vector2(movementAction.x * 100, movementAction.y * 100);
    movementDirection.Normalize();
    msi = Mathf.Clamp(movementDirection.magnitude, 0.0f, 1.0f);
    GetComponent<MovementController>().Move(movementDirection, msi);

    if (movementDirection != Vector2.zero)
    {
        attackingDirection.transform.localPosition = movementDirection * 0.5f;
    }

    if (distanceToTarget > 10.0f) SetReward(-0.005f); //far from enemy (being lame)
}
```

Image 10: AgentAction example of the WarriorAgent

The AgentAction taken from the WarriorAgent script seen in the above picture receives an action vector of size four, where the first two values are translated into the character's movement, by defining the current step's movement direction, which can be understood as a vector containing an x and y direction: Positive x: walk right, negative x: walk left, positive y: walk up, negative y: walk down.

The values are normalized to ensure that the agent always runs at the same speed, independent of the vector action value's strength. This means that the output value in the action vector for positions zero and one only distinguish between three situations: negative movement, no movement and positive movement – while this does not reduce the amount of possible outputs, it splits them into three categories, meaning that, seeing the chance for negative and positive movement are both close to 50 percent, the set of possible actions may be split into a good (beneficiary) and a bad action on each step. This in turn means that the agent is way more likely to find a beneficiary action compared to a value, where each state from -1 to 1 means something entirely different, such as a coordinate output required for targeting.

The remaining two action vector values are mapped to a trigger-Boolean. If the algorithm decides one of these float values to be equal or above 0.5, the Boolean is set to true, in which case a skill is used.

It is possible to generate way more complex constructs than displayed in the agent script above. The charge attack, for example, is aimed at an enemy – it would however be possible to aim it at a target coordinate instead, meaning that two more action vector values would be required to define the coordinate's x and y position. Like mentioned, this would give the agent more freedom, but also a larger set of actions, which in turn leads to a more complex and longer training. With limited computing power, however, it is crucial to keep the actions as straight-forward as possible and make the action vector only as big as really required. The boolean toggle comes in handy in various situations – the piece of code below lets the healer agent change between targeting its next ability at an ally or at an enemy.

```
_t = vectorAction[2] >= 0f ? true : false;
if (_t)
{
    target = enemy;
}
else { target = ally; }
```

Image 11: Boolean trigger showing the healer's targeting mechanism

The Magician's Agent script contains a variety of ways to shoot a firebolt:

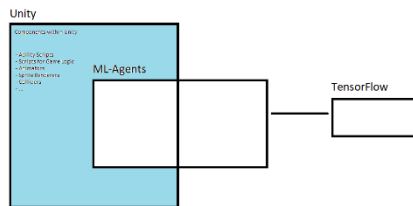
```
Vector2 attDir = Vector2.zero;
attDir.x = vectorAction[2];
attDir.y = vectorAction[3];

_q = vectorAction[4] >= 0.5f ? true : false;
if (_q)
{
    GetComponent<FireBolt>().BlastVec(new Vector2(attDir.x, attDir.y)); // -1 to 1 on both x y corresponding to vector added to user pos
    GetComponent<FireBolt>().BlastAngle(vectorAction[2]); //at angle -1 to 1 corresponding to -180 to 180
    GetComponent<FireBolt>().Blast(); //frontal blast
    GetComponent<FireBolt>().BlastTarget(new Vector2(enemy.localPosition.x, enemy.localPosition.y));
}
```

Image 12: Ways to shoot a FireBolt

At a time, only one of the four blast method variants should be enabled – while the two bottom most functions do not require any input in the form of an action vector entry, the angular blast requires one, and the vectoral blast requires two. In both cases the minus one to one possible action vector values are then interpreted by the FireBolt script's corresponding method.

## 4 The Game Environment



In addition to the components required for the reinforcement learning listed in the previous chapter, there are a certain few important components worth having a look at to get a better grasp of how the game is set up. In the current state, the game as a whole serves as a training environment, which is why distinguishing the following parts may seem redundant. The reason behind this is, however, that these parts are not specifically

required for machine learning, meaning that the game environment could be replaced through any other and in turn could exist without anything machine learning oriented in it.

### 4.1 The Arena

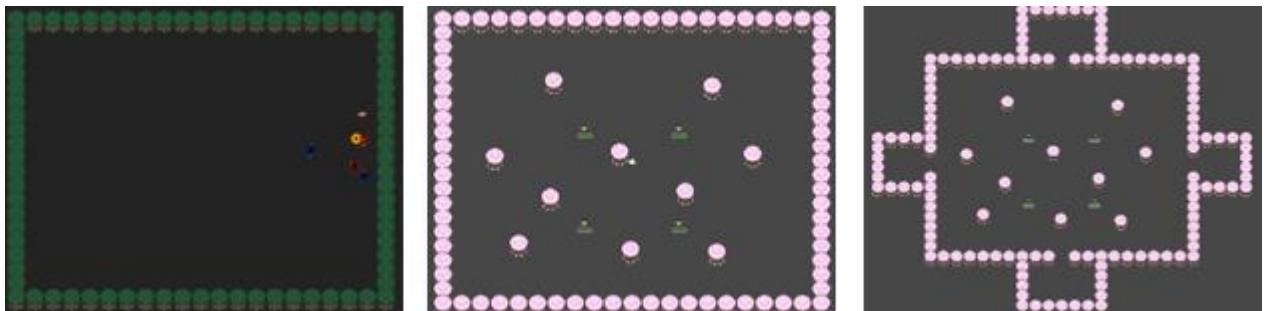


Image 13: The Arena in its various states

The image above shows the arena in various versions used during the training period. The first arena on the left was empty, while the walls mainly served to stop the agents from leaving the area and to enable multiple parallel arenas without, for example, agents from arena one fighting against agents from arena five.

In further updates, as seen on the arena in the middle, trees were added, which could be toggled to spawn at random locations either at the start of the training and/or change locations during it. The agents themselves spawned at random locations within the initial square area of the arena.

The arena to the right shows the four spawn areas that were added to it. The agents were set to randomly reappear in any one of them during training and game play. This served as an additional challenge to overcome.

The arena, at any given state was a so-called prefab.

A prefab is created by dragging a regular GameObject into a folder, saving the state of it and all its child objects and components, allowing for easier duplication and management of its contents. – Especially when having multiple arenas active at the same time, which should all be identical.

To change something on an instantiated arena, it is possible to do so directly by clicking on one of them in the hierarchy window as seen in the picture on the right. Committing a change here, however, only applies it to the selected arena. The blue font means that the selected object is a prefab.

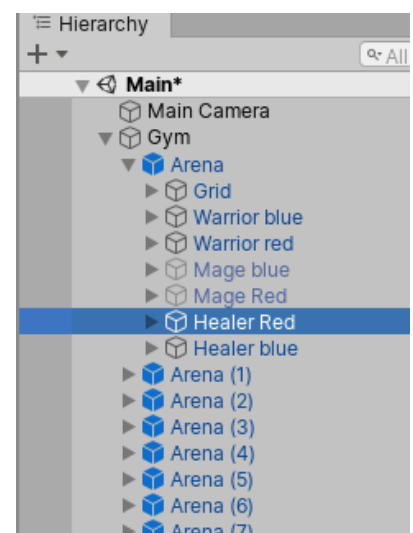
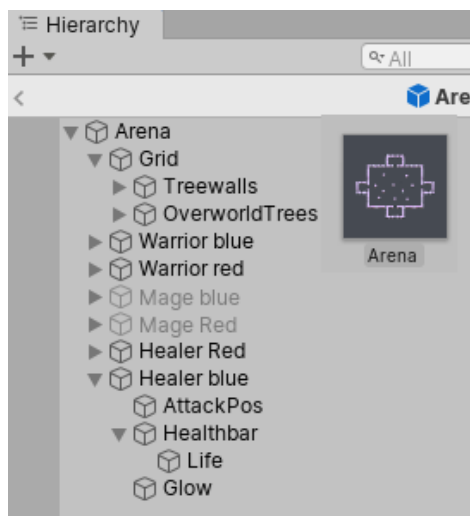


Image 14: Unity hierarchy showing the gym with its arena prefab children

By clicking on the arena icon in the env folder, however, it is possible to open the prefab into the hierarchy window. Editing it here will apply changes to all instances of the prefab. (picture below)



In addition to the trees seen in the prefab hierarchy, an arena contains all the instantiated agent-controlled character entities, whereas the greyed-out ones (Mage blue and red) are in a disabled state.

Image 15: Unity hierarchy showing the arena prefab's structure

## 4.2 The Characters

Clicking onto a character in the hierarchy opens it in the inspector window depicted on the right.

The Character itself is a GameObject. Every entry in the list is a component – the first few handle the character's interaction with the game world, how its sprite is rendered and animated. Further it contains the already described components required for reinforcement learning such as the behaviour parameters, decision requester and the healer specific agent script, the HealerAgent. The character stats script was added to gather statistical data on the agent's performance, such as lifetime and damage done per step.

The rest of the scripts control the character's health, movement and current status and allow whoever is in control of it to use any of the attached abilities if defined possible in the controlling script, which in this case is an AI agent through the Healer Agent script.



Image 16: Unity inspector showing the components of a character game object

### 4.3 The Gym

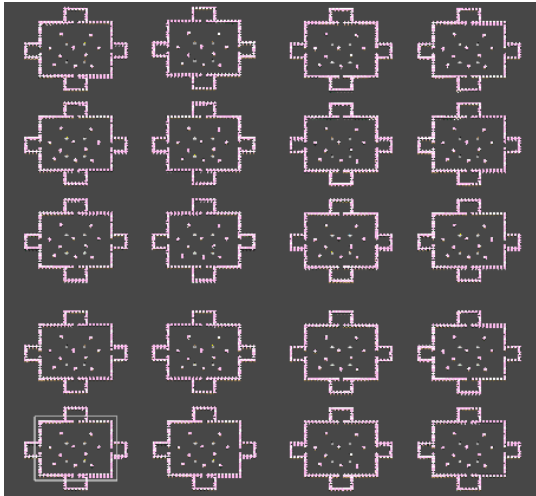


Image 17: Screenshot of the gym consisting of 20 arena instances

The gym is the collection of all arenas, the entire environment used to train the agents. It is simply a `GameObject` serving as a parent to all arenas – it contains a statistics script used to collect and evaluate data from all arenas. In its current state, it only saves the value of the highest killing spree achieved by an agent during the entire training or play session.

The term Gym is also generally used in machine learning to refer to the training setup which contains all instances of used training environments. While the gym in this case only contains one type of training environment (multiple arenas), it may generally also contain a variety of different environments, which can either be a variation of the same environment or an entirely different one.

## 5 Training and Improving the AI

This section outlines the different stages of the AI training throughout this thesis. Section 5.1 outlines the first few training attempts using the first type of agent whose goal was to run towards a door. Section 5.2 describes the second type of agent which had to fight against a training dummy. Section 5.3 shows the introduction of letting the AI train against itself. Finally, section 5.4 reports on attempts to optimise the training and resulting agent behaviour by working on the reward distribution.

### 5.1 RunBot

Like mentioned in the getting started with reinforcement learning in unity chapter, the first AI controlled agent created was called the RunBot – calling it “RunAgent” would have been a better choice.

Even though its behaviour was supposedly similar to the RollerBall example found in the ML Agent tutorial, it required a few attempts to get the agent run towards the door in a steady manner.

Having had only one arena instance at the time, even a very simple training took roughly twenty minutes and, at first, the results were so-so at best. Usually, the agent would run off towards a wall for roughly ten seconds until it reached its MaxStep count, at which point it would reset and repeat. Usually, after roughly ten minutes, the agent would sometimes start to find the door, but not on a satisfying basis.

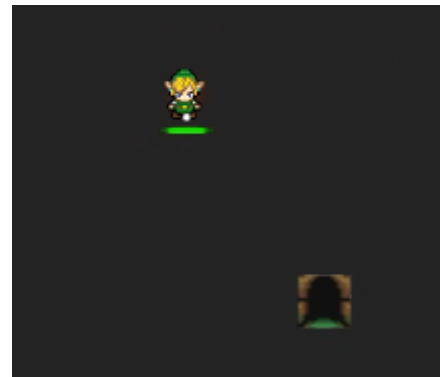


Image 18: Runbot and door as objective

To counter this, a negative reward was added to the agent based on its distance to the door – this slightly improved things, but the training still felt too slow – in retro perspective, considering the fact that there was but single arena containing one agent, the training times could have been worse, for sure.

Yet, trying to find a way to improve things, the agent script received a reset mechanism that would trigger when the bot ran away too far from the door, increasing the speed it would grasp that it's doing something wrong. With all these changes, the training was able to be completed in roughly 30000 steps, which, being quite simple, translated to roughly five minutes.

Yet, applying knowledge earned throughout the term, this time could be greatly reduced once more. Firstly, a simple environment like the RunBot would certainly be trainable with at least 32 arenas instead of just one. This form of training parallelisation would greatly increase the speed the agents learn.

Secondly, the RunBot had an observation vector of size nine, comprising out of a three-dimensional vector for both the bot and the door's position, current movement speed in x and y direction and a float for the distance between the two. A vector of size four for both positions would suffice.

## 5.2 Attacking a Stale Target



Image 19: BattleBot - Agent

Adapting the PlayerAgent script, which earlier controlled the RunBot to allow for a first combat-oriented agent, involved mainly adding two more fields to the action vector, which would enable it to blast a FireBolt in walking direction and make use of the MultiSlash ability.

The MultiSlash ability was already implemented during the project 2 module and basically lets the user attack with its sword in a four hit combo with a short internal cool down – the agent first strikes with its left weapon, then its right for equal damage and then twice for double damage with both weapons.



Image 20: The MultiSlash ability

To be able to train the agent, an old bot enemy was repurposed to serve as a dummy with its movement ability disabled. Rewards for damaging the dummy were granted by checking for health changes on each frame.

At this point, thinking that it was beneficial to the agent's learning process to know as much as possible about the environment, the observation was bloated further. In addition to the nine fields perceived by the RunBot, the target's health and the agent's attacking direction object's position were now monitored, resulting in an observation vector of thirteen and an action vector of four. First attempts to improve the barely learning agent mainly targeted the reward system and training hyperparameters. Increasing rewards to the agent for various actions such as staying close to the dummy and granting a bonus when it dies only provided very small improvements. Increasing the training time by adapting the max steps parameter in the trainer\_config file to let the agent training for longer periods up to roughly three hours provided noticeable improvement, however.

It took a few days until it became clear that increased vector sizes are not only beneficial. At this point a closer look at the agent's observation vector was taken, leading to trainings with a variety of vector sizes.

First of all, the character's horizontal and vertical movement speed felt quite obsolete and were removed, reducing the vector size to eleven. Further, the target health was no longer monitored, decreasing the size to ten.

At this point, the training time was reduced to roughly an hour to be able to see the agent attack the enemy now and then.

Seeing that this did not feel satisfying at all, a decision was made to remove the MultiSlash ability entirely and playing around with the target observations. A few trainings were run, one with only the distance to the target and the three actions, one with the agent's three position inputs plus the distance and another with both agent and dummy position, without the distance between the two. The last option turned out to be the best, probably because the agent just did not have any way to tell in which direction it had to shoot or move otherwise.

## 5.3 Self-Play

Seeing that the previous agent's training results started to become more and more acceptable, it was time to move on to the next step – to let the agent train against an enemy that fights back. Goal here was to not only let the agent fight against a random bot that would not learn itself, but to let the agent face off against itself.

For the two identical agents to be able to interact with each other, quite a few updates to the game's mechanics had to be made – most of the damaging abilities contained a value to be set in the Unity-inspector, which would then allow the ability scripts to check for what targets were to be affected by it.

The entire system was revamped in a way that the script would compare the GameObject (transform) it was attached to with the one of the colliders that was hit by the ability or the ability's projectile.

After updating the game logic, "SelfPlay", a feature of ML-Agents became implementable. To do so, a few parameters and values had to be added to the trainer\_config file and the agents had to be assigned to teams, by changing one of the two else almost identical agents' team ID in the unity inspector under the behaviour parameters to one.

The image below shows the additional lines required for self-play in the trainer\_config.yaml.

```
self_play:
  window: 10
  play_against_current_self_ratio: 0.5
  save_steps: 100000
  swap_steps: 20000
```

Image 21: SelfPlay parameters in the trainer config file

The window size defines the number of past selves that are stored and reshuffled to be played against.

The play\_against\_current\_self\_ratio parameter is quite self-explanatory; it defines how probable it is for the agent to play against another agent whose state, meaning its policy, is the same.

The save\_steps parameter defines how many steps are to be taken by the agent until its policy is saved as the newest one to be added to the window. [13]

Swap\_steps determines how frequently the enemy's policy is changed to another contained in the window. With the parameters configured as is, an agent should see five swaps before its policy is saved. With a high play\_against\_current\_self\_ratio, the agent is likely to encounter the same policy multiple times before being saved.

The swap\_steps parameter was set to match the max\_steps parameter of the agent scripts, this is not a requirement, however. It just turned out to be a good value after training multiple times with various numbers.

To distinguish the agents easier during training, their sprite colours were changed to blue for team zero and red for team one.

The first training went better than expected, however, no matter what was changed, the agents seemed to constantly develop the tendency to run into a corner and have a blast off until one of the agents' health dropped to zero, causing it to reset.

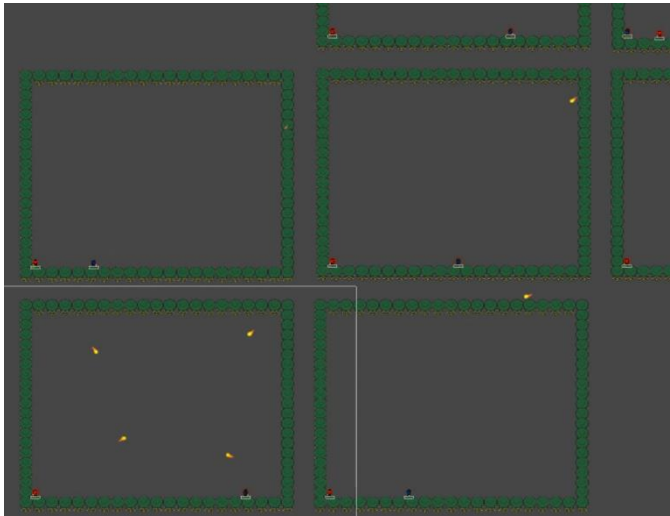


Image 22: Result of first trainings using self-play

At this point it was very hard to tell what issue caused this strange behaviour, a guess was that it was simply too hard to fight against an identical self that would dodge and be able to deal the same amount of damage if both agents behaved in a roughly equal way. Standing right next to each other would maximize the probability of being able to hit each other.

Thinking that the training was too complex to be mastered in one arena, it was time to expand the training environment to allow for multiple arenas – at first, the gym consisted out of twelve arenas:





The agents kept the tendency to storm off to a corner upon respawning, running along the walls to find their counterpart. The first training with multiple arenas was not really useful, because the arenas were too close to each other, enabling agents to shoot their projectiles towards enemy agents of different arenas. This was countered by simply moving the arenas further apart. At least the FireBolts were previously updated to check for the team ID when deciding whom to hit and apply damage to.

Later on, the FireBolts were set to trigger and explode upon colliding with the trees.

Image 23: Agents shooting at fixed spawn locations

The agents were set to spawn at a predefined location, which caused the agents to mainly shoot towards them in the middle of the arena. The agents were hence updated to spawn at a random location within the square.

With longer trainings, the agents strangely started to stop firing at each other altogether – this was very interesting to observe, it felt like they decided to live together in peace, seeing they instantly regrouped in a corner upon respawning.

A variety of attempts to counter this behaviour were taken at this point, including the removal of the tree-walls.

Up to this point, most rewards were set in the PlayerAgent script that remained in control of the agents directly.

Instead of always checking whether there was some change on the target's health value by accessing its HealthController, taking damage had to be updated to directly set rewards to both involved parties – the one causing the damage and the one receiving it.

This was done by updating the HealthController's TakeDamage method to not only receive an int, but also the transform of the one causing the damage and then handling the reward logic there:

```
public void TakeDamage(int damage, Transform dmger)
{
    dmger.GetComponent<BasicAgent>().AddReward(damage * rewardmodifier); //reward to attacker (dmger)
    dmger.GetComponent<CharacterStats>().DmgDone(damage); //update stats

    GetComponent<BasicAgent>().AddReward(damage * -rewardmodifier); //reward to attacked character (dmgd)
    GetComponent<CharacterStats>().DmgTaken(damage); //update stats

    if (damage >= health)
    {
        dmger.GetComponent<BasicAgent>().Victory();
        GetComponent<BasicAgent>().Defeat();
    }
}
```

Image 24: Agent rewards in the HealthController's TakeDamage method

The training then vastly improved when a closer look at the observation vector was taken: Instead of giving the agents 3-dimensional position vectors it would be way more efficient to simply pass them the x and y coordinates of their local position as explained previously. By stopping to observe the agents' health as well as anything else than both positions, the observation vector was reduced to the size of four.

To be able to not only visually measure the quality of the agents' behaviour, it was important to establish a set of metrics; A CharacterStats script was added to the agents, ArenaBehaviour to the arenas and the "Statistics" script to the gym. This made it possible to get an overview of how much damage the agents did per steps during training, how much damage was done and taken by single agents and total, how long agents were able to survive, how many victories over other agents they achieved, how many kills in a row they were able to perform and what the highest killing spree was during a training or gameplay. These metrics were then used to generate additional rewards based on the agents' performance.

## 5.4 Improving the Reward Distribution

Having already made quite a few adaptations to the way rewards were granted to the training agents, but not getting the anticipated changes, it was time to have a more thorough look at the ML-Agents documentation, to see whether there are alternative ways to distribute rewards.

Training so far were all based on the default proximal policy optimization (PPO) algorithm – a second option, the soft actor critic (SAC) promised to vastly improve training speeds, a very tempting promise. The key difference with the SAC algorithm is that it uses a sample buffer to evaluate a large amount of observations seen in the past, which may also come from actual gameplay. The algorithm further rewards the agents to try new things, while a variety of parameters allow for customisation of that feature, to define for example, that the agent should change its policy more frequently at the start of an episode and less so when it is about to reset. [17]

Sadly, trying to make use of the algorithm did not work out as planned – the massive calculations with the sample buffer each frame were simply too much to handle for the machine available for training, even when lowering various values that would affect this intensity and setting the environment to train with just a single arena instance.

Instead a decision was made to continue the focus on improving the agents with the PPO algorithm and, if possible, take another look at SAC at a later time – due to the covid-19 situation, however, access to the BFH's computers would remain impossible.

Instead, a further look at the ML-Agents forums provided another lead to make the rewards count for more:

The SetReward method, which had been used so far, sets the value given as parameter to be the total reward that the agent collects on its next decision step, while the newly discovered AddReward adds it to the total of previously added rewards. These are then collected in the same way.

Having thought that SetReward was the only way to add rewards up to this point, this discovery was quite shocking – this meant that over half of the rewards the agents were supposed to receive were simply overwritten and lost. [18]

New trainings showed that the error was by far not as grave as expected.

A possible reason could be that, given the length of the training and the vast amount of steps taken across the sum of agents, the probability of seeing the right rewards over time was quite high – even with a few rewards gone missing, the agents would still become able to distinguish between good and bad actions. The updated reward giving would lead to a noticeable training time reduction, however.

During roughly a week of training with the updated reward system and a rebalancing of the agents' abilities, the main goal was to increase the damage the agents did per step (dps).

The ability rebalancing included the fire bolt ability to be more frequently useable but do less damage and have a shorter range instead. The fired projectiles were also configured to travel slower to increase the odds of a successful dodging action to be rewarded.

Among other experimental trainings, the agents were also tested with the MultiSlash ability reenabled.

The agents slowly started to reach a point where they were close to achieving the maximum possible rewards, while the damage per step (dps) reached its limit as well – This was the case when every FireBolt, used on cooldown, that was not actively dodged by the opponent hit its target. Furthermore, the agents' behaviour made sense visually too; they started to make use of the entire arena space, moving around at a distance to each other which would roughly equal the FireBolt's reach. If they were facing each other in a horizontal line, they would close in to fire their shots and then move back out, whilst moving in a way to randomise their vertical position to be able to dodge the opponent's projectiles.

It was hence time to move on to a next step.

## 6 Game Mode Exploration

With the criteria of the previous two milestones being met, this last phase of the thesis focused on the exploration of the interaction between the trained AI and various changes to the game, leading to further adaptations, improvements and variations of the agents.

### 6.1 Training N Versus N Agents

Even if still used to train agents in a one versus one situation at first, various updates had to be made to the environment in order for it to allow for more than two agents to work properly.

So far, the agents all received their target through the unity inspector, by simply dragging the enemy agent into the target field.

While having a default target is still required for the game not to break and seeing that a training would not make much sense otherwise, the arenas' behaviour script has been updated to allow all characters to register themselves to a list.

While various options would be possible to implement at this point, letting the agents observe the position of  $n$  entities simultaneously would have led to an increase in the observation vector's size. Seeing that the vector sizes need to remain unchanged, as explained in the "Understanding the machine learning environment" chapter, changing the vector sized during training or gameplay would not be possible.

The way to go was hence to give the agents a way to select their target. For simplicity reasons, the agent simply receives the closest enemy as a target. To achieve this, the `ClosestEnemy` method in the `ArenaBehaviour` script basically checks to which team the agents in the list belong and returns the closest character, whose team id does not match the user's, back to it.

Having everything aside from the logic determining the agents' target remain unchanged, meant that this form of implementing multi-target-compatibility allowed for brains trained in the past to remain useable.

A more complex variant to implement this would have been to add the size of the enemy list to the agents' observation vectors and then letting them chose, through one action vector value, which target they wanted to acquire, by mapping the -1 to 1 values to a value between zero and the enemy list's size minus one.

Even if slightly more complex, this would have been simple to implement, however, it would expand both the action and observation vectors, which is why the simpler variant was chosen. Given time, this would be a very interesting aspect to experiment with.

Like mentioned earlier, it was important at this point to keep the vector sizes the same, so agents trained in the past were still reusable.

The first step now was to do exactly that and take a brain, trained previously in 1v1 combat, and see how it would perform if assigned to four agents that would play in teams of two against each other.

This went extremely well:

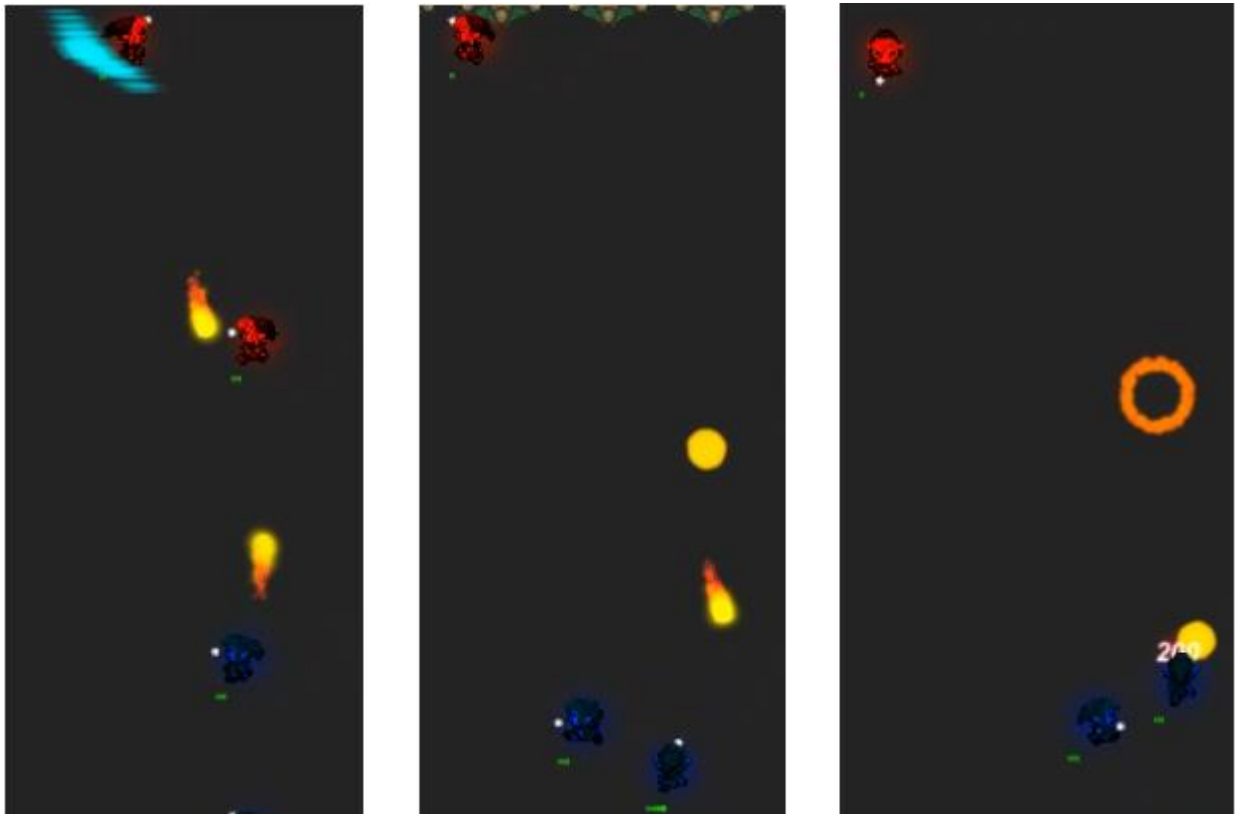


Image 25: 1v1 to 2v2 gameplay sequence

From left to right, the image shows a sequence of actions recorded during the gameplay of the 1v1 brain in the 2v2 environment as described above.

The fire bolt projectile traveling downwards (shot by the top red agent) seems to be on a path that would lead to missing the blue agent in picture one, it is however aimed at the second blue agent moving in, seen on picture two, which is then hit at the time the third picture was taken. The fire bolt traveling upwards hits the red agent on picture two, while picture three shows the projectile's explosion – the red agent's health was low before getting hit, as can be seen on picture one, which is why he dies and resets to a random position outside of the picture area.

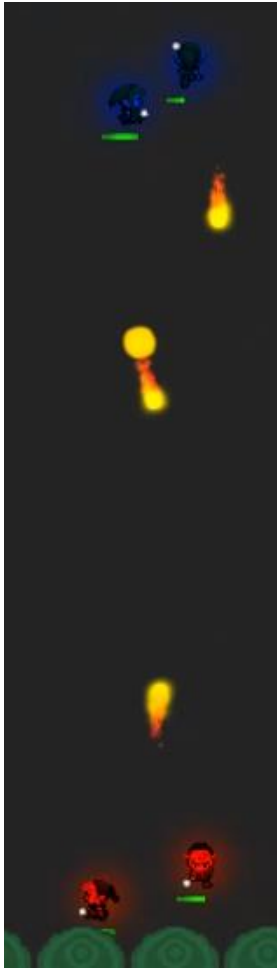


Image 26: Teammates acting as a unit

In the next step, to see the difference, a new model was trained using the setup from the 2v2 with the 1v1 trained brains – this would lead to them being trained in a way they would be aware of the two enemies.

The result was not entirely unexpected, yet interesting to see – the agents seemed to always act as a team, even though they have no awareness of their teammate at all. This is due to the fact that both agents are controlled by the same brain, meaning that the agents' policy leads to a similar decision in a similar situation – it never being identically, however, seeing that the agents cannot stand on the same position, nor may their enemies, which means that the closest enemy relative to one agent may not be the same as its teammate's.

The behaviour seen on the picture on the left extremely resembles the behaviour of the single bots in the 1v1 environment. Instead of having one agent run around at the max range of the enemy's fire bolt, it would now be a pair of agents behaving the same way. The pair of agents described on the previous page which were trained without the two additional entities acted way more independently, meaning that the teammates were often apart and, in some cases caused two separate one versus one duels.

Further tests involved having one versus one trained agents on one team (blue) and two versus two trained ones on the other (red). Oddly, yet again not inexplicably, the one versus one trained agents performed better in the, to them, unseen situation. Reason for this may be found in the fact that, to the red ones, the blue agents' behaviour was unfamiliar as well – if, for example, firing close to a targeted enemy would usually land a hit on either the target or its ally, this was not the case when facing team blue, seeing they were more spread out. In other words, the red agents learned a way to achieve optimal results against a team of united enemies, while their blue counterparts stayed focused on their target.

## 6.2 Agent Variations

While each step so far brought many new paths that could have been taken and many of them would have been worth chasing given more time, covering various types of agents was one of the larger goals set for the project and was hence the path chosen at this point.

The agent so far was some sort of all-rounder, having both range and melee potential with the FireBolt and MultiSlash abilities. It was now time to split the abilities as intended – to create a warrior that would excel in close range combat and a mage that would try to keep its distance to the approaching enemy.

At this point, the game's logic had to be updated again, mainly in regard to accessing the agents' properties and methods. To do so, a new parent class, the BasicAgent was introduced among two variations extending it, one for the warrior and one for the mage. While most of the logic was identical for both, meaning that it could be implemented in the basic class, the observation and action vector were left for the child agents to specify. While the observation vector did not need any changes, the action vector could now define a variation of abilities to be used.

Further the split made it possible to add different rewards to the roles – the warrior seemed to be overwhelmed by the magician for quite a long time, which is why he received a negative reward each step, increasing with the distance to its target.



Image 27: Arenas showing the warrior and magician agents during gameplay

The above is a screenshot of one of the trainings with the new warrior and mage agents. Both teams were assigned one warrior and one magician.

The agents now each had two abilities they could use:

In addition to the MultiSlash, the warrior received a charge attack, which would allow it rush at a selected target, by massively increasing its movement speed for 0.1 seconds, fixating its movement direction towards the selected target, disabling all further movement input for the duration. The ability would also stun the target for a short while upon impact. The charge attack can be seen in action on the bottom left arena. Due to the roughly tenfold increased game speed during training, the rendering is a bit laggy, which explains why the ability seems to have missed its target.

The magician received the FireBolt and a teleport ability, allowing it to teleport backwards a short distance. The teleport ability can be seen on the top right arena. (Blue particle-effect)

Like already mentioned briefly, the mage agents usually had the upper hand – with various efforts taken to strengthen the warriors, like for example, increasing their ability damage, which also affects the rewards they receive, updates were made to favour a more aggressive gameplay. The general issue was that the warriors, like the mages tried to stay at fire bolt range, seeing they would simply get overwhelmed when trying to reach the mages – even with the charge attack as a gap closer. Often the warriors would charge in and then instantly run away instead of staying close to the magicians to be able to hit them with the sword slashes.

The way of improving them lead through a variety of trainings with various setups, such as having two mages vs two warriors or only warriors and then pitting them against mages during gameplay.

A lot of rebalancing was needed – great change came with the repurposing of the charge ability – instead of being focused on dealing damage to the mage, it should function as a gap-closer. To achieve this, the ability's damage was decimated in return for increasing the frequency it could be used and the duration it would stun the target. The range and ability speed had to be adapted as well. Further, the damage from the MultiSlash ability was increased to make it more lucrative.

Together with the addition of the previously mentioned negative rewards the warrior would receive based on target-distance the results would now steadily improve with the time spent working on the agents.

### 6.3 Obstacles

Having simplified the observation and action vectors both to the size of four and deciding ability targeting to be target based, the agents became as lightweight as possible in terms of their parameter space. With the agents starting to reach a point where they performed well enough for it, it was now possible and hence also time to answer a few questions about how the agents would handle environments challenging them in new ways.

One of those questions was to see how the agents would adapt to a set of trees in the arena. With a rather small update to the fire bolt ability that would make the projectiles explode on collision with a tree, the addition of ten static trees would get the environment ready for the first few tests, which included having agents that have never seen them before face each other under the new circumstance.

Even though the agents were barely ever stuck walking into a tree, there was obviously no awareness, which meant that the agents often attempted to shoot through the newly added obstacles, leading to a decrease in dps for the magicians. The warriors struggled less, sometimes trying to charge their enemy and ending up walking against the tree for a brief amount, before then running around it to face the still stunned target.

Training the agents with the trees, static still, showed clear improvement of the magicians' behaviour. While some bolts still hit collided with them, the agents' policy configuration through training made it possible for them to be aware of their locations, meaning that they would usually position themselves in a way that would increase their chances towards having no tree block the path of their bolts.

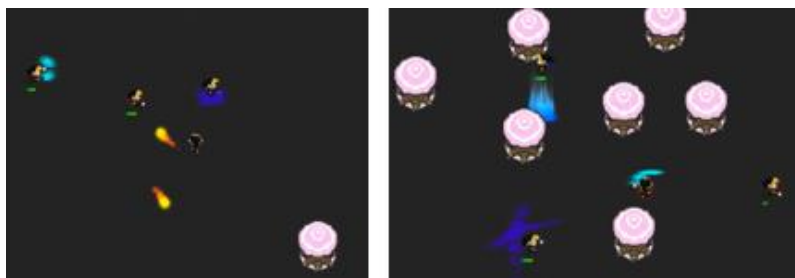
The training results started to become more interesting with the trees randomized, however. The setup, which might have been overkill slightly, revolved around having the obstacles update to a random location every fifth kill inside an arena.

The trees were set to start out at a random location for each arena, as well. In other words, having trained with 16 arenas at this point, the agent brains trained with 16 different tree clusters at any given time, while each arena updated the trees at a different time, depending on the performance of the agents within it.

Over the course of the roughly two hours long trainings the agents were exposed to the changing tree locations, it is quite safe to say that the two agent brains, each consisting of the experience of 32 agents – two magicians and two warriors per arena – have both seen enough trees to cover the entire arena with their colliders, meaning that there was not a way for the agents to learn a lot about possible tree locations.

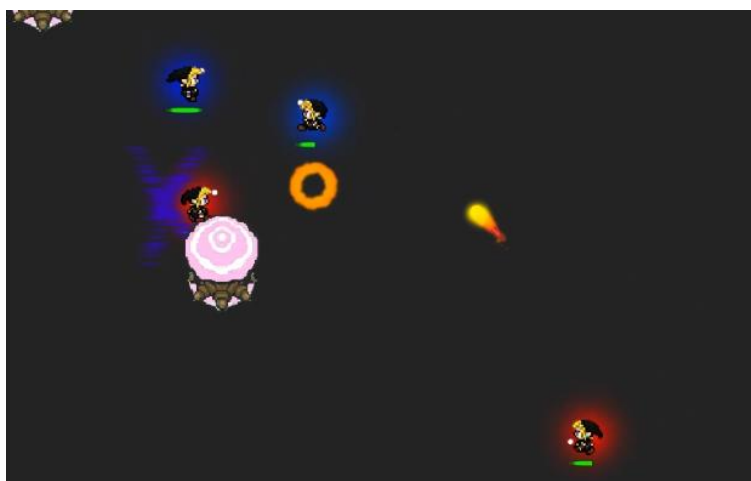
While they could not reach the exact same level of performance as seen from agents playing with static trees or none at all, the agents adapted to the vastly more difficult environment quite well.





While not always the case, the agents clearly showed a tendency to prefer an area without trees to stage their battles. Agents respawning inside a tree cluster usually showed attempts to leave the cluster, succeeding rather quickly after unavoidably bumping into them a few times.

Image 28: Agents avoiding randomly reappearing trees



Especially interesting to observe was, that unlike most previous trainings, the magician agents would not fire their bolts on cooldown anymore if there were too many trees around. Almost none of the bolts collided with trees, as the agents somehow managed to fire them only if they had a clear shot. This is even more impressive considering that the trees seen during training would, aside from a neglectable probability, never be in the same location as during gameplay, where the observations were made.

Image 29: Red agent going for a clear shot

To create an environment, where biased behaviour would become more likely than with the constantly updating trees, a training was conducted, where they would only be initialised at a random location, remaining unchanged during training. This way, the agent brains would see 16 variations of tree clusters during training. When testing the agents, each agent would then be instantiated in an arena, meaning that the agents would each only face a single tree cluster variant during gameplay.

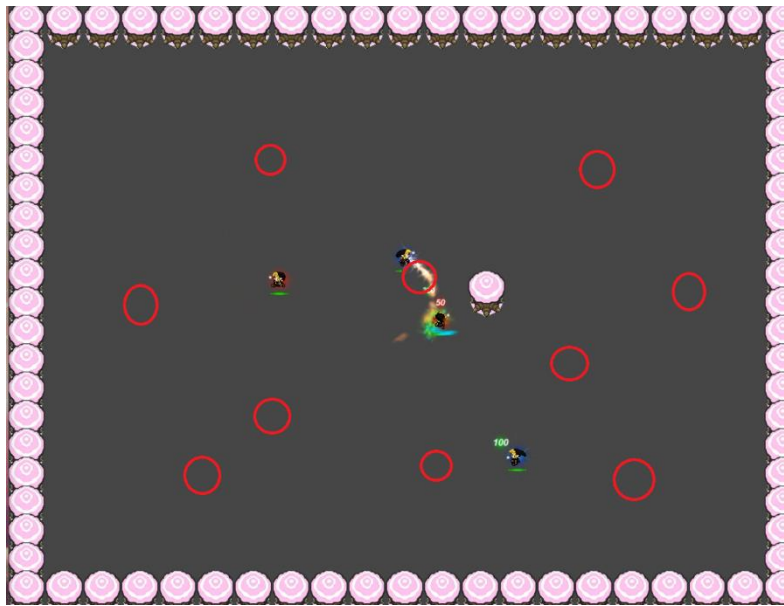
Bias would now be detectable if the agents had learned the locations of the 10 x 16 trees during training and were to, for example, try to avoid walking where they were located, but are not anymore. Aside from that, in theory, bias would also be observable if, the agents tried to take cover behind or stopped shooting through the “imaginary” trees.

Such a bias was not observable however – reason for this might be that the randomisation at start provided the agent brain with enough variations not to predict specific locations.

A further attempt to create biased agents was made by having the agents train versus each other with static trees.

The trained agents were then tested during gameplay, where each arena would start with an individual, random set of trees that would then remain static. This meant that each agent instance would see one cluster of trees during training and one different one during gameplay.

Seeing that during gameplay, the agents in different arenas share no connection, the fact that the trees’ locations would not be the same in each arena, would not matter to an agent-instance at this point.



The picture on the left shows four agents trained with a set of ten trees. During gameplay, the trees were removed and a new one placed. The red circles depict the location of the ten trees during training.

Thorough testing showed that the agents did not learn the specific locations of the trees, rather they seemed to have learned to actually work around them when encountered through collision or finding out about them when a projectile did not hit its destination as expected.

Image 30: No bias regarding tree locations

In other words, the agents were neither walking into the new tree, nor did they specifically avoid any of the past trees' locations during combat.

## 6.4 Spawn Zones

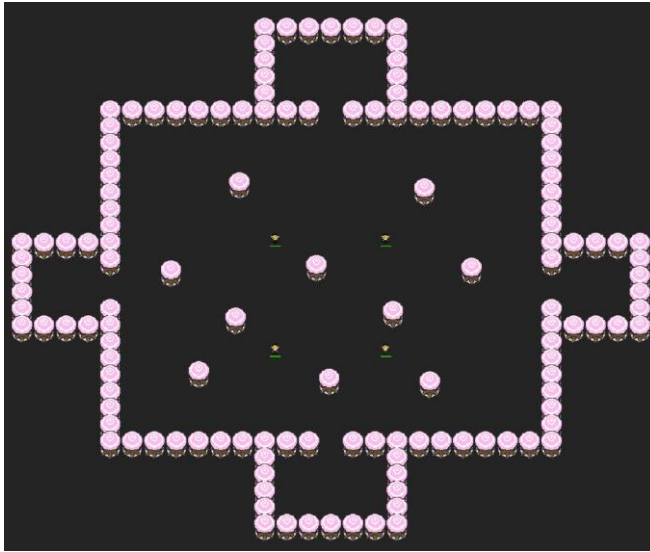


Image 31: Arena with spawn zones added to all sides

A further test scenario required the implementation of designated spawn areas as an additional step. Without adding too much logic to them, four spawn zones were added to the arena – one on each side. The agents would now spawn and respawn inside one of them, chosen at random, and be challenged to find their way out to be able to face the others - if they did not want to be affected by the negative reward for cowardice on each step by being too far away from a possible fight.

The first trainings included both the newly added spawn zone mechanic and the randomly updating trees, which together proved to be quite hard for the agents to overcome – with the usual training period of 2 million steps, taking roughly one and a half hour to finish, the agents did not manage to break free from the spawn zones fast enough – training with 6 million steps provided far better results, yet the agents still took their time finding a way out of the little squares. A very long training of 8 million steps lead to them being able to escape almost instantly, which is why an attempt was made to figure out how long the training had to be exactly, this turned out to be at roughly 7 million steps.

Taking away the randomisation of the trees vastly reduced the training time. Agents with one million training steps did not struggle with the setup at all. Randomizing the trees only at the start lead to agents requiring about 2 million steps to learn near instant escaping followed by a decent performance in battles.

Playing around with this feature once again shed more light on how little it takes to vastly intensify the complexity of trainings and the brains required to overcome the tasks at hand.

## 6.5 Healer Agent

Seeing the goal of having a variety of agents felt still very barely accomplished with simply two types of agents and having an urge to add something entirely different to their set, the healer, previously controlled by either the player or the HealBotBehaviour script, provided the ideal target for an AI implementation. In its original variant, it had three abilities it could make use of:

The HealBolt, an ability that would fire a projectile, like the FireBolt, which would instead of damaging the user's enemies, heal all allied targets in its path, the SelfHeal, which allowed the healer to instantly heal a portion of its missing life and the HealWave, a slightly more complex projectile that would travel outwards to a certain maximum range before turning around and returning to its caster. Each character that finds itself in the ability's path, depending on its team ID, is either healed or damaged by the ability - up to two times: once before and once after changing direction.



The picture on the left shows a HealWaveProjectile on its way back to its caster.

Image 32: HealWaveProjectile returning to its caster

The main challenge with the implementation of the healer was the requirement of a new targeting mechanism. First of all, seeing the healer would mainly try to heal its allies, it would require to be able to find a nearby ally. With the ClosestAlly method added to the ArenaBehaviour script, working in a very similar way as "ClosestEnemy", this was barely an issue – seeing the HealWave would be able to both damage enemies and heal allies, it felt important, to give the healer a choice between firing an ability at an ally or an enemy.

In a first attempt, the HealerAgent's observations were kept the same way as with the other agents – two observations for the selected (closest) enemy's and two for its own position.

The action vector required two more values – one to serve as a Boolean toggle for a third ability and one to toggle between choosing the closest ally or closest enemy as target to shoot its abilities at during a single step. Having no observations on the closest ally, the following training was more of a test to see how capable of switching targets the healer would be with such a simple implementation. The results however were way better than expected. Seeing the healer had no awareness of its distance to the closest ally, it sometimes tried to heal it, even if it was out of reach. Further, it sometimes tried to shoot heal bolts at the enemy, which, seeing only the heal wave has a damaging function, was quite a useless behaviour.

In a further update, the healer would observe its current target's position instead of the closest enemy's, meaning that, depending on what it chose to set as target on its last decision, it would now either observe an enemy or an ally.

As the healer's rewards mainly came from healing an ally, the trainings were always conducted in a 2v2 environment at least, usually with a warrior as partner. For simplicity, the HealBolt was disabled in order to slightly reduce the action vector of the healers.



Image 33: Healers facing off against each other

Oddly enough, the teammates showed the tendency to split up, leading to many one versus one situations between the healers and the warriors among their own.

What was most interesting, though, was that, usually shortly before one of the warriors would have died, its ally came slightly closer to save it.

While the healer seemed to correctly identify what kind of target it was observing, the lack of the HealBolt ability meant that it was not possible to see whether the healer would still choose the wrong target.

Adding the HealBolt and thereby reverting the healers' action vector back to the original size of six meant another push at the agents' limits.

While results of the first few two million step trainings showed that the overall behaviour did not change much from the previous generation, the usage of the HealBolt remained a bit too random. Firing at the enemy roughly four out of six times, lead to no reward gained for the healer (at least in theory), unless an ally was caught in the ability's path.

With the healer's warrior partner usually being close to at least one of the enemies, this was quite often the case.

Seeing this way of earning rewards is rather coincidental, yet barely avoidable, a longer, six million step training was to be taken in order to help the agents find a steadier income of rewards.

As the result did not change regarding the described issue, a closer look at the healing projectiles was taken, leading to the discovery of an error:

Instead of triggering a heal if certain colliders belonged to the user's teammates, the ability was set to heal the enemy characters. Even though this seems to be the explanation for the strange usage of the HealBolt at first, consideration of the fact that only roughly half of the bolts were aimed at the enemy shows that fixing the error might only invert the situation. A following training confirmed that assumption, which is why another solution had to be found.

Even if there were multiple ways to reduce the complexity of the agents without losing the HealBolt ability, for example by applying the chosen target only to the HealWave ability and determining the HealBolt to always shoot at the closest ally - or by removing the less interesting SelfHeal,- it was way more interesting to investigate what a longer training would bring.

A training of six million steps did not really bring the hoped-for results either, which is why a training attempt with a negative reward given to the agent upon firing the HealBolt at an enemy was launched – this worked quite well, the bolts were not shot at the wrong target anymore at all, meaning that the targeting mechanism itself actually worked.

The only slightly negative observation was, that the bolt was fired quite rarely – about once every 30 seconds, regardless of the ability only having a cool down of two seconds.

The image shows the red healer shooting a HealBoltProjectile at its warrior ally.



Image 34: Red healer agent shooting a HealBoltProjectile at its teammate

## 7 Results

While most of the results were already covered along with the trainings in the previous chapter, a short overview of the progress along the path shall be given in this.

Over the course of the term, the AI-controlled characters have certainly come a long way. At the start of the term spent working on the thesis-project, there was only a single type of agent, often found wandering around in a way that could be considered random, a variety of different, specialized agents were realised by the end of it.

While estimations tended towards taking a longer time to get started with the first trainings, the capabilities that would be achievable by the agents in a quick manner thereafter were certainly overestimated.

A first few trainings made it quite clear that the training times seen with reinforcement learning in unity would soon become multiple times longer than ones seen with different machine learning problems such as supervised sample classification.

One of the reasons for this is certainly having a way more complex decision mechanism, evolving around the prediction of an optimal path for the agents to take, where the otherwise often one-dimensional classification result stands opposed to one with the dimension of the action vector's size.

Another reason is that, seeing the agents would reinforce their policy by applying it during training-gameplay, each update to it would eventually lead to an action, bringing with it a variety of new states, which could be interpreted as new samples, which probably were not seen before.

The direct application of actions also results in a variety of mechanism being triggered in the game environment, mechanisms that, at some point, have to result in a visually represented update, which in turn costs a lot more computation power compared to a simple weight-vector update seen with other machine learning challenges.

Albeit the agents improved a lot over time and it was possible for a few special situations to be looked at, a smoothly moving agent, which would deliver an efficient, aesthetically pleasing gameplay, as dreamt of before the start of the project, was not very achievable in the short amount of time, especially with the limited computing-power available – not without adding more logic to the agents at least.

In other words, it would have been possible to hardcode a variety of actions to be taken in certain situations, while the AI simply decides which of these actions the character should take.

Instead of giving the agents direct access to the characters' movement, it would, for example, be possible to let the AI decide between moving towards the enemy for five seconds or moving away and firing a firebolt.

Overall, this would lead to a rather large set of abstract actions which, based on a given situation, could then be logically reduced to a smaller set of that make sense. The AI being simply able to choose which one of these remaining actions it wants to take would be much less in control of the character, seeing it would be a human being that defines the logic filtering out the "good" actions.

While certain steps to simplify the amount of possible decisions the agents could make were taken, such as having abilities shoot at a chosen target rather than a random point in space defined by two of the agents' action vector values, generating a complex logic to reduce the AI-involvedness in the decision-making process of the agents was opposing the goals of this thesis and was hence chosen not to be worked towards.

Even if the agents did not achieve the dream standards, the improvements on the environment, the agent scripts and the training settings helped them overcome a variety of hardships come across in the time spent working on them.

Over the term, the agents' aiming and dodging capabilities and their damage (or healing) output improved greatly – also, problems faced, such as the agents running into corners or being too afraid to attack their enemies, were put in check.

The general improvement of the environment and training setup was clearly observable with the implementation of the healing agent.

Coupled with the experience gained during the research, it took far less effort to bring the healers to a performant state than with the previous agents.

With this in mind, the implementation of a further agent variation such as a tank, which would be focused on protecting its allies for example or to increase the complexity of the game by adding certain objectives to it, should all be overcomable challenges, even if it meant implementing other things that are not directly connected to the AI.

## 8 Excuse: Open AI Five

Open AI Five is the AI famous for its results achieved in the multiplayer online battle arena (moba) game “Defense of the Ancients” (DotA), including the victory over many professional players in an 1v1 setting and over the international champions of 2018 (5v5). During the time spent working on the bachelor thesis project, it often served as an example of a large-scale project regarding AI implementation to video games. Because of this and its amazing achievements, as well as for comparison purposes, a small overview shall be given at this point. [19] [20] [21]

In general, the way open ai five was trained is quite similar to this project's – the massively larger computing power available however, allowed the AI to be able to collect way more observations and commit more actions. Open AI, too, was trained without human play data, directly from scratch with randomized parameters, using the PPO algorithm as well.

In comparison to most of the agents in this project with an observation vector of four, the one used for DotA receives a list of twenty thousand. The AI mainly receives its information by comparing the game world to a default state on each step – a difference caused, for example, by a player standing in the observation zone, leads to information about its location and movement. The list also includes a variety of stats, however, such as the various characters' health and how it changed over the past 12 frames, type of character (DotA has over 100 different heroes to be played), its abilities, the direction it faces, what buffs affect it, distance and many more values to determine the state of the game. [22] [23]

In a moba like DotA, team play is probably the most important requirement for victory – while players communicate with each other, Open AI Five made use of a hyperparameter called TeamSpirit.

The hyperparameter's purpose is to control the balance of the agents' focus on their personal versus their team's rewards earned. [22] In comparison, there is not really a direct way to enforce team play in the thesis project.

According to the team's paper on its AI, there are between 8000 and 80000 possible actions the models are able to choose from, depending on the hero controlled. The configuration is set to an action on every fourth step / frame. Further, they have the capability to keep a large experience buffer, storing every action taken in a pair with the observation leading to it. This buffer is then fed back to the AI, meaning that it has a vast number of samples to help with its decisions. [24]

Their paper's conclusion paragraph also confirmed the main experience made while working on this, in comparison, tiny project – the importance of being able to scale up. With mostly 16 arenas, agents trained for roughly two hours and a training speed increase of ten times, the agents in the thesis often had to reach a performant point with a total training of 640 hours (seeing there are at least two



of the same agents in play at each time. Open AI Five on the other hand learned over 10000 years, which is about 131.5 thousand times as long. [22]

To achieve this, they had to run trainings with massively more instances over a longer time. But even if this amount of training were possible for this project, the complexity of the observation and action space remains almost ungraspable. This required sixty thousand CPU cores for their simpler 1v1 variant and 128 thousand for the one capable of 5v5 matchups. For both variants 256 performant GPU's of varying specializations were used. [22]

## 9 Conclusion

While by far not having the same amount of computing resources, time and skilled personnel available to improve and adapt the broad variety of aspects required to establish an environment that would give birth to a complex and performant AI, the work on the thesis has been very interesting to say the least, especially learning about how, even if not all, many parts of reinforcement learning applied to a game would play together.

In retro perspective, the decision to start with a very simple AI turned out to have been spot on. With the results of the project and the things learned during it in mind, more complex and performant AI's would certainly be achievable with the given infrastructure to train it.

All in all, the choice to keep the focus of the project work goal oriented – to continuously improve the AI over the term while exploring but a limited few research questions about bias and agent stability through alterations in the game environment, turned out to be a good idea, even if there were many more interesting questions and tasks that could have been tackled, such as fine tuning the trainings with a more in-depth focus on hyperparameter optimisation, training more complex agents over larger periods of time or even less AI-related tasks such as creating an interesting game they agents could be applied to.



## 10 Sources

### 10.1 Web sources

- [1] "Open AI - PPO," [Online]. Available: <https://openai.com/blog/openai-baselines-ppo/>. [Accessed 26 05 2020].
- [2] "Arxiv Insights - Youtube tutorial "an introduction to reinforcement learning"," [Online]. Available: <https://www.youtube.com/watch?v=JgvzlkxgxF0>. [Accessed 26 05 2020].
- [3] "Flappy Bird," 18 05 2020. [Online]. Available: <https://flappybird.io/>.
- [4] "ML-Agents Repository," 21 05 2020. [Online]. Available: <https://github.com/Unity-Technologies/ml-agents>.
- [5] "Installation of ML-Agents," [Online]. Available: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.0/docs/Installation.md>. [Accessed 17 02 2020].
- [6] "Unity.com," [Online]. Available: <https://unity.com/>. [Accessed 09 06 2020].
- [7] "Getting Started," [Online]. Available: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.0/docs/Getting-Started-with-Balance-Ball.md>. [Accessed 21 05 2020].
- [8] "Learning examples," [Online]. Available: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.0/docs/Learning-Environment-Examples.md>. [Accessed 21 05 2020].
- [9] "Roller Ball," [Online]. Available: <https://github.com/Unity-Technologies/ml-agents/blob/0.14.0/docs/Learning-Environment-Create-New.md>. [Accessed 21 05 2020].
- [10] "Create a new environment - Roller Ball," [Online]. Available: <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Create-New.md>. [Accessed 26 05 2020].
- [11] "TensorFlow," [Online]. Available: <https://www.tensorflow.org/>. [Accessed 10 06 2020].
- [12] "Training agents," [Online]. Available: [https://github.com/Unity-Technologies/ml-agents/blob/latest\\_release/docs/Training-ML-Agents.md](https://github.com/Unity-Technologies/ml-agents/blob/latest_release/docs/Training-ML-Agents.md). [Accessed 21 05 2020].
- [13] "Trainer Config," [Online]. Available: <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Configuration-File.md>. [Accessed 21 05 2020].
- [14] "Unity GameObjects and components," [Online]. Available: <https://docs.unity3d.com/520/Documentation/Manual/TheGameObject-ComponentRelationship.html>. [Accessed 21 05 2020].
- [15] "Designing Agents," [Online]. Available: <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Agents.md>. [Accessed 21 05 2020].
- [16] "Unity Documentation," [Online]. Available: <https://docs.unity3d.com/ScriptReference/Transform-localPosition.html>. [Accessed 21 05 2020].
- [17] "SAC algorithm," [Online]. Available: <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-SAC.md>. [Accessed 04 04 2020].
- [18] "Unity Forums," [Online]. Available: <https://github.com/Unity-Technologies/ml-agents/issues/1417>. [Accessed 21 05 2020].
- [19] "Wikipedia - OpenAI Five," [Online]. Available: [https://en.wikipedia.org/wiki/OpenAI\\_Five](https://en.wikipedia.org/wiki/OpenAI_Five). [Accessed 18 05 2020].
- [20] "DotA," [Online]. Available: <https://blog.dota2.com/?l=english>. [Accessed 21 05 2020].
- [21] "Wikipedia: DotA," [Online]. Available: [https://en.wikipedia.org/wiki/Dota\\_2](https://en.wikipedia.org/wiki/Dota_2).
- [22] "Open AI Five blog," [Online]. Available: <https://openai.com/blog/openai-five/>. [Accessed 21 05 2020].
- [23] "neuro," [Online]. Available: <https://neuro.cs.ut.ee/the-use-of-embeddings-in-openai-five/>. [Accessed 21 05 2020].
- [24] O. A. Team, "Dota 2 with Large Scale Deep Reinforcement Learning (arXiv:1912.06680)," 2019.

## 10.2 Literature

[1] Open AI Paper on Open AI Five: “Dota 2 with Large Scale Deep Reinforcement Learning (arXiv:1912.06680),” Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębniak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, Susan Zhang, 13.12.2019

## 10.3 Tutorials

[1] ML-Agents tutorial by Unity:

<https://www.youtube.com/watch?v=32wtJZ3yRfw&list=PLX2vGYjWbI0R08eWQkO7nQkGiicHAX7IX>

[2] Tutorial regarding the setup of ML-Agents by Irene Gironacci:

<https://www.youtube.com/watch?v=MQErGQWEIAk>

## 10.4 Figures

Figure 1: Project Overview.....	4
Figure 2: Components Overview.....	9

## 10.5 Images

The flappy bird image was taken from: <https://techfruit.com/2014/02/09/popular-flappy-bird-game-shut-tomorrow-says-developer> (page accessed on: 17.05.2020)

Image 1: Flappy Bird .....	6
----------------------------	---

The roller ball image was taken from: <https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Create-New.md> (page accessed on: 26.05.2020)

Image 2: Roller Ball .....	7
----------------------------	---

The following images show parts of ML-Agents, the screenshots were taken on the author's personal computer, however the images' contents belong, aside from configuration details, either to Unity or to the ML-Agents team, as a part of Unity Technologies.

Unity Technologies: <https://unity.com/>

ML-Agents: <https://github.com/Unity-Technologies/ml-agents>

Image 3: Trainer configuration file .....	10
Image 4: ML-Agents training command .....	11
Image 5: Behaviour Parameters in the Unity inspector .....	12
Image 6: Decision Requester in the Unity inspector .....	14
Image 7: AgentReset Method .....	15
Image 8: Agent Parameters in the Unity inspector .....	15
Image 9: CollectObservations Method of an Agent script .....	16
Image 10: AgentAction example of the WarriorAgent .....	17
Image 15: Unity hierarchy showing the arena prefab's structure .....	20
Image 16: Unity inspector showing the components of a character game object .....	20
Image 17: Screenshot of the gym consisting of 20 arena instances .....	21

The following images and the one on the cover are screenshots of the game, or its written code, designed by the author of this document. Code snippets may contain parts that were taken over from the ML-Agents package or documentation. The screenshots of the game environment, to be specific, the entrance "door" seen on image 18, the trees and the sprites used for the characters are copies of the ones seen in the game, "The Legend of Zelda", which belongs to Nintendo.

Nintendo: <https://www.nintendo.com/>

Image 11: Boolean trigger showing the healer's targeting mechanism .....	18
Image 12: Ways to shoot a FireBolt .....	18
Image 13: The Arena in its various states .....	19
Image 14: Unity hierarchy showing the gym with its arena prefab children .....	19
Image 18: Runbot and door as objective .....	22
Image 19: BattleBot - Agent .....	23
Image 20: The MultiSlash ability .....	23
Image 21: SelfPlay parameters in the trainer config file .....	24
Image 22: Result of first trainings using self-play .....	24

Image 23: Agents shooting at fixed spawn locations.....	25
Image 24: Agent rewards in the HealthController's TakeDamage method.....	25
Image 25: 1v1 to 2v2 gameplay sequence.....	29
Image 26: Teammates acting as a unit.....	30
Image 27: Arenas showing the warrior and magician agents during gameplay.....	31
Image 28: Agents avoiding randomly reappearing trees.....	33
Image 29: Red agent going for a clear shot.....	33
Image 30: No bias regarding tree locations.....	34
Image 31: Arena with spawn zones added to all sides.....	35
Image 32: HealWaveProjectile returning to its caster.....	36
Image 33: Healers facing off against each other.....	36
Image 34: Red healer agent shooting a HealBoltProjectile at its teammate.....	37

## 11 Declaration of Originality



### Erklärung der Diplomandinnen und Diplomanden *Déclaration des diplômant-e-s*

#### Selbständige Arbeit / *Travail autonome*

Ich bestätige mit meiner Unterschrift, dass ich meine vorliegende Bachelor-Thesis selbständig durchgeführt habe. Alle Informationsquellen (Fachliteratur, Besprechungen mit Fachleuten, usw.) und anderen Hilfsmittel, die wesentlich zu meiner Arbeit beigetragen haben, sind in meinem Arbeitsbericht im Anhang vollständig aufgeführt. Sämtliche Inhalte, die nicht von mir stammen, sind mit dem genauen Hinweis auf ihre Quelle gekennzeichnet.

*Par ma signature, je confirme avoir effectué ma présente thèse de bachelor de manière autonome. Toutes les sources d'information (littérature spécialisée, discussions avec spécialistes etc.) et autres ressources qui m'ont fortement aidé-e dans mon travail sont intégralement mentionnées dans l'annexe de ma thèse. Tous les contenus non rédigés par mes soins sont dûment référencés avec indication précise de leur provenance.*

Name/Nom, Vorname/Prénom

Sebastian Werthemann

Datum/Date

22.05.2020

Unterschrift/Signature

S. Werthemann

Dieses Formular ist dem Bericht zur Bachelor-Thesis beizulegen.  
*Ce formulaire doit être joint au rapport de la thèse de bachelor.*