Mittuniversitetet
MID SWEDEN UNIVERSITY

Machine Learning Adversaries in Video Games

Using reinforcement learning in the Unity Engine to create compelling enemy characters.

Tim Nämerforslund

Computer Engineering BA (C), Bachelors Project
Main field of study: Computer Engineering
Credits: 15 hp
Semester, year: VT, 2021
Supervisor: Jan-Erik Jonsson
Examiner: Patrik Österberg
Degree programme: Degree of Master of Science in Engineering: Computer Engineering, 300 credits

Abstract

As video games become more complex and more immersive, not just graphically or as an art form, but also technically, it can be expected that games behave on a deeper level to challenge and immerse the player further. Today's gamers have gotten used to pattern based enemies, moving between preprogrammed states with predictable patterns, which lends itself to a certain kind of gameplay where the goal is to figure out how to beat said pattern. But what if there could be more in terms of challenging the player on an interactive level? What if the enemies could learn and adapt, trying to outsmart the player just as much as the player tries to outsmart the enemies. This is where the field of machine learning enters the stage and opens up for an entirely new type of non-player character in video games. An enemy who uses a trained machine learning model to play against the player, who can adapt and become better as more people play the game. This study aims to look at early steps to implement machine learning in video games, in this case in the Unity engine, and look at the players perception of said enemies compared to normal state-driven enemies. Via testing voluntary players by letting them play against two kinds of enemies, data is gathered to compare the average performance of the players, after which players answer a questionnaire. These answers are analysed to give an indication of preference in type of enemy. Overall the small scale of the game and simplicity of the enemies gives clear answers but also limits the potential complexity of the enemies and thus the players enjoyment. Though this also enables us to discern a perceived difference in the players experience, where a preference for machine learning controlled enemies is noticeable, as they behave less predictable with more varied behaviour.

Keywords: Unity, Machine Learning, ML-Agents, Navigation Mesh, Challenge, Video Games

Sammanfattning

I och med att videospel blir mer avancerade, inte bara grafiskt utan också som konstform samt att dom erbjuder en mer inlevelsefull upplevelse, så kan det förväntas att spelen också ska erbjuda en större utmaning för att få spelaren bli ännu mer engagerad i spelet. Dagens spelare är vana vid fiender vars beteende styrs av tydliga mönster och regler, som beroende på situation agerar på ett förprogrammerat sätt och agerar utifrån förutsägbara mönster. Detta leder till en spelupplevelse där målet blir att klura ut det här mönstret och hitta ett sätt att överlista eller besegra det. Men tänk om det fanns en möjlighet att skapa en ny form av fiende svarar och anpassar sig beroende på hur spelaren beter sig? Som anpassar sig och kommer på egna strategier utifrån hur spelaren spelar, som aktivt försöker överlista spelaren? Genom maskininlärning i spel möjliggörs just detta. Med en maskininlärningsmodell som styr fienderna och tränas mot spelarna som möter den så lär sig fienderna att möta spelarna på ett dynamiskt sätt som anpassas allteftersom spelaren spelar spelet. Den här studien ämnar att undersöka stegen som krävs för att implementera maskininlärning i Unity motorn samt undersöka ifall det finns någon upplevd skillnad i spelupplevelsen hos spelare som fått möta fiender styrda av en maskininlärningsmodell samt en mer traditionell typ av fiende. Data samlas in från testspelarnas spelsessioner samt deras svar i form av ett frågeformulär, där datan presenteras i grafform för att ge insikt kring ifall fienderna var likvärdigt svåra att spela mot. Svaren från frågeformulären används för att jämföra spelarnas spelupplevelser och utifrån detta se skillnaderna mellan dom. Skalan på spelet och dess enkelhet leder till att svaren inte bör påverkas av okända och ej kontrollerbara faktorer, vilket ger svar som ger oss insikt i skillnaderna mellan dom olika spelupplevelserna där en preferens för fiender styrda av maskininlärningsmodeller kan anas, då dom upplevs mer oförutsägbara och varierande.

Nyckelord: Unity, Maskininlärning, ML-Agents, Navigationsnät, Utmaning, Videospel

Acknowledgements

I would like to thank the people at the Mid Sweden University who have enabled me to partake in and perform interesting projects which have let me grow as a person.

I would also like to thank the people who participated in this study as testers, as well as my wife who supports me through thick and thin.

Table of Contents

Terminology

NPC	Non-player character
ML-Agents	Machine learning agents
CPU	Central processing unit
GPU	Graphics processing unit
Navmesh	Navigation Mesh
GUI	Graphical User Interface
FPS	First Person Shooter

1 Introduction

In today's video games the general design pattern for non-player characters (NPCs) is that of the state-machine. This is a programming pattern upon which the actions of the NPC are determined via which state the NPC is in. For example, a guard might have several states that tell him what to do, like patrolling, chasing, attacking and returning to his post. Which state he is in depends on what is happening in the game. If the player comes within a set distance his state shifts from patrolling to chasing, and when within attack range he will go into attacking state, which automatically switches back to chasing after the attack is finished. This programming pattern is well known and works well for NPCs who are expected to act in certain ways according to the conventions taken for granted in modern games. This can be both beneficial and detrimental for a game's user experience. It both presents the player with predictable enemies which can be just what the developer wants the player to overcome, a repeating set of behaviours which can be outplayed and overcome. But it can also lead to players feeling that the NPCs they encounter are lackluster and boring, as they know exactly how they will act every time, and given if the NPC has a too simple state machine, it can appear unnatural and immersion breaking. With the help of machine learning more intelligent models that act as more compelling adversaries in video games could be created.[1][2][3]

1.1 Background and problem motivation

The idea of smarter adversaries in video games should entice anyone who has any familiarity with video games. Enemies that react smarter to the player's actions but also think outside the box could provide a more in-depth experience. This in combination with today's accessible game engines, which allows for plug-ins and extensions, the integration of machine learning into video game environments is certainly within the realm of possibility. One such game engine which allows for easy integration of machine learning is Unity, a cross-platform game engine developed by Unity Technologies. They have developed their own library, the Unity Machine Learning Agents Toolkit, to integrate current and upcoming machine learning technology into their own game engine. The library is a open-source project which aims to aim to "enables games and simulations to serve as environments for training intelligent agents"[4].

Their implementation is based on PyTorch, which is an open-source machine learning library used for applications which work with computer vision and natural language processing, enabling hobbyists or researchers to train Unity's ML-Agents (Machine learning agents) to work within both 2D and 3D environments. With the library's ability to train ML-Agents

for multi purpose tasks, single, multi and adversarial NPC behavior, automated testing of game builds and assistance in evaluating game design decisions, the Unity ML-Agents Toolkit presents ample opportunity to research whether or not smart behavior in video games is a compelling for the futures video game players.

1.2 Overall aim

The study's overall aim, is to via the usage of currently available machine learning technology for video game engines, to investigate whether machine learning models can act as compelling enemies in video games. The ML-model will be compared to the more traditional state driven NPCs, controlled by trailed and tested movement controllers, in this case navigation mesh based movement. In the end the two types of enemies will be put in play against players who will give their opinion on their game play interactions with the enemies, from which we can compare the two enemies' gameplay value.

The study's overall aim is to, with the help of currently available machine learning technology on the market for video game engines, investigate whether or not machine learning models can act as compelling adversaries in video games compared to more traditional state driven NPCs.

1.3 Scope

The scope of this study will be limited to measuring and analyzing the player's perceived difference between a trained machine learning NPC and a simple navigation mesh controlled one. Via the use of a questionnaire consisting of numeric questions the perceived player experience can be compared between the two models. The study will mainly focus on the factors of challenge, enjoyment and interest perceived by the players when playing against the two types of NPCs. A small game with simple gameplay elements and environment will limit the influence other elements have on the player's experience. The study will strive to create similar behavior in the two types of NPCs, though emphasis will be on that they provide an equal challenge, which will be measured and confirmed via statistics provided from the players' gaming sessions.

1.4 Concrete and verifiable goals

Via presenting voluntary players with two similar video games, where one game has traditional NPCs while the other has machine learning trained models controlling them. The players will try to complete a given task in

the video game and afterwards fill out a form about their experience, which will give insight into how the player's experience is affected by the different NPC behaviors.

Via the help of the questionnaire the players answer after their gaming session, the study will try to determine which of the two NPC behaviors are perceived as the most challenging, enjoyable and interesting as adversaries. The form will use grading scales along with questions that players will answer, and the answers will be compared to verify if Machine learning trained models can provide a more compelling game experience when used as adversaries to the player. Compelling in this study will refer to a combination of a fair challenge which the player found enjoyable and interesting to play against, rated by their own scores in the respective categories.

To assure that the challenge is fair, player statistics gathered during the testing sessions will assure that the two types of NPCs provide an equally challenging experience. A similar trend of player score to time lived in each session will indicate that the two NPCs are equally challenging.

To summarise the research questions and goals of the study:

- With respect to players' perceived challenge, enjoyment and interest, is there a difference in the player experience for players when facing a machine learning controlled enemy compared to a state driven enemy?
- Is the two types of enemies equally challenging, both in regards to how players perceive them but also statistically?

1.5 Outline

Chapter two goes over the theory behind the report, beginning with the background of machine learning in video games and after that explaining the commonly used method of reinforcement learning used for machine learning. Chapter two also covers what the Unity ML-Tool kit is, other large research projects within the same field and shortly what a navigation mesh is in video game environments.

After that chapter three follows, covering the method followed in this study. Areas such as how the test game will be constructed, how the model will be trained and how the game will be tested and the results will be compared is explained. As well as an overview of the Unity engine and how it is intended to be used in this study is explained. Covering things such as which components are to be used, how scripts will interact with each other and what hardware will be used along with what inference mode. It also covers how the training of the machine learning model will be monitored.

Chapter four explains the construction of the test game and the vital parts of it. Areas such as game scenes, machine learning agent and training design, the navigation mesh agent and the player controller are explained here.

After this chapter five goes over the results of the tests, showing compiled bar charts and graphs of the questionnaire answers as well as player statistics. It also covers the final design and completed training of the machine learning model. To complement chapter five, chapter six contains an analysis, discussing the results and giving insight into exposed weaknesses and other information that is revealed when looking at the results and feedback.

Finally chapter seven goes a summarised conclusion on the findings and outlines a few ideas for future work.

2 Theory

2.1 The Unity game engine

The Unity game engine provides an easy to learn environment for constructing playable scenarios on a scene to scene basis. It also comes with a physics system which works for creating good enough physics based player controllers, as well as in-engine handling of gravity, drag and collisions. In-engine colliders along with rigidbodies provide a foundation of easy to implement movement and collision for both the player, the NPCs and the environment. A general movement script is can be written to handle the NPCs movement, with variables such as maximum and minimum speed, turn rate, slow down speed and acceleration, as to provide a *easy to change* system for movement on the NPCs which can be used even with changes to the ML-Agents physical representation in the game world and its observation vectors. This makes the Unity engine very suitable for real time simulation which other researchers are starting to take advantage of, example being a traffic obstacle avoidance based on machine learning[5]. Others have used it to try and cut down on development time in video games and help more game developers finish their games faster with better results[6].

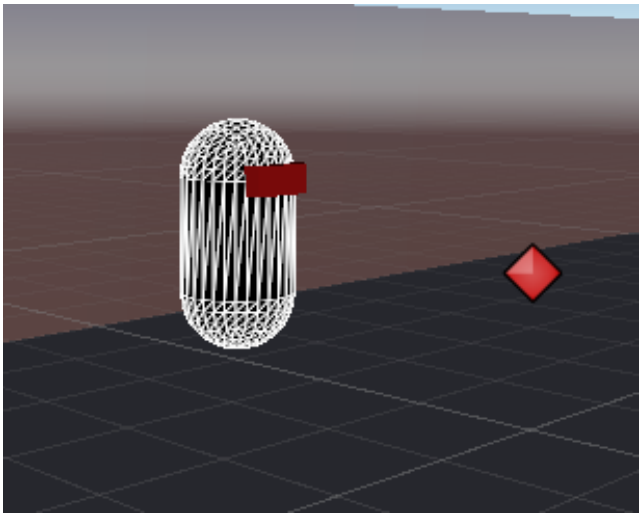

Figure 1: The NPC represented by a cylinder mesh, along with a spawn point for the training target.

Seen above in figure 1, is the intended visual representation of the NPC in form of a cylinder mesh with a box attached to indicate its forward direction. Via taking inputs via functions and applying these during Unity's *FixedUpdate()* function which has a frequency equal to that of the physics system. The inputs correspond to the desired amount of forward and rotational movement of the NPC. And as each tick of *FixedUpdate()* is called the

input is translated into movement either forward, backwards or standing still along with rotation along the Y axis either clockwise or counter clockwise or no rotation at all. The desired velocity is applied to NPCs' rigidbody, and collisions handled by its colliders, all native to the Unity engine as to keep the design simple and reliable. This gives the NPC a simple yet sufficient way of navigating the environment.

2.1.1 Unity ML-Agents Toolkit

The Unity ML-Agents toolkit comes with a large variety of tools that can be employed to feed the training algorithm with the necessary data to create smart behavioural models for the agents. The agents in the ML-Agents toolkit operate via a set of functions focused around observations, actions and rewards, each doing their part in forming the final model which will be used by the NPC during run-time to control the NPC. Each ML-Agent will, according to chosen parameters, collect observations from their own *Behaviour Parameter* component, along with observations from their *Ray Perception Sensor* components, after which actions will be taken. Actions can be both discrete and continuous values and they are made up of vectors, called branches, of set sizes.

It is also possible to choose which device, either the CPU (Central processing unit) or the GPU (Graphics processing unit), that will be used as the inference device. In this study the computer's GPU will be used as the inference device, given the choice of having ray perception sensors for each agent, which is more GPU intensive, and also that the computer's GPU is stronger than the CPU. The GPU that will be used in this study is of the model MSI GeForce GTX 1080 Gaming X.

2.2 Machine learning in video games

The idea of machine learning, specifically reinforcement learning and algorithms, in video games is nothing new. In recent years many simulation platforms such as Arcade learning[7], VizDoom[8], MuJoCo [9] which enables easy-to-use benchmarking of machine learning algorithms via the help of existing video games. These along with other environments and platforms have helped speed up the rapid development of more efficient and powerful algorithms used in machine learning. Though as stated by Juliani et al. in [10], for researchers to continue to produce high-quality algorithms, there needs to be easy-to-use, flexible and universal platforms that enable the production of new algorithms. Which is why they have developed the Unity ML-Agents Toolkit for the Unity Engine.

Generally the algorithms developed to be used in video games are reinforcement algorithms which construct models via repeated trial and error, based

on rewards and penalties given over an episode, in order to optimize the models decision making in the set situation.

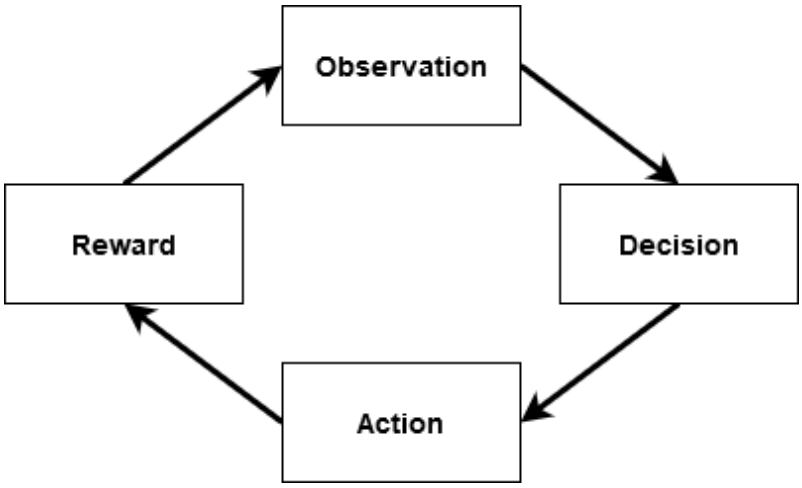

Figure 2: The reinforcement learning loop

This process, as seen in figure 2, consists of a looping process of observing the agent's environment, taking a decision, performing said decision as an action and then getting rewarded or punished depending on the results. An iterative process of these steps builds up a model of how an ML-Agent should act depending on the current state of the environment the agent is in along with observations it has been enabled to take in. So in time after repeated training, a model is generated that will control the behaviour of the NPC which will act in a manner that maximises the agent's reward. It's also possible to have the ML-Agents in the Unity ML-Agents Toolkit act as teams, working together or against each other to maximise their teams total reward, which in turn can be used to train more complex behaviours for the NPCs.

The key to being able to train these kinds of models in Unity's 3D environment is thanks to their work which is utilizing the Tensorflow and PyTorch libraries. TensorFlow, which is made by the Google Brain Team[11], is an open-source library developed to enable training and inference for deep neural networks, though is suitable for a large number of machine learning related tasks. PyTorch, developed by Facebook's AI Research lab[12], is similar to TensorFlow, but instead of focusing on training and inference it focuses on computer vision and natural language processing.

There has also been research regarding applying machine learning on video games which is not part of the Unity ML-Agents kit, Google Brains project or Facebook's AI research. These studies have look at either teaching AI to play already existing games or building a game where the players interact with and train a machine learning model in real time.[13] [14][15]

2.3 Game scenes

The game scene is constructed to fit the scope of the study. In Unity, scenes are constructed to fit the need of both training and testing. With the intended player experience of players facing off against NPCs who seek to defeat them, the player faces these NPCs in a first-person perspective in a four wall room. The room will have random obstacles generated each time the player spawns in to randomize the experience and challenge both player and NPC.

2.3.1 Training scene

The training scene must mimic the intended gameplay scene to such an extent that the ML-Agents can learn to operate both in their training scene as well as the gameplay scene. This is done via gradually adding more to the training scene as to reinforce behaviour patterns in the agents that allows it to step by step learn the complexity of the gameplay scene. The initial training scene will consist of a four wall room with the target in the center. The agents spawn in relatively similar positions and can therefore learn over time that rewards are administered when the agent touches the target. To help the agent learn this quicker the target is spawned with collision spheres set to trigger mode, meaning that they don't collide with the agent. Instead they activate a bit of code giving the agent a small reward of 0.2 each time the agent enters this sphere, which encourages the agent to get closer to the target's position.

As the agent successfully learns to move towards the target the agent's starting position will be assigned according to an increasing range, giving the agent more different starting positions to learn to navigate from. This is accompanied with the target's position starting to change with each episode, where an episode is defined as the time between the agent's start signal and up until it succeeds or fails at its task. The target will start to appear each episode on a small set of random positions, the amount of positions and the range of X and Z values increasing as the agent learns to align and move towards the target. Upon reaching satisfying performance of localizing the target and moving towards it, predefined wall placements will be introduced to the training sessions. With each episode all walls will be put in a list managed by the *Game Manager* object and will have a 70% chance to be set active, giving the agent a random environment each episode to navigate. It is expected that the mean reward of the agents' training sessions will fall greatly here as the agent is learning to identify new hinders inside the training room and moving around them, which is a different set of decisions compared to aligning with the target and moving towards it. More walls will be added over time as the agent learns to navigate around these walls until the desired random complexity of the room is achieved.

2.4 Navigation mesh

As a counterpart to the trained model produced by the ML-Agent toolkit, a traditional Navmesh (Navigation Mesh) agent will be sufficient. A navmesh can be summarised as an area defined via multiple two-dimensional convex polygons. The combined area of these polygons define an area upon which the agent can move. The Unity engine offers a native solution to generating navmeshes which can easily be used for the native navmesh agent component which acts as a basic controller which allows the programmer to program movement behaviour which uses the generated navmesh. It's a good entry level tool for getting reliable movement along predefined 3D environments without having to delve into complex coding.

3 Method

This study focuses on the question if machine learning models can act as compelling NPCs in video games, as enemies that are more fun or difficult to face. Due to scope and time limitations the study mainly focuses on the short term user experience in a prototype video game where the player tries to fight off incoming NPCs. The test players will face both a standard navmesh controlled NPC as well as the trained machine learning model, and after having played against both fill out a questionnaire regarding their perceived experience. The test players will answer questions such as "How challenging/enjoyable/interesting were the enemies to face off against?" Questions will be answered on a scale of one to five. Other questions will be questions such as "How familiar are you with video games?", covering the players previous gaming experiences.

3.1 Game Construction

To be able to test players against the more traditional navmesh based state driven NPC and the trained machine learning model, a simple game will be constructed where the player will be tasked with surviving as long as possible. They will first face off against the navmesh based NPC and after either winning or losing against it, the same game will load but with the machine learning model controlled NPC instead. The use of the Unity game engine will both simplify the process of implementing this game but also the integration of both a navmesh controlled NPC and a machine learning model controlled NPC. When the training of the machine learning model reaches sufficient performance it will be implemented in a similar scene as the navmesh NPC, and player controls sufficient to play against the NPCs will be implemented. Due to the current situation with an ongoing pandemic the test players will be found online via various communities and social media platforms, where requests for test players will be submitted.

A first person shooter game is the natural choice given the scope of the study. The ML-Agents Toolkit has a lot of examples of how to train a ML-Agent in a 3D space with conventional movement controls that lends itself well for a game focused around movement. This gives the player a clear objective of staying alive via hitting enemies with spheres while the enemy can focus on the simple task of moving to the player. This gives a dynamic relationship between player and the enemy while keeping the game simple and limits the factors influencing the machine learning training.

3.1.1 Game Scenes

The game scenes will be built in a linear fashion to create a simple clear cut test experience for the players. Starting off with an intro screen with information regarding the study and the game they are about to play, as well as a field to enter their name. After this the navigation mesh scene will load, letting the player play until defeated or the time of 180 seconds has passed. A time limit is good as it accounts for really good players who might play forever due to never being defeated. After this an intermission scene will load to allow players a moment to rest, they continue via pressing a button which loads the machine learning scene. Here the same procedure will take place, play for 180 seconds or until defeated. After that a ending scene loads which thanks the player for participating and gives them a choice between opening the link to the questionnaire or quitting the game. Either button will close the application but one opens the questionnaire website also.

3.1.2 Machine Learning Agent

The machine learning agent will be designed to behave similarly to a racecar. It will be able to accelerate either forward or backwards and turn left or right. This gives the machine learning model very few input factors to consider, meaning that it might take some time to get some good results due to limited movement controllers or that it can't handle a super cluttered environment, but given the scope of the study, we want to keep the models inputs simple as to achieve results faster. Being able to combine motion either forward or backward while turning gives the agent enough control and mobility to act as a challenging opponent with enough training. Its rewards will be that of a base value of 1 with two progression spheres around the target, which gives an additive reward of 0.2 when the agent touches them, meaning the total reward will be 1.4. This follows the guidelines of the documentation for the Unity ML-Agents Toolkit which recommends holding reward values small as it is easier for the model to understand the value of these rewards if they don't bloat overtime with long training sessions, which would cause the rewards to lose meaning for the model.

For data-input the agent will be aware of its own position and the target's position, both consisting of a X,Y and Z value. This along with the dot product between the two known positions gives the model an angle between the two positions. This in theory would give the agent enough information to understand where it is, where it needs to go and how to turn and accelerate to get there. To help the agent further it will have a raycast component consisting of nine raycasts spread as fan in-front of it feeding it information of hit objects, returning their distance and tag, meaning that the agent can identify the target and walls hit by the raycasts and the distance to them. This helps it further to make precautionary adjustments to its movement.

3.1.3 Navigation Mesh Agent

The counterpart of the machine learning enemy will be a navigation mesh controlled enemy. This is due to the navigation mesh being a native part of the Unity engine, cutting down on work time needed to develop a reliable movement system, which can both move around the play area while avoiding obstacles, for the counterpart enemy. This system is also suitable as the default behaviour for the navigation agent component in Unity because it resembles that of the intended movement of the machine learning agent, meaning they will move similarly and be more comparable.

3.2 Data collection and questionnaire

The data from these tests, which will include players time lived and score, along with their answers from the questionnaire will be used to evaluate the perceived challenge, enjoyment as well as how interesting they found the two types of enemies. The questionnaire will cover how challenging, enjoyable and interesting the player found the enemies, and an overview comparing how the two types of enemies scored in these areas will tell us which one the player might prefer. These factors are chosen to give insight into if the enemies are equally challenging. Enjoyment is asked about as it is important to understand if the players want to play against these types of enemies or finds them fun, and this in combination with interest can indicate if it might be something future players will actively be drawn to, if they will be intrigued by games in the future who have machine learning models for their enemies.

Both NPCs scoring similarly on challenge is preferred, as then their scores on enjoyability and interest will be deciding which is more compelling. To assure that the enemies are of equal difficulty, time lived and score will be plotted against each other, with the hopes of showing two similar curves, telling us that players on average achieve similar scores for similarly long play sessions. If one of the enemies causes players to either get much less score or players on average don't stay alive as long, then the enemies are not of equal challenge. The study will focus on these two scores mainly because they can be plotted to give an indication if player performance is equal between the two enemies, while also keeping the data collection simple and easily understood in graph form, while at the same time telling if the challenge is equal.

3.3 Training Monitoring

The Unity ML-Agents toolkit includes as an extension of Tensorflow the performance overview library TensorBoard, which enables us, via a web-based

graphical user interface (GUI), to overview and monitor both currently ongoing and previous training sessions. This tool will be used to monitor the progression of our agents training, to see when the agent reaches a satisfactory level of performance.

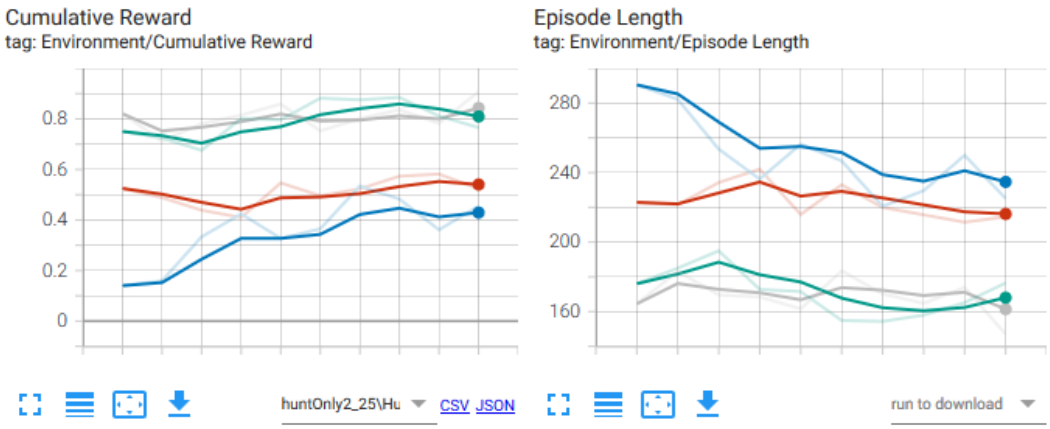

Figure 3: Overview of model training session in TensorBoard.

The main method of monitoring the progression of the training is done via the graphs provided by Tensorboard. As seen above in figure 3 there are two graphs present at all times in Tensorboard, one for monitoring cumulative reward of the models and one for episode length. These graphs will be monitored after each training session, where we desire to see higher reward numbers with shorter episode lengths, indicating that the agents finish their tasks faster and with better results. Due to the nature of machine learning, where training results can go up and down rapidly before a successful pattern is found by the model, we apply smoothing of a factor 0.6 to the graphs to make the graphs more readable.

4 Construction

4.1 ML-Agent design

The design of the ML-Agent is aimed to be general, meaning it should be able to navigate any environment in 3D space consisting of a flat surface cluttered with obstacles and a given target to chase. The Unity ML-Agents rely on observations as to make the most optimal decisions, and given the nature of our scene being a 3D environment which the agent is supposed to navigate, the use of observational vectors of the agents own position, the target's position, the dot product of the agents and targets location along with a ray sensor component, the agent is sufficiently equipped to take in its environment as well as understand where its target location is.

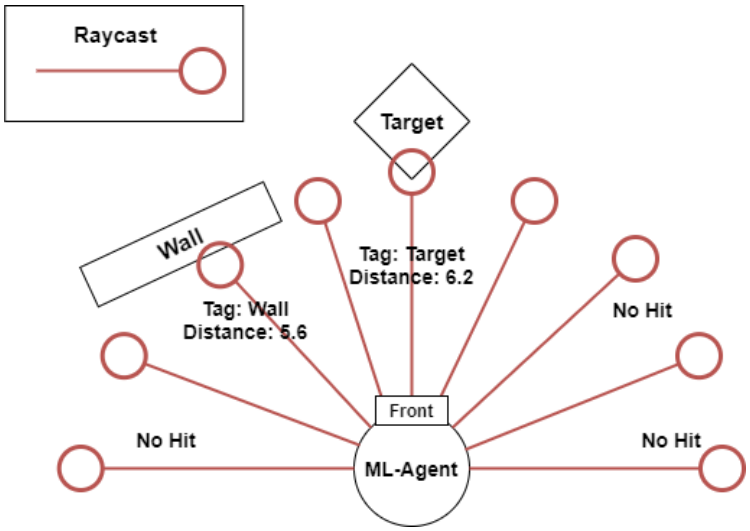

Figure 4: Overview of visual representation of agent's raycast sensor component.

In the *CollectObservations(VectorSensor sensor)* function provided by the Unity ML-Agents toolkit, we dictate what information the agent is to observe before taking any actions. The agent in our study receives its own transform position, the target's transform position and the calculated dot product between itself and the target with respect to their given forward direction. This means our agent receives in total seven values in our *CollectObservations(VectorSensor sensor)* function. In addition to this a child object of our agent is provided with a ray perception sensor component, giving the agent the ability to shoot raycasts into the 3D environment and receive observations of whether or not the raycasts hit anything, what they hit and the distance to said object. The discrete action values are assigned via switch statements and passed as inputs to the function *SetInputs(forwardAmount, turnAmount).* To encourage the agent to complete its task as quickly as pos-

sible the agent receives an existential penalty each step the learning process takes. Meaning each step that is taken in the learning process in the agent's current episode the agent receives a penalty calculated by dividing the total amount of steps the agent can take per episode with the desired max reward, which in our case is a value of one and not 1.4 due to it being impossible for the agent to actually achieve.

As seen in figure 4, the agent has nine raycasts spread over a set amount of degrees. Each raycast has a sphere at the end which is used to detect collisions with desired objects. The size of the spheres are set to help the agent detect objects even as small as the target, which is in the training case a small sphere. These raycasts can be given specific rules that dictate what they collide with as to control what the agent is actually observing. In our case the agents' raycasts only hit designated walls and the target, both which have different tags and layers to make sure the agent understands if it is looking at a wall or the target, and how far away they are.

The Unity ML-Agents toolkit supports agents operating with either discrete or continuous values. Our agent is designed to operate with discrete values due to their simple nature. The agents *OnActionReceived(ActionBuffers actions)* function handles the agents assigning of values to its action vectors, which in our case is used as input for the attached movement script. The agent operates via assigning a value of zero, one or minus one to a set of two variables, which will be used as input to determine if and how the agent wants to move and rotate.

The agent also has functions which can be called both by the agent and from outside the script to reward the agent for successfully catching the target or the player as well as entering a progression trigger zone. Both the training target and the player have two spherical triggers around them which when the agent enters is deactivated and the agent receives a small reward of 0.2 as to tell the agent it is getting closer to its target. The decision to include this was to help the agent learn faster that moving to the target is the desired result, as a training agent might still miss the target though still graze close by, and thus giving the agent a small reward for being close helps incentive the agent to move towards the target's observed transform position.

4.1.1 Training

The ML-Agents Toolkit training is run via a command prompt where a command to initialize the training is entered, along with desired parameters, such as run ID and configuration file to use. In the initial stages of this project several configuration files were used and tested.

The Unity ML-Agents toolkit supports was is called as behavioural cloning, which in the ML-Agents toolkit's case is when a user via the heuristic con-

trols plays the part of the agent and performs the agents intended task during a recording session, after which a demo is created that will serve as a cheat-sheet for the agent to look at as to know what is expected of it. The user can then in the configuration file add one or several demo files for the agent to learn from. In the Unity ML-Agent toolkit the agent will try to mimic the behavior in the demo, to a settable degree, while still deviating from it. After each episode the agents performance is judged by the discriminator and if the discriminator can not guess correctly if the agents episode was a clone of the demo or a attempt of mimicking the demo behaviour, then the agent gets it reward and learns, resulting in the agent only learning when it figures out a new unique way to mimic the demo without being to similar.

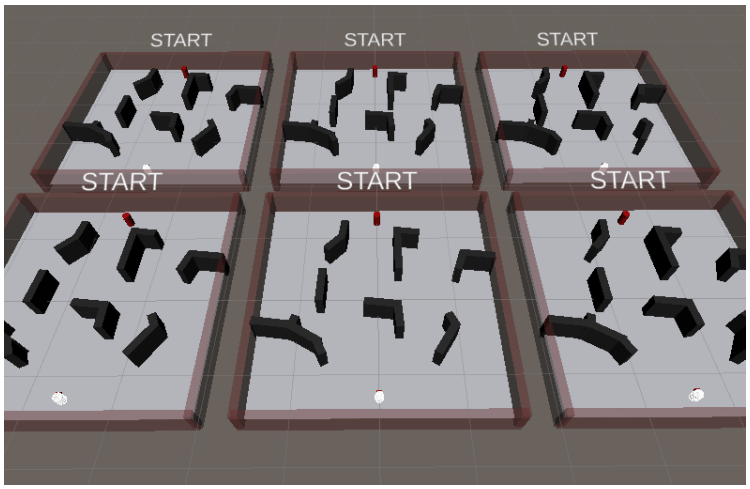

Figure 5: In engine view of scene set up for mass training.

To ensure a relatively rapid progression of the training, the usage of multiple agent instances can be used, letting several agents train at the same time, all contributing to the same model. As 3D navigation based on random environments is a complex task it is a given to use multiple instances of the training environment to archive tangible results in a meaningful time. Though in our construction where the agent utilizes ray perception to a large degree, a lot of processing power is needed so a middle ground between the amount of agents and needed computing power has been struck. A total of six agents train at the same time, giving the model many times over the training it needs to progress while not pushing the computer and the GPU over their limits. See figure 5 for a screenshot of the mass training scene.

4.2 Navmesh counterpart

In the project navmesh agent and its respective scene acts as the counterpart to the ML-Agent driven scene. The navmesh agent works in a similar

fashion as the ML-Agent that it is bound by a 2D plane, can move, break and turn. Though instead of analyzing its environment it operates along a navigation mesh generated from the two dimensional plane acting as the ground in the scene.

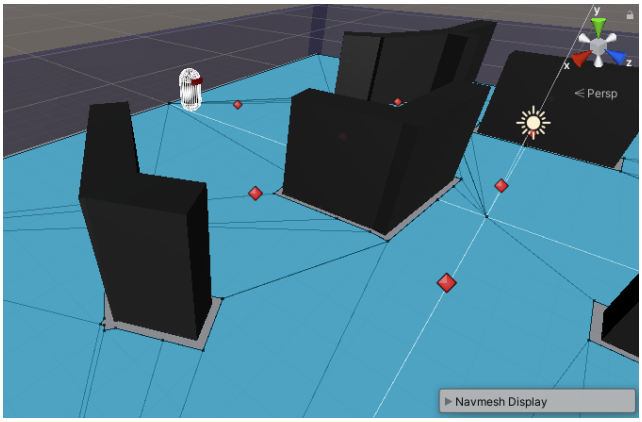

Figure 6: In editor visualisation of generated navigation mesh.

Above in figure 6 the generated navigation mesh can be seen as a blue plane with the polygon edges drawn as dark lines. This is the plane which the navmesh agent can walk along. Thanks to inbuilt tools, Unity offers the possibility to define spaces that are not walkable, such as walls or other obstacles, these can have the *"carve"* attribute, meaning that they dynamically remove form the navmesh areas where the object and the navmesh intersect. This is seen in figure 6 as the walls have a gap between them and the generated navmesh, showing that the walls have dynamically created a non-walkable area around them, hindering agents from walking through them.

The navmesh agent operates on the simple logic of having the players current position assigned as their target location, meaning they use the built-in navigation mesh library in unity to move themselves along the generated navigation mesh toward the player. If they collide with the player they reduce the player's health by one and delete themselves from the scene.

4.3 Player Controller

Contradictory to the agents movement script, the player will be using the standard Unity *CharacterController* component, meaning a reading of the keyboard input along the horizontal and vertical axis will be fed into the standard move function of the *CharacterController* component. This accompanied with a custom *MouseLook* script enables the player to look around via moving their mouse. This script also handles inputs from the mouse which allows the player to fire a projectile with a limited rate of fire, with the in-

tention of fending off the NPCs, as the projectile upon collision destroys the NPCs. The two scripts allow the player to move Forward/Backwards and Left/Right relatively in combination with free-look functionality.

5 Results

The overall result of the study is an insight in how players perceive some simple yet decisive difference in design of enemies in video games. Comparing questionnaire results gives insight in how they judge and perceive the two types of enemies.

This along with ground work having been done on understanding the challenges in training machine learning in an interactive 3D environment has resulted in knowledge and possible future works which will be discussed in following chapters.

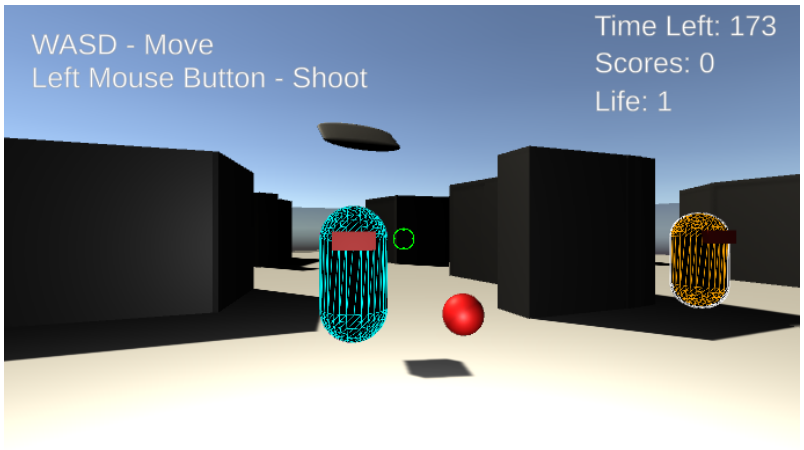

Figure 7: Screenshot of a test session.

Seen above in figure 7 a screenshot of the test game running can be seen. Here the player is looking at two enemies homing in on the player position, a red sphere is seen which is a shot recently fired by the player. Information such as the player's score, time left and the controls are visible so the player does not have to remember this and feels in control of the situation.

5.1 ML-Agent Design

The machine learning agent designed as a part of this study is rather simple given the complex nature of interactive video games. This leaves much to be desired when it comes to the agent's behaviour given the time it takes to train a somewhat decently performing model.

The agent has also proven to perform irregularly under testing due to the unpredictable nature of player movement. Many of the players move constantly and unpredictably, causing the machine learning agent to either act randomly or freeze up. This can be attributed to the agents being trained to optimize a route from their start position to a random static target position,

meaning if the players keep moving the agent can only rely on its trained sense of using raycasts to home in on the player. Often the player moves out of these raycast fans leaving the agent to try its best to apply the model on a situation it is not prepared for.

5.1.1 ML-Agent Training Progress

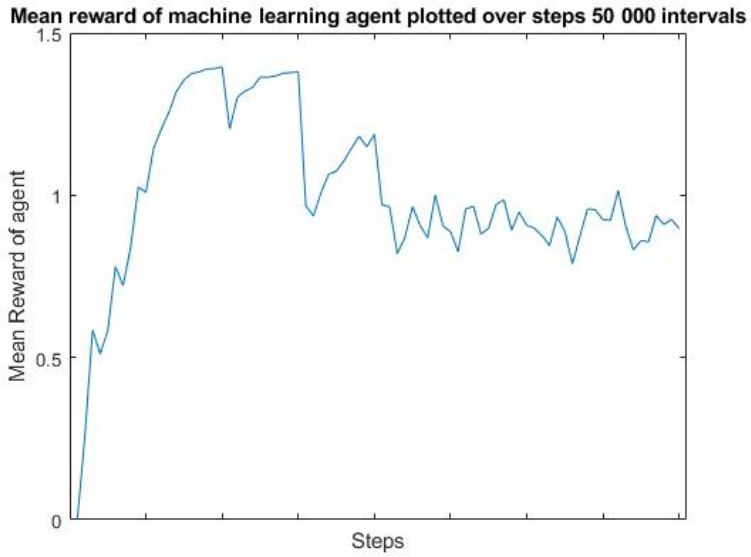

Figure 8: Progress over time of machine learning model.

The training progress of the machine learning model went as expected once the initial necessary factors in the training environment had been correctly set up. The mean reward was steadily increasing, as can be seen in figure 8, until a new obstacle, factor or hindrance is introduced which causes the mean reward to fall until the agent figures out how to handle the new elements in the environment. In the early stages of training the agent manages to maximise its reward to almost 1.4 which was the maximum amount of reward value the agent could achieve in the training environment. The maximum reward in each scene adds up to 1.4, 1 for reaching the target and 0.2 for each trigger sphere the agent passes which there are two of. This adds up to 1.4 which is an arbitrarily chosen number, apart from it being recommended by the documentation for the ML-Agents toolkit to start out with a base max reward of 1 for the start of training and adding very small numbers to a cumulative reward value. This would only last until elements such as walls or a greater range of possible spawn locations of the target were introduced. In the end the agent manages a mean reward of about 0.9, which out of the maximum of 1.4 is sufficient for this study's scope. When observing the agent play against the stationary target it reliably moves to

the target, avoids walls and doesn't get stuck. What drags down the score is that the agent has to search its environment for passages and after that navigate, which it has trouble with due to not braking and turning sufficiently to take some corners efficiently enough to raise its mean reward further.

5.2 Questionnaire Answers

With the machine learning model trained and paired with a navmesh counterpart, the game was sent to willing participants who played the game and answered a questionnaire. The answers provided by the participants along with the statistics gathered from their play sessions gives us valuable insight into their perception of the different kinds of enemies.

Overall the experience with video games of the participants are either average or high, along with most of them being experienced with first person shooter games (FPS). This is good as they will most likely adapt quickly to the game instead of being overwhelmed by the controls or concept of a FPS game.

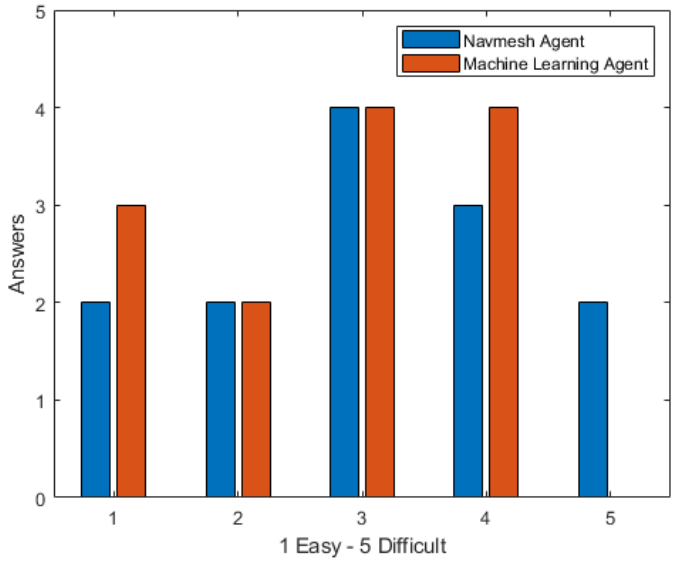

Figure 9: How challenging the enemies was to play against, *"Easy"* to *"Difficult"*.

Looking at figure 9 we can see the answers regarding how challenging the participants found the two sets of enemies. Overall it seems the two agent types are of equal challenge, with the navmesh agent scoring just slightly higher on average.

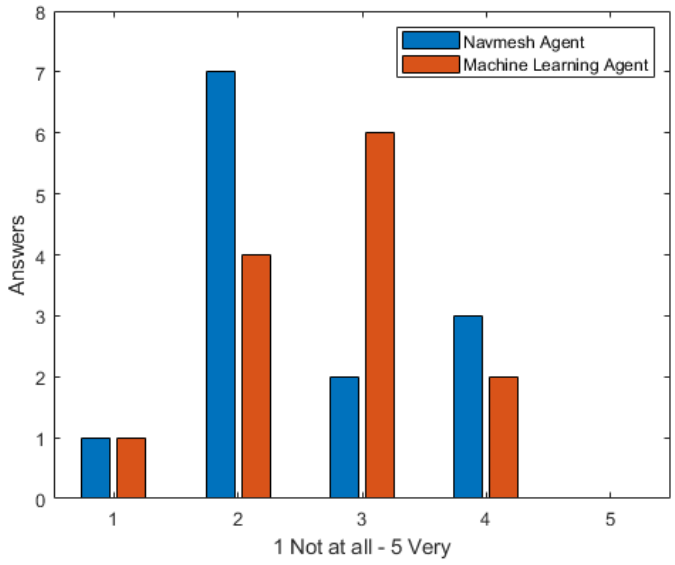

Figure 10: How enjoyable the enemies was to play against, *"Not at all"* to *"Very"*.

This can be put in contrast with the figure 10 which shows how enjoyable the different enemies were for the players to play against. A comparison of the two graphs shows that even if they provided a somewhat equal challenge, the navmesh agent was less enjoyable to play against. Though the machine learning based ones score was average, it still got a higher enjoyment score than its navmesh counterpart.

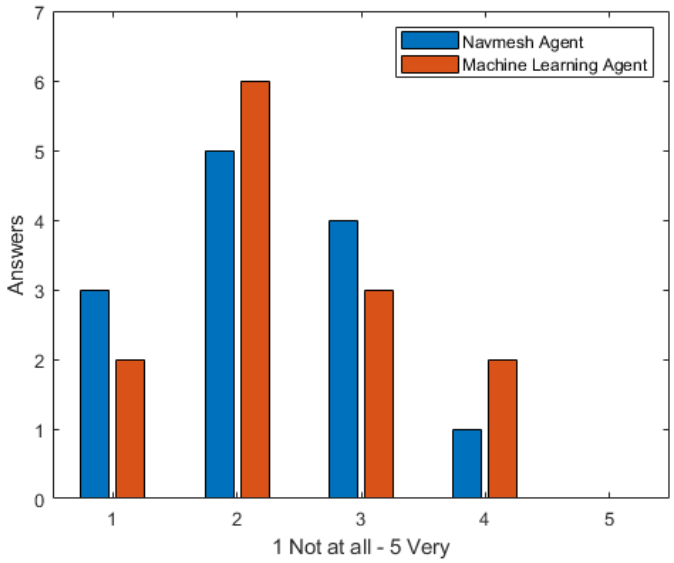

Figure 11: How interesting the enemies was to play against, *"Not at all"* to *"Very"*.

The final figure for the questionnaire answers, being figure 11 seen above, relates to how interesting the players found the two enemy types. A similar score is seen for both agent types here, which tells us more about the test game itself rather than the enemy types.

	Challenge	Enjoyment	Interest
Navigation mesh Enemy	3.0769	2.5385	2.2308
Machine Learning Enemy	2.6923	2.6923	2.3846

Table 1: Mean values of questionnaire questions.

Finally the questionnaire answers are summarised as mean values, which can be seen in table 1 above. Looking at these values we can see subtle differences between the perception of the two enemies, such as the difference between mean challenge, as well as difference in mean enjoyment between the two models.

5.3 Player gameplay statistics

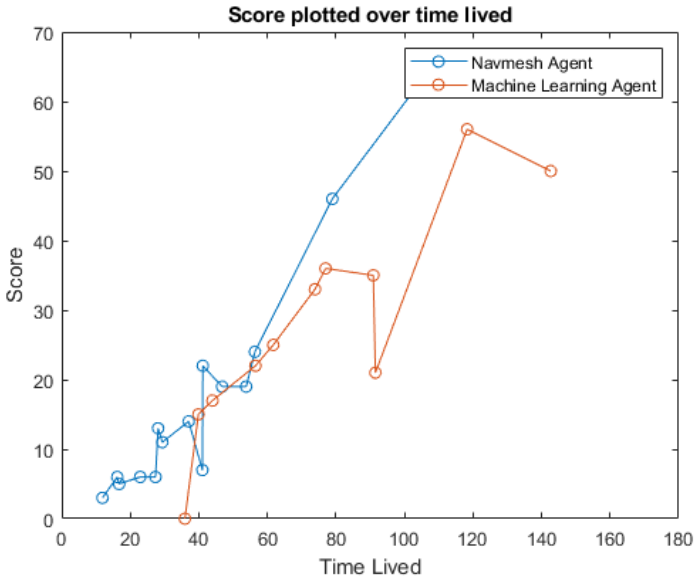

Figure 12: Score achieved during the played round plotted over the time the players managed to stay alive.

Finally we map the gathered player statistics in regards to how much score they gathered and how long they stayed alive. This can be seen above in figure 12 where we can see, as hoped, two linear lines increasing as higher scores were achieved. This tells us that the two types of enemies were similar in challenge.

6 Analysis

Upon reviewing the results, several insights come to mind. Looking at the figure 8 and comparing to the score of the ML-Agent given by the player, it is noteworthy that even if the agent progresses in its training and manages to achieve higher reward score as the training progresses, it seemed it gave no greater impact on the player experience. This shows the difficulty of training a machine learning model for complex tasks with many factors. It might be easy to just train a machine learning model, but to train correctly for the task it is to face or solve is very difficult. One must always iterate back and look over what the machine learning model is being trained to do, and if the training reflects the coming challenges.

Looking over the barcharts for the answers from the questionnaire, we see a similarity with the challenge chart on figure 9 and the line graph of figure 12. Looking at these two figures we see that players rated the two types of enemies as almost equally challenging, and the line graph confirms this. For both enemy types enemies achieved a similar score in the same time, only at the far end of the spectrum for score achieved was there some players who managed to stay alive far longer. These values could be outliers or the machine learning model was easier to handle in the long run due to the faulty behavior of the model, such as getting stuck on walls or *"giving up"* when the trained model could no longer be applied to the given situation.

Looking over figure 11 regarding how interesting the player perceived the enemies, the overall score is on the low end of the scale. This can lead to discussion about how to actually test these kinds of fundamental changes in a game. Is the enemies not interesting due to the bland and simple design of the game, which lacks immersion factors such as complex 3D models, music and a compelling environment. Or was the game too simple? Would a larger, more complex game have been more interesting? If so, how large does a study of this type have to be in scope to train a sufficiently advanced testing model for the machine learning agent such that it can provide a meaningful challenge and still be simple enough that we can analyze and control the different factors of the test?

Calculating the mean of the questionnaire answers gives another perspective on the difference. Looking at the mean values, found in table 1 found in chapter 5.2, one can see that the difference is not as large as it may appeared by just looking at the barcharts. Some differences to note though is that the challenge is about 0.3 higher for the navmesh agent, which can probably be attributed to the two fives given in challenge by the players to the navmesh agent, meaning the mean value is shifted higher. These fives can be assumed to be averaged out if the test had a larger sample size of test players. Still noteworthy.

Also one can see slightly higher scores overall for the machine learning enemy compared to the navmesh enemy in both areas of enjoyment and interest, which gives merit to the studies research question if there is a future for machine learning models as enemies in future games.

7 Conclusion

After comparing the answers from the questionnaire along with looking at achieved scores from players and taking into account the free form replies, a series of conclusions can be drawn regarding machine learning models in games and the process of which to get them to work.

Overall players seemed to prefer the ML-Trained Agent compared to the navmesh, due to its more varied and interesting behaviour. Feedback from test players suggest that even if the ML-Agent was unpredictable, sometimes providing an entertaining and good challenge, it was also at times lackluster and merely got lost or freezed. This lackluster behavior can be attributed to the machine learning model not having been trained against moving targets, resulting in that the common practice of the players to strafe around the arena when playing caused the agents to *glitch out* due to their model not being trained for a moving target that avoids them. One can imagine that this is both the weakness and strength of the ML-Agent, with incorrect or too little training it can't act according to the situation and therefore does not fulfil its objective reliably. But if trained enough and correctly it acts more interestingly, in ways that normal NPCs either can't or need a lot of programming to even come close to mimicking.

Either way the scope of the study enabled the training of a machine learning model which put up a statistically comparative challenge for the player, but in edge cases it either behaved unfairly or did not even try to defeat the player, resulting in what can be assumed to be weighted mean value for challenge score. These models achieved the intended goal to see if there was a difference in how the player perceived a more traditional NPC compared to a trained machine learning model driven one. There seemed to be a higher level of interest and enjoyment in playing against the machine learning trained one indicating that there is possibly a future for smarter enemies in video games. This was achieved while at the same time assuring that the challenge presented by the enemies was equal and fair. This could be done via both asking the players how they perceived the challenge put up by the two enemies, which showed an equal perceived challenge by both enemies. But also plotting player performance showed similar trends in player performance indicating that the enemies were of equal challenge.

To be able to use machine learning in games it needs to be applied in environments where it can be trained via reinforcement learning where rewards can be given in a controlled manner, fostering a desired behaviour before introducing annoyances and disturbance, perfecting the model. Certain genres seem more suited for adapting machine learning for the NPCs, such as racing games or 2D platforming games, as they have a limited amount of factors to consider or the conditions remain local and assessable.

Still this study proves that machine learning can be used in video games in the future to provide more interesting behaviours from NPCs, even if they were simple and sometimes faulty, the enemies in our study still managed to challenge the players while giving an enjoyable gaming session in some sense.

7.1 Ethical aspects

Before implementing and training a machine learning model to use in their game, one should consider how this machine learning model will learn to play and what it will use to improve over time. It might be fair to train a machine learning model against a base set of scenarios so it performs those optimally, but one of the interesting aspects of machine learning is that the machine learning model should improve over time or throughout a play-through of a game. This raises the questions if the machine learning model should be trained individually or collectively. It might be tempting to train the model against all players who play the game as a summarised model. This might yield a more complex and smarter enemy over time, but it might also present an unfair model towards players with less experience with games or with players with disabilities. If the model is trained against a majority of players who have no disabilities and a lot of gaming experience, the models might become unfair and outright punishing to play against for minorities with less experience or disabilities.

So it is of great importance to consider what one's targeted and potential player group is. How the machine learning model will learn from these and how the model will be employed. In more multiplayer oriented or competitive experiences it might be best to train a collective model so that all face a similar enemy, while in single-player games individual training might be preferred as the model adjusts according to each player's needs and capabilities. One should also do some research on how they can limit the machine learning model so it does not optimize the problem of outsmarting the player. It might be fun to have enemies who adapt overtime and learn new tactics, but if they learn unfair tactics to outplay the player by the end of the game, the game will not be enjoyable.

7.2 Future Work

For future work, it would be suitable to train adversarial models to achieve a more dynamic relationship, where one model is trained in the role that the player will take upon playing, meaning that the enemy model is more suited in actually outsmarting and acting against the player. This can be done in the Unity engine as several test examples already exist of small scale adversarial play.

References

[1] *GameplayKit Programming Guide: State Machines*. Mar. 2016. URL: https://developer.apple.com/library/archive/documentation/General/Conceptual/GameplayKit_Guide/StateMachine.html.

[2] Mengchen Hu. *Game Design Patterns for Designing Stealth Computer Games*. 2014. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:mau:diva-20294.

[3] Robert Nystrom. *Game Programming Patterns - Design Patterns Revisited - State*. Nov. 2014. URL: https://gameprogrammingpatterns.com/state.html.

[4] Unity-Technologies. *Unity ML-Agents Toolkit*. URL: https://github.com/Unity-Technologies/ml-agents.

[5] Qi Jiangyuan. "Simulation of auto obstacle avoidance based on Unity machine learning". eng. In: *Journal of physics. Conference series*. Vol. 1883. 1. Bristol: IOP Publishing, 2021.

[6] Muhammad Aminul Akbar. "NPC Braking Decision for Unity Racing Game Starter Kit Using Naïve Bayes". eng. In: *Fountain of Informatics Journal* 4.2 (2019), pp. 61–68. ISSN: 2541-4313.

[7] Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, et al. "Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents". In: *Journal of Artificial Intelligence Research* 61 (2018), pp. 523–562.

[8] Marek Wydmuch, Michał Kempka, and Wojciech Jaśkowski. "ViZDoom Competitions: Playing Doom from Pixels". In: *IEEE Transactions on Games* (2018).

[9] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A physics engine for model-based control." In: *IROS*. IEEE, 2012, pp. 5026–5033. ISBN: 978-1-4673-1737-5. URL: http://dblp.uni-trier.de/db/conf/iros/iros2012.html#TodorovET12.

[10] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, et al. *Unity: A General Platform for Intelligent Agents*. 2020. arXiv: 1809.02627 [cs.LG].

[11] Martin Abadi, Paul Barham, Jianmin Chen, et al. "TensorFlow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[12] Adam Paszke, Sam Gross, Francisco Massa, et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. arXiv: 1912.01703 [cs.LG].

[13] Benjamin Geisler. “An Empirical Study of Machine Learning Algorithms Applied to Modeling Player Behavior in a" First Person Shooter" Video Game”. PhD thesis. Citeseer, 2002.

[14] Niels Justesen, Philip Bontrager, Julian Togelius, et al. “Deep learning for video game playing”. In: *IEEE Transactions on Games* 12.1 (2019), pp. 1–20.

[15] Kenneth O Stanley, Bobby D Bryant, Igor Karpov, et al. “Real-time evolution of neural networks in the NERO video game”. In: *AAAI*. Vol. 6. 2006, pp. 1671–1674.