2019

Reinforcement Learning: Q-learning in Third Person Shooter Games

Juan Ramon Aguero Quinteros
The College of Wooster, jagueroquinteros19@wooster.edu

Reinforcement Learning: Q-learning in Third Person Shooter Games

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in the
Department of Computer Science at The College of Wooster

by
Juan Aguero Quinteros
The College of Wooster
2019

Advised by:

Nathan Sommer (Computer Science)

THE COLLEGE OF
WOOSTER

Abstract

The purpose of this study is understanding Q-learning through the theory that structures Reinforcement Learning and implementing Q-learning in a Third Person Shooter game. Markov Decision Processes are used as frameworks to solve Reinforcement Learning problems. The methods that are covered in this research are categorized into Dynamic Programming, Monte Carlo, or Temporal Difference. The three methods are used to derive the structure of Q-learning. The Third Person Shooter game is created using the Unity Game Engine. The game is modified to fit the Q-learning method and train the game enemy to play against the player. The training was performed over 6 weeks and results showed a satisfactory increase in the AI performance over the weeks. The results demonstrated that Q-learning successfully improved the enemy AI through a better approximation of the optimal policy.

This work is dedicated to my family for always supporting me unconditionally.

Acknowledgments

I would like to thank the Computer Science Department for making the last four years a great experience. I would like to acknowledge the Computer Science professors for giving me the inspiration that led me to choose this major. I would also like to thank the friends I made over my time here for helping me grow in many aspects.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Games have always been a platform in which many fields apply their methods. One of these fields is Artificial Intelligence (AI). AI aims at creating a game where characters behave similarly to how human controlled characters behave. As the games grow in complexity, explicit AI implementations become less accurate. Machine Learning (ML) is a sub-field of AI; it attempts to teach the AI how they ought to behave based on training data. Among the ML approaches is Reinforcement Learning (RL), a ML method where AI learns by having an agent interact with an environment. One specific RL algorithm is Q-learning, an off-policy Temporal Difference Method. This research is intended to understand Q-learning and observe its implementation in a Third Person Shooting game. The goals we expect to meet are:

1. Understand the theory behind Q-learning.
2. Create a game suitable for Reinforcement Learning using the Unity Game Engine.
3. Teach the game AI by implementing Q-learning.

Chapter 2 will cover the background necessary to understand the theory behind Q-learning. It starts with a comparison between three different ML approaches. These are supervised learning, unsupervised learning, and reinforcement learning.

RL is explained by introducing Markov Decision Processes (MDPs) as a framework to solve RL problems. The MDPs are expanded to introduce Dynamic Programming methods, Monte Carlo methods, and Temporal Difference methods to solve RL problems. The last part introduces Q-learning with an example. Chapter 3 covers the use of the Unity game engine to create a Third Person Shooter game. This is followed by applying the RL theory to adapt the game into a MDP model. Lastly, Q-learning is implemented in the game. Chapter 4 covers the results obtained from training. This is followed by the research conclusion and future work.

CHAPTER 2

Background

2.1 Machine Learning

Machine learning is a growing subject in computer science, and it has been increasing in popularity over the past years. Several factors have contributed to this increase in popularity. These include an increase in computing power, the need for more complex algorithms to perform tasks , and the need to handle large data sets and the results they produce. Machine learning examples are present in the artificial intelligence of automated cars. Other examples include facial recognition in cameras and improving search engine results. These are all models which have large data sets and require pattern recognition.

Machine learning is useful in instances such as web searches or online shopping because the machine AI requires very detailed information about how to process information. This level of detail would result in a very complex algorithm. It is also used in programs that require managing and classifying large amounts of data. Implementing an algorithm that could process all cases and have the desired result would be impossible. Machine learning is also useful when the machine has to adapt to new information, such as a machine that could classify data with little

relation to their current knowledge. This is followed by the machine being able to ignore irrelevant data.

To implement machine learning, algorithms are used to teach the computer how to recognize data. The machines are fed information that will train the machines to classify the information. The input will produce an output which will increase the expertise of the machine. The machine will learn to recognize sets in the data and properly output the desired result, which increases the adaptability of the machine. Furthermore, new data results in increased knowledge of the machine. Machine learning applications are classified in different areas based on the learning approach. The learning approaches are: Supervised learning, unsupervised learning, and reinforcement learning.

2.1.1 Supervised Learning

Supervised learning training is implemented with labeled training data [8]. The process is done by giving the machine examples of data with labels such that it determines different possible classifications it has to learn. Based on the algorithm implemented, the learner improves their knowledge by creating a relation between data and label. Once training is done, the machine is given data with no label and it classifies the data based on the relation it can create between the input and the labeled data. Ideally, if the training data was enough to create a relationship, the machine should be able to classify correctly new input.

Supervised learning is used on image recognition, an example is a machine that identifies dogs and cats in pictures. The training is done by feeding the machine pictures labeled as dog or cat. Once there is enough training data, the machine is given a picture with no label and based on the relation present it should classify it as either dog or cat. However, to improve the classification the labels could include more specific data. Consider any distinctive feature that could separate most dogs

from cats, for example: size, snout, and breed. If the training data also contains information about the three characteristics. The learner, with sufficient training data, will create a mapping such as a big snout for dogs or small size for cats.

Supervised learning is also effective in problems that require prediction of data that is continuous. For example, predicting the income of a person based on their career and age. Using regression methods, the machine can learn to predict the answers. One of the approaches is using linear regression to create a predictive model with the best possible approximation. To predict the income, the training data will consist of sets with career, age, and income values. The machine will learn an approximation of a linear relation between the career, age and income. Once the training is over, the machine will predict income based on career and age input.

2.1.2 Unsupervised Learning

Unsupervised learning, as opposed to supervised learning, is implemented with unlabeled data [8]. The algorithms use training data containing information that represents the attributes present in the data. The machine learns the structure present in the data and creates relations between the data. The relations are arranged into groups based on how similar the data is to a group. One of the approaches in unsupervised learning algorithms is called clustering. These algorithms create groups of data based on how similar they are.

Clustering examples are present in shopping websites. In Amazon, the data is clustered such that the users will browse through content and receive offers of related items. The clusters get more specific as the search criteria is more detailed. Another example is Facebook using face recognition to tag users. When a user uploads a picture, they are asked to tag a specific friend whose face is the most similar to a friend's face.

2.1.2.1 k-means clustering

One of the clustering methods is called *k*-means clustering. In a cluster, a centroid is the data that represents the main member of the cluster. Consider a set of data with n points, k of the data elements are chosen and set as centroids. Each piece of the data set is assigned to its closest centroid and the cluster it belongs to, where proximity is defined by similarities between data. For each cluster, the average between the data is taken and the centroid is moved to that position. The reassigning of data points to clusters and shifting centroids are repeated until the centroids stop moving. Each centroid represents the most similar data to its cluster. When the training ends, the classification of new data is determined by the cluster in which it falls.

An example of *k*-mean clustering is classifying handwritten digits [2]. Consider a data set that represents values of a pixel brightness and an image of $16x16$ pixels. If each pixel is considered as a value that determines similarity between the images, there are in total $16x16 = 256$ different quantities that measure proximity between images. Using *k*-means clustering, we can create clusters for the images that are closely related based on the 256 values in each data.

Another example of clustering is present in Amazon, hierarchical clustering [6]. If you consider the website, there are many categories present in the main page. As the navigation goes deeper into different areas, the shown items start to get more specific to certain area. Hierarchical clustering is similar to *k*-means clustering. The difference is that given n clusters, the algorithm merges clusters that are close to each other. This creates a hierarchy relation between clusters. This can be seen as a tree where the root would be the main page of Amazon. This is useful when constraints are needed on the number of labels that the machine will learn.

2.2 Reinforcement Learning

Reinforcement learning is an area of machine learning that uses a model where an agent interacts with an environment to learn. There is no training data input as there is in supervised and unsupervised learning. There is an agent that chooses from a set of actions and attempts to maximize the reward resulting from those actions. There is no guidance as to which actions result in the most rewards. The agent learns from the experience it receives after choosing different actions. The agent's actions are influenced by the agent's state, and the agent's state also influences its actions. The state represents a condition in which the agent can be in its environment. The agent has a goal or multiple goals that it attempts to achieve, and those goals influence the actions chosen in a given state. Moreover, there are cases in which the agent must consider future rewards when choosing actions. In other words, the objective is to maximize the cumulative reward. Figure 2.1 shows the resulting interaction between the agent and environment.

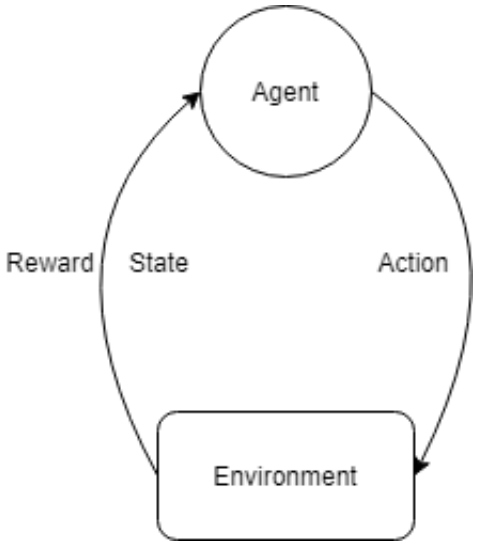

Figure 2.1: Interaction between the agent and the Environment.

Consider a case in which an agent is learning by choosing actions. Since it aims at maximizing the rewards, the agent will always choose the action that rewards

the most. However, suppose there is a state that has an even higher reward than previous rewards. In order to reach that state, the agent must choose an action that does not maximize the reward in the previous state. If the agent always opts for the action with the most reward, the cumulative reward will not be maximized. To address this problem, the idea of exploitation vs exploration is considered in reinforcement learning.

To maximize the reward in a state, the agent must know all the rewards based on the actions in the state. The agent explores different actions that could possibly result in better rewards. As the agent learns more about the rewards, it can exploit its knowledge by choosing the best actions in a given state. As the agent learns from its environment and determines rewards, it requires less exploration. One of the challenges in reinforcement learning is to determine how much an agent should explore or exploit. In most cases, the agent initially focuses on exploring. Once there is progress, the agent's focus changes to exploitation.

There are 4 main subelements in reinforcement learning: a *policy*, a *reward signal*, a *value function*, and sometimes, a *model* of the environment [9].

A policy is a mapping from states to actions. Policies are used to determine what actions are possible in a given state. A policy creates behavior in the agent by using probabilities that help the agent choose actions. A reward signal is what affects the policy when an agent receives rewards from its environment. If an agent receives a low reward in a given state, the reward signal modifies the policy to value other actions in the future. Reward signals are used to shape the goal of the agent. A value in a state represents the total cumulative reward expected in the future. A value function is used to determine the long term rewards an agent can receive; it considers what actions the agent will take as it moves from states. A model is similar to the environment's behavior. The agent uses the model to plan what actions to take before experiencing their rewards. A model can be understood

as pre-established information that the agent acquires before starting the training. This information could be considered assumptions about the environment which influence the actions taken by the agent. Models are used to aim at specific behaviors in the agent. *Model-based* methods are approaches to reinforcement learning that use models. *Model-free* methods are considered the opposite of planning because agents learn from their choices as they experience.

2.2.1 Tic-tac-toe example

A machine can learn to play tic-tac-toe using a value function [9]. Consider an agent-environment set up in which the agent plays against another player. The environment is the table that represents the possible locations to have an X or O. The states are defined as the current arrangement of Xs and Os in the table. The actions are the empty slots in which an X can be written, assuming the agent uses X. The value function is a table where we can store values for each state. These values define the probability of winning the game. For states in which there are 3 aligned Xs the value is 1 since the agent wins the game in those states. For states where there are 3 Os the value is 0. All the other states are set to 0.5 to represent a 50% chance of the agent winning.

The agent chooses his action based on what state it will transition and aiming at the states with value 1. The actions can be chosen *greedily* to reach high value states. Sometimes the agent chooses a random action to explore. The values are updated by setting the value of a state closer to the state it transitioned on whenever a greedy action is chosen to get to that state. This is written as

$$V(s) \leftarrow V(s) + \alpha[\, V(s') - V(s)] \tag{2.1}$$

where s is the state before the action, s' is the state after the action [9]. $V(s)$ is

the estimated value of s which is updated. α represents the *step-size* parameter, also known as learning rate and $V(s') - V(s)$ is a temporal difference learning method. If the learning rate is gradually decreased, the method converges. In other words, the estimates converge to the expected value of the states. Once the expected values are known, there is an optimal policy that the agent will use. If the learning rate is never set to 0, the agent will always have a chance of exploring and adapting to different change in plays.

Reinforcement learning is applied to problems that fit the action-environment model. There is an agent that takes actions and these affect the future outcome. The future states and rewards are a result of past decisions. One main difference is that reinforcement learning evaluates the actions taken while other learning methods instruct the learner by giving it the right action choice. This model of choosing actions based on feedback is represented by the multi-armed bandit problems [9]. The multi-armed bandit is a tabular method, since the states and action space is small enough to fit in tables. These models usually converge to exact values for their optimal policies.

2.3 k-armed Bandit Problem

Given k actions, consider the following problem. After choosing one of the actions a reward is given. Repeat this process n times, also known as *time steps*, with the goal of maximizing the total reward. One of the analogies to this problem is pulling a slot machine lever that grants some reward, known as one-armed bandit; except that there are k levers [9]. Each time a lever is pulled a reward is received. The reward is perceived as positive if it is a jackpot and negative otherwise. After pulling different levers several times, there is an estimate of what rewards should be expected from

pulling one of the levers. If we knew the expected values of each action, solving this problem would be trivial.

Let A_t denote the action taken at time step t and its reward be R_t. Let $q_\star(a)$ denote the expected reward given that a is chosen.

$$q_\star(a) = \mathbb{E}[R_t | A_t = a] \tag{2.2}$$

In this problem, we assume that we are not certain of the values of each action. Since we tried using several levers, there is an estimate of what those values are. Given that those values are stored, we can choose any of the actions based on their estimated value. If actions with high value are chosen we say that it is a *greedy* action and we *exploit*. If other actions are chosen it is *exploring*. Exploitation is good to maximize instant rewards and exploration might result in higher cumulative reward over time. The goal is to get the estimate of an action to be as close as possible to its expected value.

One method would be by having the estimate of an action a be the average of rewards received when choosing a. Let $Q_t(a)$ denote the estimate value of action a, then

$$Q_t(a) = \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{t=1}^{t-1} R_i \cdot \mathbb{L}_{A_t=a}}{\sum_{t=1}^{t-1} \mathbb{L}_{A_t=a}} \tag{2.3}$$

where $\mathbb{L}_{predicate}$ is 1 if *predicate* is true and 0 otherwise [9]. With this approach as the denominator goes to infinity, $Q_t(a)$ converges to $q_*(a)$. This is called a $sample-average$ method for estimating action values. The action selected will be based on some greedy method described as

$$A_t = argmax_a Q_t(a) \tag{2.4}$$

where $argmax_a$ is the action that maximizes $Q_t(a)$. Since just choosing greedy actions will not always yield the best cumulative reward, an approach that allows exploration must be used. Let ϵ be the probability that an action is chosen randomly without considering the action-value estimates. This results in better estimates since in theory all actions will be sampled infinite times under the *epsilon* probability. These methods are called $\epsilon - greedy$ and they allow better convergence.

Since this approach requires that the average of rewards be computed every time a reward is received, it results in too much computation per reward obtained. A different approach is using incremental implementation of action-value estimates [9]. Considering one action, let R_i denote the reward received on the *ith* time this action was chosen, Q_n is the estimate of the action value after it has been chosen $n-1$ times. Then

$$Q_n = \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1} \tag{2.5}$$

would compute the estimate value of an action without considering the new reward. There is a method that updates the average with a small constant computation instead of recalculating the average of rewards. Given Q_n and the R_n, the new average of all n rewards is

$$Q_{n+1} = \frac{1}{n}\sum_{t=1}^{n} R_i = Q_n + \frac{1}{n}[R_n - Q_n] \tag{2.6}$$

With this approach we only need to know Q_n, n and the new reward to compute the estimate [9]. The equation to compute the new estimate of the action value 2.6 is similar to 2.1. These estimates have a general form:

$$NewEstimate = OldEstimate + StepSize[Target - OldEstimate] \tag{2.7}$$

where $[Target - OldEstimate]$ is an error in the new estimate [9]. To approximate the

result to *Target*, we use *StepSize*, a value that scales how much of the error will be used to change the old estimate. The parameter *StepSize* is commonly denoted as α and ranges from 0 to 1. The constant α determines how much weight is given to the new rewards. This can be used to give more weight to recent rewards instead of old rewards. The next section introduces Markov Decision Processes, a problem well suited for reinforcement learning applications.

2.4 Markov Decision Process

There is a decision problem type called *sequential decision problems under uncertainty* [7]. Sequential decision refers to a sequence of actions an agent must take based on the current decision and future decisions. The uncertainty refers to the lack of knowledge about the outcome of all the actions. One of the approaches to uncertainty is maximizing utility from decisions since there is no knowledge about the outcomes of decisions made. This model can be formalized using Markov Decision Processes (MDPs).

A Markov Decision Problem mathematically represents shortest path problems in stochastic environments. MDPs are structured with states that describe an agent's circumstance, actions that affect the process, and rewards obtained after transitions between states. In other words, given a state s, an agent chooses an action a, receives a reward r and transitions to a state s'. To solve an MDP the agent must be influenced to choose actions to maximize the cumulative reward. This can be achieved with policies that define the behavior of the agent. MDPs are useful in reinforcement learning since they provide a framework to solve reinforcement learning problems.

The learner is referred to as an *agent* in a reinforcement learning MDP. The agent is in a world where anything it can interact with is considered the *environment*. The agent and environment interact continuously and the agent receives feedback from

the environment in the form of *rewards*. The elements of the interaction between the agent and environment are represented as *states* and *actions*. The agent chooses which actions to take using *policies*. Policies create behavior by giving the agent information on the available actions to be taken in a given state. The information is represented as the rewards associated with choosing an action. The agent aims at maximizing the cumulative rewards gained from a sequence of actions. The interaction occurs in time steps where each step is the result of an agent choosing an action on a state, receiving a reward from the environment, and transitioning to a new state.

MDPs are very flexible in their application to RL problems. Choosing the agent and its representation in an RL problem can be done by understanding the boundaries between the agent and environment. The boundary is defined around the agent, and anything that cannot be changed arbitrarily by the agent is part of the environment. This relation also allows the use of more than one agent in a problem. MDPs are used to abstract goal-oriented problems by reducing the tasks to three signals that the agent and environment exchange [9]. These three signals are the action signal, the state signal, and the reward signal. The action signal represents the choices the agent makes in the environment. The state signal represents the agent's situation. The reward signal represents the numerical value the agent receives from the environment as a result of its choice.

The use of reward signals to represent goals is one of the most known applications in reinforcement learning. The goal is the maximization of the cumulative sum of rewards received by the agent. The reward signal is a numerical value that the agent receives from the environment at each time step. For example, the learner is trying to learn how to stack blocks of increasing size. A positive reward will be given each time a new block is placed on the stack and does not fall. A negative reward is given if the stack falls. As more blocks fall, the learner stops placing the falling blocks

on top of the blocks from which they fell. This will lead to the learner placing the blocks in increasing order once every block is tested against all the others. There are different methods of representing reward signals, and each method impacts how effective the learning is. Choosing reward signals is more complex because it is an imprecise approach to gaining training results. One cannot know the exact outcomes of the training until it produces results after long periods of time. This is why choosing appropriate reward signals is highly influential on the results.

The blocks example is a reinforcement learning problem that has multiple runs on the same task. Such a task has a series of interactions between the agent and the environment. These series of interactions are called episodes, which are the sequence of rewards obtained each time step. An episode ends in a terminal state in which the task is restarted for another episode. The sequence of rewards obtained is called the return. In this case, the return is the sum of rewards in an episode. The agent's goal is to maximize the return where it can be denoted as:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T \tag{2.8}$$

where R_t is the reward obtained on the time step t.

The agent tries to maximize the return and chooses actions based on their expected rewards. The *discount* term γ is used to represent a scalar that reduces the impact of rewards received in the future. A reward that is expected to be received k steps ahead is γ^{k-1} times what it would be valued if it was received immediately. The use of this term is to manage how the agent considers future rewards when choosing an action. A value of $\gamma = 0$ results in the agent only considering immediate rewards while a value of γ closer to 1 results in the agent weighing the future

rewards higher. The discounted return is:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.9}$$

where $0 < \gamma < 1$.

Once there are rewards present, the agent requires a method to choose which actions to take in a given state to maximize the expected discounted return. The *policy* is a mapping from states to the probabilities of choosing an action in the current state. This is how the agent is able to choose the actions with the best return. If an agent follows a policy π at time t, then the probability that $A_t = a$ and $S_t = s$ is $\pi(a|s)$ [9]. The probabilities in a policy are determined by *value functions* where $v_\pi(s)$ is the expected return of being in state s and following the policy π. The value of taking an action in a specific state using a policy π is $q_\pi(s, a)$ and q_π is the *action value function*. The value functions can be estimated from experience. If an agent is able to sample infinite observations of each state, the average of the values converges to $v_\pi(s)$. Similarly, if averages of each action are sampled infinitely in a state, the values will converge to action values $q_\pi(s, a)$. The use of averages to estimate value functions is known as Monte Carlo methods.

The value $v_\pi(s)$ under a policy π can be formally defined by

$$v_\pi(s) = \mathbf{E}[G_t|S_t = s] \tag{2.10}$$

The action value $q_\pi(s, a)$ under a policy π can be formally defined by

$$q_\pi(s, a) = \mathbf{E}[G_t|S_t = s, A_t = a] \tag{2.11}$$

The value functions can be expressed recursively in terms of the expected discounted return of the future state values. This core property is used in reinforcement

learning to find value functions following the Bellman equation. The Bellman equation is used to find optimal policy in an MDP [3]. This was proved by Bellman by demonstrating the asymptotic behavior of sequences defined by the Bellman equation. Following 2.9 the **Bellman equation** of the value function $v_\pi(s)$ can be expressed as

$$\begin{aligned} v_\pi(s) &= \mathbf{E}[G_t|S_t = s] \\ &= \mathbf{E}[R_{t+1} + \gamma G + t + 1|S_t = s] \end{aligned}$$

This is a relation between the value of a state and the value of future states. We can see that the value of a state s is expressed as the reward after choosing an action plus the expected return of one of the possible states s'. This implies that the Bellman equation considers all possible values of future states to estimate the value of the current state. The **optimal policy** is the policy that maximizes the expected return of all states. This means that if policy π has $v_\pi \geq v_{\pi'}$, where π' represents all the other policies, then π is the optimal policy. The optimal state-value function is defined as

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.12}$$

The optimal action-value function is defined as

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{2.13}$$

Finding the optimal policy is the goal of reinforcement learning. However, in most real cases the state size is too large to find the best policy. This means that with reinforcement learning we approximate optimal policies. These approximations are still useful since agents potentially evaluate the most important states. Consider an agent learning how to play chess that trains against high skill players. Even though

the agent will not learn all the optimal actions in all states, the agent will estimate the most common optimal actions since high skill players will have a similar pattern. This results in agents that can learn to play against such players even though many action choices are not optimal. Next we introduce dynamic programming methods that estimate optimality given perfect knowledge of an MDP model.

2.4.1 Dynamic programming

Dynamic programming (DP) algorithms are the basis to understanding the theory of computing optimal policies. Those algorithms require a perfect MDP model, a model in which all state transition probabilities and rewards are known, to find an optimal policy. DP algorithms are not very practical since they have high computational requirements. Many reinforcement learning algorithms attempt to solve the same problems that DP algorithms address in a less computational approach. The main approach in DP algorithms is to find the optimal policy by using the value functions $v_*(s)$ and $q_*(s,a)$ using the Bellman equation as update rules.

Policy Evaluation is a DP method to estimate the state-value function $v_\pi(s)$ for a policy π. This is done using iterative solution using a sequence of state-value functions v_k where $k = 0, 1, 2, ...$ Since we know the rewards and transition probabilities we can express $v_{k+1}(s)$ as a function of $v_k(s)$ using the Bellman equation as an update rule to determine the state-value function of a policy π

$$v_{k+1}(s) = \mathbf{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] \tag{2.14}$$

This is called iterative policy evaluation and it is used to approximate $v_{k+1}(s)$ by updating values through sweeps on the state values of $v_k(s)$ by considering all future state values [9]. In other words, it updates an old value of s based on all its

successor state values and their expected rewards for every state s. The Bellman update ensures that $v_k = v_\pi$ and it can also be shown that as $k \to \infty$ the sequence converges to v_π. Once we determine the value function v_π for a policy π, we can improve the policy π using an action that does not follow the policy π. Consider a value function v_π and suppose we are currently in the state s. We know which action a is part of the policy π. Choose an action a' that does not follow the policy $v_\pi(s)$ and then follow the policy π. If the action-value $q_\pi(a', s) \geq q_\pi(a, s)$, then we found a better policy π'. The improvement of a policy π given the value function v_π can be expressed as

$$\pi'(s) = arg \max_a q_\pi(s, a) \tag{2.15}$$

This leads us to policy iteration, the use of policy evaluation and policy improvement to find an optimal policy [9]. Consider estimating the value function v_π for a policy π and improving the policy to obtain π', we estimate $v_{\pi'}$ from π' and loop indefinitely. The iteration between policy estimation and policy evaluation converges to the optimal policy π_* and optimal value function v_*. However, policy evaluation can result in unnecessary calculation, an optimal state-value can be achieved before ending the evaluation since the algorithm calculates for all possible state transitions. To address this issue, value iteration is used to estimate $v_{k+1}(s)$. Value iteration consists in changing the policy estimation by computing the maximum over all actions. This results in the following update rule

$$v_{k+1}(s) = \max_a \mathbf{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \tag{2.16}$$

where the policy evaluation is stopped after one update on each action. The dynamic present in the iteration between evaluation and improvement is known as Generalized Policy Iteration (GPI) [9]. The processes alter the value function with respect to the policy and vice versa. This results in a constant change in opposing

directions as the policy estimation causes the policy to not be greedy while the policy improvement causes the value function to be inconsistent with the policy. Convergence is achieved once the values of the value function and policy do not alter. In other words, the optimal value function and optimal policies are found when policy is greedy in relation to its value function.

We can see that DP algorithms update values by sweeping through the states. An interesting property of DP algorithms is that the updates on states are based on future states, which is known as **bootstrapping** [9]. Bootstrapping is common on reinforcement learning algorithms, as estimates are usually based on other estimates. One downside of DP algorithms is that they require a perfect MPD model, an environment model in which the agent knows everything. This is not the case in many reinforcement learning applications, as creating an accurate model is not always possible. Consider a chess example, providing a perfect model of the different state transition probabilities and rewards is unfeasible. A different approach to this problem are Monte Carlo methods, reinforcement learning algorithms that do not require a model and learn from experience.

2.4.2 Monte Carlo methods

Monte Carlo (MC) methods are used to estimate value functions and finding optimal policies. The estimations are based on samples of different sequences of states, actions and rewards. The agents do not have complete knowledge of the environment and they learn from experience. The learning can be from actual experience or simulated models with samples of transitions as opposed to a complete probability distribution. MC approximations are based on averaging sample returns [9]. The updates to the value functions in Monte Carlo are done by learning from sample experiences as opposed to computing as in DP. The GPI still applies to Monte Carlo methods as the policy and value functions interact to reach optimality.

The MC estimation of $v_{\pi(s)}$, the value of a state s following a policy π, is done by computing the average of the returns obtained from sample episodes following policy π. The average of the returns is determined by the number of visits on a state s during an episode. The first-visit MC is an algorithm used to estimate $v_\pi(s)$ given the return following the first visit to a state s, where a visit is defined as the first time the agent is in the state s following a policy π on an episode. It has been shown that first-visit converges to $v_\pi(s)$ as the visits to s go to infinity [9]. This algorithm is useful as it allows achieving an optimal state-value function for a specific s as the estimation can start every episode on s under the policy π. We can also estimate an action-value function $q_\pi(s, a)$ using first-visit MC. This is achieved by averaging the returns following a visit to the state s and action a. Similarly, this approach converges as the visits go to infinity. There are drawbacks in following a policy π as it can be biased into always choosing the same action a on a state s. Exploration is a solution to this, we can specify that the episode starts on a specific state-action pair and the following pairs have positive random probabilities. This results in visits to all state-action pairs as the number of visits goes to infinity.

MC methods can be used to achieve optimal policies following the GPI [9]. The policy estimation is done by experiencing episodes and approximating the value function to the optimal value function. The policy improvement has a different approach in comparison to DP, the action-value function is used to obtain a greedy policy. The resulting policy is the policy that chooses for all s, the action with maximum action-value. This can be expressed as

$$\pi(s) = arg\max_a q(s, a) \tag{2.17}$$

Similar to the DP policy improvement approach, we can obtain a greedy policy

π_{k+1} using the action-value function $q_(\pi k)$ to obtain

$$q_{\pi k}(s, \pi_{k+1}(s)) = \max_a q_{\pi k}(s, a) \tag{2.18}$$

where $q_{\pi k}(s, \pi_{k+1}(s)) \geq v_{\pi k}(s)$ since the value is chosen greedily. This ensures that π_{k+1} is equal or better than π_k. If it is equal, then we have found an optimal policy. This approach assumes infinite episodes, this can be avoided using value iteration between episodes [9]. At the end of each episode, the new returns are used to estimate new action-value functions and the policy is improved on the visited states in the episode. With this method, we can move the value function closer to the optimal value function. This algorithm is called MC Exploration Start and it assumes exploring starts on every episode. Convergence is achieved as the change to the action-value function decreases over the episode. With a fixed bound in the change, one can obtain good approximations to optimal value function and policy. With this approach we assume exploring starts, the ϵ-soft approach is used in such cases.

To estimate optimal policies without exploring starts we can have an ϵ-soft policy on the first-visit MC method [9]. The ϵ-greedy policy initializes π such that a probability of $1 - \epsilon + \frac{\epsilon}{A(s)}$ is given to the greedy action while the other actions are given $\frac{\epsilon}{A(s)}$ probability. This approach weights the action choices such that a non greedy action can be chosen to estimate the new value function. Using an ϵ-greedy policy π, we can improve the policy since any ϵ-greedy policy derived from q_π is equal or better than π. This results in approximating the policy to the greedy policy in each episode but not the optimal policy since the learner still explores. Methods that use policies to update their corresponding value function are called on-policy methods. It is possible to use policies to create behavior that will be used to find an optimal policy, this approach is called off-policy method.

The off-policy method considers two policies, a behavior policy, used to generate behavior, and a target policy, the optimal policy we attempt to obtain [9]. Let b be a behavior policy, we can initialize the probability distribution of b such that all state-action pairs are nonzero. We create samples by following the policy b and update the value function by using weighted returns based on the probability that an action is chosen on the policy b and π.

The main aspect of MC methods in comparison to DP methods is that they do not require perfect knowledge of the model. MC estimates are based on sampling experience and finding the average of the returns for each state-value and action-value. This means that MC methods do not base their estimates on other estimates. In other words, MC methods do not perform bootstrapping. MC methods have the advantage that they can find optimal approximations to their policies by using few samples as opposed to a complete computation using DP. There are methods that combine DP and MC by bootstrapping and using sample experience to estimate optimal value functions and policies, these approaches correspond to Temporal Difference methods.

2.4.3 Temporal Difference methods

Temporal Difference (TD) algorithms use both MC and DP approaches. Similar to MC methods, TD approaches use experience from samples to estimate without the need of a model. TD methods use other estimates to update their estimates without having to wait for the final return. These two factors result in TD algorithms being the most adopted in reinforcement learning as they are simple and easy to implement in real life applications. TD prediction also applies the GPI approach to estimate optimal value function and optimal policies.

The first TD approach can be derived from a combination of DP and MC ideas. The TD(0) prediction uses experience from samples to estimate their value function.

Unlike MC approaches, TD(0) does not need to wait until the end of the episode to update the state-value of a given state s since the updates are based on one step ahead estimations similar to DP methods [9]. Given a state-value $V(S_t)$ and using a MC approach, the estimate of the state-value can be expressed as

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{2.19}$$

where G_t is the return after time t and α is the step-size parameter used to weight the target in approximation. The 2.19 update function requires the episode to end to be able to update the value function. With TD we only need one time step to estimate the state-value since TD(0) determines the target using the reward that follows the state. This is expressed by

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.20}$$

where $R_{t+1} + \gamma V(S_{t+1})$ is the target for TD(0). The difference in $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is due to the error in estimating $V(S_t)$ [9]. This is caused by not having the next state and reward before time $t + 1$.

To estimate optimal policies using TD methods and following the GPI approach, we need to estimate action-value functions. This can be done using Sarsa, an on-policy TD method. To estimate $V(S_t)$ using TD(0) we needed the transition of a state to another. Using Sarsa, we need to consider the transitions from state-action pair to state-action pair to estimate $q_\pi(s, a)$. Using the same approach as TD(0), we can estimate the action-value of $Q(S_t, A_t)$ at time t as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1} - Q(S_{t+1}))] \tag{2.21}$$

The name Sarsa is derived from the elements the update function 2.21 uses, the

elements being $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. Sarsa converges to the optimal policy and action-value given that the state-action pairs have infinite visits [9]. If the policy chosen is ϵ-greedy, the policy converges to the limit of the greedy policy.

TD algorithms have the benefit of quick estimations and simplicity in implementation. They combine ideas of MC approaches by learning from experience and DP approaches by estimating values based on other estimates. TD methods estimate value functions on each time step and use a step-size parameter to move the result towards a defined target. The TD(0) prediction is used to estimate state-value functions by looking at state to state transitions. The Sarsa method estimates the action-value function by looking at action-state pair to action-state pair transitions using a policy π. The next section introduces an off-policy TD algorithm of interest in this research, Q-learning.

2.5 Q-learning

One of the most famous reinforcement learning algorithms is Q-learning [12], an off-policy Temporal Difference algorithm that estimates the action-value function. Similar to the TD methods mentioned, Q-learning implements a combination of Dynamic Programming estimations and Monte Carlo sample experience learning. The advantage of Q-learning over TD(0) or Sarsa method lies on the off-policy approach it offers, optimal policies can be found with randomized behavior policies. Watkins has demonstrated that the algorithm converges to the optimal policy given than the action-value pairs are continuously updated [12].

Similar to the Sarsa method, Q-learning estimates the action-value $Q(S_t, A_t)$ with the difference that it uses the discounted estimate of the future highest valued action-pair $\max_a Q(S_{t+1}, a)$. This results in incremental updates that drive the Q-values $Q(S_t, A_t)$ closer to the optimal action-value function q_* [12]. The Bellman updates

ensures that there is an optimal policy that can be achieved [12]. The Q-learning method can be expressed as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.22}$$

where the Q-value $Q(S_t, A_t)$ at time t is the incremental update of the Q-value $Q(S_t, A_t)$ with the α weighted target that contains the immediate reward R_{t+1} plus the discounted future state's maximum valued state-action pair $\gamma \max_a Q(S_{t+1}, a)$ minus the Q-value $Q(S_t, A_t)$. One property of Q-learning is that the Q-values can be initialized to random states and the algorithms still converge to an optimal policy [12]. This caused by the greedy approach on the maximization of the target estimates. Consider the following example of Q-learning application.

Suppose there is a bridge that a person wants to cross to reach the next town. The bridge contains a few holes that the person has to avoid in order to fully cross the bridge. Additionally, the bridge is wide enough such that the person can only walk on a straight line when crossing the bridge. We can convert this situation in a reinforcement learning problem and apply Q-learning to it. The person becomes the agent, lets assume the bridge can be divided in sections and the agent can walk forward when moving through the next section or jump when moving through the next section. Furthermore, we assume that the person does want to spend the least effort. In other words, the person wants to cross the bridge by jumping the least.

We can construct a two dimensional table in which each entry represents the Q-value of choosing an action a in a state s. In this case, the states are the sections of the bridge and a state s_k is the section k. The starting state is at the beginning of the bridge and final state at the end of the bridge. With this information, we can create a system of rewards to allow the agent to learn an optimal policy with the behavior

we want it to learn. In this case, an agent that only jumps holes and reaches the other side of the bridge.

With the goal in consideration, let's build a system of rewards. The two objectives are to avoid the holes in the bridge and jump the least. If the agent chooses to walk forward and falls in a hole, the reward R_{t+1} will be negative. This allows the Bellman updates to lower the Q-value $Q(S_t, A_t)$ of choosing the action of walking forward when moving into a section with a hole. If the agent chooses to walk forward and there is no hole, the reward will be positive. This allows the agent to move forward when there is no hole in the next section. If the agent jumps through the next section and it had no hole, the reward will be negative. This results in the agent jumping only when necessary. Lastly, if the agent chooses to jump and there was a hole in the next section, we give the agent a positive reward. Figure 2.2 shows the transition from section to section and rewards obtained given the action chosen. With these four rewards, we can ensure that the agent:

1. Walks forward when there is no hole in the next section of the bridge.
2. Jumps forward when there is a hole in the next section of the bridge.
3. Jumps only when there is a hole in the next section of the bridge.

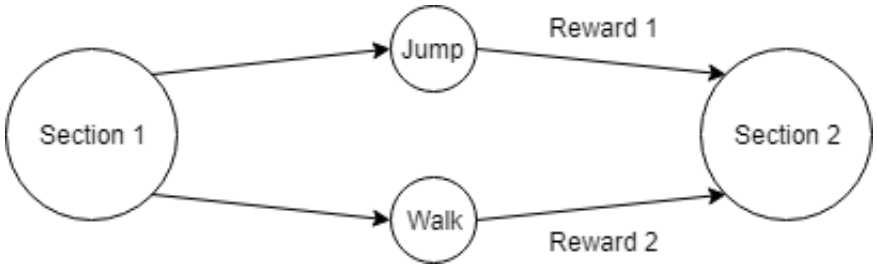

Figure 2.2: Transition from sections with rewards corresponding actions.

With this representation, lets assume that we have 4 sections in the bridge. This yields a Q-table with 8 entries, one entry for each section-action pair. Table 2.1 shows

Q-table	Action = jump	Action = walk
State = Section 1	0	0
State = Section 2	0	0
State = Section 3	0	0
State = Section 4	0	0

Table 2.1: Q-table with all values initialized to 0.

Q-table	Action = jump	Action = walk
State = Section 1	-	0
State = Section 2	0	+
State = Section 3	0	-
State = Section 4	+	0

Table 2.2: Q-table after one episode.

the Q-table with values initialized to 0. Lets assume the section 3 and 4 have holes. Suppose the following sequence of actions has been taken: *jump*, *walk*, *walk*, *jump*. Considering our reward system and knowing where the hole is, Table 2.2 shows the changes in signs of the Q-values with respect to the rewards obtained. This problem has two actions per state. This will cause the optimal policy approximation converges fast since optimal actions can be determined by visiting one state-action pair. If we were to choose greedy actions in the second episode, the policy followed would be the optimal policy.

Q-learning is an off-policy reinforcement method, it estimates approximation to optimal policies without a model. Q-learning belongs to the Temporal Difference methods of reinforcement learning, algorithms that combine the bootstrapping present in Dynamic Programming and sampling experience present in Monte Carlo methods. Markov Decision Processes are the basis of implementing Reinforcement Learning methods, they offer a framework in which RL methods can find optimal results. Games have a MDP representation as they can be transformed into sequences of actions in a finite state space where rewards shape the goal of the game. The next

chapter covers the creation of a Third Person Shooter game and the implementation of Q-learning to train and agent and observe game play results.

CHAPTER 3

My Work

This chapter presents the process in implementing Q-learning in a third person shooting game. Unity is a game engine that can be used to create third person shooting games. The game used in this study was created by expanding a tutorial found on the Unity website. Next, we cover what we expect the game to contain and how we achieve those expectations using Unity. Lastly, the software implementation is presented with detailed information about how the training environment is set up by modifying and adding Reinforcement Learning functions to the game.

3.1 Unity software

Unity is a game engine that facilitates creating projects that require graphical displays of scenes with elements in it. Figure 3.1 is the main interface that displays the working scene and tools. These tools can be used to alter and manipulate objects and their properties in the scene as shown in Figure 3.2. Initially, users might consider the interface complicated due to many tools being displayed, however, with a few days of practice the user becomes familiar with it. The game engine also provide scripts to create behavior in the scenes displayed. The Unity website provides tutorial videos on implementing scripts and using their tools. Their website also includes description of their implemented methods and examples of

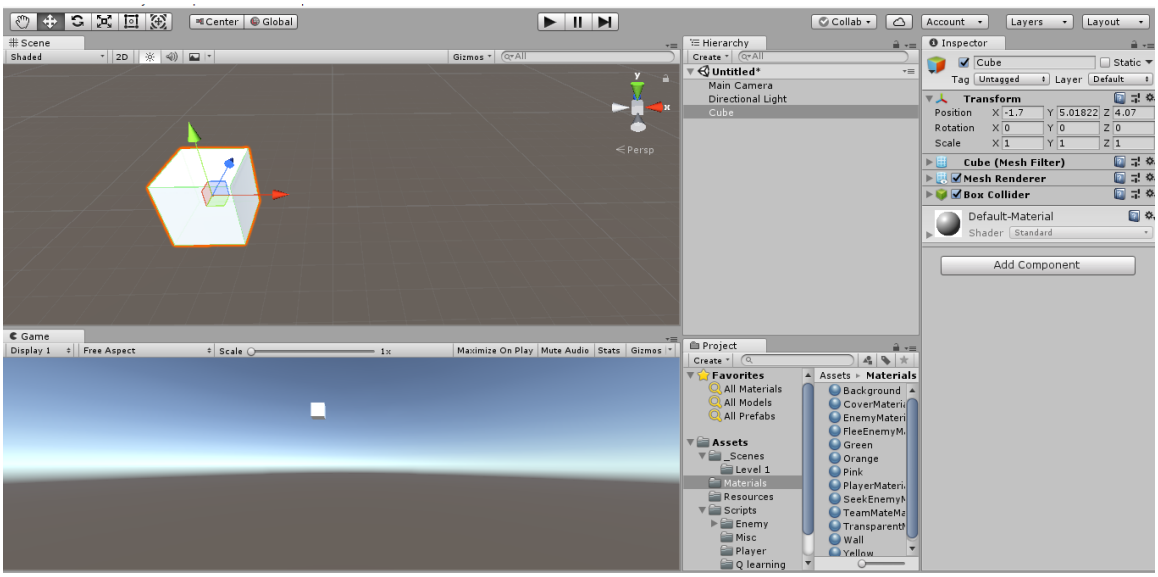

Figure 3.1: The main display shown when a Unity project is open.

how to use them. Unity also includes an asset store where users can download or buy premade elements to be used in the scene. The asset store provides graphically interesting objects but the game is based only on elements present by default in the game engine. Among the tools present in Unity, the most used in the game implementation are game objects, scripts, and physics.

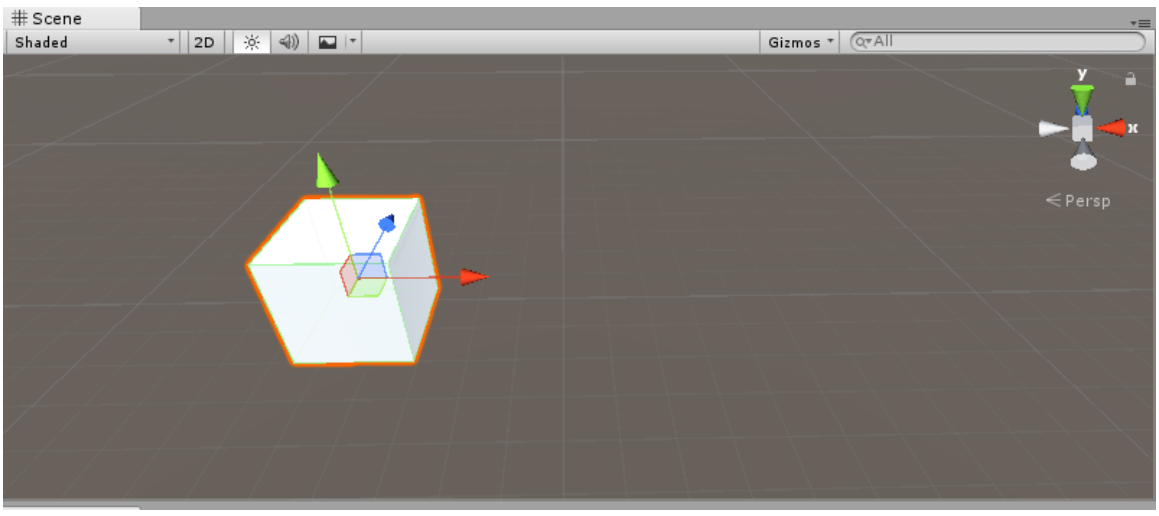

Figure 3.2: Unity Scene window

3.1.1 Game Objects, Rigid Body, and Scripts

Game objects are the entities that constitute a scene in Unity. Game objects come in different types, some of them are 3D objects, cameras, light, wind, audio, including UI elements. The game engine includes 3D Object prefabs of common shapes that would be the basis to the structure of a 3D object in the scene. The prefabs used in the game are three: cubes, spheres, and terrains. Objects can be created through the menu and they can have a hierarchical structure, it is possible to attach an object as a child of another object. This results in a unified object with the property that any change in the parent affects all the children attached to it. One of the benefits of this hierarchical structure is that one can build different 3D objects with individual parts that can behave independently.

Once objects are created, they are added to the hierarchy window shown in Figure 3.3. The hierarchy window contains the list of all game objects present in the current scene. If an object is selected, their components are shown the inspector window shown in Figure 3.4. Among the components present in an object, the most used in the game are Transform, Rigid Body, and Scripts. Transform refers to any property of the object in relation to position, size, and rotation. Rigid Body is the component with properties related to physics such as mass, gravity or collision detection. Scripts can be attached to objects to create behavior. A script consists of coding in either JavaScript or C# to use functions that Unity offer. Most component properties can be changed directly through the inspector window. A different way of altering these components is using scripts. The next section introduces the game we will use in the Q-learning implementation.

Figure 3.3: Hierarchy window containing objects present in the scene.

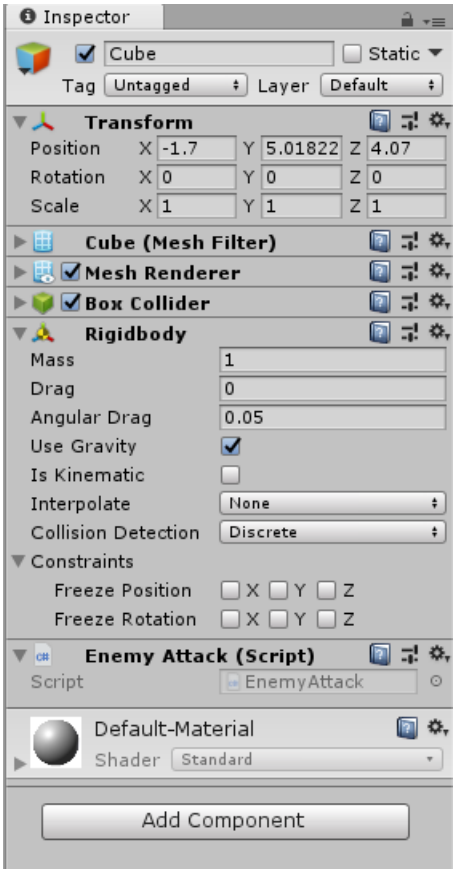

Figure 3.4: Inspector window with component properties of the selected cube object.

3.2 Third Person Shooter Game

Our goal is to implement Q-learning in a reinforcement learning problem and observe training results. Among the different problems that can be represented as a reinforcement learning problem are games. Games have a structure that can be reduced to an agent, an environment, and a goal. These three elements are our basis to formulate a reinforcement learning problem. Unity offers a valuable platform to create an environment in which we can implement the Q-learning algorithm. In this research the game of choice is a third person shooter (TPS) game. TPS games were chosen since they offer a better visual environment to create interesting reinforcement learning problems. The elements we need to create TPS are a main character, an enemy, and an environment in which the player and enemy interact. Our goal is to have a game with the following elements:

1. The player controlled character can move, shoot, and take damage from enemy shots.
2. The enemy can move, shoot, and take damage from player shots.
3. The player and enemy are inside the environment.
4. The environment is enclosed.
5. The goal of the player is to score the highest points by eliminating enemies.

3.2.1 The Main Character

The main character in a TPS game is usually controlled by the player. To have a player that fits our goal we need to further develop its properties:

1. The player should be able to move around the environment given that there are no obstructing objects in the path.

2. The player can shoot at any direction based on mouse input.

3. The player takes damage if hit by the enemy projectiles.

First, we need to create a 3D object that will represent the player. Using two 3D cube objects we can structure the player. The first cube is left in its original form. We change the transform values of the second cube such that the cube becomes elongated and smaller. This second cube will represent the gun and is attached as child of the first cube. We can add the material component to choose a color for the player. The blue color is chosen since it is common for the main character to be associated with the color blue. Figure 3.5 shows the final result of the player in the scene view. The player requires a projectile that is created when the player shoots. We create a 3D sphere object and change its transform scale parameter to make it smaller.

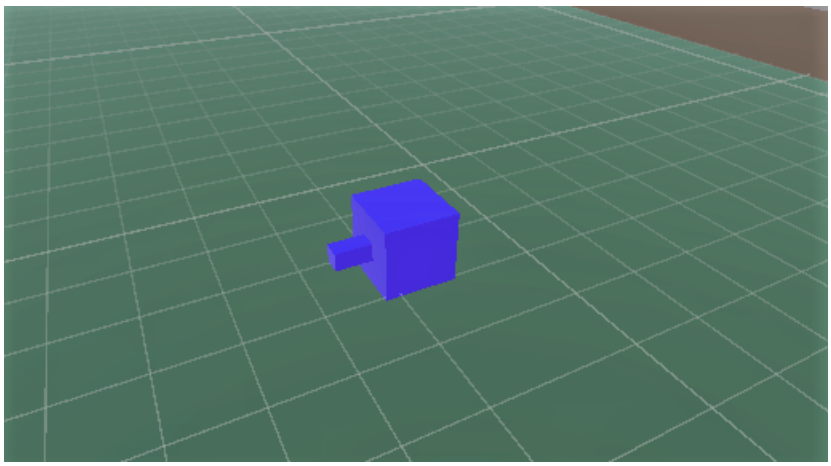

Figure 3.5: Player object in scene window.

Once we have a player object, we can create behavior in the player using scripts. The player has to shoot in the direction the cursor is pointing if the right mouse button is clicked. We create a script called **PlayerAttack** to manage shooting and aiming directions. The script has two methods: (1) A method that always aims the gun at the cursor, (2) A method that shoots a projectile if the left mouse button

is pressed. The first method is accomplished by changing the rotation transform property of the gun to always point at the cursor. This is possible since Unity has a method referencing the cursor's position. The second method is accomplished by using a Unity function that triggers if the mouse button is pressed. We can instantiate a projectile in front of the gun and set it to move towards the direction the gun is pointing at. Figure 3.6 shows the projectile that is shot once the player clicks the left mouse button.

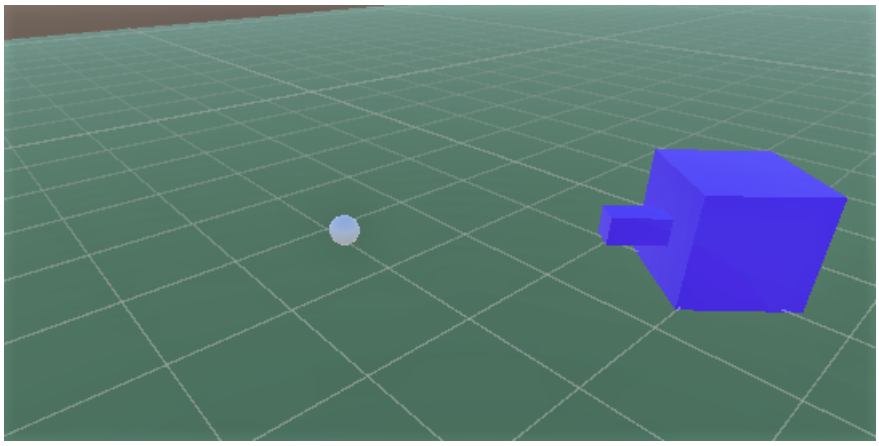

Figure 3.6: Player shooting a projectile.

To create movement, we create a second script **PlayerController** and attach it to the player object. The script has one method that takes the WASD keys as input and moves the player towards one of the four cardinal directions. This is possible by attaching a rigid body component to the player and using physics methods in the script. Lastly, we create a third script **PlayerHealth** that will handle the player taking damage. The script has a private integer that stores the current player health. The script has two methods: (1) A method that determines using a boolean if the player health has not fallen below 0. If the player health falls below the threshold, the player object is destroyed. (2) A method that decreases the player health if there is collision between the player and an enemy projectile.

We have a fully functional main character. The player can be moved using the

WASD keys as input. The player can shoot at the cursor's direction if the left mouse button is pressed. And the player can take damage if it is hit by an enemy projectile and gets destroyed if the player health falls below 0. The next element we need for the TPS game is an enemy with similar capabilities to the player.

3.2.2 The Enemy

The enemy is the second major element of our TPS, it has similar properties as the player. We can list the properties the enemy should attain:

1. The enemy should be able to move around the environment given that there are no obstructing objects in the path.

2. The enemy shoots at the player's direction.

3. The enemy takes damage if hit by player projectiles.

Similar to the player, we build the enemy with the same structure as the player. Figure 3.7 shows the enemy character with a red color as the material component. The enemy movement is implemented using the script **EnemyController**. The script has one method that uses physics to move the enemy, it randomly chooses a direction and the enemy object is moved towards the chosen direction using its rigid body component. The second script **EnemyAim** contains one method: The player position is always known for the enemy through the player transform component. The method alters the enemy gun transform rotation to aim the gun towards the player. This results on the enemy always aiming at the player. The last script **EnemyAttack** is implemented similarly to the **PlayerAttack**. The method present in the script causes the enemy to shoot a projectile in the direction the enemy gun is aiming. The method uses a coroutine, in Unity a coroutine can be used to delay returns. This can be useful to create intervals between method calls. The coroutine

creates intervals between shooting a projectile and shooting the next projectile. This results in a close approximation to how often an enemy ought to shoot. Lastly, the enemy health is implemented similarly to the player health using a script called **EnemyHealth**. The script contains the same approach as **PlayerHealth**

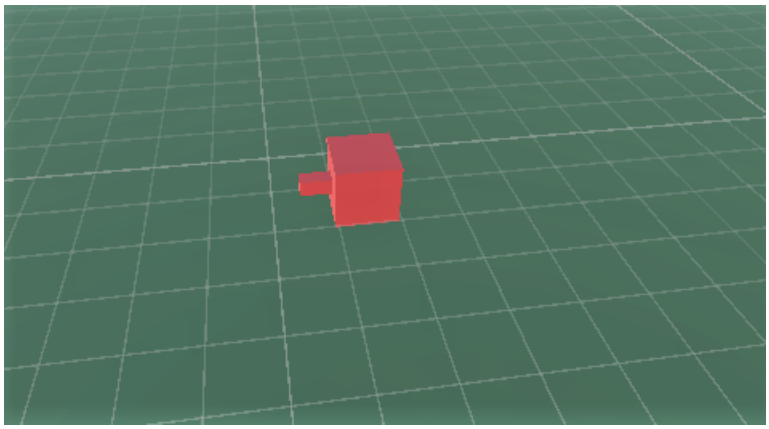

Figure 3.7: Enemy character.

The player and enemy met the goals we listed in the beginning of the section. The enemy has the same capabilities as the player with minor handicaps to avoid making the game difficult. This is because the player will be facing many enemies at the same time. Once the two characters are set up, an environment is needed for the player and enemy to interact.

3.2.3 The Game Environment

The game environment represents a physical environment where the player and enemies interact. Unity allows an infinite expansion of the world in which the player and enemy are. However, it is required that the environment be bounded to derive a finite MDP and apply Q-learning. The physical environment is built from the Unity Terrain object. The terrain object is a two-dimensional object. Once the terrain is set, cube objects are added to the game and altered such that they are

expanded to create walls that encompass the area in which the player and enemies are. Figure 3.8 shows the closed environment where the player and enemy interact.

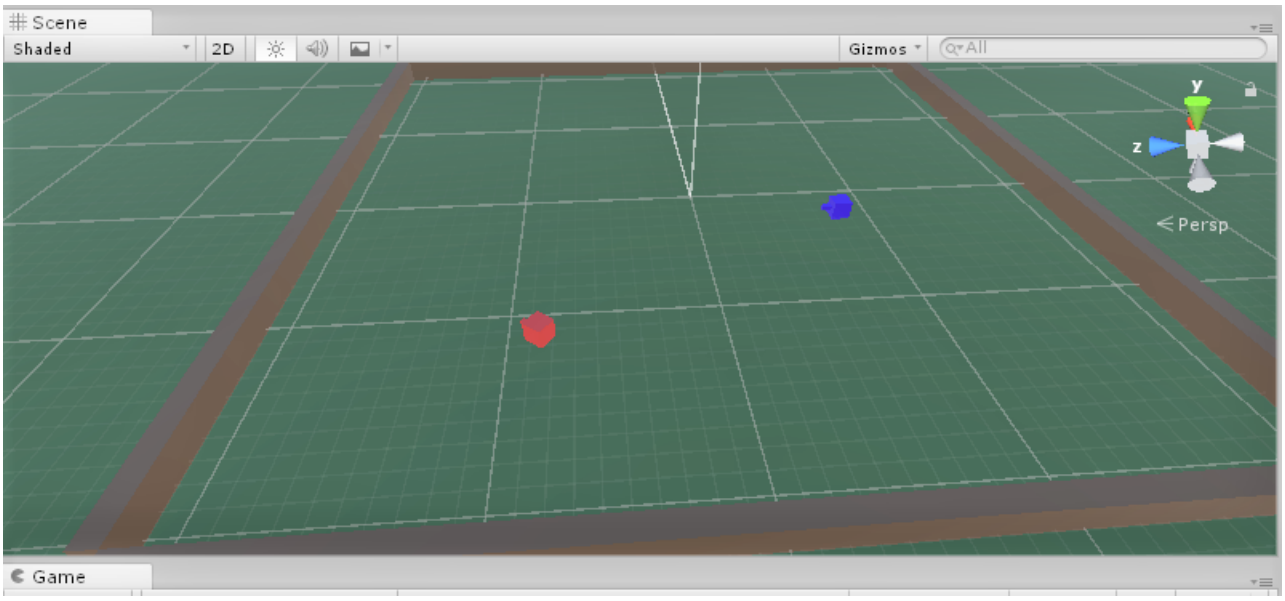

Figure 3.8: Enclosed environment with a player and one enemy inside of it.

A complete environment with player and enemies is not enough to have a functional game. Game managing methods are required to process common dynamics in a TPS game. One of the goals is to have a scoring function in relation to the enemies the player has eliminated. Three scripts are added to an empty object: (1) **Score** is used to calculate the score the player has achieved based on enemy eliminations. (2) **GameOverManager** contains methods to restart the game once the player is eliminated. (3) **SpawnManager** contains methods that spawn enemies in different spawn points using a given interval. The last script allows the player to increase game difficulty by increasing the spawn rate of enemies.

There is a complete TPS game that satisfied the goals established in the section. The game has a main character that can perform basic TPS actions based on the player's input. There are enemies that have similar capabilities as the player and attempt to eliminate the player. There is a closed environment in which the interaction between the player and enemies occur. There are methods that encourage

the goal of maximizing the score a player can obtain. The next section alters the game to fit the reinforcement learning problem and applies Q-learning to find an optimal policy.

3.3 Q-learning on Third Person Shooter Game

The TPS game present in the last section can be represented as a MDP. However, this process requires translating many aspects of the game. Instead, we can consider a set of goals that we further expand to obtain a fully implemented Q-learning approach to the TPS game. Before stating the goals, the agent we intend to train has to be chosen. This research aims at using Q-learning to train the enemies to play against the player. The expectation is having the enemies train to decrease the time the player stays alive in each round. To implement Reinforcement Learning on the TPS game we need:

1. The enemy as an agent.
2. An environment that interacts with the agent.
3. Q-Learning implementation.

3.3.1 The enemy as an agent

Since the enemy is chosen as the agent of this reinforcement learning problem, goals that we expect the agent to achieve must be determined. We expect that once the agent learns an approximate optimal policy, the agent will behave similar to the implemented AI version of the enemy. Goals for the agent have to be chosen carefully, as this has an impact on the state, action, and reward representations. In this case, the goal of the agent is to learn to aim and hit shots at the player. Notice that the currently implemented enemy AI does not ensure that all shots hit since

the enemy aims and shoots at the player. The distance is a factor that affects the accuracy of those shots. With the new approach, we expect that the enemy will be able to improve the accuracy as the goal is to hit the shots as opposed to aiming and shooting at the player. Then, the agent training goal can be subdivided in the following tasks:

1. The agent needs actions.

2. The agent needs states to choose an action.

Consider a stationary player and agent. To create a set of actions such that the agent hits at the player under the rules of TPS game, we need to consider two things: (1) The player position, and (2) the aiming direction. Ideally, both the player position and aiming direction can be represented with continuous values. For practical purposes, the player position and aiming direction are chosen as discrete and the positions can be represented using a grid. Suppose the player can only be in one of the 4 cardinal directions in relation to the agent. To be able to cover all player positions the agent needs 4 different aiming directions. If the player can be in any of the adjacent grids around the agent, the agent needs 8 different aiming directions. If the player can be in any of the grids 2 steps away from the agent the aiming directions required increase at a very high rate.

Recall that the Q-table the agent uses is a two-dimensional table of size $s \times a$. It is better to sacrifice aiming directions for better approximations to optimal policies. In other words, an agent that can hit most shots in a few directions but is not able to shoot at most directions as opposed to being able to shoot in many directions but rarely hitting the player. Considering this we decided to choose 8 different aiming directions. However, the agent and player can move around the terrain. We have to consider the moving directions the agent can have. Following the same aiming direction factor we decided to have 4 moving directions. This results in $8 \times 4 = 32$

possible actions that the agent can choose to decide in which direction to aim and move. With these actions we can achieve an approximation to the optimal policy such that the agent can aim and hit the player in 8 different directions.

Once actions are decided, we need a state representation for the agent. Assume the player and agent positions are discrete and can be represented with a grid. One approach would be to define the states as all possible combinations of the player and agent positions. With this approach, consider a $n \times n$ grid. The combination of possible player and agent positions are $n \times n \times (n-1)$ different arrangements of the player and agent. The corresponding Q-table will have $n \times n \times (n-1) \times a$ different entries. This state size is on the order of n^3. With the algorithm only one of these entries is updated at every time step. A better approach to the state representation is considering the player position relative to the agent position. In other words, the agent can be seen as stationary in the center of the grid and the states are all possible positions in which the player can be. This approach yields on a $n \times n$ grid $((n \times 2) - 1)^2$ different states. This state size is on the order of n^2. This significantly reduces the Q-table size and allows faster training.

3.3.2 The Environment Representation

Once there is an agent and a representation of the states and actions, the environment is partially completed. There is still the need for a system of rewards to update the Q-values. Recall that the environment is any factor that is beyond the agents control. In other words, the player and its behavior. One of the drawbacks of training is that they could last days or even months to approximate a good optimal policy. It is unfeasible for a human player to spend that time helping the agent train. Then, to have a complete environment in which the agent can train we need an AI implementation of the player such that the AI player behaves as a human player. From this we can have a set of properties the AI player needs:

1. The AI player has to move the same way human player would.

2. The AI player has to choose targets the same way a human player would.

Choosing how the AI player will move is based on assumptions. It is unclear which strategy the average player would choose when deciding where to move. There is a spectrum on the different player behaviors, the most notable being aggressive and passive players. Aggressive players are risk takes and would charge at the enemies as opposed to passive players who stay away from danger. With this factor we decided that a good approximation to how a player would move is based on stochastic movement. The AI player would choose in which direction to move randomly at all times. Choosing how the AI player decides on targets is more straightforward. The target is chosen under the assumption that human players will most of the times eliminate the most important threat. The highest threat is the closest enemy, the closest the enemy is to the player the less reaction time the player has to dodge a projectile. Additionally, it also follows that the next possible player positions are most likely to be one of the directions the enemy is capable of shooting at. Considering these factors the AI player always aims and attempts to eliminate the closest enemy. The agent and environment are set up. The next step is to implement in scripts the new changes to the agent and the environment.

The system of rewards is divided into three possible rewards. Every time step, if the agent does not hit the player, the agent receives a small negative reward. This causes the agent to avoid choosing the same action that has failed in the past when choosing greedily. The agent receives a high positive reward if the projectile hit the player. This results in the agent being able to determine the aiming direction in the current state with one hit. The reason this approach was chosen is to increase training speed. If the agent has 32 possible actions for each state, where actions are selected randomly most of the time, then visiting all actions on a state s is most likely to take many visits to s. If we give a small positive reward when the player is

hit, there is a probability that aiming in the same direction the agent hit before does not hit on all cases due to the player movement and distance. If this is the case, there would be a sequence of decreases to an action that has been determined to hit before, possibly decreasing the action enough to not consider it the maximized action. Using a high reward would suppress this effect for longer visits to the state-action pair. If the agent is hit by the player, the agent receives a negative reward larger than the one received for not hitting the player. This is to create a behavior in which the agent learns to move in relation to the player. For instance, if the player is close, the agent could learn that moving perpendicular to the player direction in relation to the agent will increase surviving chances as opposed to moving in the line of fire.

3.3.3 Implementing Q-learning

Before implementing the agent and environment changes, a Q-table and Q-learning methods are required. A new script **Trainer** is attached to a Trainer object in the scene. The script contains a class that implements Q-learning. The class **Trainer** contains a two dimensional array of size $s \times a$ where s is the state size and a is the action size. The most important methods present in the class are: (1) A method that performs the 2.22 bellman updates that take as arguments a quadruple $(S_t, S_{t+1}, a, R_{t+1})$. (2) A method that returns the action to be performed in the current state based on the probability that the action is chosen greedily or randomized using exploration and exploitation. (3) A method that uses another class to save and load Q-tables. This method is implemented since the game restarts when the player dies. This method is called at the beginning and end of every episode.

Once the Q-learning class has been implemented, we need to make changes to the enemy and player. The AI player changes are straightforward, new scripts **AIPlayerAttack**, **AIPlayerAim**, and **AIPlayerController** replace their corresponding implementations for human control. The major changes are randomizing moving

to 4 different possible directions and aiming and shooting at the closest enemy. The enemy changes are more complicated, there needs to be a complete implementation of all possible actions, determining states using the player position, and evaluating rewards.

To implement the actions and states on the agent, the script **MovementTrainer** replaces all other scripts present in enemy. The reason being Unity not allowing passing values between scripts. The most important methods in the script are the following: (1) A method that performs one of the 32 actions based the returned value from the class **Trainer**. Once the action is performed a reward is received. (2) A method that determines the state in which the agent is in based on the player position. This is used when the agent is sending the arguments to the **Trainer** class to update the Q-table and obtain a new action based on the agent's state. Figure 3.9 shows what the agent performs in each time step.

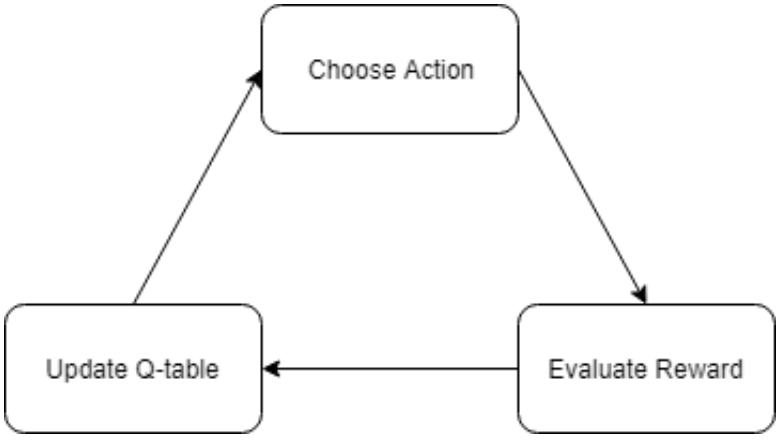

Figure 3.9: Flowchart for the agent.

There are some factors that can affect the results from this implementation. The factors have been addressed to improve the Q-learning implementation. They are:

1. If an agent shoots in a direction that will hit the player but is obstructed by another agent, the agent will receive a negative reward even though the state representation does not consider other agents' position.

2. Q-value updates are done every time step. However, some projectiles could hit the player after the agent already obtained a negative reward since the projectile did not hit the player before the time step was over.

3. The agents could potentially not visit many state-action pairs. This could cause to have a sub optimal policy.

4. The discrete representation of the TPS game.

The first item is addressed by allowing projectiles belonging to enemies to pass through each other. With this change, the state representation does not consider other enemy positions as they do not interfere with rewards. The second item is addressed by creating a new script **Projectile** attached to the projectile object. The script contains a class **Projectile** that stores the state in which the agent is and the future state when the projectile was created. The class has a method that performs a Q-value update if the projectile hits the player using the stored states. The third item is addressed by adding a method to the **Trainer** class. The method sweeps the Q-table and finds the state with the least visited state-action values. This is possible since the Q-table was initialized to all values being 0. Once the state is found, the player and one agent are placed such that the state in which they are is the least visited state. This does not ensure that a zero valued state-action pair is updated since the actions are always chosen either randomly or greedily.

The discrete representation of the TPS game was chosen since it allows faster training. It was possible by rounding the positions of the agent and players to the closest integer. This resulted in a simpler state representation and is aided by having the time steps be small enough that the interval between time steps roughly results in the enemies and player moving almost a unit in the grid. From these small time steps we can also perceive the game as continuous as the time steps are fast enough to allow almost real time reaction from agents and the player AI.

We have successfully adapted the TPS game to fit a Reinforcement Learning problem. The enemies have been implemented as agents to train how to aim and hit the player. The player has been altered to have an AI and a system of reward was chosen to aid in the structuring the environment. Q-learning was implemented as the algorithm to estimate an approximation of the optimal policy the agent should follow. To obtain the an approximation of the optimal policy, the agent has been training for 6 weeks. Figure 3.10 shows an in game figure of many instances of the agent learning against the player AI. The next section contains the results obtained from different weeks and discusses future work.

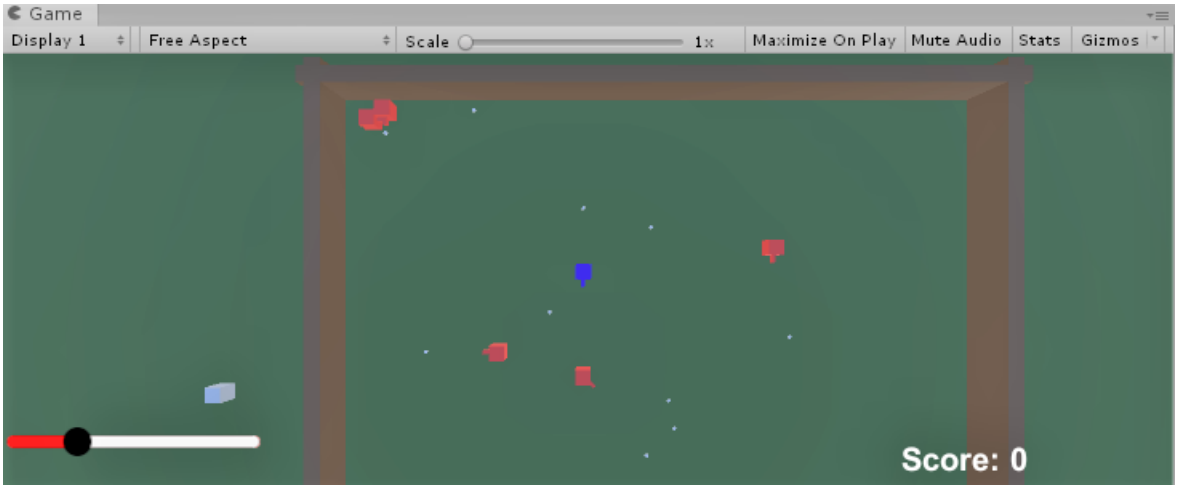

Figure 3.10: In game display.

CHAPTER 4

Results and Future Work

This chapter covers the results obtained from the training performed using Q-learning. The results are followed by the conclusion of the research. Lastly, there is a discussion on future work.

4.1 Results

The agent training lasted 6 weeks. Each week the Q-table would be saved separately to observe progress as the result. The goal of the agent is to hit shots at the player as often as possible. As the agent becomes better at hitting the player, the player lives less on each round. Following this idea, we can record the average time the player spent alive over a time range for each week of training. Figure 4.1 shows the result of calculating for each week the average time the player spends alive over 2 hours of rounds.

Figure 4.1 shows that the training improved over the weeks. Notice how the training improves at a faster rate over the first weeks. From week 1 to 3 there is over 2 seconds of difference in the average time spent alive while from week 4 to 6 there is less than a second of difference in the average time spent alive. Moreover, the change from week 5 to 6 is almost unnoticeable. This approach still uses exploration, many actions chosen by the agents are not optimal. This can affect the performance

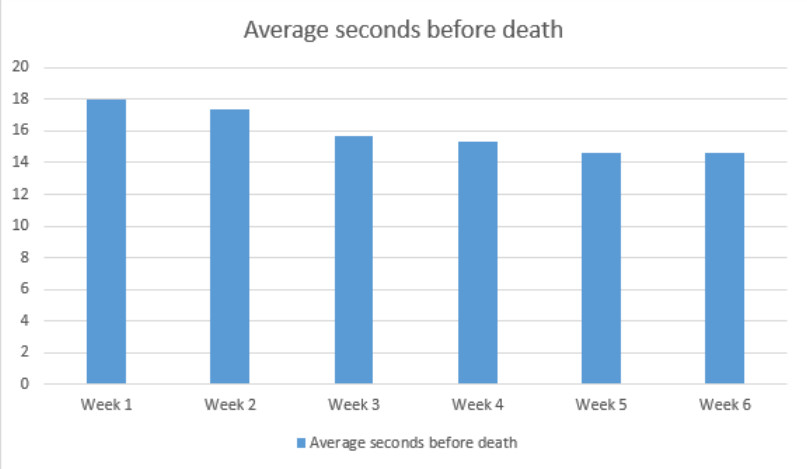

Figure 4.1: The average time the player spent alive.

of the agent. A greedy approach can be used to compare results from different weeks.

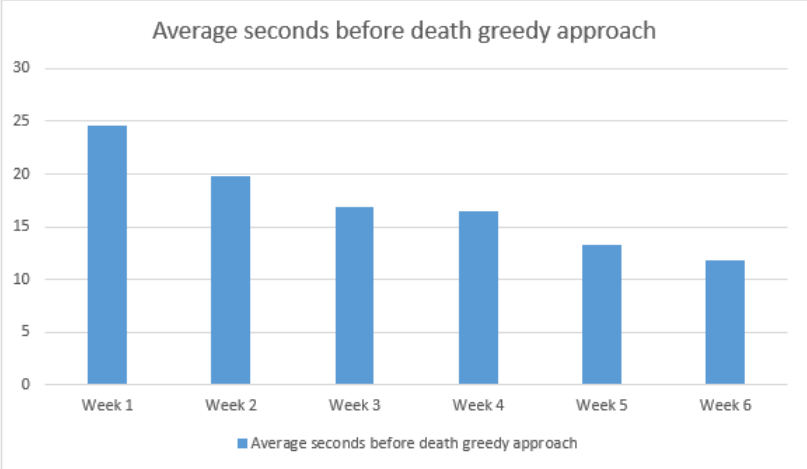

Figure 4.2: The average time the player spent alive when agents are greedy.

Figure 4.2 shows the average time the player spent alive if the agents chose optimal actions in their current Q-tables. The results reflect better the improvement of the agent over the weeks. Notice how the player spends more time alive in the first week on the greedy approach in comparison to the first week of the non greedy approach. This is due to having non optimal actions that are considered optimal due to lack of training duration. The non greedy approach has better

performance since exploration could attain optimal actions. Similarly, the last week has better performance in the greedy approach. The training lasted long enough to have good approximations of the optimal policies. Lastly, there is not a significant difference between the last week for either approaches. This shows that even under randomization using exploration, the policy achieves almost the same performance as an approach of optimal action choices. This shows that under Q-learning the approximated policy is converging to an optimal policy.

4.2 Conclusion

In this research a Third Person Shooting game was created to implement Q-learning using Unity 3D. We learned to use some features from Unity while creating the TPS game, the scripts written in C# being the most notable. To implement Q-learning we had to learn the theory behind Reinforcement Learning. In the process we learned about three different methods to approximate optimal policies: Dynamic Programming, Monte Carlo, and Temporal Difference. To study these methods we learned about Markov Decision Problems and how these prove to be useful in solving Reinforcement Learning problems. Through results of the training we were able to closely observe the Q-learning algorithm approximating our training to optimal policies.

4.3 Future Work

The agent aim representation is not very similar to the AI aim implementation. This was due to the Q-table size exploding and not having significant results. An approach to this is using approximations to the action representation [4] with neural networks to fit values. This is a combination of supervised learning and

reinforcement learning. One example of this can be seen in [10] where Q-learning is combined with neural networks to improve zombies in minecraft.

Q-learning is known to overestimate results [9]. Double Q-learning is a variation of Q-learning where these overestimates are decreased by having two different Q-tables [5]. A different variation to this is the Deep Double Q-learning [11], similar to the last example, a combination of supervised learning and reinforcement learning. This is done by generalizing the discrete representation of Double Q-learning to work with approximations.

We can improve the game by allowing agents to cooperate. This avoids having to make the projectiles go through the enemies. An approach to having agents consider each other in their learning is using Friend-or-Foe Q-learning [1]. With this approach we would have to change the dynamics of the game to fit one of a general-sum game. The algorithm attempts to maximize the expected payoff, this is influences by the trade-off that could happen between different agents' actions.

Future work in relation to the game set up is expanding the game environment. Our current environment is enclosed to create a grid with a finite state representation. Currently, the agent learns how to hit the player up to a fixed range. We could include a new state variable that determines if the player is in that range or not. With this the environment can be expanded so that the agent would attempt to find the player if the player is not in shooting range. A further improvement to the state is adding walls that could block potential hits if the wall was not present. With this state representation and the appropriate rewards the agent can learn to play in a more complicated environment. One last addition can be considering pathing as part of the state. The agent can learn to use walls as cover or find strategic approaches to the player. This would require more complex state representations and this is with the assumption that the map is not randomly generated.

References

1. Friend-or-foe q-learning in general-sum games, 2011. 52

2. U. Ravi Babu, Y. Venkateswarlu, and Aneel Kumar Chintha. Handwritten digit recognition using k-nearest neighbour classifier. *IEEE*, (14220407), 2014. URL https://ieeexplore.ieee.org/document/6755106. 6

3. RICHARD BELLMAN. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957. ISSN 00959057, 19435274. URL http://www.jstor.org/stable/24900506. 17

4. Chris Gaskett, David Wettergreen, and Alexander Zelinsky. Q-learning in continuous state and action spaces. In Norman Foo, editor, *Advanced Topics in Artificial Intelligence*, pages 417–428, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. 51

5. Hado V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010. URL http://papers.nips.cc/paper/3964-double-q-learning.pdf. 52

6. Vishal Maine and Samer Sabri. *Machine Learning for Humans*. ml4humans, 2017. 6

7. Sigaud Olivier and Buffet Olivier. *Markov Decision Processes in Artificial Intelligence*. ISTE, 2010. ISBN 978-1-84821-167-4. 13

8. Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning*. Cambridge University Press, 2017. 4, 5

9. Richard Stutton and Andrew Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2017. 8, 9, 10, 11, 12, 14, 16, 18, 19, 20, 21, 22, 23, 24, 25, 52

10. Hiroto Udagawa, Tarun Narasimhan, and Shim-Young Lee. Fighting zombies in minecraft with deep reinforcement learning, 2016. 52

11. Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015. 52

12. Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8 (3-4):279–292, 1992. doi: 10.1007/bf00992698. 25, 26