Reinforcement Learning in Gaming

Seifeldien Soliman

Relatório de Projeto apresentado à Escola Superior de Tecnologia e Gestão para obtenção do Grau de Mestre em Informática

Trabalho realizado sob a orientação de:

Professor Rui Pedro Lopes

Bragança

October 2022

Acknowledgments

I would like to express my deepest appreciation to the special people in my life who have contributed to the enhancement of my life and who have supported me in daily basics with their unparallel love.

First of all, I am immensely grateful to my supervisor Professor Rui Lopes for the continuous support and encouragement to complete my project with the correct guidelines.

I am also grateful to all the professors of Instituto Politécnico de Bragança from the Masters in Informatics for guiding and supporting me starting from my first year to the last one.

I would like to thank my family for the countless times they have helped me during my study journey as well as all my colleagues, especially Isabela Sá, Carmen Silva and Srison Kadariya for their care and support. Thank you.

Resumo

A inteligência artificial em videojogos é uma área de investigação de longa data. É um conceito importante em muitos jogos e estuda como utilizar tecnologias de IA para alcançar o desempenho a nível humano durante o jogo. No entanto, quando se trata de IA e videojogos, a Reinforcement Learning tem de ser mencionada. RL define os agentes que enfrentam os problemas que aprendem a tomar boas decisões apenas através da acção e observação.

Este projecto centra-se na integração de um algoritmo de Machine Learning chamado Reinforcement Learning no desenvolvimento de um videojogo do género Tower Defense. O projeto foi desenvolvido pelo motor Unity3D que incorpora um agente que utiliza a técnica RL para simular o comportamento de um jogador humano e continuar a melhorá-lo, com base em experiências de jogo anteriores, até ser totalmente optimizado com uma pontuação imbatível pelo jogador médio.

O agente irá imitar o comportamento de um humano, comprando, actualizando e colocando torres enquanto obtém a pontuação mais alta, utilizando o menor número de moedas. Além disso, o relatório irá também rever vários conceitos de Aprendizagem Automática, incluindo o Processo de Decisão de Markov e o Q-Learning.

Palavras-chave: "defesa da torre", "inteligência artificial", "reinforcement learning", "machine learning", "redes neuronais".

Abstract

Artificial intelligence in video games is a longstanding research area. It is a major concept in a lot of games and it studies how to use AI technologies to achieve human-level performance when playing games. However, when it comes to AI and video games, Reinforcement Learning has to be mentioned. RL defines the problem-facing agents that learn to make good decisions through action and observation alone.

This project focuses on integrating a Machine Learning algorithm called Reinforcement Learning in the development of a video game of the Tower Defense genre developed by the Unity3D engine that incorporates an agent that uses the RL technique to simulate the behavior of a human player and keep on improving it, based on previous game experiences, until it's fully optimized with a score unbeatable by the average player.

The agent will imitate the behavior of a human, buying, upgrading, and placing towers while getting the highest score by using the lowest number of currencies. Moreover, the report will also review several Machine Learning concepts, including Markov-Decision Process and Q-Learning.

Keywords: "tower defense", "artificial intelligence", "reinforcement learning", "machine learning", "neural networks".

Table of contents

List of figures

List of figures

List of tables

Chapter 1: Introduction

1.1. Technical Proposal

Video game development has evolved significantly from the early days of computer games and the first versions of Nintendo and Atari, with the main goal being to provide entertainment for children and adults alike. Nowadays video games have become more lifelike than ever, leaving the days of pixelated screens and limited sounds as a distant memory. This made digital gaming so appealing that it became a temporary escape from the pressures of the real world for many, but the question is what exactly helped game development rise that much?

While the easy answer could be the evolution of video game graphics, gameplay, or even storylines, it's something else, every gamer looks for when a new video game is released on the market and that is how real-life alike is the artificial intelligence in that game.

Artificial intelligence (AI) in video games is a longstanding research area. It is a major concept in a lot of games and it studies how to use AI technologies to achieve human-level performance when playing games. These AI-powered interactive experiences are usually generated via non-player characters (NPCs) or even enemies, that act intelligently or creatively, as if controlled by a human game-player or were acting with a mind of their own, and without it would be hard for a game to provide an immersive experience to the player.

The proposed Final Degree Project showcase the usage of a Machine Learning algorithm called Reinforcement Learning in the development of a video game of the Tower Defense genre developed by the Unity3D engine that incorporates an agent that uses the RL technique to simulate the behavior of a human player and keep on improving it, based on previous game experiences, until it's fully optimized with a score unbeatable by the average play.

1.2. Project Motivation

Since the development of the game Nim in 1951, one of the first examples of AI in computerized games, machine learning has been integrated into games as a way to challenge the player; however, several times the AI has been based on rules that often

become predictable, which then leads to the player's loss of desire to continue playing as he would learn the AI's approaches and reactions to the possible actions outdone by the player.

This led me to look at a specific machine learning technique called Reinforcement learning, which takes a suitable action to maximize reward in a particular situation by learning from its previous experience, which makes sure to get rid of the predictability since it has no training dataset.

However, while it would be optimal for the agent to fully behave like a human, it would take the fun out of the games if the agents tend to usually outsmart the players. We want the agents to be as smart as it is necessary to provide fun and engagement, but not exceed the limit. A perfect agent should be imperfect, imitating a human-like behavior while still providing entertainment.

1.3. The game

The game chosen to demonstrate the work is a single-player Tower Defence game [16] developed using the Unity 3D engine [21]. To best exhibit the Reinforcement Learning techniques, a strategy-based game was chosen where the player's target is to defend an end area from the opposition by building towers that impede the opponent's movement.

The player starts the game with a currency that can be used to buy and upgrade different types of towers. There are several types of towers to choose from, with varying of different costs and abilities. The towers can be placed in specific tiles and upon that can be upgraded up to three levels, given that the player has enough currency to do so.

The game is won after all 10 waves of enemies are destroyed by the towers and the player's final health has not to be depleted by the enemies.

1.4. Objective

The main objective is to develop an agent that uses Reinforcement Learning to imitate the behavior of a human, buying, upgrading, and placing towers while getting the highest score by using the lowest number of currencies. This will come as a result of previous experiences which the agent will learn from other games while applying some of the Reinforcement Learning techniques. The agent must always detect the best action given a game status.

1.5. Target

The game is a single-player Tower defence game that is suitable for all ages of causal gamers who are looking for a light game to pass the time while still having a

competitive and quick reaction edge in it. The game might be very difficult at the latter level, which adds the fun element as well.

1.6. Expected results

The main objective of the work is to integrate the Reinforcement Learning algorithm in a Tower Defence video game while understanding the main aspects behind the algorithm and what makes it special from other Machine Learning-like algorithms.

This alongside creating a well-structured agent to be able to win the game with the highest score. The agent not only must win the game with the optimal score, but tries to imitate the behaviour of a human being as much as possible in terms of currency handling and strategic tower placements.

Chapter 2: What is Machine Learning?

The concept of Machine Learning [20] has changed in recent years from the past and that's mainly because of new computing technologies emerging. The term Machine Learning refers to pattern recognition and the theory that computers can learn without being programmed to perform specific tasks. It is seen as a subset of artificial intelligence and its algorithms build a mathematical model based on a data sample, known as "training data", in order to make predictions or decisions without being explicitly programmed to do so.

The iterative aspect of machine learning is important because as models are exposed to new data, they are able to independently adapt. They learn from previous computations to produce reliable, repeatable decisions and results.

Machine learning algorithms are used in a variety of ways. Applications, such as email filtering and computer vision, where it is difficult or impractical to develop traditional algorithms to perform the required tasks

There are different ways to train machine learning algorithms, each with its own advantages and disadvantages. To understand the pros and cons of each type of machine learning, we must first look at the four types it's divided into which are:

1) Supervised Machine Learning.
2) Unsupervised Machine Learning.
3) Semi-Supervised Machine Learning.
4) Reinforcement Learning.

2.1. Supervised Machine Learning

Supervised machine learning is based on supervision, as its name suggests. In the supervised learning approach, this implies that we train the machines using the labeled dataset, and then the machine predicts the output based on the training. Here, the labeled data indicates which inputs have already been mapped to which output. More precisely, we may state that after training the machine with input and related output, we ask it to predict the outcome using the test dataset.

Assume we have a dataset of photos of dogs and cats as our input. Therefore, we will first train the computer to comprehend the photos, teaching it things like the size and

form of a dog's tail, the shape of a cat's eyes, their color, and their height (dogs are taller than cats, for example). After training, we input a cat image and ask the computer to recognize the object and forecast the outcome. Now that the machine is educated, it will examine every characteristic of the thing, including height, form, color, eyes, ears, tail, and so on, and determine that it is a cat. As a result, it will be classified as a cat.

This is how the computer recognizes things in Supervised Learning, and the technique's main objective is to map the input variable (x) with the output variable (y). Applications of supervised learning in the real world include spam filtering, fraud detection, and risk assessment.

2.2. Unsupervised Machine Learning

In contrary to Supervised Machine Learning, Machine learning techniques that don't have a known or labeled output are referred to as "unsupervised" algorithms. They have a predetermined output that has been labeled.

Knowing the difference enables you to comprehend why unsupervised machine learning techniques cannot be used to solve regression or classification issues since you are unsure of the potential value or solution for the output data. You can't train an algorithm as normally as you would if you don't know the value or solution; however, it is possible to apply unsupervised learning to identify the fundamental structure of the data.

An example would be in a scenario like this: You're at the grocery store and spot a fruit that isn't labeled that you've never seen before. You can distinguish the unknown fruit from other fruit around based on your observations of the shape, size, or color of the unusual fruit. This approximately describes the process of unsupervised machine learning.

The technique’s primary goal is to classify or group the unsorted dataset based on commonalities, patterns, and differences. The hidden patterns in the input dataset are to be found by the machines.

2.3. Semi-Supervised Machine Learning

A mix of both Supervised and Unsupervised Machine Learnings, It employs both vast amounts of unlabeled data and little labeled data, combining the advantages of both supervised and unsupervised learning without the difficulties associated with obtaining a lot of labeled data. Therefore, it’s not needed to utilize as much labeled training data when training a model to label data.

2.4. Reinforcement Learning

Reinforcement Learning [1] [20], which is the Machine Learning technique used in my project, is different from supervised learning as it relies only on the experiences of the agents.

With RL, an AI agent autonomously explores its surroundings by striking and trailing, acting, learning from experiences, and increasing performance. Reinforcement learning operates on a feedback-based method.

The objective of a reinforcement learning agent is to maximize the rewards since the agent is rewarded for every good activity and penalized for every negative action.

Reinforcement Learning will be discussed in detail in the next section, but for now, check the following comparison between the mentioned Machine Learning types in figure 1.

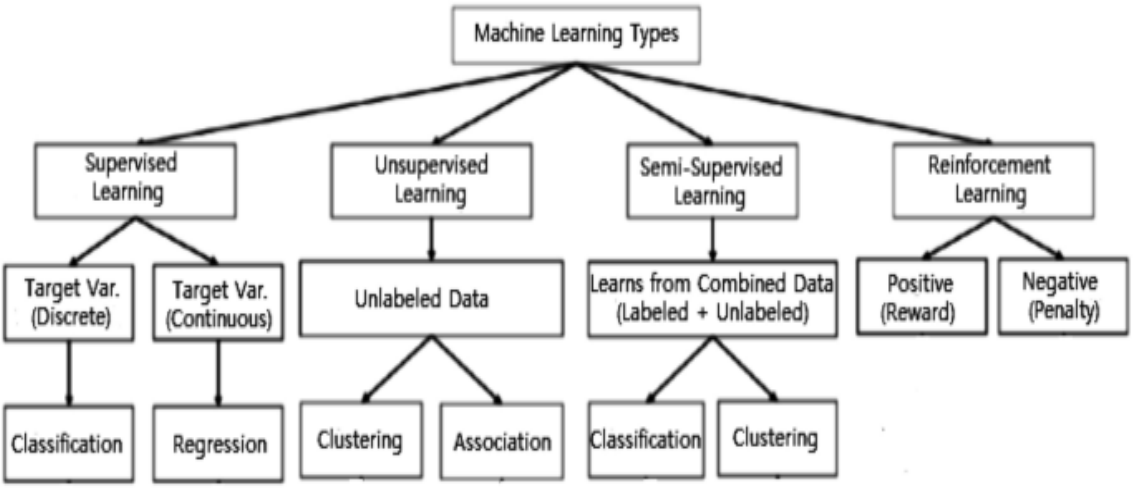

Figure 1: Machine Learning Types[22]

Chapter 3: Deep dive into Reinforcement Learning

As was previously discussed, Reinforcement learning, a type of machine learning technique, enables an agent to learn in an interactive environment via trial and error while using feedback from its own actions and experiences.

Although both Supervised Learning and Reinforcement Learning use the mapping between input and output, the latter uses rewards and punishments as signals for positive and negative behavior, in contrast to Supervised Learning, which provides the agent feedback in the form of the proper set of actions to perform a task.

Moreover, Reinforcement Learning has distinct goals from Unsupervised Learning. While finding similarities and differences between data points is the aim of unsupervised learning, the aim of Reinforcement Learning is to identify an appropriate action model that would maximize the overall cumulative reward of the agent. The action-reward feedback loop of a general RL model is shown in figure 2.

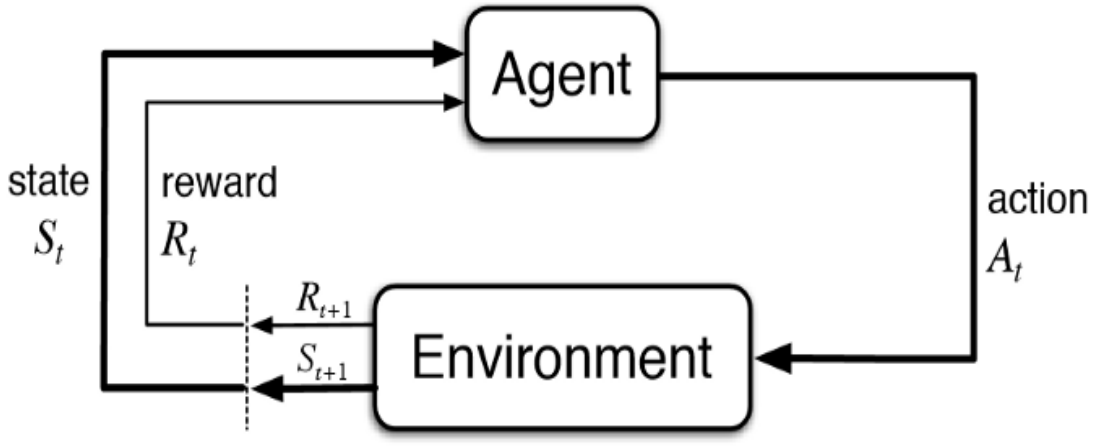

Figure 2: Reinforcement Learning Process

Although the developer establishes the reward scheme or the game's rules, he offers the agent no tips or advice on how to win or reach the optimal solution. The agent must

determine how to complete the objective to maximize the reward, starting with completely arbitrary trials and ending with complex strategies and superhuman abilities.

If a reinforcement learning algorithm is performed on supercomputer infrastructure, artificial intelligence can learn from thousands of concurrent

games, unlike humans. The best way to demonstrate an RL problem is through a classic game of Pac-Man.

In a game of Pac-Man, the main goal of the agent (Pac-Man) is to consume all the food in the grid (interactive environment) while dodging ghosts. The reward, in this case, is awarded when the agent consumes food while he gets punished when is killed by a ghost (loses the game).

The states represent the agent's position within the grid, and the agent's final cumulative reward is winning the game. However, in order to reach the optimal policy, the agent must decide how to explore new states while simultaneously maximizing its total reward. A trade-off between exploration and exploitation is what this is. To balance both, the agent might also make short-term sacrifices to gather sufficient data to enable future decision-making at the highest level.

To formulate a basic Reinforcement Learning problem in a Pac-Man game (figure 3), the key terms of an RL problem can be seen like this:

1) **Environment —** The game's grid.
2) **State —** The location of the Pac-Man in the grid world.
3) **Reward —** The amount of food consumed by Pac-Man.
4) **Cumulative Reward** — Pac-Man winning the game.
5) **Policy —** Method to map Pac-Man's state to actions.
6) **Value** — Future reward that Pac-Man would receive by taking an action in a particular state.

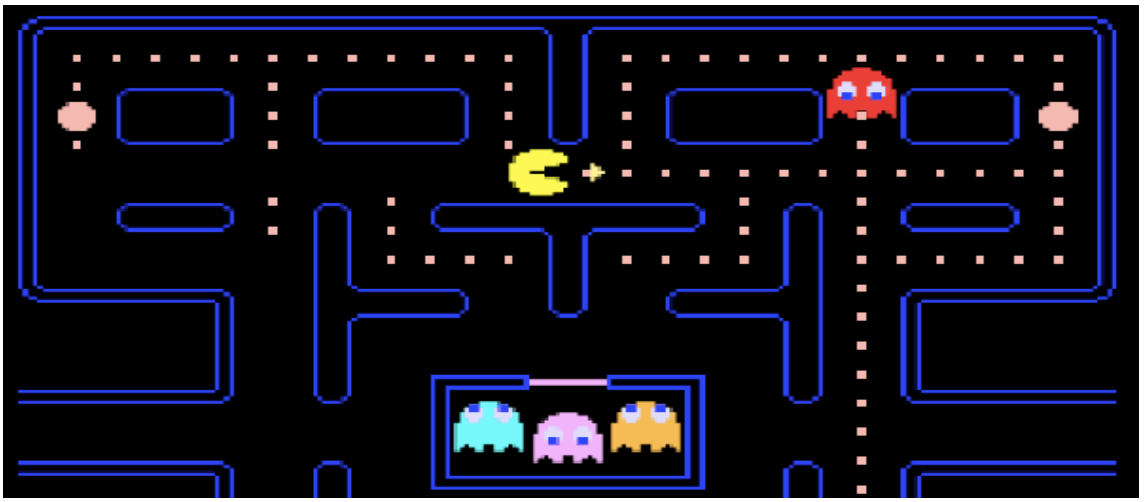

Figure 3: PacMan

However, as we mention the Environment, it's should be noted that there are two types of environments when it comes to Reinforcement Learning, Deterministic and Stochastic as seen in figure 4.

Deterministic refers to a scenario in which the reward model and state transition model are both deterministic functions. Simply put, if an agent repeats an action in a given state, it can anticipate receiving the same reward and moving on to the next state.

While when it's with a random likelihood of happening is said to be stochastic. In such a setting, an agent cannot be certain that repeating an action would result in the same reward or the next state.

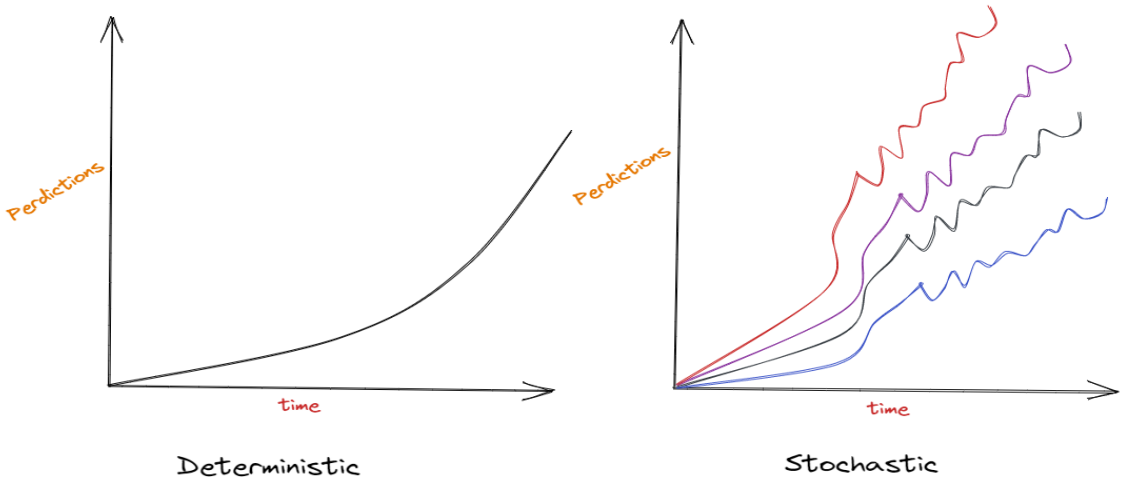

Figure 4: Determistic vs stochastic

3.1. Model-Based vs Model-Free Reinforcement Learning

Although there are many distinct kinds of Reinforcement Learning algorithms, Model-Based and Model-Free RL are the two main types [7]. Both of them are motivated by our knowledge of how humans and animals learn, and are demonstrated in figure 5.

The law of effect, which was put forth by psychologist Edward Thorndike in the late nineteenth century, states that responses that have negative effects become less likely to occur in the future and actions that have positive effects in a situation become more likely to occur again in that situation.

Later, the law of effect helped to establish behaviorism, a school of psychology that looks at stimuli and responses to explain how people and animals behave.

The foundation of Model-Free Reinforcement Learning is the Law of Effect. An agent senses the environment acts and measures the reward in model-free reinforcement learning. Typically, the agent begins by performing random behaviors before eventually repeating those that are linked to greater rewards.

Model-Free Reinforcement Learning lacks any direct world knowledge or world models. Trial and error must be used by the RL agent to directly feel every result of every action.

Thorndike's Law of Effect was widely used until Edward Tolman, a different psychologist, made a significant discovery while examining how quickly rats could learn to navigate mazes. Tolman discovered during his research that animals could discover their surroundings without being rewarded.

A rat released into a maze, for instance, would freely explore the tunnels and gradually come to understand the layout of the surroundings. The same rat can achieve its objective considerably more quickly than animals who were not given the chance to explore the maze if it is later reintroduced to the same environment and is given a reinforcement signal, such as finding food or looking for the exit. Latent learning is what Tolman referred to as.

Latent Learning gives both animals and people the ability to create a mental model of their environment, simulate potential outcomes in their thoughts, and predict the conclusion. Additionally, this is the cornerstone of Model-Based Reinforcement Learning.

Model-Based reinforcement learning's key advantage is that it spares the agent from having to learn through trial and error in its surroundings. For instance, Model-Based RL will enable you to mentally simulate alternate routes and alter your course if you learn that an accident has closed the road you usually use to work. You wouldn't be able to apply the new information using model-free reinforcement learning. As soon as you get to the accident scene, you would update your value function and begin investigating further options.

The Model-Based approach is useful when creating AI systems that can master deterministic board games like chess and go.

In some circumstances, it is either impossible or too complex to construct a good model of the environment. Additionally, Model-Based Reinforcement Learning has the potential to be exceedingly time-consuming, which in instances where time is of the essence may prove to be hazardous or even fatal.

In essence, neither Model-Based nor Model-Free Reinforcement Learning offers the ideal solution. Additionally, there is a good likelihood that any Reinforcement Learning system solving a challenging task is utilizing both Model-Based and Model-Free RL and perhaps even additional learning methods.

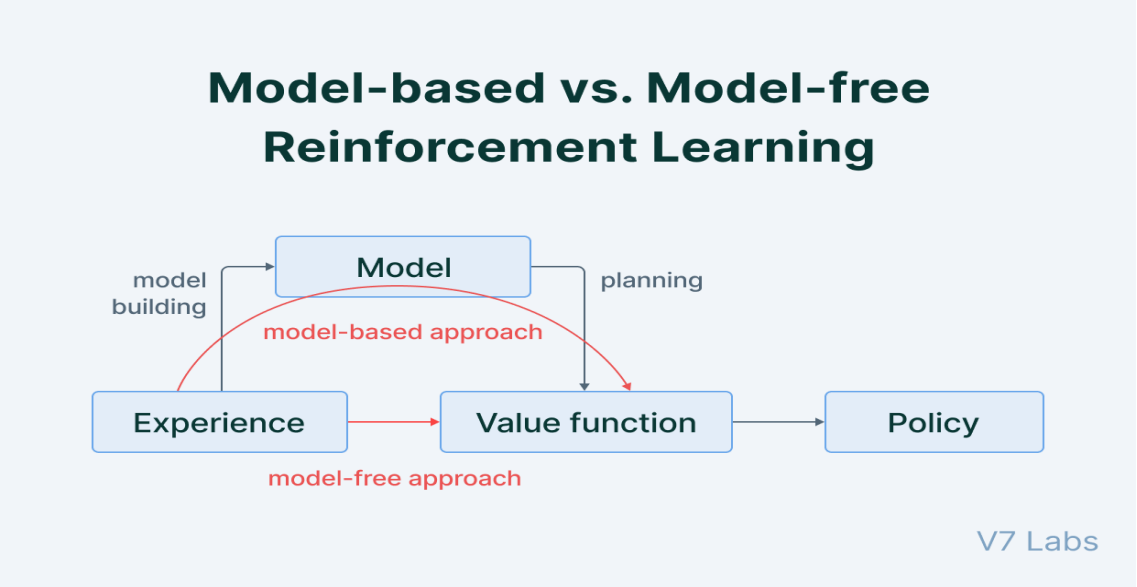

Figure 5: Model-Based vs Model-Free

3.2. Markov-Decision Process

The Markov-Decision Process is a way to mathematically model how agents make decisions. It can be used to find the optimal decision in any situation where there are a finite number of options and outcomes. The process involves breaking down the decision into a series of smaller decisions, each of which can be represented by a Markov chain. This makes it possible to use dynamic programming to find the best decision for each situation.

It's often used in artificial intelligence and machine learning applications, as it can help agents learn how to make optimal decisions in uncertain environments.

A Markov Process is defined by ***(S, P)*** where ***S*** are the states, and ***P*** is the state-transition probability. It consists of a sequence of **random** states S_1, S_2, **…** where all the **states obey the Markov Property**. See figure 6.

Markov Property requires that “the future is independent of the past given the present”.

A state S_t is *Markov* if and only if

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, ..., S_t]$$

Figure 6: Markov Property

3.2.1. Markov Reward Process (MRP)

An MRP is defined as (S, P, R,), where S are the states, P is the probability of state change, R s is the reward, and is the discount factor (will be covered in the coming sections).

The state reward R_s is the anticipated reward across all potential states that one could enter after leaving state s. For being in the state S t, you get this prize. By convention, it is treated as R (t+1) and is said to have been received after the agent departs the state.

3.2.2. Markov Decision Process (MDP)

MDP [3] can be used to model and solve problems in which an agent needs to make decisions in order to maximize some goal. For example, an MDP could be used to help a robot navigate through an unknown environment by choosing the best action at each step in order to reach its goal.

$$\mathcal{P}^{a}_{ss'} = \mathbb{P}\left[S_{t+1} = s' \mid S_t = s, A_t = a\right]$$
$$\mathcal{R}^{a}_{s} = \mathbb{E}\left[R_{t+1} \mid S_t = s, A_t = a\right]$$

Figure 7: Markov Decision Process

(S, A, P, R) is the definition of an MDP [12], where A is the set of actions. Essentially, it is MRP with actions. Since the state transition probability and state rewards were previously more or less stochastic, the introduction of actions generates a sense of control over the Markov Process. Now, though, the agent's choice of action also affects the rewards and the subsequent condition. In essence, the agent is now in charge of its own destiny.

3.2.2.3 Bellman Expectation Equation

In decision theory, the Bellman expectation equation, figure 8, is a key result that provides a way to optimize decisions when agents face uncertainty. This equation was first proposed by Richard Bellman in 1957 as a way to deal with the "curse of dimensionality" in dynamic programming.

The curse of dimensionality refers to the fact that the number of possible states that an agent can occupy grows exponentially with the number of variables considered.

Bellman's equation provides a way to reduce the number of states that need to be considered by focusing on expected values.

It's is derived from the Bellman optimality principle, which states that an optimal decision must be made in every state such that the expected value of the total reward is maximized. The total reward is composed of two parts: the immediate reward R_(t+1) and the future reward γ.v(S_(t+1)). The future reward is often unknown, so it must be estimated.

This estimation is done using the concept of discounting, which assigns a lower value to future rewards than to immediate rewards. The discount rate reflects how much value an agent places on future rewards relative to immediate rewards.

$$v_\pi(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s \right]$$

$$q_\pi(s, a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a \right]$$

Figure 8: Bellman Expectation Equation

3.3. Q-Learning

Given the agent's present state, Q-learning [5] is a model-free, off-policy reinforcement learning technique that will determine the appropriate course of action, meaning the agent will choose what to do next based on its state in the environment.

Its primary goal is to determine the optimum course of action given the situation as it is and in order to accomplish this, it could devise its own set of rules or might deviate from the prescribed course of action. This indicates that there is no real need for a policy, which is why it is referred to as an off-policy.

The 'Q' in Q-learning stands for quality. Quality here represents how useful a given action is in gaining some future reward.

3.3.1. Important Terms in Q-Learning

1. States: The State, S, represents the current position of an agent in an environment.
2. Action: Action, A, is the step taken by the agent when it is in a particular state.

3. Rewards: For every action, the agent will get a positive or negative reward.
4. Episodes: When an agent ends up in a terminating state and can't take a new action.
5. Q-Values: Used to determine how good an Action, A, taken at a particular state, S, is. Q (A, S).
6. Temporal Difference: A formula used to find the Q-Value by using the value of current state and action and previous state and action.

Let's take a look at a simplistic example of Q-Learning [4] that would clear the logic more. Imagine, there is an agent that's the main objective is to find a way through a maze. However, there are hidden mines on the road and the agent can only move one tile at a time. The accumulative reward for the agent is to reach the final point in the shortest time possible, while the negative reward for him is if he steps on the mine and dies. See figure 9 for demonstration.

The scoring/reward system is as below:

1. The robot loses 1 point at each step. This is done so that the robot takes the shortest path and reaches the goal as fast as possible.
2. If the robot steps on a mine, the point loss is 100 and the game ends.
3. If the robot gets power, it gains 1 point.
4. If the robot reaches the end goal, the robot gets 100 points.

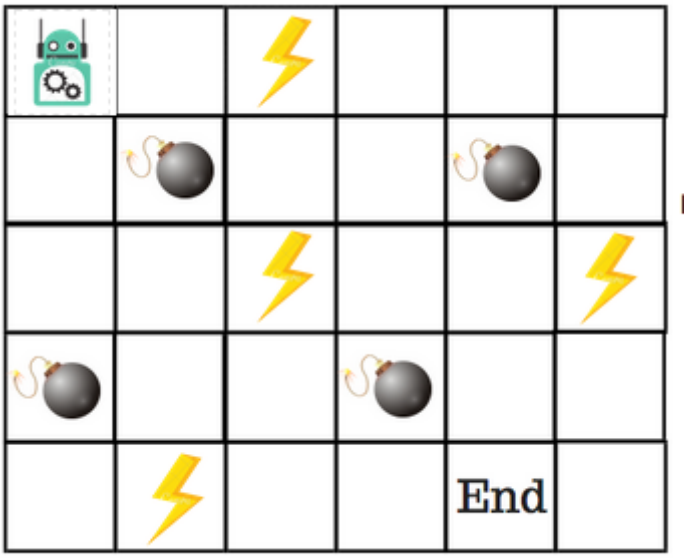

Figure 9: Q-Learning example 1

To train the agent to reach the end goal and achieve the accumulative reward, a concept called Q-table is introduced. Using a Q-Table, we can determine the maximum predicted future rewards for action in each stage as seen in figure 10.

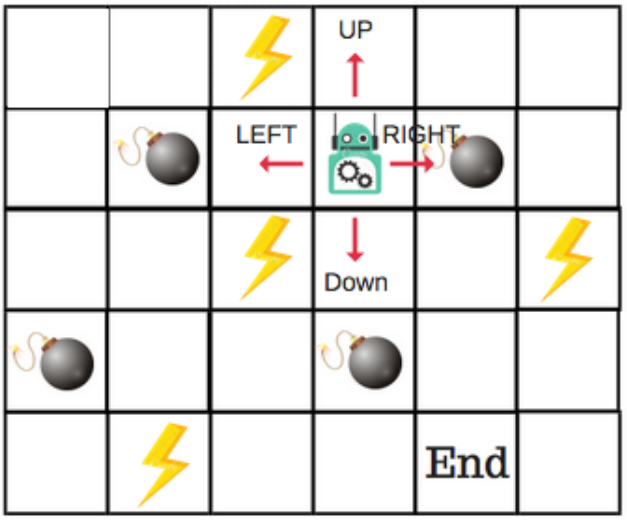

Figure 10: Q-Learning example 2

Taking a look at figure 11, four actions could be taken by the agent, either moving up, down, right, or left. This can be mapped and converted into a lookup Q-Table, where columns are the actions and rows are the states.

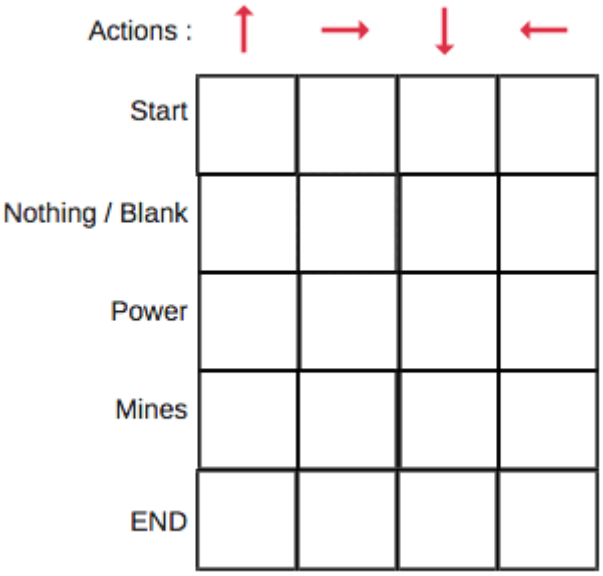

Figure 11: Q-Learning example 3

The greatest projected future reward for each action that the agent would take at that stage is represented by the Q-Table score. As we need to enhance the Q-Table at each iteration, this is an iterative procedure. However, to calculate the respective values, a Q-Learning algorithm is applied, which uses the **Bellman equation** (figure 12) and takes two inputs: state (**s**) and action (**a**).

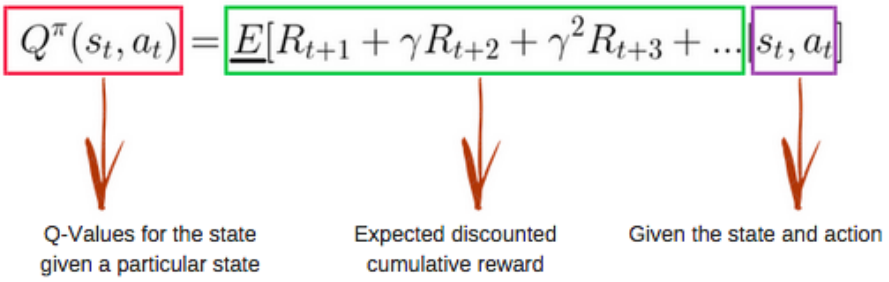

Figure 12: Q-function

Starting with all zero values, using the above function, we can get the values of **Q** for the cells in the table, but this is only the start of an iterative process of updating the values. As we start to explore the environment**,** the Q-function gives us better and better approximations by continuously updating the Q-values in the table.

The process can be summarized in figure 13 but will be discussed in detail in the upcoming section.

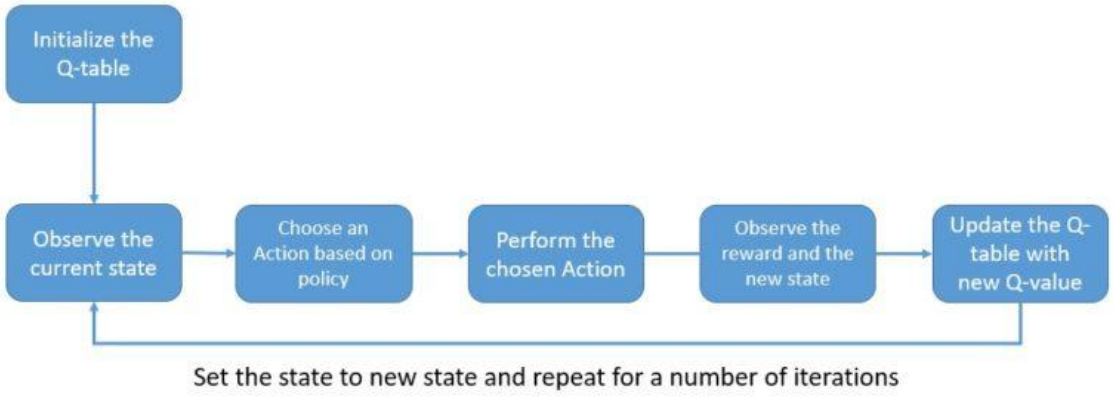

Figure 13: Q-Learning algorithm process

3.3.2. Q-learning algorithm process

Step 1: Initialize the Q-Table

As mentioned before, the Q-table start with all-zero rows (m) and columns (n), where n stands for the number of actions and m for the number of states. In the example mentioned earlier, there are four actions (up, down, right, left) and five states (start, end, blank, power, mine).

Steps 2 & 3: Choose and perform an action

Based on the Q-Table, the agent will select an action (a) in the state (s). However, as was already established, at the beginning of the episode, each Q-value is 0, and this is where the trade-off between exploration and exploitation comes into play. For this, the epsilon greedy approach will be used to select the action with the highest estimated reward most of the time. See figure 14.

The epsilon rate is always higher in the beginning, which would force the agent to explore the environment and randomly chooses an action. With more exploration, the epsilon rate decreases and the agent instead opts to exploit the environment.

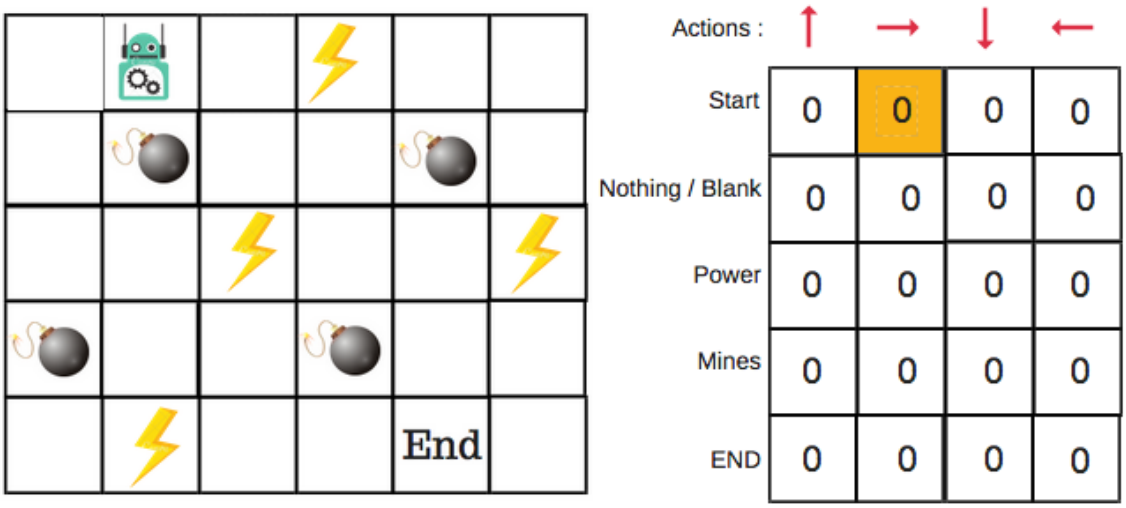

Figure 14: Q-Learning example 4

Steps 4 & 5: Evaluate and repeat

After an action was taken, the outcome and reward will be calculated, leaving the Bellman equation to be updated like the following, and the action with the highest value will be taken. The same function will keep on being repeated, making the Q-Table always updated until achieving the goal with the optimal score.

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

Figure 15: Q-function equation

3.4 Reinforcement Learning with Neural Network

While creating and using a Q-table is easy in simple situations, it can be extremely challenging in some real-world environments. It is quite inefficient to manage Q-values in a table because there can be thousands of actions and states in a real-life scenario. Here, rather than using a table to forecast Q-values for actions in a given state, we can use neural networks [2]. In the Q-learning process, we initialize and train a neural network model rather than initialize and update a Q-table.

Each layer of a neural network is made up of several processing nodes that are coupled closely together, and these layers are made up of three things.

1) **Input Layer:** The input layer typically has a predetermined number of nodes that match the input data, like the number of states in an environment.
2) **Hidden Layers:** The architecture of a neural network often has one or more hidden layers. The architecture's hyperparameters include the number of layers and nodes in each layer.
3) **Output layer:** Likewise contains a predetermined amount of nodes that correlate to the output that is needed, such as the number of actions in an environment.

An example if exists 16 states in the environment, they will be represented with the same number of nodes in the input layer, while the number of actions in the environment will be represented by nodes in the output layer. Moreover, there is a fully-connected single hidden layer comprised of 20 nodes as demonstrated in figure 16.

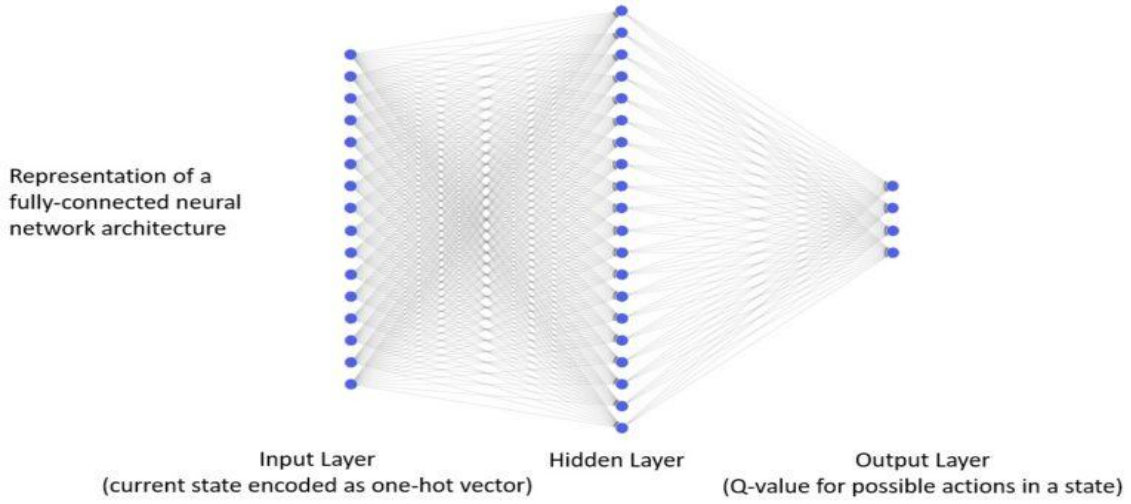

Figure 16: Neural Networks

Depending on the inputs received from the preceding nodes, weights, and biases it learns, a processing node generates an output while also making use of an activation function. The goal of that is to give the output, which is primarily linear, with some non-linearity, which makes the neural network able to learn complex and real-world patterns. [6]

Given that selecting an activation function for a neural network involves optimization, it is included in the list of hyperparameters. But we may get off to a solid start by considering the nature of the incoming data and the desired outcome. We'll use the Linear activation function (figure 17) in the output layer and the Rectifier Linear Unit (ReLU) as the activation function in the hidden layer.

To lower the error in the predictions that they can make, Neural Networks operate by iteratively updating the model's weights and biases. To need to be able to determine the model inaccuracy at any given time, the loss function is used. In Neural Network models, loss functions are frequently used like cross-entropy and mean-square-error.

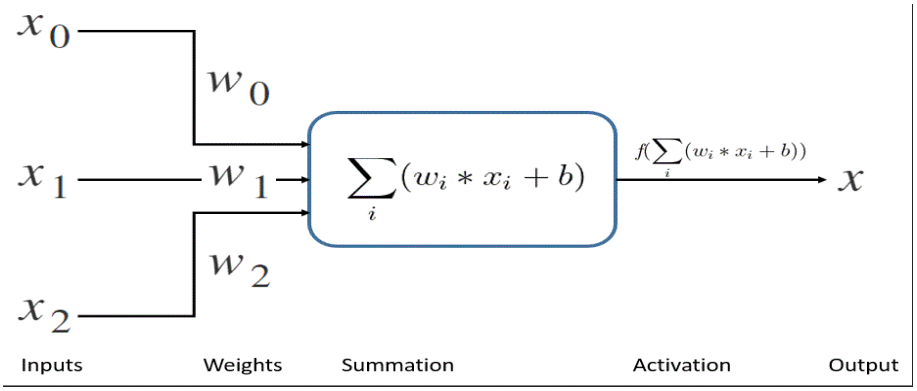

Figure 17: Activation Function

The square value of the difference between the prediction and the target is measured by the mean-squared-error loss function:

$$\text{loss} = \{(\text{r} + \gamma * max_{a'} Q'(s', a')) - Q(s, a)\}^2$$

Figure 18: Mean-squared-error loss function

The idea behind calculating the loss function is to update the weights by taking the feedback backward through the network. This process is referred to as backpropagation and can be accomplished using a variety of algorithms, starting with the conventional stochastic gradient descent.

Chapter 4: The Game concept

As mentioned before, the game chose to demonstrate the work is a single-player Tower Defence game developed using the Unity 3D engine (figure 19). Tower Defence games are of a strategic/tactical subgenre, in which the player must protect their territory from waves of enemy attackers.

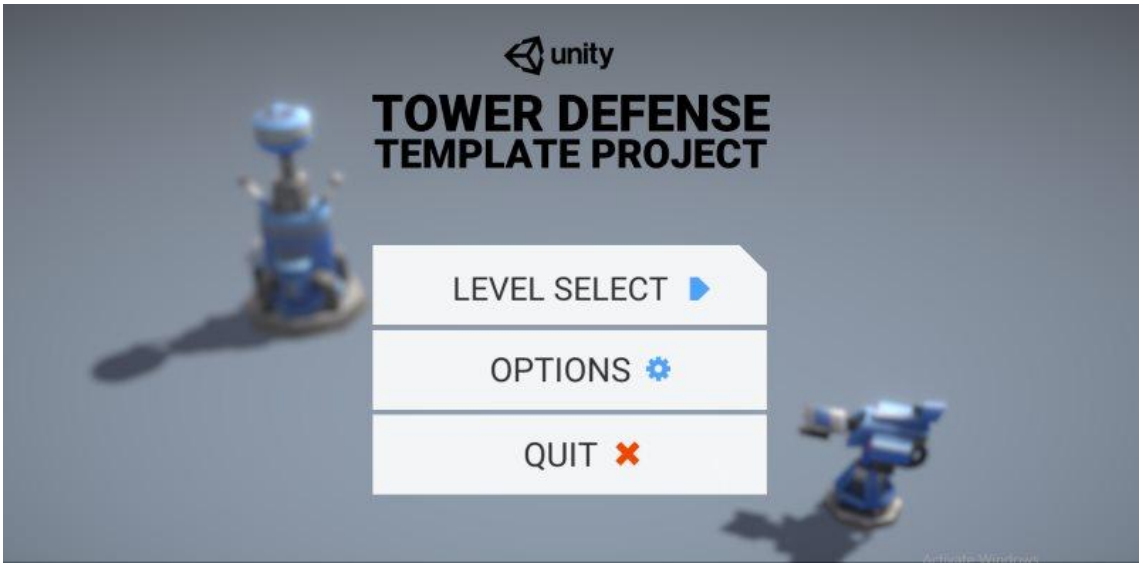

Figure 19: Tower Defence menu

Typically, the player must line the path where the waves of enemies are moving down with towers and traps. However, the player doesn't have direct control over these towers as they only start a fire when the enemy comes closer to them.

Tower Defence games are often used to test and develop AI algorithms, as they provide a challenging environment in which to learn and optimize strategies.

This project has been much more oriented to getting a functional game where the ML-Agents plugin [10] could be tested and a nice result in terms of machine learning could be obtained.

4.1. Game walkthrough

There isn’t much difference in the game used to other games out there, as the player finds themselves having the choice of selecting of a number of levels, each one with a

different environment, towers, and enemies. However, the main difference comes in the difficulty, with higher levels having several paths for the enemy tanks to pass through, which makes it harder to defend the base.

For simplicity and demonstration, the level chosen for our work will be the first level, see figure 20, where there is only one path for the enemy to go through, making it easier and faster for the agent to develop an approach to stop and destroy the tanks and defend the base.

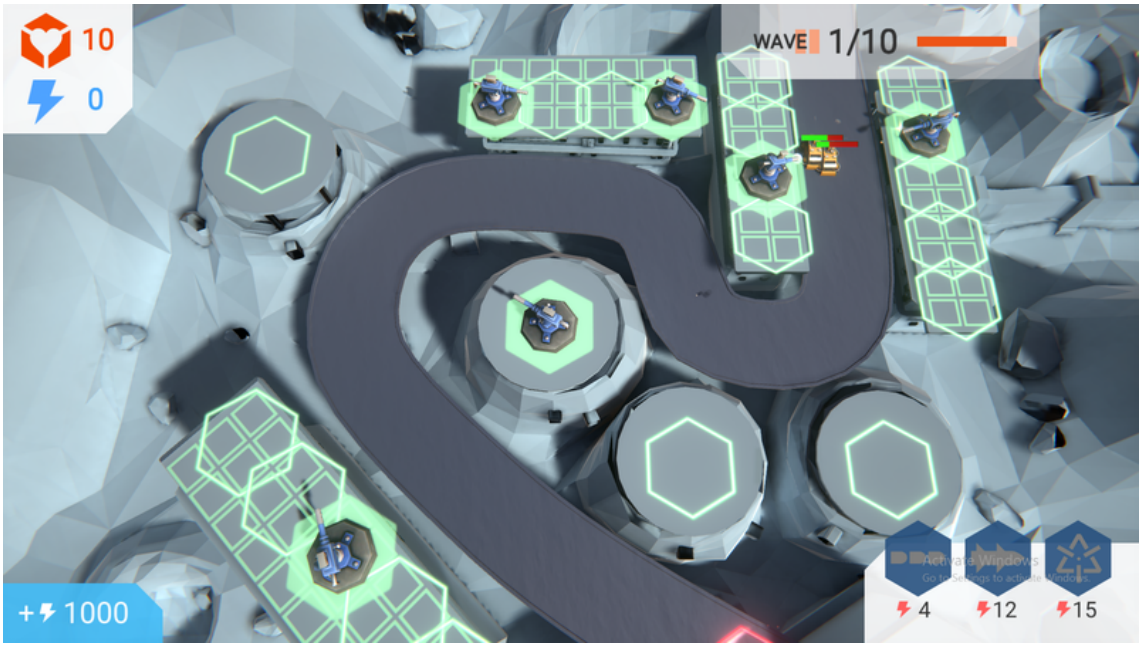

Figure 20: Tower Defence environment

4.2. Environment

As it's seen in figure 20, there is a single entry, where the enemies start to try to reach the base, shown in red, which the player must defend. For each second the enemy tank stay in the base, the health, in the top left corner, will drop by one, meaning tanks can't stay for more than 10 seconds in the base or the player will lose the game.

On the other hand, to win the game, the player must survive the ten waves without losing the full health he started with. The higher health he ends the game with, the more the final score he achieves.

Moreover, the player can place towers in 22 different placements, three grids, and four singles, where two towers can't be placed in the same placements. The towers are purchased by a game currency, shown in the top left corner as a blue thunder icon, where each tower has a different cost, depending on its power and range.

4.2.1. Towers types

Figure 21: Tower Defence types

1) **Assault Cannon (Tier 1):** The cheapest option with a price of 4 currencies. Has the shortest of ranges, but a high fire rate; however with low damage (1.00 DPS).

2) **Assault Cannon (Tier 2):** An upgrade of tier 1 with a price of 8 currency. It Posses the same attributes of the tier 1, with the difference coming in the damage (1.50 DPS).

3) **Assault Cannon (Tier 3):** An upgrade of tier 2 with a price of 14 currency. Posses the same attributes of the tier 1 and 2, with the difference coming in the damage (2.00 DPS).

4) **Rocket Platform (Tier 1):** Starts at the price of **12 currency**. Has a higher range and damage, but only affects ground enemies (2.70 DPS).

5) **Rocket Platform (Tier 2):** An upgrade of tier 1 with the price of **24 currency**. The same attributes of tier 1, but with higher damage (3.50 DPS).

6) **Rocket Platform (Tier 3):** An upgrade of tier 2 with the price of **32 currency**. Has a higher range and damage, but only affects ground enemies (6.00 DPS).

7) **Plasma Lance (Tier 1):** The most powerful tower available with a huge range; however, with a very slow fire rate starts at the price of **15 currency** (3.60 DPS).

8) **Plasma Lance (Tier 2):** An upgrade on the damage of tier 1 with the price of **30 currency** (5.60 DPS).

9) **Plasma Lance (Tier 3):** An upgrade on the damage of tier 2 with the price of **40 currency** (8.00 DPS).

4.2.2 Actions

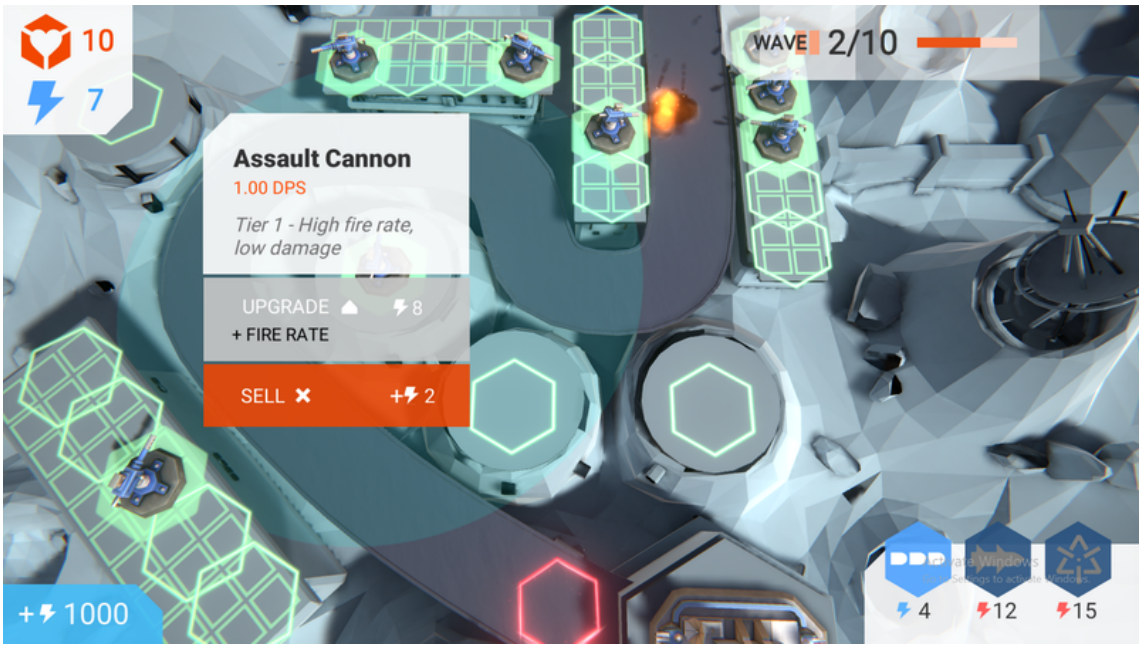

Figure 22: Tower Defence actions

While there are several actions the player can take in the game, in our work the agent will just have three actions to choose from, see figure 22, which include:

1) Buy a new tower and place it in an empty placement tile.
2) Upgrade an existing tower.
3) No action.

The actions will be made after taking into consideration the currency available, the progress of the game, health, etc... However, this will leave some actions out of the hands of the agent, which include:

1) Sell a tower.
2) Restart the game.
3) Add currency (a cheat button that was added for testing reasons)

4) Pause the game.
5) Quit.

4.2.3 Software needed

• Unity3D 2020.3.36f1 engine with the C# programming language, for the development of video game and artificial intelligence techniques. [15]

• Tensorflow 1.4 open source software library for high performance numerical computation. [14] [17]

• Machine Learning Agents (ML-Agents) 0.3 and TensorFlowSharp as the plugins to integrate reinforcement learning along with Unity.

• Anaconda 3 5.1.0, which includes Python 3.6.4, Conda 4.4.10, and Jupyter Notebook 4.4.0 under which Tensorflow runs. [13] [18] [19]

• Visual Studio 2019 version 16.11 as integrated development environment IDE.

4.3. Adding neural networks to the video game

As Reinforcement Learning witnessed a huge breakthrough recently, Unity made it easier for developers to implement the Machine Learning algorithm in their projects using a toolkit called Unity ML-Agents. It's a plugin that was developed only in 2017 and it allows developers to use the Unity Game Engine as an environment builder to train agents.

The benefits of using ML-Agents also include it being open source [11] with a very easy setup that requires minimal coding. Moreover, it has strong documentation with great example projects, making AI/Machine Learning expertise not required to master.

Reinforcement Learning in ML-Agents works following the Markov Decision Processes, which were discussed earlier in the report.

Within this context, the main functions within the learning loop are introduced. Calling the agent's RequestDecision() executes the following process, called Experience:

1) Observing the environment with the CollectObservations() function.
2) Taking an action using the OnActionReceived() function.
3) Get rewards returned from the SetAgentReward() function.

4.3.1. Observations, actions, and rewards for the video game

The ideal course of action is attempted via Reinforcement Learning given several observations. The observations must be sufficiently indicative of the state of the game because the agent will base its actions on these observations and its prior experiences each time it is asked to make a new decision.

All observations the agent has to take into consideration are shown in table 1.

The Number of Towers is retrieved for the agent to decide whether the agent should take the decision of buying a new tower or instead upgrade an existing one. Similarly is the number of empty placements, which the agent can decide upon the maximum number of towers he can buy in the future. Meanwhile, the number of currency decide whether the agent can buy a new tower, upgrade an existing one, or just do nothing. Finally, the number of lives is a way of penalizing the agent, because if the number falls to zero, the agent will lose the game, and subsequently gets a negative reward.

Observation	Values	Description
Number of Towers	[0,1,...,22]	The number of Towers placed in the environment
Number of Currency	[0,1,...,n]	The number of Currency the agent has
Number of lives	[0,1,...,10]	The number of lives left for the agent
Number of empty placements	[0,1,...,22]	The number of empty tiles left for the agent

Table 1: Observations Table

Depending on the observations retrieved from each state, the agent has the choice of making three different actions, which are to either buy a new tower, upgrade an existing one, or just take no action at all. See table 2.

These three actions are directly affected by the number of towers existing, the number of currency, and the number of empty placements. However, several times, the agent will observe empty placements and will have a huge number of currencies, but will still decide to take no action because it wouldn't affect its final reward.

Actions	ID	Description
Place Tower	[0]	Buys a new tower and places it on an empty tile
Upgrade Tower	[1]	Upgrades an existing tower in the environment
Do Nothing	[2]	Takes no actions

Table 2: Actions Table

The Reward Function is an incentive system that instructs the agent through reward and punishment on what is right and wrong. The agent will always want to maximize overall rewards and will even sometimes forego immediate rewards to increase overall rewards.

In this case the reward system is simple, the agent will get a positive reward in case he manages to win the game and punishment when he loses. See table 3.

Reward	Description
+50.0	Win the game
- 50.0	Lose the game

Table 3: Rewards Table.

4.3.2 Code

While the ML-Agent package allows minimal coding, there are some classes needed to be coded to implement the Reinforcement Learning algorithm. These classes each serve as a GameObject for a specific purpose.

It's recommended to take a look at the several examples that come with the ML-Agent plugin, especially the 3D-Ball Environment one, as its simplicity acts as a great introduction to how the plugin works. The plugin also comes in with great and in-depth GitHub documentation that will clear any doubt you may face.

However, in this section, we will not get into every class or piece of code that has been written, but just two important ones that shaped the project to the way it's right now.

The first step was to create the **AgentAI** class which is responsible for several aspects, including collecting observations, actions received, and several other methods which will be explained in detail later. The primary purpose of the Agent class, which is a GameObject with a script that inherits from the Agent Unity class, is to provide the brain with observations and rewards while also computing the actions the brain returns to it.

Agents in an environment operate in *steps*. At each step, an agent collects observations, passes them to its decision-making policy, and receives an action vector in response. In general, agents make observations using ISensor implementations; however, the ML-Agents have several implementations for visual observations, including CameraSensor, RayPerceptionSensor, and VectorSensor, which all can be added as components to an agent's GameObject. For the usage of these vector observations, the CollectObservations(VectorSensor) function can be implemented in the Agent class, which the agent will use before taking an action.

A decision-making policy can be implemented for the agent using a BehaviorParameters component attached to the agent's GameObject. The BehaviorType

setting determines how decisions are made:

- Default: Decisions are made by the external process, when connected. Otherwise, decisions are made using inference. If no inference model is specified in the BehaviorParameters component, then heuristic decision-making is used.

- InferenceOnly: Decisions are always made using the trained model specified in the BehaviorParameters component.

- HeuristicOnly: when a decision is needed, the agent's Heuristic(Single[]) function is called. The implementation is responsible for providing the appropriate action.

The following are different fields, properties, and methods that are necessary for the implementation of the agent.

Fields:

MaxStep: The max step value determines the maximum number of steps the agent takes before being done.

Methods:

OnEpisodeBegin(): Called to set up an Agent instance at the beginning of an episode, with EndEpisode() called at the end to set the done flag to true and reset the agent. See code block 1.

Code block 1: OnEpisodeBegin

```
public override void OnEpisodeBegin() {
  if(!first_scence)
   {
    string currentSceneName = SceneManager.GetActiveScene().name;
    SceneManager.LoadScene(currentSceneName);
    }
       first_scence = false;
}
```

CollectObservations(): An agent's observation is any environmental information that helps the agent achieve its goal. For example, for our project, the observations include the number of towers, the number of lives left, and the number of currencies. See code block 2.

Code block 2: CollectObservations

```
public override void CollectObservations(VectorSensor sensor)
    {
        int TowerCounterInt = spawnManager.GetTowerCount();
        sensor.AddObservation(TowerCounterInt);

        int currencyCounterInt = spawnManager.GetCurrency();
        sensor.AddObservation(currencyCounterInt);
    }
```

OnActionReceived(): Specifies the agent behavior at every step, based on the provided action. An action is passed to this function in the form of an array vector. The array is used to direct the agent's behavior for the current step. Actions for an agent can be either Continuous or Discrete, and can be of any size. In our case, the agent has three actions to choose from, either place a tower, upgrade a tower, or do nothing, meaning we had to use an array size of three. See code block 3.

Code block 3: OnActionReceived

```
public override void OnActionReceived(ActionBuffers actions)
    {
        int placementIndex = actions.DiscreteActions[0];
        int towerIndex = actions.DiscreteActions[1];
        int action = actions.DiscreteActions[2];
        if(action == 1)
spawnManager.TowerPlacementAI(placementIndex,towerIndex);
        else if(action == 2)
        {
            spawnManager.upgradeTowerAI(towerIndex);
        }
    }
```

AddReward(): One of the most important methods in the agent class and its purpose is to increment the step and episode rewards by the provided value. The method is used for both positive (to reinforce desired behaviour) and negative (penalizing mistakes) rewards. Moreover, the SetReward is used to assign a specific reward to a certain and current step, rather than increasing or decreasing it. See code block 4.

Code block 4: AddReward

```
public void SetAgentReward(float Reward)
{
    AddReward(Reward);
    reward += Reward;

}
public void gameWin()
{
    AddReward(50f);
    reward += 50;

    Debug.LogWarning("gameWin...." + reward);
    EndEpisode();
}
public void gameLose()
{
    AddReward(-50f);
    reward += -50;
    Debug.LogWarning("gameLose...." + reward);
    EndEpisode();
}
```

Finally, the **SpwanManager** class, which acts as the preparator for the agent and the ML-Agent. It can be viewed as the one who takes care of setting up the environment, and agent, starting the waves, updating the level, as well as placing and upgrading the towers. It also retrieves the number of towers each state, the number of currencies, and the number of lives left so the agent can act accordingly.

Overall it has 10 methods, each one with different purposes, and while it doesn't directly affect the training, it acts as a facilitator for the agent to observe and act on the environment. The class can be seen in the code block 5 in the appendix.

4.3.3 Training the agent

After writing the code for the AgentAI class, the next step is to implement it as a GameObject inside the Unity with Behavior Parameters so that we can start training the agent in real-time as seen in figure 23.

The first thing we can notice in the Behavior Parameters is that the AgentAI has a Vector Observation of Stack Size 2 because as was mentioned before, there are only two observations that matter for the agent to take actions, which are the Tower count and the Currency count.

Meanwhile, the Actions has 3 discrete branches, specifying the actions of placing a tower, upgrading a tower, or taking no action, with the first having 23 branches, which relates to the number of placements tiles.

It can also be seen that the model has a model called AgentAI and a behaviour type of Interference Only because this image was taken after the agent already was trained.

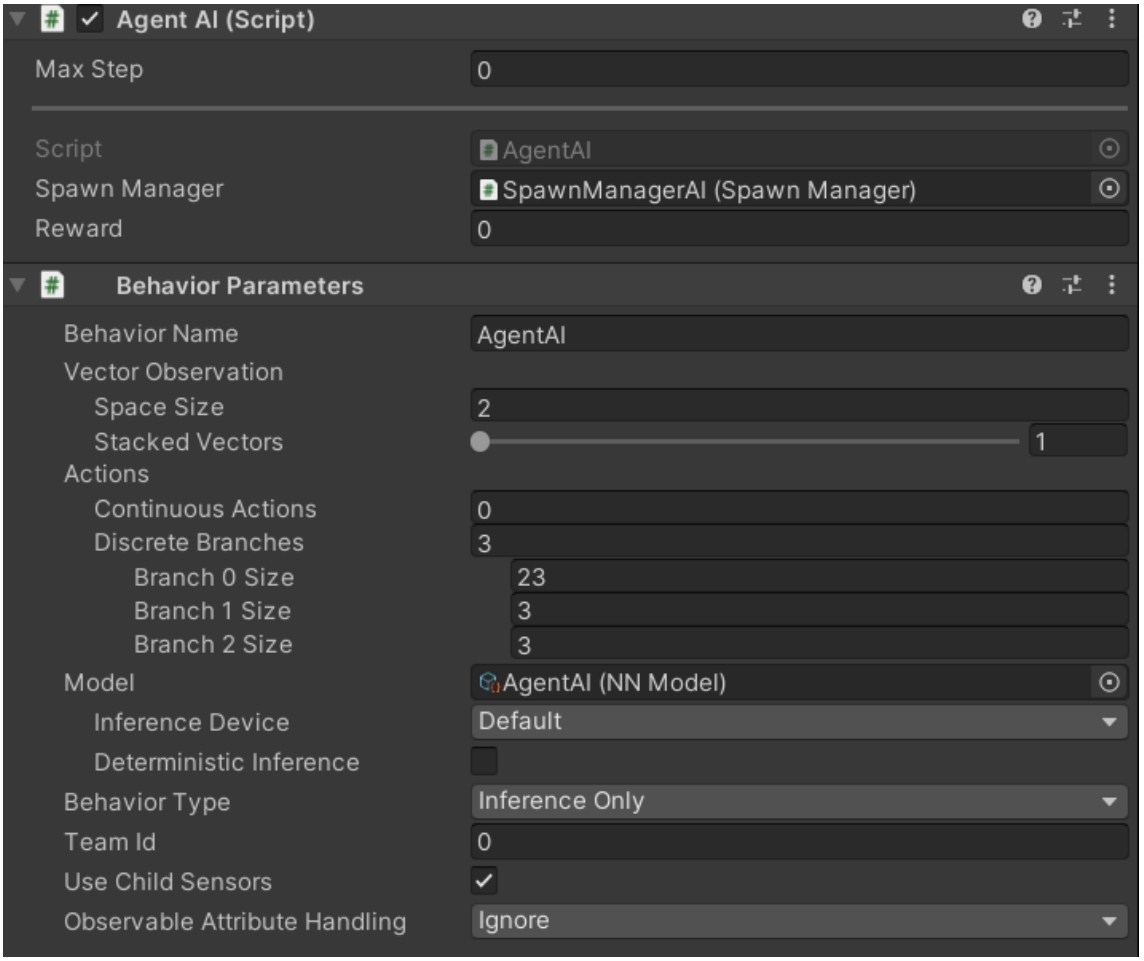

Figure 23: Agent Behaviour

After specifying the Behavior Parameters inside Unity, the next step that is needed to be done is to open the cmd and move to the directory of the game this will be followed by moving into the TrainerConfig folder, where the following command will be triggered:

“mlagents-learn trainer_config.yaml --run-id="TowerDefence_1"

The trainer_config.yaml refers to the trainer_config file inside the folder, while the run-id can be called anything the user wants.

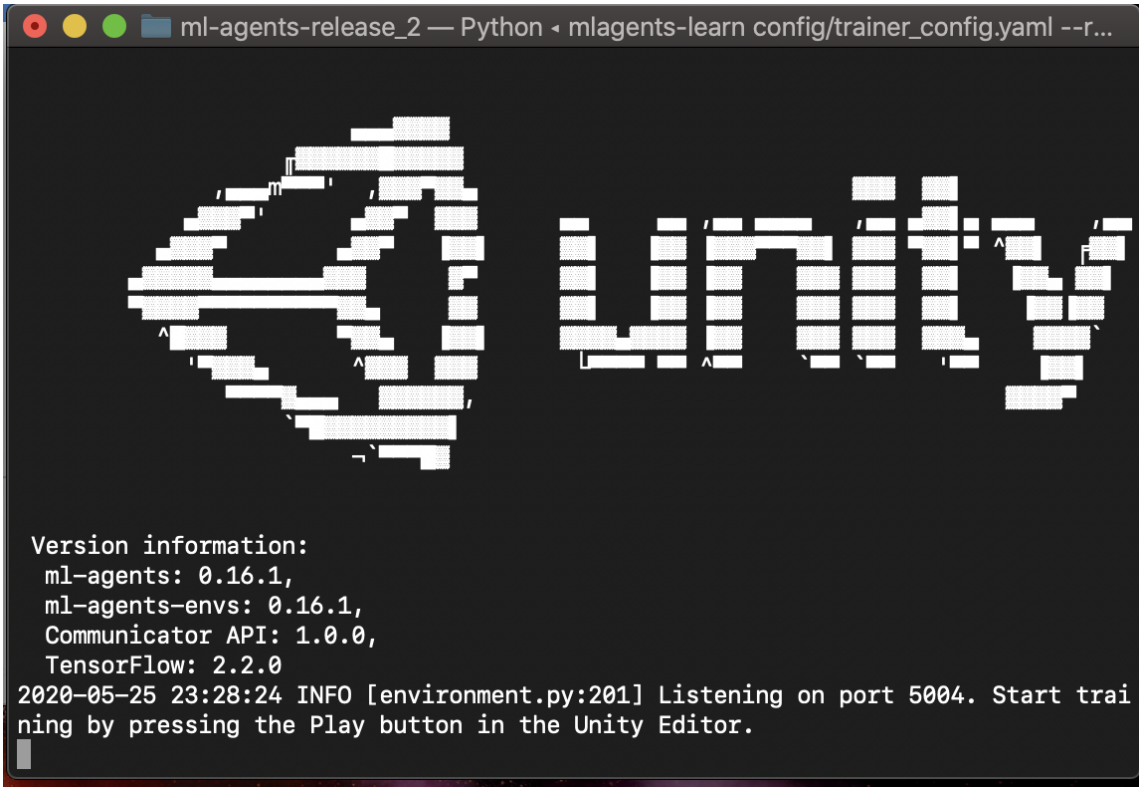

Figure 24: Unity Terminal

A Unity logo should appear inside the cmd terminal, see figure 24, as now the agent is ready to be trained. The only thing left to do so is to return back to the Unity Engine and press the play button and let the AI train in real time.

To speed up and stabilize the training, multiple agents could be trained together at the same time and this is done by duplicating the Agent GameObject multiple times inside the environment.

4.3.3.2. Understanding the trainer_config.yaml file

The trainer_config.yaml is a configuration file responsible for specifying the hyperparameters and other settings specific to ML-Agents, including the training. While these values can be altered optionally, their values are specified to adapt to the way the agent learns.

Let's have a look at the most important parameters inside the file of our project.

Batch Size: Refers to the number of training examples utilized in one iteration. The default size is 1024, which was applied in our project given its simplicity.

Buffer Size: Indicates how many experiences (observation, action, rewards loop) should be collected before updating the model. The larger the experience replay, the less likely correlated elements will sample, hence the more stable the training of the NN will be. The default value is 10240, which was left in our project.

Learning Rate: Often referred to as alpha or α, can simply be defined as the amount that the weights are updated during training. It usually has a small positive value, often in the range between 0.0 and 1.0. The default value is 0.0003 and has been used.

Gamma: This decides whether the agent should be in favour of a long-distant future reward or an immediate one, the higher the values, the more future rewards the agent will search for. In our game, we opted for the default value of 0.99 to take into consideration more observations.

Lambda: The lambda parameter decides how much you bootstrap on earlier learned value versus using the current Monte Carlo roll-out. This indicates a trade-off between more bias (low lambda) and more variance (high lambda). In many cases, initiating lambda to zero is already a fine algorithm, but setting lambda higher helps speed up things. The typical range is between 0.9 and 0.95, and the latter was used in the project.

Max Steps: Indicates how many steps there are in the current training procedure. When loading older models from which further learning is desired, it must be enhanced. The higher the number the more complex problem, but since our project is simple, we used the default values of 500000.

Beta: Its objective is to regularize the entropy, which ensures the agent properly explores all the possible actions during training. . The typical range is between 0.0001 and 0.01, and in our project, the values used were at 0.005.

Number of Layers: Refers to how many hidden layers are present after the observation input. More layers may be necessary for complex control problems. The typical range values are between 1 and 3. The default is 2 and it was used.

Hidden Units: Correspond to how many units are in each fully connected layer of the neural network. For problems where the correct action is a simple combination of the observations, this should be small. The typical range is between 32 and 512. The default is 128 and 64 was used.

Chapter 5: Results

The results are retrieved after several runs of the training have finished and the agent has managed to find the optimal and highest reward possible. The process was done and the graphs were made using TensorBoard.

TensorBoard [9] provides the visualization and tooling needed for machine learning experimentation, including tracking and visualizing metrics, visualizing the model graph, viewing histograms of weights, biases, or other tensors, and many more.

We will not review every metric that has been retrieved, but the main ones that provide significant information regarding the quality of the training.

5.1 Tensorboard summaries

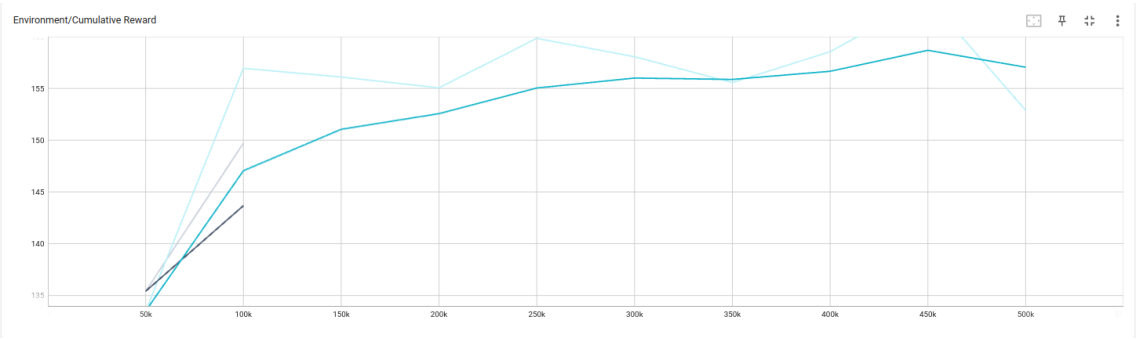

Figure 25: Cumulative reward

Figure 25 represents the average cumulative reward for all training agents. Since the goal is to get the agent to reach the biggest reward possible, it ought should rise after a productive training session. As we can from our training, the graph goes frequently ups and some downs, resulting in successful training.

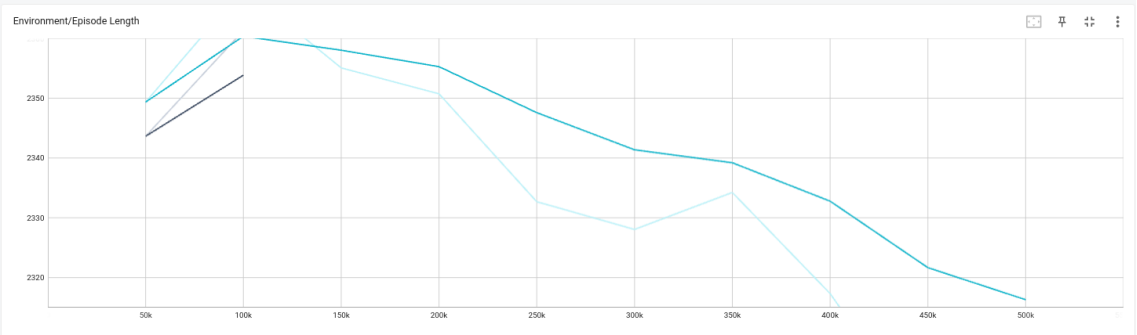

Figure 26: Episode Length

The mean length of each episode in the environment for all agents, and as it's shown in figure 26, has seen a severe drop during the training. This is due to the agent finding the optimal reward fast.

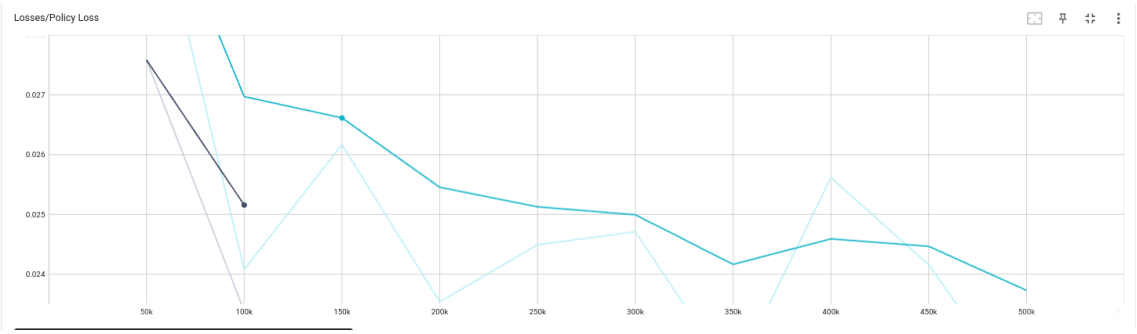

Figure 27: Policy Loss

The mean magnitude of the policy loss function. Pertains to the rate of change of the policy (process for deciding actions). The magnitude should decrease during successful training, which is seen in figure 27.

The Value Loss, figure 28, refers to the mean loss of the value function update. This measures how accurately the model can forecast the value of each state and it should increase while the agent is learning and the reward is increasing and decreasing once the reward becomes stable. See figure

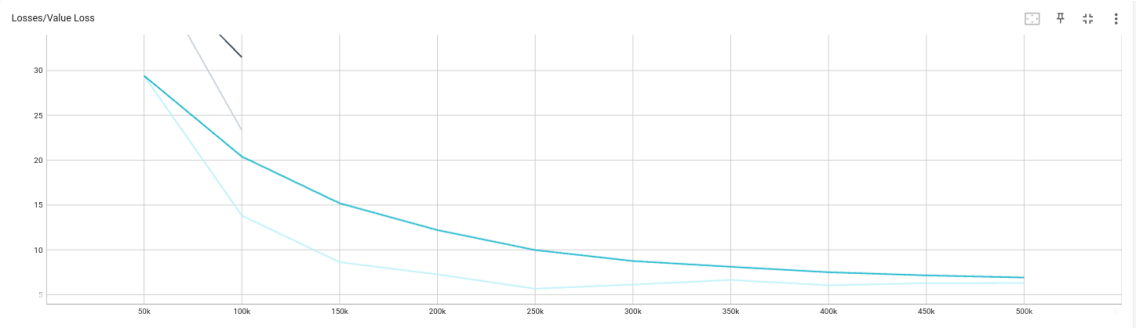

Figure 28: Value Loss

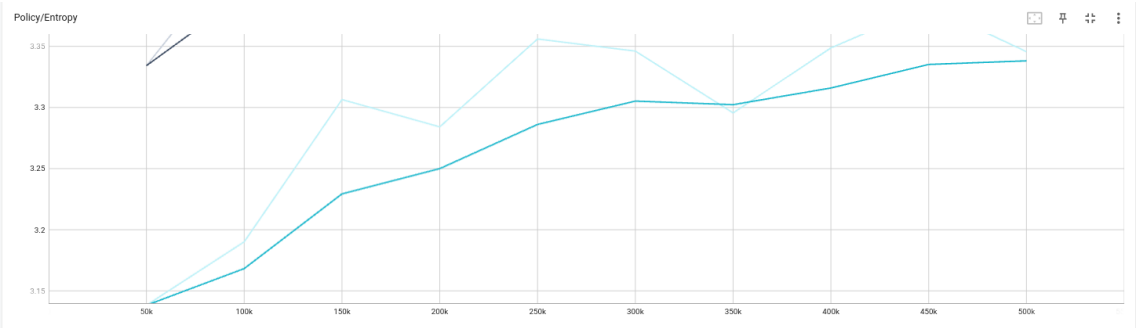

Figure 29: Entropy

Figure 29 corresponds to how random the decisions of a Brain are. This should consistently decrease during training. If it decreases too soon or not at all, beta should be adjusted (when using discrete action space), which should have happened during the training. In other words, the higher the entropy, the harder it is to draw any conclusions from the given information.

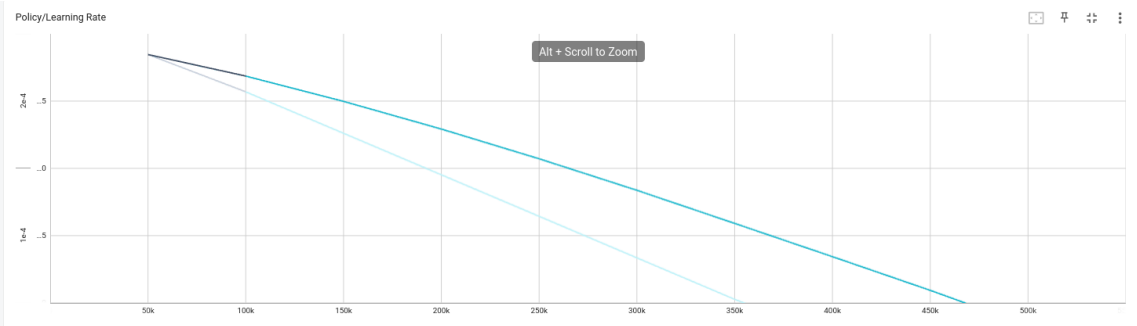

Figure 30: Learning Rate

Figure 30 representes how large a step the training algorithm takes as it searches for the optimal policy. Should decrease over time, which is exactly what happened in the Tower Defence game.

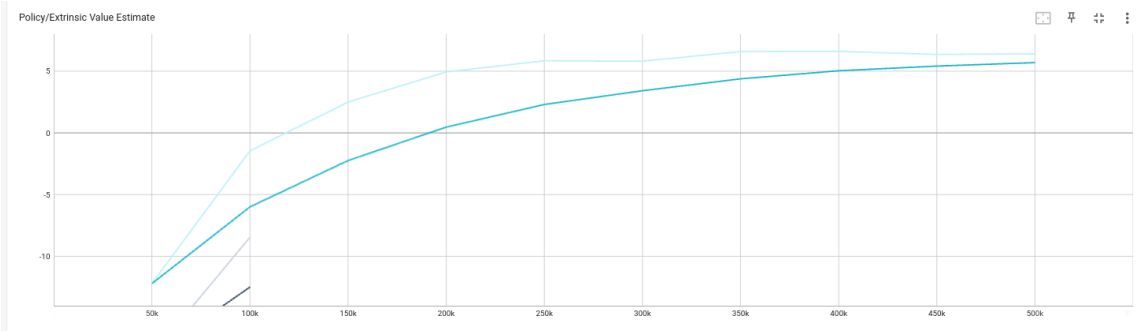

Figure 31: Value Estimate

The mean value estimate for all states visited by the agent. Should increase during a successful training session as the cumulative reward increases, which can be seen in figure 31. They correspond to how much future reward the agent predicts itself to receive at any given point.

5.2 Mistakes in setting up observations and rewards

As mentioned before in the adding neural networks into the video game section**,** only four observations were getting retrieved for training, namely the number of towers, number of currency, number of lives, and number of empty placements, which were enough to make the agents win with the highest score, but not enough to give a human-like behaviour.

The most noticeable example would be the agent in the final wave, wave ten, (figure 32) having 267 currencies left, and several tier one towers. Moreover, it can be seen that there are only four Rocket Platform towers all of tier one, and only one of Plasma Lance, also of tier one. This is all despite it being possible to upgrade all the towers to the highest tier.

However, the reason the agent decided to take such an action (do nothing) was that it was specified that currencies and the number of towers have no impact on the score, while which it’s understandable from the agent's perspective, it shows the difference between how a typical gamer would play the game to an AI agent.

Figure 32: Tower Defence wave 10

Also as the game score only gets affected by the number of lives left, the agent didn't try to destroy all the enemy tanks as far away from the base, with his only concern being finishing the game with all ten of his lives left.

Had a reward been for destroying the enemy as far away from the base as possible, something which most human players would have preferred to do, the observations would have changed as well, and the last two issues wouldn't have appeared during the training and the agent would have used all of his money to buy new towers, while upgrading all of them, especially the ones closer to the starting point. See figure 33.

Another issue that was discovered by the Tensorflow board is the increase of the Entropy over time, which reveals that the agent randomness continued even after managing to reach the final and optimal reward. The way that would have had us avoid that would be adjusting the hyperparameters, especially the beta.

Figure 33: Tower Defence wave 6

5.3 Unexpected problems

Several issues arose during the installation of python, Unity, and the rest of the software applications; however, they were due to permissions. Moreover, due to the longevity of the development process, several software, namely Unity3D and Tensoflow changed their versions, which led to complications to resume the development process and the integration of the software with each other.

When it comes to the development part, an issue was found with the agent's ability to place the towers in the grid placements. This is due to the gird's size being 3x8 while the single tower is at 2x2, meaning a tower can be placed in a different format, which will affect the placements of the future towers. See figure 34.

This led to the agent getting confused when placing towers in the grid and as a result, sometimes it would place just two or three towers in the grid when the maximum can reach up to four towers per grid.

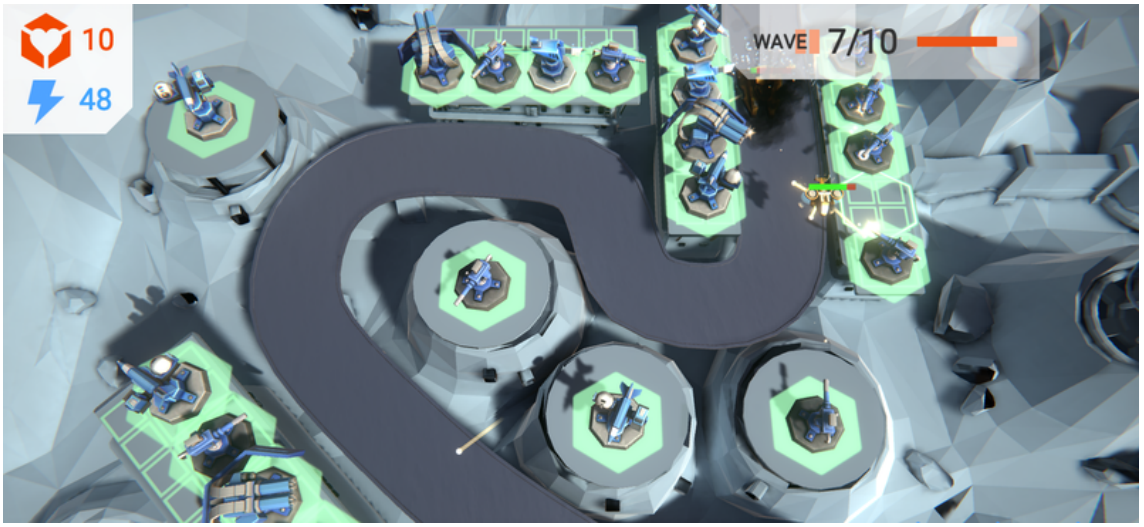

Figure 34: Tower Defence wave 7

After long research, the solution was found to be specifying four extra single-placement tiles over the grid, as that would force the agent to place the tiles in a 2x2 format and wouldn't lead to a situation in which the agent places just two or three agents because of bad placements

5.4 Areas to Improve and future work

As mentioned before, while the agent managed to achieve the highest score possible, his behaviour wasn't fully similar to that of a human player. This is due to the way the observations and rewards have been set, and its only concern is to achieve a three-star rating as seen in figure 35.

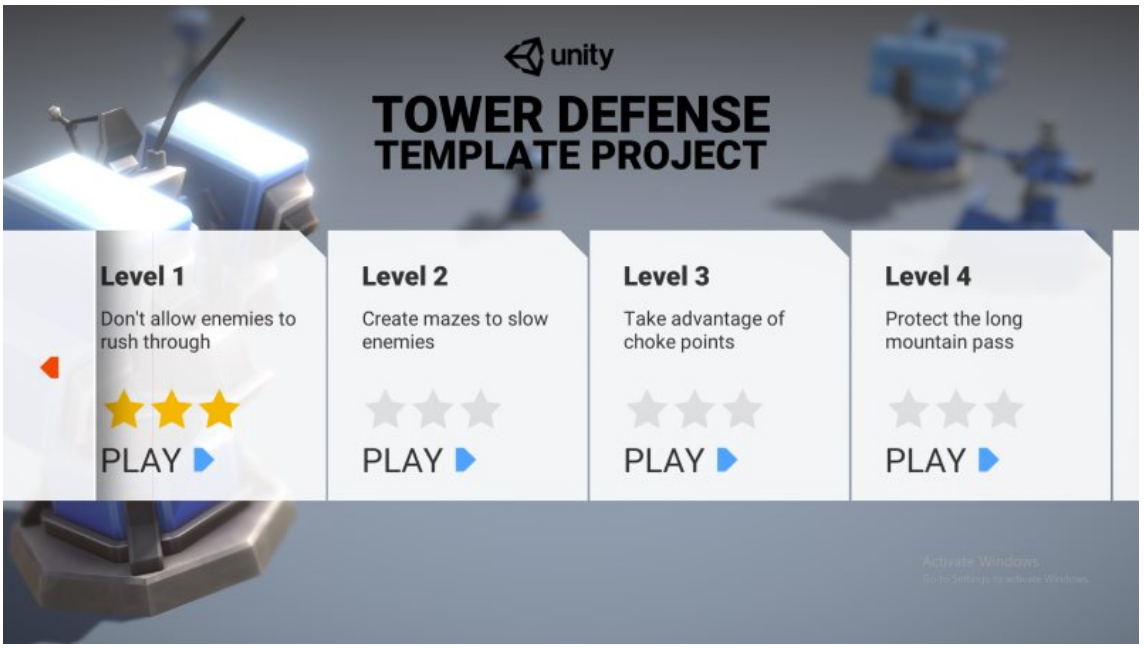

Figure 35: Tower Defence score

Therefore, new observations and rewards should be added based on the distance between the start point and the base as it would make the agent try to destroy the enemies as away from the base as possible, which will most humans will do in a Tower defence game.

Chapter 6: Conclusion

This chapter will conclude the study by summarising the key research findings in relation to the research aims and questions. It will also review the limitations of the study and propose opportunities for future research.

The purpose of the project was aimed at creating integrating the Reinforcemet Learning algorithm in a Tower Defence video game, with the following objectives being met:

- Research and fully apprehend the Reinforcement Learning algriothim.
- Restructure the Tower Defence video game to be AI-integeration ready.
- Integrate the Reinfrocemet Learning to the game.
- Create a well-structured agent to be able to win the game with the highest score.

Further findings show that while Reinforcement Learning and Machine Learning in general have developed a lot recently, it’s quite obvious that an agent would still not be able to 100% imitate a human-being. While, yes the agent managed to finish the game with the highest score, the little challenges a normal player would do, the agent didn’t consider as it’s only goal was to win the game.

This could be an argument of why Machine Learning would never be able to 100% replace humans despite being fast, more accurate, and consistently rational, but they aren't intuitive, emotional, or culturally sensitive; however, they will for sure help make humans smarter and more efficient.

Moreover, I explored different aspects of the Reinforcement Learning, including the Markov-Decision Process as well as the Q-Learning algorithm, with the latter being used in a clear and simple example that it would act as a starting points for anyone interested in learning the topic.The topic of Reinforcement Learning with Neural Network was also given the highlight with some examples.

Some development using C# and Unity3D was made during the work of that project, alongside some solutions to future problems that would face the developers when it comes to setting up the game for Artificial Intelligence, and in doing so, this report could act as a reference for any future work related with Machine Learning in game development.

Regarding improvements and future works, extra observations and rewards should be considered by the developer in order to make the agent more human-like as it still has the limitation that its actions are made with a logic not comparable to the one of a human.

As an improvement proposal, the developer should consider observations in regards of a normal gamer that would set personal challenges for himself aside from the game score.

Summarizing on a personal level, the project has taught me a lot regarding Machine Learing in general and Reinforcement Learning in specific, while also giving me an insight on how game development is done. Moreover, it helped me learn the importance goals-seeting, deadline meeting, communication, research, and working under pressure.

Chapter 7: Bibliography

[1] Bhatt, S. (2019, April 19). *Reinforcement learning 101*. Medium. https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292

[2] Artificial neural network. (2022, September 13). In *Wikipedia*. https://en.wikipedia.org/wiki/Artificial_neural_network

[3] Jagtap, Rohan. "Understanding Markov Decision Process (MDP)." *Medium*, Towards Data Science, 10 Feb. 2021, https://towardsdatascience.com/understanding-the-markov-decision-process-mdp-8f838510f150.

[4] freeCodeCamp.org. "An Introduction to Q-Learning: Reinforcement Learning." *FreeCodeCamp.org*, FreeCodeCamp.org, 9 Aug. 2018, https://www.freecodecamp.org/news/an-introduction-to-q-learning-reinforcement-learning-14ac0b4493cc/.

[5] "A Beginner's Guide to Q Learning." *KDnuggets*, https://www.kdnuggets.com/2022/06/beginner-guide-q-learning.html.

[6] Chandrakant, Kumar. "Reinforcement Learning with Neural Network." *Baeldung on Computer Science*, 20 June 2022, https://www.baeldung.com/cs/reinforcement-learning-neural-network.

[7] Ben Dickson, et al. "A Gentle Introduction to Model-Free and Model-Based Reinforcement Learning." *TechTalks*, 13 June 2022, https://bdtechtalks.com/2022/06/13/model-free-and-model-based-rl/.

[8] Ben Dickson, et al. "Why Lifeless AI Is Not Intelligent." *TechTalks*, 2 Dec. 2021, https://bdtechtalks.com/2021/11/15/birth-of-intelligence-book-review/.

[9] AurelianTactics. "Understanding PPO Plots in Tensorboard." *Medium*, Aureliantactics, 13 Dec. 2018, https://medium.com/aureliantactics/understanding-ppo-plots-in-tensorboard-cbc3199b9ba2.

[10] “Introducing: Unity Machine Learning Agents Toolkit.” *Unity Blog*, https://blog.unity.com/technology/introducing-unity-machine-learning-agents.

[11] Unity-Technologies. “Unity-Technologies/ML-Agents: The Unity Machine Learning Agents Toolkit (ML-Agents) Is an Open-Source Project That Enables Games and Simulations to Serve as Environments for Training Intelligent Agents Using Deep Reinforcement Learning and Imitation Learning.” *GitHub*, https://github.com/Unity-Technologies/ml-agents.

[12] Markov decision process. (2022, August 24). In *Wikipedia*. https://en.wikipedia.org/wiki/Markov_decision_process

[13] Tensor Processing Unit. (2022, September 9). In *Wikipedia*. https://en.wikipedia.org/wiki/Tensor_Processing_Unit

[14] “Guide ： Tensorflow Core.” *TensorFlow*, https://www.tensorflow.org/guide.

[15] Technologies, Unity. “Welcome to the Unity Scripting Reference!” *Unity*, https://docs.unity3d.com/ScriptReference/.

[16] Tower defense. (2022, September 8). In *Wikipedia*. https://en.wikipedia.org/wiki/Tower_defense

[17] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. *arXiv*. https://doi.org/10.48550/arXiv.1707.06347

[17] “Welcome to Python.org.” *Python.org*, https://www.python.org/.

[18] “Conda¶.” *Conda*, https://conda.io/en/latest/.

[19] Reinforcement learning. (2022, September 16). In *Wikipedia*. https://en.wikipedia.org/wiki/Reinforcement_learning

[20] Machine learning. (2022, September 21). In *Wikipedia*. https://en.wikipedia.org/wiki/Machine_learning

[21] “Tower Defense Template: Tutorial Projects.” *Unity Asset Store*, https://assetstore.unity.com/packages/essentials/tutorial-projects/tower-defense-template-107692.

[22] Sarker, Iqbal. (2021). Machine Learning: Algorithms, Real-World Applications and Research Directions. SN Computer Science. 2. 10.1007/s42979-021-00592-x.

Appendix

Code block 5: Spawnmanager class

```
public class SpawnManager : MonoBehaviour
   {
        [SerializeField]
        public Tower[] Towers;
        [SerializeField]
        public TowerPlacementGrid[] TowerPlacementGrids;
        [SerializeField]
        public SingleTowerPlacementArea[]
singleTowerPlacementAreas;
        [SerializeField]
        public AgentAI agentAI;

        private List<Tower> TowersSpawned;
        public int state = 0;
        private bool done = true;

        void Start()
        {
            TowersSpawned = new List<Tower>();
        }
        private void Update()
        {
            if (LevelManager.instance.levelState ==
LevelState.Win)
            {
                done = true;
                gameWin();
            }
            if (LevelManager.instance.levelState ==
LevelState.Lose)
            {
                done = true;
                gameLose();
            }
            if (LevelManager.instance.levelState ==
LevelState.Building && done)
            {
```

```
                Debug.LogWarning("press Start Wave");
                startWaves();
                done = false;
            }
        }
        public void gameWin()
        {
            agentAI.gameWin();
        }

        public void startWaves()
        {

GameObject.FindGameObjectWithTag("StartWaveButton").GetComponent<
Button>().onClick.Invoke();
        }
        public void gameLose()
        {
            agentAI.gameLose();

        }
        public int GetCurrency()
        {
            return LevelManager.instance.currency.currentCurrency;
        }
        public int GetTowerCount()
        {
            return TowersSpawned.Count;
        }
        public int GetLifes()
        {
            return
(int)LevelManager.instance.GetAllHomeBasesHealth();
        }

        public void TowerPlacementAI(int gridIndex, int
towerIndex)
        {
            Utilities.IntVector2 newTowerPosition = new
Utilities.IntVector2(0, 0);

            bool successfulPurchase = false;
```

```
            if(TowerFitStatus.Fits ==
singleTowerPlacementAreas[gridIndex].Fits(newTowerPosition,
Towers[towerIndex].dimensions))
            successfulPurchase =
LevelManager.instance.currency.TryPurchase(Towers[towerIndex].pur
chaseCost);

            if (successfulPurchase)
            {
                if (TowerFitStatus.Fits ==
singleTowerPlacementAreas[gridIndex].Fits(newTowerPosition,
Towers[towerIndex].dimensions))
                {
                    Tower newTower =
Instantiate(Towers[towerIndex]);

newTower.Initialize(singleTowerPlacementAreas[gridIndex],
newTowerPosition);
                    TowersSpawned.Add(newTower);
                }
            }
        }
        public void upgradeTowerAI(int index)
        {
            if(index < TowersSpawned.Count)
            {
                int newLevelCost =
TowersSpawned[index].GetCostForNextLevel();
                if (newLevelCost != -1 &&
LevelManager.instance.currency.CanAfford(newLevelCost))
                {
                    TowersSpawned[index].UpgradeTower();
                    agentAI.SetAgentReward(5);
                }
            }
        }
    }

}
```