**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

BACHELOR THESIS

Una Adilović

Playing a 3D Tunnel Game Using Reinforcement Learning

Department of Software and Computer Science Education (KSVI)

Supervisor of the bachelor thesis: Adam Dingle, M.Sc.

Study programme: Computer Science

Study branch: Artificial Intelligence

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In . . . . date . . . .
                                                    Author's signature

i

Title: Playing a 3D Tunnel Game Using Reinforcement Learning

Author: Una Adilović

Department: Department of Software and Computer Science Education (KSVI)

Supervisor: Adam Dingle, M.Sc., Department of Software and Computer Science Education (KSVI)

Abstract: Tunnel games are a type of 3D video game in which the player moves through a tunnel and tries to avoid obstacles by rotating around the axis of the tunnel. These games often involve fast-paced gameplay and require quick reflexes and spatial awareness to navigate through the tunnel successfully. The aim of this thesis is to explore the representation of a tunnel game in a discrete manner and to compare various reinforcement learning algorithms in this context. The objective is to evaluate the performance of these algorithms in a game setting and identify potential strengths and limitations. The results of this study may offer insights on the application of discrete tabular methods in the development of AI agents for other continuous games.

Contents

Introduction

Reinforcement learning is a subfield of machine learning that aims to train agents to make decisions that will maximize a reward signal [Sutton and Barto, 2018]. This approach has been widely applied in the field of artificial intelligence, particularly in the context of training agents to play games. In a game setting, an agent's actions can be evaluated based on their impact on the agent's score or likelihood of winning. Through the process of reinforcement learning, the agent learns to make strategic decisions that maximize its reward by receiving positive reinforcement for good moves and negative reinforcement for suboptimal moves. This allows the agent to adapt and improve its performance over time as it plays the game. Research on reinforcement learning in games has demonstrated its effectiveness in a variety of contexts, including board games, video games, and real-time strategy games.

In the field of artificial intelligence, games can be classified as either continuous or discrete based on the nature of the action space and state space. Continuous games have a continuous action space, meaning that the possible actions an agent can take are not limited to a fixed set of options, but can vary continuously within a certain range. In contrast, discrete games have a discrete action space, meaning that the possible actions are limited to a fixed set of options. Continuous games are often characterized by a high-dimensional state space, as they may involve a large number of variables that describe the game state. Discrete games, on the other hand, typically have a lower-dimensional state space, as the number of possible states is limited by the discrete action space. In general, continuous games are more challenging to model and solve than discrete games, as they require more complex decision-making algorithms and may require more computational resources.

In this thesis, we will investigate the application of reinforcement learning to train agents to play a continuous 3D tunnel game, which I designed and implemented myself for this thesis work. The continuous game environment will be discretized into a set of states, and different reinforcement learning algorithms will be applied to train agents to play the game. The goal of this study is to determine whether it is possible for any of the agents to win the whole game, and to compare the performance of different agents that use different reinforcement learning algorithms.

The results of this study will contribute to the understanding of the potential of reinforcement learning for training agents to use discrete algorithms in a naturally continuous environment, and to provide insight into the strengths and weaknesses of different reinforcement learning algorithms in this context.

1. Game Design

For this thesis work I designed and implemented a game called "Space-Run", which involves attempting to accumulate the highest score possible by navigating through three distinct tunnels while avoiding various obstacles. The game is endless in nature, as the speed increases each time the player successfully completes all three tunnels.

It is worth noting that, in designing this game, I was inspired by the pre-existing "Tunnel Rush" (tun [2022]). "Tunnel Rush" and "Space Run" are both 3D tunnel games that involve advancing through a tunnel to avoid traps. However, there are several key differences between the two games. "Tunnel Rush" is a web-based game played in first-person perspective, while "Space Run" is a desktop-based game played in third-person perspective. "Tunnel Rush" has levels, some of which are inverted with the traps on top of the tunnel and the player outside of it. "Space Run", since inspired by "Tunnel Rush", also has levels, but they are all inside the tunnel and features not only traps but also creatures that the player must avoid or shoot. Additionally, "Space Run" has a computer-themed setting, with elements such as battery, bugs, and viruses.
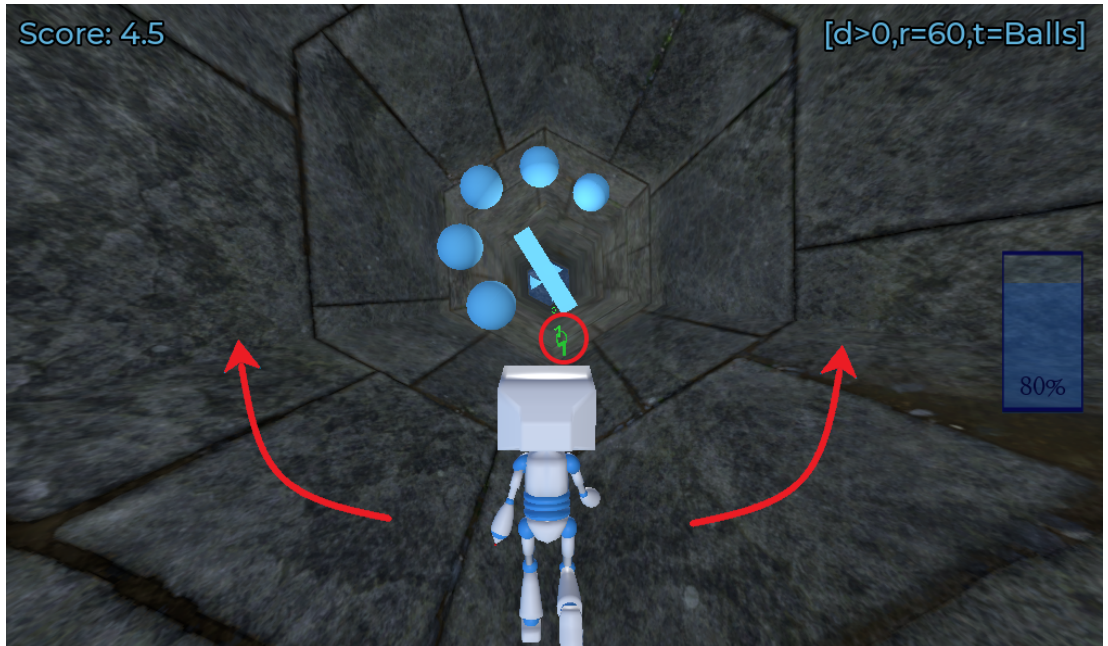
1.1   Player and Movement

Figure 1.1: Movement

In "Space-run" the player assumes control of a character named Hans (see Figure 1.2) who continually advances at a constant speed through the tunnels. To navigate through the game, the player must use the left and right arrow keys to rotate the current tunnel and avoid obstacles (as shown in Figure 1.1).

In addition to these lateral movements, Hans also has the ability to shoot bullets (also shown in Figure 1.1) by pressing the space key, which can be used to defeat certain in-game creatures and earn a higher score. The player must utilize these abilities in order to progress through the game and achieve a high score.

1.2 Obstacles

Figure 1.2: Hans

As previously stated, the player must navigate through various obstacles in the game. These obstacles can be divided into three distinct categories, and each tunnel contains a unique subset of them. In the subsequent sections, we will delve deeper into these categories in order to better understand the challenges faced by the player.

1.2.1 Traps
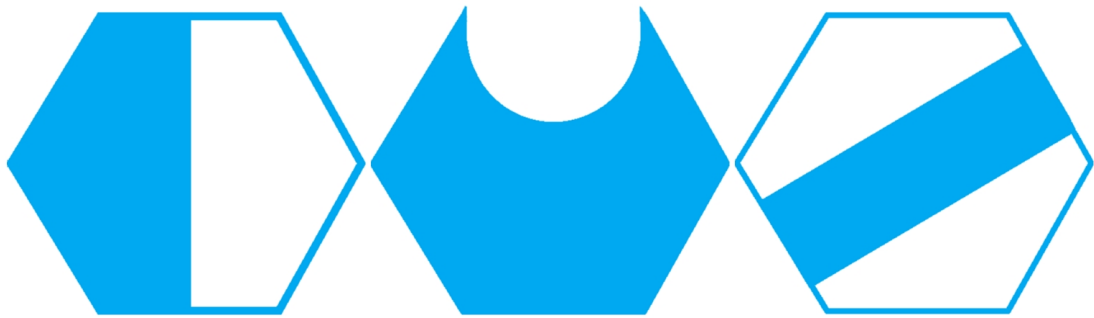
Figure 1.3: Trap examples

As depicted in Figure 1.3, a selection of the various trap types that the character Hans must avoid is presented. These traps, of which there are a total of 10, vary in their level of difficulty and can be either static or animated. These traps can be encountered in any of the three tunnels, and if the player fails to successfully evade them, they result in an instant death.

1.2.2 Bugs

Figure 1.4: Bugs

In addition to traps, the game also features bugs as an obstacle (as shown in Figure 1.4). These bugs typically appear in the second tunnel and are designed to rotate around the tunnel toward the player's position, making them more challenging to evade. However, they can still be avoided by the player. If the player chooses to engage with the bugs, they can be defeated by shooting three bullets at them. If the player collides with a bug, Hans will lose 25% of his battery life (for more information on battery life, see Section 1.3).
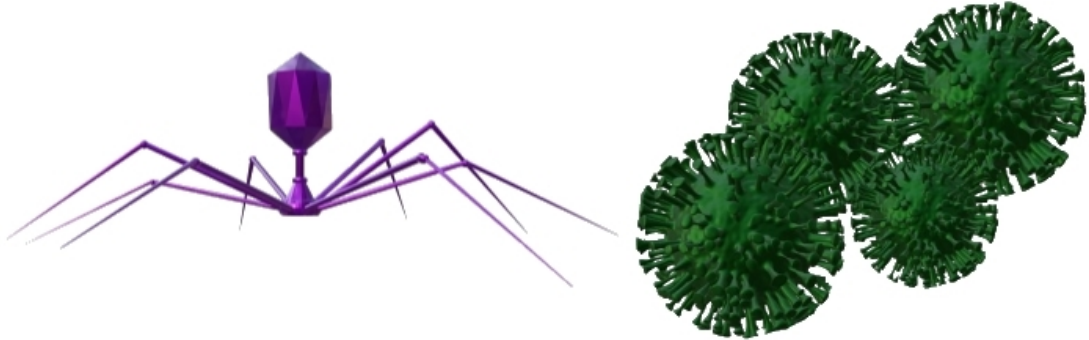
1.2.3 Viruses

Figure 1.5: Bacteriophage and Rotavirus

The third and final type of obstacle in the game are viruses (illustrated in Figure 1.5). These viruses are typically found in the third tunnel and, similar to bugs, can be eliminated through the use of three bullets. They also, just like bugs, rotate around the tunnel toward the player's position. Bacteriophage, a subtype of virus, will result in an instant death if the player comes into contact with them. Rotaviruses, on the other hand, will cause the player's character to become sick for a brief period of time. During this illness, it is crucial for the player to avoid coming into contact with another Rotavirus, as this will result in the end of the game.

1.3   Additional Features

There are several other features of the game that are worth mentioning. One of the most significant of these is the battery life of the player's character, Hans, which is displayed on the right side of the screen (as shown in Figure 1.6). As Hans is designed to resemble a computer, it is necessary for him to recharge his battery throughout the game by collecting energy tokens (Figure 1.6). This will fully restore his battery capacity. There are three main ways in which Hans can lose battery life: running causes a constant reduction of 1% every 0.5 seconds, each bullet shot costs 1% of the battery life, and coming into contact with a bug results in a reduction of 25% (as described in Section 1.2.2). If the battery reaches 0%, Hans will die and the game will end.

Finally, it should be noted that upon successfully navigating through all three types of tunnels, the game will increase in speed and the player will once again encounter the same tunnels, looping through them indefinitely until the player loses.

Figure 1.6: Battery and Energy Token

1.4 Score Count and Winning

The score of the game is based on the length of time that the player is able to survive. Additionally, each time a player successfully shoots down a bug or virus, their score increases by 10 points. As previously mentioned, the game is designed to be played indefinitely, but for the purpose of this study, we have set the game to be considered won after an agent successfully completes nine tunnels, reaching level 10.

2. Implementation of the Game

"Space-run" was developed using the Godot Engine (version v3.2.3.stable), an open-source game engine licensed under the MIT License. It is a cross-platform tool that offers a range of features for game development, including a visual scripting language, 2D and 3D graphics support, and a powerful physics engine. The Godot Engine utilizes a node-based architecture, where nodes are organized within scenes that can be reused, instanced, inherited, and nested. This structure allows for efficient project management and development within the engine. The game was written entirely in GDScript, the primary scripting language of the Godot Engine (et al [2021]).

In addition to using the Godot Engine, I also utilized Blender (version 6.2.0) (et al [2022]) for creating and animating the characters in the game. Blender is popular open-source 3D modeling and animation software that offers a range of features for creating detailed and realistic characters. The characters were then imported into the Godot Engine using the .glTF 2.0 Group [2022] file format, which is a widely supported file format for exchanging 3D graphics data.

The game can be run on any platform provided within the Godot engine, and its source code, and the source code for the whole thesis can be found online (Adilovic [2022]).

2.1 The top-level organization
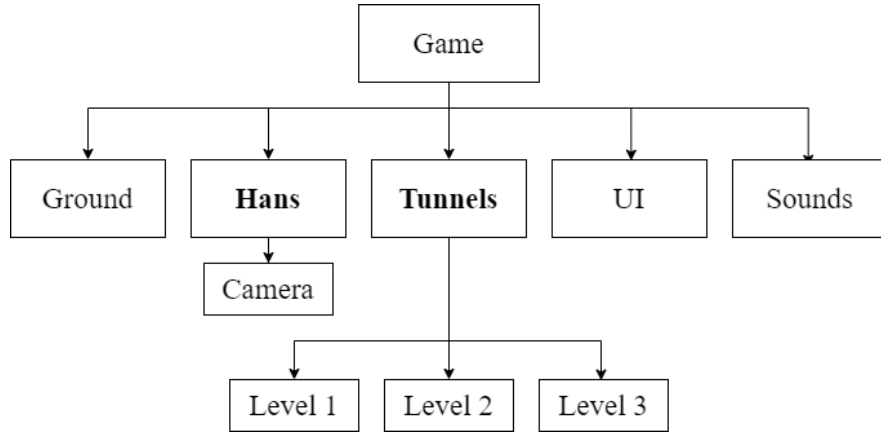
Figure 2.1: Structure of Game.tscn

The main scene for the game, referred to as `Game.tscn`, is depicted in Figure 2.1. It includes several nodes, including Ground, UI, Sounds, Game, Hans, and Tunnels. The Ground node is a CSGBox[1] that serves as the ground in the game, while the UI and Sounds nodes handle the user interface and audio aspects, respectively. The Game, Hans, and Tunnels nodes contain the majority of the game's functionality. Specifically, the Game node manages the overall gameplay,

---

[1]A CSGBox is a 3D object that represents a box with a Constructive Solid Geometry (CSG) shape.

the Hans node controls the player character, and the Tunnels node manages the movement and appearance of the tunnels.

For a more in-depth understanding, let us examine some of the core aspects of the game in the following sections.

2.2   Game

The script for the Game node is the initial point of the game session and includes both the `_start()` and `_game_over()` methods. It also serves as a link between the game and the agent environment described in Chapter 3, and as such includes all of the necessary set methods for the agent environment. These methods allow for communication between the game and the agent environment, enabling the agent to interact with and influence the game.

The following text describes the core functionalities of the main methods within `Game.gd`:

- The `_ready()` function is called at the start of the game's execution and, after setting up the environment, it triggers the `_start()` function. This function initiates the gameplay and sets the necessary conditions for the game to proceed.

- As described in more detail in Chapter 3, the user can specify environment parameters and a starting level for the agent through the command line. These parameters determine the obstacles that the player will face and the starting position of the player character, Hans. The `_start()` function incorporates these parameters into the obstacle arrays and positions Hans accordingly. The function also generates the obstacles for the designated starting level. The creation and deletion of obstacles during gameplay is discussed in Section 2.4 of this chapter.

- The `_game_over` function manages the end of the game and sends a signal to the top-level script, `Main.gd` (described in Chapter 3), indicating that the game has ended. It also provides `Main.gd` with the necessary information about the game's status and outcome.

2.3   Hans

The next node we want to examine is Hans. While `Hans.tscn` is a scene with the main character and its necessary animations, what interests us more is the `Hans.gd` and its key components.

The primary function within `Hans.gd` is the `_physics_process()`, which is called on every tick of the game. It handles the main aspects of the player character through the use of various methods and functions. These include deleting passed obstacles, creating new obstacles every 50 meters, updating the score, handling the movement of the player character, bugs and viruses, and determining the current state of the player. The state label, which is displayed on the upper right corner of the screen (as shown in Figure 2.2), is the primary information that agents receive when making decisions about their next move, as described in

```
 1: function PHYSICS_PROCESS(delta)
 2:     tunnels.deleteObstacleUntilX(. . . )
 3:     if translation.x < new_trap then          ▷ create a trap in the next tunnel
    every 50 meters
 4:         CREATENEWTRAP
 5:     end if
 6:     score._on_Meter_Passed()                            ▷ update score
 7:     velocity ← Vector3.LEFT ∗ speed
 8:     velocity ← MOVEANDSLIDE(velocity)
 9:     CHECKCOLLISIONS
10:     tunnels.bugVirusMovement(delta, curr_tunnel)
11:     if isShootingButtonPressed then
12:         SHOOT
13:     end if
14:     type ← CALCTYPE
15:     state.updateState(calcDist(), calcRot(), type)
16: end function
```

Chapter 3. The `_physics_process()` function also handles collisions and shooting if the player chooses to do so. Overall, this function plays a crucial role in the gameplay and management of the player character.

It is also worth noting that this script handles the movement of the tunnels to the back as Hans passes them, with the first tunnel being moved to be after the third one. This feature allows for the game to be infinite, as the tunnels are constantly cycled and reused.

Figure 2.2: State

2.4 Tunnels

The Tunnels node, which is a child of the main scene in the game tree, contains three child nodes of the Spatial type[2] (level1, level2, and level3) and each of these nodes includes a CSGTorus[3] node, which represents the physical appearance of the tunnels. Obstacles are added to the appropriate level node as instances. The `Tunnels.gd` script, which is attached to the Tunnels node, handles many of the previously mentioned functions such as obstacle creation and deletion and tunnel rotation. In the following code snippets, we will examine the `Tunnels.gd` script in greater detail.

1: **function** PHYSICS_PROCESS($delta$)
2:     $move \leftarrow game.agent.\text{move}(\dots)$
3:     **if** $move[0] = 1$ **then**
4:         $tunnel \leftarrow \text{get\_child}(hans.\text{get\_current\_tunnel}())$
5:         $tunnel.\text{rotate\_object\_local}(Vector3.\text{RIGHT}, -ROTATE\_SPEED * delta)$
6:     **else if** $move[0] = -1$ **then**
7:         $tunnel \leftarrow \text{get\_child}(hans.\text{get\_current\_tunnel}())$
8:         $tunnel.\text{rotate\_object\_local}(Vector3.\text{LEFT}, -ROTATE\_SPEED * delta)$
9:     **end if**
10:     **if** $\neg hans = \text{null}$ **then**   ▷ If it is not instanced we can't call the function
11:         $hans.\text{switch\_animation}(move[1] = 1)$ ▷ Shoot if necessary
12:     **end if**
13: **end function**

The `_physics_process()` function within the `Tunnels.gd` script serves as the primary connection between the agent and the game. As shown in the provided code, the function retrieves the next move from the agent and rotates the tunnel accordingly, potentially including shooting as well. It should be noted that, in the case of the `Keyboard` agent, function move returns users input from the keyboard.

The function depicted in the code above serves to generate obstacles in the starting tunnel. By periodically creating traps in the tunnel ahead, the game is able to prevent lag caused by an excessive number of objects existing simultaneously. For that reason, this function is used only once, at the beginning of the game.

The tunnels are positioned along the x axis, and this function allows for the creation of obstacles within them at specific x positions and a random rotation.

As previously mentioned, by dynamically deleting passed obstacles, the game is able to maintain a stable performance and avoid overloading the system.

---

[2]A Spatial node is a type of node that represents a 3D object or transformation in the game world. It is a versatile node that can be used to create and manipulate 3D objects, including meshes, materials, and lighting. Spatial nodes are often used as the root node for 3D objects in a scene, and they can be nested inside other Spatial nodes to create hierarchical transformations.

[3]A CSGTorus node is a type of 3D object that represents a torus shape in the game world.

```
 1: function CREATE_FIRST_LEVEL_TRAPS(tunnel)
 2:     level ← level we are making traps for
 3:     num_of_traps ← randomly pick number of traps to be added
 4:     x ← x position of the first trap
 5:     for n in num_of_traps do
 6:         x ← update x to next position
 7:         if x is outside the tunnel then
 8:             break
 9:         end if
10:         create_one_obstacle(level, x)
11:     end for
12: end function
```

```
 1: function CREATEONEOBSTACLE(level, x)
 2:     scene ← pick which kind of obstacle will be added
 3:     tunnel ← get the level we are making traps for
 4:     i ← randomly pick an obstacle
 5:     make an instance of the obstacle using previous information
 6:     set its position at x and rotate it randomly by n · 60 degrees
 7: end function
```

```
 1: function DELETE_OBSTACLE_UNTIL_X(level, x)
 2:     tunnel ← get current tunnel
 3:     for child in children of the tunnel do
 4:         if child is an obstacle then
 5:             if x is less than the child's position then
 6:                 remove child
 7:             else
 8:                 return
 9:             end if
10:         end if
11:     end for
12: end function
```

3. Structure of the Experimental Setting

To train an agent on a specific environment, the user must utilize a command-line interface. There are various options available to cater to the user's needs. This chapter will outline all of the provided options and how they are encoded. However, we will first consider how the game represents discrete states.

3.1 State

The majority of the implemented agents in the game utilize the concept of state to facilitate learning. The agent will determine its next action based on the current state in which it finds itself. By dividing the game into discrete states, we are able to utilize tabular method algorithms to train an agent to play this continuous game. Once one obstacle is passed, the value of the state indicated refers to the next one. Each state is represented by a triplet consisting of distance, rotation, and next obstacle type.

Before further explaining this notion, lets look at the standard unit of distance measurement in Godot - a "meter" .It is used to represent the size, position, and movement of objects in the game world. In Godot, one meter is equal to the size of a standard cube. To better understand the notion, it would be useful to note that Hans is approximately 12m tall and his starting speed is 35 meters per second, while each tunnel has width and height of approximately 45m and depth 2800m.

The distance value indicates the distance of the agent from the next obstacle in meters. For example, if we set the `dists` parameter (indicated by the user, see 3.2.1) to 2, there are two possible distance values for the state: >50, >0, indicating that the agent is 0-49 meters away from the obstacle and at least 0 meters away from the obstacle, respectively. If the `dists` parameter is set to 1, the distance value remains constant at >0.

The rotation parameter divides the obstacle circumference into `rots` number of intervals (see 3.2.1). As the tunnel rotates, the state label will indicate the rotation value to which Hans is aligned at the moment (in Figure 3.1 Hans is aligned with rotation value 240). If the `rots` parameter is set to 360, this would correspond to the number of degrees in a circle and result in 360 possible rotations. However, the obstacles in this game do not require such a high number of rotations and agents can be trained to avoid most obstacles using fewer than 10 rotations.

Finally, the type parameter indicates the type of obstacle that Hans must avoid. Each obstacle has its own string representation. With this information, the agent can learn to recognize the safe rotation for different types of obstacles and, along with the distance parameter, decide whether to move left, right, forward, or shoot in combination with any of these actions. To further explain this notion lets look back at the Figure 3.1. Here, seemingly the 240 rotation value should be the way to go and Hans can easily pass through the obstacle. However, with closer observation, it is visible that in fact, not the whole of rotation 240 is safe. The agent does not know the difference between being at any point of the 240
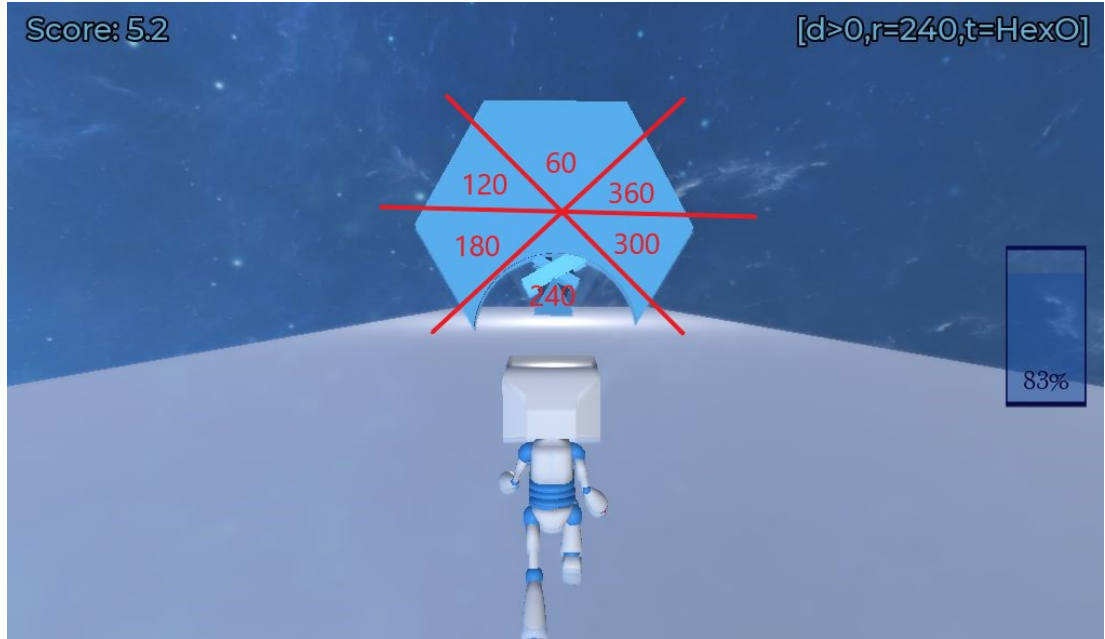
Figure 3.1: Rotations when `rots = 6`

rotation and thus can easily die. For this obstacle, indeed, there would be more `rots` needed in order for Hans to have at least one safe rotation value (meaning that the whole piece of the obstacle that rotation value covers is considered safe).

3.2 Command line options

+ n=int	number of games
agent=string	name of the agent
level=int	number of the level to start from
env=[string]	list of obstacles that will be chosen in the game
shooting=string	enable or disable shooting
dists=int	number of states in a 100-meter interval
rots=int	number of states in 360 degrees rotation
database=string	read the data for this command from an existing file and/or update the data after the command is executed
ceval=bool	performs continuous evaluation
debug=bool	display debug print statements
options	displays options

Note: any of these options can be omitted as they all have default values. If no options are specified, a normal game with the `Keyboard` agent will be run.

3.2.1 Command line option descriptions

n - Number of games the agent will train on in this session. The default is 100.

agent - Name of the desired agent
    Options: [`Keyboard`, `Static`, `Random`, `MonteCarlo`, `SARSA`, `QLearning`,

14

ExpectedSARSA, DoubleQLearning]
Sub-options (only for the listed agents): MonteCarlo, SARSA, QLearning,
ExpectedSARSA, DoubleQLearning =[float, float, float, float] :

[gam (range [0,1]), eps (range [0,1]), epsFinal (range [0,1]), initOptVal
(range [0,∞))]
Example usage: "agent=MonteCarlo:eps=0.1,gam=0.2"
The meaning of the suboptions will be explained in a later section.

level - Number of the level to start from. Default value is 1.
Options: [1, ... , 10]
Note: after the 10th level, the agent is considered to have won the game

env - List of obstacles that will be chosen in the game
Options (any subset of): [Traps, Bugs, Viruses, Tokens, I, O, MovingI,
X, Walls, Hex, HexO, Balls, Triangles, HalfHex, Worm, LadybugFlying,
LadybugWalking, Rotavirus, Bacteriophage]
Note: if this parameter is not included, the environment will contain all
available obstacles (i.e. the full game).
Example usage: "env=HexO,I,Bugs"

shooting - Enable or disable shooting
Options: [enabled, disabled]
Note: this option is disabled by default.

dists - Number of states in a 100-meter interval
This parameter is part of the state label and typical options range from 1
to 3. Default value is 1.

rots - Number of states in 360 degrees rotation
This parameter is part of the state label and the minimum viable option is
6. This is also the default value.

database - Read or write data for this command from/to a file
Options: [read, write, read_write]
Note: these files are typically used to start another session of the agent's
training from the last point of the previous session, to run a game with
visuals and observe the agent's performance, or for plotting the results. This
option does not affect the Keyboard, Static, or Random agents. Default
option for this parameter is to neither read nor write.

ceval - Performs continuous evaluation
This parameter indicates that after each training game, a test game will
be played using only the policy(s) learned thus far. For example, if the
user specifies "n=100", a total of 200 games will be executed, with 100 of
them being training games and the remaining 100 being test games. This
allows for the assessment of the agent's progress and performance during
the training process. Options: [true,false (default)]

debug - Display debug print statements
Options: [true,false (default)]

options - Displays all of the mentioned options

3.2.2 Running the program

There are several possibilities for running the game from the command line. In addition to various combinations of the options listed above, the user has the choice of running an experiment with or without the graphical interface. If they opt for the first possibility, the window will open and the game will be played at its normal speed. On the other hand, if the experiment is run without graphics, it will be over 200 times faster and the output will only be displayed in the terminal. This is achieved by a few things, mainly hiding the CSG geometry in every node. The computational power required to perform union, intersection, etc. on the CSG shapes is quite high and thus by not performing those calculations a game can run much faster. Of course, these shapes are not necessary for the experiments run without graphics, since the collision shapes are the ones that play a role in determining what happened in the game[1]. It should also be noted that the experiments are fully reproducible since the seed values for all random variables are predetermined.

To run the program in the command line, the user may benefit from adding the Godot executable to the PATH environment variable. This will allow them to start the application from the command line simply by entering the command `godot` while inside the same directory as the `project.godot` file.

By default, running the program in this manner will launch a normal game with the graphical interface and the `Keyboard` agent. However, the user can customize their experiment by using a combination of the options listed above. For example:

$$\mathrm{godot\ database{=}write\ agent{=}SARSA{:}initOptVal{=}100.0, eps{=}0.3\ env{=}HexO\ n{=}}$$

Alternatively, the user may choose to train the agent faster by disabling the graphical interface and increasing the speed of the program. This can be achieved by modifying the previous command as follows:

$$\mathrm{godot\ {-\!-}no{-}window\ {-\!-}fixed{-}fps\ 1\ {-\!-}disable{-}render{-}loop\ database{=}write}$$

To view a list of available options, the user can simply enter the command `godot options`.

3.3 Main

The `Main.tscn` scene is the top level scene in the game and consists of a single Node type node. The script attached to this node, `Main.gd`, is responsible for ensuring that all options specified in the command line (as discussed in Section 3.2) are executed correctly. This script is the starting point of the training and handles the initialization and execution of ore or more game sessions. There are several key functions within the `Main.gd` script that are worth discussing in more detail.

---

[1]The collision shapes refer to the shapes that are used to define the physical bounds of an object for the purpose of collision detection.

```
 1: function _READY
 2:     unparsed_args ← OS.get_cmdline_args()
 3:     if unparsed_args.size() = 1 and unparsed_args[0] = "options" then
 4:         DISPLAY_OPTIONS
 5:     end if
 6:     ...                                                          ▷ parse args
 7:     if SET_PARAM(args) = false then
 8:         DISPLAY_OPTIONS
 9:     else
10:         INSTANCE_AGENT
11:         BUILD_FILENAME
12:         if not agent_inst.init(actions, read, write, command, n, debug) then
13:             PRINT("Something went wrong, please try again")
14:             DISPLAY_OPTIONS
15:         end if
16:         PLAY_GAME
17:     end if
18: end function
```

The _ready() function is the starting point of the program when run from the command line. It is responsible for parsing all of the arguments and checking their validity. If any issues are encountered, the program will display options and terminate. If the arguments are valid, an agent will be instantiated and initialized. In cases where everything is in order, first game will be played by calling the play_game() function.

```
 1: function PLAY_GAME
 2:     if agent = "Keyboard" and VisualServer.render_loop_enabled then
 3:         ...                                               ▷ play a regular game
 4:     else if n > 0 then
 5:         n ← n − 1
 6:         game ← game_scene.instance()
 7:         SET_PARAM_IN_GAME()
 8:         agent_inst.start_game(is_eval_game)
 9:     else
10:         agent_inst.save(write)
11:         PRINT_AND_WRITE_ENDING()
12:     end if
13: end function
```

The play_game() function is called each time a game is played.Firstly, it will check if the agent selected was the Keyboard agent, and if so, the program will start one game session where the user has controls of Hans. Otherwise, one of the remaining agents will take over and play specified number of games (defined by the "n" parameter). If n games have already been played, the program will terminate after performing the last few tasks needed to save all of the knowledge gained form this particular session. Otherwise, a single game will be executed and number of games left decreased.

1: **function** ON_GAME_FINISHED($score, ticks, win, time$)
2:     PRINT_AND_WRITE_SCORE(score, win)
3:     $agent\_inst$.end_game($score, time$)
4:     PLAY_GAME()
5: **end function**

The `game_over()` function is called when the game emits a signal indicating that it has finished. Upon execution, this function outputs the necessary information, updates the agent through the `end_game()` function, and then calls the `play_game()` function to continue the game session.

4. Applied Algorithms

In this chapter, we will discuss the algorithms employed to create agents for the game, which broadly fall into the categories of Monte Carlo methods and Temporal Difference (TD) Learning. In essence, the agents aim to maximize their reward by selecting actions that yield the greatest possible benefit. To comprehend the functioning of these algorithms, it is necessary to introduce several fundamental concepts.

A **state** represents the current status of the environment, while an **action** denotes a decision made by the agent in response to the current state. A **reward** is a scalar value that reflects the immediate feedback received by the agent for its action in a given state. The **discount factor**, usually denoted as γ, is a value between 0 and 1 that represents how much the agent values future rewards compared to immediate rewards. An **episode** refers to a sequence of states, actions, and rewards that begins with an initial state and ends when a terminal state is reached. It represents one run or iteration of the agent interacting with the environment. The length of an episode can vary depending on the problem and the algorithm being used (e.g. in a game, an episode may correspond to a single game). A **policy** is a mapping from states to actions that determines the actions an agent takes in each state. An ϵ-**greedy policy** is a policy in which the agent selects the action that maximizes the expected reward with a probability of $(1 - \epsilon)$, while taking a random action with a probability of ϵ. Two types of value functions exist, namely **state-value** functions and **action-value** functions. The former predict the expected long-term reward of being in a particular state, while the latter predict the expected long-term reward of taking a specific action in a particular state and always following the optimal policy thereafter. Action-value and state-value functions evaluate evaluate the relative effectiveness of different actions or states, serving as a measure to determine the optimal action in a particular state. The **value of a state** varies between algorithms, and it is used to select the best action to be taken in that state.

Before looking into individual algorithms, there is one more key concept to introduce: **exploration vs exploitation**. In the field of reinforcement learning, the exploration-exploitation trade-off refers to the balancing act between discovering new information or strategies and utilizing existing knowledge to maximize reward. Exploration involves trying out different actions or strategies in order to gather more information about the environment and its rewards, while exploitation involves utilizing the information gathered to maximize reward. Finding the right balance between exploration and exploitation is crucial in reinforcement learning, as excessive use of either can result in suboptimal results. To balance out these two concepts within these algorithms two methods are used: the formerly mentioned ϵ-**greedy policy** as well as the **optimistic initial values** which is a technique which sets the initial value of all state action pairs to a high number, encouraging the agent to visit as many states as possible in order to learn their true value [Sutton and Barto, 2018].

4.1 Monte Carlo

Monte Carlo (MC) methods are a type of reinforcement learning algorithm that estimate the value of a state or action by averaging the total reward received from sample episodes. Unlike some other methods, such as dynamic programming, MC methods do not require knowledge of the transition probabilities between states or the reward function. Instead, they learn from experience by directly observing the outcomes of sample episodes.

During each episode, the agent follows its policy to select actions, receives rewards from the environment, and transitions to the next state. Once an episode terminates, the total reward received from that episode is recorded. This total reward is used to update the value estimates for each state and action that were encountered during the episode.

There are two types of MC learning: on-policy and off-policy. On-policy learning means that the agent is using the same behavior policy to collect samples as it is using to improve the value function. Off-policy learning, on the other hand, means that the agent is using a different policy to collect samples than the policy it is using to improve the value function.

One variant of on-policy MC learning is first-visit Monte Carlo. This method only considers the first time a state is visited in an episode, as opposed to all visits. The goal of using this method is to reduce variance in the value estimates and improve learning efficiency [Sutton and Barto, 2018].

To ensure exploration during learning, the ϵ-greedy policy is often used in conjunction with MC methods. Additionally, setting an initial optimistic value can encourage the agent to visit more states to learn their true values. While Monte Carlo methods are guaranteed to converge to an optimal policy with an infinite number of samples, convergence can be slow and estimates can be noisy (i.e., have high variance) with a small number of samples.

The pseudocode for the first-visit Monte Carlo can be seen in Algorithm 1 [Sutton and Barto, 2018][1].

To clarify this and future algorithms, here are further explanations to some of the elements that might be encountered:

- $\mathcal{S}$- The set of all possible states.

- $\mathcal{A}(s)$ - The set of actions possible in that state.

- S_t or A_t - Specific state or action taken at time step t.

- R_t - The reward received by the agent at time step t.

- G - The actual total return that the agent received from a single episode that started in state s. In other words, it is the sum of all the rewards that the agent received from the start state until the end of the episode.

- $Returns(s, a)$ - A list that stores the observed returns (i.e., sum of rewards) that are obtained from following the policy and taking action a in state s. These returns are later used to update the action-value function Q(s,a) for the state-action pair (s,a). The list is maintained for each state-action pair

---

[1]Note that all of the pseudocode in this chapter is derived from here.

to keep track of the returns obtained from that state-action pair across different episodes.

- $Q(s, a)$ - The expected cumulative reward an agent would receive if it takes action a while in state s and follows a certain policy thereafter. It is a function that maps a state-action pair to a scalar value. The value of Q(s, a) is updated iteratively as the agent interacts with the environment and learns from experience.X

- π - particular policy the agent is following. Furtheremore, $\pi(a|S_t)$ represents the probability of taking action a at state S_t and $\pi(S', a)$ represents the probability of taking action a at state S' under a given policy π.

Algorithm 1 On-policy first-visit Monte Carlo

Require: small > 0
 1: Initialize
 2: $Q(s, a) \leftarrow$ initial optimistic value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
 3: $Returns(s, a) \leftarrow$ empty list, $\forall s \in S, a \in A(s)$
 4: **repeat**
 5:     Generate an episode following policy π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
 6:     $G \leftarrow 0$
 7:     **for** $t = T - 1, T - 2, \ldots, 0$ **do**
 8:         $G \leftarrow \gamma G + R_{t+1}$
 9:         **if** (S_t, A_t) does not appear in $S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$ **then**
10:             Append G to $Returns(S_t, A_t)$
11:             $Q(S_t, A_t) \leftarrow$ average($Returns(S_t, A_t)$)
12:             $A^* \leftarrow \operatorname{argmax}_a Q(S_t, a)$ ▷ with ties broken arbitrarily
13:             **for** all $a \in A(S_t)$ **do**
14:                 **if** $a = A^*$ **then**
15:                     $\pi(a|S_t) \leftarrow 1 - \epsilon + \frac{\epsilon}{|A(S_t)|}$
16:                 **else**
17:                     $\pi(a|S_t) \leftarrow \frac{\epsilon}{|A(S_t)|}$
18:                 **end if**
19:             **end for**
20:         **end if**
21:     **end for**
22: **until** convergence

In Chapter 5 of this work, we will provide a detailed discussion of the specific Monte Carlo algorithm implementation employed.

4.2   Temporal Difference Learning

Temporal Difference (TD) learning is another type of reinforcement learning algorithm that is, similarly to Monte Carlo, model-free. The main idea behind it is to update the estimated value of a state or action based on the difference between the expected return and the actual return obtained from that state or action.

TD learning is similar to Monte Carlo methods in that it learns from experience by interacting with the environment and observing the rewards received. However, these methods update their estimates after every time step, rather than waiting for an entire episode to complete like in MC methods. This makes Temporal Difference learning more efficient in terms of the amount of data needed to learn a good estimate of the value function.

Like Monte Carlo methods, TD methods can also be on-policy or off-policy. In on-policy learning, the agent learns about the value of the policy it is currently following, whereas in off-policy learning, the agent learns about the value of a different policy.

One important parameter in TD learning is the **step size** or **learning rate** (usually denoted by the symbol $\boldsymbol{\alpha}$), which determines the size of the update to the value estimates. A larger step size will result in faster learning, but may also make the learning process more unstable.

In this chapter, we shall introduce four distinct TD learning algorithms, namely SARSA, Q-Learning, Expected SARSA and Double Q-Learning [Sutton and Barto, 2018]. We shall illustrate that while SARSA and Q-Learning are prominent algorithms for control problems, Expected SARSA and Double Q-Learning are variations that cater to specific limitations of the original algorithms. The objective is to highlight both the commonalities and differences among them.

4.2.1 SARSA

SARSA stands for State-Action-Reward-State-Action. With this algorithm, the agent learns the value of a state-action pair $Q(S', a)$ by estimating the expected return over all possible actions from state S'. SARSA is an on-policy algorithm, meaning it learns the value of state-action pairs while following the same policy used to select actions. This makes it well-suited for control problems, where the goal is to find an optimal policy.

Algorithm 2 SARSA

1: Initialize
2: $Q(s, a) \leftarrow$ initial optimistic value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
3: **repeat**
4:     Initialize S
5:     Choose A from S using policy derived from Q (e.g., ϵ-greedy)
6:     **repeat**
7:         Take action A, observe R, S'
8:         Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
9:         $Q(S, A) \leftarrow Q(S, A) + \alpha\Big[R + \gamma Q(S', A') - Q(S, A)\Big]$
10:        $S \leftarrow S'$
11:       $A \leftarrow A'$
12:     **until** S is terminal
13: **until** convergence

4.2.2 Q-learning

Q-learning, unlike SARSA, is an off-policy algorithm that learns the value of a state-action pair Q(s,a) by estimating the maximum expected return over all possible actions from state s. In other words, it learns the value of the best action in each state. Since Q-learning is an off-policy algorithm it learns the optimal action-value function regardless of the current policy being followed. This makes Q-learning more flexible in terms of exploration and can result in faster convergence to the optimal policy. However, Q-learning tends to overestimate the value of actions in environments with high variance, which can lead to suboptimal policies. On the other hand, SARSA is more stable and less prone to overestimating the value of actions. Nevertheless, it can converge to suboptimal policies if the exploration is insufficient, and it can take longer to converge to the optimal policy compared to Q-learning.

Algorithm 3 Q-learning

1: Initialize
2: $Q(s, a) \leftarrow$ initial optimistic value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
3: **repeat**
4:     Initialize S
5:     Choose A from S using policy derived from Q (e.g., ϵ-greedy).
6:     **repeat**
7:         Take action A, observe R, S'
8:         $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A) \right]$
9:         $S \leftarrow S'$
10:     **until** S is terminal
11: **until** convergence

4.2.3 Expected SARSA

Expected SARSA is another off-policy TD algorithm that learns the value of a state-action pair Q(S',a) by estimating the expected return over all possible actions from state S', taking into account the probabilities of selecting each action according to the current policy. Expected SARSA can be seen as a compromise between SARSA and Q-learning, as it considers the value of both the current and the best action in each state. This algorithm considers all possible actions and their expected values, which makes it more robust to noisy or uncertain rewards.

4.2.4 Double Q-learning

Double Q-learning is a variant of Q-learning that uses two action-value functions to estimate the value of each action. The two functions are updated independently, and the final action-value estimate is the average of the two estimates. In Q-Learning, a single estimate of the action values is used to update the policy and make decisions. This means that when selecting an action in the next state, we always select the action with the highest estimated value (here we do not take into account using ϵ-greedy policy), even if that estimate is not accurate. This can

---
Algorithm 4 Expected SARSA

1: Initialize
2: $Q(s,a) \leftarrow$ initial optimistic value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
3: **repeat**
4:     Initialize S
5:     **repeat**
6:         Choose A from S using policy derived from Q (e.g., ϵ-greedy)
7:         Take action A, observe R, S'
8:         expected$Q \leftarrow \sum a\pi(S',a)Q(S',a)$
9:         $Q(S,A) \leftarrow Q(S,A) + \alpha \cdot [R + \gamma\text{expected}Q - Q(S,A)]$
10:         $S \leftarrow S'$
11:     **until** S is terminal
12: **until** convergence
---

result in overestimation of the true value of that action, particularly in situations where the policy is still exploring the environment. Double Q-Learning addresses the overestimation issue in Q-Learning which can lead to more accurate value estimates and better performance in some cases. However, it is possible that both estimates may be overestimating or underestimating the true value of an action, which can still lead to biased estimates.

Algorithm 5 Double Q-learning

1: Initialize
2: $Q(s,a) \leftarrow$ initial optimistic value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
3: **repeat**
4:     Initialize S
5:     **repeat**
6:         Choose A from S using policy derived from $Q_1 + Q_2$ (e.g., epsilon-greedy)
7:         Take action A, observe R, S'
8:         **if** rand() < 0.5 **then**
9:             $A' \leftarrow \text{argmax}_a Q_1(S',a)$
10:             $Q_1(S,A) \leftarrow Q_1(S,A) + \alpha[R + \gamma Q_2(S',A') - Q_1(S,A)]$
11:         **else**
12:             $A' \leftarrow \text{argmax}_a Q_2(S',a)$
13:             $Q_2(S,A) \leftarrow Q_2(S,A) + \alpha[R + \gamma Q_1(S',A') - Q_2(S,A)]$
14:         **end if**
15:         $S \leftarrow S'$
16:     **until** S is terminal
17: **until** convergence
---

5. Implementation of the Agents

ADDED FROM 4

The `eps` and `epsFinal` were discussed in Chapter 3. With these parameters, the user can indicate starting and ending value of the ϵ which will then adequately decrease after each played game. The formula for the decrease is the following:

$$decrease = (\frac{epsFinal}{eps})^{\frac{1.0}{n}}$$

Here, n represents the number of games being played. The reason behind this epsilon decrease is to change the ratio between the exploration and exploitation over time. In the beginning of the experiment, the `eps` value is higher and thus random moves happen more often, causing the agent to try actions it would otherwise oversee. Later, then the policy is a bit stabilized, the `eps` value becomes smaller so it would allow the agent to play longer games and possibly win (if the `eps` value was high throughout the whole experiment, the agent would have a bigger chance of choosing an inadequate move and thus untimely ending the game).

In this project, the RL agents are designed such that their functionality is encapsulated within a top-level scene called `Main.tscn`. Within this scene, there is an instance of the selected agent, and the code makes use of several functions implemented by the agent in order to interact with the environment. Specifically, the required functions are: `move()`, `init()`, `start_game()`, `end_game()`, `save()`, `set_seed_val()`, `get_and_set_agent_specific_parameters()`, and `get_n()`.

The purpose of most of these functions is self-explanatory. `init()` and `save()` are used to initialize and save the agent's internal state, respectively, and are called only once per experiment. `start_game()` and `end_game()` are called at the beginning and end of each episode, while `move()` is called by the `Tunnels.gd` script, and it is in this function that the agent makes a decision about which action to take based on the current state and score. The remaining functions pertain to the internal structure of the agent and are not relevant to the reader.

5.1   Hierarchy

In the current design of the game, there are a total of 8 agents implemented, 5 of which utilize some form of reinforcement learning (RL) algorithm. These RL agents share a common superclass called "Learning Agent", while 3 of them are further subclassed under the "TD Agent" class (see Figure 5.1). As previously discussed, the RL algorithms can be broadly divided into two categories: Monte Carlo methods and temporal difference (TD) learning. The TD algorithms differ only in their update function, and so it was deemed appropriate to group them under the same superclass. However, the Double Q-Learning agent, which utilizes separate policies and requires additional modifications, was implemented as a separate subclass or the "Learning Agent". The implementation details of these agents will be further elaborated upon in the subsequent sections.
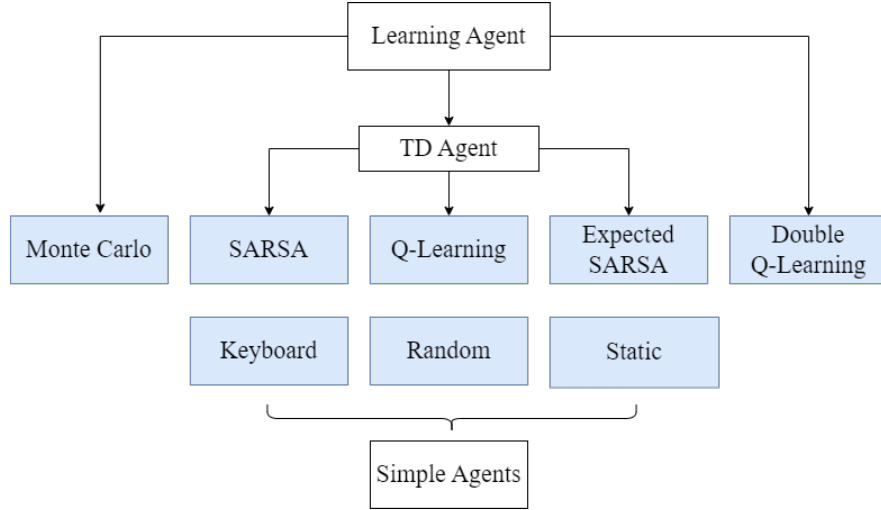
Figure 5.1: Agents hierarchy inside the project

5.2 Simple Agents

To facilitate testing of the game environment, several simple agents were implemented. These agents serve as baseline models and are used to ensure that the environment is functioning as intended before more sophisticated RL agents are developed. There are three simple agents in total: a "Keyboard agent" that receives input from the player via the keyboard, a "Static agent" that always chooses the forward action, and a "Random agent" that chooses a random action at each time step.

5.3 Learning Agent

The Learning Agent class serves as a base class for all the reinforcement learning agents in this project. It provides a set of shared functions and features that are used by all agents, such as reading and writing data to a file and debugging statements. In terms of decision-making, these agents all follow an epsilon-greedy policy, whereby they select the action with the highest value for a given state with a certain probability, or randomly choose any action with the remaining probability. Each of the subclasses of the Learning Agent class then implements specific code that is unique to that particular agent.

5.3.1 Monte Carlo Agent

$$R = next\_step.score - curr\_step.score$$
$$G = \gamma^{next\_step.time-curr\_step.time} * (R + G)$$
$$total\_retun[state\_action] = total\_retun[state\_action] + G$$

Figure 5.2: Total return update for Monte Carlo

26

The Monte Carlo method is a type of reinforcement learning algorithm that updates its policy only after an episode is completed. This is done by iterating through the entire episode and increasing the number of visits and total return for each state-action pair, if this is their first visit inside this episode. The total return is calculated using the formula shown in Figure 5.2, while the number of visits is simply incremented by 1. To determine the optimal action, the agent compares the ratio of total return to number of visits for each possible action at a given state. This calculation is performed at each state transition during the episode [1]. The γ variable in the equation shown in 5.2 serves as a discount factor, meaning that the last move made, which resulted in the agent's death, will receive the highest penalty. As we move further down the list of moves, their significance decreases. It is important to note that if the value of γ is set to 1, all moves are given equal weight. The `R` variable represents the return value, which is calculated as the difference in scores between two steps.

5.3.2   TD Agent

Unlike the Monte Carlo methods, which update their policies only after the completion of an episode, TD agents update their policies in real-time, after each action is taken. To accomplish this, all TD agents have a shared function called `move()`, which contains the update formula shown in Figure 2. However, the specific implementation of this formula varies slightly between the different TD algorithms. As visible on the Figure 5.3 the update of the policy for a particu-

$$alpha = 1.0/visits[state\_action]$$

$$gamma = \gamma^{next\_step.time-curr\_step.time}$$

$$q[state\_action] = q[state\_action] + alpha*(gamma*(R+new\_state\_val) - q[state\_action])$$

Figure 5.3: Policy update for Temporal Difference Learning

lar state value in TD learning requires several variables. The first is the overall number of visits, represented by the variable alpha and divided by 1. The second variable is gamma, which serves the same purpose as in the Monte Carlo method. Lastly, the `new_state_val` variable, which is unique to each agent, is needed for the update. Different methods of calculating the `new_state_val` variable can be seen in Figure 5.4. In SARSA, the `new_state_val` is calculated based

$$\text{SARSA: } new\_state\_val = q[state_{new_a}ction]$$

$$\text{Q-Learning: } new\_state\_val = q[state_{best_a}ction]$$

$$\text{Expected SARSA: } new\_state\_val = \sum_a \pi(action|state)Q(state, action)$$

Figure 5.4: new_state_val variable for individual TD Agents

---

[1]Instead of calling the move function periodically, the agents will always choose the same action based on the current state. Only once the state has changed, the new action is chosen based on the accumulated score and the new state.

on the value of the next action the agent will take, denoted as `new_action`. On the other hand, Q-learning uses the value of the best action possible in the next state, denoted as `best_action`. These two variables are equal if greedy policy is implemented. However if we consider ϵ-greedy policy, then they might differ based on weather a random action has been chosen. Expected SARSA combines these two approaches by taking the expected value of all possible actions in the next state.

5.3.3 Double Q-Learning Agent

$$gamma = \gamma^{next\_step.time - curr\_step.time}$$

if probability <0.5:

$$new\_state\_val = q2[get\_state\_action(state, best\_action)]$$

$$q[state\_action] + = alpha * (gamma * (R + new\_state\_val) - q[state\_action])$$

else:

$$new\_state\_val = q[get\_state\_action(state, best\_action)]$$

$$q2[state\_action] + = alpha * (gamma * (R + new\_state\_val) - q2[state\_action])$$

Figure 5.5: Policy update for Double Q-Learning

Similar to the agents in the TD class, the update for the Double Q-Learning agent occurs each time the agent changes its state. The update process is slightly different. In this method, two separate action-value functions, denoted as Q1 and Q2, are used to estimate the maximum action value for a given state. At each update step, one of the Q-values is selected randomly and updated using the other Q-value as a reference. This process helps to reduce the overestimation of action values and leads to more stable learning.

6. Experiments

In this chapter, we will evaluate the performance of the various agents when confronted with different combinations of obstacles. One of the key questions we aim to answer is whether any of the agents are able to successfully learn to play the entire game.

6.1 Discounting

Before diving into the results of the experiments, it is important to note that the discount factor was found to be inconsequential during the experimentation process. Specifically, the agents demonstrated better performance when the discount factor, represented by the variable gamma, was set to 1 (indicating that each step carries equal weight) as opposed to a value less than 1. This outcome is surprising as it would be intuitive to assume that the obstacle in the final step of the episode, which resulted in the agent's death, should be met with a more severe punishment compared to the obstacles encountered earlier in the episode.

6.2 Individual Traps

Triangles	→	6		X	→	11
I	→	6		O	→	13
MovingI	→	6		Balls	→	15
HalfHex	→	6		Walls	→	20
HexO	→	7		Hex	→	20

Figure 6.1: `rots` value used for each trap type

To start off let us analize each individual type of trap and how the agents perform under the same conditions. For this purpose we have performed experiments for each agent and each trap type with the hope that they can learn to play in such an environment in under 50 games. In figure 6.1 we can see the number of rotations used for each individual trap type. The aim was to minimize this number so that the state space itself would be minimized. To calculate how many states there are in an experiment, it is sufficient to multiply the number of obstacles by the number of possible rotations and distances. However, it is important to mention that throughout this experimentation, `dists` value was always 1 since all the obstacles were successfully trained by at least one agent with that number of distances.

In the Figures 6.2 and 6.3, we can observe the average performance of each agent in 50 games for individual traps and with different initial optimistic values. Both of these experiments utilized an epsilon value of 0.2. It is evident that the Double Q-Learning agent had the lowest performance among the agents tested in these experiments. However, it is important to note that when trained on a larger number of games, this agent demonstrated improved performance (as seen
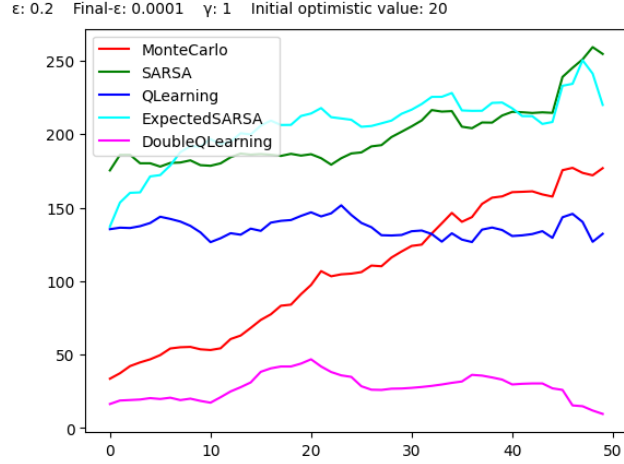
Figure 6.2: Average performance of the agents on individual traps, initOptVal=20
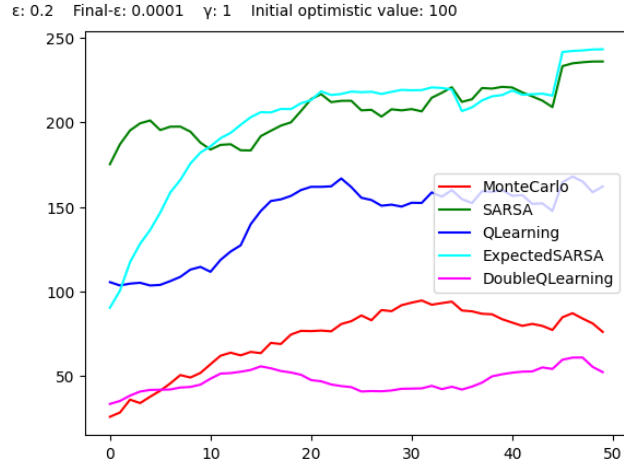
Figure 6.3: Average performance of the agents on individual traps, initOpt-Val=100

in Figure 6.5[1]).

Once peculiar case while training on individual traps is shown in Figure 4. If we look more closely to the policy the agent has chosen, it is visible that instead of aiming to reach a specific rotation value, it decided to rather quickly switch in between two rotations as, in this case, it will always be a safe option. This is indeed the optimal policy for this trap type under six rotations.

6.3   Bugs

In this section, we will analyze how the Bugs environment responded to training with different agents. As seen in Figure 6.6, the SARSA and Expected SARSA

---

[1]The table on the figure shows the move agent will choose in a certian state. Columns represent rotation value while rows show us which trap and dists value the state has. The value of a particular row and column, then, is an arrow pointing to which direction the agent will move. In case the agent chooses to shoot, a little * will be shown next to the arrow
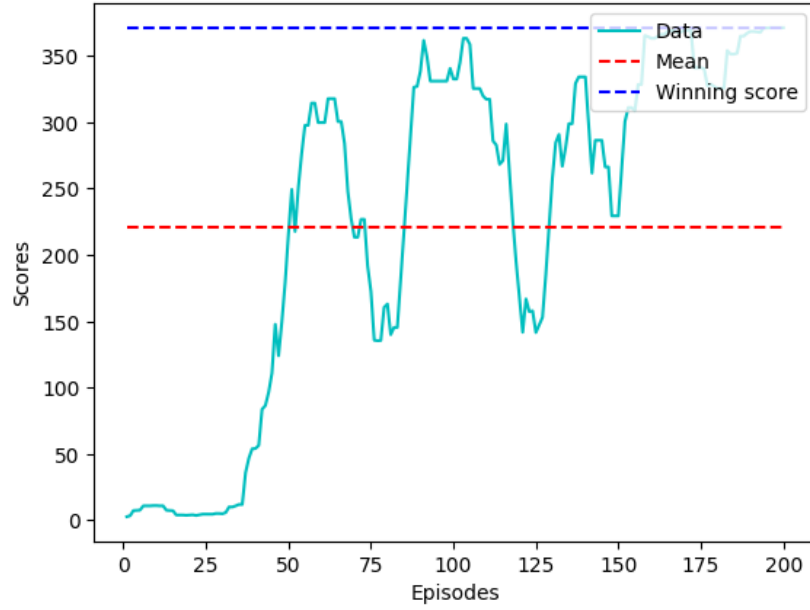
Figure 6.4: Double Q-Learning trained with 200 games on I-type trap

agents had the best overall performance in this environment. Interestingly, when shooting was enabled, the policies trained by these algorithms were the same as when shooting was disabled (see Figures 6.7 and 6.8). This result might be counter intuitive since the shooting gains point for the player and, since we did not have energy tokens in this environment, the agent was allowed to shoot indefinitely.

6.4 Viruses

The results of the experiments on the viruses environment indicate that the Expected SARSA and Q-Learning algorithms performed the best. While Q-Learning resulted in the same policy with both shooting enabled and disabled, the Expected SARSA algorithm produced two distinct policies depending on the presence of shooting (see Figures 6.10, 6.11 and 6.12). Even when given the option to shoot, the agent preferred to avoid the obstacle instead. A comparison of the performance of all algorithms can be seen in Figure 6.9.
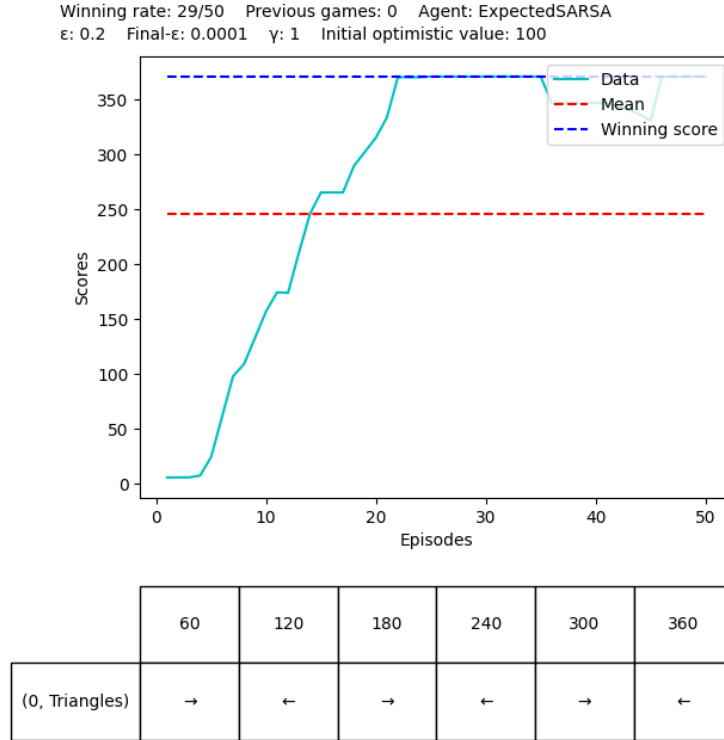
Figure 6.5: Triangle-type trap with Expected SARSA algorithm

6.5 Tokens

The results of the experiments conducted in the token environment show that all of the agents had a high success rate in winning the game. Experiments themselves have been performed on 20 games only, since this is the simplest environment we can observe in this project. The comparison of the performance of the different algorithms can be seen in Figure 6.13.

6.6 Combinations

In this section, we will examine the performance of the agents in a more complex environment with a larger group of obstacles. This analysis will provide a transition between the simpler experiments and the most challenging task of training on a full game.

6.6.1 Traps

As previously discussed, the algorithms generally demonstrated satisfactory performance when facing individual traps. However, when confronted with all trap types simultaneously, only two algorithms were able to successfully find an optimal policy. It is plausible that with more training or alternate parameters, all algorithms would be capable of reaching an optimal policy. Nonetheless, in these experiments, Monte Carlo and Expected SARSA demonstrated particularly strong performance. For visual representations of the policies developed by these algorithms, see Figures 6.14 and 6.15.
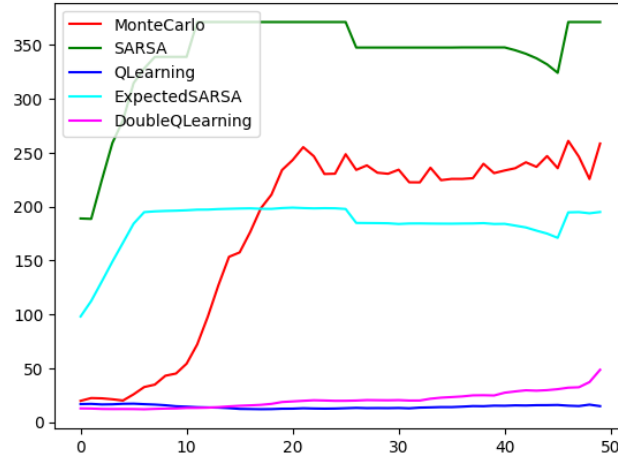
Figure 6.6: Average performance of the agents with Bugs environment

6.6.2 Bugs, Viruses and Tokens

As previously demonstrated, the Bugs, Viruses, and Tokens agents individually exhibited excellent performance. However, when these agents were combined in our experiments, only the Expected SARSA agent was able to determine an optimal policy for this environment. This policy, depicted in Figure 6.16, prioritizes avoiding obstacles rather than shooting them down. The agent appears to prioritize safety over the potential for a higher score by shooting down a bug or virus, as this strategy carries the risk of energy depletion. This preference for a risk-averse approach is consistent with the agent's previous behavior.

6.7 All obstacles

Despite its strong performance in previous experiments, the Expected SARSA algorithm did not successfully learn the optimal policy for the whole game during this experimentation. While the policy it developed did win some games, it was not a significant amount. It is possible that different parameters may have resulted in a different outcome, but further research would be necessary to confirm this. The results of this experiment can be seen in Figure 6.17.

	60	120	180	240	300	360
(0, LBF)	→	→	↑	↑	←	←
(0, LBW)	→	→	→	→	←	←
(0, W)	←	→	→	→	←	←
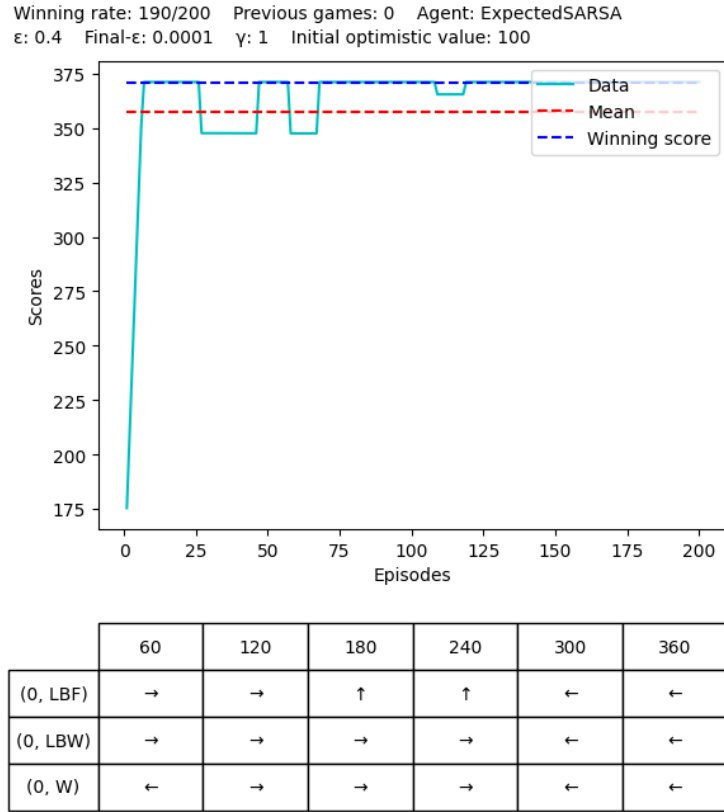
Figure 6.7: Policy of the Expected SARSA algorithm on the Bugs environment

	60	120	180	240	300	360
(0, LBF)	→	→	↑	←	←	←
(0, LBW)	→	←	↑	←	←	→
(0, W)	→	→	→	↑	↑	←
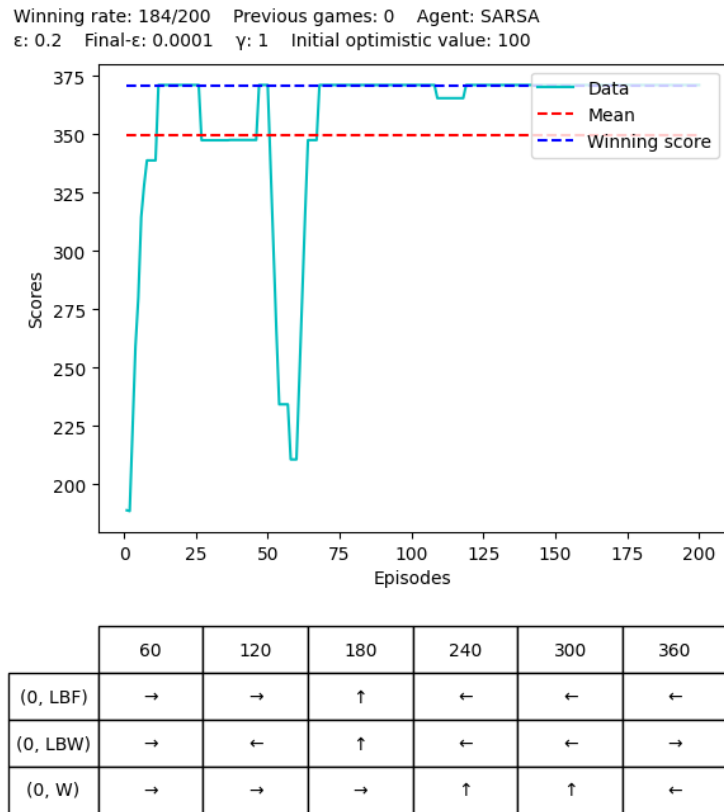
Figure 6.8: Policy of the SARSA algorithm on the Bugs environment
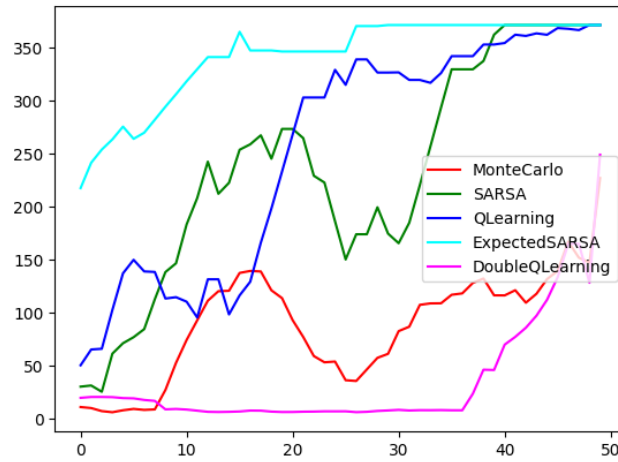
Figure 6.9: Average performance of the agents with Viruses environment

Figure 6.10: Policy of the Q-Learning algorithm on the Viruses environment
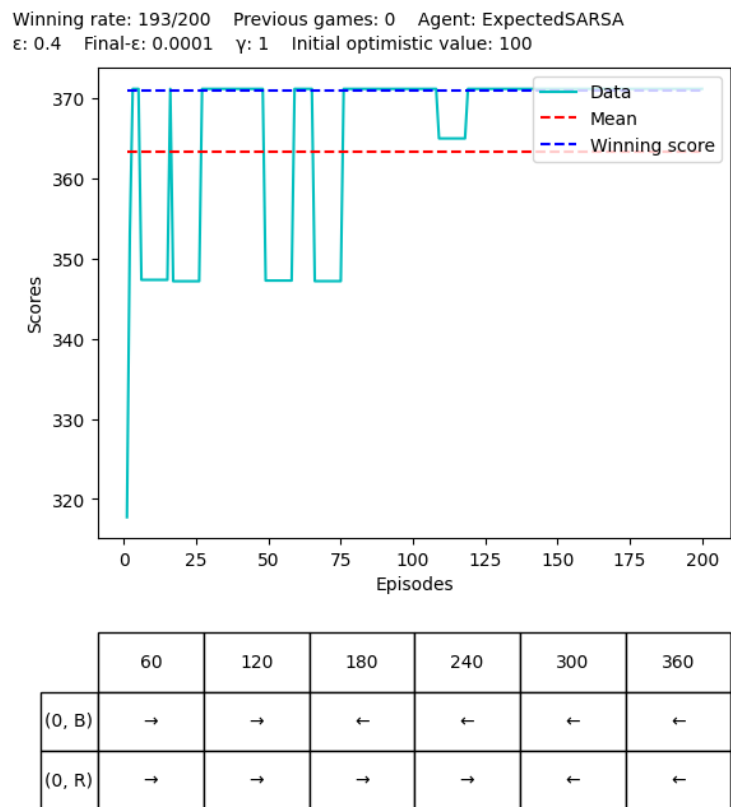
Figure 6.11: Policy of the Expected SARSA algorithm on the Viruses environment without shooting
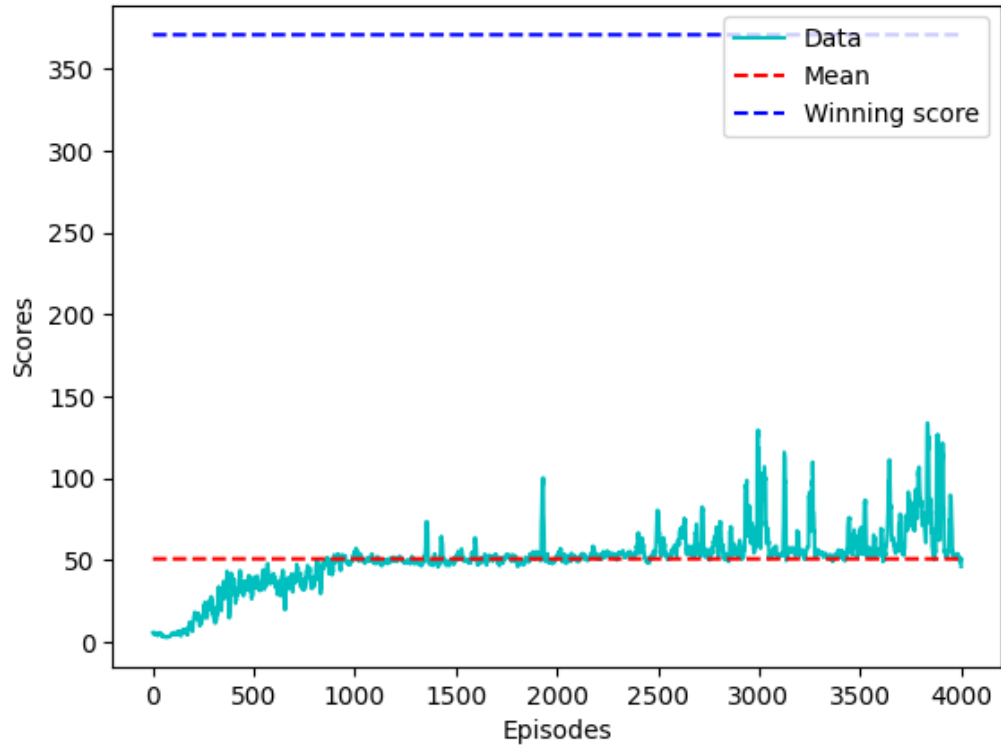
Winning rate: 82/100   Previous games: 0   Agent: ExpectedSARSA
ε: 0.4   Final-ε: 0.0001   γ: 1   Initial optimistic value: 100
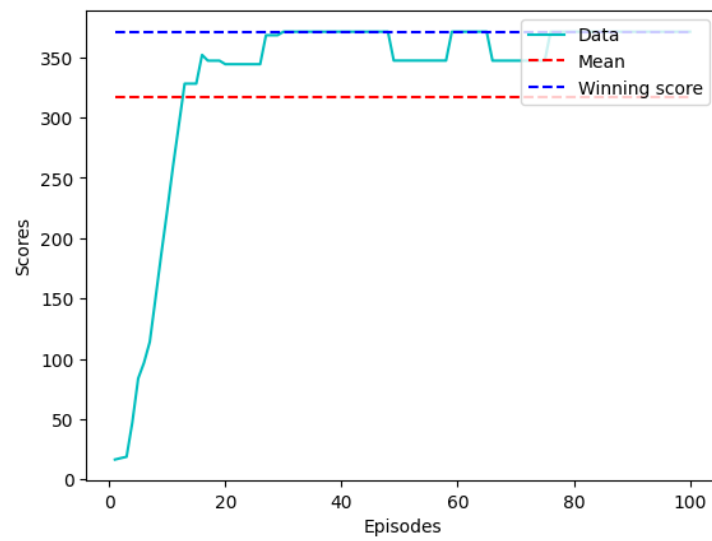
	60	120	180	240	300	360
(0, B)	←	→	→	←	←	←
(0, R)	→	→	↑	←	←	←

Figure 6.12: Policy of the Expected SARSA algorithm on the Viruses environment with shooting
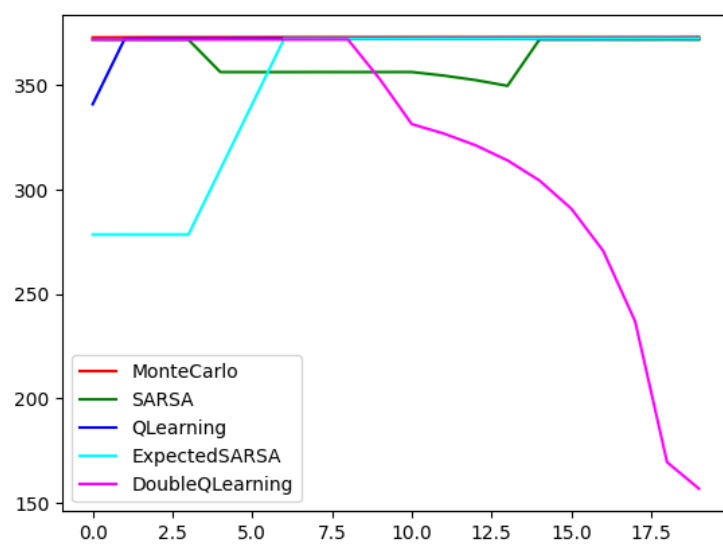
Figure 6.13: Performance comparison of the algorithms on the Token environment

37

Figure 6.14: Policy of the Expected SARSA algorithm on the Traps environment

Winning rate: 276/4000   Previous games: 0   Agent: MonteCarlo
ε: 0.2   Final-ε: 0.0001   γ: 1   Initial optimistic value: 30
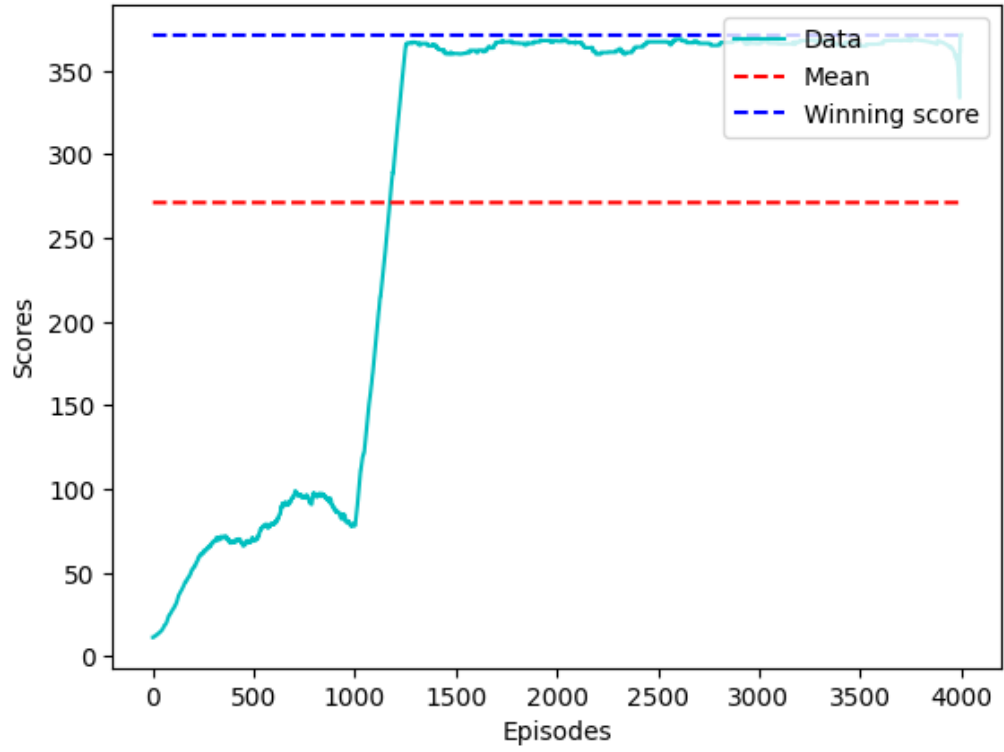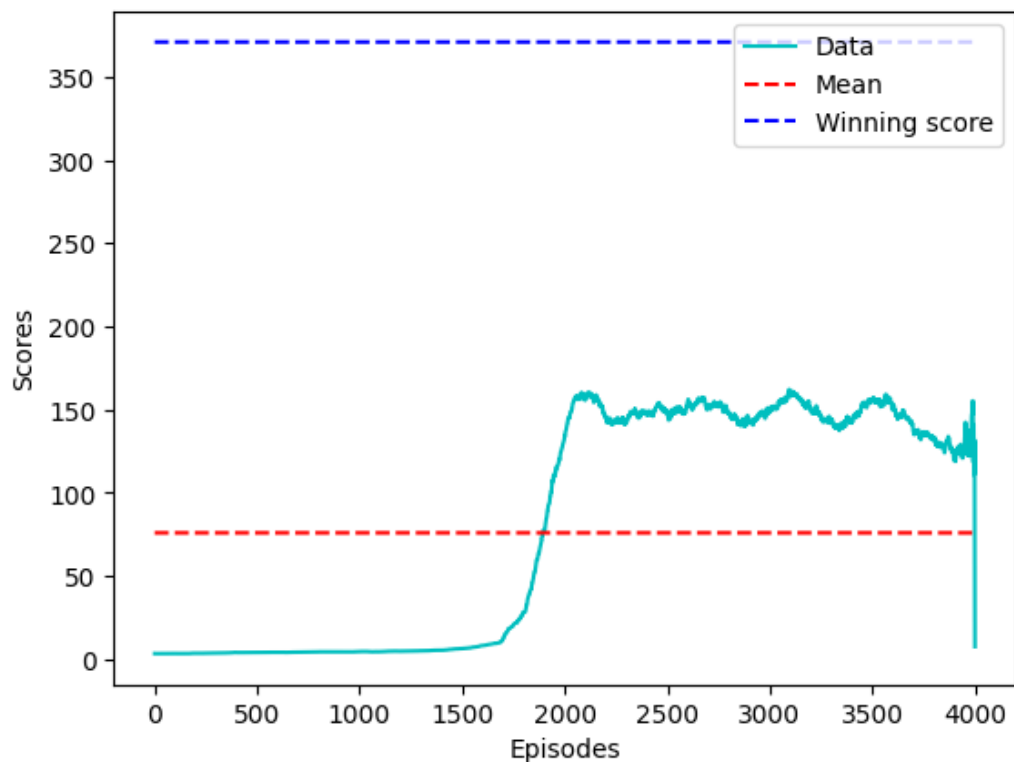
	12	24	36	48	60	72	84	96	108	120	132	144	156	168	180	192	204	216	228	240	252	264	276	288	300	312	324	336	348	360
(0, Balls)	→	→	↑	→	←	←	↑	→	↑	→	←	←	→	←	←	←	←	←	←	←	←	←	←	←	←	→	→	→	→	→
(0, HalfHex)	→	→	←	←	↑	←	↑	↑	↑	↑	↑	→	→	←	←	←	←	←	←	←	←	←	→	→	→	→	→	→	→	→
(0, Hex)	↑	←	←	←	←	←	←	←	←	←	→	→	→	→	↑	←	→	←	←	←	→	→	→	→	→	→	→	→	↑
(0, HexO)	←	←	←	→	→	→	→	→	→	→	→	→	→	→	↑	←	↑	←	←	←	←	←	←	←	←	←	←	←	←
(0, I)	↑	↑	→	↑	←	→	↑	←	←	←	→	→	→	↑	↑	←	←	↑	→	↑	→	↑	←	←	→	→	→	→	↑	↑
(0, MovingI)	→	→	→	↑	↑	→	→	→	↑	↑	←	→	←	←	←	←	→	→	↑	←	↑	←	↑	↑	↑	→	↑	←	←	←
(0, O)	→	↑	↑	←	←	←	←	←	→	→	→	↑	↑	←	←	←	→	→	→	→	→	↑	↑	←	→	←	←	→	→	→
(0, Triangles)	←	→	→	→	→	↑	→	→	→	→	→	→	→	↑	→	←	↑	↑	→	→	→	→	→	↑	↑	→	↑	←	←	←
(0, Walls)	→	←	←	←	←	←	←	→	→	→	→	→	→	↑	↑	←	→	←	←	←	←	→	→	→	→	→	→	↑	→	→
(0, X)	↑	←	→	→	→	→	→	→	↑	←	←	←	→	→	→	↑	↑	↑	↑	←	→	→	→	→	↑	←	←	←	→	↑

Figure 6.15: Policy of the Monte Carlo algorithm on the Traps environment

Figure 6.16: Policy of the Expected SARSA algorithm on the Bugs, Viruses and Tokens environment combined

Figure 6.17: Policy of the Expected SARSA algorithm on the whole game

Conclusion

In conclusion, this thesis has examined the performance of different reinforcement learning algorithms on various environments in the Godot game engine. The results showed that the Expected SARSA algorithms performed consistently well multiple environments, while the Double Q-Learning algorithm struggled with some but improved with more training. All algorithms performed quite well with the individual traps, while with all trap types combined, Monte Carlo and once again Eexpected SARSA gave the best results. In the token environment, all algorithms performed well. A very interesting observation is that in all cases that include bugs and viruses, the agents chose to rather avoid the obstacle than shoot it down. Unfortunately, none of the algorithms were able to achieve optimal performance on the full game. Further research is needed to determine the optimal parameters and strategies for these algorithms to accomplish that task.

Bibliography

Tunnel rush, 2022. URL `https://tunnelrush2.com/`.

Una Adilovic. Space run ai, 2022. URL `https://github.com/AdilovicUna/Space-run-AI`.

Juan Linietsky et al. Godot engine - documentation, 2021. URL `https://docs.godotengine.org/en/stable/index.html`.

Ton Roosendaal et al. Blender, 2022. URL `https://www.blender.org/`.

Khronos Group. gltf - runtime 3d asset delivery, 2022. URL `https://www.khronos.org/gltf/`.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, London, England, second edition, 2018.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment