

Preface

This bachelor thesis was completed at the Norwegian University of Science and Technology (NTNU) during the spring semester of 2020 in collaboration with Riddlebit Software.

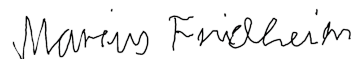
The subject of this thesis was first presented to us by our external supervisor Jonathan Jørgensen last semester during our machine learning course. The original goal was to develop a framework for advanced bots inside of Unreal Engine by using machine learning in combination with traditional AI-tools. As this is a relatively unexplored area in machine learning, and most of the knowledge we had about machine learning was from this course, the research part of the thesis gradually became more substantial in addition to the development.

It is expected that the reader has some prior knowledge about machine learning, comparable to that of one that has completed the TDAT3025 machine learning course at NTNU.

We would like to thank Jonathan Jørgensen, our external supervisor at Riddlebit Software, for his guidance and support on research and development. We would also like to thank our internal supervisor at NTNU, Ole Christian Eidheim, for continuously following up on our work and guiding us throughout the development.



Magomed Bakhmadov



Marius Fridheim

Problem description

Our task is to create a plugin for machine learning in Unreal Engine 4, which is a 3D-graphics engine typically made to build games. This plugin will allow users to create intelligent bots using reinforcement learning for their 3D-application made with Unreal Engine 4. The goal with this plugin is both to be used in Riddlebit's game "Setback", and to make machine learning more accessible for other game developers, as we plan to release and open source our project.

It is also important to emphasize that our task is not only to create a user-friendly plugin, but also to research how machine learning can be done in an advanced 3D-engine like Unreal Engine 4. This is because there hasn't been any substantial development in adding support for machine learning in Unreal Engine and is something we typically don't see in 3D-engines, though the potential with using triple A 3D-engines to make environments for machine learning is endless. In addition we also need to figure out what kind of networks and algorithms work best in such a complex 3D-environment. That is why we have split our task in 1/3 development and 2/3 research, which is the only change we have done in terms of our problem description that was originally 50% development and 50% research.

Abstract

The bachelor thesis is split into a development and research part. In the development part we create a system for Unreal Engine 4 that lets people create controllers for characters that utilize machine learning to make decisions based on input. There is no support directly from UE4 for machine learning, which meant that we had to research and experiment to figure out how we would make this combination work. Our goal is to make this as intuitive as possible so that any UE4 developer can use this system with limited knowledge about machine learning. The other part of the thesis is to research how we can make the connection between UE4 and a machine learning library, and then how we can go about training and using an agent in a game engine like UE4.

We solved the development part of our thesis by creating a plugin for UE4. This plugin has a backend written in Python that handles all the machine learning calculations while the frontend is a Blueprint class that inherits from UE4's controller class. This way we can use this controller to build on top of UE4's AIController with added support for machine learning functions and tweakable variables that communicates with our Python backend. Therefore, because we have created our system based on UE4's architecture, one can simply change the controller or the brain of a bot. For the research part we used different input types as well as different environments to test what worked best in such a system as well as figuring out the best way to train a machine learning agent in UE4. All this research has been compiled into examples showing off different ways to use the system that the user can take inspiration from when creating

their own agents.

Our results show that the agents are able to learn from different environments, and that there is potential for even better results through further experimentation with the state representation, adjustments to the hyperparameters and additions to the Deep Q-Network (DQN). While the agents were able to learn from using images as input, a state representation of handpicked values yielded the best results.

Sammendrag

Bacheloroppgaven er delt inn i en utviklings- og forskningsdel. I utviklingsdelen lager vi et system for Unreal Engine 4 som lar folk ta i bruk maskinl ring for   lage bots. Det er ingen st tte direkte i UE4 for maskinl ring, s  vi m tte unders ke og eksperimentere for   finne ut hvordan vi ville f  denne kombinasjonen til   fungere. M let v rt er   gj re dette s  intuitivt som mulig, slik at enhver UE4-utvikler, selv med begrenset kunnskap om maskinl ring, kan bruke dette systemet. Den andre delen av oppgaven er   forske p  hvordan vi kan lage forbindelsen mellom UE4 og et maskinl ringsbibliotek, og deretter hvordan vi kan l re opp agenter i en spillmotor som UE4.

Vi l ste utviklingsdelen av oppgaven v r ved   lage en plugin for UE4. Denne pluginen har en backend skrevet i Python som h ndterer alle maskinl rings kalkulasjonene, mens frontend er en Blueprint-klasse som arver fra Controller- klassen i UE4. P  denne m ten kan vi bruke denne kontrolleren til   bygge p  toppen av AIController-klassen i UE4 med ekstra st tte for maskinl rings- funksjoner og justerbare variabler som kommuniserer med Python-backenden. Fordi vi har opprettet systemet v rt basert p  arkitekturen i UE4, kan man ganske enkelt endre kontrolleren eller hjernen til en bot. For forskningsdelen brukte vi forskjellige typer inndata, og forskjellige milj er for   teste hva som fungerte best i et slikt system. I tillegg pr vde vi   finne ut av den beste m ten   trene en maskinl ringsagent i UE4. All denne forskningen er samlet i form av eksempler som viser frem forskjellige m ter   bruke systemet p , som brukerne kan bruke som inspirasjon n r de oppretter egne agenter.

Resultatene v re viser at agentene er i stand til   l re fra forskjellige milj er, og at det er potensiale for enda bedre resultater gjennom videre eksperimentering med tilstandsrepresentasjonen, justeringer av hyperparametere og Deep Q-Network (DQN) forbedringer. Agentene var i stand til   l re av   bruke bilder som inndata, men en tilstandspresentasjon av h ndplukkede verdier ga de beste resultatene.

Contents

1	Introduction	10
1.1	Thesis Statement	10
1.2	Research Questions	10
1.2.1	How will our agent perform in different environments? . .	11
1.2.2	How will different state representations affect training and performance?	11
1.3	The structure of the report	11
2	Background	12
2.1	Game Engine	12
2.1.1	Navigation Mesh	12
2.1.2	Perception and stimuli source	12
2.2	Reinforcement Learning	13
2.2.1	Markov Decision Process	14
2.2.2	The Bellman Equation	14
2.2.3	Q-Learning	14
2.2.4	Deep Q-Learning	16
2.2.5	Prioritized Experience Replay	17
2.2.6	Z-score Normalization	18
2.2.7	Double Deep Q-Network	18
2.2.8	Exploration and exploitation	19
2.2.9	Convolutional Neural Network	19
3	Related Work	22
3.1	Playing Atari with Deep Reinforcement Learning	22
3.2	Emergent Tools Use From Multi-Agent Autocurricula	22
3.3	Unity: A General Platform for Intelligent Agents	23
4	Method	24
4.1	Deep Q-Network	24
4.2	Convolutional Neural Network	24
4.3	Prioritized Experience Replay	25
4.4	Z-score	26
4.5	Network Architecture	27
4.6	Hyperparameters and training details	29
4.7	Class Structure	30
4.8	Technology	31
4.8.1	Unreal Engine 4	31
4.8.2	UnrealEnginePython	31
4.8.3	TensorFlow-UE4	31
4.8.4	Github	32
4.9	Hardware	32
4.10	Development process	32
4.11	Distribution of roles	33

5	Results	35
5.1	Scientific Results	35
5.1.1	Race Game Mode	35
5.1.2	Control Point Game Mode	41
5.1.3	Double Deep Q-Network	50
5.1.4	Prioritized Experience Replay	52
5.1.5	Z-score	53
5.1.6	Convolutional Neural Network	55
5.2	Product Results	58
5.2.1	Product	58
5.2.2	Functionality	59
5.2.3	Design	59
5.3	Engineering Results	60
5.3.1	Goals	60
5.4	Administrative Results	61
5.4.1	Project handbook	61
5.4.2	Development methodology	62
6	Discussion	63
6.1	Scientific Results	63
6.1.1	The Environments	63
6.1.2	Comparing State Representation	64
6.1.3	The Training Session	65
6.1.4	Double Deep Q-Network	66
6.1.5	Prioritized Experience Replay	66
6.1.6	Z-score	66
6.1.7	Convolutional Neural Network	67
6.2	Product Results	68
6.2.1	Functionality and design	68
6.3	Engineering Results	69
6.4	Administrative Results	70
6.4.1	Work process	70
6.4.2	System perspective and ethics	71
6.4.3	Evaluating the cooperation	72
7	Conclusion	73
7.1	Future work	75

List of Figures

2.1	Birdview of Agent B (bottom) perceiving Agent T (top) with a sight sense	13
2.2	Flowchart of Q-learning with connection to the agent and environment	16

2.3	If the first hidden layer only had one fully connected neuron, we would have n weights. In practice we have more than one neuron in each layer, meaning we have many more weights.	20
3.1	Pseudo code from “Playing Atari with Deep Reinforcement Learning” (Mnih, Kavukcuoglu, et al. 2013)	22
4.1	Graph PER paper (Schulman, Wolski, et al. 2017)	25
4.2	Image of model described above	28
4.3	A CNN with two convolutional layers and one fully connected layer	29
4.4	Class diagram of the classes used. The classes from UE4 are in grey, and the ones we created are in white	30
4.5	Gantt diagram	33
5.1	Image of the runner game mode described above	36
5.2	Plot of training session from 1 training agent in the race mode .	38
5.3	Plot of training session from 5 training agent in the race mode .	39
5.4	Plot of training session in the race game mode with image input	41
5.5	Image of the control the point environment	42
5.6	Plot of training session from 1 training agent in the control point mode	44
5.7	Plot of training session from 1 training agent in the control point mode vs trained agents	46
5.8	Plot of training session for all agents in the control point mode .	47
5.9	Plot of training session in the control point mode with image input	49
5.10	Plot of training session using Double DQN	51
5.11	Plot of training session using DQN	52
5.12	Plot of training session using Double DQN and PER	53
5.13	Plot of training session using z-score	54
5.14	Plot of training session without z-score	55
5.15	Plot of training session in the race game mode with CNN	57
5.16	Plot of training session in the control point mode with CNN . . .	58

List of Tables

2	The model and hyperparameters for 1 agent in the race mode . .	37
3	Hyperparameters used for training session in the race mode with image input	40
4	The model and hyperparameters for 1 agent in the control point mode	43
5	The model and hyperparameters for 1 complex agent in the race mode	44
6	Hyperparameters for 4 agents in the race mode	47
7	Hyperparameters used for training session in the control point mode with image input	48
8	The model and hyperparameters for Double DQN	50
9	The model and hyperparameters for z-score	53

10	Hyperparameters used for the training sessions plotted in figure 5.15 and 5.16.	55
----	---	----

Definitions	
Unreal Engine 4 (UE4)	A 3D-graphics engine mostly known for making games.
Character	Refers to the body of an agent.
Controller	A controller is used to control the body of an agent (character). They function as the brain of the agent.
Machine learning (ML)	Field within Artificial Intelligence, that provides systems the ability to learn and improve from experience without human interference.
Neural network (NN)	Or an artificial neural network(we will just refer to it as a neural network) is a network of connected artificial neurons. This allows us to digitally constitute a brain, that is the main component in ML.
Layers	Collection of neurons operating together. Typically a neural network has one input layer, a set amount of hidden layers and then one output layer.
Deep neural network (DNN)	A deep neural network is just a neural network with at least 2 hidden layers.
Reinforcement learning (RL)	Area within machine learning that focuses on training neural networks using a system of reward and punishment in an environment.
Agent	Software programs that learn and make decisions in an environment. In our case this is a bot/player that is running a neural network for making decisions.
Game mode	A set of rules given to a map. This makes so that a map has objectives and is playable.
Environment	A demonstration of a problem that agents can interact with and potentially solve. In our case this is a map with a game mode that the agent spawns in and can interact with. In our case this is for example a 3D-map, the game mode and the opponents playing the game mode.
State	A The input vector for a neural network, what the agent can observe.
Percept	An impression of an object obtained by use of the senses.
Actionspace	The actions a machine learning agent can perform.
Blueprint	A gameplay scripting system for UE4 that uses a node-based interface to create gameplay elements.

Behaviour tree	A mathematical model of plan execution. In UE4 this is used to define the necessary circumstances for an AI character to perform different actions, by connecting a series of nodes together and adding conditional logic.
Mesh	Is a collection of vertices, edges and faces that makes up a polyhedral object in 3D computer graphics.
Hyperparameters	Variables that are set before training starts to define the structure of the network and how the network is trained.
Gamma (γ)	Discount factor from 0 to 1. In reinforcement learning used to adjust how much we should value future reward. 0 not value future reward at all and 1 value future reward just as much as current. In our case this is usually 0,9 or 0,95 as we want to value future reward.
Epsilon (ϵ)	Hyperparameter to adjust randomness of the agent. This value ranges from 1 to 0, where 1 is 100% randomness, while 0 is 0% randomness
Overfitting/underfitting	Overfitting occurs when our ML model doesn't generalize well, meaning that it performs well on the specific cases we train it on, but drops in performance when used with unseen data. Underfitting means that the model is unable to find a pattern in the data, causing it to underperform during training.
Activation function	A function that decides whether a neuron is activated by calculating the weighted sum and adding bias to it. The purpose is to add non-linearity to the outputs of the neurons, which makes the NN capable to learn more complex tasks. Sigmoid, tanh and ReLU are popular activation functions.
External supervisor	Our supervisor from Riddlebit Software.
Internal supervisor	Our supervisor from NTNU.

1 Introduction

As the field of reinforcement learning has gotten increasingly more popular over the years, we are starting to see the need for systems capable of combining RL with games and 3D-applications. More capable 3D-engines are required to be able to build the environment for agents to learn in, both for the purpose of creating more intelligent bots for games, as well as for research purposes. While solutions like OpenAIGym (*Gym*) are very useful for testing and comparing algorithms, they are less practical for more complex projects and purposes. Even though we have started to see some 3D-engines like Unity (*Unity*) implement support for machine learning, there's still a lack of research on the topic as well as a good framework for one of the most popular 3D-engines, Unreal Engine 4 (*Unreal Engine*).

Combining machine learning with UE4 requires both theoretical and practical knowledge on the subject, something that discourages and restricts access to this combination for some game developers. A solution that simplifies this process by taking care of the practical implementation of the reinforcement learning algorithm would be an interesting addition for game developers, as well as providing relevant research for machine learning enthusiasts. A proof of usefulness and examples that show the functionality could also spark interest for further development on the field.

1.1 Thesis Statement

In UE4, we typically create bots by making high level actions, then use a behavior tree to decide the conditions that are required to perform each of these actions. The problem with this is that it requires an increasingly more complex behaviour tree to make the bot showcase more advanced behaviour. Our solution to this is to combine UE4's AI-tools with RL, more specifically Deep Q-Network (DQN). This would be a replacement for the typical clusters of programming necessary for the decision making. This means we will use UE4's tools to create actions the bot can perform, and DQN to choose between these actions. This leads us to our thesis statement:

- How can we combine DQN with the existing AI-tools in UE4?

1.2 Research Questions

Because of the lack of support for machine learning in UE4, we would have to research and test how such an implementation would work. Since this is a vastly unexplored field, we wanted to narrow down the focus of our research to fit the scope of a bachelor thesis. Therefore, we have summarized the direction of the research into two questions.

1.2.1 How will our agent perform in different environments?

UE4 allows us to create more complex environments than the ones we typically see used with reinforcement learning. The performance of reinforcement learning algorithms can vary based on several variables, including the complexity of the environment. Since one of the uses of our system is to implement RL agents into games, we needed to find out if these agents would be applicable to popular game modes. By mimicking these game modes, we could investigate how different environments impact the learning process of an agent.

1.2.2 How will different state representations affect training and performance?

The data that gets sent as input to a reinforcement learning model is the only representation of the state that is available to an agent. This is why it is extremely important to figure out the best way to present this data. Our idea is to try both input with images, which is the state representation typically used with DQN, and a state based on different handpicked features that we think are important to understand the environment. In addition to this, we want to see how we can improve the performance of each of these state representations with further enhancements.

1.3 The structure of the report

Both the research and development part of our thesis is integrated into this report. We start off by going through the theoretical background needed to understand the rest of the report. Note that we assume that the reader of this paper knows basic machine learning theory, equal to a fundamental ML course. After the theory we go through some related work that we used, or took inspiration from that is not necessarily theory. Then in the methods chapter we explain what choices were made and why they were made to complete the thesis. Next is the results chapter. Here we will present all our results from both our research and development, before we elaborate on them in the discussion chapter. In this chapter we will be taking our results and discuss the different aspect of them and how they connect to our thesis statement. We will also discuss different weaknesses and strengths our system has based on our results and experiences. In our conclusion we will summarize what we have done through this project and how that completes our thesis statement and research questions. Finally, under future work, we discuss how one can go forward with development and what would be some interesting improvements to see that we did not have time to implement.

2 Background

In this chapter we will explain some of the theory needed to get a better understanding of our implementation and results. We first explain the theory behind some of the tools in UE4, and then elaborate on the theory behind reinforcement learning, more specifically DQN.

2.1 Game Engine

A game engine is a software development environment for building video games. These video games are typically for consoles, mobile devices or computers. Some of the main attributes of a game engine is rendering of 2D or 3D graphics, physics simulation, collision detection, sound, scripting, animation, game AI-tools and networking.

The reason we have game engines is because when making games there are a lot of computer instructions that will be given that are not specified to each game. Therefore instead of writing these over and over again for each game we use game engines as a framework for making games and 3D applications. In addition this makes game developers able to focus on the game itself, instead of for example how to write to a hard drive or draw on the screen, as the engine handles that.

2.1.1 Navigation Mesh

A navigation mesh (navmesh) is a collection of two-dimensional convex polygons, commonly used in game engines, that encompass a given area of a map. Polygons that are adjacent to each other are connected in a graph. The purpose of a navmesh is to define areas of the map which the agents (AI like an enemy or friendly character in a game) can travel across and help the agents with pathfinding. When an agent wants to move to a new position, the agent uses the navmesh to navigate through the area towards the goal. The navmesh utilizes a graph searching algorithm to run through the collection of polygons to find a path through complicated areas for an agent. A* is one of the most popular algorithms for this, and is used in navmeshes in Unreal Engine with some tweaks added to it.

2.1.2 Perception and stimuli source

A stimuli source is used to update the perception of an agent. We can give an agent perception, which means that the agent can perceive the surroundings through senses and allows them to react to things they can see, hear, feel etc. We can choose which things are a valid source for any of those perceptions by making something a stimuli source.

To make agents sense each other we can give each of them perception and set each other as stimuli sources. This way we can, for instance, have an Agent B

that walks around and senses an Agent T because Agent T is in the line of sight of Agent B. A basic illustration of this can be seen in figure 2.1.

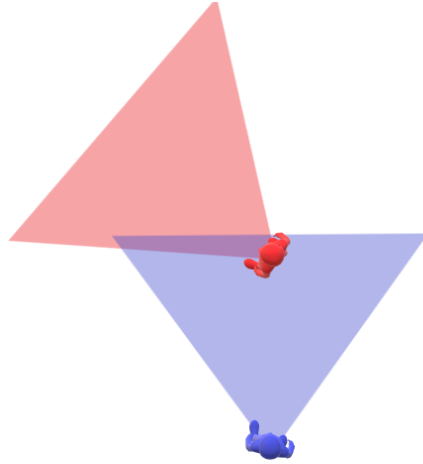


Figure 2.1: Birdview of Agent B (bottom) perceiving Agent T (top) with a sight sense

In summary, an agent's perception are the senses of an agent, and a stimuli source is something the agent can sense because it stimulates the senses of the agent, thus updating the agent's perception. UE4 offers this functionality through an *AIPerception* (*AIPerception*) component that can be added to characters.

2.2 Reinforcement Learning

Our plugin builds on a lot of research that has been done in the field of reinforcement learning (RL). Reinforcement learning is a branch of machine learning which uses statistical methods and sample data to create mathematical models. The goal of machine learning is to make computers able to perform human-like tasks without using specific instructions. Typically this is done with either feeding the algorithm with labeled data or finding features, but reinforcement learning takes a different approach. In RL we only tell the algorithm when an action was good or bad in terms of score. In other words, a reinforcement learning agent learns from exploring the environment and figuring out how to get the best score possible, much like how humans would. Like said in the book "Deep Reinforcement Learning in Action", "All human behavior is fundamentally goal-directed" (Zai and Brown 2018), which is why it makes sense to create agents that are supposed to play against humans with reinforcement learning,

as they are both fundamentally goal-directed.

2.2.1 Markov Decision Process

The Markov decision process (MDP) is a sequence of states S_1, \dots, S_n that can be represented as Markov properties. The Markov property (P) is a state that is independent of the past given the present. In mathematical terms:

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, \dots, S_t] \quad (2.1)$$

Here t is the time or in our case the numbered state for a prediction request and S is the state. This is important in RL because we can not have a state that is dependent on anything else than the previous state at every time step. Note that $S \in s$, meaning $S_t = s$. This will be the same for all our equations.

Another important MDP concept is the state transition probability. This probability is usually represented into a matrix.

$$\mathcal{P} = \begin{pmatrix} \mathcal{P}_{1,1} & \mathcal{P}_{1,2} & \cdots & \mathcal{P}_{1,n} \\ \mathcal{P}_{2,1} & \mathcal{P}_{2,2} & \cdots & \mathcal{P}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{P}_{m,1} & \mathcal{P}_{m,2} & \cdots & \mathcal{P}_{m,n} \end{pmatrix} \quad (2.2)$$

\mathcal{P} is the probability, and each row represents a probability for moving from our state to any successor state. Sum of each row is 1.

It is important that the dynamic of the system can be defined using these states and transitions. A transition can be states, actions, rewards and decisions. This lets us use MDP to describe the environment to the RL agent.

2.2.2 The Bellman Equation

Another important concept in reinforcement learning is the Bellman equation. This equation handles how an agent should try to maximize the reward gained over time to find the optimal policies. The Bellman equation makes it possible to express the value of each state as mathematical functions of other states in what we call a value function. Mathematically we can define The Bellman equation for stochastic environments like this:

$$V(s) = \max_a (R(s, a) + \gamma \sum_{s'} \mathcal{P}(s, a, s') V(s')) \quad (2.3)$$

This is called a value function and gives a value for a state based on current state (s), reward (R), action (a), next state (s'), probability (\mathcal{P}) and the constant discount factor (γ). We will be using the same symbols for the rest of the report.

2.2.3 Q-Learning

Now to get to Q-learning we can take the Bellman equation (2.3) that gave us the value footprint for one possible action, or the quality of that action. Here

is where the Q in Q-learning comes from, estimating the quality of all actions. So if we take $V(s)$ and remove the max, we get an equation for the quality of any practical action. Now we have:

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} \mathcal{P}(s, a, s') V(s') \quad (2.4)$$

However we still have $V(s')$ left that we need to find a substitute for. So let's take 2.4 and let's replace $V(s')$ with $\max_a Q(s', a')$ which is essentially the same.

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} \mathcal{P}(s, a, s') \max_a Q(s', a') \quad (2.5)$$

Now we are left with an equation to evaluate the quality of each action in a stochastic environment, hence our first step towards Q-learning. The only variable from this equation we have not used before in this paper is a' which is the next action.

Q-learning is an algorithm that uses the 2.5 equation to evaluate what action from an action space is the best one to perform based on the input given. This is in an off-policy algorithm because the Q-learning function learns by taking random actions, meaning a policy is not needed. That is why a Q-learning agent only learns to maximize the total reward.

Here is an example on Q-learning with a Q-table. A Q-table is a matrix in the shape of [state, actions], that works as a reference table for our agent to pick a value.

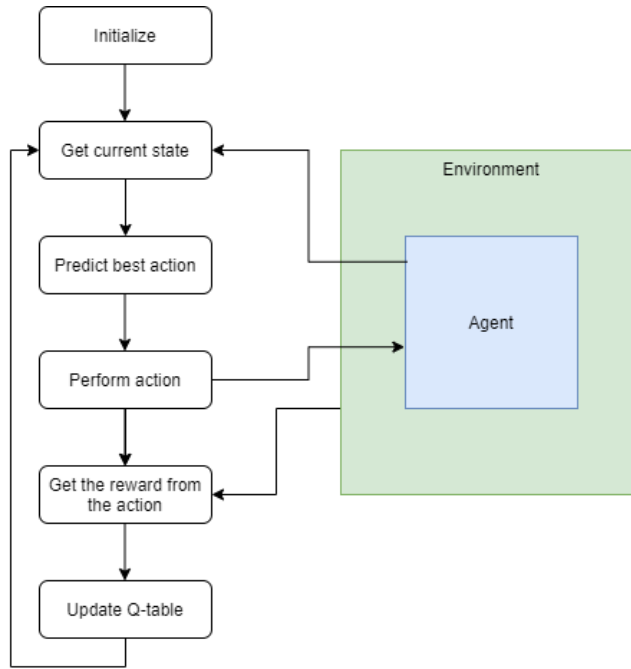


Figure 2.2: Flowchart of Q-learning with connection to the agent and environment

In figure 2.2 we first initialize the Q-learning algorithm, agent and Q-table. Then we get the current state from the agent, this could for example be pixels of what the agent can see. Then we use our algorithm to predict what it thinks is the best action given the state. Remember that it is probably best to have 100% randomness when we first start the learning algorithm and then we can have less randomness and more of the table being used as we go. This is to make sure the agent explores the environment and doesn't get stuck never trying new actions, more about this in chapter 2.2.8. Then we make the agent perform this action we have chosen. After that we use the environment that is the representation of the problem we want to solve to score the action and give the algorithm this score as positive or negative reward. We then update the Q-table based on the state, prediction and score, and repeat. Remember that doing this with a Q-table can be very heavy for the CPU, and that is why in the following chapter we will look at better ways of doing this.

2.2.4 Deep Q-Learning

In chapter 2.2.3 we introduced Q-learning with a Q-table, however if there are a lot of combinations of states and actions the memory and computation requirement will be too high. To solve this we use a deep neural network (DNN)

to approximate $Q(s,a)$ instead of a Q-table. This combination of Q-learning and DNN is called deep Q-learning (DQL). Now instead of remembering each solution in the Q-table we approximate the Q-value function (2.5) with a DNN, which is known to be a powerful function approximator. Now the DNN becomes a Deep Q-Network (DQN), meaning the network is structured to have Q value estimations as output. This was suggested in the paper "Playing Atari with Deep Reinforcement Learning" (Mnih, Kavukcuoglu, et al. 2013), more about this paper in chapter 3.1. From now on we will use DQN as a common denominator for Q-learning with DNN and the other algorithm suggested in the paper (Mnih, Kavukcuoglu, et al. 2013), as this is the norm in the RL field.

2.2.5 Prioritized Experience Replay

Prioritized Experience Replay (PER) is an algorithm built on the concept that some experiences are more important than others in terms of the learning process. In the original DQN paper (Mnih, Kavukcuoglu, et al. 2013) we have a way of training the neural network that picks a random set of transitions from a buffer (the amount of transitions picked is the mini batch size) and then iterates over those to do the training process. Instead PER introduces a way to prioritize the most interesting experiences and then train with these rather than random experiences. We can see this in the paper "Prioritized Experience Replay" (Schaul et al. 2016) where they implement PER on a DQN to show that this combination outperforms a normal DQN.

How we decide if an experience is valuable or not is with the calculation of the TD error ($|\delta_i|$), which is the difference between the old target Q-value and the new Q-value. When the difference is large we have something interesting that happened and learning this is usually more valuable. i is the transition, containing action, state, next state and reward. In addition we add a constant (ϵ) that ensures that no experiences has 0 probability of being chosen. The priority value we use then becomes:

$$p_i = |\delta_i| + \epsilon \quad (2.6)$$

Now we still have a problem of overfitting as if we only choose experiences by this calculation we will be training on the same experiences over and over again, which means we will be overfitting our agent. That is why PER introduces stochastic prioritization like so:

$$P(i) = \frac{p_i^a}{\sum_k p_k^a} \quad (2.7)$$

Here p_i the priority value (2.6), and $\sum_k p_k^a$ is normalization by using all the values in the replay buffer. a is a hyperparameter for reintroducing randomness, and k is the mini batch size. After this all we need to do is to use a binary tree to store our experiences and then sort this tree with equation 2.7 after we have taken a set of experiences. It is also worth noting that there are two versions of

PER, rank-based and proportional. The version we cover here is proportional as that is the one we use in our implementation, more about this in chapter 4.3.

2.2.6 Z-score Normalization

When we have a vector of inputs of for example distance to target and rotation to target, we notice that distance and rotation are two very different values. Rotation ranges from for instance 180 to -180, while distance can possibly range from 0 to 10 000. This creates a big difference in how a neural network will prioritize these values. A value of 5 000 can be more dominant than a value of 100. That is why we normalize the input to make sure that all the values are equally important to a neural network.

Z-score normalization gives us a way to normalize input so that all inputs are on the same scale by using the mean(μ) and standard deviation(σ).

$$Z = \frac{x - \mu}{\sigma} \quad (2.8)$$

This formula outputs how many standard deviations from the mean the input value is, which is very helpful for a reinforcement learning algorithm.

2.2.7 Double Deep Q-Network

A problem with the DQN proposed in "Playing Atari with Deep Reinforcement Learning" (Mnih, Kavukcuoglu, et al. 2013) is the tendency to overestimate the Q-values. This is because we use equation 2.3, and specifically the $\max_a Q(s', a')$ part. When we take the estimate of the maximum value, it can introduce a maximization bias in learning.

The solution to this problem is to create a second neural network model to estimate the Q-value. Then we use two models to update each other. This will make us able to avoid maximization bias by untangling our estimates. The implementation of this is called Double Deep Q-Network (DDQN), and was introduced in the paper "Deep Reinforcement Learning with Double Q-learning" (Hasselt, Guez, and Silver 2015). This changes equation 2.3 to:

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} P(s, a, s') \max_a Q_{target}[argmax(Q(s', a'))] \quad (2.9)$$

The Q-target being the new neural network being introduced with DDQN, which is a copy of the DQN from the last episode. So instead of estimating the Q value for the next state with the Q-network, we use the index we get from estimating the Q-value for next state with the Q-network, and then use the Q-value from the Q-target at this index.

2.2.8 Exploration and exploitation

An agent chooses actions in a state based on the Q-values, in an effort to maximize the reward. However, an agent hasn't learned anything about the environment at the start of the learning process, and thus can't reliably predict the best course of action in a given state. As in life, the agent will have to try and fail so that learning can occur. In other words, the agent will have to explore the environment to be able to adapt to it. At first the actions will have to be random, but after a while the agent can start exploiting what it has learned.

Exploration is however not something that only should occur at the start of the learning process, because if an agent starts exploiting after finding the first strategy that yields positive rewards there's still a possibility there are other choices that could lead to better rewards. A simple example of this can be an agent that tries to learn moving from position A to B in a grid-like environment, and gets more reward the quicker it gets there. This agent may from some early exploration get to position B, but if it starts to exploit this and disregards further exploration it may possibly ignore an alternative route that would get it to B faster. An agent deliberately doing sub-optimal actions can therefore lead to the discovery of more optimal choices in the long run.

There are different methods used to implement this balance between exploration and exploitation, further elaborated in this study (Thrun 1992). The simplest and most used one is probably the "epsilon-greedy" strategy. This strategy simply allows the agent to take random choices at random times rather than taking the choice the agent deems optimal at the time. This is often implemented with a decay, meaning that the actions the agent chooses are close to or completely random in the beginning, but slowly decays to a nether threshold over time. This ensures that the agent explores a lot in the beginning, starts exploiting what it learns over time, and still makes sure to occasionally explore even after finding a way to receive rewards.

2.2.9 Convolutional Neural Network

Convolutional neural networks (CNNs) are very similar to fully connected (also called dense) neural networks, the difference being that CNNs (Karpathy and Li 2015) assume that the inputs are images. To further elaborate, neural networks receive a single input vector and transform that data by sending it through a number of hidden layers. If the inputs are images, the number of values in the vector are equal to $(width) \times (height) \times (number\ of\ color\ channels)$. If an image has a size of 28×28 and 1 color channel, the input vector consists of 784 values. If the images are bigger and in addition also have more color channels, the size of the input vector increases by a lot.

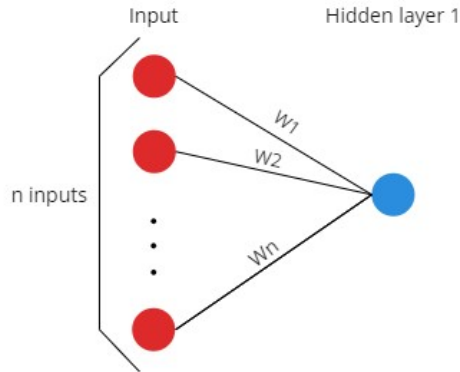


Figure 2.3: If the first hidden layer only had one fully connected neuron, we would have n weights. In practice we have more than one neuron in each layer, meaning we have many more weights.

Since every neuron in a neural network is fully connected to all the neurons in the adjacent layers, the amount of weights in the network increases by a lot. Figure 2.3 shows a simplified illustration of how the neurons in a neural network are connected. In CNNs, the neurons aren't fully connected, meaning that every neuron in one layer is only connected to a small portion of the neurons in the previous layer.

The architecture of a CNN consists of three main types of layers, convolutional layers, pooling layers and fully connected layers. The convolutional layer consists of a set of filters (also called kernels) that are used to compute the output of the layer. A filter is an array of numbers that can for example have a size of $5 \times 5 \times 1$. The 1 refers to the depth of the input, meaning that it only has 1 color channel. So for instance, if the image was in RGB, the depth would be 3. The output of the convolutional layer is computed by placing the filters over small parts of the input image and performing a computation on the elements of the filter array and the original pixel values. The filter slides, or convolves, across the image and does this computation across the whole image. The output is a new array of numbers that are called activation maps or feature maps. The purpose of the convolutional layer is to identify different features within the

image by using a number of filters that try to identify everything from simple curves and lines, to larger and more complicated structures.

Pooling layers are used to reduce the amount of computation and parameters by reducing the size of the input. There are different ways to do this, for example max-pooling and average-pooling. Similarly to how the filters in the convolutional layers move across the image, the pooling layers also have a window that moves across the input. Maxpooling, being the more popular one, takes the values inside of the pooling window and only saves the biggest one. The idea is that the size is reduced, but the most defining features (being the ones with the biggest values) still remain.

The final layer in a CNN is a fully connected layer that takes in the input volume of the previous layer and outputs an N -dimensional vector where N refers to the number of classes the network can choose from. In our case N refers to the number of actions our agent can choose between. You can also have several of these fully connected layers, where the last one will be the one connected to the output layer of actions.

3 Related Work

In this chapter we will look into related research and development that has been done in the field of reinforcement learning.

3.1 Playing Atari with Deep Reinforcement Learning

A project on creating human-like control for bots in Atari games was created in 2013. In this paper is where we first see the suggestion of using a Deep Q-Network (DQN) in virtual environments. By taking the typical Q-learning concept and combining it with a convolutional neural network, creating the first DQN, they managed to create one network capable of learning from multiple Atari games with just using the frames provided. This paper also provides the pseudocode and math needed to create a basic DQN. In addition they provided a method for training that combines stochastic mini batch updates with a memory of experiences from previous predictions. These breakthroughs become very important in our thesis as our reinforcement learning algorithm builds on the principles and math provided in this paper.

Algorithm 1 Deep Q-learning with Experience Replay

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
    Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
    for  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
    end for
end for

```

Figure 3.1: Pseudo code from “Playing Atari with Deep Reinforcement Learning” (Mnih, Kavukcuoglu, et al. 2013)

The pseudo code above shows how to train a DQN using the experience replay method with mini batches as suggested in the paper.

3.2 Emergent Tools Use From Multi-Agent Autocurricula

Baker et al. 2019 presented a new algorithm capable of playing hide and seek in a 3D-environment with movable objects. This is very interesting to us, as this is first of all reinforcement learning in a 3D-environment like ours. In addition

they managed to make these agents learn incredibly complex behaviors by not using images, but an input vector of selected values. Which gave us the idea to use this with a DQN that we have only seen used with images. They also show how one would go about feeding these values to a neural network. We decided not to use their specific algorithm called Proximal Policy Optimization (more about this in the method chapter), but this paper is still very relevant as it shows the most intelligent agents we have seen. They also show a lot of ways they optimized these agents, one being z-score normalization. Lastly they give us a baseline for typical RL value tweaking, and a suggested DNN size for vector input.

3.3 Unity: A General Platform for Intelligent Agents

Here we have a framework and research (Juliani et al. 2018) that shows of the benefits of using Unity, a game engine, to create environments to train machine learning agents in. This is very interesting to us as Unity is usually looked at as UE4's main competitor, and to have a system like this in a very similar game engine gives a lot of inspiration. They did not use any of Unity's built-in AI-tools like navmesh and stimuli source, but instead focused on making a usable system that researchers can use to implement ML agents into the engine. They also provide examples of ML agents using PPO in a Unity environment. It is important to emphasize that this is a different engine so we can not use anything from this paper or framework directly, but it gave us tons of ideas and insight before designing our system.

4 Method

This chapter will address what methods and technologies we used to complete our thesis. In addition we will go through our development process and distribution of roles.

4.1 Deep Q-Network

In reinforcement learning there are a lot of different algorithms, or rather sets of algorithms and neural network structures to choose from. Among the ones that show the best results are Proximal Policy Optimization (PPO), Deep Q-Network (DQN), Asynchronous Advantage Actor-Critic (Mnih, Badia, et al. 2016), Deep Deterministic Policy Gradient (Lillicrap et al. 2019) and Trust Region Policy Optimization (Schulman, Levine, et al. 2017). Out of these it seems that PPO and DQN perform the best with a reasonable amount of computational power in a continuous environment. Then when we compare these algorithms it seems that DQN has a better sampling efficiency, but PPO might be able to eventually outperform in terms of score. However, DQN has a lot of different additions that can be implemented like Double, Dueling Double (Wang et al. 2016) and Noisy (Fortunato et al. 2019) Deep Q-Networks. When we look at comparisons between these it looks like DQN might be able to perform as well or even better than PPO when using these additions, while still having better sampling efficiency.

The paper "Rainbow: Combining Improvements in Deep Reinforcement Learning" (Hessel et al. 2017) looked at combining all the most popular DQN improvements together, and found that combining the different additions into one algorithm improved both score and training time. The choice of using this algorithm and network would also allow us to create a basic DQN first, then add improvements as we go. By doing this we made sure we had time to implement a high-end reinforcement learning algorithm, make it work within UE4 and create a plugin for UE4 that others could use. This way we could add the improvements based on how much extra time we ended up having after making a basic DQN in UE4.

In our system we use DQN in a Python backend connected with UE4 through JSON messages (Crockford 2006). The DQN algorithm and network (more about the network in chapter 4.5) is responsible for making decisions, based on the input we send through these JSON messages. The DQN also responds by sending a JSON to UE4 containing an index of which action the agent should perform.

4.2 Convolutional Neural Network

Using images as input for a DQN is very common, though most of the earlier examples do this with 2D games and we are using this in a 3D environment. So it was only natural that we would send in images as input to our model,

to explore how this state representation would affect the training results and performance. CNNs are widely used for image recognition, but generally require a large amount of data to learn and are therefore mostly associated with supervised learning. The bigger these images are, the heavier the load on the hardware. The performance of using a CNN is also very reliant on how the hyperparameters are chosen. The addition of CNN into our project is therefore an exploration to find the best possible performance we can get by balancing all these factors.

4.3 Prioritized Experience Replay

Prioritized experience replay is, as mentioned in chapter 2.2.5, a way to sample experiences that is more valuable to the learning process. This was first suggested to us by our external supervisor. When we look at the research paper "Prioritized Experience Replay" (Schulman, Wolski, et al. 2017) we see that combining this algorithm with DQN creates a huge increase in learning speed. As shown here:

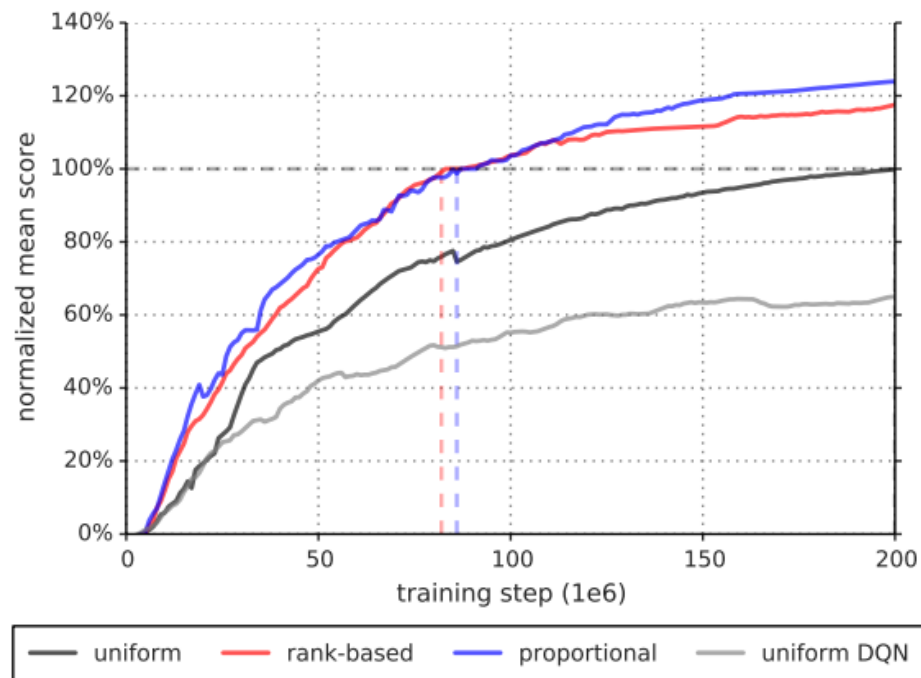


Figure 4.1: Graph PER paper (Schulman, Wolski, et al. 2017)

The graph is showing score over time on Atari games with two different normal uniform sampling DQNs, and two different versions of PER, rank-based and proportional. As we see here it shows both an improvement in learning speed

and score, and after further research shows that this might be one of the single best additions to DQN.

The difference between rank-based and proportional PER is in how we sort and sample from the binary tree. Rank-based typically sorts in $O(\log n)$ time, where n is the size of the replay buffer. While proportional does not need sorting, but $O(\log n)$ time when we add to the buffer. So essentially because we in RL train each time we add new experiences, they use the same amount of process power to keep the binary tree sorted. When we look at the time it takes to sample, rank-based is a little faster with a sampling time of $O(k)$, while proportional takes $O(k \log n)$, n being the size of the replay buffer and k being the mini-batch size. So based on this, rank-based might be a little faster, but it also has hyper-parameters that needs to be tweaked to make it perform as well as proportional does right out the box. This makes proportional a little slower, but much better for a system that we want others to be able to use. In addition proportional seem to be overall able to pick better experiences to train on than rank-based. We can see this in figure 4.1, as proportional is able to achieve a better score.

We use PER in conjunction with DQN in our Python backend. You can think of PER as just a way to administer what experience we train our DQN with. In our implementation we have a class that handles the PER algorithm and binary tree. Then we have a method that lets us fetch experiences, the input for this method is how many experiences we want to get. Then when we are done we calculate the absolute errors, then we call a update method that sort the tree based on these new absolute errors. Also every time we get a new experience we call an add function that adds the experience in the right place in the binary tree.

4.4 Z-score

Machine learning algorithms attempt to find connections between features of data points. When these data points are then on different scales this becomes very difficult, and might lead to some features being more dominant. There are a bunch of practices that fix this problem, the two most popular being min-max and z-score normalization. We ended up choosing z-score normalization because this strategy handles outliers, values that are way off the norm, much better than min-max normalization, while still making sure the values are normalized and standardized. In addition, the paper on PPO (Baker et al. 2019) that also uses input in terms of a vector of different values showed great results using z-score normalization.

Because of the nature of how one has to first calculate the means and standard deviations from a magnitude of pre-recorded data before we can use z-score normalization, we have a two part implementation and process of how to use z-score. First you have to sample the agent's observation to do the calculation. To make this as simple as possible for the user we created an option that will print the observation to a text file. Then we made software that takes this

text file and does all the calculations and presents it in a way that our system will accept. We have an input field in the blueprint where you can paste these calculations and tick on the option to use z-score. When the observations come to the Python backend, we use equation 2.6 to normalize it.

4.5 Network Architecture

Inside of the DQN class we have a class handling the neural network model. The neural network model contains an input and output layer with a set of hidden layers in between. This can for example be an input layer of 16x128, a hidden layer of 128x256, then another hidden layer of 256x256, and then the output layer of 256x9. This is an example of a model with an input state of 16 values, 2 hidden layers and one output layer that can evaluate 9 actions. We use tanh as activation function.

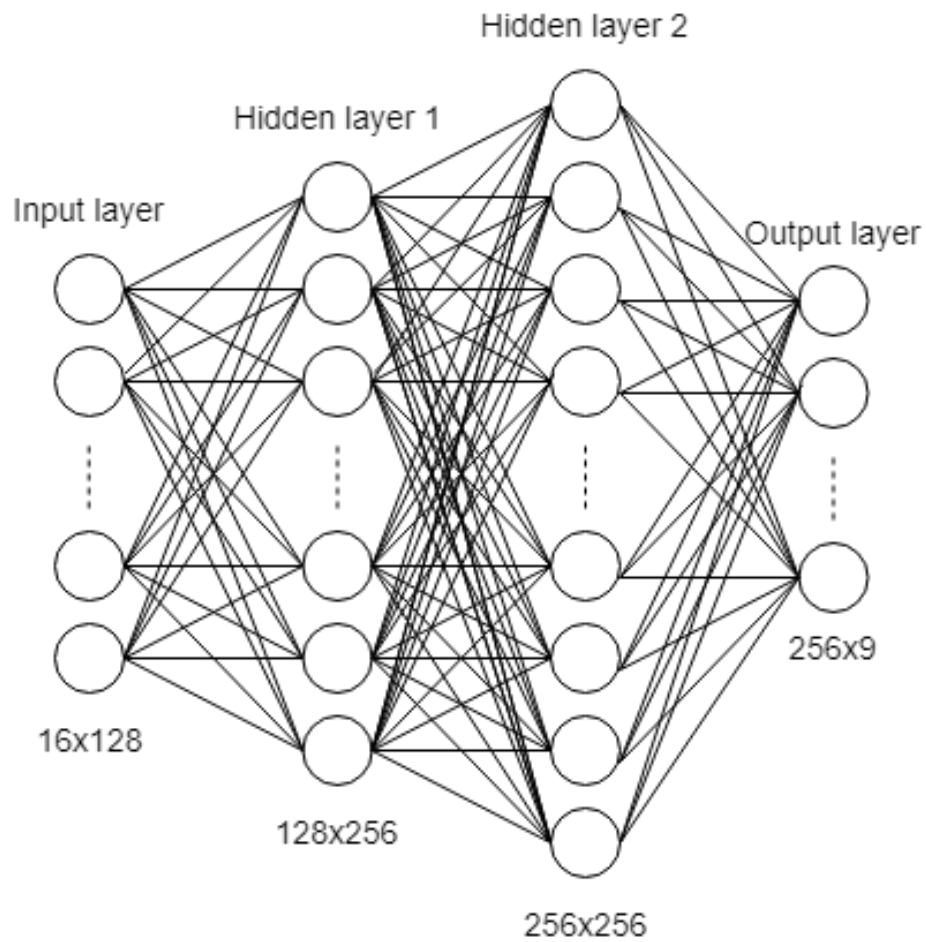


Figure 4.2: Image of model described above

When using images as input we have used the same type of network as above, with varying input sizes and experimentation with the layer sizes. We have also tried using a CNN, since we wanted to explore how it performs.

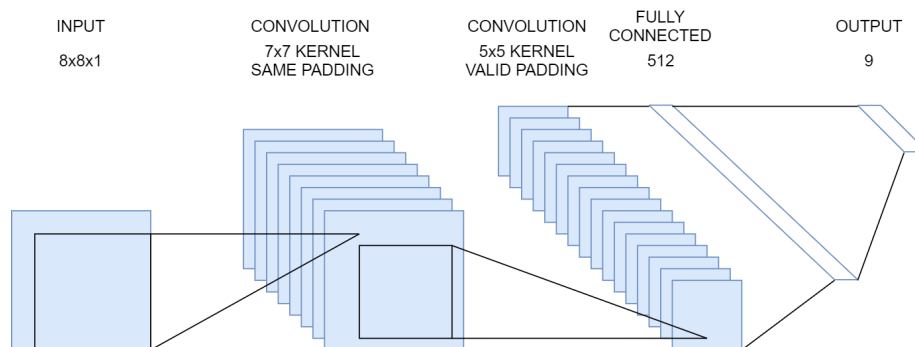


Figure 4.3: A CNN with two convolutional layers and one fully connected layer

We have mainly used a CNN with two convolutional layers before a fully connected layer. The structure can be seen in figure 4.3. A vector of normalized pixel values are sent in, reshaped back to an image and sent through the two convolutional layers. In the convolutional layers we have tried a different number of filters (kernels) and filtersizes, restricting ourselves to a max of roughly 32 total filters, as the increased number negatively affects the stability of the FPS. Since the input images were $8 \times 8 \times 1$ we used mostly 5×5 and 7×7 filters, the thought being that smaller filters wouldn't find useful patterns. We also add zero-padding in the first layer, since we don't want to decrease the already small image size further at this point. The feature maps that are being output then get flattened back to the right shape so that the following fully connected layer can process them. We use one fully connected layer with a size of 512. The output layer has a size of 9, just like in the normal neural network. In the CNN we use ReLU as the activation function for all the layers, except the output layer that has a linear activation. We have tried different architectures for the CNN based on other examples we have seen, but this structure was the one that gave the best results with a stable FPS.

It is important to understand how our models work and the differences between them to be able to understand the results chapter. This is because we will be using different models when we plot and show the results, and our experiences with the these models. Then when we for example write input: 16, hidden layer 1: 128, hidden layer 2: 256, output: 9, we are referring to the same model as the first one we showed in this chapter.

4.6 Hyperparameters and training details

There have been many variables involved in all the training sessions we ran, the amount depending on the state representations and network structures. The details and hyperparameters involved for the results will be presented in tables along with the graphs in the results chapter, to make it easier to understand which combinations gave the results shown in the graph. The content of the

tables will vary based on the type of state representation and network architecture that was used. For instance, a table will show the number of neurons in each layer, the mini batch size, the learning rate, whether or not PER was used, etc., but will also show additional details like image size and details about the convolutional layers if a CNN was used for the training session.

4.7 Class Structure

You can think of the way we setup the system as layers of abstraction. First we have the Blueprint interface, then the Python backend that handles requests from the Blueprint, then an underlying layer that handles all the DQN related algorithms. Lastly we have the neural network layer, which is the layer that handles the weights. Here is a class diagram that shows how it is all connected.

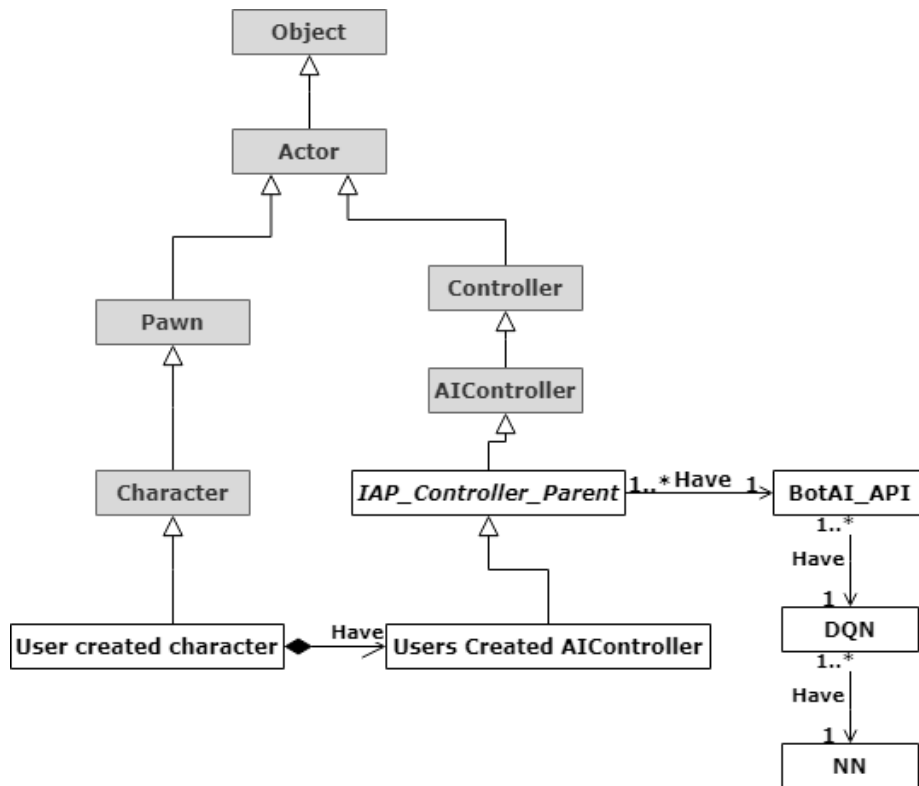


Figure 4.4: Class diagram of the classes used. The classes from UE4 are in grey, and the ones we created are in white

Note that when it comes to the class diagram it is actually a JSON system connecting the Python "BotAI_API" with the Blueprint "IAP_Controller_Parent", but it makes sense to visualize it like this to understand how it works.

4.8 Technology

This section will address which technologies we used and why we chose to use those over other options.

4.8.1 Unreal Engine 4

The use of Unreal Engine is an essential part of the task, as the goal of the thesis was first presented to us as being the development of a framework for advanced bots in Unreal Engine. The developers at Riddlebit Software use UE4, and we had experience with the engine from an earlier project. Being familiar with the engine and the AI tools it offered was useful for the early progress in development. If we were to start over and choose an engine, we might have considered Unigine (*Unigine*), more specifically the Sim SDK it offers. Since it is designed for simulation and training of AI, it may have been more efficient with our machine learning models and hardware, and thus improved the performance.

4.8.2 UnrealEnginePython

UnrealEnginePython (Kaniewski 2019b) is a plugin that embeds Python in UE4. It encapsulates Python and PIP and allows dependency plugins to be built on top of itself. Python is one of the most popular programming languages for machine learning. The language offers several libraries that simplify the development of machine learning models and testing them. The plugin allowed us to write our machine learning scripts in Python with access to the Unreal Engine environment, which has been useful for debugging.

4.8.3 TensorFlow-UE4

TensorFlow-ue4 (Kaniewski 2019a) is a plugin that enables the training and implementation of machine learning algorithms into Unreal Engine projects. It depends on the UnrealEnginePython plugin. It allows us to make our neural network in our scripts with the use of the TensorFlow library (Abadi et al. 2016). With this plugin we can send the observations of our agents inside of UE4 to the scripts, send it through our network and the return the results back to the agent. It conveniently simplifies the the communication between these two by making a component inside of the Blueprint scripting system that we can use to send and receive data. This is important to note because the AI tools inside of UE4 are mostly only available within the Blueprint scripting system, and having access to the component directly there simplified the development. Furthermore the plugin comes with examples and functions for sending images both in gray-scale and colored format.

This is the most popular and most complete plugin that allows for the use of a machine learning library with UE4. Early on we tried to get LibTorch, a library version of the PyTorch framework (Paszke et al. 2019) that uses C++, to work with UE4. Since we were concerned about performance we wanted to

use C++ because of the speed it provides and because UE4 uses C++. We couldn't manage to get it to work, because of some errors with building the project when trying to add the LibTorch files. There aren't any examples of how this can be done available at the time, and there was only one other person on the PyTorch forums that was trying the same thing, unfortunately without success. Therefore we chose to use the TensorFlow plugin as it was the next best choice, and it was already successfully combined with UE4, which was a big obstacle in itself.

4.8.4 Github

GitHub is an open source repository hosting service. It allows developers to host the source code at a remote repository and see all the changes made over time. Since there were two of us working on the development of the code we needed a version control system to save the progress remotely and conveniently merge our work. We have previously used GitLab in correlation with school projects, but chose GitHub as we have used it more in recent projects and during our spare time.

4.9 Hardware

Game engines and machine learning can both become very reliant on the hardware, so naturally the combination of the two doesn't depend any less on it. Throughout this project we worked mainly on four computers for development, two laptops and two desktop machines. In addition we had one last computer we used to test the system on for low-end computers. You can see more details about the hardware in "Appendix C: Hardware".

4.10 Development process

Because we where only two people working on the project using an agile development method like Scrum would be too much overhead. This is because Scrum methods are usually applied to teams of 4-6 people. Instead we took inspiration from the agile manifesto and took away things that would not benefit our process. It also worth mentioning that we had a 1/3 development and 2/3 research based task, and we wanted to work in sprints, therefore we had to do some changes to the agile methods to make it work for us.

We ended up with an iterative development process where at the start of each week we set a plan on what we wanted to accomplish throughout the coming week. Then we split these up in tasks for each one of us to work on. At the start of the next week we reviewed, wrote and talked about our progress, and updated our long-term plan in the Gantt diagram if any significant changes had occurred. We also had weekly meetings with our external supervisor where we showed our progress and figured out how we would continue. We did another meeting with both our internal and external supervisor every two to three weeks. In addition

we utilized stand-up meetings at the start of every day to synchronise our work. Here is our Gantt-diagram for our entire development process:

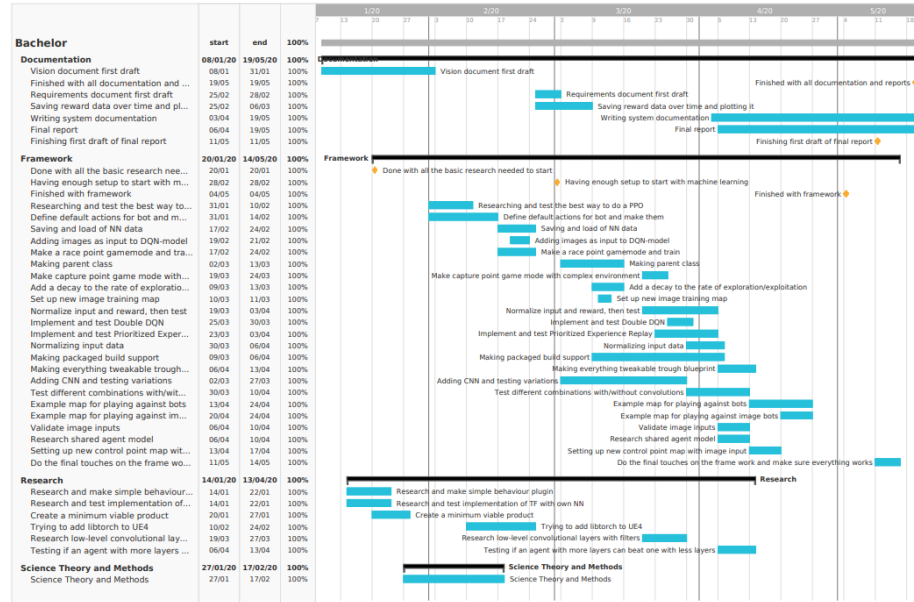


Figure 4.5: Gantt diagram

As you might notice by looking at our Gantt-diagram, we planned to use most of our first month developing a "minimum viable product". This was to show proof of concept that our method would work, check that what we were tasked could be done, and to make sure this is what our external supervisor wanted.

In the first half of our project we worked Mondays at Riddlebit's office, and the rest of the week we worked at NTNU's group rooms. We have to mention that because of the corona virus we had to work from home in the second half of our project. Though not optimal, this did not create too many difficulties as all our work could be done from home with the required hardware, which we both had. We did however have to change how we communicated, or rather our communication medium. We decided to use Discord to communicate between us when we were working, as well as with our external supervisor. Then for our meetings with both the external and internal supervisor we used Whereby. By using these tools we managed to keep our development process the same as it was before.

4.11 Distribution of roles

We have worked together on several projects during our time at NTNU, and are aware of each others strengths and weaknesses. We worked together earlier on

a project in 3D-graphics with Unreal Engine and a project in machine learning, so it was natural that we would cooperate on a project that combined the two. The distribution of roles came naturally both because we had worked together so many times before, and because of the different experience we had with topics relevant for the thesis.

Marius Fridheim had experience with Unreal Engine from a summer internship, projects during his free time, as well as a project we cooperated on. He also got some experience with machine learning during that summer internship. As mentioned earlier, we both mainly got our machine learning experience from the machine learning course at NTNU. Magomed Bakhmadov got experience with Unreal Engine through the project we cooperated on, mainly with the AI tools the engine offered. We got experience with different parts of the engine because of our role distribution on the earlier project, and that shaped some of the distribution during this project. For example, for our proof of concept Marius Fridheim researched and connected the reinforcement learning model while Magomed Bakhmadov prepared the agents with the AI tools in UE4. However, this wasn't a strict distribution, and many of the tasks overlapped.

There were many tasks that required exploring uncharted waters and therefore could be taken on by anyone of us. Tasks with development within the Unreal Engine were split based on what kind of earlier experience would be most useful, while tasks that required research on new things were split based on what each of us preferred working with at the time. This way we would be the most efficient, while also making sure that we were openly communicating about what we would prefer working with as this could change over time.

Our training and testing of agents started splitting into two categories, where one of us worked with using images as input, while the other worked with using other handpicked data as input. To summarize, the development part of the project was mostly split based on experience, while the research was split based on what we wanted to explore and started shaping over time.

5 Results

This chapter is about the results we have accumulated throughout this thesis. We will be going through the results from our research, where we will look at numerous training sessions. After that we will be looking at the system we have from the development part of our thesis. Then we will elaborate on our achievements based on the goals from the vision document. At last we will be looking at our process and the results we have gathered from the development process.

5.1 Scientific Results

Throughout this bachelor project we have run a magnitude of training sessions to both test our implementation at different stages as well as doing some comparisons. In this section we will be looking at the results we have gotten from sessions with different variations of our agents as well as different environments. These are the results we will use to formulate answers to both our research questions, "How will our agent perform in different environments?", and "How will different state representations affect training and performance?" in the discussion chapter.

5.1.1 Race Game Mode

The race game mode was design to be a simplified version of Setback's main game mode. Each player has a goal and different obstacles in the way of this goal. All players also have a weapon that they can attack the other players with. If one of them get hit they will be sent to their start location (spawn). The first player that reaches the goal will be the winner.

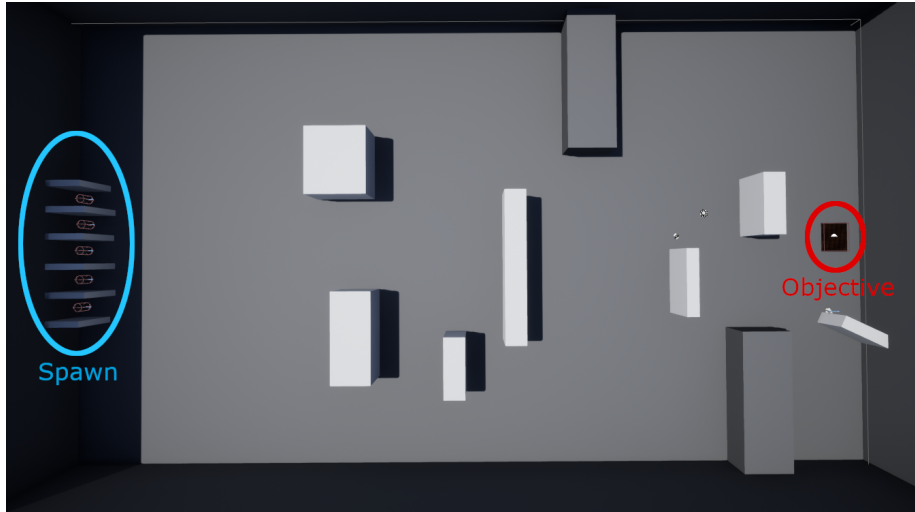


Figure 5.1: Image of the runner game mode described above

Here we can see the spawn point for the 5 agents to left in the blue circle. To the right in the red circle we have the objective that the agents need to reach to win the game.

The action space consists of 9 outputs. Each of these outputs is an evaluation of these options: rotate left (10 degrees), rotate right (10 degrees), move forward (high speed), move right (medium speed), move back (medium speed), move left (slow speed), shoot closest target, jump or move towards the goal using the navmesh (slow speed).

Handpicked Input This version of the system has handpicked values as the state that is being sent to the DQN. This handpicked state consists of an agent's distance to goal, rotation, distances of what 8 line traces around the agent with 2000 in length can hit, the direction and rotation to the enemies the agent can see, and at last we fill the vector with values for the distance and rotation of the enemies the agent can not see. We fill the vector with dummy values because we have 4 other agents playing and each agent will not be able to see all the other agents at all times. This would cause a crash if we did not fill the vector because a neural network needs to have the state size when setting up the input layer, and we always need to have a state with the same size as the input-layers columns. The idea is that eventually the DQN would learn that it does not need to act on these specific "dummy" inputs. We also make sure to set these values to something that the agent would most likely not observe and should not act on if observed. As a reference point for distances, it is roughly 7000 from the spawn points to the goal.

How we reward the agent is the same for all the examples of the handpicked input in this game mode. We reward the agent 1 point if they reach the goal,

and -0,25 if someone else reaches the goal. When an agent reaches the goal, everyone will also get sent back to spawn. Then if the agent kills someone they will gain 0,05 reward, and if an agent gets killed they will get -0,05. Also the bullets disappear after they hit any mesh with collision.

Here is a training session from this game mode with all the optimal extensions we have implemented for the DQN.

Table 2: The model and hyperparameters for 1 agent in the race mode

The model and hyperparameters	
Input layer	18
Hidden layer 1	64
Hidden layer 2	128
Output layer	9
PER	Yes
Z-Score	Yes
DDQN	No
Memory capacity	60 000
Mini batch size	128
Learning rate	0,000001
Start ϵ	1
ϵ decay rate	0,03
End ϵ	0,1
γ	0,9
Opponents	4 untrained with $\epsilon = 1$

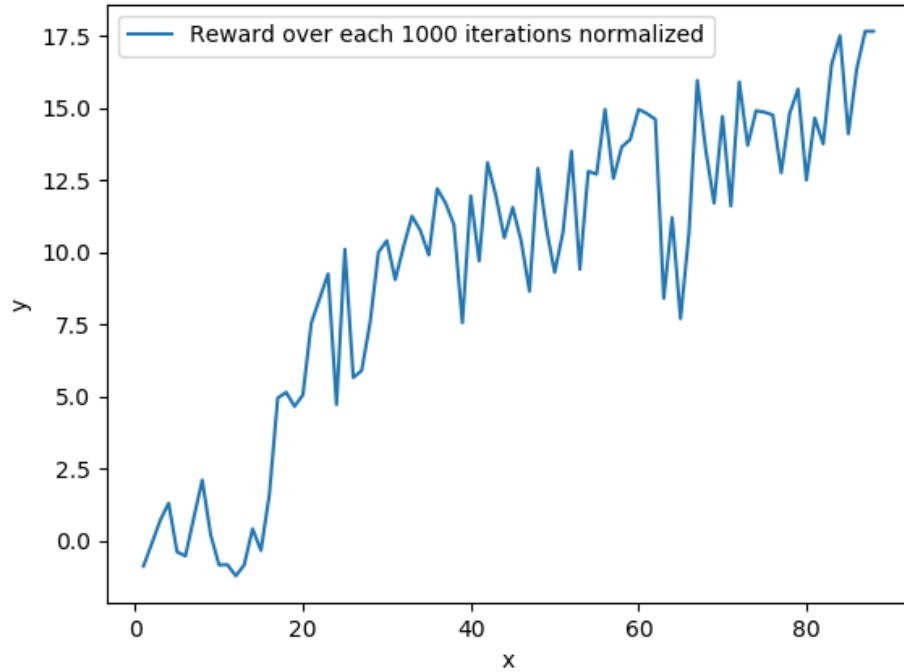


Figure 5.2: Plot of training session from 1 training agent in the race mode

Here is a video¹ of the agent trained in this plot vs random agents.

In addition we have another training session where we train all the 5 agents at the same time. Here they have the same model and hyperparameters with the exception of the mini batch size which we had to lower to 64 to be able to train this many bots at the same time, and the memory that we set to 10000. This was to make sure we did not mix samples of widely different opponent difficulties. Also we decreased simulation speed from 6 to 2, but this does not have an impact on the plot as we track reward over iterations.

¹Video of agent from figure 5.2: <https://youtu.be/t9GQJkPjQo>

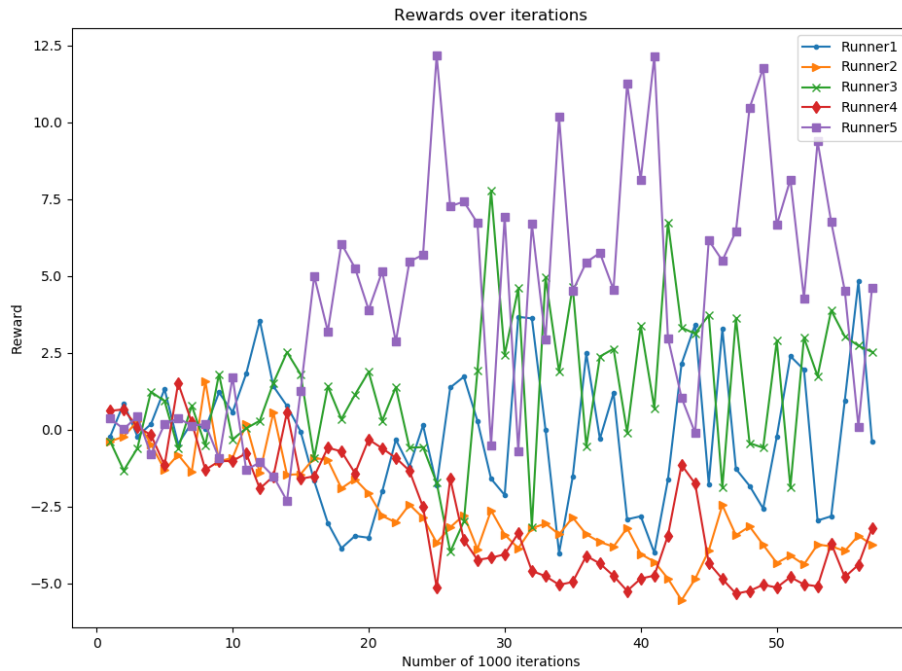


Figure 5.3: Plot of training session from 5 training agent in the race mode

Video² of the agents trained in this plot.

Note that what we see in this plot is each agent against each other, so when one gets better the other's reward will go down. This is why we won't see an increase in reward over the iterations, but they still learn as we can clearly see in this video. What we can see here is how the training process becomes while playing against other intelligent agents, as well as seeing the different strategies the agents find to gain the most reward.

Image Input This version uses images as input to the network. The bots have a camera on their head, which shows us what they are seeing at all times. The view of the bot is sent in as frames every time we send in data to the model. The data can be sent either in grayscale or in color, but the training sessions have been restricted to grayscale images as using color increases the input size to such a degree that we get extremely low FPS. We downsized the number of bots from 5 to 3, because the game would run more smoothly with fewer bots training. In all the training sessions for this mode with image input, the bots were rewarded with a score of 1 for reaching the point, a score of -1 when someone else reached the goal, a score of 0,1 when killing another bot, and a score of -0,1 when killed.

²Video of agent from figure 5.3: <https://youtu.be/-LPg1Yo98zk>

Table 3: Hyperparameters used for training session in the race mode with image input

The model and hyperparameters	
Image size	8x8
Color	grayscale
Input layer	64
Hidden layer 1	64
Hidden layer 2	32
Output layer	9
PER	Yes
Z-Score	No
DDQN	No
Memory capacity	20 000
Mini batch size	256
Learning rate	0,0000000001
Start ϵ	1
ϵ decay rate	0,005
End ϵ	0,1
γ	0,97
Opponents	2 training with larger learning rates and smaller mini batch sizes

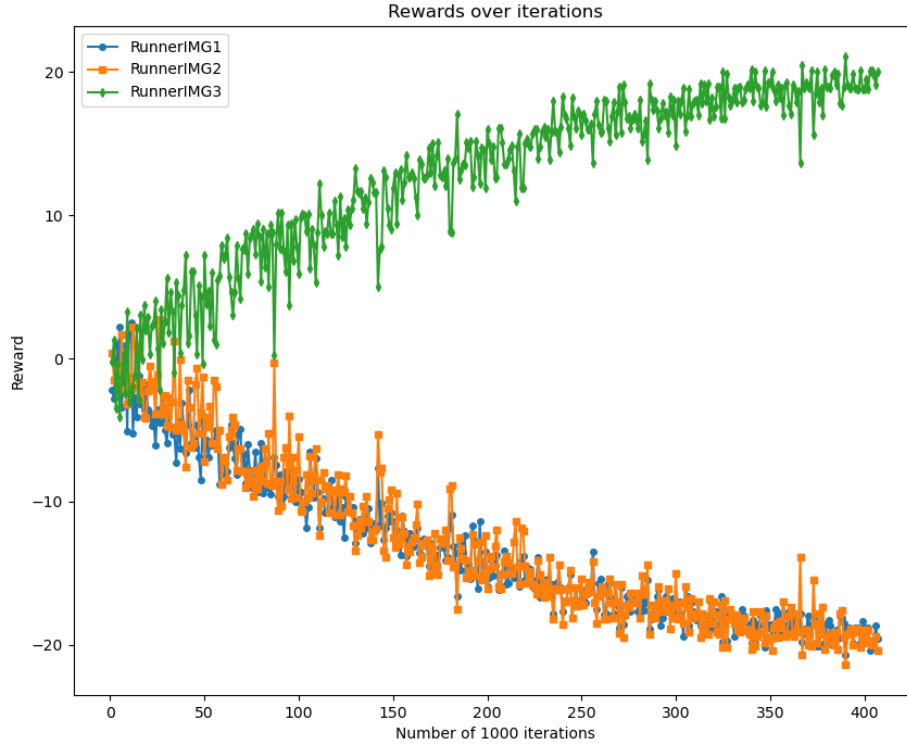


Figure 5.4: Plot of training session in the race game mode with image input

The best agent from this training session runs straight towards the point, and occasionally shoots enemies, more so when they also are running towards the point. The results of the training can be seen in this video³.

5.1.2 Control Point Game Mode

This environment was created to really challenge the agents with both complicated rules and meshes. We have a control point (marked with blue circle) in the middle of the map on a heightened platform with stairs and rocky ramps to get up. The point tracks time of each player and how long they have been on the point, and it takes 10 seconds to capture the point. In addition you can walk off the point and keep your time controlled, but if someone else is on the point while you are off you will lose 1 second on your control time every second they gain. Each player spawns in each corner of the map (marked in red), and if they get killed or someone takes the point they get sent back to their spawn location. Instead of 5 players like the last environment we have 4 players here.

³Video of agent from figure 5.4: <https://youtu.be/skoQQjibipg>



Figure 5.5: Image of the control the point environment

As we can see in figure 5.5, this is a map with much more complex meshes. This is both to see how the image version will handle an environment with different colors and meshes, and to challenge the agent's navigational behaviours.

The action space is the same as the race game mode: rotate left (10 degrees), rotate right (10 degrees), move forward (high speed), move right (medium speed), move back (medium speed), move left (slow speed), shoot closest target, jump or move towards the goal using navmesh (slow speed).

Handpicked Input

Here we have a state of 18 handpicked values. This state consists of an agent's distance to goal, rotation to goal, rotation, agent's velocity, distances of what 7 line traces around the agent with a length of 2000 can hit, time the agent has controlled the point, the direction and rotation to the enemies the agent can see, and then we fill the agents we can not see with the same type of values as

the other game modes handpicked values. As a reference point for the distance it would be 9000 if we draw a diagonal line between two of the corners on the map.

How we reward the agent is the same for all the examples of the handpicked input in this game mode. We reward the agents with 1 if they capture the point and -0,33 if someone else captures the point. If an agent gets killed it will receive -0,05 reward, and if an agent kills another player it will gain 0,05 reward.

Here is a training session from this game mode with all the optimal extensions we have implemented for the DQN.

Table 4: The model and hyperparameters for 1 agent in the control point mode

The model and hyperparameters	
Input layer	18
Hidden layer 1	128
Hidden layer 2	256
Output layer	9
PER	Yes
Z-Score	Yes
DDQN	No
Memory capacity	400 000
Mini batch size	512
Learning rate	0,000001
Start ϵ	1
ϵ decay rate	0,02
End ϵ	0,05
γ	0,9
Opponents	3 untrained with $\epsilon = 1$

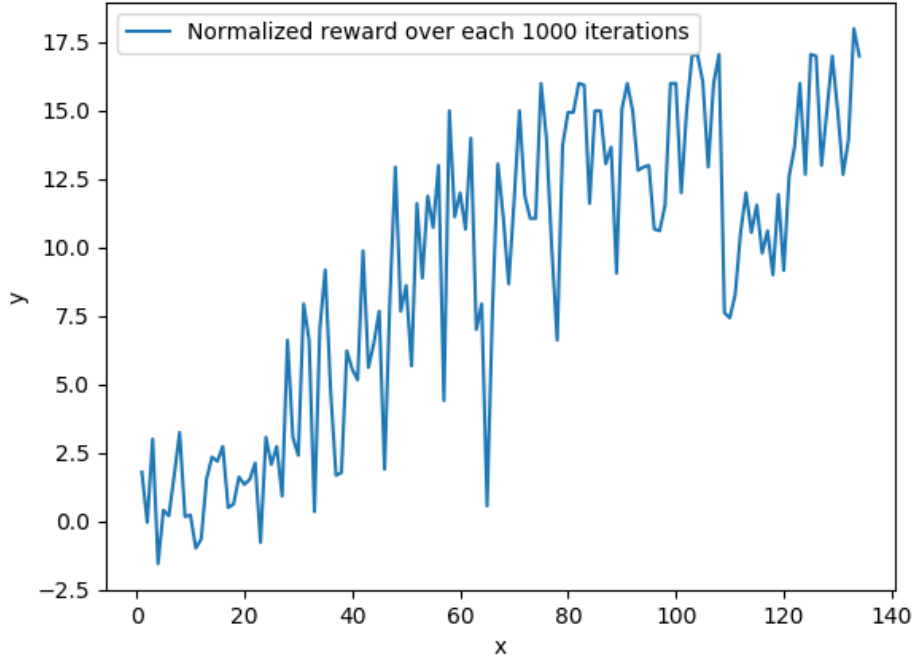


Figure 5.6: Plot of training session from 1 training agent in the control point mode

Here is a video⁴ of the agent trained in this plot vs random agents in 3x speed.

Here we have a session of 1 agent with a larger network playing against 3 agents with the network from the session above. This agent is starting out with trained weights. These weights are only trained against random agents, to make sure it learns the environment first, then how to deal with intelligent opponents. We are at times decreasing the randomness of the simple network to make sure that the complex network learns how to play against an increasingly better opponent. This creates some spikes up and down as the opponent becomes more difficult, but the important thing to notice here is that the complex network keeps beating the other simpler network.

Table 5: The model and hyperparameters for 1 complex agent in the race mode

The model and hyperparameters	
Input layer	18
Hidden layer 1	128
Hidden layer 2	256
Hidden layer 3	256

⁴Video of agent from figure 5.6: <https://youtu.be/MUBJcMz4Ibw>

Output layer	9
PER	Yes
Z-Score	Yes
DDQN	No
Memory capacity	400 000
Mini batch size	512
Learning rate	0,000002
Start ϵ	1
ϵ decay rate	0,02
End ϵ	0,2
γ	0,9
Opponents	3 trained agents with the network described above in our last training session. They also had ϵ starting at 1

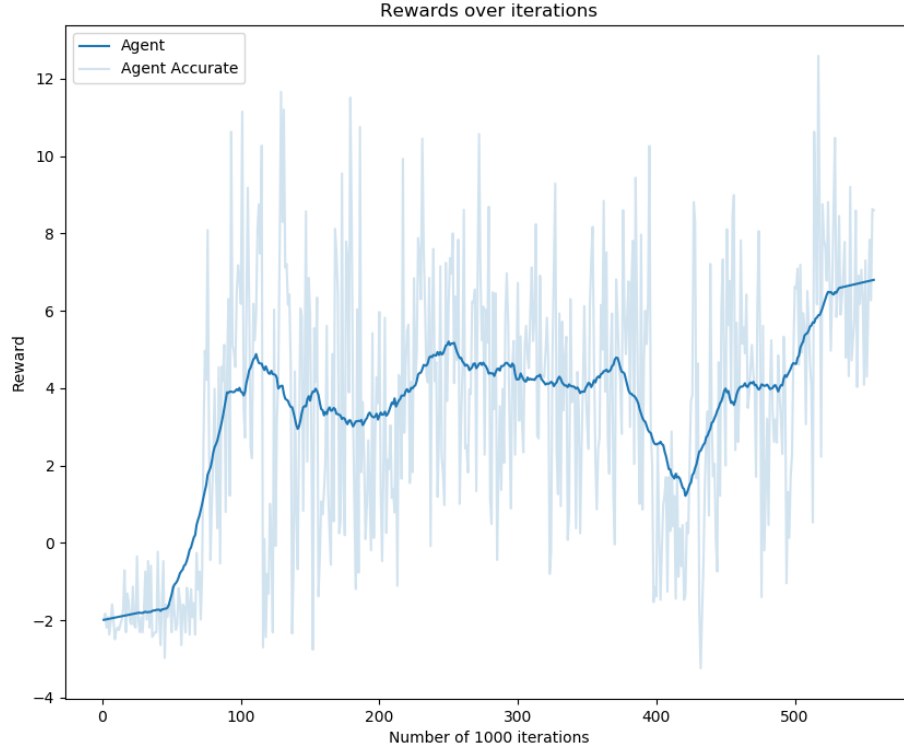


Figure 5.7: Plot of training session from 1 training agent in the control point mode vs trained agents

In this plot we have smoothed out the values to give a better understanding of the training session. The accurate values are the ones plotted in the background with transparency. Here is a video⁵ of the agent trained in this plot starting in the top right corner vs 3 copies of the agent from the previous training session shown.

At last we have a training session showing off what happens if all the agents are training with different models and hyperparameters in the control point environment.

We have 4 different neural networks with the same inputs as our example above, the size of each of them are: $[18, 64, 128, 9]$, $[18, 128, 256, 9]$, $[18, 128, 256, 256, 9]$ and $[18, 256, 9]$.

⁵Video of agent from figure 5.7: <https://youtu.be/9zMckgrrFpE>

Table 6: Hyperparameters for 4 agents in the race mode

The model and hyperparameters	
PER	Yes
Z-Score	Yes
DDQN	No
Memory capacity	40 000
Mini batch size	256
Learning rate	0,0000015
Start ϵ	1
ϵ decay rate	0,01
End ϵ	0,2
γ	0,97

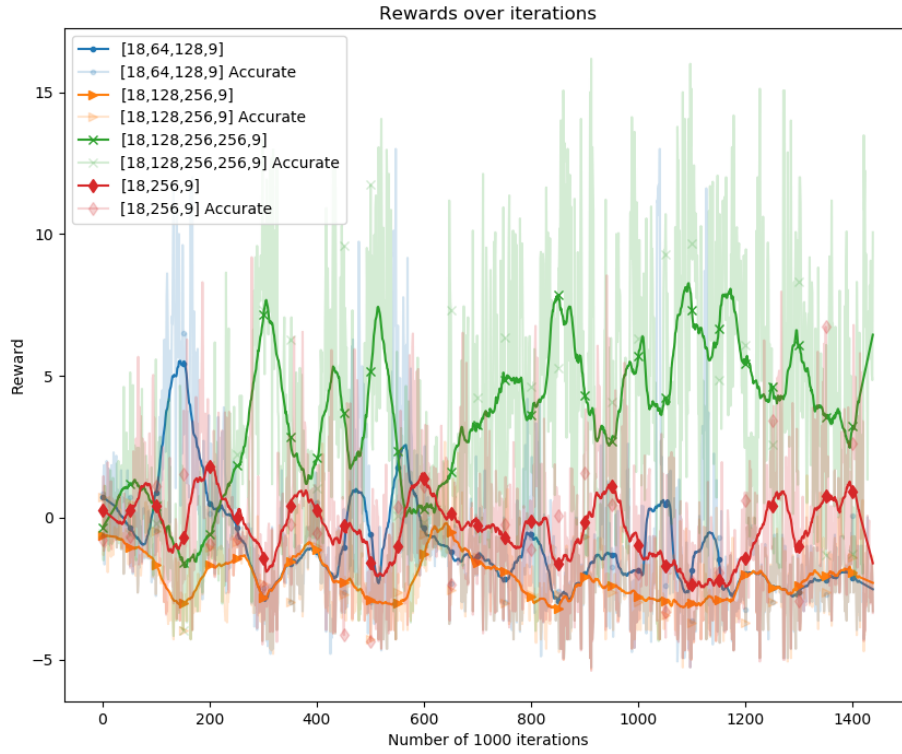


Figure 5.8: Plot of training session for all agents in the control point mode

In this plot we have the smoothed out the plot of each thousandth iteration of a training session running for 1 400 000 iterations. The transparent plot in the

background shows the accurate values. Here is a video⁶ of the agents trained in this plot. [18,64,128,9] is the agent spawning in the bottom left corner, [18,128,256,9] is the agent in the right bottom corner, [18,128,256,256,9] is the one in the top left corner, and [18,256,9] is the one in the top right corner, at the start of the video.

Image Input

Like in the race game mode, the image input on the control point mode is trained on normalized grayscale pixel values taken from what the agents are seeing. We also down-sized to 3 bots in this game mode, for the same reason as in the race game mode. The reward values are the same as those used in the version with the handpicked values. Beneath we can see one of the best results we had in this game mode with image input.

Table 7: Hyperparameters used for training session in the control point mode with image input

The model and hyperparameters	
Image size	8x8
Color	grayscale
Input layer	64
Hidden layer 1	64
Hidden layer 2	32
Output layer	9
PER	Yes
Z-Score	No
DDQN	No
Memory capacity	100 000
Mini batch size	256
Learning rate	0,0000000001
Start ϵ	1
ϵ decay rate	0,005
End ϵ	0,1
γ	0,97
Opponents	2 agents training with identical hyperparameters, except for larger learning rates

⁶Video of agent from figure 5.8: https://youtu.be/1BZ08_SqRIY

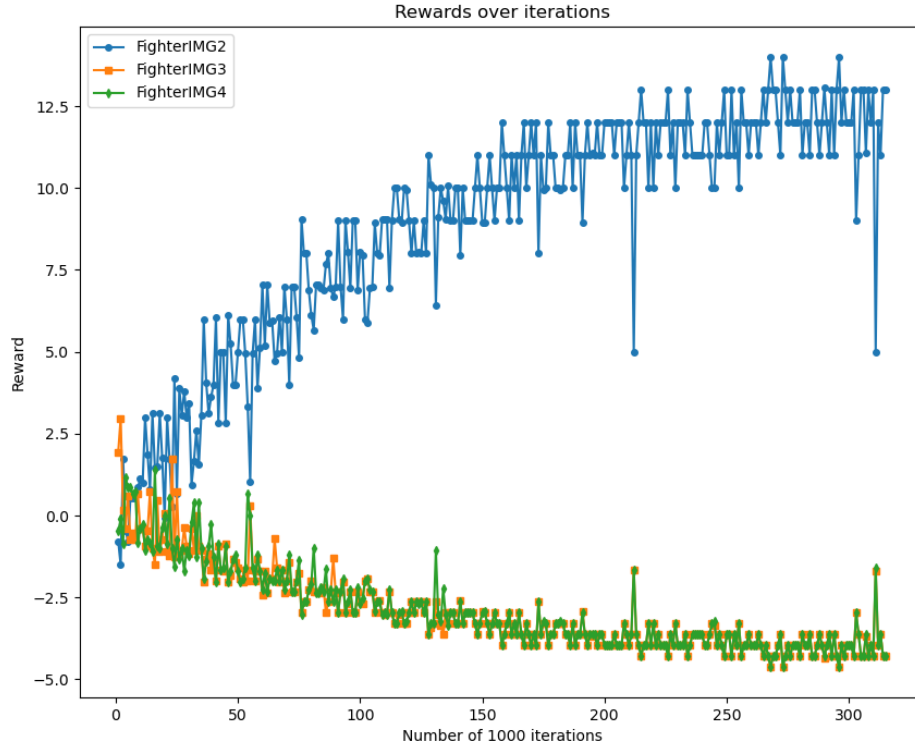


Figure 5.9: Plot of training session in the control point mode with image input

The agent showed a behaviour that was very focused on getting to the point, but extremely hesitant to shoot other agents. This behaviour can be seen very clearly when giving the same trained weights to all the agents in the environment. We see that they generally get to the point, but when there are several of them they only move around on the point, rather than shooting each other. The reason for this behaviour is that the agent that was trained started outperforming the other agents early on, and didn't get any resistance when trying to take over the point. Because the agent didn't need to defend the point for the most part of the training session, the agent doesn't understand that other agents should be eliminated to get more reward. So when several agents use these trained weights and get on the point, they will stand there trying different things until someone randomly shoots the others. The results of the training can be seen in this video⁷.

⁷Video of agent from figure 5.9: <https://youtu.be/BGOQFHDh9no>

5.1.3 Double Deep Q-Network

Here we have two different results with the same network, with the only difference being one uses Double DQN and the other using DQN. The environment is the control point mode, and we have an action space of 9, the same actions that we have in 5.1.1. The state consists of 16 inputs: distance to goal, agent's rotation, distances of what 8 line traces around the agent with a length of 2000 can hit, time the agent has, the direction and rotation to the enemies the agent can see, and then we fill in the agents we can not see with dummy values.

Table 8: The model and hyperparameters for Double DQN

The model and hyperparameters	
Input layer	16
Hidden layer 1	64
Hidden layer 2	256
Output layer	9
PER	No
Z-Score	No
DDQN	Yes
Memory capacity	20 000
Mini batch size	256
Learning rate	0,0000001
Start ϵ	1
ϵ decay rate	0,01
End ϵ	0,03
γ	0,9
Opponents	3 untrained with $\epsilon = 1$

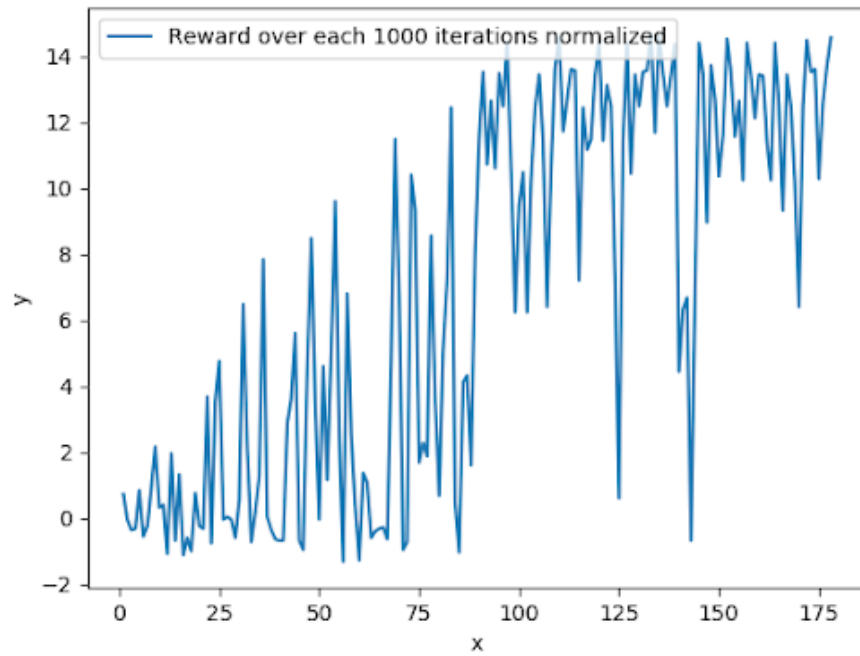


Figure 5.10: Plot of training session using Double DQN

As a reference point for the discussion we have the same exact model and hyperparameters just without Double DQN here:

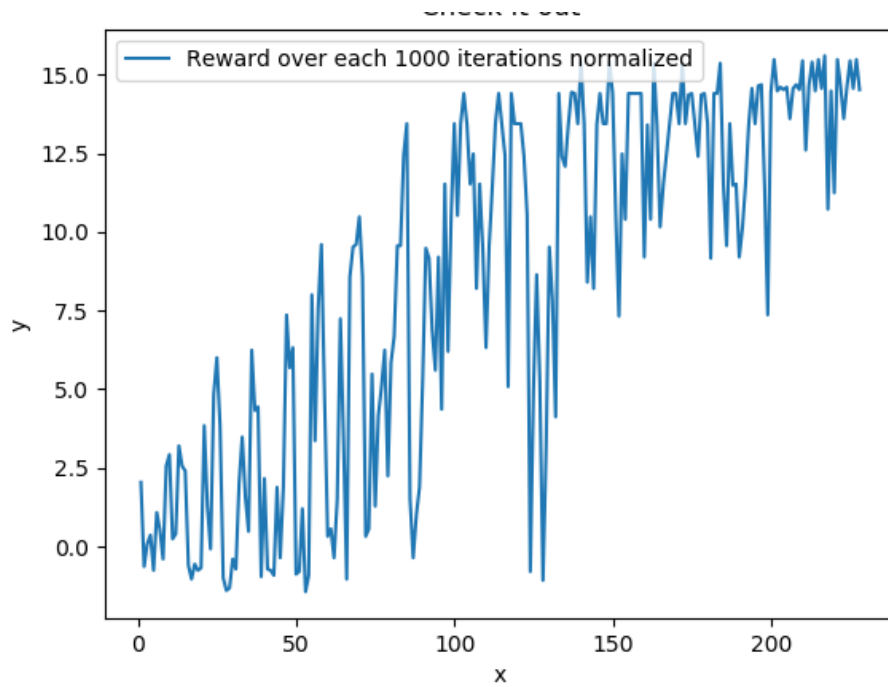


Figure 5.11: Plot of training session using DQN

5.1.4 Prioritized Experience Replay

Another improvement we added was Prioritized Experience Replay. Here we have the same exact model, hyperparameters and environment that we use in the Double DQN chapter above, but here we add PER.

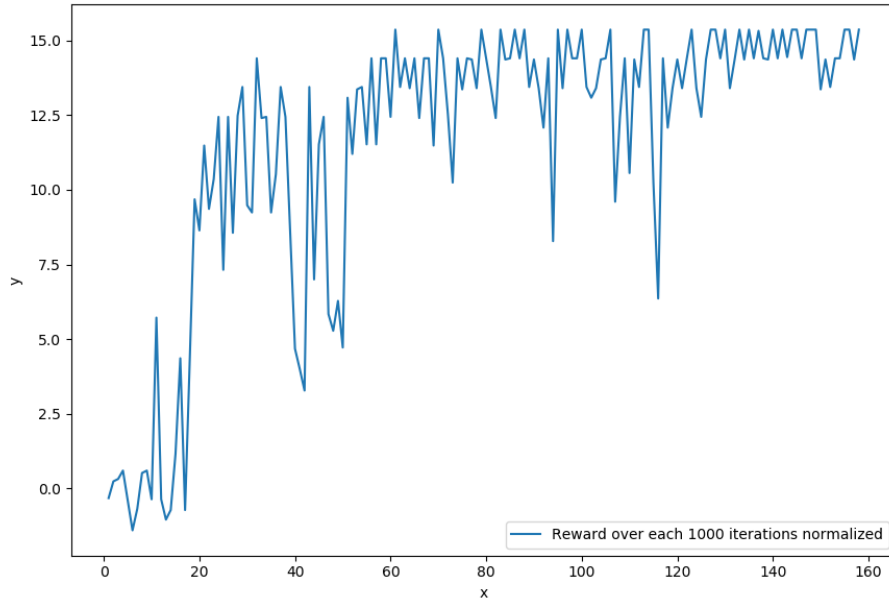


Figure 5.12: Plot of training session using Double DQN and PER

5.1.5 Z-score

As mentioned in the method chapter we decided to use z-score normalization. We have two plots that we need to discuss the impact of z-score normalization. First we have one with z-score, then we have the exact same model and hyperparameters, with the exception of learning rate, without z-score. Both are used in the control point mode.

Table 9: The model and hyperparameters for z-score

The model and hyperparameters	
Input layer	16
Hidden layer 1	128
Hidden layer 2	256
Output layer	9
PER	Yes
Z-Score	Yes
DDQN	No
Memory capacity	400 000
Mini batch size	512
Learning rate	0,000002
Start ϵ	1
ϵ decay rate	0,02

End ϵ	0,1
γ	0,9
Opponents	3 untrained with $\epsilon = 1$

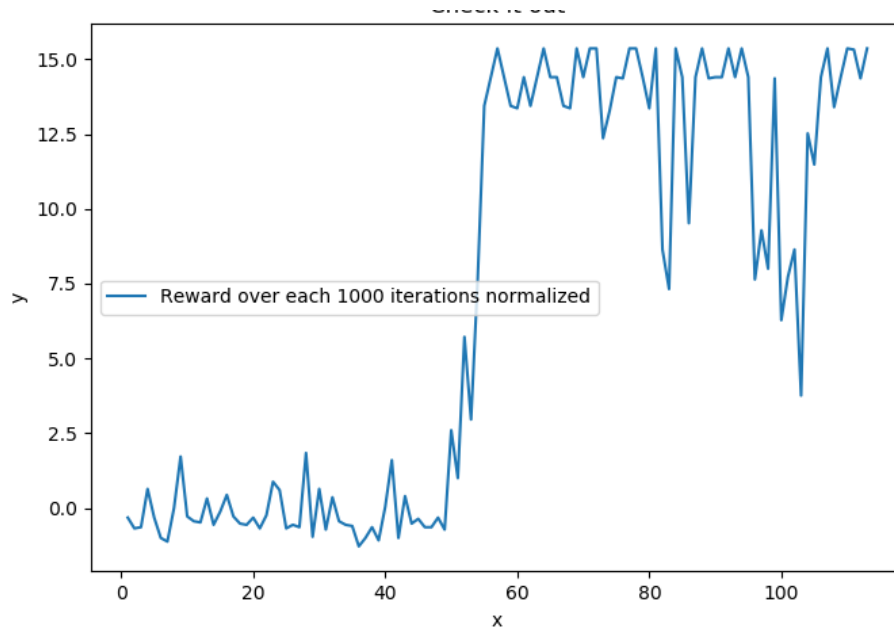


Figure 5.13: Plot of training session using z-score

Here is the one without z-score and a learning rate of 0,0000001. There is a difference in the learning rate, because the learning rate will have to change with the size of the values in the state.

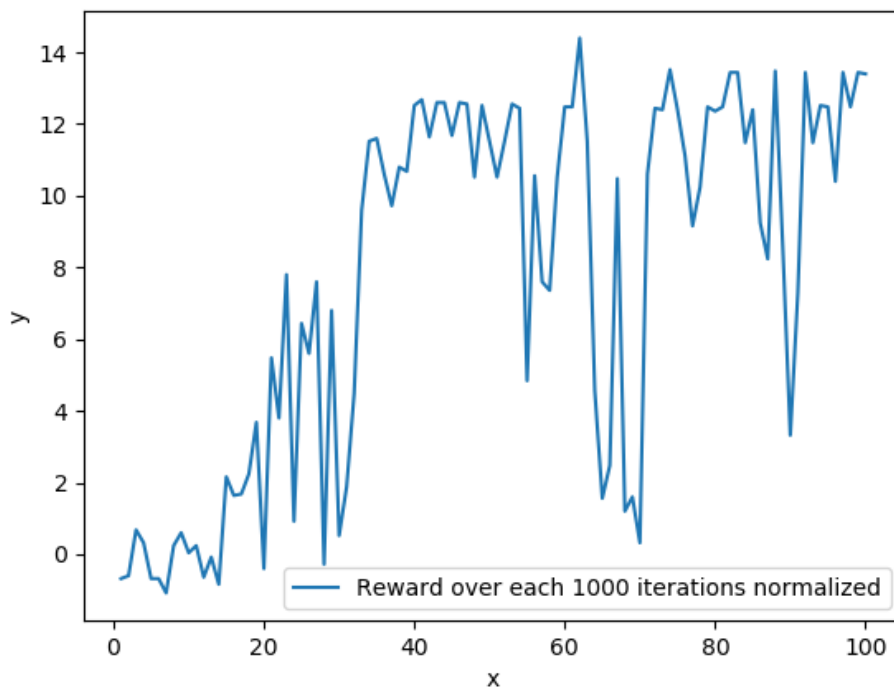


Figure 5.14: Plot of training session without z-score

5.1.6 Convolutional Neural Network

We had to prioritize a game mode when running training sessions with a CNN, and decided on the race mode. The reason for this is that the race mode is simpler, making it likelier that we would get results there quicker. The results with a CNN on the control point mode are less promising, because we had less time and fewer training sessions. Below we can see the results of training an agent with a CNN using the same hyperparameters on the two different modes.

Table 10: Hyperparameters used for the training sessions plotted in figure 5.15 and 5.16.

The model and hyperparameters	
Image size	8x8
Color	gray scale
Input layer	64
Convolutional layer 1	filters: 16, kernel-size: 7, strides: 1, padding: SAME, activation function: RELU

Convolutional layer 2	filters: 8, kernel-size: 5, strides: 1, padding: VALID, activation function: RELU
Fully connected layer 1	512
Output layer	9
PER	Yes
Z-Score	No
DDQN	No
Memory capacity	200 000
Mini batch size	256
Learning rate	0,0000001
Start ϵ	1
ϵ decay rate	0,02
End ϵ	0,1
γ	0,99
Opponents	2 untrained with $\epsilon = 1$

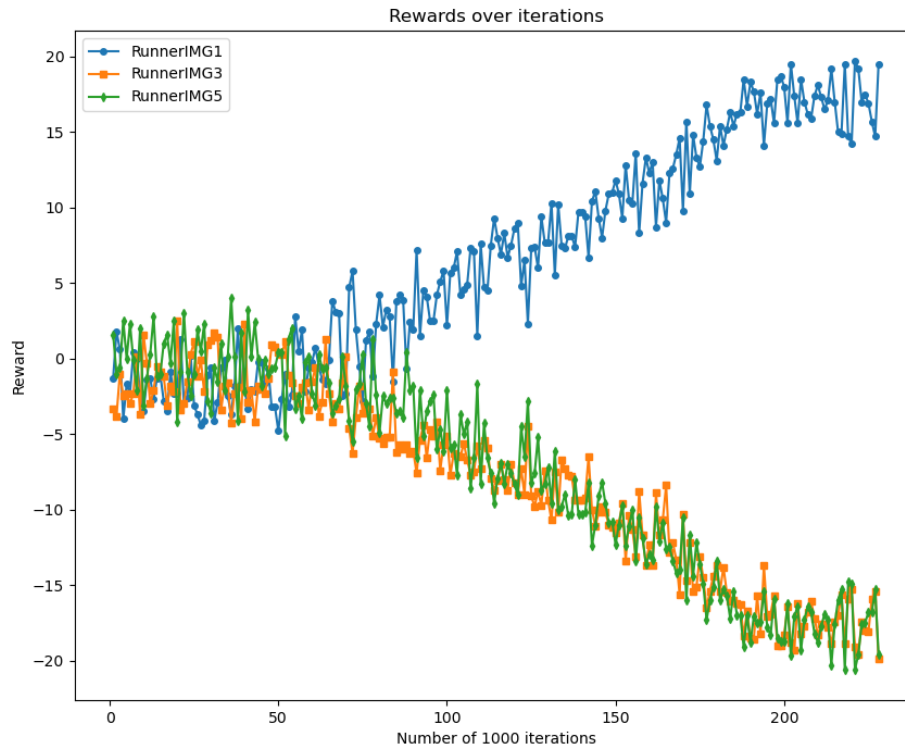


Figure 5.15: Plot of training session in the race game mode with CNN

The trained agent behaves very similar to the one trained with a dense neural network, but seems to be more flexible when it comes to navigating towards the goal and able to recover better when different circumstances make him stray away from the optimal path from the start.

A small clip showing the trained agent in action can be seen here⁸.

⁸Video of agent from figure 5.15: <https://youtu.be/rQqG8x8hTxw>

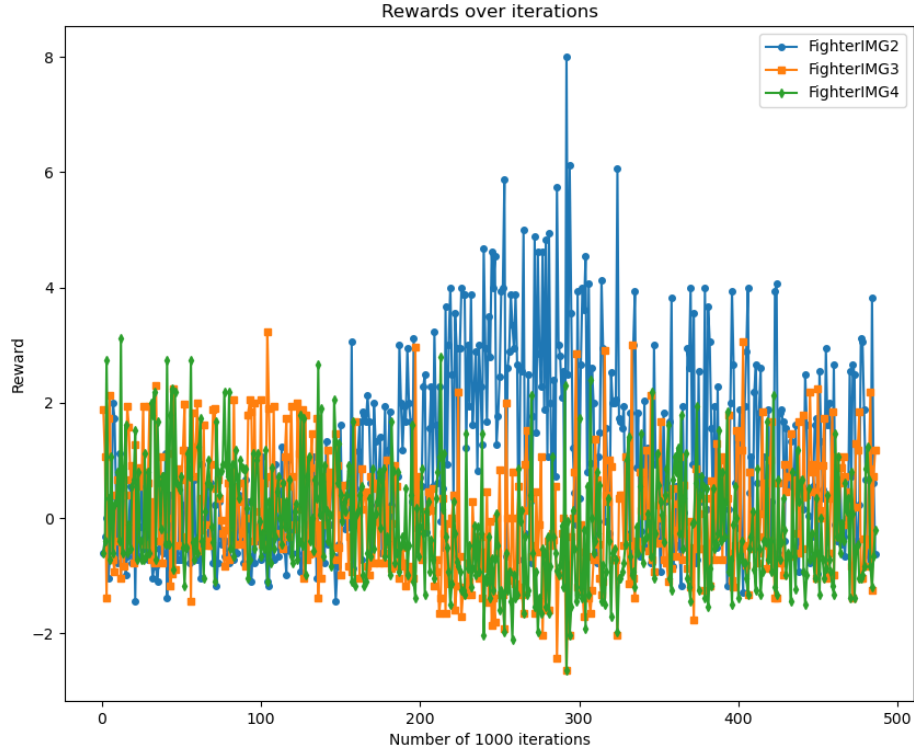


Figure 5.16: Plot of training session in the control point mode with CNN

As we can see in figure 5.16, the agent didn't manage to consistently increase the rewards over time. While several adjustments are likely to improve the results, we assume that tweaks to the learning rate and the exploration rate would result in a more stable graph, based on the spikes the graph shows.

5.2 Product Results

In this chapter we will look at the plugin that we developed in correlation with our thesis statement, "How can we combine DQN with the existing AI tools in UE4?".

5.2.1 Product

The system that was delivered on 20.05.20, is a plugin for UE4. The purpose of the plugin is to show how machine learning can be combined with AI tools, by giving users an interface inside of UE4 to simplify the process of setting up DQN agents that can be trained in a UE4 environment.

5.2.2 Functionality

The plugin has functionality for connecting a DQN model to an agent in a UE4 environment. At the time of delivery, 20.05.20, the following functionality, written in correlation with goals in the vision document, is available in the plugin:

- **Default examples**
The user has access to default examples with environments and agents that have already been set up and tested.
- **Controller connectable to new bots**
The plugin comes with a parent controller class that can be inherited. The user can set up their own environments and agents that can be connected to a controller that inherits from the parent controller.
- **Set up reward functions**
The user can define how an agent receives a reward, so that they can train the agent to show a certain behaviour.
- **Send percept data**
The user can decide which data they want to send as input. The default examples show how both handpicked data and images can be used.
- **Define actions**
The user can set up the code for an action they want their agent to do, and simply connect this as a possible output action by inheriting from the parent controller class.
- **Save and load models**
The plugin automatically saves the TensorFlow session at an interval. The training progress can be saved and loaded to continue training.
- **Configure hyperparameters**
The user can decide the structure of the DQN model and neural network by changing the hyperparameters that are available and editable inside of UE4.
- **Simulate and speed up training**
The user can watch as the agent interact with the environment and can speed up the game speed so that training takes less time. This is limited by how many inputs the agents have, the complexity of the model architecture and the hardware that the user has access to.

5.2.3 Design

Since this is a plugin, the design of the product will refer to how we have structured the plugin and how we have used the AI-tools in UE4 in combination with the DQN.

Throughout the project we had several ideas about how the plugin would be structured, but over time we gravitated towards a structure where we implement a superclass controller that inherits from the `AIController` class in UE4. Users can inherit from this parent controller and modify it to fit their environment. This structure makes it simple to run the default examples, while making it easy for users to make their own agents. Since the `AIController` class in UE4 is fundamental to building bots, it was natural that we would use this tool in combination with the DQN.

The parent controller gives the user access to a details panel to adjust hyperparameters and other variables. This panel will be available to users when they set up their own controller or use one of the ones we have made for one of the examples. Hyperparameters like learning rate, number of hidden layers and their sizes, memory capacity, batch size, etc., as well as some other variables like the respawn position of the agent can be changed. This allows the users to change the architecture of the neural network through a familiar interface.

We have used stimuli sources and perceptions to check when an agent actually can see another agent, and make a list of the currently visible agents. We needed this for achieving our goal of making default actions for an agent to perform (as a part of the examples). A list of which agents our agent can currently see is useful when the deciding who the agents should aim at when deciding to shoot.

Because the controllers used to control the agents inside of the UE4 environment don't require us to specify which character it will be possessing, we can arbitrarily refer to the character and prepare actions that the character will be able to perform. In the controller we define all the movement actions. Most of the movement actions, like moving forward, backwards, right and left, don't require pathfinding. However, the action that moves the agent towards the objective uses the navmesh to effectively navigate in the direction of the goal. This means that the agents don't have to learn how to navigate towards the point, but rather that the action that moves them closer is usually favourable.

5.3 Engineering Results

This section elaborates on our engineering results by comparing our results to goals we had at the start of the project. The goals that are discussed here are based on what we wrote in our vision document. We will describe each goal and to which degree it was accomplished.

5.3.1 Goals

- **The plugin should have support for training agents inside of UE4 with reinforcement learning.**

The purpose of the plugin is to allow users of UE4 to use reinforcement learning on agents inside of a UE4 environment. A user can either use a default example, or make their own environment and make a controller that inherits from the parent controller to set up an agent that uses a

DQN. On delivery, the plugin comes with two simple game modes that showcase how the plugin can be used. The examples function both as a test to see if everything has been set up correctly when downloading the plugin, as well as functioning as examples when the user wants to create their own environments and agents.

- **All the functionality should be available through downloading a UE4 plugin.**

All the functionality for using the DQN will be available when the plugin is downloaded (assuming that the user has installed the necessary prerequisites). The purpose of the plugin is to simplify using DQN with agents in UE4, so it is important that the download and usage isn't too complicated.

- **The plugin shouldn't require a background in machine learning from the user.**

The plugin should be easy to use and set up, so that you won't have to set up all the code for the DQN as well as the connections to it. This is one of the reasons that the plugin comes with examples that can be used without having to set up anything.

- **The user should have access to the complete structure of the plugin.**

The whole code and structure should be open source. Making everything available makes it easier for users to understand what happens behind the scenes, as well as giving users the option to make changes in the scripts if they want to make their own improvements and additions. On delivery, everything will be available to the user.

5.4 Administrative Results

This section elaborates more on our development process and what's been written in the project handbook and our progress plan.

5.4.1 Project handbook

The project handbook contains the Gantt diagram and weekly updates that elaborate on what each of us spent time on during that week, and whether or not we achieved our goals. The Gantt diagram is mainly split into three parts, development of the plugin, research and documentation. As we mentioned in chapter 4.10, we decided what we wanted to achieve each week (or over several weeks depending on the size of the task) and planned this out in the Gantt diagram accordingly. The diagram shows how we worked together on many of the tasks early in the project to form a baseline for the project, and started working more independently as time passed. This change starts showing in the Gantt diagram after we reached our earliest milestone, which was to make an MVP, thus forming the baseline of the project.

5.4.2 Development methodology

We used some principles from agile development when working on our project. Since our project is split into $1/3$ development and $2/3$ research, it would be difficult to strictly follow a development methodology, so we only used the principles that we felt were practical for our case. Our project was prone to quick changes and the specifics of many of the tasks were uncertain before we started researching them, so we often planned the Gantt diagram with more general descriptions. The Gantt diagram and the weekly updates give an overview of how we worked and show whether or not things always went according to plan.

6 Discussion

Here we will assess the results we got from development and research. We will first discuss the scientific results from all the training sessions we ran. Then we discuss the plugin and reflect on the choices we made during the development. Finally, we quickly discuss our development methodology and how the collaboration went.

6.1 Scientific Results

In this section we will discuss our scientific results, or more accurately the results we have gotten from running numerous training session with different algorithms, hyperparameters, environments and state representations. Throughout our discussion we hope to show what worked, what did not, and why that might be.

6.1.1 The Environments

Based on the results we can see that both the image and handpicked values state learn fairly quickly from the race game mode, which is perfect to test if a DQN model works or not. One problem with this environment was that it was too simple as the agent quickly learned that the best cause of action was to look to the side and shoot while just running towards the goal. You can see this behaviour in this video⁹. This caused us not to be able to see any complex behavior from the agent, as that was not necessary to accumulate the maximum reward possible. Another issue was when we train agents against other smarter agents they might get stuck in spawn killing each other, as that is the best way to stop the other agents from getting to the objective, and get some score. You can see this with one of the agents in this video¹⁰. This could have been solved by adding a little timer before re-spawning the killed agent so that it could not be consistently killed by another agent to accumulate reward. A problem with this solution was finding a way to tell a neural network that it is in time-out for a set amount of time before any action it predicts will actually be performed.

We created our second environment both to test our agent in a different environment and to solve the problems mentioned above. When looking at the results from the control point environment, the first thing we see is a much longer time to reach the "best" policy. This makes sense as the environment is much more complicated, and the best policy seems to be constantly changing based on the other agents that we train against. The best agent we have managed to train is the one from figure 5.7. This agent learned when to run to the point, when to stop, when to shoot at enemies and when to spin its field of view around to get more information, as we can see here¹¹ with the agent starting in the top

⁹Video of agent showing described behaviour: "<https://youtu.be/-179XsIUD3Y?t=17>"

¹⁰Video of agent from figure 5.3 starting at 23 seconds: <https://youtu.be/LPg1Yo98zk?t=23>

¹¹Video of agent from figure 5.7: <https://youtu.be/9zMckgrrFpE>

right corner. This environment has a lot of advantages, but it also introduces one difficulty. When each agent starts in each corner they won't be able to see each other, meaning they will be missing a part of the percept more often than in the other environment. This can cause problems during training. Let's say we have two agents playing against each other, first our agents run to the point and learn that this is good, then it tries to run to the point again, but now the other agent will reach it first. This means that it will be first learning that moving towards the point is good, then it might do the right thing, but get beaten and learn that moving towards the point is not good. Of course with time the number of times it does something good and reaches the point will outweigh the times it gets beaten by random circumstances. Interestingly, the same thing happens when people play video games. You might do everything correct based on the information you have, but still lose. Just as with humans this makes it more difficult to learn, but not impossible as we see through our results.

6.1.2 Comparing State Representation

Through this thesis we have been using two very different ways to represent a state. One uses handpicked values that is picked specifically based on how valuable we think they are for the agent to understand the environment. The other uses each pixel from an image as values in the state. Depending on whenever it is grayscale or RGBA, it uses 1 or 4 values for each pixel. The most noticeable difference between these two state representations is the training time. When we look at figure 5.2 and 5.4, we can see that it takes 90 000 iterations to get 17 in score with handpicked input, but it takes roughly 300 000 iterations for the image version in the same environment.

Even though the handpicked value version has a faster training time, images seem much more consistent. Meaning we do not have the spikes in score that we typically see with handpicked input. This might be because the image version has a much larger input size, which makes the input layer much bigger for images than handpicked. Another impacting factor could be because images have a longer training time, which also often means they have a lesser epsilon decay rate. This means that we will have very small drops in epsilon, that could lead to a more fluent training session.

Based on the behavior we see in the different videos we notice that the most intelligent agent we have managed to train is the one from plot 5.7. This agent was trained using handpicked values, so it stands to reason that the handpicked state might give better results than images. We also have to remember here that images need longer time to train, so we might be able to reach the same results with better hardware and longer training time. We also noticed that using CNN became very heavy for our machine and we could not speed up the training process while using CNN. This means that if we had better hardware and more time we might have been able to train a better agent using images and CNN. By using our hardware, which is pretty decent, it seems that the

handpicked version gives better results.

Even though the handpicked version might give better results, images have one big advantage above handpicked values. Images can be used in different environments without changing the input. This means that we can use the same weights from one environment as a base in another when we start the training session, which might give better starting results than just using random weights. We could also just use the same agent and weights in similar environment. On the other hand the handpicked version needs to change the input unless the environments are completely equal in terms of input, which it rarely is. When we change the input for different environments it means that we can not use the same weights, as we have to change the input layer to accommodate for the new state size.

6.1.3 The Training Session

How one might go about training the agent is a very interesting topic, as there are many tactics that completely change the results. We have also done a lot of research and testing on how one might do this with our system in UE4. If we take the control point game mode as an example, we can see our two agents from figure 5.6 and 5.7 are very different in how they behave. One has only learned to get to the point and stay on it, while the other one learned to get to the point, then shoot the other agents that tries to contest it, as well as spinning the field of view around to get more information and see other agents. If we look at the two plots we can see that it looks like 5.6 had the better training process because he gradually learns over time, but 5.7 is actually the one that learned the most. This is because of the way we train them. 5.6 was just always training against agents that performed random actions. This made just running to the point the best policy. However, 5.7 was first trained with random agents to make sure it learned the rules of the environment. Then we made the opponents more intelligent until the agent learned to play against them, and then we repeated this process of upping the intelligence of the opponents until we had an agent we were satisfied with.

Another approach to create agents that can play against more advanced opponents was to train all the agents playing the environment at the same time. However, this seemed to be a little dependent on the environment and how easy it is to give frequent reward. Based on our experience it seemed that when we let the opponents get better without controlling it, they circled between getting better and worse for a very long time. We can see this in plot 5.8. The most likely cause of this is what we discussed in the last section in chapter 6.1.1, agents get punished for what might be the right action. This is not to say that an agent can not learn while playing against other agents that also learns, it just takes longer time than the method proposed above, and might not always give consistent results. We have also found that it is not beneficial to have a big memory size when multiple agents are training at the same time. This is because we will be mixing samples of agents with different intelligence and

strategies. If the memory size would be large it would almost be mixing different environments as the opponents for an agent are a part of the environment to that agent. Instead having smaller memory size when training will let the agent only sample experiences with roughly the same opponents.

6.1.4 Double Deep Q-Network

When we compare an agent using Double DQN and one using normal DQN, the differences we see are within the variation arising from random weight initialization. This means that for the environments we have tested Double DQN in, it does not provide any notable difference in terms of learning rate or score. Double DQN also requires an extra network that we need to run predictions on. This causes Double DQN to use roughly twice the computer power that DQN does. This is why we usually do not use Double in our other plots, so that we can run the simulation faster. This does not mean that Double DQN is useless. It only means that for our test Double DQN did not perform good enough that it was worth slowing down simulation time to use it. In another environment where the overestimation of q-values is more crucial, Double DQN will most likely perform better. This is why we implemented an option where the user can choose to use Double or normal DQN just by ticking a checkbox in the Blueprint class.

6.1.5 Prioritized Experience Replay

If you take a look at the plots from figure 5.11 and 5.12 we can clearly see that 5.12, the one with prioritized experience replay, learns much faster. The one without PER (5.11) uses roughly 150 000 iterations to get 15 in score, while the one with PER (5.12) uses roughly 65 000 iterations. We tested this multiple times and every time our system with PER learns at least twice as fast as without PER. This is why we did not even make it an option to turn PER off, as it's just a better way of sampling data compared to the random approach suggested in the DQN paper (Mnih, Kavukcuoglu, et al. 2013).

6.1.6 Z-score

Looking at the plots from figure 5.13 and 5.14, we see that 5.14 that does not use z-score normalization has more fluctuations in the score compared to 5.13 that uses z-score. These fluctuations are not major, but we can see that the training process becomes more stable with z-score than without. It is also worth noting that z-score normalization will give all the different observations the same importance when training. This means that in a number of cases if some observations actually are more important than others it could train faster using non-normalized input. We can see this here with the agent not using z-score being a little better in the first 50 000 iterations of the training session. However, we see that the one using z-score beats the other one after 60 000 iterations. This happens because when all the input values are prioritized equally, it creates a state with more information for the neural network than

values that are very different in size. This should eventually, after some training, give a better agent. We can see this here where the highs of the agent without z-score is 13, while with z-score it reaches 15. We can see that z-score gives a better ending result, so generally we want to use it. However, if the user for some reason doesn't want to use z-score or maybe wants to do their own normalization, we created a checkbox that can be ticked on and off depending on if one wants to use z-score or not.

6.1.7 Convolutional Neural Network

The addition of a CNN was an attempt to improve the results with image input. While the results show that there is potential for this, the opportunity to use bigger images with a more complex layer architecture could have led to more interesting discoveries in the behaviour of the agents. We used mostly 8×8 grayscale images as input, because anything bigger would have an increasingly heavier toll on the FPS. Since a CNN is used to recognise features in an image, it would be easier to find more recognisable differences between different states. The number of training sessions with a CNN were limited, relative to the number of sessions we ran with a fully-connected neural network. This was because the simulation speed needed to be reduced further, in proportion to the increase in size and complexity of the network. Furthermore, a CNN introduces more hyperparameters that need to be accounted for, which means that there are more combinations that can lead to improved results.

When we compare the results from figure 5.4 and 5.15, we can see that the agent using the CNN is increasing the reward towards similar values as the agent without a CNN. By watching the behaviour from the agent after training, we have also seen that the one with a CNN seems to be more capable to adapting to different parts of the map. We wanted to test this further by training an agent using a CNN on the control point map, and then using these values on all the agents on that map. As we can see in figure 5.5, the different corners of the map have variations in recognisable features. This means that an agent that is trained by respawning in the top right corner, won't necessarily be as effective when the starting point is changed. Based on the resulting behaviour we got from training an agent with a CNN in the race mode, it seems plausible that with more time to train an agent with better results in the control point mode, we would observe a behaviour that is better able to adapt to different spawn points.

While we can't say that there is an evident improvement when using a CNN based on our results, there is potential for the addition being beneficial, especially if future additions to the plugin allow for larger input sizes with stable performance.

6.2 Product Results

The product has all the core functionality that we felt would be most relevant for the developers that could use the plugin. In this section we will go into more detail about the thought process behind the choice of functionality and implementation.

6.2.1 Functionality and design

We had to make several choices regarding the direction we wanted to go with the project. Early on we decided that making a UE4 plugin would both make the project simpler to download and use for others, as well as clearly separating what we made from everything else inside of UE4.

Most of the functionality that has been added has had the purpose of making it easier to set up the DQN, focusing on minimizing the required knowledge about machine learning from the user. This is not to say that the user doesn't need to have any knowledge about machine learning, but rather that the user doesn't need to have as much technical experience. We focused on making the user need more knowledge about UE4 rather than machine learning in practice. The user needs a solid understanding of how the AI tools in UE4 work to really understand how everything in the plugin works, and one might consider this a weakness. However, since the plugin is meant for people that develop in UE4 and want to use reinforcement learning in their projects, this isn't necessarily a weakness as the user group we're targeting should have sufficient enough experience with UE4. Earlier in the project we didn't have the parent controller class setup, as we were considering whether or not the users should go inside the scripts and change the code to change the architecture of the DQN. This would require knowledge from the user when it comes to machine learning, but would mean that they could make more changes if they had the experience. We ultimately decided against this because of several reasons.

Firstly, the users shouldn't have to go into the scripts to make changes to the architecture, because there isn't a hot-reload function for changes made in the deeper classes in the scripts. This means that the users would have to go through the tedious process of closing and reopening UE4 every time they made a change, no matter how minuscule the change. Secondly, by requiring less practical machine learning knowledge from the users, we can satisfy a larger user group. This way most of UE4 developers with little knowledge about machine learning can use and get an understanding of how the plugin works, while the more knowledgeable users still have access to the open-source scripts if they want to make more changes than the user interface allows.

It is worth noting that the user gets limited to some choices we have restricted ourselves to by trying to simplify the process of changing the architecture of the DQN. Examples of this could be the activation functions we have chosen, as well as how we have implemented the different layers in the neural network. This is because there is a limit to how many options we can give the user before

the amount of choices unnecessarily overcomplicate the code. In other words, if we gave too many options inside of UE4 that change the structure of the DQN, it would just have been easier if the user went in to the code and made those changes themselves. The more choices we give, the more code has to be added just to account for all the possible choices. Therefore we had to find a balance, and only give options we deemed essential.

It was important that we should be able to increase the simulation speed inside of UE4, so that we could be able to train our agents as quickly as possible. As we mentioned, we had to decrease the simulation speed if we increased the number of inputs, batch size and scale of the network architecture. This was something we foresaw at the beginning, and was the reason we tried to combine LibTorch with UE4, in an effort to maximize the performance. Since we didn't manage to get LibTorch to work, we used the TensorFlow plugin for UE4. Unfortunately, this plugin had a bug that resulted in memory leaks when using the multi-threading function at increased simulation speeds. As the cause of the bug hadn't been located by the developer, we couldn't use the multi-threading function. We might have managed to make LibTorch work with more time on our hands, but had to prioritize progressing.

The TensorFlow plugin we were using already used controllers as a way to connect to the backend, so making a superclass controller that contained all the setup for the connection seemed like a user-friendly choice. We chose to use perception updates and stimuli sources, because we wanted the bot to know whether or not there were any visible targets when the bot wanted to shoot someone. We could have made the actions more simple, but we wanted to use the functionality that the AI-tools in UE4 offered and make some higher-level actions. The movement actions that we made are a good example of this. Some of the movement actions are as simple as move forward and backwards. On the other hand, we have the action that moves the bot towards the point. Since UE4 provides a navmesh so that bots can use efficient pathfinding to move to a position, we could use it to make the bot move a certain distance towards the goal. This way, the bot didn't have to use a long time to figure out how to use all the simple movement options to move towards the goal. By finding a balance between what the bot should have to learn and what the AI-tools should provide, we made it easier for the bot to learn the objectives of the game modes.

6.3 Engineering Results

We had several goals for the end product that were elaborated on in chapter 5.3.1. In this section we will go through the goals and discuss the thought process behind them and any weaknesses they might lead to for the final product.

- **The plugin should have support for training agents inside of UE4 with reinforcement learning.**

At the start of the project we were trying to decide our approach to the

problem of adding reinforcement learning to bots in UE4. This meant exploring existing solutions and deciding which machine learning frameworks/libraries could be used to approach a solution to our problem. We wanted to try different reinforcement learning algorithms, but realised that the development of several would increase the scope of the project too much. A weakness the plugin has is that it doesn't have other algorithms that can be used, meaning that users are limited to using DQN.

- **All the functionality should be available through downloading a UE4 plugin.**

One of the reasons we were working on a combination between reinforcement learning and UE4, was because there were few that had developed something with this combination and made it available for others. Other similar combinations of UE4 and machine learning that we found would either be hard to reuse in new cases or require third-party programs running. A plugin is easy to download and can be easily added to existing projects. A potential drawback with our plugin is that it relies on several other plugins to function, but not utilizing existing work that had been done in the field would be detrimental to our results.

- **The plugin shouldn't require a background in machine learning from the user.**

As mentioned earlier, there is a lack of solutions when it comes to the combination of UE4 and machine learning. Since UE4 hasn't introduced any functionality for this, the development of something like this by a third-party leads to complexity for the user. Since the plugin allows the user to change the architecture of the DQN from within UE4, the user doesn't need to think about the practical implementation as long they at least have some knowledge about reinforcement learning.

- **The user should have access to the complete structure of the plugin.**

Everything that the plugin consists of will be available to the user. This allows users to better understand how everything is working and will allow them to make changes and improvements. Since there aren't many similar plugins, it would be detrimental to the development on this field to not open-source everything that we worked on.

6.4 Administrative Results

In this section we will discuss some of the strengths and weaknesses of our work methodology. We will also share our thoughts on our cooperation throughout the project.

6.4.1 Work process

As mentioned in the results chapter, we chose to use principles from agile development in our development process. The weekly update meetings were useful

because it gave us constant guiding input from our external supervisor, and were particularly important early on in the project when deciding how we wanted to approach our problem. We prioritized meeting up in person when working, especially in the earlier phases, as the stand-up meetings were important to keep each other updated on the progress from day to day. The value of being able to meet up and work together became more apparent when the virus outbreak required us to practice social distancing. While we still tried to communicate daily online and the development process continued at a steady pace, it was evident at times that it was more convenient when we could meet up in person more often. The weekly updates we were writing in the project handbook became more useful as each of us updated it with the goals we had for that week as well as what was achieved, giving each other more insight into the overall progress.

The Gantt diagram has been useful for splitting tasks and planning out our work towards different milestones. The Gantt diagram has at times been impractical to use, since our project is mostly research based. While it has been more practical when planning out the development of the plugin, it has been less applicable when working on research. It was difficult to approximate the time needed for research, since we didn't know beforehand how quickly we would get results. In hindsight it might have been more practical for us to use a research-based work methodology, or a combination of research and development based methodologies. This was however difficult to foresee early, as the task was first introduced and interpreted as a development project.

6.4.2 System perspective and ethics

Over the course of this project we have developed a plugin that allow the users to train agents inside of UE4 with a DQN. The plugin comes with examples, so the user can quickly see it in action and also help the user when they make their own environments. It is free and the code is open-source, which means that the user can see and modify how the whole plugin functions. Users that want to make something similar also can make use of the plugin, by using it as a reference or a base to improve on. The results we got from testing the plugin in different game modes can be useful as research on the performance of DQN. The project has been an effort to develop something that can be of use to the game development community and machine learning enthusiasts alike.

The progress of machine learning over the years has led to discussions around the potential ethical dilemmas of using it, and we as developers have a responsibility to follow the ethical guidelines of development. Something worth considering in our project is the power consumption that the training of machine learning models require. Our computers would often run overnight for the training sessions to get useful results, which meant an increase in power consumption. The best results we see in machine learning often require powerful hardware, which can become an ethical dilemma as the consumption is detrimental in the pursuit of the sustainable development goals (Nations 2020) that we should strive for.

While the resources that were available to us in this project aren't comparable to large companies and organizations, we still have a responsibility to make our system as efficient as possible, to minimize the training time and power consumption.

6.4.3 Evaluating the cooperation

It has been easy to cooperate throughout the project since we have done it so many times before, and are aware of how each one of us likes to work. The varying experience we have with the tools we used has been useful when deciding how to split the tasks. We have focused on constantly communicating openly with each other, as well as our supervisors, about the development of the system and research. We haven't had any problems with disagreements, as we often asked each other for advice and thoughts when we had to make defining choices in our tasks. We tried to keep each other updated on each other's progress, and offer a helping hand when one of us was struggling with something. It has been a mutually beneficial learning process that both of us have enjoyed partaking in.

7 Conclusion

This thesis explores the possibility of using RL algorithms to create intelligent agents in UE4, the main challenge with this being the lack of development in the field. We propose a solution where we use a DQN for decision-making combined with the AI tools in UE4 to create high-level actions. In addition we use UE4 to both simulate the environment for the agent to train in, and to simulate the agent itself. One of the challenges with this was making UE4 work with a machine learning library, which we solved by making a plugin that builds on other plugins that formed a foundation for this combination. This plugin allows UE4 developers with little to no machine learning experience to implement RL-agents in their projects.

The system we developed also had to be tested extensively to ensure that it actually served its purpose and yielded useful results that could help us answer our research questions. We explored how the system performed by testing with different game modes that would challenge different aspects of the system and the architecture of the DQN. For this purpose we created different game modes with their own set of rules and map layout. This would allow us to observe and compare how our implementation would perform based on environmental changes. These tests allowed us to determine whether or not our implementation was applicable to use for actual video games, and what type of games. Based on our results we can conclude that a DQN agent in such an environment would perform the best while being able to observe as much as possible in any state. In addition, our research shows the importance of making sure that the design of the reward and map is optimal for generating the desired behaviors from our agent. Based on our results and videos we can observe that there is an evident similarity in the learning process that our agent and a human player go through, meaning that game modes that work for humans with a measurable set of rules should work for our agent as well.

We researched the effect of using different observational data by comparing results from feeding images and handpicked data to the neural network. From the results we can conclude that both forms of input have their advantages, and one should be picked over the other based on the needs and limitations of the environment. The handpicked data is faster to train and has led to the most intelligent behaviour, but is more susceptible to requiring modifications to the chosen data based on environmental changes. Images have the advantage that the data doesn't necessitate changes based on environmental changes, since the input is what the agent is seeing at all times. It is however much heavier on the performance and requires more time to train, because of the generally larger amount of input values and computation required, especially if convolutions are used.

Through the system we have developed we have shown that it is fully possible to use UE4 to simulate environments to train RL agents in, and use RL agents to play popular game modes. Based on the vast number of training sessions

we ran throughout the project, we can conclude that combining reinforcement learning algorithms with AI-tools in UE4 in the form of a plugin yield promising results both in usability as well as the behaviour the agents exhibit.

7.1 Future work

Because this system makes it possible to train agents in a powerful 3D-engine, it introduces a lot of possibilities that we can not explore in one bachelor thesis. The first thing that comes to mind is to implement other RL algorithms than DQN, as well as implementing more optional DQN extensions. As mentioned in the method chapter, the paper "Rainbow: Combining Improvements in Deep Reinforcement Learning" (Hessel et al. 2017) shows the benefits of implementing a wide range of DQN improvements to a single algorithm, and it would be a great addition to have these optional extensions available. This would allow the user to pick whatever algorithm they felt like would be best for their environment. It would also have been nice to be able to test the agents in even more game modes, with even a wider range of input and different action spaces.

Our internal supervisor came with a great idea on how one could go about making an agent behave like a human player. The idea was to record a human playing in an environment, then use those experiences as labeled data to train the NN with machine learning. We could then use these weights as a base when we start the reinforcement learning in UE4. Though being a great idea, we did not have time to work on this as it was introduced a bit late in the project and being a very big task that does not seem to be tested that much based on our research. Nonetheless, it could be a great task for another bachelor or even master thesis to look into the possibility of implementing this on top of our system. Our suggestion might be to look into research like "Combining Supervised, Unsupervised, and Reinforcement Learning in a Network of Spiking Neurons" (Handrich et al. 2011), and use this to extend our system with an optional supervised learned alternative to create weights to initialize the RL training session with.

Another topic that was out of the scope of our thesis was distributed training. This would have allowed us to use multiple training sessions to train one agent at the same time. The main problem with such a solution is that UE4 is mainly for Windows computers while distributed training tools are for Linux machines. Though it might not have been possible to share the weights for one agent across multiple training session because of the game engine we are using, we could have utilized networking to make one training session where we train multiple agents easier. This would work by having one machine or instance of UE4 running for each bot, then use networking to connect them together in one environment. This way we could train multiple agents at the same time at higher speed than we could if we were to run them all on one machine. One could argue that anyone using the system could do this, as we have provided all the tools necessary to do so and UE4 has built-in networking tools, however it would have been nice if we had time to make an example that uses this strategy to train multiple agents.

A final thought that could be interesting to explore is to find a way for all the agents to effectively share their experiences in a shared network. Meaning that the behaviour of an agent can be updated based on the experiences of

one or several other agents. In theory this would mean that an environment could be learned more quickly by releasing several agents into it, and it would be interesting to see how such a solution would affect their actions towards each other. "Experience Sharing Between Cooperative Reinforcement Learning Agents" (Souza, Oliveira Ramos, and Ralha 2019) and "Parameter Sharing Reinforcement Learning Architecture for Multi Agent Driving Behaviors" (Kaushik, S, and Krishna 2018) are two papers that give a good introduction to the topic of exploring the effect of sharing parameters between agents. Multi-agent reinforcement learning is a relatively fresh field, so there are many exciting options to explore. While these additions were out of the scope of our thesis, there is a strong foundation for adding further improvements.

References

- Abadi, Martín et al. (2016). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. arXiv: 1603.04467 [cs.LG].
- AI Perception*. URL: <https://docs.unrealengine.com/en-US/Engine/ArtificialIntelligence/AI Perception/index.html>.
- Baker, Bowen et al. (2019). *Emergent Tool Use From Multi-Agent Autocurricula*. arXiv: 1909.07528 [cs.LG].
- Crockford, Douglas (2006). *The application/json Media Type for JavaScript Object Notation (JSON)*. URL: <https://tools.ietf.org/html/rfc4627> (visited on 05/19/2020).
- Fortunato, Meire et al. (2019). *Noisy Networks for Exploration*. arXiv: 1706.10295 [cs.LG].
- Gym*. URL: <https://gym.openai.com/>.
- Handrich, Sebastian et al. (2011). “Combining Supervised, Unsupervised, and Reinforcement Learning in a Network of Spiking Neurons”. In: *Advances in Cognitive Neurodynamics (II)*. Ed. by Rubin Wang and Fanji Gu. Dordrecht: Springer Netherlands, pp. 163–176. ISBN: 978-90-481-9695-1.
- Hasselt, Hado van, Arthur Guez, and David Silver (2015). *Deep Reinforcement Learning with Double Q-learning*. arXiv: 1509.06461 [cs.LG].
- Hessel, Matteo et al. (2017). *Rainbow: Combining Improvements in Deep Reinforcement Learning*. arXiv: 1710.02298 [cs.LG].
- Juliani, Arthur et al. (2018). *Unity: A General Platform for Intelligent Agents*. arXiv: 1809.02627v1 [cs.LG].
- Kaniewski, Jan (2019a). *tensorflow-ue4: TensorFlow plugin for UE4*. <https://github.com/getnamo/tensorflow-ue4>.
- (2019b). *UnrealEnginePython: A forked plugin for embedding Python in UE4*. <https://github.com/getnamo/UnrealEnginePython>.
- Karpathy, Andrej and Fei-Fei Li (2015). *Convolutional Neural Networks for Visual Recognition*. URL: <https://cs231n.github.io/convolutional-networks/> (visited on 05/16/2020).

- Kaushik, Meha, Phaniteja S, and K. Madhava Krishna (2018). *Parameter Sharing Reinforcement Learning Architecture for Multi Agent Driving Behaviors*. arXiv: 1811.07214 [cs.LG].
- Lillicrap, Timothy P. et al. (2019). *Continuous control with deep reinforcement learning*. arXiv: 1509.02971 [cs.LG].
- Mnih, Volodymyr, Adrià Puigdomènech Badia, et al. (2016). *Asynchronous Methods for Deep Reinforcement Learning*. arXiv: 1602.01783 [cs.LG].
- Mnih, Volodymyr, Koray Kavukcuoglu, et al. (2013). *Playing Atari with Deep Reinforcement Learning*. arXiv: 1312.5602 [cs.LG].
- Nations, United (2020). *Sustainable Development Goals*. URL: <https://www.un.org/sustainabledevelopment/sustainable-development-goals/> (visited on 05/19/2020).
- Paszke, Adam et al. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. arXiv: 1912.01703 [cs.LG].
- Schaul, Tom et al. (2016). *Prioritized Experience Replay*. arXiv: 1511.05952 [cs.LG].
- Schulman, John, Sergey Levine, et al. (2017). *Trust Region Policy Optimization*. arXiv: 1502.05477 [cs.LG].
- Schulman, John, Filip Wolski, et al. (2017). *Proximal Policy Optimization Algorithms*. arXiv: 1707.06347 [cs.LG].
- Souza, Lucas Oliveira, Gabriel de Oliveira Ramos, and Celia Ghedini Ralha (2019). *Experience Sharing Between Cooperative Reinforcement Learning Agents*. arXiv: 1911.02191 [cs.LG].
- Thrun, Sebastian B. (1992). *Efficient Exploration In Reinforcement Learning*. Tech. rep.
- Unigine. URL: <https://unigine.com/>.
- Unity. URL: <https://unity.com/>.
- Unreal Engine. URL: <https://www.unrealengine.com/en-US/>.
- Wang, Ziyu et al. (2016). *Dueling Network Architectures for Deep Reinforcement Learning*. arXiv: 1511.06581 [cs.LG].

Zai, Alexander and Brandon Brown (2018). *Artificial Intelligence: A Modern Approach*. 2nd ed. Manning Publications. ISBN: 9781617295430.

Appendix

Appendix A: Videos

Video of agent from figure 5.2: <https://youtu.be/t9GQEJkPjQo>

Video of agent from figure 5.3: <https://youtu.be/-LPg1Yo98zk>

Video of agent from figure 5.4: <https://youtu.be/skoQQijbipg>

Video of agent from figure 5.6: <https://youtu.be/MUBJcMz4Ibw>

Video of agent from figure 5.7: <https://youtu.be/9zMckgrrFpE>

Video of agent from figure 5.8: https://youtu.be/1BZ08_SqRIY

Video of agent from figure 5.9: <https://youtu.be/BG0QFHDh9no>

Video of agent from figure 5.15: <https://youtu.be/rQqG8x8hTxw>

Video of agent showing the described behavior in chapter 6.1.1: <https://youtu.be/-179XsIUD3Y?t=17>

Video of agent from figure 5.3 starting at 23 seconds: <https://youtu.be/-LPg1Yo98zk?t=23>

Appendix B: Implementation

GitHub repository containing the plugin we created: <https://github.com/magomedb/IAP>

GitHub repository containing our project and code before it was made into a plugin: <https://github.com/magomedb/IAP-Project>

Appendix C: Hardware

Lenovo Legion Y740

This laptop was used for development and running training sessions mainly with images as input. The laptop has an Intel Core i7-9750H, 16 GB RAM and a GeForce RTX 2070 integrated with 8 GB memory and Max-Q Design.

ASUS TUF FX705DT

This laptop was used for development and running training sessions with hand-picked values as input states. The laptop has an AMD Ryzen 7 3750H, 16 GB RAM and a GeForce GTX 1650.

Custom desktop machine 1

This is a desktop machine mostly used for development while our other machines were busy running training sessions. This machine has a AMD FX-9370, 16 GB RAM and a GeForce GTX 1070 Ti.

Custom desktop machine 2

This machine was mostly used for development, as the specifications are too outdated to reliably run heavy training sessions. This machine has an Intel Core i5-4690, 8 GB RAM and a Geforce GTX 760.

HP 250 G6

We used this laptop to make sure our system worked on less capable computers, that did not have any previous development tools installed. The laptop has an Intel Core i5-7200U, 8GB RAM and there is no dedicated graphics card.