Deep learning in FPS games

Enabling players to interactively train unique behaviours of deep learning-based game characters in first-person shooters

---

Luca Bruder

AI8 – Serious Games
University of Bayreuth

May 31, 2022

Supervisors:
Miroslav Bachinski, Florian J. Fischer

Abstract

By now many fields greatly benefit from machine learning. One that thus far has not really explored the full potential of machine learning however is video games. While in some niche cases it is used to test for bugs or generate levels and worlds it has largely been ignored in the broader sense.
In this thesis we seek to explore the potential of machine learning for controlling non-player characters to learn deep strategies and behaviours. To this end we implement a machine learning agent in the *Unreal Engine*, let a human player interact and train with it, and then let the agent play against other agents trained in this way.

Contents

1 Introduction

> *Before we work on artificial intelligence why don't we do something about natural stupidity?*
>
> — **Steve Polyak**
> (American Neurologist)

1.1 Motivation

Non-player characters have been a staple in video games for decades. And while sometimes referred to as "AI's" in most cases these agents do not learn or show any deep behavioural patterns. Usually they follow simple behaviour trees and easy to understand rules. Hence it is a simple task to distinguish between human players and non-player characters.

In this thesis we seek to develop a deep-learning agent that challenges these structures. The agent is supposed to be actually trainable by the player and develop human-like strategies and behaviours. Since I am most comfortable with developing first-person action games this is the genre that will be used. One player will train an agent which will then play against an agent trained by a different player.

1.2 Implementation details

Since the aim of this thesis is to quickly develop and test an agent and see whether current state of the art algorithms can be used to achieve this, actually developing a machine learning algorithm will not be part of this project. We will rather use a given framework to speed up the process and focus on adjusting hyper-parameters, reward structures, observations and controls of the network.

To enable player interaction with the system and actually visualise the game in a natural environment the *Unreal Engine*[1] is used to develop a prototype. This game engine was picked due to my personal experience with the engine as well as its versatility and flexibility. The *Unreal Engine* is a complete suite of creation tools for

anything from games, over simulations to film productions. It is based on C++ but also supports so called "Blueprints", which is their version of visual coding. For this project version 4.27 of the engine is used.

Besides many quality of life features the engine provides a market place where users can share code, assets and much more. One of these products is a machine learning framework, which will be the basis of this thesis.
The framework in question is Mindmaker [2]. It provides an easy to use interface as well as a Python implementation of Stable Baselines [3], a set of ready to use reinforcement learning algorithms.

1.2.1 Interface between *Unreal Engine* and Python

To bring the project together the mentioned Python implementation needs to communicate with the *Unreal Engine*. This is done with a simple IO connection using TCP/IP protocols with the Python code as the sercer and the *Unreal Engine* part as client.
The initial setup is handled in the *Unreal Engine*. That entails setting up the environment, loading the map, spawning actors, setting starting positions and so on. These initial values are then send to Python using the IO connection. Along side these values we send the hyper parameters for the used algorithm such as learning rate, network structure and observation structure.

Python then builds a network to the given specifications. The first output of the network is then send back to the *Unreal Engine*. Here we apply the selected actions (like moving and rotating the agent), calculate and visualise new positions, rewards and obeservations and send the results back to Python. This continues for the length of the training.

1.3 Iteration process

Using this framework will ideally make the process of iterating different version fast and easy. The goal would be to start with a basic task, like locating and getting close to the opponent and then expanding on that with aiming and shooting.
We will iterate in small steps and one change at a time to minimise the risk of miss diagnosing the error source.

1.4 Research question

As briefly mentioned earlier the prototype will be a first-person action game. The goal will be simply to shoot and kill the opponent without dying yourself. We want to spawn an agent, that has been pre-trained to a basic level, in a map[1] together with the player. Here the agent is then trained with new strategies from the player. This might for example be flanking the opponent, using walls to your advantage to shoot but not get hit or similar strategies.
Ideally the agent will either learn to use these strategies himself or defend against them. To enable a meaningful change in behaviour the learning rate of the agent will be increased in this form of training in contrast to the initial self-training.

In the end different agent will play the game against each other and hopefully the one that has been trained by a better or more strategic player will win. Our primary interest is in researching the potential of machine learning in a video game context, since it has not been explored very deeply before. The question we hope to answer is whether it is feasible to have a player train a machine learning agent in real time and without further self-play. This is interesting, since usually it takes many episodes for an agent to show meaningful behaviour. If we could have an agent learn new strategies from a player in a reasonable training session this could pave the way for a new way of implementing non-player characters in video games in general.

1.5 Disclaimer

The following chapters will show the detailed process of iterating different versions, adding and removing features and discovering bugs. Since I am fairly new to the field of machine learning quite a bit of this process was me getting to know the intricacies of the subject. This lead to some avoidable mistakes and initial troubles. Some of the discussed problems should therefore not be regarded as problems with the approach in general, but rather necessary steps in learning to work in the field of machine learning.

[1] Map in this context refers to a video game environment both player and agent can move in.

2 Related works and planning

> *It's so much easier to suggest solutions when you don't know too much about the problem.*
>
> **— Malcolm Forbes**
> (American Entrepreneur)

In this Chapter we will look at a few examples of how deep learning can be used in the context of playing video games, what results were achieved and how this might relate to our project.

2.1 General deep learning approaches for Atari games

In [4] Anoop Jeerige et al. compare two different general deep learning approaches against each other. The algorithms in question are DQN and A3C. While relevant to their work, concrete implementation details and inner workings of these algorithms are not the focus of this thesis. They will thus not be discussed.
In their paper they used both algorithms to train playing the game "Breakout" for the Atari 2600. Both were presented with the same inputs (environment observations like position of the controlled platform and the position of the ball). With this they had to learn control policies to achieve a high score. A screenshot from the game and both agents can bee seen in Figure 2.1.

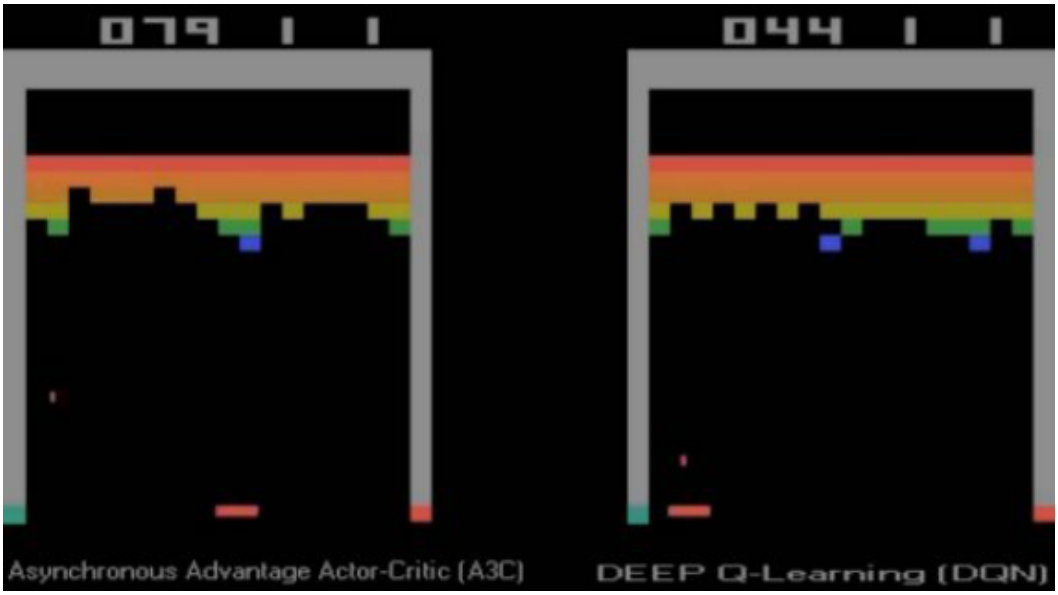


Fig. 2.1: Screenshot from the Breakout game results for DQN and A3C [4]

In the project they encountered a number of roadblocks which might help us better plan our own project. One of the biggest roadblocks was engineering knowledge about the environment and deciding which variables to observe. To overcome this a simple interface as well as monitoring tools were implemented. These tools and interfaces helped observing and adjusting certain variables and maybe exchanging observation variables but brought another issue to light. The cycle of designing, implementing and testing took too long. With each adjustment a new agent had to be trained in a lengthy process due to the complex environment. The solution was choosing a simpler environment and better understanding the reinforcement learning algorithms at work.

With these roadblocks overcome both algorithms were able to produce an agent that could play reasonably well and achieve a decent score. Of the two A3C did better achieving a high score of 79 while DQN only achieved 44. Furthermore A3C trained faster and more stable.

Playing Atari games has been a staple of machine learning approaches in the field of video games. Other works include [5] which considers some more DQN-related algorithms and [6] which proposes evaluation methodologies for comparing differently trained agents. It is also notable to look at papers trying to improve upon previous results by increasing the adaptability of the DQN approach [7] as well as decreasing the learning time [8].

2.1.1 Consequences for our project

The most important take-away of this paper seems to be the importance of a simple environment and the ability to test many iterations in a short amount of time. Since our goal is quite a complex task involving not only moving but also aiming, shooting and even developing tactics it seems all the more important to start out with a simple environment and work on incremental tasks.
Lastly we can speculate about the amount of training necessary to achieve reasonable results. In this paper both agents showed relatively good behaviour after about 500.000 taken actions. After that both still improved a bit with DQN sometimes getting worse with more training. Again with our more complex task one might assume that we need more training, but for earlier iterations with less complexity it might be possible to achieve results between 500.000 and 1.000.000 actions taken. Though further tests will without a doubt be necessary.

2.2 Deep learning for the first-person shooter game Doom

While the approach for Atari games shows general applicability to a video game context we will now look at a more specific example. The paper we will discuss again examines different approaches and even different genres. We will focus on the first-person shooter genre.
Within this section of the paper we see a discussion of an AI Doom competition. A more thorough examination of the competitions from 2016 and 2017 [9] explores the different approaches further than we will do in this section. While many competitors used A3C[10][11] and DQN, some used variants of DQN like DRQN [12] and DFP [13].

In the article we will focus on [14], Niels Justesen et al. discuss different recent deep learning approaches playing video games in general. From this discussion we can take away a few interesting points. Firstly, back in 2016 Kempka et al. [15] proved that an agent trained using DQN was able to achieve human-like behaviour.

Secondly we again compare different algorithms but this time in form of a competition in which different teams using different approaches and algorithms trained agents to play Doom against each other. In this competition DQN and A3C both performed near the top of all competitors. The competition was a Deathmatch[1] between all agents. Each of these agents had been trained with the same visual

[1] All agents are spawned in an arena and try to kill each other. The agent with the most kills at the end of the round wins the game.

inputs, using each pixel of the screen as one input.
While both algorithms performed well, A3C outperformed DQN again though by a smaller margin than in Section 2.1. Both were however outperformed by direct future prediction (DFP). A video of the whole match can be seen on YouTube [16] and a screenshot has been included in Figure 2.2.

Fig. 2.2: Screenshot from Visual Doom AI Competion in 2016 [16]

Lastly we can extrapolate some more information and predictions for our own training. For DQN and A3C training was done in increments with only small tasks each training like collecting a medikit. While perfect results for single episodes could be observed fairly quickly, these were countered by steep drops in performance in the next episodes. The average results however gradually improved and was at a good level after about 1.000.000 steps.

2.2.1 Relevance for our approach

We will now quickly go over the main three points we have discussed in order and see what lessons can be learned for our project. Firstly while human-like is not necessarily the specific goal of our project[2] seeing that DQN is generally able to achieve this level of behaviour seems to encourage the use of DQN in our project.

The second point shows another comparison between DQN and A3C. This time however both performed more equally than they did in Section 2.1. This suggests

[2]Meaning that human-like behaviour would be a byproduct rather than the final goal since we aim to enable players to train agents with their own tactics.

that the distinction between the performance of DQN and A3C is not as clear as initially thought. This further encourages the use of DQN and maybe a later comparison to A3C.

The last point concerns the amount of training we will likely have to do for our own approach. While training in small incremental tasks seems to be a good idea for our approach as well we do not plan on training visually and it is therefore likely that we will be able to train these increments much faster due to our smaller observation space. Where networks for the competition took thousands of pixels as inputs we will probably only need to observe about 20 different variables.

2.3 Self-play for fighting games and the impact of bigger action spaces

Lastly we want to take a look at two equally short but nevertheless valuable papers. In the first one Seonghun Yoon and Kyung-Joong Kim [17] explore the impact that compounding different actions can have on an agent. In this example a DQN agent was trained to play a fighting game[3] against a static opponent.
The agent had either 13 or 22 possible actions he could perform. In fighting games many actions are state-sensitive. Meaning that an action might results in different outcomes depending on the state of the player. If for example a player throws a punch while standing on the ground that differs from a punch while being in the air. While the first case just leads to the character performing a punch the second case might lead to the character prematurely ending its jump. Or in more general terms: an action can lead to different successor states and therefore rewards, depending on the current state.

While training with 13 actions the agent did not have separate outputs depending on states. A "punch" output was always the same regardless of the players state. For the training with 22 actions the author distinguished between for example an "air-punch" and a "ground-punch". The impact of these compound actions is somewhat surprising. On the top-end the agent using more actions outperformed the other one by a significant margin. On average however the agent using less actions performed better. This was also true for the bottom-end, where the agent using more actions performed a lot worse.
This suggests that compounding actions can be a good idea in some instances (especially when high performance is required) but is likely not necessary. It is

[3] Fighting game in this instance describes something similar to games like Street Fighter, Tekken or Mortal Kombat, where a player moves in two dimensions and chains together different inputs to perfom combo attacks and beat his opponent.

however wise to keep this in mind and try it out should the need to distinguish between actions in different context arise in our project.

And finally in the last paper we will take a look at how Yoshina Takano et al. [18] examine the impact of self-play and specifically deciding which agent to play against during self-play.
Self-play describes the practice of training an agent against iterations of itself in order to improve. A big benefit of this kind of training is that neither data sets nor external training partners (behaviour tree based agents, human players or other) are required. The comparison the authors make is between an agent that exclusively trained against the latest iteration of itself and one that trained against a random iteration of itself from a selected set. Both agents were again trained using DQN and played a fighting game.

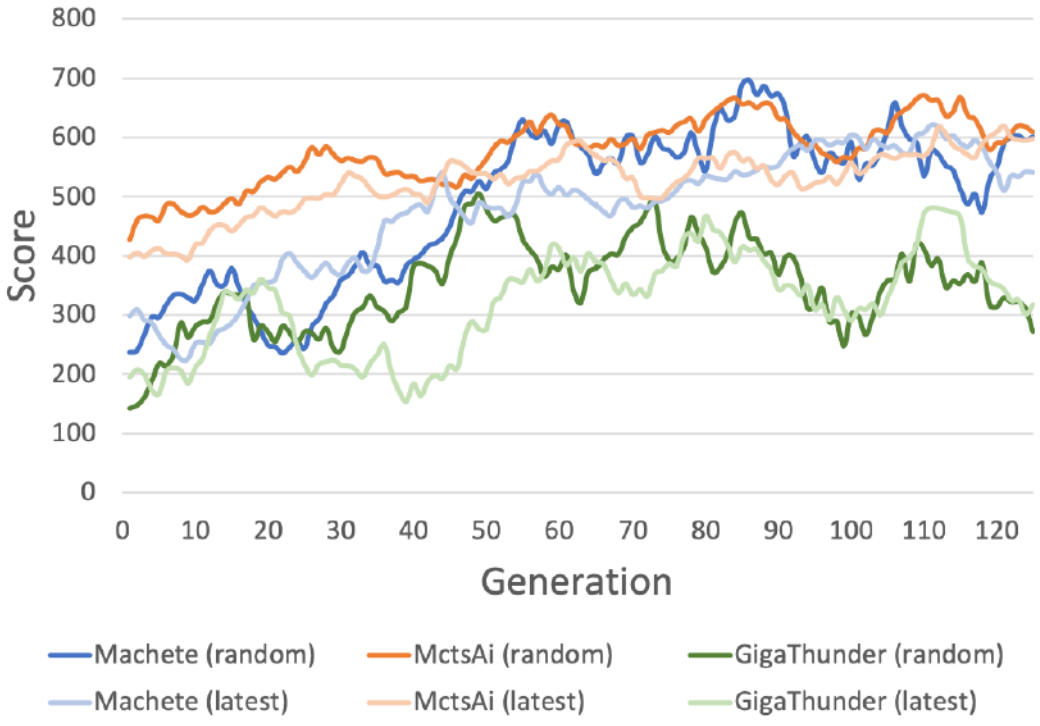


Fig. 2.3: Scores for approaches training against random vs. latest iterations [18]

The achieved results (Figure 2.3) suggest that training against random iterations performs always either similar or better than only training against the latest iteration. This is likely due to cementing locally optimal behaviour quickly and discouraging the agent from exploring. If we simplify the fighting game to something like rock, paper, scissors this can be seen quite easily.
In early iterations the agent might learn that picking rock often results in a win. It thus starts to pick rock more often and with each iteration it picks rock more. Since the iteration it plays against also already uses rock a lot picking rock again results in

a draw. While not optimal this at least does not lose the agent any points. Since the opponent will not pick anything else either both might get stuck just picking rock. Playing against a random iteration however might results in the opponent suddenly picking paper. This would encourage the agent to explore new options since it is now actively losing points.

These problems can be circumvented by using appropriate learning and exploration rates as well as shaping the reward function more precisely. Also the problem is exaggerated and would play out differently in reality. It does however illustrate the point of training against older iterations quite well.

But what does this have to do with out approach? While not necessarily directly applicable to this project (depending on how we will train the agent) it does show the potential of outside influences and suggests the possible value an interaction with a human could have. The human could introduce strategies the agent has not come across during training and thus influence its' training for the better.

2.4 Planning

With these references we can now start to plan the actual project. While certain details like tools that will be used have already been laid out in Chapter 1.2 we need to work out the specifics of training stages.
The related works that we have discussed all have one major difference to our approach. They trained visually. This leads to many more inputs to the network than this project will have. It is therefore likely that our training will produce meaningful results much sooner. However letting a human play against an entirely untrained network will still take too long. The plan will be to train a generic network to a point where it plays reasonably well against opponents that are not using any complicated strategies.

For this training the network will work on a fairly low learning rate. This is to prevent learning local optima too fast and hopefully learning a robust and versatile behaviour. When this agent is fully trained it then competes against a player while using a much higher learning rate. We are not as concerned with local optima at this stage, since the agent is already pre-trained. Also we can not expect a human to play hours against the network to see any results.
Once these two stages are completed the agent will play against other agents trained in the same manner. This will purely be for evaluation and the agents will not learn from their encounters.

Besides getting an idea of how long we will have to train the agents for the related works also shed some light on which algorithms to use. The Mindmaker toolkit that we will use enables us to easily swap between many already implemented reinforcement learning algorithms.
Two algorithms which consistently produced good results in similar projects were DQN and A2C (or A3C). Of the two we will start by testing Deep Q-Networks (DQN)[19]. These have been around since 2013 and were initially developed with playing Atari games in mind. Even though many of these games are different in genre, perspective and controls to our project, on a fundamental level their goals should align with ours.

The second algorithm we will take a look at is a newer approach that produced results more quickly (and sometimes better) than DQN. Advantage Actor Ciritc (A2C) was initially presented in 2016 [20] and is a synchronous alternative to the similar A3C. In many situations it produces better results than DQN. In the specific examples discussed earlier however DQN lead to mostly comparable results but sometimes more required training.
Since DQN still is used alot for similar tasks we will start by using it. If we either can not produce good results with it or have enough time at the end of the project we will also use A2C and compare the two.

Iterations, problems and solutions

3

> *If you torture the data long enough it will confess to anything.*
>
> **— Ronald Coase**
> (British Economist)

In this chapter an outline of the different stages of this project will be discussed. To improve readability as well as brevity only big changes will be discussed. Between all the discussed iterations are a large number of smaller changes (usually including a number of small changes to parameters like learning rate, position of characters, amount of episodes trained and so on). The section will therefore only discuss the final and best iteration of that specific change.

3.1 Initial setup

As briefly mentioned in chapter 1.2 we want to build an agent that can interact with the player (or any opponent for that matter) in a way that another player would in a first-person shooter game. To make this possible the agent needs some basic "senses" as well as controls that resemble the inputs of a human. Other than that we will need animations, models, and some other fundamental functionalities including shooting where the agent is pointing, collisions with the environment and physical forces like gravity. While without a doubt very important these latter points will not be discussed any further since they are of little impact to this thesis.

3.1.1 Senses

We will start by discussing the senses of the agent. For both hearing and seeing the *Unreal Engine* provides useful tools which can be attached to the agent and adjusted according to the desired specifications. For hearing an invisible sphere with a radius of 3000 units is cast around the agent. If anything moves within this sphere it emits a signal. This signal has an amplitude depending on distance. The agent then gets provided with the distance and direction of this signal. Which actions actually emit a signal can be controlled. In our implementation just shooting and walking will

make sounds. If a character is "sneaking" (walking in a crouched state) or simply not moving at all he will not emit any sounds.

Seeing works in a fairly similar way. Instead of a sphere we cast a cone in front of the agent. The radius and length of the cone can be adjusted. For now the agent can see over a distance of 5000 units (which covers the whole map) and has a field of view of 90°. The latter being oriented on usual field of view settings for first-person shooter games. If anything enters this cone (later referred to as "field of view") the agent knows its exact location.

3.1.2 Controls

In the field of first-person shooter games on a computer the player usually controls his character with four different inputs for movement (in most cases W,A,S,D), an input for shooting and sometimes additional inputs for jumping or crouching.
We firstly focus on movement. If a player were to press the forward input for example the character would move in the direction he is looking at for the duration the button is pressed. Ideally we would therefore let the agent move forward as long as he outputs the forward command. Within the *Unreal Engine* however this leads to some problems. It is very easy to trigger an endless loop exception and thus an exact mapping of what a player would do is not possible. We therefore use a small workaround. Every time the network outputs a forward command the agent will move to a point in a distance of 50 units in that direction. This will prevent the agent from stopping his movement before the next command is issued, will however on the other side allow him to overwrite this movement[1] with his next command. In practice the agent now has the same movement options a player would.

While some games allow diagonal movement others do not and since this will make the task easier we do not allow movement in two directions at the same time to create diagonal movement. We disable this on the player side as well so both parties are on an even playing field. So the network picks one of five discrete movement options, not more.

Rotation would be handled by a player by using his mouse. Contrary to movement, rotation is possible in two directions at the same time. A human player would be limited in the amount he can rotate precisely in a given timeframe. If we would allow the agent to rotate a full 360 degress in each timestep this would not be the

[1] Overwriting is possible in any other direction and will immediatly move the agent in the new direction. This also works for stopping the movement entirely by just picking a spot right at the location of the actor.

case for him. Because while a player could theoretically rotate that amount he could very likely not do it precisely. We therefore limit the agent in the amount he can rotate each step. This amount is initially arbitrarily set to a maximum 15 degrees in each direction per step.
Similar to movement the network outputs a discrete amount to rotate in each direction. The options are set in 1° increments so everything between -15° and 15° is possible.

Lastly there is an option for crouching. While this was not strictly necessary it seemed to enable more interesting behaviour to be able to crouch behind obstacles. This is an additional discrete output which either crouches or does not.

In summary the network picks one movement option, two rotation options and one crouch option each timestep. this concludes the controls for now but will be further expanded at a later point with options for example shooting.

3.1.3 Map

If we want to move characters in a meaningful space we also need a map. For this we initially just use the "starter map" that *Unreal Engine* provides with each new project. It includes one block of half height (a character can crouch behind), a stair leading up to a platform as well as four walls limiting the space. While this is not very interesting from a gameplay perspective it should suffice for some early testing. A picture of this initial map can be seen in Figure 3.1.

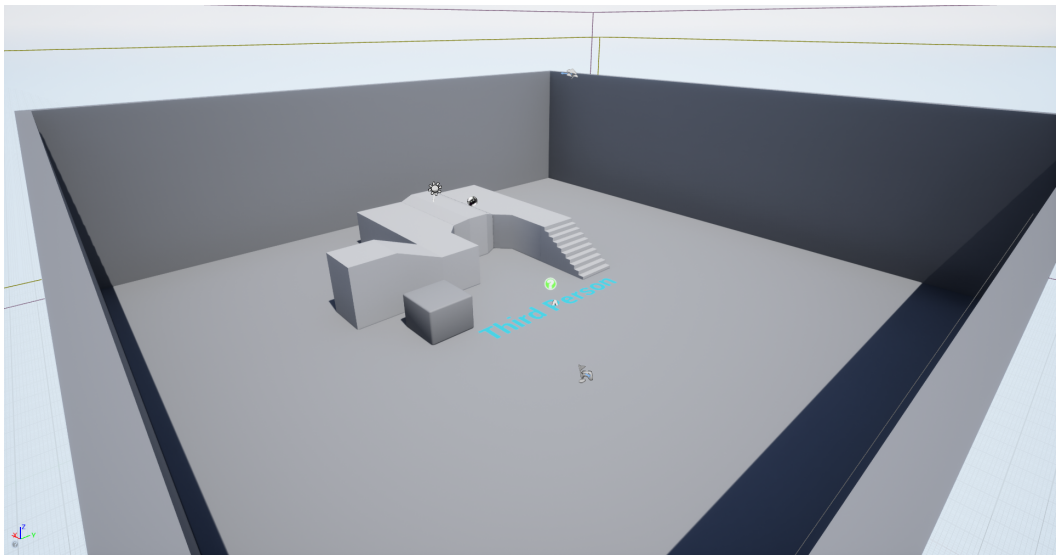


Fig. 3.1: Screenshot of the starter map

3.1.4 Observations and rewards

Next we need to figure out which variables the agent should be able to observe to give him as much information as the player, but not more. First and foremost he should observe his own position and rotation. He also can observe his own senses. They are represented in the form of two booleans for hearing and seeing respectively. He also observes his distance to the enemy. This only updates if he either hears or sees the enemy. Lastly he observes the location of the enemy, which similarly only updates when he sees said enemy. With these observations we hope to achieve a similar level of knowledge that a player would have.

We now need to shape a reward function to encourage our desired behaviour. In this first iteration we plan on having the agent simply find, see and chase the opponent. While this differs from our final goal in that getting close to the opponent will not be strictly necessary, it will still help the agent along the way.
Nonetheless we shape the reward with future iterations in mind. Seeing the enemy will thus be a requirement to receive points, since it is vital to shoot the opponent at a later point in time. The agent gets awarded 50 points for seeing the enemy and 100 for seeing **and** getting closer (if he gets closer without seeing the enemy no points are awarded).
Since we also want to encourage actively searching for the enemy we punish the agent with -10 points if he does not see the enemy and does also not move in a meaningful way. Meaningful in this case is defined by a distance to a set location. For example at the start of the game this location is set to the agents current location. If he moves 50 units away from that location in any direction it is meaningful. This 50 unit mark can however only be reached once. If he moves back to a distance of 20 units and then again to 50, that is not meaningful and will be punished. After reaching a total distance of 1500 units from this initial location (for reference: the map is 5000 by 5000 units), the location is again reset to the agents current location.

$$\text{Reward} = (s * 50) + (s * d * 100) + ((m * -10) + (s * 10))$$

$$s = \begin{cases} 1 & \text{if agents can see the opponent} \\ 0 & \text{otherwise} \end{cases}$$

$$d = \begin{cases} 1 & \text{if current distance to enemy is shorter than} \\ & \quad \text{previously reached shortest distance} \\ 0 & \text{otherwise} \end{cases}$$

$$m = \begin{cases} 1 & \text{if agent did not move in a meaningful way} \\ 0 & \text{otherwise} \end{cases}$$

3.1.5 Opponent

The initial setup for the opponent was quite rudimentary and is only supposed to assist us in tuning the reward function and fixing initial bugs. As such the earliest setup had the opponent moving to random locations on the map in a set radius. Upon reaching this location it would be a new destination and move there. This behaviour was at first not influenced by the agent.
As we ran into problems where the agent would not move at all we tried expanding on this behaviour by adding two new rules. Firstly the opponent would always pick a destination where it was further away from the agent than 1000 units. Secondly whenever the opponent came too close the opponent would stop what it was doing and pick a new destination which would be further away from agent then his current position.

While primitive this movement was deemed sufficient for these early tests where we mostly locked down the agents movement, rewards and observations.

3.1.6 Architecture and hyper-parameters

Concerning architecture and hyper-parameters we decided to start initial tests with the default values, since these were already proven to suffice for learning to play Atari games. This means that we used a network with two layers of 64 neurons each. A full list of the hyper-parameters can be seen in Table 3.1. For further iterations these parameters can be assumed to be identical unless explicitly changed.
While most of the listed parameters have been changed little if at all, we want to take note of the learning rate since it will play a big role in this project. As discussed earlier it is likely that different learning rates will be necessary depending on whether the agent is training independently or with a human player. In the latter case we will increase this learning rate to see results faster. Logging of the training was enabled and used where necessary.

Parameter	Value
gamma	0.99
buffer_size	5000
learning_rate	0.00025
exploration_fraction	0.1
exploration_final_eps	0.02
exploration_initial_eps	1.0
train_freq	1
batch_size	32
double_q	True
learning_starts	50
verbose	1
seed	None
target_network_update_freq	100
prioritized_replay	False
act_func	tf.nn.relu

Tab. 3.1: Standard parameters for DQN in Stable Baselines

With the focus of this project lying elsewhere not every parameter listed here will be discussed. A full explanation for each one can be found on the DQN help page from Stable Baselines [21]. One thing of note is that we disabled prioritised experience replay. This is disabled by default and therefore we did not list further parameters for this feature.

In a last step we needed to figure out how long the agent should to be trained for. We started by training for only 10.000 steps an increasing this number depending on the outcome of the training. By doing so we figured out two rough breakpoints for this current implementation. After about 100.000 steps the behaviour of the agent seemed to be relatively stable. Most of its' actions were coherent. The next breakpoint, 350.000 steps, ironed out the rough edges. All of the agents actions seemed to follow the same pattern and we could not observe any deviation.
Training even more yielded no further change in behaviour. We tried it with 700.000 and even 1.000.000 steps. Both sessions resulted in identical behaviour to the iteration with 350.000 steps. For most of the further iterations we thus stuck to 100.000 or 350.000 steps, depending on the level of behaviour we wanted to see. We only changed this up once the task got more complex and the training logs suggested that more training might be beneficial.

3.1.7 Problems

A big downside of the chosen framework was the time it took to train a new network. While making adjustments was, as expected, fast and easy, the training itself took about 7 hours for 350.000 steps. This came down to two major problems. Firstly the connection between Python and Unreal. The IO-Socket connection was using TCP/IP protocols and was therefore somewhat limited in its' speed.
The second and admittedly bigger problem was the simulation we where doing in Unreal itself. After each new action the network chose and send to Unreal, we had to move the agent in the game world, simulate the physics, wait for everything to have moved the appropriate amount and then send the new observations back. All of this was calculated in real-time. Changing this was problematic since we could easily change the speed the game was running at, but then we would not receive the actions from Python fast enough.

Since we had assumed that this initial step of the project would not take too long we accepted the long duration of each training for now. Speeding up the training would require building a whole virtual environment which would not have to be simulated in real time and then translating the agent back to a real-time implementation once we train against a human player.
In hindsight we can say that this was a mistake. We had assumed that getting the

agent to a reasonable level of performance would be much easier then it actually turned out to be. This misjudgement lead to some time loss in the first half of the project.

Beyond issues with training itself, this initial setup proved to be insufficient for the task at hand. Even though many different reward structures were tested the agent seemed to move very little if at all. The assumption at this point was that since the opponent moves randomly throughout the small map, the agent is able to get a decent amount of points without moving by just waiting for the enemy to run into his field of view.

3.2 Map and opponent changes

3.2.1 Detailed changes and reasoning

With these problems in mind we implemented three big changes. Firstly we increased the size and complexity of the map. We roughly doubled the size of the map and added obstacles that would occlude vision for the agent and allow for more complex paths through the environment. These changes decrease the chance of the enemy randomly running into the agent and make standing still a much less viable strategy. A screenshot of the new map can be seen in Figure 3.2.

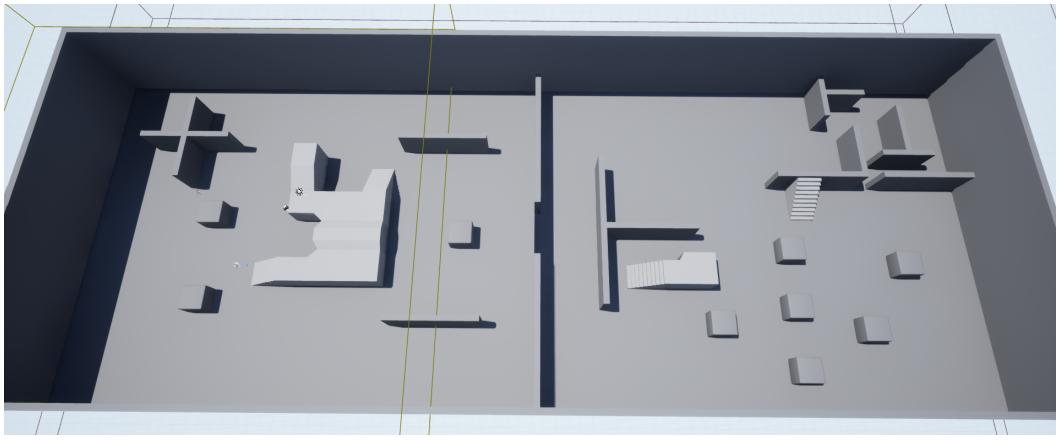


Fig. 3.2: Screenshot of the reworked map

Secondly we adjusted the behaviour of the opponent a bit more. It would be more reactive now and path according to its distance to the agent. Where possible it would avoid getting close to the agent and react to being seen with running away. This should further decrease the viability of standing still for the agent.

After a few iterations of this new setup we also adjusted the reward function some more. In early games on this new map the agent seemed to prioritise seeing the opponent over getting closer. We counteracted this with stopping to reward seeing the enemy and just rewarded the agent if he saw the opponent **and** got closer.

$$\text{Reward} = (s * d * 100) + ((m * -10) + (s * 10))$$

3.2.2 Achieved improvements

With these changes the agent showed first signs of tactical behaviour. It usually seeked out a spot near the middle of the map where the opponent could often be seen coming through. Since the opponent still had a tendency to get closer to the agent while moving to its desired location this yielded points in some instances. The agent also made sure to orient itself towards the center of the map.

3.2.3 New or persistent problems

This behaviour however still yielded mostly negative points since the agent stopped moving (asides from some games where he would just spin in circles). Regardless of how much we increased the punishment for standing still the agent could not be encouraged to move. Even worse in certain games he would just stand in a corner of a map while looking into said corner. This made receiving points impossible and was the worst behaviour the agent could show.

3.3 Testing varying episode lengths and simplifying the task

3.3.1 Detailed changes and reasoning

At this point we had no explanation for the bad behaviour of the agent. We therefore implemented quite a few changes to see how this would affect the agent. All of these changes were made in increments as to not lose information about which change actually had an effect on the agent.

We started by adding a new observation variable. We wanted to observe the last known location of the enemy as well as the actual position. If the agent actually saw the opponent both of these variables would be identical, but if he lost sight the actual position would be set to some big number (so the distance between agent

and opponent would be big and encourage the agent to seek out the enemy) and the last known location would just stay the same. We would then reward the agent with a few points for getting closer to this last known location. We imagined that this would lead to somewhat human behaviour.
In reality however the agent found a problem with this system. After seeing the opponent once he would turn away and thereby fix the last known location. He would then slowly walk backwards to this last known location and constantly receive points while actively avoiding seeing the enemy.

For this reason we adjusted the observations again. We removed the last known location variable and instead added a certainty. This certainty was 1 while seeing the enemy. The position of the enemy would be updated as expected in this time. If the agent lost sight of the opponent the position would stop changing and keep the value of the last location the opponent was seen at. The certainty however would go down by 0.2 each step until it reached 0 or the agent saw the opponent again. This would hopefully express to the agent a certainty of the opponent actually being in that position and remove the need for a last known location.

In a second change we made the opponent teleport instead of actually moving through the environment. It would teleport to a random location and wait there until the agent would come to close, at which point it would teleport again. This change was aimed at elimination the chance of the opponent getting closer to the agent without the agent actually doing anything.

Lastly we switched the used algorithm to A2C. As discussed in Chapter 2 this algorithm yielded better results in many cases.

3.3.2 Notable experiments

We also experimented a bit with different episode lengths and even network structures. The Mindmaker framework provided a few examples and some of them used an episode length of one. Which, while clearly being highly unorthodox, lead to good results in these examples. In the example an agent had to find a specific ball out of four balls. The agent just had to pick one action, moving to one of the four balls. This example used DQN.
At this point we need to mention that Mindmaker lacks documentation in a few vital areas. Most importantly the use of the "done" variable. In the Python implementation of Stable Baselines this parameter controlls the end of an episode. This parameter is however set in Unreal and suggest an end of the task. In Python it should reset the environment and agent but in Unreal it does not do this. Without the reset however it makes evaluating single episodes against each other nearly impossible. For the

given example a reset is not necessary since the round ends after a single action anyways. At the time this was however not obvious and we therefore just took this approach tried it for our project. As one would expect with the necessary context this resulted in even worse behaviour.
We experimented a bit further and landed on an episode length of 200 steps. This amount of steps is enough to walk across the whole map and find the opponent. But with the misuse of the "done" parameter we did not reset the environment after the end of this episode. The goal was a form of continual training. Since the opponent would teleport each time upon being reached anyways we wanted to just keep training. An explicit reset might discourage certain behaviour like reaching the opponent to fast or getting close enough for him to teleport.

Another idea we tested was one of external randomness. Since the agent would simply refuse to behave in any goal oriented way we tried randomly moving the agent every few steps during training in order to make him explore different options he was not using before. Many slightly different iterations of this yielded no difference to previous results however. The idea was therefore dropped.

We also wanted to test different network structures. Some online resources suggested a good starting point for a structure would be to use an amount of neurons per layer between the number of inputs (our observations) and the number of outputs (the action our agent can take). For this specific project this would results in 9 neurons in two layers. This is again unusual but seemed to be at least worth a try.
Interestingly enough the resulting agent showed almost identical behaviour to previous iterations with the added benefit of reaching this behaviour a lot faster. Fewer neurons drastically decreased the amount of steps we would have to take during training to converge on a certain behaviour. Without any change in behaviour we could not extract any relevant information from this experiment beyond the fact that the behaviour was likely not caused by a bad architecture. We therefore switched back to 64 neurons in two layers since more complex behaviour would require denser layers.

3.3.3 Achieved improvements

While these changes still did not yield the desired behaviour we managed to eliminate instances in which the agent would be reward for doing nothing or even doing bad. The biggest factor in this change was the opponent teleporting instead of moving. While this is not representative of our final goal it should improve the training.

3.3.4 New or persistent problems

The agent however still showed sub-optimal behaviour in most instances. It would now mostly run in circles while looking outward. This at least gave him a good chance of seeing the enemy every now and then but would still only yield negative rewards. While not being as bad as running into a corner we could not explain this behaviour. Further improvements were clearly necessary.

3.4 Switch to continuous action space with PPO2

3.4.1 Detailed changes and reasoning

At this point of the project we wanted to try a new approach and see if this would yield better or at least different results. We switched from a discrete to a continuous action space. This change would give the agent more control. The agent could now pick a normal vector to indicate the direction it wanted to move in as well as a number of degrees from -15 to 15 to indicate the direction it wanted to turn in.
The algorithm we used for this switch was PPO2. The new algorithm also made some new parameters necessary which can be seen in Table 3.2 Again the switch itself was easy due to Mindmaker having these algorithms available and ready to use. The change in control scheme resulted in a big change from previous iterations. Earlier the agent controlled just like a player would. Using the forward action would move it in the direction it would be looking. The other directions worked accordingly. This is identical to how most modern shooter games are controlled by humans. This new scheme however resulted in what is usually referred to as "tank controls". Movement and rotation were independent of each other. This means that the same movement action would always result in a move in the same direction whereas in previous iterations a forward action might result in any number of different move directions depending on the current rotation.

Parameter	Value
gamma	0.99
n_steps	128
ent_coef	0.01
learning_rate	0.00025
vf_coef	0.5
max_grad_norm	0.5
lam	0.95
nminibatches	4
noptepochs	4
cliprange	0.2
cliprange_vf	None
verbose	1
seed	None
act_func	tf.nn.tanh
n_cpu_tf_sess	None

Tab. 3.2: Standard parameters for PPO2 in Stable Baselines

We speculated that this change might positively impact the agent since it would be easier to draw a connection between action and change in observations.

Beyond this new algorithm we also tested different activation functions in hopes of impacting the early behaviour of the agent. In many instances the agent would start out not moving and all or only moving in certain directions. Using a hyperbolic tangent function instead of the rectified linear unit function solved this issue and overall resulted in better initial behaviour and quicker improvement.

We then simplified the task even further. After so many iterations with almost worst case behaviour it was clear that even just moving to the opponent was more difficult than anticipated. We therefore build a new map with roughly the same size as the first one but without any obstacles (see Figure 3.3). The opponent still only teleported, we therefore circumvented the earlier problem of the agent receiving points without doing anything.

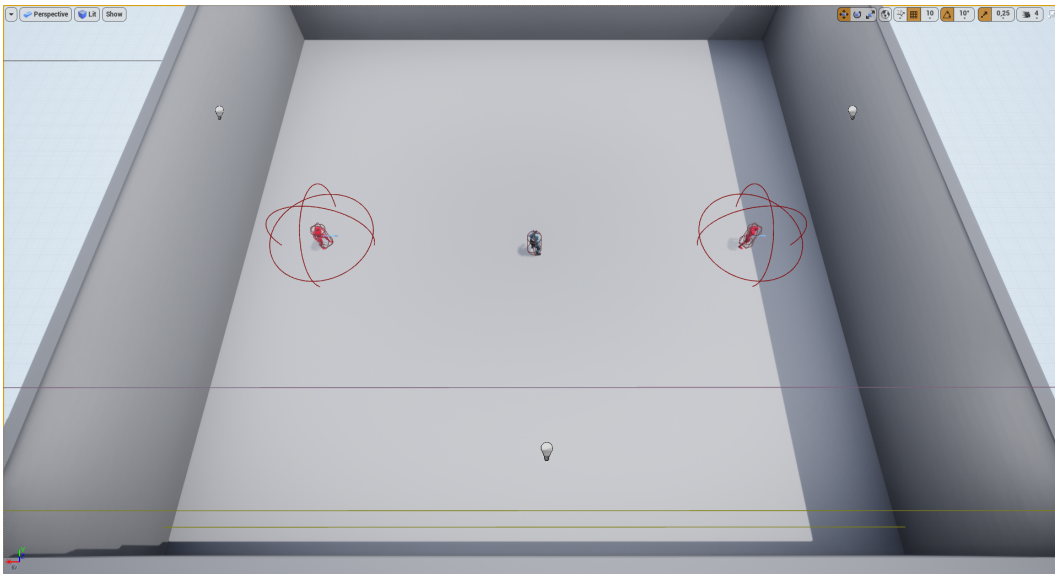


Fig. 3.3: Screenshot of the simplified map

Furthermore we changed the reward function again to better reward certain behaviour and provide a better gradient for the algorithms optimisation. We did this by awarding increasing points for getting closer to the enemy. We also removed the obstacle of only being able to reach each new distance only once by just constantly awarding points depending on distance. This should stabilise the training. The need for seeing the opponent was also removed for now as well as the punishment for bad actions. The reward function now looked like this:

$$\text{Reward}= \begin{cases} 200 & \text{if agent reached the opponent} \\ f(d) & \text{otherwise} \end{cases}$$

$f(d) = (250/d) * 100$ | With d describing the distance to the opponent

Reaching the opponent was defined as being within 250 units of the opponent. This acceptance radius can be seen in Figure 3.3 and is shown by the red spheres around each opponent. The distance (d variable) could be up to 5000 units large if agent and opponent stood in opposite corners of the map.
As the new reward function shows rotating was not necessary at this point as well. We therefore changed the outputs of the network to just movements without rotation.

With the smaller map and simplified task we also had to adjust the length of episodes. 80 steps were enough to reach any goal from any point of the map. We also implemented a hard reset at the end of each episode which would reset the agent to its initial position in the middle of the map and set the opponent to one of two random location in the map. To prevent early termination we always played out all 80 actions regardless of if and when the agent reached its goal.

3.4.2 Achieved improvements

These changes finally resulted in meaningful behaviour. At the start of each episode the agent immediately moved towards the right direction (right meaning wherever the opponent had spawned) and received increasing points. To the right location this worked fairly well. Upon reaching the acceptance radius however the agent did not stop but rather kept on moving past the opponent.

3.4.3 New or persistent problems

This good behaviour towards the right location was however not always the case. In some instances the agent completely diverged for seemingly no reason and started running in circles again. It would not stop with this behaviour until the episode ended. The behaviour towards the left location however was even worse. The agent clearly reacted to the opponent being in another location but did so by running either in small circles or into walls. No clear pattern could be recognised.

3.5 Training in pure Python environment

3.5.1 Reasoning

Many iterations in Unreal yielded similar results and at this point it seemed likely that maybe something was going wrong between Python and Unreal. We therefore decided it would be best to implement a version of the project purely in Python. This version should be as close to the Unreal version as possible. For this reason we copied the precise Unreal coordinates of the map, player and opponent to Python. We also used the same version of each Python package that was used for the Unreal implementation as well as the same parameters for PPO2.
In order to visually check the resulting agent we also build a simple map using PyGame [22]. Since PyGames coordinate origin is not in the bottom left but in the top left of the screen and the y-axis values are reversed this resulted in a mirrored version of the game. This was however purely visual. The agent was still using the same values and orientations. For this reason it did not seem necessary to change this visualisation beyond the primitive state it was in.

3.5.2 Discovered sources of problems

This Python implementation lead to the discovery of the underlying problems we were having up until now. Initial training with this new setup which was identical to Unreal yielded identical results. Fairly good but unstable behaviour towards the right and complete divergence towards the left. In Python this problem was however a lot more apparent than it had been in Unreal. The "done" parameter that we mentioned earlier was never set. This parameter get emitted at the end of each step and controls the length of episodes. Once it is set to True the episode ends, the reset function is called and the networks weights are adjusted according to the experiences of the previous episodes. If this is never set the weights can never be properly adjusted. Without it training would be one continuous episode without breaks.

This was likely the single biggest mistake of the project and was the result of a misunderstanding of the parameter itself. My assumption was that the parameter had to be set at the end of the training. But since we wanted to train continuously without resetting the environment (just like one big round of hide and seek where the agent would constantly hunt its opponent) this seemed to not be necessary. Working with Python immediately would have likely made this problem apparent much sooner.

After fixing this problem the agent rapidly improved. We achieved perfect behaviour on the right target but somehow still struggled on the left. Without any clear source of this problem we tested different targets and found out that the problem only occurs in this one specific spot. Further discussion of this in Chapter 4.1.1.

Since other locations worked fairly well we ignored the problem for now and expanded the project to eight instead of two locations. This still worked fairly well but lead to some imprecisions. We tried improving them by normalising all values the network worked with between -1 and 1. That includes observations and rewards. This initially resulted in much worse behaviour. We discovered that one value was forgotten in the normalisation, the distance to the opponent. This was easily fixed and resulted in nearly perfect behaviour in all eight locations. The huge impact this one mistake had showed the importance of normalisation as well as the relation between different observations. Since the new normalised positions were so small a relation to the still big distanced could likely not be made. A discussion of this lesson follows in Chapter 4.2.

3.5.3 Achieved improvements

Encouraged by the new behaviour we lastly tried having the opponent randomly move each step. The agent followed the opponent perfectly within the acceptance radius. The goal of finding and catching the opponent was achieved. We could therefore continue building on this success.

3.6 Implementing aiming and shooting

Since the end of the project was approaching at this point we decided to forego Unreal for now and utilise the much faster speed of training with the pure Python implementation. Where training for 350.000 steps previously took seven hours we could now achieve the same amount of steps in five minutes. This drastically improved the iteration speed.

3.6.1 Detailed changes

To implement shooting and move closer to the goal of this project we first needed general aiming. We started by adding a control for rotation again (similar to previously mentioned) as well as adjusting the reward function to reflect and reward

these new controls. To encourage reaching the goal quickly we punished the agent continuously depending on distance and vision until he reached his goal.

$$\text{Reward} = r_{vision} + r_{distance}$$

$$r_{vision} = \begin{cases} 0.5 & \text{if opponent is within 5 degree cone infront of agent} \\ g(a) & \text{otherwise} \end{cases}$$

$$r_{distance} = \begin{cases} 0.5 & \text{if agent reached the opponent} \\ h(d) & \text{otherwise} \end{cases}$$

The terms for $r_{distance}$ and r_{vision} are limited to a range between -0.5 and 0 for all possible situations on the map in order to keep the rewards between -1 and 1. The direction error a is a value between 180 and 0 which displays the difference between the orientation the agent should have and the one it actually has. In general both terms serve to punish the agent more heavily the further it deviates from its target. While h does so in a linear fashion g does so logarithmically. That is due to the fact that we want very precise aiming and with a linear punishment the difference between aim-offsets get very small the closer we are to actually hitting the target. With a logarithmic punishment however we have big differences in punishment between being for example 1 degree off of the target and being 2 degrees off.

$$g(a) = -(0.1099 * \ln(24.8396 * a))/2$$

$$h(d) = (-0.00025 * d + 0.05)/2$$

These changes resulted in the agent actually aiming at the enemy. It did however exercise maximum controls wherever possible. This resulted in difficulties while aiming since agent jumped from left to right within the acceptance radius of its target. We countered this with an addition to the reward function penalising big controls. con_i describes output i of the network.

$$\text{Reward} = r_{vision} + r_{distance} - 0.1 * {con_1}^2 - 0.1 * {con_2}^2 - 0.1 * {con_3}^2$$

This smoothed the movement of the agent within the acceptance radius, made aiming easier and more stable while not hindering the speed at which the agent reached its goal.

With aiming capabilities at a reasonable level we started working on shooting. The initial idea was to reward the agent for shooting the enemy and punish it for missing. Since shooting the enemy was the most important goal we wanted the agent to achieve we decided to reward each hit with 0.5 points and punish a miss with -0.25 points, as we did not want the agent to just keep shooting. We accepted a shot as

hitting the enemy if it was in our desired cone of aiming (+- 2.5° on each side of the target).
This lead to the agent not shooting at all. It is likely that this behaviour was learned because the agent tried shooting early in the training, kept missing because it could not properly aim yet and was continuously punished. Once it actually was able to aim properly the learned penalty for shooting was too high to try it.

In a next step we tried to train the agent in two stages. In the first it would just learn to aim and walk and in the second it would learn to also shoot. The reward structure stayed the same. This again lead to the agent not shooting at all.

The actual solution to getting the agent to shoot and hit his target was to increasingly punish misses. This gave enough leeway for missing a few shots every now and then but prevented the agent from contuously shooting. The final reward function looks as follows:

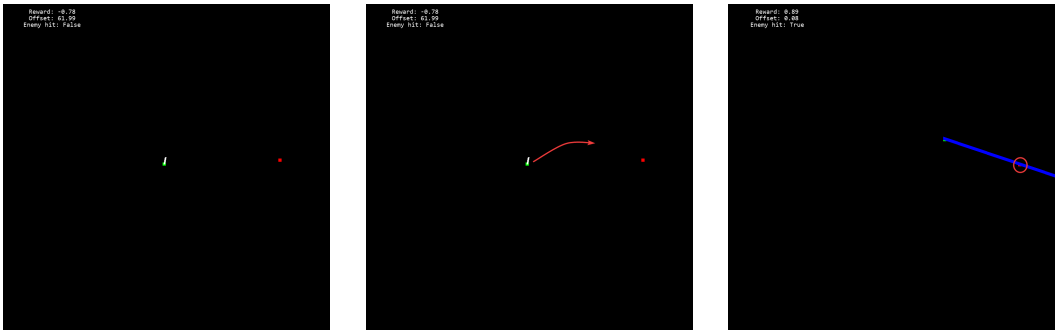
Reward = $r_{vision} + r_{distance} + r_{hit} - 0.1 * {con_1}^2 - 0.1 * {con_2}^2 - 0.1 * {con_3}^2$

$$r_{hit} = \begin{cases} 0.5 & \text{if agent hit the opponent} \\ 0.1 * n_{missed} & \text{otherwise} \end{cases}$$

In this function n_{missed} counts the number of subsequent misses. This number is reset after a successfully hit. Strictly speaking this can hurt our reward normalisation by decreasing the reward below -1 and it is even likely that this happens during training. Since the decrease is however very gradual and the relation to previous iterations can still be drawn this does not seem to hurt the overall training. Further investigations might however necessary.

3.6.2 Achieved improvements

With these changes the agent was able to chase the opponent as well as aim and shoot at it. A solution could be found to almost every scenario and in many of them perfect behaviour was achieved. To illustrate the current state of the agent you can see a typical round in Figure 3.4. 3.4a shows the initial position of the agent right after starting the episode. The agent is shown in green while the opponent on the right is shown in red. The little white line attached to the agent symbolises its rotation. 3.4b shows the path the agent would typically take to get to a position from which it can shoot. Lastly 3.4c shows the agent taking his shot. The blue line represents said shot and marked in red you can see the hit against the opponent.



(a) Start of the episode **(b)** Movement direction **(c)** Agent shooting at target

Fig. 3.4: A typical episode for the agent

You can also see the statistics used to debug and monitor the agent. In the first frame for example the agent is punished for neither being close enough to the opponent, nor aiming in the right direction. While in the last frame it is rewarded for hitting the enemy. The listed offset displays the direction error we have discussed earlier.

4 Limitations

> *Computers are good at following instructions, but not at reading your mind.*
>
> **— Donald Knuth**
> (American Computer Scientist)

4.1 Unresolved issues

Due to time constraints the presented project still shows some unresolved issues. We will briefly discuss two of these issues and possible causes in this chapter.

4.1.1 Divergence in one specific location

The first problem arose in an earlier version of the project. Since it was not a major problem and did not hinder further development we decided against further investigations. Since the problem is however quite unique it seems appropriate to discuss it here.
The earlier version we are talking about is one without movement of the opponent. Instead of moving it switches between initially two and a bit further down the line eight static location. These locations are picked at random at the end of each episode.

With two of these locations we ran into the problem of achieving nearly perfect behaviour towards one of the locations, while completely diverging on the other one. This manifested itself as follows. For the example with two locations one of them was on the left side of the arena, while the other one was on the ride side of the arena. Both had identical height and differed only in their x axis values. They had identical spacing towards their respective walls as well as to the initial spawn location of the agent. Furthermore the first location (as well as all others) was always picked at random, meaning different trainings of the same network should start with different initial locations of the opponent.

Despite this seemingly identical setup of both locations the agent behaved differently for each of them. On the ride side it achieved the desired behaviour. It approached the target in the quickest way possible and then perfectly held the distance of 250 units. After reaching this sate the agent never lost any rewards. It has however to be stated that the agent, while not losing any points, also did not stand still in the desired perimeter. It jumped from side to side using maximum controls. This is likely due to the fact, that we were not punishing unnecessary actions, or rather unnecessarily high effort, at the time.
On the left side however this behaviour drastically differed. The agent still approached its target in the fastest way possible and reached the desired perimeter. It did however not stop in this perimeter but rather kept on walking to the edge of the arena. Upon reaching the edge it would start going down towards the left corner were it would wait until the end of the episode. This behaviour for obvious reason is not only sub-optimal, but nearly the worst thing the agent could do from a reward perspective.

In an effort to fix this behaviour we conducted multiple training sessions of varying lengths, but the outcome was always the same. This suggests that neither the amount of trained episodes are the cause for this, nor is it some random coincidence during training. Further training with different locations then seemed to point towards this specific location as a source of the problem. With just slightly different locations the agent behaved as expected towards both locations. And even in later stages with eight locations this worked perfectly. If we however included this one problematic location in the pool of eight location again the agent would diverge on this specific point.

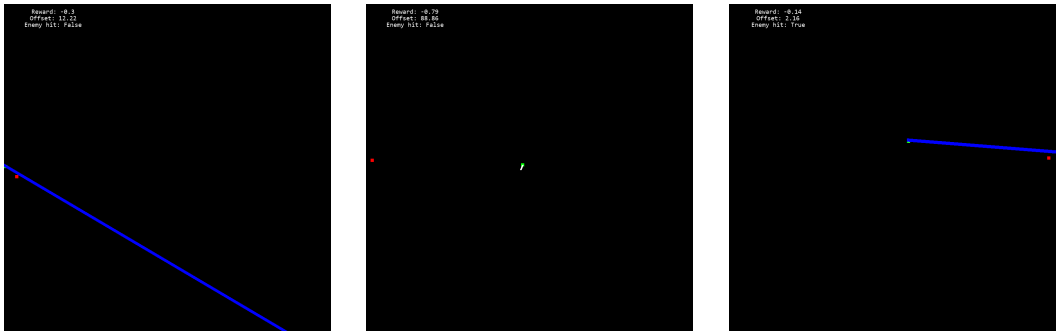
Since the problem could not be fixed in a timely manner we moved on to a moving target just to see whether or not that would work and the agent did indeed chase the target in each direction and to each location on the map. Whatever caused the problem in this specific location is, until now, not known.

4.1.2 Bias towards certain directions

A problem that is still present in the final version of this project might be caused by the same underlying problem. As discussed in this final stage the agent can walk, rotate and shoot at his opponent. The opponent on the other hand just moves randomly across the map.
The problem is now a bias towards certain directions or angles of attack if you will. The agent seems to always pick a side from which to attack the enemy. As far as we know this can be any side or angle and which one it is seems to be random. But the angle always stays the same. After the first successful training for example the

agent always attacked from the left. Even if the agent spawned on the right side of its opponent it would take the time to move all the way across to the other side to shoot from the left. In further training session we would see this behaviour from the top, bottom and top-left.

In most cases this behaviour was unproblematic. Sometimes sub-optimal, sometimes by chance optimal, but mostly useful. In certain edge cases however this behaviour made it impossible for the agent to reach its goal. If it wanted to approach from the left but the opponent was staying close to the left wall the agent had no way of getting to the left side of its target. Instead of adapting and just shooting from the right side however it would move above or below the opponent as close to the edge as possible and try to squeeze between wall and opponent.
A representation of this problem can be seen in Figure 4.1. Figure 4.1b shows the initial state of the map with opponent (in red) being spawned on the left side. The agent (in green) is in the middle of the map. Figure 4.1a shows the moment the agent hit his first shot against the target on the left side. The shot is represented with a blue line. It is clear to see that the agent moved all the way around the target to end up on the left side before it took its shot. Figure 4.1c lastly shows the first shot the agent takes when its target has spawned on the right side of the map. It fires almost immediately upon starting the episode since it always attacks from the left anyways.



(a) First shot on the left **(b)** Starting point **(c)** First shot on the right

Fig. 4.1: First shot at targets on different sides of the map

In Figure 4.2 we can see the impact this issue currently has on the performance of the training. The setup for this graph was symmetrical. We spawned the agent in the center of the map and the opponent on the left wall of the map (in blue) and on the right (in orange). Both locations had the same distance to the agent and their respective walls. We then counted the amount of steps the agent took until he hit his first shot over 100 episodes of 100 steps each. The agent we used was one that had a bias towards attacking from the left side and trained for 3.000.000 steps on the latest iteration of the code.

What can be clearly seen is that due to the agents bias it took him a lot more steps to get a shot off when the opponent spawned on the left side of the map, in contrast to the right. Beyond that we also see a lot more variance in cases where the opponent spawned on the left side of the screen. This is due to its random movement. If the opponent decided to stay closer to the wall it would be even harder for the agent to hit it.

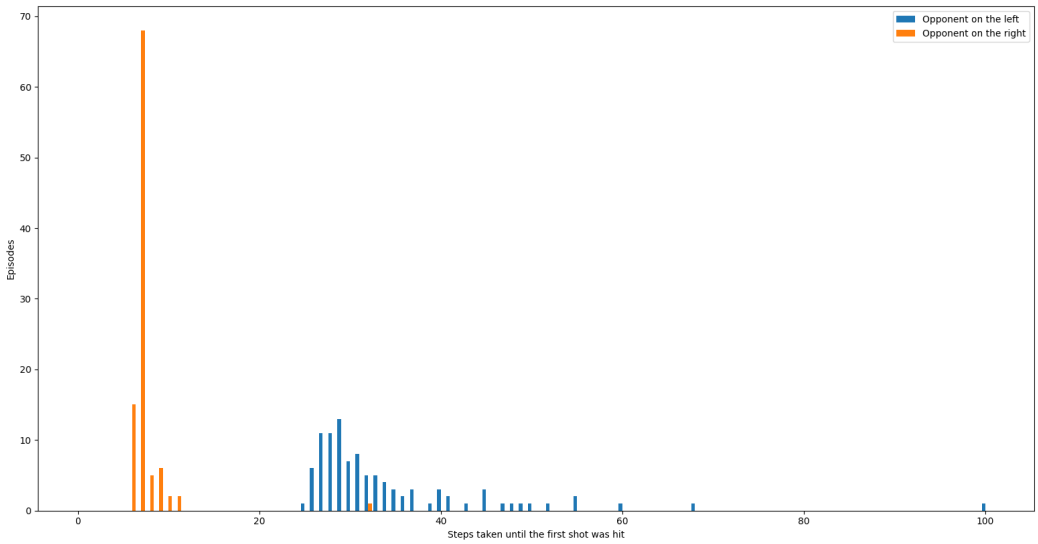


Fig. 4.2: Statistic of how many steps the agent had to take to hit a shot

This is not the ideal behaviour we would have liked to achieve and with more time fixing this issue would have been a priority. For now the only thing that seems to be clear is that a lack of training does not cause this issue since training with 1, 5 and even 10 million actions taken all yielded the same results (although as mentioned towards different directions).
The current and untested theory would be, that this behaviour might be caused by an insufficient exploration rate. So the agent does not explore different ways of approaching its target after it found one direction that works.

4.2 General problems and lessons learned

With all that in mind, what lessons can we take from this project? The first and most important lesson concerns the significance of normalising data. This step had without a doubt the biggest impact on this project. While the agent could barely find its target let alone achieve the maximal reward after normalising the data the exact same code yielded almost perfect results. It is also important to note that this normalisation has to include each and every aspect of the observation space, rewards and controls as well. As long as some of these aspect where not normalised

but others were the agent behaved even worse than before. In our specific case especially not normalising the distance to the opponent led to some big problems.

The next interesting lesson concerns the amount of training required to yield positive results. An initial fear was that an agent would take millions of episodes to achieve behaviour that we deemed good enough to play against a human. In Figure 4.3 you can see the Tensorboard for the latest iteration of the project. One thing that is obvious fairly quickly is that after about 400.000 steps (actions taken) we see only marginal improvements.
For reference, this iteration was trained on a consumer grade computer using a CPU implementation. The CPU used is a *AMD Ryzen 5 1600X Six-Core Processor with 3.90 GHz*. Training for 1.000.000 steps took about 12 minutes for this system. With some optimisation and a GPU implementation it is very feasible that an agent could be trained in under two minutes.

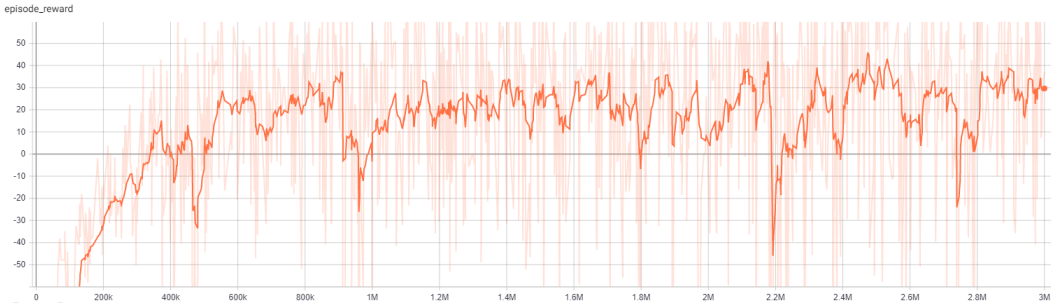


Fig. 4.3: Tensorboard of the final iteration of the project

This clearly indicates that, for an actual product, we would not even have to use pre-trained agents, but could rather train the agent on the users computer while installing the software. This speed of training is obviously only achievable in a purely virtual environment where we do not render the process. How long the agent would take to learn new behaviours from a human player and which learning rate would be ideal will have to be tested in the future.

Lastly we can talk about the flexibility of these agents. While switching from the *Unreal Engine* to purely using Python a big fear was that a pure Python agent would not translate backwards, since in the *Unreal Engine* the agent was subjected to different forces like gravity and impulse and could therefore not move as freely as in Python where a very rudimentary collision system was the only thing affecting the agent.
This fear however turned out to be unfounded, since the agent behaved identical in both *Unreal Engine* and Python. The external forces seemed to play a lesser role than was initially theorised.

5 Conclusions

> *A program is never less than 90% complete, and never more than 95% complete.*
>
> — **Terry Baker**

5.1 Alignment with initial vision

Sadly due to troubles during the implementation as well as time constraints we were not able to fully realise the initial vision of the project. We did however still learn some interesting lessons as mentioned in Chapter 4.2. These lessons for normalising values and expected training duration are likely the most valuable outcomes of this project. Beyond that however we also saw how a machine learning algorithm could communicate with a game engine and thus, in theory, be influenced by a players behaviour. While we did not get to the state of actually trying this approach against human opponents an implementation in the future upon the foundation we created seems very feasible.

Beyond lessons learned we also developed a functioning Python platform which can be used to quickly develop and iterate different agents. Training an agent in Python without taking the detour through Unreal dramatically increases the speed of the iterations as we saw. Furthermore the resulting agent can be ported and used in Unreal (as long as observation and action space are identical in both implementations). So we could not only do tests in Python, but actually develop the pre-trained agents in Python in a timely manner and then let them train against humans in Unreal.

5.2 Outlook

With the knowledge of how a network would have to be configured and trained in order to solve the task at hand we could now load the network with the Unreal implementation and start testing different hyper-parameters in order to figure out how to best learn behaviour from a human opponent. This would likely entail

experimenting with different learning as well as exploration rates.
From a general machine learning standpoint it seems to be out of the question that learning these kinds of strategies from a player would be possible. It is however unclear at this stage how long this training would take and how flexible the agent would be in utilising these learned strategies. With the current network for example we ran into the issue of only ever approaching from one set direction (see Chapter 4.1.2. If this was not fixed it could likely also lead to bad generalisation of the learned strategies. For example learning to flank, but only ever doing so from one specific side or learning to crouch but only behind a specific obstacle.
A big concern would therefore likely be to evaluate the learning progress the agent is making while training against a human as well as testing its adaptability to new situations.

5.3 Closing words

The reason for starting this project was a belief in the general potential of machine learning in the field of video games and the possible use for non-player characters. It can provide a great deal of depth to the interactions between player and character and could further enable unique behaviours that can not be achieved using behaviour trees and similar approaches. The learned behaviours could be uniquely different from human patterns and could even help find bugs in the game since, as we have seen in this project, a trained agent will find and abuse any flaws in the system before a human would notice them.
Proving all these points is hopefully something that future works can achieve. With this specific project however I believe we at the very least showed the potential of unique behaviours and therefore more interesting interactions between players and non-player characters. With a classic behaviour tree many games run into problems of their non-player characters being too predictable due to following very simple rules. Examples of this are always taking the shortest path without any adaptable strategy behind their movement, attacking and hiding in predictable intervals or simply failing to adapt to unusual situations.

The trained agent however does not follow a small set of strict rules and it can adapt to a certain degree. In the example in Chapter 4.1.2 we can see the agent (which at this point wants to shoot from the top left) trying to get an angle on its target. Since the target is too close to the wall however it can not attack from the specific location it would ideally want to do so. Instead of not shooting at all it moves as close to the wall as possible and shoots at a different angle.
As discussed the bias towards a certain direction of attack is not intentional and should be fixed in later iterations. The adaptability however can already be seen and

would likely lead to more interesting behaviour in a more complex environment. With a good framework this behaviour can be achieved in only a few minutes of training. Constructing a behaviour tree complex enough to cover all possible situations on the other hand usually takes developers weeks and months. So not only is the behaviour we can achieve more complex than most behaviour tree based characters, it is also faster. I hope that with more work in this field, approaches like the one we have shown here can help push the current limits of non-player characters in a more complex direction.

References

Literature

[4] Anoop Jeerige, Doina Bein, and Abhishek Verma. „Comparison of Deep Reinforcement Learning Approaches for Intelligent Game Playing“. In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. 2019, pp. 0366–0371. DOI: 10.1109/CCWC.2019.8666545 (cit. on pp. 5, 6).

[5] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. „A Survey of Deep Reinforcement Learning in Video Games“. In: *CoRR* abs/1912.10944 (2019). arXiv: 1912.10944 (cit. on p. 6).

[6] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, et al. „Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents“. In: *J. Artif. Int. Res.* 61.1 (2018), 523–562 (cit. on p. 6).

[7] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, et al. „Reinforcement Learning with Unsupervised Auxiliary Tasks“. In: *CoRR* abs/1611.05397 (2016). arXiv: 1611.05397 (cit. on p. 6).

[8] Dan Horgan, John Quan, David Budden, et al. „Distributed Prioritized Experience Replay“. In: *CoRR* abs/1803.00933 (2018). arXiv: 1803.00933 (cit. on p. 6).

[9] Marek Wydmuch, Michal Kempka, and Wojciech Jaskowski. „ViZDoom Competitions: Playing Doom from Pixels“. In: *CoRR* abs/1809.03470 (2018). arXiv: 1809.03470 (cit. on p. 7).

[10] Yuxin Wu and Yuandong Tian. „Training Agent for First-person Shooter Game with Actor-critic Curriculum Learning“. In: 2017 (cit. on p. 7).

[11] Dino Ratcliffe, Sam Devlin, Udo Kruschwitz, and Luca Citi. „Clyde: A deep reinforcement learning DOOM playing agent“. In: Feb. 2017 (cit. on p. 7).

[12] Guillaume Lample and Devendra Chaplot. „Playing FPS Games with Deep Reinforcement Learning“. In: Sept. 2016 (cit. on p. 7).

[13] Alexey Dosovitskiy and Vladlen Koltun. „Learning to Act by Predicting the Future“. In: *CoRR* abs/1611.01779 (2016). arXiv: 1611.01779 (cit. on p. 7).

[14] Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. „Deep Learning for Video Game Playing“. In: *CoRR* abs/1708.07902 (2017). arXiv: 1708.07902 (cit. on p. 7).

[15]Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. „ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning“. In: *CoRR* abs/1605.02097 (2016). arXiv: 1605.02097 (cit. on p. 7).

[17]Seonghun Yoon and Kyung-Joong Kim. „Deep Q networks for visual fighting game AI“. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. 2017, pp. 306–308. DOI: 10.1109/CIG.2017.8080451 (cit. on p. 9).

[18]Yoshina Takano, Hideyasu Inoue, Ruck Thawonmas, and Tomohiro Harada. „Self-Play for Training General Fighting Game AI“. In: *2019 Nicograph International (NicoInt)*. 2019, pp. 120–120. DOI: 10.1109/NICOInt.2019.00034 (cit. on p. 10).

[19]Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. „Playing Atari with Deep Reinforcement Learning“. In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602 (cit. on p. 12).

[20]Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, et al. „Asynchronous Methods for Deep Reinforcement Learning“. In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783 (cit. on p. 12).

Websites

[1]*Stable Baselines*. URL: https://stable-baselines.readthedocs.io/en/master/ (visited on May 10, 2022) (cit. on p. 1).

[2]*Mindmaker: Deep Reinforcement Learning*. URL: https://www.unrealengine.com/marketplace/en-US/product/neurostudio-self-learning-ai/ (visited on May 10, 2022) (cit. on p. 2).

[3]*Unreal Engine*. URL: https://www.unrealengine.com/en-US (visited on May 18, 2022) (cit. on p. 2).

[16]*Visual Doom AI Competition @ CIG 2016 Track 2 All Rounds*. URL: https://www.youtube.com/watch?v=tDRdgpkleXI (visited on May 24, 2022) (cit. on p. 8).

[21]*DQN documentation, Stable Baselines*. URL: https://stable-baselines.readthedocs.io/en/master/modules/dqn.html (visited on May 28, 2022) (cit. on p. 18).

[22]*PyGame documentation*. URL: https://www.pygame.org/ (visited on May 28, 2022) (cit. on p. 26).

List of Figures

List of Tables

Decleration of authorship

I, Luca Bruder, hereby declare that I have composed the presented paper independently on my own and without any other resources than the ones indicated. All thoughts taken directly or indirectly from external sources are properly denoted as such.

Bayreuth, May 31, 2022

Luca Bruder