Implementing machine learning (AI) in game development with Unity

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor thesis

Degree Program in Business Information Technology
or Computer Applications
Hämeenlinna University Centre
fall, 2021

Arno Breugelmans

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology **Abstract**

Hämeenlinna University Centre

Author	Arno Breugelmans	Year 2021
Subject	Implementing machine learning (AI) in game development with Unity	
Supervisors	Deepak Kc	

ABSTRACT

The purpose of this thesis was to research the possibilities of machine learning in video game development and implement machine learning in a Unity environment. This thesis was self-commissioned, the author sought to improve his knowledge of machine learning and the potential implementation in video games. The thesis aims to answer the following questions: (1) What does machine learning in video games mean? (2) What will the machine learning agent replace (nonplayable character/procedurally generated content/simulation)? (3) How is machine learning implemented in Unity?

The research questions were answered in the following way. First there is the theoretical framework research which covers the foundation of machine learning and why machine learning is a tool that should be investigated by game developers. The second part of the thesis is a practical implementation of machine learning using ml-agents and Unity highlighting the potential machine learning has.

The result of this thesis is a theoretical framework and a practical demonstration of machine learning implemented in a video game using ML-Agents and Unity. This combination of research and demonstration proofs that machine learning has many potential use cases and is ready to be implemented in video game development. Developers should investigate implementing machine learning in their development process.

Keywords machine learning, video game development, deep reinforcement learning, Unity

Pages 50 pages and appendices 3 pages

Glossary

2D	Two-dimensional
3D	Three-dimensional
AI	Artificial intelligence
ANN	Artificial neural network
DDA	Dynamic difficulty adjustment
MDP	Markov decision process
ML	Machine learning
MOBA	Multiplayer online battle arena
NPC	Non-player character
PCG	Procedural content generation
PCGML	Procedural content generation via machine learning
PPO	Proximal policy optimization
RL	Reinforcement learning
SDK	Software development kit

Contents

1 Introduction

The IT sector is an ever-evolving cycle of changes and improvements. This applies to machine learning as well. The term “Machine learning” goes as far back as 1949 (Foote, 2019). The technology has lived through different names and changes, but the base concept of learning without specific instructions has remained the same. Dynamic programming is the predecessor of machine learning. Dynamic programming solves for the optimal policy at a given time (Bellman, 1954). Some of the techniques from dynamic programming still find use in today’s machine learning algorithms.

Even though machine learning dates back over 50 years, it still grows immensely and relatively new techniques are still being documented. Improving games using AI (artificial intelligence) is nothing new, but just like machine learning went through its development, so did the integration of video games and machine learning. There is no grand-scale industrial implementation of machine learning in video games but machine learning in video games has potential growth in the future. There are several ways where machine learning can improve or innovate in the game industry.

The aim of this thesis is to research and implement the use of machine learning in video game development. It does so by answering the following three research questions:

(1) What does machine learning in video games mean?

(2) What will the machine learning agent replace (nonplayable character/procedurally generated content/simulation)?

(3) How is machine learning implemented in Unity?

2 Methodology

This thesis is structured in two main sections followed by a result and summary. The first part of the thesis covers the theoretical background of machine learning in video games. The second main part consists of the implementation of machine learning based in Unity and the results concluded from that implementation. From the theoretical framework and the results from the implementation a conclusion was drawn. In the following sub-chapters the methods used to form these sections is described.

2.1 Research

The research type of this thesis can be described as a qualitative research based on secondary research data collection (Bhandari, 2020). To reach a greater understanding on the thesis topic, suitable literature was analyzed and collected using these keywords: ”Machine learning”, “ANN”, “Reinforcement learning”, “ML-Agents” and more on the following databases, platforms and libraries: https://scholar.google.com, https://github.com/Unity-Technologies/ml-agents, https://deeplizard.com, https://arxiv.org and others.

2.2 Scope

The scope of this thesis consists of the basic principles and theory of machine learning, the implementation chapter focuses on implementing one learning technique in a two-dimensional (2D) Unity game. The thesis does not cover the financial aspect of implementing machine learning and it does not cover creating the game in Unity.

3 Machine learning theoretical framework

To answer the first two research questions, some theoretical background knowledge is needed. Firstly, clarification of what machine learning means in general and how it applies to video games is required. After that, a more thorough investigation in the different types of techniques and how these algorithms work is required. Lastly, the use cases of machine learning in video game development are explored.

3.1 Machine learning in video games

Machine learning provides computer systems the ability to automatically learn and improve from experience. Machine learning can be defined as a system or "machine" that can learn from experiences without following specific instruction that are written. This is done through the use of algorithms and models so that the system is able to analyze patterns in data. (IBM Coud Education, 2020)

Machine learning (ML) is usually divided in three categories/paradigms (Parikh, 2018).

- Supervised learning
- Unsupervised learning
- Reinforcement learning

The most common paradigm of machine learning in video games is reinforcement learning.
There are many ML techniques that video games can utilize. The basic premise of these techniques is that the machine or commonly called the agent, takes an action in the environment, and based on that action it gets a positive or negative reward. (Richard S. Sutton, 2018)

3.2 AI compared to machine learning

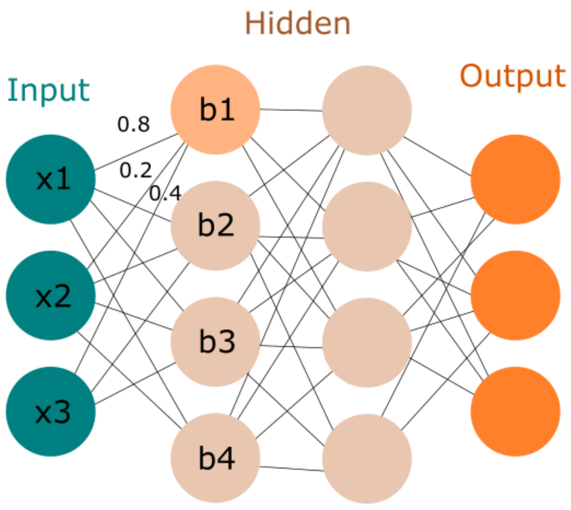
Most video games utilize some form of artificial intelligence (Safadi, Fonteneau, & Ernst, 2015). Whether it is the final boss the player is struggling to defeat at the end of the level or the training bots that are fought during the tutorial of a multiplayer game. Even the automatically generated levels found in dungeon crawlers make use of artificial intelligence. So, with the rise and common use of AI in games there inevitably comes the rise of ML. ML is a subset of AI, therefore all machine learning can be considered artificial intelligence but not all artificial intelligence can be considered machine learning (Garbade, 2018).

AI is most commonly used to create non-player characters (NPCs), if this NPC were to learn from actions it takes then that can be stated as ML. The line to separate AI and ML is thin for the two are heavily correlated because ML is a subset of AI. For the player there is no real use to know whether something is machine-learned or not. The difference in ML and AI is mainly semantic. (Iriondo, 2018)

3.3 Artificial Neural Networks (ANN)

An ANN is a technique based on biological neural networks; this system or technology tries to mimic functions of the human brain. An ANN is a subset of machine learning (Dormehl, 2019). Each node is called a neuron, the lines pointing between the neurons are the connections. These neurons and connections loosely resemble how biological neurons work. A network usually consists of three layers: the input, the hidden and the output layer. The hidden layer can consist of multiple layers. The input layer receives inputs such as user given data, images and environments. The output layer predicts the result. The hidden layer processes these inputs to get to predict the outcome. Neurons can hold a value between 0 and 1. The closer to 1, the more chance this neuron activates. (3Blue1Brown, 2017)

Figure 1 Composition of an artificial neural network



The input layer is connected to the hidden layer. Each of these connections has a weight, a number identifying how important that connection is. The hidden layer neurons activation depends on what they receive from the input neurons. It calculates the sum of all weights times their input combined with a numerical number called the bias if this exceeds a certain threshold also known as the activation function, then the neuron activates. These neurons do the same to the next layer, the act of forwarding data to the next neurons is called forward propagation. When the output layer is reached the neuron with the highest value activates. Here is the formula for one node in the hidden layer:

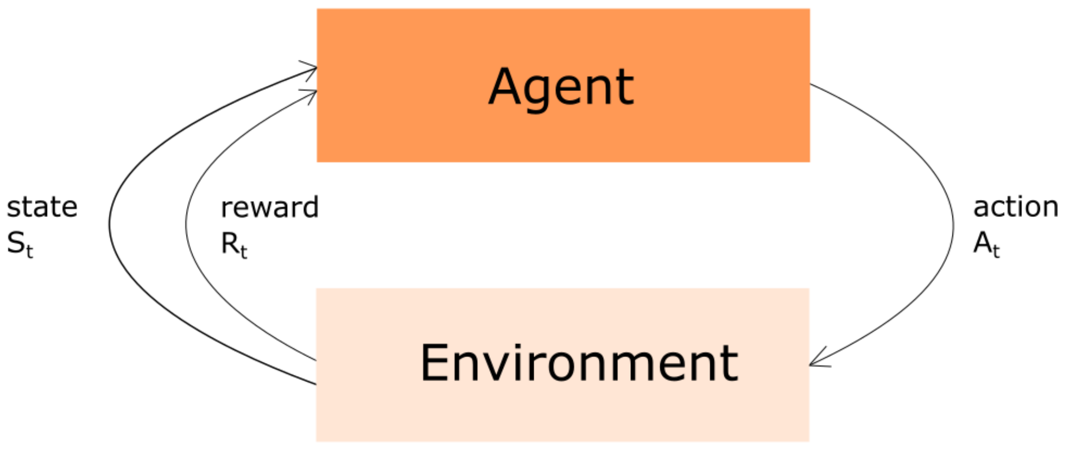
$$(x1 * 0.8 + x2 * 0.2 + x3 * 0.4) + b1 > activation\ function$$

Neural networks usually work on the supervised learning paradigm. Training the neural network requires a human to designate inputs and the desired outputs (Trevor Hastie, Robert Tibshirani, 2009). Neural networks improve their predictions by adjusting the weight between each node. This is done through backpropagation; the data is sent back through the network. This process of forward and backpropagation continues until the network can make near correct predictions.
A comparison to artificial neural networks can be made with the human sensory system. For instance, a human sees the light turning red. That information is processed and an appropriate action is taken, which in this case is to stop. The eyes are the input layer, what is thought is the hidden layer and the stopping of the human is the output layer. (Simplilearn, 2019)

3.4 Reinforcement learning

One of the main ML paradigms used in video games is reinforcement learning. The agent uses the following variables when training with reinforcement learning: Action, Environment, State and Reward. The concept of reinforcement learning is that there is an agent and an environment. These two communicate through actions and states. This can be modeled as a reinforcement cycle. (Richard S. Sutton, 2018)

Figure 2 Reinforcement learning cycle



3.4.1 Markov decision process (MDP)

An agent makes an action onto the environment. The environment sends the state back to the agent. This state-action pair results in a positive or negative reward for the agent. This process is also known as the Markov decision process. In the MDP there are a set of states (S), a set of actions (A) and a set of rewards (R). Each step in time (t) the agent receives the environment's state (S_t) and makes an action (A_t), this is the state-action pair (S_t, A_t). Iteration over the process of actions and states is started at S_t, based on this state the agent makes an action A_t. The environment then in turn gives a new state and reward S_{t+1} and R_{t+1}. ("Markov Decision Processes (MDPs) - Structuring a Reinforcement Learning Problem - Deeplizard," 2018)

3.4.2 Expected return and episodic learning

Reinforcement learning is an unsupervised iterative process, this means that there is no external source that provides the system with expected outputs (Richard S. Sutton, 2018). Therefore, the agent needs to find actions that give it the highest cumulative reward. To find the highest reward the agent makes use of expected return. The expected return is what drives the agent to make its decisions. The expected return can be defined as the sum of the current and all future rewards defined as R. The return at time t is defined as G. Where T is the final time step. (Blackburn, 2019)

$$Gt = R_{t+1} + R_{t+2} + \dots + R_T$$

The expected return G calculates the sum of all rewards for each time step. This definition of expected return only works for episodic learning. Episodic learning is the act of splitting up the training in certain parts, after each part the environment is reset and the new episode begins. For example, in Mario the episode would end when Mario dies or reaches the goal. This means that there is always an end so there is always an end time step. Not all reinforcement learning use cases are defined as episodic tasks, when the environment is not divided into episodes continuing tasks are acquired. In a continuing task the end step T would be infinite and therefore the reward would be infinite as well. ("Expected Return - What Drives a Reinforcement Learning Agent in an MDP - Deeplizard," 2018)

The problem of continuing tasks can be solved by calculating the discounted return instead. With the discounted return the goal of the agent is to find the maximum expected discounted return of rewards. The discounted return is calculated as following:

$$Gt = R_{t+1} + \gamma R_{t+2} + \gamma R_{t+3} + \dots$$

$$\Rightarrow Gt = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

This formula contains the discount rate γ, a number between 0 and 1. The discount rate makes the agent value future reward less than immediate rewards. This is done through discounting future rewards at later time steps. The discounted return means that for each time step further in the episode the agent is, the rewards are diminished. The agent considers rewards near the immediate future more valuable. ("Expected Return - What Drives a Reinforcement Learning Agent in an MDP - Deeplizard," 2018)

3.4.3 Policies

In the previous chapters it is explained how the agent calculates the excepted return but not how the agent chooses which actions to take. Policies and value functions are responsible for determining the actions the agent takes. A policy is a function that calculates the probability of choosing each possible action from a specific state. An agent follows a policy which is denoted as π. A value function is a measurement of how good a certain state is or state-action pair. The policy and value function are closely related. The value function is given in the form of the expected return and this expected return changes the way an agent acts as this drives the agent. The way an agent acts is bound to the policies; therefore, the value function is defined according to the policy. The value function calculates the value of either a state or state action pair. This means that there are two value functions. The state-value function and action-value function. The state-value function calculates the value of any state following policy π, this is defined as $V_{\pi}(s)$. The action-value calculates how good any action is for that specific state when following the policy. The action-value function is defined as $Q_{\pi}(s,a)$, this action-value function is also known as the Q-function and the result from this function is known as the Q-value. The agent tries to find to find the policy that gives the most rewards. In other words, the agent tries to find the best policy, better known as the optimal policy. The optimal policy is the policy that yields a better expected return than any other policy. Policies are linked to the value functions, this means that the optimal policy has both a optimal state-value function and an action-value function. ("Policies and Value Functions - Good Actions for a Reinforcement Learning Agent - Deeplizard," 2018)

3.4.4 The Bellman equation

$$The\ Bellman\ equation:$$
$$V(s) = max_a(R(s,a) + ɣV(s'))$$

The Bellman equation solves for the optimal state-value function, to understand this formula an example to explain how the bellman equation calculates the value of a certain state was made, this example reinforces the concept of discounted return. The value of a state $V(s)$ equals to the value of the reward of the action the agent took in a certain state $R(s,a)$ plus the discount factor Gamma ɣ multiplied by the value of the future state $V(s')$. Max action max_a means that from all the possible actions, the agent takes the action that gives the best value. (Geek, 2018)

This is Spider Lonky Leg (Figure 3) and to win the game the spider must slash the treants (tree enemy) but must avoid falling on spikes. Lonky gets a positive reward of 1 when hitting a treant. A neutral reward of 0 is given when the spider lands on an empty square and a reward of -1 (a punishment) is given when the spider lands on spikes. Lonky has four possible moves. Up, left, right or down. If Lonky gets to the treant or hits the spikes the episode ends.

Figure 3 Base setup for Bellman equation example

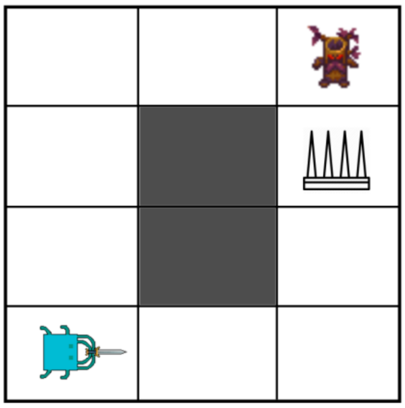
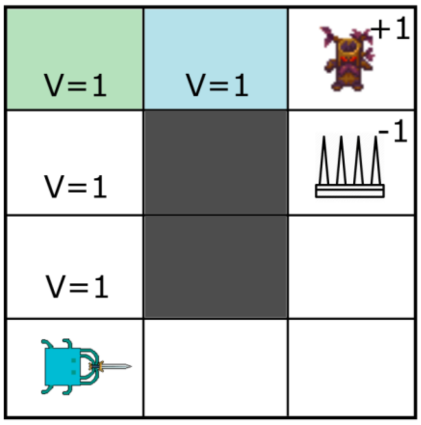


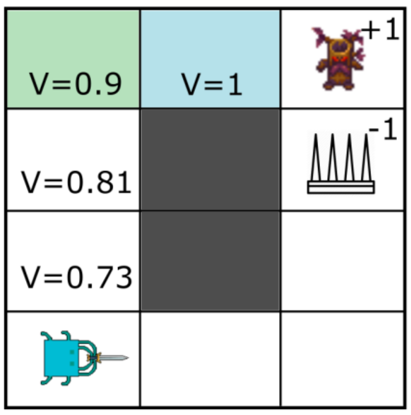
Figure 4 Calculating the path and colored squares



When the Bellman equation is applied the value of the state Lonky has can be calculated. To simplify the discount factor is set to 1. It is known that the treant gives a positive reward of 1. Calculating the value starting one square to the left (the blue square, Figure 4): $R(s, a) = 1$, the reward for getting to the treant. $V(s') = 0$, the value of the future state is zero because the episode ends when the agent gets to the treant. Combining these 2 values returns a state value of 1. Move one square to the left (the green square) and apply the Bellman equation again: $R(s, a) = 0$ and $V(s') = 1$. The reward the agent gets for being on that square is 0 as this is an empty square, it is known that the next square gives the spider a reward of +1. This is assuming the agent takes the optimal decision, which the developer knows is to go right (the blue and green square are the same as the white/empty squares, the color is for clarity). (Geek, 2018)

It is found that the value of all the squares equals to 1 (Figure 4) if this is done for every square. The cause is that all squares are technically equally valuable, however the developer knows that the squares next to the treant are better positions. This is where the discount factor takes its place. Assuming a discount factor of 0.9. Calculating the value of the blue square gives 1: $R(s,a) = 1$ and $\gamma V(s') = 0.9 * 0 = 0$. Calculating the value of the green square: $R(s,a) = 0$ and $\gamma V(s') = 0.9 * 1 = 0.9$ The path values in Figure 5 are acquired if this is done for every square. This is how the Bellman equation solves for the value of a state. (Geek, 2018)

Figure 5 Calculating the path and colored squares with discount factor

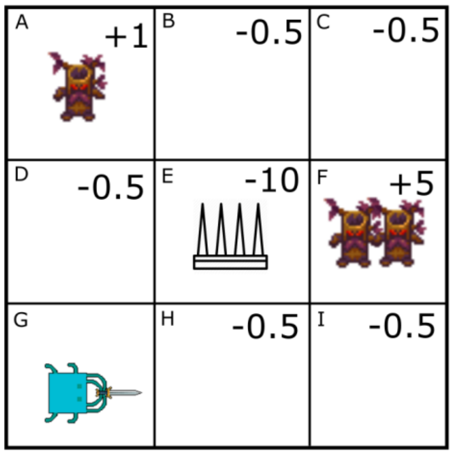


3.4.5 Q-learning

To find the optimal policy the agent makes use of Q-learning. Q-learning finds the most optimal policy, the policy that gets the highest expected return. Q-learning finds the optimal policy through finding the optimal Q-values, this is done through iteratively updating Q-values until the Q-function reaches the optimal value, this is called value iteration. (“Q-Learning Explained - A Reinforcement Learning Technique - Deeplizard,” 2018)

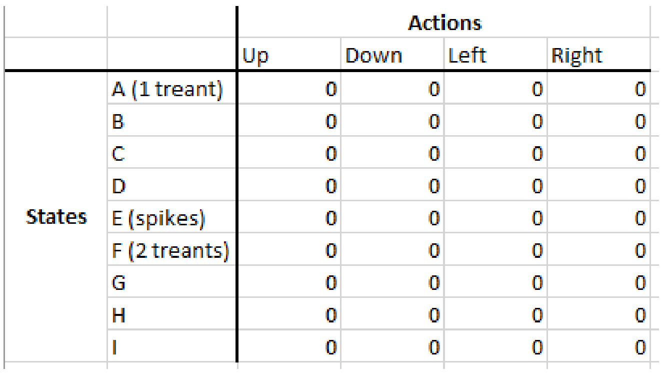
The base setup of the game is the same. The spider can still choose to move in either of the four directions but this time the agent gets a +1 reward for landing on the normal treant and a +5 reward for getting to the two treants. The developer does not want the spider to hit the spikes so a negative 10 reward is given when the spider lands upon the spike. The agent is given a small negative reward for landing on empty spaces, this will reduce the amount of unnecesary steps the agent takes.

Figure 6 Base setup for Q-learning example



Each of these tiles is a possible state the agent can be in. In the beginning the agent does not know anything about the environment, it does not know what gives positive or negative rewards. This means the Q-value of each state-action pair is zero. Throughout the game this value is updated through value iteration. The state-action values aka the Q-values are stored in a Q-table.
In the beginning the table's value are zero (Figure 7).The possible actions are on the horizontal plane and the possible states on the vertical plane. All the values are set to zero as the agent has not explored the area yet. ("Q-Learning Explained - A Reinforcement Learning Technique - Deeplizard," 2018)

Figure 7 Q-table example at start of game



		Actions			
		Up	Down	Left	Right
States	A (1 treant)	0	0	0	0
	B	0	0	0	0
	C	0	0	0	0
	D	0	0	0	0
	E (spikes)	0	0	0	0
	F (2 treants)	0	0	0	0
	G	0	0	0	0
	H	0	0	0	0
	I	0	0	0	0

3.4.6 Exploration vs Exploitation

As the agent goes through the training of mutliple episodes, the table's value content starts to change. In following episodes the agent can refer to the table and make actions based on the value stored in it. The agent makes decisions based on the table values but at the start of the training the table is empty, therefore the agent does not know what action to take. In another scenario the agent found the treant on the "A square" and gets a positive reward, then the agent might only go to this square because in the Q-table it is a good decision to go there and the agent might never explore the rest of the area. This problem is solved through the balance of exploration vs exploitation. Exploitation means that knowlede the agent already has is exploited, the agent makes actions based on previous knowledge that it knows gives a positive result. Exploration means that the agent explores or searches for new things in the environment and ignores the already known states; the agent searches for different states for a potential better reward. If the agent leans too much on exploitation it never finds all positive rewards. If the agent leans too much on exploration it doesn not use previous knowledge and makes too many random moves. This balance between exploration and exploitation can be found with the help of the Epsilon-Greedy algorithm. ("Exploration vs. Exploitation - Learning the Optimal Reinforcement Learning Policy - Deeplizard," 2018)

The Epsilon-Greedy alogrithm has exploration rate ε which is similar to the discount rate, the exploration rate diminishes with each episode. If the exploration rate is set to 1 then there is a 100% chance that the agent chooses exploration. The algorithm starts of by setting the rate to 1 and diminishes it each episode. The lower the rate goes the more the agent chooses to exploit rather than explore, in other words, the agent becomes more greedy. The way the agent chooses exploration or exploitation is simple. On each step the agent rolls a random number between 0 and 1, if the randomly generated number is higher than the exploration rate then the agent chooses to exploit, if it is lower it chooses to explore. This is why the agent always explores in the first episode if the exploration rate is set at 1 at the start. ("Exploration vs. Exploitation - Learning the Optimal Reinforcement Learning Policy - Deeplizard," 2018)

3.4.7 Learning rate

To conclude the discussion on reinforcement learning one more topic is covered, the learning rate. Previous knowledge tells us that the algorithms iteratively update the Q-values in the table. The table is filled after a couple of episodes. The next episode the agent learns something new about the environment and goes back to one of the previously known states. This state's value is now different, the agent needs to decide on how to update this value. To calculate the new Q-value learning rate is used. The learning rate denoted as α is the rate of how much the newly calculated value should replace the previously already calculated value. The learning rate is a value between 0 and 1. If the learning rate is 1 then the agent chooses to simply replace the Q-value as a whole. The closer to zero the more it keeps the already calculated values. ("Exploration vs. Exploitation - Learning the Optimal Reinforcement Learning Policy - Deeplizard," 2018)

3.5 Deep reinforcement learning

In the game example of reinforcement learning the agent had quite a limited number of states. In many games this is not the case, for many games the MDP states are of too high dimensions. The raw pixels from the game are often used as input. This results in far too many states and Q-values to store in a Q-table or other standard reinforcement learning methods. Deep reinforcement learning is the act of combining deep learning and reinforcement learning (François-Lavet et al., 2018).

The Markov decision processes are kept but they are solved with the use of artificial neural networks. This solves the problem of having a highly dimensional environment. The Q-learning algorithm was explored in one of the previous chapters. This algorithm can be modified to work in the deep reinforcement learning environment, the combination of these 2 techniques is called deep Q-learning. These Q-learning algorithms are called value-based algorithms. In chapter 5 the agent is using policy gradient algorithms. This technique still uses the same cycle of an agent that makes an action onto the environment and gets a state and reward back in return (Figure 2). It still uses the MDP, but this time the algorithm does not solve the probabilities with Q-learning. The definition is still the same. "Reinforcement Learning Objective: Maximize the "expected" reward following a parametrized policy" (Kapoor, 2018). The way value-based and policy gradient algorithms differ is on how they determine what action to take. While Q-learning tries to find the maximum expected return from a single deterministic action from a set of actions, policy gradients tries to learn a map from state to action. This means that policy gradient methods can work on larger spaces and work with continuous action spaces as well. That is the reason why in most game environment policy gradients are used. Policy gradients can be used on stochastic environments. (Spinning up, 2018)

3.6 Using machine learning in video games

Machine learning can be a very powerful tool when used in the right way. Before ML is applied in video games it is important to analyze where exactly ML could be of use. The question game developers should be asking is what problem is being solved, and is machine learning the best option. Simplicity and cost makes classic AI a common option. (Penelope Sweetser, 2002)
Here are some potential goals machine learning can strive to solve or improve upon:

- NPCs
- Training professional players
- Adjusting difficulty level
- Bug finding
- Procedural content generation

3.6.1 NPCs

The current state of AI is rather advanced already and is not in a spot where it needs an immediate change. Traditional AI mainly makes use of Pathfinding and State machines. Pathfinding is finding the path between two given points and State machines is having different states which can be switched between. (Statt, 2019)

There are of course more advanced techniques like decision trees and behavior trees. So there is no real need to improve upon NPCs with machine learning, there is in fact the disadvantage of the agent being too unpredictable or too difficult for players to beat if an agent is trained with ML. Machine learned agents have the tendency to not adapt well to different environments (Grosse, 2018).

When applying ML to NPCs it is important to know what the goal for the NPC is. Most use cases of NPCs is to make something feel like a human or behave smartly (Henrik, 2015). Most of the time traditional AI has enough tools to implement the wanted NPC behavior without the need for machine learning (Penelope Sweetser, 2002). It is when the game developer wants to enhance this experience or make radical changes that ML is of use. A possible NPC machine learned agent use case is when the goal is to create an extreme difficulty game with enemies the game developer wants to feel frustrating and unbeatable.

3.6.2 Training professional players

Many competitive multiplayer games include tutorials or a practice mode of some sort, where the player faces off against bots. Bots are AI players, computer-controlled players. Usually, these are meant to be easy and predictable so that new players can learn the game without having to face other real people who have mastered the game already. In most multiplayer games the options one agent/bot can take is endless, there would need to be multiple decision trees/state machines if an advanced type of bot using AI is desired. Usually, bots do not need to be more advanced, their main use case is to teach new players. (Geisler, 2004)

Professional players play the game 8-12 hours a day. They improve at the game by facing real enemies. The tutorial or practice bots are too easy and exploitable, they are no use for professional players who want to improve their gameplay. Machine learning can solve the problem of making bots more advanced. Machine learning even has the capabilities of making bots better than the top players of that game. (Berner et al., 2019)

A great example of machine learned bots competing against real players is the OpenAI Five project. The goal OpenAI Five successfully achieved is to make a team of machine learned bots beat the top human professional players at the 5 v 5 multiplayer online battle arena (moba) Dota 2. Readers less versed in the gaming community might assume this is an easy task but this is not the case. In MOBA (multiplayer online battle arena) games such as Dota 2 the terms micro and macro mean the following: micro is how the player executes the mechanics of the hero the player is playing; micro is the individual gameplay that the player makes in the moment. Macro refers to controlling the game in a broader sense, grouping up with the player’s teammates, being at the right place at the right time. One might expect that bots would be good at the micro part and that would be correct, as this is purely mechanical skill and bots can execute orders are the most precise level. However, the macro gameplay involves a more human part, intelligence of knowing what to do is required, being able to read situations and act accordingly is mandatory. Since the OpenAI Five bot successfully beat the best players one can conclude that even the macro was learned. The OpenAI Five project shows the true capabilities of ML in video games. (Berner et al., 2019)

3.6.3 Adjusting difficulty level

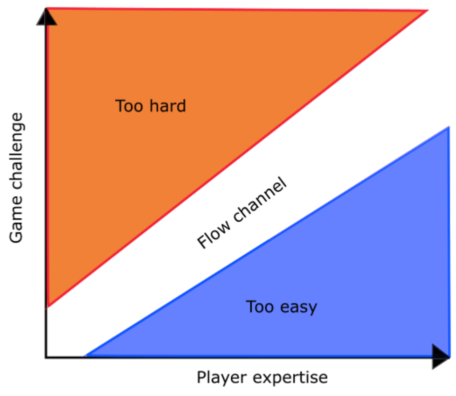
As previously stated, with the power of machine learning developers can make NPCs rather difficult or impossible to beat. A game developer needs to make the game feel fair and rewarding, making enemies impossibly difficult might lead to less enjoyable gameplay. One problem in setting the difficulty in games is that the solution never caters to every single player in the game's audience. Even with the help of difficulty sliders ranging from easy to very hard there are always players that want the game to be easier or harder. (Aponte, Levieux, & Natkin, 2009)

Machine learning can remedy this problem in two ways. The first way is covered to some extent in the part about what machine learning can do on improving NPCs, with ML the developer can train the agent to the level that is desired. ML allows the developer to train the agent for different time periods or make a different easier/harder environment for the agent to learn from if the game requires a different difficulty setting. (Geisler, 2004)

The second way to adjust difficulty level has to do with the balancing itself, game developers want players to find their game not too hard but also not too easy all while feeling natural. This problem was first solved by difficulty sliders. But this quickly became outdated for one static selection does not provide enough stimulus to keep the player entertained. Entertaining games require balance so that the challenge is always around the player's expertise level. If the game challenge is too high for the player then the game feels frustrating, if the game challenge is too easy then the game feels boring. This space in between is what is called the flow channel which is where a game developer wants their game to be Figure 8 (Zohaib, 2018)

In the present time there is (Dynamic Difficulty Adjustment (DDA). DDA is the act of balancing the game's difficulty so that the player is always engaged. This is done through changing several key factors like speed, power, items the player or the opponent has. DDA can be implemented in a multitude of ways. Machine learning gives a real time response to what the player needs. (Zohaib, 2018)

Figure 8 Concept of flow by Csikszentmihalyi



3.6.4 Bug finding:

Game physics engines can display some rather strange behavior including various aspects such as: clipping through walls, gaining too much velocity, floating objects. Machine learned agents are excellent at finding bugs. Most of the time the machine learned agent does this by itself, without the need to program this behavior. An agent going through the many iterations in the training process is bound to find some bugs. The agent does not know it is a bug so if the behavior it encounters helps the agent, it exploits that behavior (Baker et al., 2020). Even if game developers have no plans in implementing machine learned agents into their game it might be beneficial for just the testing phase. It can replace or be combined with traditional game testers to speed up or improve this process. (Zheng et al., 2019)

There is also the possibility of fine tuning the agent to specifically find bugs in the game or bugs in the code. One specific way to find graphical glitches is using convolutional neural networks. This network finds visual bugs on certain three different approaches: object classification, object detection and semantic segmentation. (LING, 2020)

3.6.5 Procedural content generation

PCG (Procedural Content Generation) is the act of generation random or pseudo-random content for a game. Using procedural generation, the developers can make games less repetitive, more random and create bigger worlds. According to Summerville's research paper:
"It is employed to increase replay value, reduce production cost and effort, to save storage space, or simply as an aesthetic in itself" (Summerville et al., 2018, p1).
The base idea of PCG is using algorithms to automatically create content, this content can be in the form of levels, items, graphics and more. This type of content generation is defined by certain parameters, constraints and objectives, these criteria are usually manmade. This means that to create organic levels in the bounds of what should be possible, game developers need to manually fabricate these criteria so that they get the content they desire. Machine learning can improve upon this technique. This is called Procedural Content Generation via Machine Learning, abbreviated as PCGML. There are many use cases of PCGML, one of the simplest use cases is the autonomous content generation. In contrast to traditional PGC which needs a human to create the functions and evaluate the generated content. PGCML does not require any human input but takes predefined inputs and generates suitable outputs. (Summerville et al., 2018)

4 Setting up machine learning in Unity (Windows)

Unity is a cross platform game engine where developers can make real-time three-dimensional (3D) projects for games, animation, film and more. The focus in this thesis is the game making aspect where machine learning is implemented. ML-Agents is an open source project that makes ML available in the Unity engine. (ML-Agents, n.d.-g)

Requirements:

- Knowledge of C# or other supported language for Unity
- Basic understanding of Unity engine
- Python (3.6.1 or higher)
- Unity (2018.4 or later)

Installations:

- Unity package ml-agents
- Unity package ml-agents-extensions (optional)
- Python package PyTorch
- Python package ml-agents
- ML-Agents GitHub repository (optional)

4.1.1 Unity packages

The unity ml-agents package contains the Unity C# software development kit (SDK) that is integrated into the Unity scene. The extensions package are some experimental components that are not implemented in the base package yet.

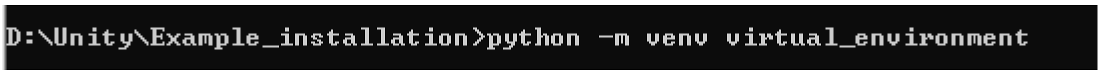
1. Open or create a Unity project where ML is desired.
2. Go to Unity Package Manager (Window -> Package Manager).
3. Find the ml-agents package and click install.

4.1.2 Python PyTorch package

PyTorch is an open source machine learning library. This is what ml-agents is based on and needs to be installed for ml-agents to work. To install the Python packages, it is best to create a virtual environment first. This virtual environment makes sure each project works independently and does not cause any bugs. (PyTorch, 2019)

Creating the virtual environment:

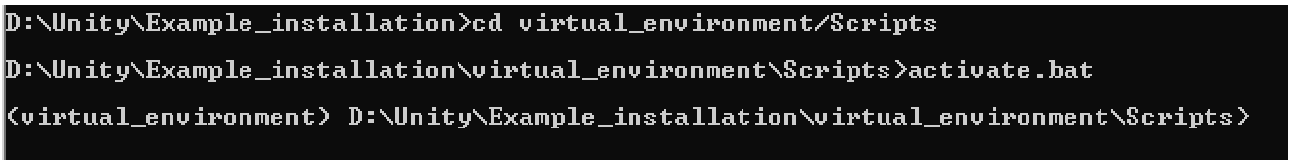
1. Open the command line in the base folder of the Unity project.
2. Create the virtual environment. This command creates a virtual environment with the folder name “virtual_environment”.



```
D:\Unity\Example_installation>python -m venv virtual_environment
```

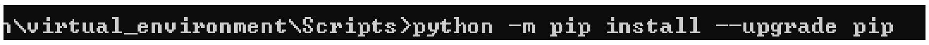
Installing the PyTorch package:

1. Navigate into the Scripts folder from the virtual environment folder and start the activate.bat file. The virtual environment is running if the folder name can be seen in between these ”<>” signs.



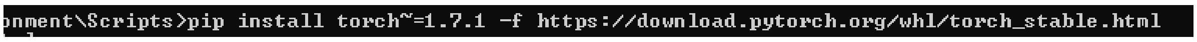
```
D:\Unity\Example_installation>cd virtual_environment/Scripts
D:\Unity\Example_installation\virtual_environment\Scripts>activate.bat
(virtual_environment) D:\Unity\Example_installation\virtual_environment\Scripts>
```

2. (optional) Update the pip installer.



```
n\virtual_environment\Scripts>python -m pip install --upgrade pip
```

3. Install the latest recommended PyTorch package according to the GitHub page.

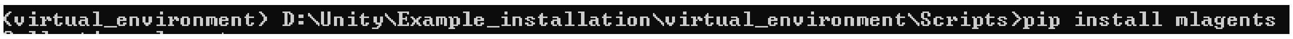


```
onment\Scripts>pip install torch~=1.7.1 -f https://download.pytorch.org/whl/torch_stable.html
```

4.1.3 Python ml-agents package

This package contains the machine learning algorithms that allows the agent to be trained.

1. Be sure the virtual environment is located to and activated.
2. Install the ml-agents package.



```
(virtual_environment) D:\Unity\Example_installation\virtual_environment\Scripts>pip install mlagents
```

3. Check if the installation was succesfull by running the command ”mlagents-learn --help”.

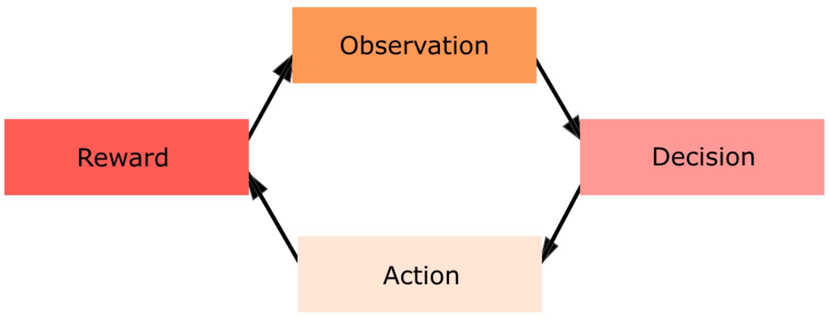
5 Training Agents in Unity

ML-Agents is the toolkit that allows us to implement machine learning in Unity. More specifically ML-Agents can train Agents with the use of reinforcement learning, imitation learning, neuroevolution and more ML methods using Python packages. (ML-Agents, n.d.-d)

5.1 Reinforcement learning in Unity

In chapter 1 this thesis has explored how the MDP and reinforcement cycle works. In Unity these terms are simplified and it is important to understand these terms. This cycle is the basis of how the Agent learns. This cycle is theoretically the same as the one previously covered; this one is simpler to understand because the components in this cycle are components that are used in Unity to train the agent.

Figure 9 Reinforcement learning cycle in Unity



Observation: the agent gathers data from its surrounding environment. The environment in Unity is the scene the agent is in.

Decision: based on the gathered data from the observation the agent makes a decision.

Action: the agent takes an action based on what it decided in the step before.

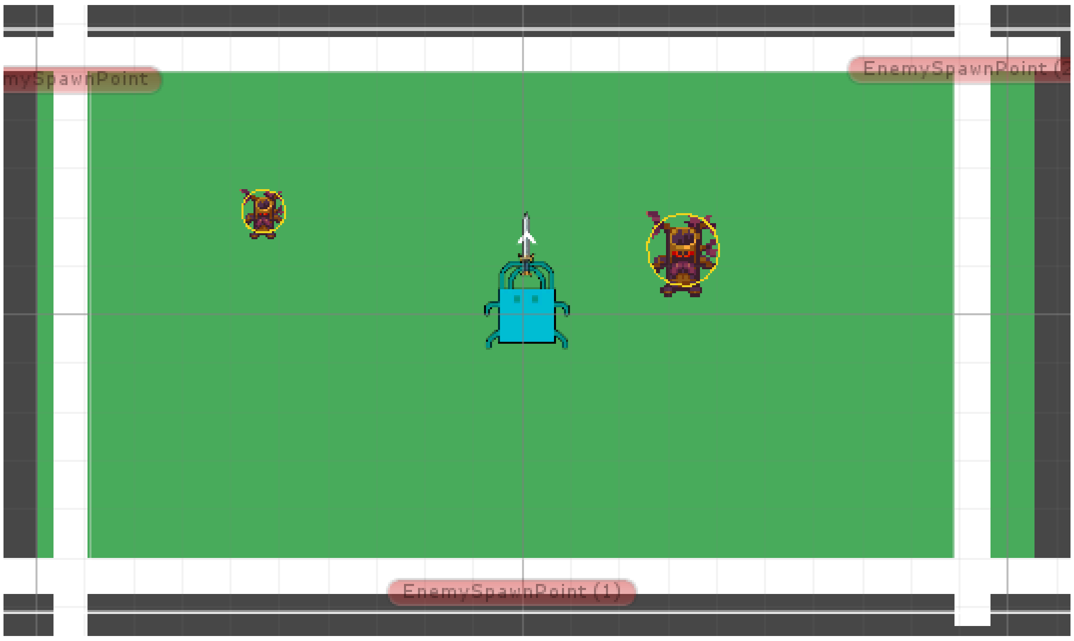
Reward: if the agent made a good action it is rewarded, if it made a poor action it is negatively rewarded.

(ML-Agents, n.d.-a)

5.2 Unity project: Spider Lonky Leg

This is the author’s game: Spider Lonky Leg. This is a very basic game as the creator of this thesis had no previous knowledge of creating a game in Unity, but it is a useful example for clarifying the way machine learning works in Unity. In this game the player controls the spider, the spider can move in four directions and can swing his sword. The goal of the game is to survive the waves of incoming treants. In this game the agent replaces the human player and becomes the trained agent to beat the game. If the spider gets hit too many times, it dies. The treants try to find a path to the spider, this is done through classic AI with the help of A* pathfinding.

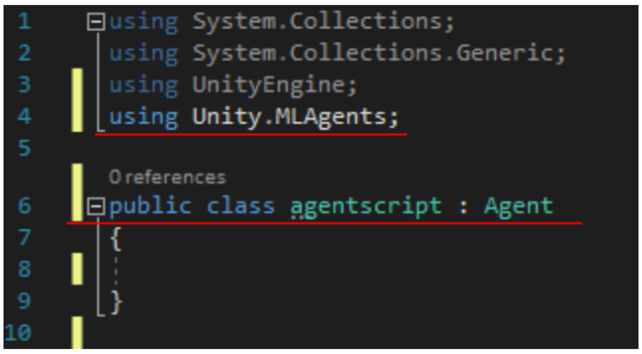
Figure 10 The game’s Unity scene



5.3 Create agent

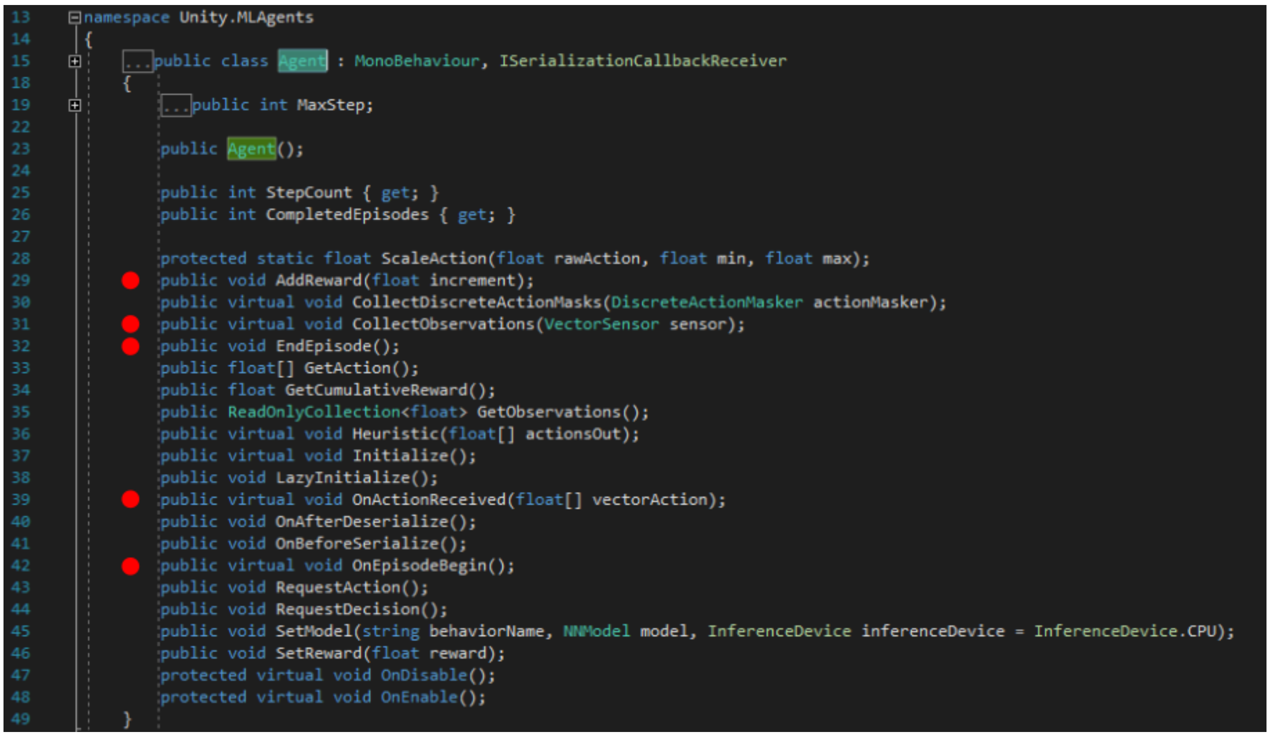
The agent is what is eventually machine learned. The agent can take many forms but usually takes the form of an NPC. In this game the agent is the spider. The first thing needed to create an agent in Unity is to have a game object that becomes the agent, also known as the game character. An agent can be created in Unity by making a new C# script. This script needs to inherit from the “Agent” class instead of the standard “MonoBehaviour”. The functionalities from the ml-agents package can be used by adding the namespace “Unity.MLAgents”. The base functions of this script can be deleted as those are not necessary for machine learning. This script (Figure 11) is the basis of the agent. Some of the functions inherited from the Agent class that are overridden or used in the following sections are shown in Figure 12. When this script is dragged onto the desired game object, Unity automatically adds a behavior parameter script component. This behavior component are the parameters that the agent uses.

Figure 11 C# Agent script



```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.MLAgents;

0 references
public class agentscript : Agent
{

}
```

Figure 12 MLAgents class



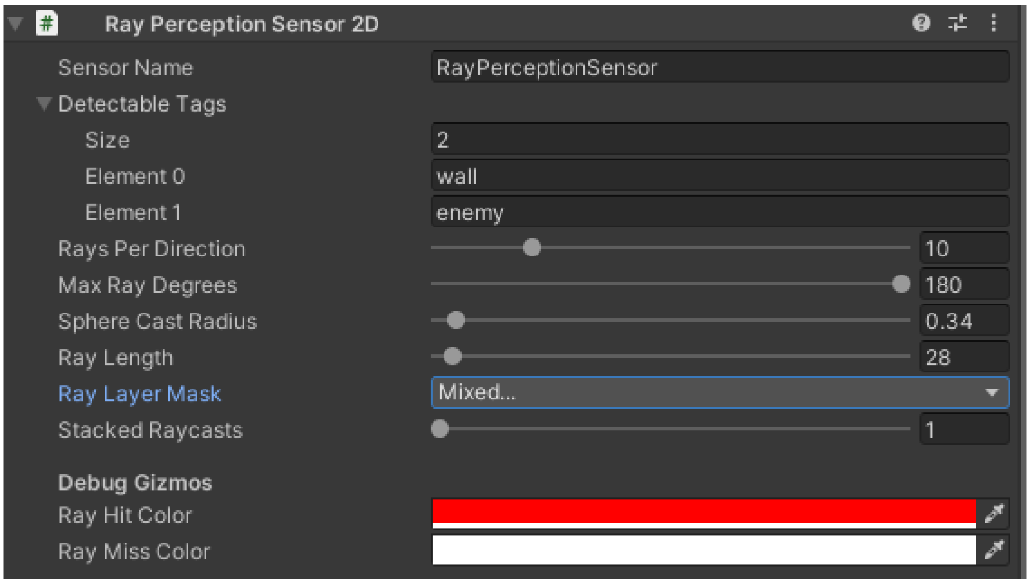
```
namespace Unity.MLAgents
{
    ...public class Agent : MonoBehaviour, ISerializationCallbackReceiver
    {
        ...public int MaxStep;

        public Agent();

        public int StepCount { get; }
        public int CompletedEpisodes { get; }

        protected static float ScaleAction(float rawAction, float min, float max);
        public void AddReward(float increment);
        public virtual void CollectDiscreteActionMasks(DiscreteActionMasker actionMasker);
        public virtual void CollectObservations(VectorSensor sensor);
        public void EndEpisode();
        public float[] GetAction();
        public float GetCumulativeReward();
        public ReadOnlyCollection<float> GetObservations();
        public virtual void Heuristic(float[] actionsOut);
        public virtual void Initialize();
        public void LazyInitialize();
        public virtual void OnActionReceived(float[] vectorAction);
        public void OnAfterDeserialize();
        public void OnBeforeSerialize();
        public virtual void OnEpisodeBegin();
        public void RequestAction();
        public void RequestDecision();
        public void SetModel(string behaviorName, NNModel model, InferenceDevice inferenceDevice = InferenceDevice.CPU);
        public void SetReward(float reward);
        protected virtual void OnDisable();
        protected virtual void OnEnable();
    }
```

5.4 Observation

This is what the agent sees in the game, the eyes of the agent. This is where the agent gets the information about the scene it is in. acquiring observations can be done in a multitude of ways, some requiring more computing power than others. In the base example the agent makes 2 observations. It observes where in the scene the agent itself is and it ray casts several rays to see whether its surrounding environment is empty, a treant or a wall. A ray cast is a line that starts from the objects point of origin and detects whether something is in front of it (“Unity - Scripting API: Physics.Raycast,” n.d.). This can be thought of as shining a light at a reflective surface, if the angle is straight on, the light is reflected on the object back to the agent and the agent knows that there is an object in front of it.

The ML-Agents toolkit has an easy way to implement ray casting to use for machine learning. There is the “Ray Perception Sensor 2D” for 2D game scenes and the “Ray Perception Sensor 3D” for 3D game scenes. This component comes with some parameters that need configuring (Figure 13).

Figure 13 Ray Perception Sensor 2D parameters

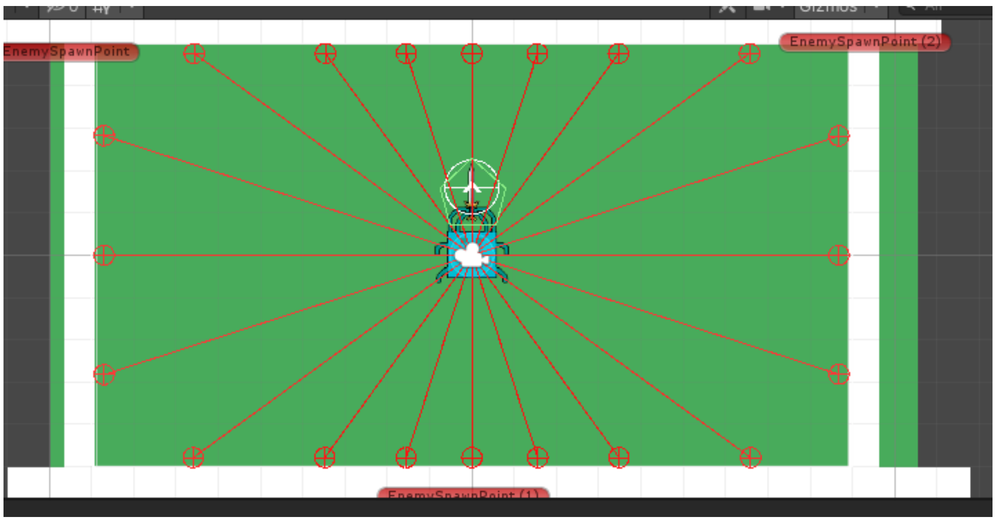


Ray Perception Sensor 2D

Sensor Name	RayPerceptionSensor
Detectable Tags	
Size	2
Element 0	wall
Element 1	enemy
Rays Per Direction	10
Max Ray Degrees	180
Sphere Cast Radius	0.34
Ray Length	28
Ray Layer Mask	Mixed...
Stacked Raycasts	1
Debug Gizmos	
Ray Hit Color	🟥
Ray Miss Color	⬜

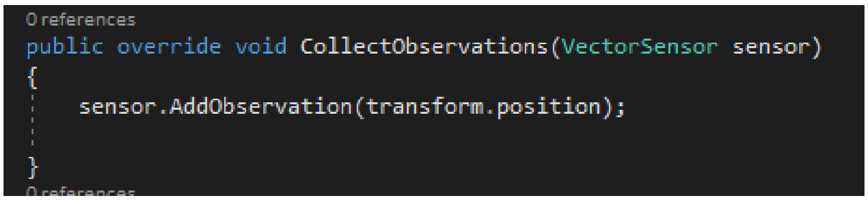
The “Detectable Tags” are what the agent observes. The size parameter indicates how many tags the agent observes. The element parameters indicate the name of the tags that should be detected. “Rays Per Direction” is how many rays are emitted from agent. The more rays per direction the more accurate the observations are, however, the more rays the agent uses the more calculations the algorithm must make. The rays should be limited to the minimum needed for the game's setup. The "Max Ray Degrees" is how far around the agent the rays are. If put on 180 the rays fully surround the agent, 90 only shoots rays in front. The "Sphere Cast Radius" shows the size of the circle at the end of the ray. Picking the radius should be done according to the scenario. Too big and the agent might have undesired detections. If this is put on 0 a true ray cast is acquired. The "Ray Length" is how far the rays travel. "Ray Layer Mask" controls what the rays can hit. The agent needs to be disabled in this setting. If this the agent is left enabled the agent gets no observations because it only hits itself and the rays stop tracking after the first target hit. The "Stacked Raycasts" decides how many ray casts are stacked before being sent to the neural network. (ML-Agents, n.d.-b)

Figure 14 Raycast setup with parameters from Figure 13



The ray casts provide the agent with one of the observations. This allows the agent to know where the walls are and where the enemies are. The other observation is the agent's own position. To process these observations the developer overrides the "CollectObservations" function. This function collects the observations. The rays are automatically processed and do not require any more code, just overriding the function makes these work. The agent's position can be determined by adding its position to the code part of the sensor. Figure 15 shows how this is done in Unity.

Figure 15 Collection the observations and adding own position

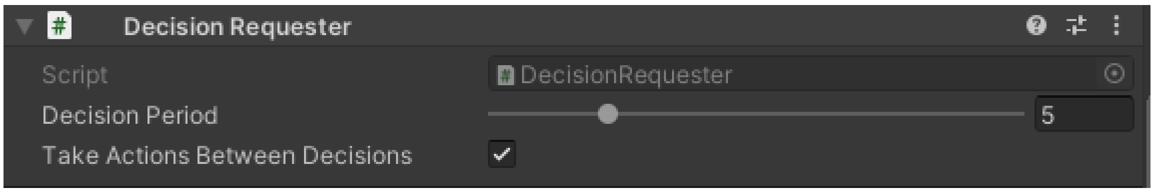


```
0 references
public override void CollectObservations(VectorSensor sensor)
{
    sensor.AddObservation(transform.position);

}
0 references
```

5.5 Decision

There are multiple ways the agent can request decisions but one of the easier ways to request decisions in Unity is to add the “Decision Requester” component. The “Decision Requester” requests a decision every certain number of steps, this is configured through the “Decision Period”, the number given is how many steps the requester waits to make a decision. “Take Actions Between Decisions” checkbox marks whether the agent takes actions in between the steps that no decisions are taken. (ML-Agents, n.d.-b)

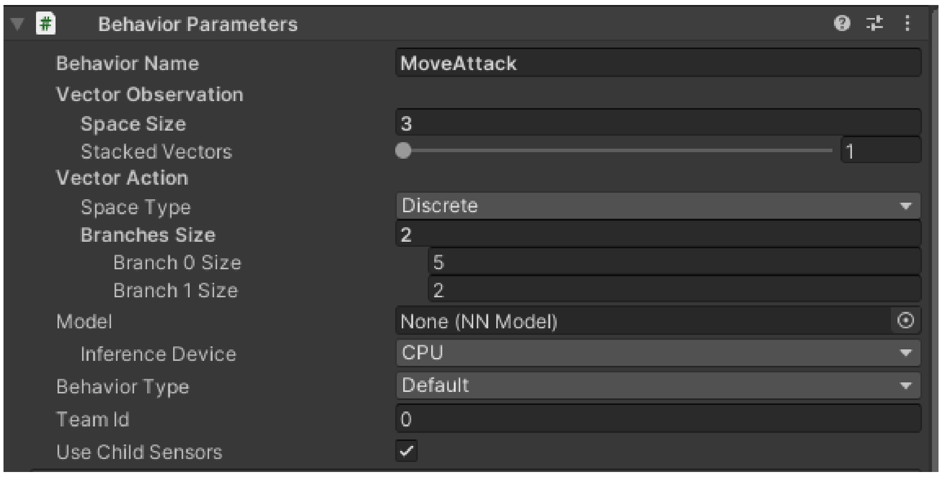
Figure 16 Decision Requester component


Decision Requester
Script: DecisionRequester
Decision Period: 5
Take Actions Between Decisions: ✓

5.6 Action

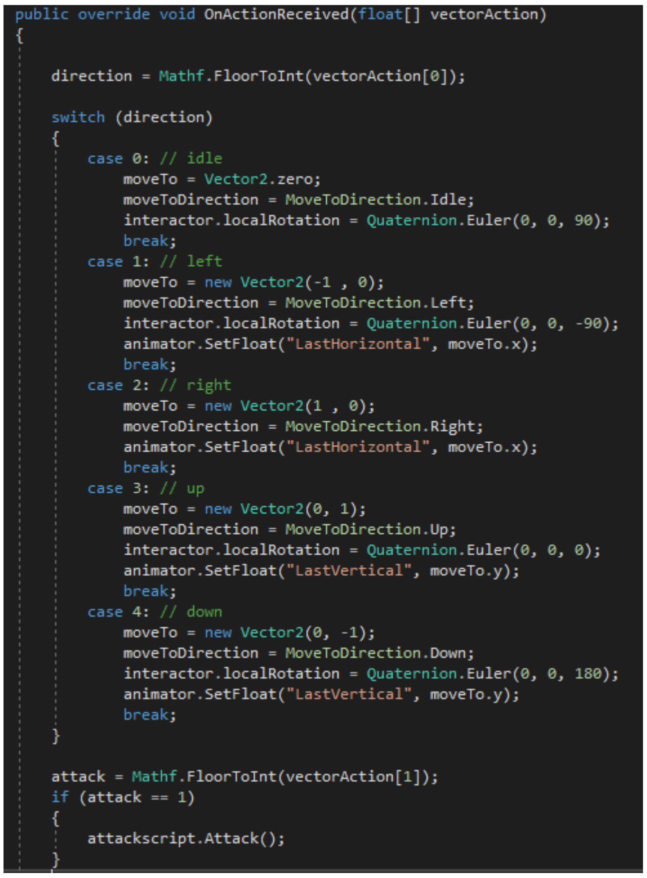
As previously mentioned the agent makes actions based on the decision it made. The agent has seven potential actions, it can go in either of the four directions or hold its position which make up a total of five actions; it can swing its sword, or it can choose to not swing its sword. To understand how to implement the actions in Unity the developer first needs to comprehend the “Behavior Parameters” component that is automatically added when the agent script is dragged on the game object.

Figure 17 Behavior Paremeters of the agent


Behavior Parameters
Behavior Name: MoveAttack
Vector Observation
Space Size: 3
Stacked Vectors: 1
Vector Action
Space Type: Discrete
Branches Size: 2
Branch 0 Size: 5
Branch 1 Size: 2
Model: None (NN Model)
Inference Device: CPU
Behavior Type: Default
Team Id: 0
Use Child Sensors: ✓

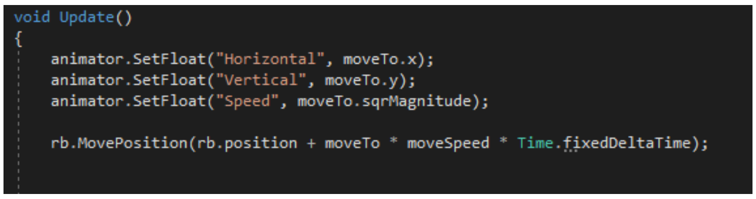
The “Vector Action” parameter is the one focused on here. The “Space Type” parameter can be either discrete or continuous. With a continuous space type, the actions received are floats, numbers between -1 and 1. With a discrete space type whole numbers are received. In the example game the discrete type is chosen. Having a branch size of two means that an array with two indexes is made, 0 and 1. On the first index there are five possible values: 0, 1, 2, 3 or 4. This corresponds with the five moving actions the agent can take (left, right, up, down, idle). On the second index of the branch there are two possible values. These values can be 0 or 1 representing whether the agent attacks or not. It is important to understand that for the agent these numbers do not have any meaning. The code behind these numbers is what make up the actions. (ML-Agents, n.d.-b)

Figure 18 Overriding the OnActionReceived function



```
public override void OnActionReceived(float[] vectorAction)
{

    direction = Mathf.FloorToInt(vectorAction[0]);

    switch (direction)
    {
        case 0: // idle
            moveTo = Vector2.zero;
            moveToDirection = MoveToDirection.Idle;
            interactor.localRotation = Quaternion.Euler(0, 0, 90);
            break;
        case 1: // left
            moveTo = new Vector2(-1 , 0);
            moveToDirection = MoveToDirection.Left;
            interactor.localRotation = Quaternion.Euler(0, 0, -90);
            animator.SetFloat("LastHorizontal", moveTo.x);
            break;
        case 2: // right
            moveTo = new Vector2(1 , 0);
            moveToDirection = MoveToDirection.Right;
            animator.SetFloat("LastHorizontal", moveTo.x);
            break;
        case 3: // up
            moveTo = new Vector2(0, 1);
            moveToDirection = MoveToDirection.Up;
            interactor.localRotation = Quaternion.Euler(0, 0, 0);
            animator.SetFloat("LastVertical", moveTo.y);
            break;
        case 4: // down
            moveTo = new Vector2(0, -1);
            moveToDirection = MoveToDirection.Down;
            interactor.localRotation = Quaternion.Euler(0, 0, 180);
            animator.SetFloat("LastVertical", moveTo.y);
            break;
    }

    attack = Mathf.FloorToInt(vectorAction[1]);
    if (attack == 1)
    {
        attackscript.Attack();
    }
```

Figure 19 Moving the agent in the update function



```
void Update()
{
    animator.SetFloat("Horizontal", moveTo.x);
    animator.SetFloat("Vertical", moveTo.y);
    animator.SetFloat("Speed", moveTo.sqrMagnitude);

    rb.MovePosition(rb.position + moveTo * moveSpeed * Time.fixedDeltaTime);
```

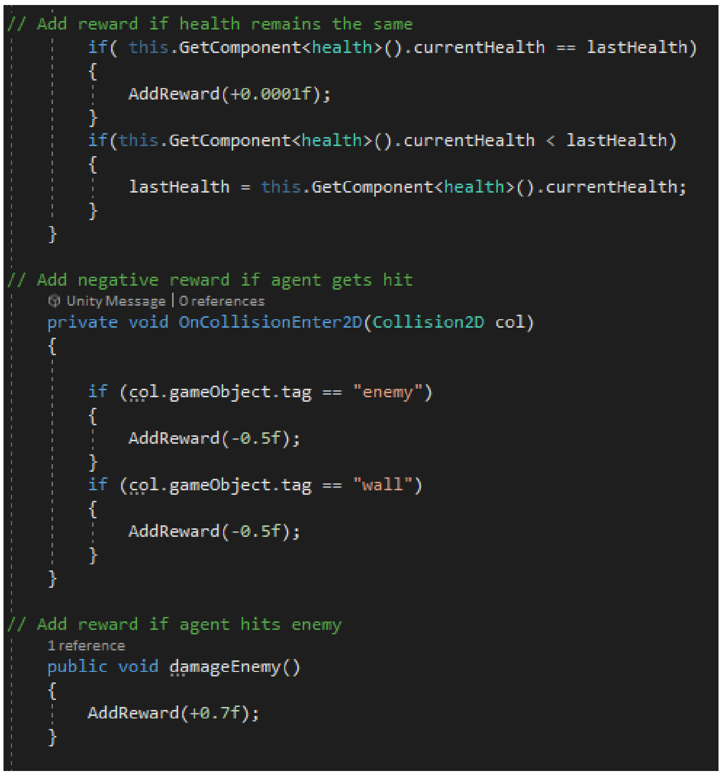
This code (Figure 18) shows how the agent receives the actions and then transfer those numbers into actions the agent makes. Some of the code is to make the animations work, the main components for making the actions are the numbers that are stored in the direction and attack variable. These are the numbers the agent receives from the decision requester. The values that are saved on first branch which is index 0 is the variable direction. For deciding the movement there is the switch statement, in this statement there are five possible values which are to either idle, go left, right, up or down. In each of these cases the “moveTo” vector2 variable is set to the corresponding vector values, the “interactor.localRotation” is to make sure the character points the right directions as well. The position of the agent is changed in the update loop. Here the “moveTo” variable is filled in with the value returned from one of the cases, this is how the agent receives a number on the first branch and takes an action accordingly.
The attack variable is somewhat simpler to process. In this variable the value of the second branch index is stored, index 1. This value is either a 1 or 0, if the agent receives a one then an attack is performed.

5.7 Reward

The rewards are what drive the agent. At the start of the learning experience the agent does not know anything about its environment, it does not know whether an action is good or bad. Based on the rewards the agent knows whether it is improving its ability at playing the game. Finding the best rewards can be challenging, this requires trial and error from the developer. If rewards are chosen poorly then the agent exploits those rewards and potentially does not train the way hoped for. In the example four different rewards were set with a unique purpose for each. (Li, 2018)

The agent receives a positive Reward for staying alive. Each update the agent checks whether it has lost hit points or whether its health bar has remained the same. A small reward is given if the agent did not lose health. The developer does not want to give too high of a reward because the update function is called every frame. This reward should incentive the agent to remain at maximum health points. The agent receives a negative reward if it gets hit by an enemy or touches the wall. This overlaps partly with the previous reward of staying alive but here the agent gets punished for touching an enemy. The agent also gets a negative reward if it touches the walls, so the agent stays more in the center where the agent has more beneficial positions. The agent receives a positive reward each time it damages an enemy. This should make the agent focus on clearing all enemies which would improve its survival rate.

Figure 20 Adding rewards for the agent by using the AddReward function



```
// Add reward if health remains the same
        if( this.GetComponent<health>().currentHealth == lastHealth)
        {
            AddReward(+0.0001f);
        }
        if(this.GetComponent<health>().currentHealth < lastHealth)
        {
            lastHealth = this.GetComponent<health>().currentHealth;
        }
    }

// Add negative reward if agent gets hit
    Unity Message | 0 references
    private void OnCollisionEnter2D(Collision2D col)
    {

        if (col.gameObject.tag == "enemy")
        {
            AddReward(-0.5f);
        }
        if (col.gameObject.tag == "wall")
        {
            AddReward(-0.5f);
        }
    }

// Add reward if agent hits enemy
    1 reference
    public void damageEnemy()
    {
        AddReward(+0.7f);
    }
```

Implementing these rewards can be done by using the “AddReward” function from the “MLAgent“ class. “SetReward” can also be used if it is desired to set the reward at a certain level. This can be done at the end of the episode when the agent has successfully completed the tasks it was meant to do. Functions are created and called in other scripts which keeps the code clean.

5.8 Training the agent

When all the four steps of the reinforcement cycle are set up then it is time to train the agent. The agent uses the Proximal Policy Optimization (PPO) policy and certain parameters if all settings are left as they are (ML-Agents, n.d.-e). To start out the training it is ok to use the standard setup. If maximum efficiency of training is desired changing some of the hyperparameters can be optimal. To start the training, the developer first opens the virtual python environment. In this environment the command “mlagents-learn --run-id=trainagent” is run. This starts the training after the play button in Unity is pressed.

5.9 Using the learned brain

When the training is stopped a usable, trained brain is created. This brain can be found in the results folder under the folder with the name of the run id. In that folder there is the ".onnx" file, this is the neural network model. Use this trained brain by adding it to the behavior parameters at the model parameter. Selecting "Inference Only" as the behavior type makes the agent use the trained brain instead of trying to learn.

5.10 TensorBoard

TensorBoard is a tool that visualizes the learning experience of the agent (ML-Agents, n.d.-f). To use TensorBoard, open the virtual environment and type in the command "tensorboard --logdir results". The TensorBoard results run on the localhost 6006. This page contains many different charts that help understand and improve the learning rate of the agent. Two of the more important graphs are the cumulative reward and the episode length graph. The cumulative reward graph shows the developer the reward the agent is getting at each point in time. An upwards trend shows that the agent is earning more rewards and thus is improving at the game. The second graph is the episode length. In the example game the episode length should increase as time goes on. An increase in episode length means that the agent survives longer. In some use cases the episode length should go down, meaning the agent is performing the tasks quicker. (ML-Agents, n.d.-f)

5.11 Optimizing the learning process

After the training is set up and the results the agent is achieving are understood, it is time to start improving the learning process. These are some methods that improve the learning success of the agent.

- Creating multiple agents
- Changing the rewards given
- Incremental learning
- Randomization
- Changing the training parameters

5.11.1 Creating multiple agents

One of the fastest ways to speed up the training process is multiplying the numbers of agents trained. In Unity, one can simply copy the scene they have and make sure all agents are using the same brain, the learning algorithm then combines the input of all these agents and learn at a far quicker rate. However, this does not improve the training. Any flaws in the environment or parameters are not improved. (ML-Agents, n.d.-c)

5.11.2 Changing the rewards given

It is important to analyze well what goals the agent needs to achieve and proper rewards need to set in order to achieve those goals. The rewards are what drives the agent, if these are incorrect the training process might be longer than needed. All rewards should have a reason and the logic behind the rewards needs to be implemented correctly. It is important that the agent is receiving the rewards at the right times, running the agent with non-optimized rewards results in a non-optimized learning experience. Sometimes changing the amount of points a reward give or changing the reward criteria can make a big change. It is recommended to minimize negative rewards as these are proven to be less effective at making the agent learn efficiently. (ML-Agents, n.d.-b)

5.11.3 Incremental learning

Humans learn how to walk or solve complex equations step by step, it all starts with the basics. When humans learn mathematics they start with the easy to learn material like addition and subtraction, they do not immediately start learning about trigonometry and limits. The same can be applied in machine learning. Start out the environment easy and add more challenging task later, incrementally increase the difficulty of the environment. This process is also known as curriculum learning. (Hacohen & Weinshall, 2019)

5.11.4 Randomization

Machine learned agents like to find patterns in things, if it finds one working tactic it uses that tactic repeatedly (Cobbe, Klimov, Hesse, Kim, & Schulman, 2019). In adaptive game environments this is un-desired because the developer wants the agent to be able to succeed in many different scenarios and environments. Therefore, it is important to add randomization to the training scene or randomize the agent itself. The whole learning process can be changed by for example alternating the starting position of the agent or enemies each new episode. That said, randomization parameters should only have relatively small impacts. Do not change everything drastically each episode as this could slow the agents learning capabilities down, or even make the agent unable to gain any rewards over a long period. (Lee, Lee, Shin, & Lee, 2020)

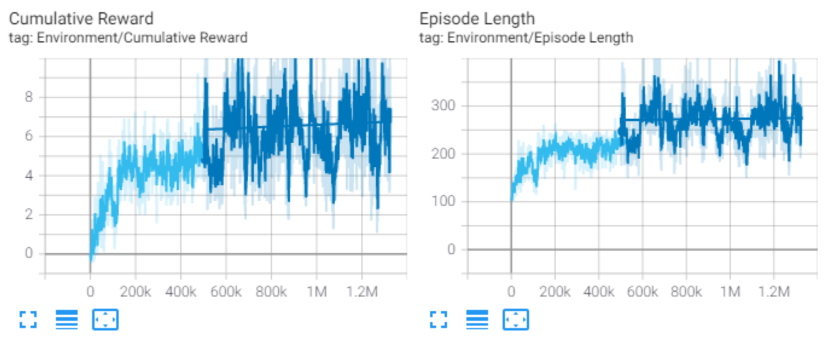
5.11.5 Changing the training parameters

There are quite many parameters that the learning algorithm has. Changing these parameters requires the developer to have deep understanding of what those parameters do. It is recommended to only change the standard parameters once the developer has a good grasp on the mathematics behind the algorithms. Changing the hyperparameters can speed up the training but might lead to reduced generalization. (Cobbe et al., 2019)

5.12 Training results

Analyzing what the agent has achieved in a four-hour training period with the basic settings, a couple of findings can be concluded.

Figure 21 TensorBoard cumulative reward and episode length



The agent has improved its ability to play the game. The cumulative reward graph in Figure 21 shows that the agent achieves a higher cumulative reward as time progresses. At the start the cumulative reward rises faster because the level is easier, after five hundred thousand steps the agent struggles to get a consistently higher reward level. This is because the level spikes in difficulty, if the agent would complete the second phase of the game the cumulative reward would rise significantly. The Episode length graph shows that as the reward increases so does the episode length, in this case it is to be expected as some the rewards are based on how long the agent survives.

In this example game the author decided to implement gradual difficulty scaling from the very start. The game starts out by sending only two enemies at a time, if those are defeated the game sends out four more, if those are defeated the game sends out eight enemies. When the agent manages to beat all three stages the game cycles back to the beginning. This would cause the agent to receive an exponential growth in rewards once it manages to beat one cycle.

6 Results

This thesis aimed to find whether or not machine learning is ready to be implemented in video games. The practical part showed that it is possible to have machine learned agents in video games. Once the developer has a good understanding of machine learning, the implementation using ml-agents is rather straightforward. The theoretical framework highlights some of the techniques used in present use cases of machine learning in video games. Since the development of both machine learning and video games is an ever growing and changing subject, there will likely be new machine learning techniques used in video games in the future.

Unity and the ML-Agents toolkit allows individual creators to experiment and implement reinforcement learning and other machine learning techniques in their game development process. As explained in the theoretical framework, having a machine learned agent is not the only use case. There are many use cases for machine learning inside and outside of video game development. The industry will continue to find new uses for machine learning as time goes on.

The results from the example game show that reinforcement learning is an effective tool that is readily available to be implemented in video game development. Agents have the capability of learning and adapting to changing environments. Having well thought out rewards and an optimized environment will speed up the learning process. The practical part only covered deep reinforcement learning, there are more learning techniques to be researched and used.

The game made to implement machine learning is a 2D top down game, the results drawn from this game are therefore applicable for this genre of game. Because of the broad range of video game genres these results should only be used for this type of game. This does not mean that machine learning is only applicable in this genre but that other genres might have different use cases which are not covered in this thesis.

Some parts of the thesis require more background information about different subjects. The author could not fully cover the following topics:

- The OpenAI Five project
- The intricacies of how Dota 2 works

- Optimizing the learning process

To conclude, machine learning in video games development is already being researched to a fair extent. Research will continue to improve and find new ways to implement machine learning in video games. In the near future machine learning will be a technology just as common as AI.

7 Summary

Throughout the thesis the reader will find an answer to all three research questions. The first and second part of the theoretical framework answers the first question of what exactly machine learning is, the third part of the theoretical framework provides an answer to what parts of video game development could be replaced or improved upon by machine learning. The practical part of the thesis shows how the implementation of machine learning can be done using ML-Agents and Unity.

Researching and implementing machine learning in a video game gave the author insight in the current situation and the future of machine learning. The ML-Agents toolkit is a great way for developers who are new to machine learning learn the advantages and disadvantages of reinforcement learning.

This thesis covered the basics of machine learning and how some of the machine learning techniques work. This was a difficult challenge and there is still a lot to be learned. This thesis does not cover how the mathematics and algorithms work in depth. Understanding the mathematics behind the algorithms and techniques would be a different subject and thesis on its own for someone with a more mathematical background.

This thesis gave the author the opportunity to research and develop his knowledge of machine learning as a whole. It was a great experience to be able to combine my passion for playing games and my studies as an ICT student. Making a game for the first time was challenging and while there certainly are some lacking parts about video game development in this project, it was great to learn this new skill as well. This thesis also improved the author's ability to research and write a thesis in the English language with correct citations.

References

3Blue1Brown. (2017). But what is a Neural Network? | Deep learning, chapter 1 [Video file]. Retrieved from https://www.youtube.com/watch?v=aircAruvnKk&t=527s&ab_channel=3Blue1Brown

Aponte, M.-V., Levieux, G., & Natkin, S. (2009). *Scaling the Level of Difficulty in Single Player Video Games*.

Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., … Brain, G. (2020). *EMERGENT TOOL USE FROM MULTI-AGENT AUTOCURRICULA*.

Bellman, R. (1954). *The theory of dynamic programming*.

Berner, C., Brockman, G., Chan, B., Cheung, V., Dennison, C., Farhi, D., … Zhang, S. (2019). *Dota 2 with Large Scale Deep Reinforcement Learning*. Retrieved from https://arxiv.org/pdf/1912.06680.pdf

Bhandari, P. (2020). What is Qualitative Research? | Methods & Examples. Retrieved April 19, 2021, from Scribbr website: https://www.scribbr.com/methodology/qualitative-research/

Blackburn. (2019). Reinforcement Learning : Markov-Decision Process (Part 1). Retrieved February 17, 2021, from towards data science website: https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da

Cobbe, K., Klimov, O., Hesse, C., Kim, T., & Schulman, J. (2019). *Quantifying Generalization in Reinforcement Learning*.

Dormehl, L. (2019). What is an artificial neural network? Here's everything you need to know | Digital Trends. Retrieved February 25, 2021, from digitaltrends website: https://www.digitaltrends.com/cool-tech/what-is-an-artificial-neural-network/

Expected Return - What Drives a Reinforcement Learning Agent in an MDP - deeplizard. (2018). Retrieved February 19, 2021, from Deeplizard website: https://deeplizard.com/learn/video/a-SnJtmBtyA

Exploration vs. Exploitation - Learning the Optimal Reinforcement Learning Policy - deeplizard. (2018). Retrieved February 19, 2021, from Deeplizard website:

https://deeplizard.com/learn/video/mo96Nqlo1L8

Foote, K. D. (2019). A Brief History of Machine Learning. Retrieved March 10, 2021, from Dataversity website: https://www.dataversity.net/a-brief-history-of-machine-learning/

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., Pineau, J., Brain, G., & -Delft, B. (2018). *An Introduction to Deep Reinforcement Learning*. https://doi.org/10.1561/2200000071

Garbade, D. M. J. (2018). Clearing the Confusion: AI vs Machine Learning vs Deep Learning Differences. *Medium*. Retrieved from https://towardsdatascience.com/clearing-the-confusion-ai-vs-machine-learning-vs-deep-learning-differences-fce69b21d5eb

Geek, S. the. (2018). Bellman Equation Basics for Reinforcement Learning [Video file]. Retrieved from https://www.youtube.com/watch?v=14BfO5lMiuk&ab_channel=SkowstertheGeek

Geisler, B. (2004). *Integrated Machine Learning For Behavior Modeling in Video Games*.

Grosse, R. (2018). Generalization. Retrieved March 4, 2021, from https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/readings/L09 Generalization.pdf

Hacohen, G., & Weinshall, D. (2019). *On The Power of Curriculum Learning in Training Deep Networks*.

Henrik, W. (2015). *Towards an updated typologo of non-player character roles*. Retrieved from https://www.researchgate.net/publication/295079901_Towards_an_updated_typology_of_non-player_character_roles

IBM Coud Education. (2020). What is Machine Learning? Retrieved February 25, 2021, from IBM website: https://www.ibm.com/cloud/learn/machine-learning

Iriondo, R. (2018). Machine Learning (ML) vs. Artificial Intelligence (AI) — Crucial Differences. *Towards AI*. Retrieved from https://pub.towardsai.net/differences-between-ai-and-machine-learning-and-why-it-matters-1255b182fc6

Kapoor, S. (2018). Policy Gradients in a Nutshell. Everything you need to know to get… | by Sanyam Kapoor | Towards Data Science. Retrieved March 2, 2021, from towards data science website: https://towardsdatascience.com/policy-gradients-in-a-nutshell-8b72f9743c5d

Lee, K., Lee, K., Shin, J., & Lee, H. (2020). *NETWORK RANDOMIZATION: A SIMPLE TECHNIQUE FOR GENERALIZATION IN DEEP REINFORCEMENT LEARNING*.

Li, Y. (2018). *DEEP REINFORCEMENT LEARNING*.

LING, C. G. (2020). *Graphical Glitch Detection in Video Games Using CNNs*. Retrieved from https://media.contentapi.ea.com/content/dam/ea/seed/presentations/garcialing2020-graphical-glitch-detection-in-video-games-using-cnns.pdf

Markov Decision Processes (MDPs) - Structuring a Reinforcement Learning Problem - deeplizard. (2018). Retrieved February 19, 2021, from Deeplizard website: https://deeplizard.com/learn/video/my207WNoeyA

ML-Agents. (n.d.-a). ml-agents/Learning-Environment-Create-New.md at main · Unity-Technologies/ml-agents · GitHub. Retrieved March 22, 2021, from GitHub website: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Create-New.md

ML-Agents. (n.d.-b). ml-agents/Learning-Environment-Design-Agents.md at main · Unity-Technologies/ml-agents · GitHub. Retrieved March 22, 2021, from GitHub website: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design-Agents.md#rewards

ML-Agents. (n.d.-c). ml-agents/Learning-Environment-Design.md at main · Unity-Technologies/ml-agents · GitHub. Retrieved March 22, 2021, from GitHub website: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design.md

ML-Agents. (n.d.-d). ml-agents/ML-Agents-Overview.md at main · Unity-Technologies/ml-agents · GitHub. Retrieved March 22, 2021, from GitHub website: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md

ML-Agents. (n.d.-e). ml-agents/Training-Configuration-File.md at main · Unity-Technologies/ml-agents · GitHub. Retrieved March 22, 2021, from GitHub website: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Training-Configuration-File.md

ML-Agents. (n.d.-f). ml-agents/Using-Tensorboard.md at main · Unity-Technologies/ml-agents · GitHub. Retrieved March 22, 2021, from GitHub website: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Using-Tensorboard.md

ML-Agents. (n.d.-g). Unity-Technologies/ml-agents: Unity Machine Learning Agents Toolkit. Retrieved March 22, 2021, from GitHub website: https://github.com/Unity-Technologies/ml-agents

Parikh, D. (2018). Learning Paradigms in Machine Learning. Retrieved February 11, 2021, from Data Driven Investor website: https://medium.datadriveninvestor.com/learning-paradigms-in-machine-learning-146ebf8b5943

Penelope Sweetser, J. W. (2002). *Current AI in Games: A Review*.

Policies and Value Functions - Good Actions for a Reinforcement Learning Agent - deeplizard. (2018). Retrieved February 19, 2021, from Deeplizard website: https://deeplizard.com/learn/video/eMxOGwbdqKY

PyTorch. (2019). PyTorch documentation — PyTorch 1.8.1 documentation. Retrieved April 17, 2021, from PyTorch website: https://pytorch.org/docs/stable/index.html

Q-Learning Explained - A Reinforcement Learning Technique - deeplizard. (2018). Retrieved February 19, 2021, from Deeplizard website: https://deeplizard.com/learn/video/qhRNvCVVJaA

Richard S. Sutton, A. G. B. (2018). Reinforcement Learning, second edition: An Introduction. Retrieved March 22, 2021, from Google Books website: https://books.google.fi/books?hl=nl&lr=&id=uWV0DwAAQBAJ&oi=fnd&pg=PR7&dq=reinforcement+learning&ots=mioKq723h5&sig=TyY4uXSiuUuDjetwcvri7urTOFA&redir_esc=y#v=onepage&q=reinforcement learning&f=false

Safadi, F., Fonteneau, R., & Ernst, D. (2015). Artificial intelligence in video games: Towards a unified framework. *International Journal of Computer Games Technology*, *2015*. https://doi.org/10.1155/2015/271296

Simplilearn. (2019). Neural Network In 5 Minutes | What Is A Neural Network? | How Neural

Networks Work | Simplilearn [Video file]. Retrieved from https://www.youtube.com/watch?v=bfmFfD2RIcg&ab_channel=Simplilearn

Spinning up. (2018). Part 2: Kinds of RL Algorithms. Retrieved March 2, 2021, from OpenAi website: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

Statt, N. (2019). How AI will revolutionize the way video games are developed and played. Retrieved February 11, 2021, from The Verge website: https://www.theverge.com/2019/3/6/18222203/video-game-ai-future-procedural-generation-deep-learning

Summerville, A., Snodgrass, S., Guzdial, M., Holmgård, C., Hoover, A. K., Isaksen, A., … Togelius, J. (2018). *Procedural Content Generation via Machine Learning (PCGML)*.

Trevor Hastie, Robert Tibshirani, J. F. (2009). Overview of supervised learning. In *Springer*. Retrieved from https://link.springer.com/content/pdf/10.1007/978-0-387-84858-7_2.pdf

Unity - Scripting API: Physics.Raycast. (n.d.). Retrieved March 22, 2021, from Unity website: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

Zheng, Y., Xie, X., Su, T., Ma, L., Hao, J., Meng, Z., … Fan, C. (2019). *Wuji: Automatic Online Combat Game Testing Using Evolutionary Deep Reinforcement Learning*. Retrieved from https://www.neteasegames.com/

Zohaib, M. (2018). Dynamic difficulty adjustment (DDA) in computer games: A review. https://doi.org/10.1155/2018/5681652

List of figures

Annex 1: C# agentscript

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Unity.MLAgents;
using Unity.MLAgents.Sensors;

public class agentmoveattack : Agent
{
    public Rigidbody2D rb;
    public Transform interactor;
    public Animator animator;

    public float moveSpeed = 5f;
    private Vector2 moveTo = Vector2.zero;
    private int direction = 0;
    private int attack = 0;

    private int maxHealth = 0;
    private int lastHealth = 0;
    private playerAttack attackscript;

    public enum MoveToDirection
    {
        Idle,
        Left,
        Right,
        Up,
        Down
    }

    public void Awake()
    {
        maxHealth = this.GetComponent<health>().maxHealth;
        lastHealth = maxHealth;
        attackscript = this.GetComponent<playerAttack>();
    }

    public override void OnEpisodeBegin()
    {
        maxHealth = this.GetComponent<health>().maxHealth;
        lastHealth = maxHealth;
        transform.localPosition = new Vector2(2.29f,3.12f);
    }


    private MoveToDirection moveToDirection = MoveToDirection.Idle;

    public override void CollectObservations(VectorSensor sensor)
    {
        sensor.AddObservation(transform.localPosition);
    }
    public override void OnActionReceived(float[] vectorAction)
    {
        direction = Mathf.FloorToInt(vectorAction[0]);
        switch (direction)
        {
            case 0: // idle
                moveTo = Vector2.zero;
                moveToDirection = MoveToDirection.Idle;

                break;
            case 1: // left
                moveTo = new Vector2(-1 , 0);
                moveToDirection = MoveToDirection.Left;
                //interactor.localRotation = Quaternion.Euler(0, 0, 90);
                //animator.SetFloat("LastHorizontal", moveTo.x);

                break;
            case 2: // right
                moveTo = new Vector2(1 , 0);
                moveToDirection = MoveToDirection.Right;
                //interactor.localRotation = Quaternion.Euler(0, 0, -90);
                //animator.SetFloat("LastHorizontal", moveTo.x);

                break;
            case 3: // up
                moveTo = new Vector2(0, 1);
                moveToDirection = MoveToDirection.Up;
                //interactor.localRotation = Quaternion.Euler(0, 0, 0);
                //animator.SetFloat("LastVertical", moveTo.y);

                break;
            case 4: // down
                moveTo = new Vector2(0, -1);
                moveToDirection = MoveToDirection.Down;
                //interactor.localRotation = Quaternion.Euler(0, 0, 180);
                //animator.SetFloat("LastVertical", moveTo.y);
                break;
        }

        attack = Mathf.FloorToInt(vectorAction[1]);
        if (attack == 1)
        {
            attackscript.Attack();
        }
    }


    void Update()
    {
        animator.SetFloat("Horizontal", moveTo.x);
        animator.SetFloat("Vertical", moveTo.y);
        animator.SetFloat("Speed", moveTo.sqrMagnitude);

        rb.MovePosition(rb.position + moveTo * moveSpeed *
Time.fixedDeltaTime);

        if (moveTo.x == 1 || moveTo.x == -1 || moveTo.y == 1 || moveTo.y ==
-1)
        {
            animator.SetFloat("LastHorizontal", moveTo.x);
            animator.SetFloat("LastVertical", moveTo.y);
        }
        switch (moveToDirection)
        {
            case MoveToDirection.Idle:
                break;
            case MoveToDirection.Left:
                interactor.localRotation = Quaternion.Euler(0, 0, 90);
                break;
            case MoveToDirection.Right:
                interactor.localRotation = Quaternion.Euler(0, 0, -90);
                break;
            case MoveToDirection.Up:
                interactor.localRotation = Quaternion.Euler(0, 0, 0);
                break;
            case MoveToDirection.Down:
                interactor.localRotation = Quaternion.Euler(0, 0, 180);
                break;
            default:
                break;
        }

        // Add reward if health remains the same
        if ( this.GetComponent<health>().currentHealth == lastHealth)
        {
            AddReward(+0.0001f);
        }
        if(this.GetComponent<health>().currentHealth < lastHealth)
        {
            lastHealth = this.GetComponent<health>().currentHealth;
        }
    }

// Add negative reward if agent gets hit
    private void OnCollisionEnter2D(Collision2D col)
    {
        if (col.gameObject.tag == "enemy")
        {
            AddReward(-0.5f);
        }
        if (col.gameObject.tag == "wall")
        {
            AddReward(-0.5f);
        }
    }

// Add reward if agent hits enemy
    public void damageEnemy()
    {
        AddReward(+0.7f);
    }

}
```