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 would like to express my heartfelt appreciation to my supervisor, Adam Dingle, for his invaluable guidance and support throughout my academic journey, beginning with our first Programming 1 class and continuing through to the completion of this project. His patience and assistance were greatly appreciated and played a significant role in the success of this work.

In addition, I am deeply thankful to my mother for her unwavering belief in me and to my grandparents who made it possible for me to be here. Their contributions are greatly appreciated and will never be forgotten.

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.

Keywords: tunnel game, reinforcement learning, artificial intelligence, algorithms

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 [2023]). 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

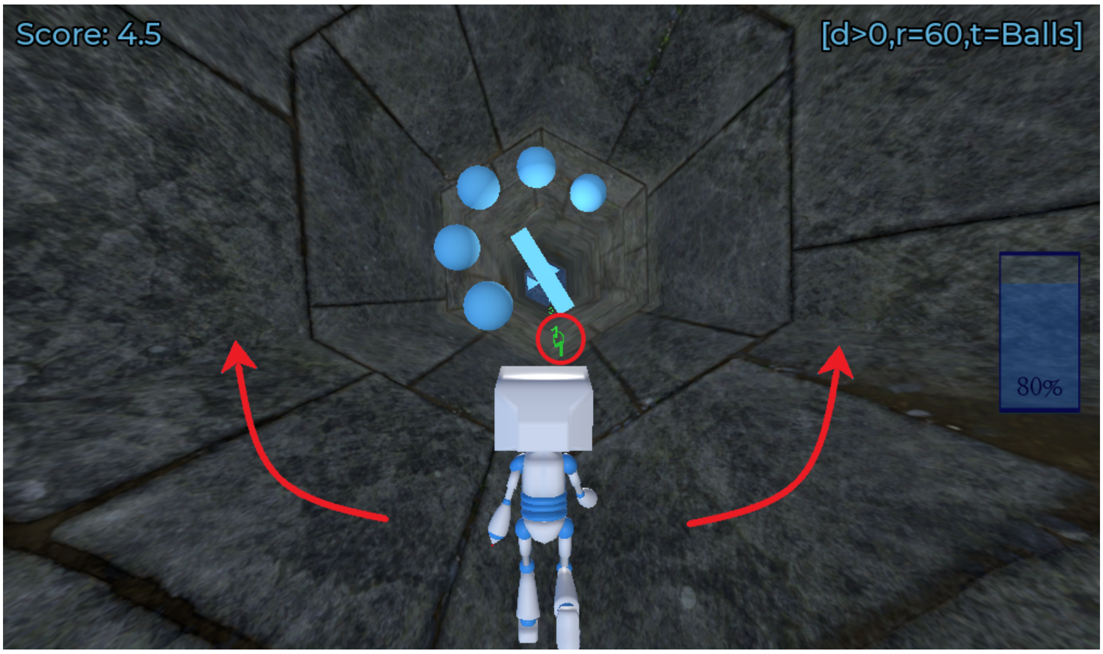


Figure 1.1: Movement

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.

Figure 1.2: Hans

1.2 Obstacles

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

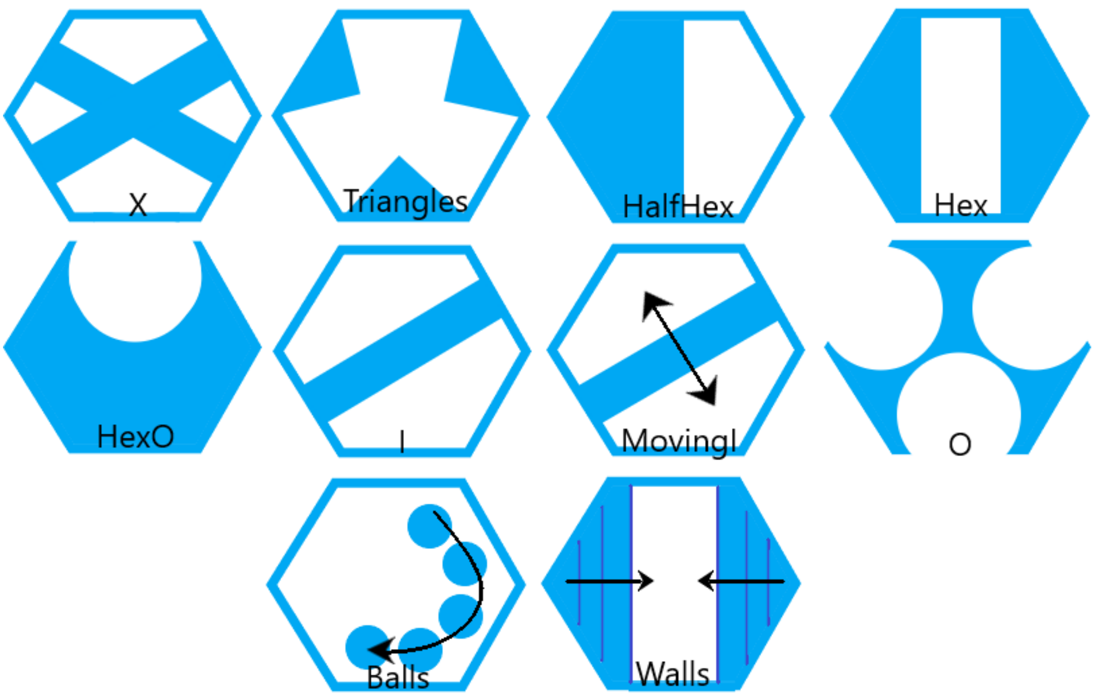


Figure 1.3: Traps

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. In Figure 1.3 the arrows show in which direction the animated traps move. The traps can be encountered in any of the three tunnels, and if the player fail to successfully evade them, they result in an instant death.

1.2.2 Bugs

In addition to traps, the game also features bugs as an obstacle. In Figure 1.4, going from left to right, we see LadybugWalking, Worm and LadybugFlying bugs. They appear in the second tunnel and are designed to rotate around the tunnel toward the player's position, making them more challenging to evade. Nevertheless, it is still possible to avoid these obstacles. 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, eventually leading him to loose the game (for more information on battery life, see Section 1.3).

Figure 1.4: Bugs

1.2.3 Viruses

The third and final type of obstacle in the game are viruses (illustrated in Figure 1.5). These viruses are 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.

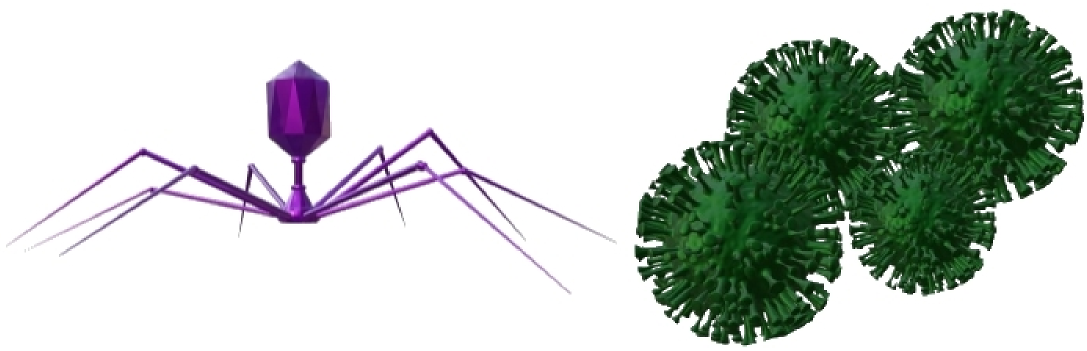


Figure 1.5: Bacteriophage and Rotavirus

Figure 1.6: Battery and Energy Token

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.

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 15.

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 (Linietsky [2021]).

In addition to using the Godot Engine, I also utilized Blender (version 6.2.0) (Roosendaal [2023]) for creating and animating the characters in the game. Blender is a popular open-source 3D modelling and animation program 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 (Khronos [2023]) 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 (Adilović [2023]).

2.1 The top-level organization

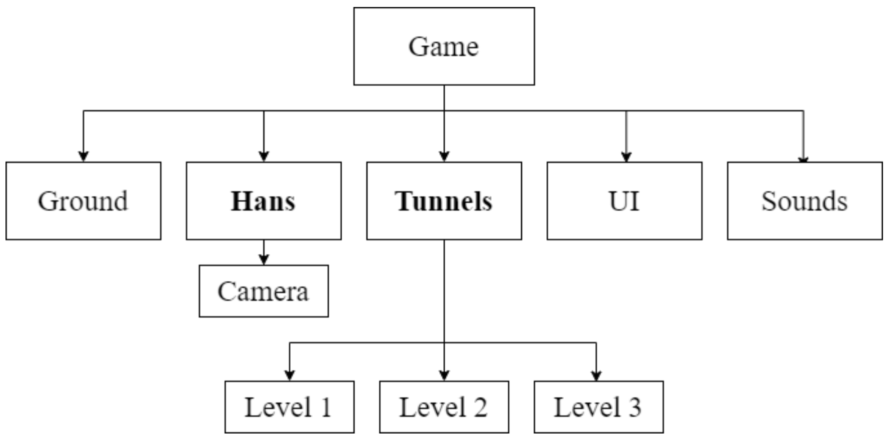


Figure 2.1: Structure of Game.tscn

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 `Hans.gd` and its key components.

The primary function within `Hans.gd` is `_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

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.

```
func physics_process(delta):
    tunnels.delete_obstacle_until_x(...)
    if translation.x < new_trap:
        create_new_trap()

    score._on_Meter_Passed() # update score
    velocity = Vector3.LEFT * speed
    velocity = move_and_slide(velocity)
    check_collisions()
    tunnels.bug_virus_movement(delta, curr_tunnel)

    if isShootingButtonPressed:
        shoot()

    state.update_state(...)
```

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.

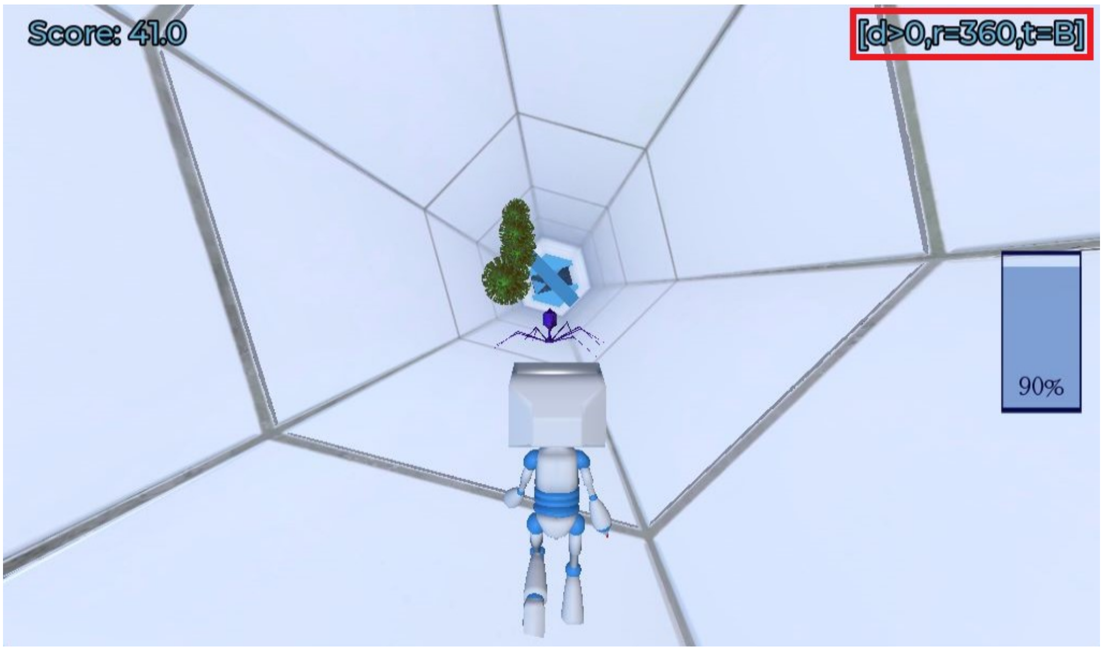


Figure 2.2: State

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.

```
func physics_process(delta):
    var move = game.agent.move(...)
    if move[0] == 1:
        tunnel = get_child(hans.get_current_tunnel())
        tunnel.rotate_object_local(Vector3.RIGHT, -ROTATE_SPEED *
            delta)
    elif move[0] == -1:
        tunnel = get_child(hans.get_current_tunnel())
        tunnel.rotate_object_local(Vector3.LEFT, -ROTATE_SPEED *
           delta)

        # If it is not instanced we can't switch animation
    if hans != null:
                # Shoot if move[1] returned 1
        hans.switch_animation(move[1] == 1)
```

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 the user's input from the keyboard.

```
func create_first_level_traps(tunnel):
    var level = tunnel
    var num_of_traps = rand_range(MIN_TRAPS_PER_TUBE,
       MAX_TRAPS_PER_TUBE)
    x = position of the first trap

    for n in range(num_of_traps):
        # update x positon
        x -= rand.randi_range(TRAP_RANGE_FROM,TRAP_RANGE_TO)
        if x is outside the tunnel:
            break
        create_one_obstacle(level, x)
```

[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.

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.

```
func create_one_obstacle(level, x):
    var scene = pick_which_kind_of_obstacle_will_be_added
    var tunnel = get_the_level_we_are_making_traps_for
    var i = randomly_pick_an_obstacle

    # make an instance
    var obstacle = scene[i].instance()
    obstacle.translation.x = x

    tunnel.add_child(obstacle)
    rotate_obstacle(obstacle)
```

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.

```
func delete_obstacle_until_x(level, x):
    var tunnel = get_current_tunnel()
    for obstacle in tunnel.get_children():
        if obstacle is an obstacle type:
            if obstacle.translation.x > x:
                obstacle.queue_free()
            else:
                return
```

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

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 discrete learning 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.

It is important to note that the game uses meters as the standard unit of distance measurement in Godot. This unit is used to represent the size, position, and movement of objects in the game world. Hans is approximately 12 meters tall and has starting speed of 35 meters per second. Additionally, each tunnel in the game has a width and height of approximately 45 meters and a depth of 2800 meters.

The distance value indicates Hans' 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 more than 50 meters away from the obstacle and somewhere between 1 and 50 meters away from the obstacle, respectively. If the `dists` parameter is set to 1, the distance value remains constant at >0. In the Chapter 7 we will see that all types of environments in this game are able to learn when the `dists` parameter is equal to 1.

The tunnel circumference is divided into into a fixed number of intervals determined by the command-line parameter `rots` (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 obstacle type value indicates the type of the next obstacle ahead of Hans in the tunnel. Each obstacle has its own unique string representation, which allows the agent to learn to recognize the safe rotation for different types of obstacles. Using this information, along with the distance parameter, the agent can decide whether to move left, right, forward, or shoot in combination with any of these actions. To further explain this notion let's look back at Figure 3.1. At first glance, it may seem that a rotation of 240 would allow Hans to easily

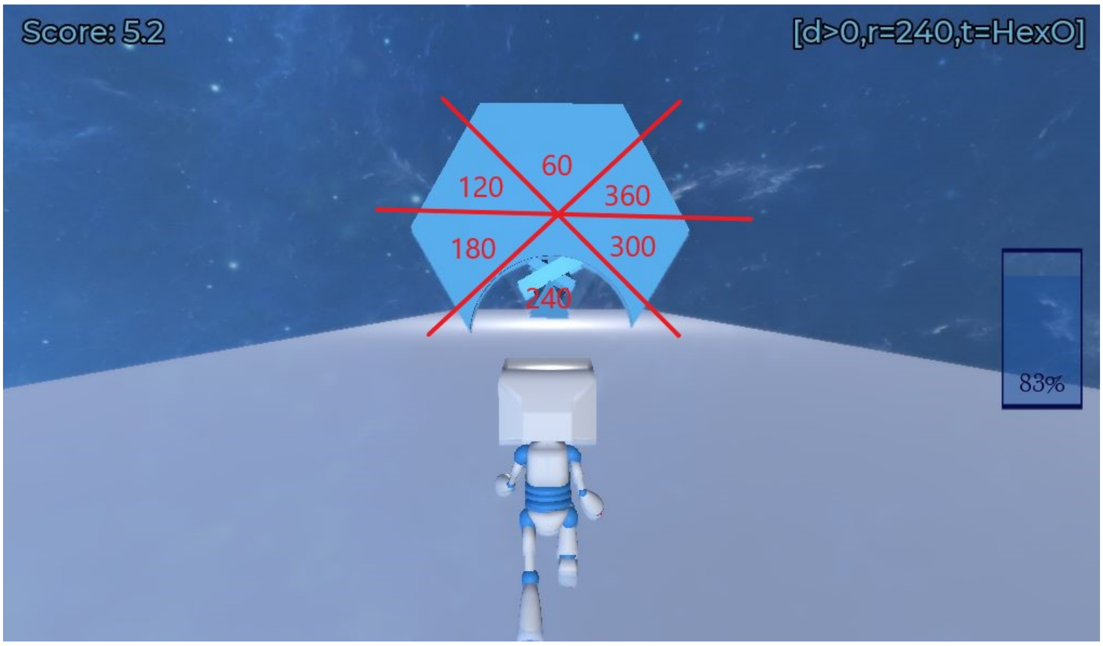


Figure 3.1: Rotations when `rots = 6`

pass through the obstacle. However, upon closer inspection, it becomes clear that only a specific portion of the 240 rotation value is safe. The agent does not know the difference between being at any point of the 240 rotation and thus can easily make a fatal mistake. 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).

In Chapter 7, a noteworthy concept is presented where the agent can recognize a safe edge between two rotations, even if it is not provided with a completely safe rotation. By quickly shifting between these rotations, the agent is able to successfully navigate through the obstacle.

3.2 Command-line options

n=int	number of games
stoppingPoint=int	stop after the agent wins this many consecutive 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 of rotation
agentSeedVal=int	seed value for the random moves the agent takes
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

3.2.1 Command-line option descriptions

In this section, a more comprehensive clarification for the table presented earlier is provided. It is important to note that any of the options listed can be left out, as they all have default values assigned to them. In the event that no options are specified, a regular game with the `Keyboard` agent will be executed.

n

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

stoppingPoint

If the agent manages to win this many consecutive games, the experiment is stopped. This is used to avoid long experiments if the agent has already learned the right policy and would simply keep winning. The default value is 25.

agent

Name of the desired agent.
Options: [`Keyboard`, `Static`, `Random`, `MonteCarlo`, `SARSA`, `QLearning`, `ExpectedSARSA`, `DoubleQLearning`]
sub-options (only for the learning agents):
[`gam` (range [0,1]), `eps` (range [0,1]), `epsFinal` (range [0,1]), `initOptVal` (range $[0,\infty)$)]
Example usage: “`agent=MonteCarlo:eps=0.1,gam=0.2`”
The meaning of the suboptions is explained in Section 5.3.1.

level

Number of the level to start from. Default value is 1.
Options: `[1, ... , 15]`
Note: after the 15th 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`, `forced`]
This option is disabled by default or if the environment does not have any bugs or viruses.

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 of rotation. This parameter is part of the state label and the minimum viable option is 6. This is also the default value.

agentSeedVal

Seed value for the agent.
This parameter is used to produce multiple experiments with different random actions to verify if the agent can really learn under certain conditions or if it has simply been "lucky" with the random moves.

database

Read or write data for this command from/to a file.
The results of the experiments are generally stored in two folders:
Agent_databases and Command_outputs. So by data, we mean storing the final policy of the agent inside the Agent_databases and score of each game and the final policy inside Command_outputs. If continuous evaluation is performed, it stores only the evaluation games. The files from the former folder can be used start another session of the agent's training from the last point of the previous session or to run a game with visuals and observe the agent's performance, while the files from the latter are used for plotting the results.
Options: [`read`, `write`, `read_write`]
Note: 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. The program achieves this speedup by hiding the CSG geometry in every node and performing a few other tricks. 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. 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].

The experiments we conducted in this study are entirely reproducible owing to the predetermined seed values for all the random variables. However, it should be noted that the values produced by a seed can differ across different versions of the Godot Engine. In order to obtain identical results to the experiments detailed in Chapter 7, we recommended to use Godot Engine v3.2.3.stable.

To run the program in the command line, the user should add the directory containing 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:

```
$ godot database=write agent=SARSA:initOptVal=100.0,eps=0.3 env=
    HexO n=10 dists=1 rots=8
```

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:

```
$ godot --no-window --fixed-fps 1 --disable-render-loop database=
    write agent=SARSA:initOptVal=100.0,eps=0.3 env=HexO n=10 dists
    =1 rots=8
```

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

[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.

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 one or more game sessions. There are several key functions within the `Main.gd` script that are worth discussing in more detail.

```
func _ready():
    var unparsed_args = OS.get_cmdline_args()
    if unparsed_args.size() == 1 and unparsed_args[0] == ''
        options'':
        display_options()

    ... # parse args

    if set_param(args) == false:
        display_options()
    else:
        instance_agent()
        build_filename()
        if not agent_inst.init(...):
            print(''Something went wrong, please try again'')
            display_options()
        play_game()
```

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.

```
func play_game():
    if agent == ''Keyboard'' and VisualServer.render_loop_enabled
        :
        ... # play a regular game
    elif n > 0:
        n -= 1
        game = game_scene.instance()
        set_param_in_game()
        agent_inst.start_game(is_eval_game)
    else:
        agent_inst.save(write)
        print_and_write_ending()
```

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 control of Hans. Otherwise, one of the remaining agents will take over and play the 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 from this particular session. Otherwise, a single game will be executed and number of games left decreased.

```
func on_game_finished(score, ticks, win, time):
    print_and_write_score(score, win)
    agent_inst.end_game(score, time)
    play_game()
```

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 - ϵ), 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 the relative effectiveness of different actions or states, serving as a measure to determine the optimal action in a particular 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 state s.
- 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. 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 to keep track of the returns obtained from that state-action pair across different episodes.

[1] Note that all of the pseudocode in this chapter is derived from the Sutton/Barto text.

- $Q(s, a)$ - 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.

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

Algorithm 1 On-policy first-visit Monte Carlo

Require: small $\epsilon > 0$
Initialize:
  $Q(s, a) \leftarrow$ initial optimistic value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
  $Returns(s, a) \leftarrow$ initial optimistic value , $\forall s \in \mathcal{S}, a \in \mathcal{A}s$
repeat
  Generate an episode following policy π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for $t = T - 1, T - 2, \ldots, 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    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**
      Append G to $Returns(S_t, A_t)$
      $Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$
    end if
  end for
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

Initialize:
  $Q(s, a) \leftarrow$ initial optimistic value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
repeat
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  repeat
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\Big[R + \gamma Q(S', A') - Q(S, A)\Big]$
    $S \leftarrow S'$
    $A \leftarrow A'$
  until S is terminal
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

Initialize:
    $Q(s,a) \leftarrow$ initial optimistic value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
repeat
    Initialize S
    Choose A from S using policy derived from Q (e.g., ϵ-greedy).
    repeat
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha\,[R + \gamma \max_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$
    until S is terminal
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.

Algorithm 4 Expected SARSA

Initialize:
    $Q(s,a) \leftarrow$ initial optimistic value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
repeat
    Initialize S
    repeat
        Choose A from S using policy derived from Q (e.g., ϵ-greedy)
        Take action A, observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha \cdot [R + \gamma \sum_a \pi(S',a)Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$
    until S is terminal
until convergence

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 an ϵ-greedy policy), even if that estimate is not accurate. This can 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.

Algorithm 5 Double Q-learning

Initialize:
    $Q(s, a) \leftarrow$ initial optimistic value $\forall s \in \mathcal{S}, a \in \mathcal{A}(s)$
repeat
    Initialize S
    repeat
        Choose A from S using policy derived from $Q_1 + Q_2$ (e.g., ϵ-greedy)
        Take action A, observe R, S'
        if rand() < 0.5 **then**
            $A' \leftarrow \mathrm{argmax}_a Q_1(S', a)$
            $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha[R + \gamma Q_2(S', A') - Q_1(S, A)]$
        else
            $A' \leftarrow \mathrm{argmax}_a Q_2(S', a)$
            $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha[R + \gamma Q_1(S', A') - Q_2(S, A)]$
        end if
        $S \leftarrow S'$
    until S is terminal
until convergence

5. Implementation of the Agents

In this project, the reinforcement learning (RL) agents 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 include: `move()`, `init()`, `start_game()`, `end_game()`, `save()`.

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.

5.1 Hierarchy

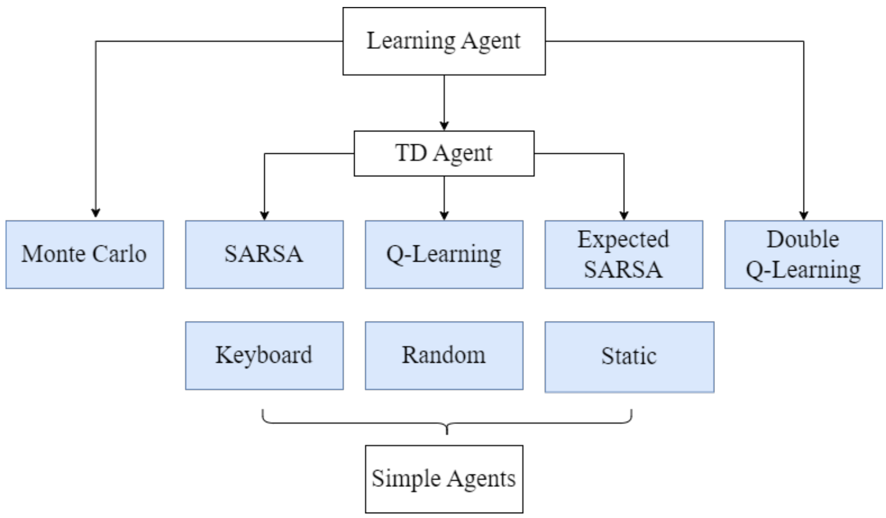


Figure 5.1: Agents hierarchy inside the project

In the current design of the game, there are a total of 8 agents implemented, 5 of which utilize some form of reinforcement learning algorithm. These RL agents share a common superclass called `LearningAgent`, while 3 of them are further subclasses under the `TDAgent` 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 uses separate policies and requires additional modifications, was implemented as a separate subclass of the `LearningAgent`. The implementation details of these agents will be further elaborated upon in the subsequent sections.

As previously mentioned when describing the state in Section 3.1, the agent has the ability to perform movements in various directions, including moving right, forward, or left, and each of these movements can be performed in combination with shooting. Therefore, on each time step, the move function of the agent returns a list of two elements: the first element signifies the movement direction, where a value of -1 represents left, 0 represents forward, and 1 represents right; while the second element determines whether the agent will shoot or not, with a value of 1 indicating yes and 0 indicating no for shooting.

5.2 Simple Agents

To facilitate testing of the game environment, several simple agents were implemented. These agents serve as baseline models and were used to ensure that the environment was functioning as intended before more sophisticated RL agents were 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 without shooting, and a "Random agent" that chooses a random action at each time step.

5.3 Learning Agent

This 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 ϵ-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 `LearningAgent` class then implements specific code that is unique to that particular agent.

```
func choose_action(action):
    epsilon_action = false
    if not is_eval_game: # Used for continuous evaluation
        epsilon_action = rand.randf_range(0,1) < EPSILON
        if epsilon_action:
            action = ACTIONS[rand.randi_range(0,len(ACTIONS) - 1)]
    return action
```

5.3.1 Common parameters and behaviours

In regards to the present implementation of the reinforcement learning algorithms for the 3D tunnel game, certain aspects of the game learning step and parameter implementation are unique to this project and merit discussion. For instance, we should look into the learning step in the game. The agent will not request a new move until the state has changed, despite the fact that it may seem more natural for a new decision to be made on every game tick. This has the effect of reducing the number of decisions that the agent must make in a given episode, but also results in intriguing policy behaviours that are elaborated upon in Chapter 7.

Furthermore, there are several parameters that are common to all learning agents, some of which were briefly discussed in the preceding chapter. In this section, we will examine in more detail how these parameters were integrated into this particular project. The initial optimistic value parameter is the simplest one to explain, as it is implemented in a straightforward manner. In particular, each time a state-action pair is added to the policy, its value is set to a predetermined number. This number (`initOptVal`), along with the other agent's sub-options parameters mentioned subsequently in this subsection, are specified through the command line (see 3.2).

The following two parameters worth noting are `eps` and `epsFinal`, which are responsible for the random moves executed by the ϵ-greedy policy. These parameters allow the user to specify the starting and ending values of ϵ. Then, at the end of each game, the new ϵ value is calculated by multiplying the current ϵ value with the *decrease* which is computed as follows[1]:

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

Here, `n` represents the number of games being played. The reason behind this epsilon decrease is to change the ratio between exploration and exploitation over time. At 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, when 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).

Finally, it is pertinent to discuss the discounting value γ (defined by `gam`). In this project, discounting is used in the following manner:

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

In many reinforcement learning environments, all learning steps have the same duration, so each reward is discounted by γ^n, where `n` is the number of steps that elapsed between the time when an action was taken and the time when the reward was received. And so in a typical implementation that iterates over the steps in an episode, the accumulated discount rate is multiplied by γ in each iteration. However, as previously stated, learning steps in this implementation do not occur on every tick, but instead occur when the state changes. As a result, they may vary in size. To avoid uneven discounting, the time (in milliseconds) is calculated for each new decision made by the agent using this formula:

$$(game.num\_of\_ticks * 33)/1000.0$$

5.3.2 Monte Carlo Agent

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

[1] This computation is performed once, when initializing the agent.

through the entire episode, going from the last step towards the first, 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 the code above, 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[2]. This calculation is performed at each state transition during the episode. To clarify, instead of calculating a new move each time the `move()` function is called, 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. This implementation has resulted in a certain behaviour of the agents which will be more discussed in Section 7.7. As previously mentioned, the γ constant in the equation shown in the code serves as a discount factor, meaning that the last move made, which resulted in termination of the game, 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.

```
# MonteCarlo agent update
var R = (next_step.score - curr_step.score)
G = pow(GAMMA,next_step.time - curr_step.time) * (R + G)

# since we are using the first visit approach,
# we only need the first occurrence  of this state_action
if is_first_occurrence (...):
    total_return[curr_step.state_action] += G
    visits[curr_step.state_action] += 1
```

5.3.3 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 calls the function displayed in the code provided below.

```
# TD agents update
visits[last_state_action] += 1
var alpha = 1.0 / visits[last_state_action]
var new_state_val = 0 if terminal else new_state_action
var new_gamma = pow(GAMMA,curr_time - prev_time)
q[last_state_action] += alpha * (new_gamma * (R + new_state_val)
   - q[last_state_action])
```

[2]The pseudocode in Algorithm 1 keeps a list of all returns for each state/action pair, while in our implementation a return for a particular state/action pair is calculated with the mentioned ratio.

The update of the policy for a specific state value in TD learning involves multiple variables, most of which have been previously discussed. One new variable is α, which represents one devided by the total number of visits for a given state action pair. Furthermore, to make this calculation, a variable uniquely computed by each TD algorithm, known as `new_state_action`, is required. Various methods for computing this variable can be found in the code-snippets below. If the current state is a terminal state, the `new_state_action` will not be used and instead, it will be replaced with a value of 0, as indicated in the update code.

```
# SARSA agent new_state_action variable calculation
func get_update(state, new_action, _best_action):
    return q[get_state_action(state, new_action)]
```

```
# QLearning agent new_state_action variable calculation
func get_update(state, _new_action, best_action):
    return q[get_state_action(state, best_action)]
```

```
# ExpectedSARSA agent new_state_action variable calculation
func get_update(state, _new_action, best_action):
    var sum = 0.0
    # Calculate and return the expected value
    for action in ACTIONS:
        var probability = EPSILON / len(ACTIONS)
        if action == best_action:
            probability += 1 - EPSILON
        sum += probability * Q(state,action)
    return sum
```

In SARSA, the `new_state_val` is calculated based 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 a greedy policy is implemented. However if we consider a ϵ-greedy policy, then they might differ based on whether 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.

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 one as a reference. This process helps to reduce the overestimation of action values and leads to more stable learning.

```
# DoubleQLearning agent update
visits[last_state_action] += 1
var alpha = 1.0 / visits[last_state_action]
var new_gamma = pow(GAMMA, curr_time - prev_time)

if rand.randf_range(0,1) < 0.5:
    var new_state_val = 0 if terminal else
        q2[get_state_action(state,.best_action(state,q))]
    q[last_state_action] += alpha * (new_gamma * (R +
       new_state_val) -q[last_state_action])
else:
    var new_state_val = 0 if terminal else
        q[get_state_action(state,.best_action(state,q2))]
    q2[last_state_action] += alpha * (new_gamma * (R +
       new_state_val) -q2[last_state_action])
```

6. Testing and Plotting

In this chapter we want to explain how the experiments were conducted and how to interpret the plots what will be used to showcase the results of those experiments in Chapter 7. We also want to show how to use the testing and plotting systems implemented as part of this project.

6.1 Conducting the experiments

The process of conducting experiments encountered several challenges and fluctuations. the primary concern was to determine the minimum number of rotations (`rots`) and distances (`dists`) required to make learning feasible for each obstacle type. After experimenting with up to 30 rotations in some cases, and not getting satisfying results with seemingly any combination of other parameters, it was determined that the game's difficulty in later levels was the root of the problem, as it was impossible to play with some environments(confirmed by human players). Consequently, the game had to be adjusted and thus, there are slight variations in parameters, such as the starting speed and distances between obstacles, in the version of the Space-Run game used in this thesis, as compared to the original. As per a human player's assessment, it is now possible to play all environment combinations until level 15 or even beyond. It is noteworthy that level 10 was the initial choice for the winning level, which was later shifted to level 15 to prevent the agent from settling for a mediocre policy and to find the optimal policy. However, this shift did not yield significant results, and the same behaviour could likely be achieved by increasing the number of games, allowing the ϵ-greedy policy to perform random moves more frequently for an extended period. Despite this, level 15 was used in all subsequent experiments.

LadybugFlying = 6	Rotavirus = 6	MovingI = 6	O = 13
LadybugWalking = 6	Tokens = 6	HexO = 7	Balls = 15
Worm = 6	HalfHex = 6	Triangles = 7	Hex = 22
Bacteriophage = 6	I = 6	X = 11	Walls = 22

Figure 6.1: `rots` values used for each obstacle type

However, for obstacles such as `Walls` and `Balls`, this method was not viable. The reason being that running `env=Balls` or `env=Walls` with visuals caused the game to lag significantly due to the number of animations playing simultaneously. As a result, `Walls` received the same number of rotations as `Hex` trap, while `Balls` received 15 rotations, the number at which the agent managed to learn. For bug, virus, and token obstacles, the default and minimum value of 6 rotations was assigned, with which they all trained successfully. Concerning the `dists` parameter, it was concluded during the experiments that all agents could learn any obstacle with `dists=1`, and increasing this number needlessly would only increase the number of states required for training.

After addressing the the issue of game not being playable, the most straightforward method for identifying the number of rotations required was to play the game manually with decreased speed. The results obtained from these experiments are displayed in Figure 6.1 and were used in all experiments conducted. However, this method was not viable for obstacles such as `Walls` and `Balls` due to game lag. Thus, we assigned the same number of rotations to `Walls` as to the **Hex** trap, and to `Balls`, we assigned 15 rotations, the number at which the agent learned. `Bug`, `virus`, and `token` obstacles were assigned a default and minimum value of 6 rotations, which proved to be sufficient for successful training. During experiments, it was observed that all agents could learn any obstacle with `dists=1`, and increasing this parameter needlessly increased the number of required states.

The remaining values to be determined for the experiments were the sub-options for each agent (see 3.2.1). Even thought some of the experiments were omitted from this study, they provided useful insights that could be utilized. For instance, in many of those experiments, the agent performed best with `eps` values ranging from 0.2 to 0.4, with 0.2 being the most common, in combination with an `epsFinal` value of 0.0001. The rationale for using a low `epsFinal` value is that towards the end, the agent almost exclusively exploits the current policy, and a more gradual decrease in epsilon values is suitable for larger values of `n`. Furthermore, `initOptVal` of 20.0 and 100.0 was promising in most experiments. The `gam` value will be discussed in a later subsection.

Throughout the months dedicated to conducting experiments, they gradually converged towards checking combinations of the values mentioned above. Due to their length, the experiments and the number of parameters requiring adjustment, were kept systematic towards the end. Unless specified otherwise, each combination was tested on ten seeds for each agent, with the combinations consisting of `eps` parameter taking a value of either 0.2 or 0.4, and the `initOptVal` being either 20.0 or 100.0. Discounting was kept at the value of 1.0 (see Section 7.7) and `epsFinal` was 0.0001.

Finally, it should mentioned how we shoose the number of games for an experiment. The general practice was to choose `n` 10 to 15 times greater than the number of obstacles the environment used for a particular experiment. This approach was found to be sufficient for facilitating the training of the agents. However, when incorporating the shooting actions, it was deemed appropriate to increase the value of `n` by approximately 25%. Although this increase appeared reasonable, there was no specific rationale behind it.

6.2 Interpreting the plots

In Chapter 7, a number of plots similar to the one shown in Figure 6.2 will be presented. This section has been dedicated to explaining the different components of these plots and how to read them. While some plots may differ slightly from the one in Figure 6.2, the meaning behind them can still be easily deduced.

The top part of the figure displays all the necessary information that was used in the experiment. Most of the values have been previously described, except for “Previous games”. This value is meant for experiments that used the `database=read` option and were performed on an agent that had trained for

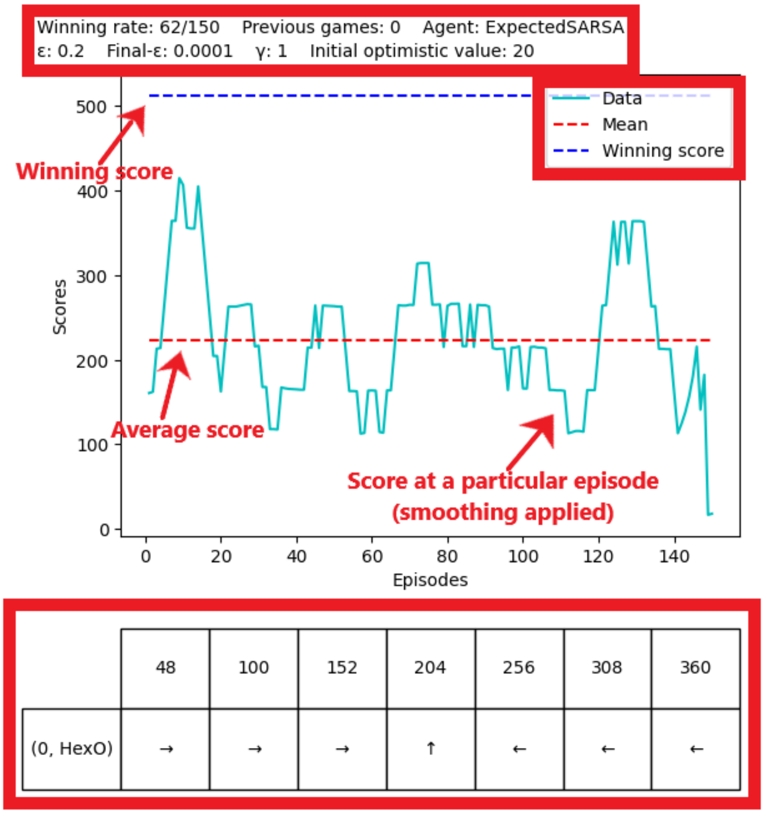


Figure 6.2: Plot example

some number of games. This number specifies the how many of games the agent had previously trained on.

Moving further down, there is a plot with three lines and a legend in the top right corner. The data line, as indicated on the figure, represents the score that the agent achieved on a particular episode. However, some plots may show the average value of the agent's score for each episode over several different seeds used for the random actions. Additionally, there may be multiple data lines on the plot, each averaged between many seeds and each representing a different agent. Every agent is labelled with their respective color inside the legend.

The mean line represents the average score value for the entire experiment, while the winning score represents the winning threshold which is the score that the agent achieves after passing level 15. This score will be constant (a little over 500), except in the cases where the environments contains any of the bugs and viruses and the agent is allowed to shoot. The winning score then depends on how many of those obstacles the agent successfully shoots. In this case the line will show the winning score of the last game in which the agent won. It should be noted that if the agent did not win any games, or the plot is displaying multiple agents, the winning score line will be omitted. Furthermore, the data line is averaged to appear smoother. This is why even though the winning rate is 62/150 in this sample plot, the data line doesn't touch the winning score threshold at any point.

As mentioned in Section 3.2, it is possible to perform continuous evaluation on experiments, meaning one learning game is played with random actions, followed immediately by an evaluation game using only the current policy that the agent is performing. In all experiments conducted, the data line only represents the evaluation games.

At the very bottom of the figure, there is a table which only appears in plots that contain a single data line that is not averaged over different seeds and has only one seed value. In the table, all rotation values for this experiment define the columns, while each row has a tuple of distance value and type of the next obstacle ahead. Each cell then represents a particular discrete state, and the arrow it shows is the action that the agent takes in that state. If the arrow has * attached to it, it means that the agent will also shoot. Since all experiments are performed with `dists=1`, the number of rows will match only the number of different obstacles used in the experiment.

It should be emphasized that parameters such as the number of seeds in the averaged data line or the size of the smoothing window will be clearly specified for each plot mentioned in the rest of the chapter. This prevents any form of ambiguity or confusion regarding the details of the experiment.

6.3 Testing and plotting systems usage

To produce the experiments and plots for this study, two scripts were written - one for conducting tests and the other for creating plots. In this chapter, we provide a brief overview of how to use these scripts in case readers wish to replicate our experiments. Both scripts have an additional .txt file describing the process in more detail. Additionally, both of these systems produce a folder containing log files so that the user might see more closely what was happening during the run of either of the programs.

6.3.1 Testing

The `./test.py` script is designed to facilitate the running of multiple experiments at once, eliminating the need for users to manually run each experiment through the command line.

There are two types of variables: immutable and mutable. Immutable variables remain constant throughout all experiments, while mutable variables can be specified as a range of values. For each combination of mutable variables, the agent will run an experiment.

Immutable variables include `n, stoppingPoint, shooting, env, agent, m, level, database, ceval,` and `debug`, which are defined in Section 3.2. The variable `m` represents a range of `agentsSeed` values, with the default value of `m=[0,9]` (inclusive).

Mutable variables include sub-options for each agent, which are specified in the following format: `[min,max(not inclusive),step]`.

Additionally, there are top-level experiment options such as `all_traps`, `all_bugs`, and `all_viruses`, which run experiments for each individual trap, bug, or virus, respectively. There is also the option of `all_agents`, which runs experiments with each learning agent, and `all_shooting`, which runs experiments with both shooting on and off. These options overwrite equivalent immutable variables for `env`, `agent`, and `shooting`.

Note that the values for `rots` and `dists` for each `env` are hardcoded and cannot be changed.

Example usages are provided in the following code excerpts.

```
# print description options
$ python ./test.py --description
```

```
# for 10 seeds run 100 full games with MonteCarlo agent
# perform continuous evaluation and write the output to the
  database
$ python ./test.py
```

```
# for each agent 5 times run 50 games on the environment with X
  trap
$ python ./test.py --m=[0,4] --n=50 --all_agents --env=[X]
```

```
# for 10 different seed values run 800 games for all combinations
   of:
# each of the DoubleQLearning and SARSA agents,
# on the Bugs environment with epsilon value 0.2
# and each of the initOpt values 20.0 and 100.0
$ python ./test.py --n=800 --agent=[DoubleQLearning,SARSA] --env
  =[Bugs] --eps=[0.2,0.3,0.1] --initOptVal=[20.0,180.0,80.0]
```

6.3.2 Plotting

The `plots.py` script utilizes the files generated in the Command_outputs folder to plot the outcomes of the experiments. The folder and its contents are automatically produced when the write option of the database is enabled. These files contain scores for each game in a single experiment, the resulting policy, and other relevant information that are displayed on the plots as described in Section 6.2 of this chapter.

Compared to the testing script, this system is much simpler as it only involves 2 parameters. The first parameter, `window`, has a default value of 10 and determines the number of neighboring points that will be averaged to produce smoother plots.

The second parameter is `option` whose value can be either 1 or 2. Option 1 generates 1 plot for each file in the Command_outputs folder, while option 2 combines multiple files. In this case, for each distinct environment presented in the files, a single plot is created, with a data line for each agent averaged over multiple seeds. The number of seed values included and which agents are displayed depend solely on the accuracy of the files in the Command_outputs folder.

Example usage is depicted in the command below.

```
$ python ./plot.py --option=2 window=100
```

7. Experiments

In this chapter, we will evaluate the performance of the various agents when confronted with different combinations of obstacles, as well as presenting interesting observations made during the experiments. One of the key questions we aim to answer is whether any of the agents are capable of learning to play the entire game.

7.1 Individual traps environments

This section discusses the performance of agents in environments containing only a single type of trap, and no other obstacles such as bugs or viruses.

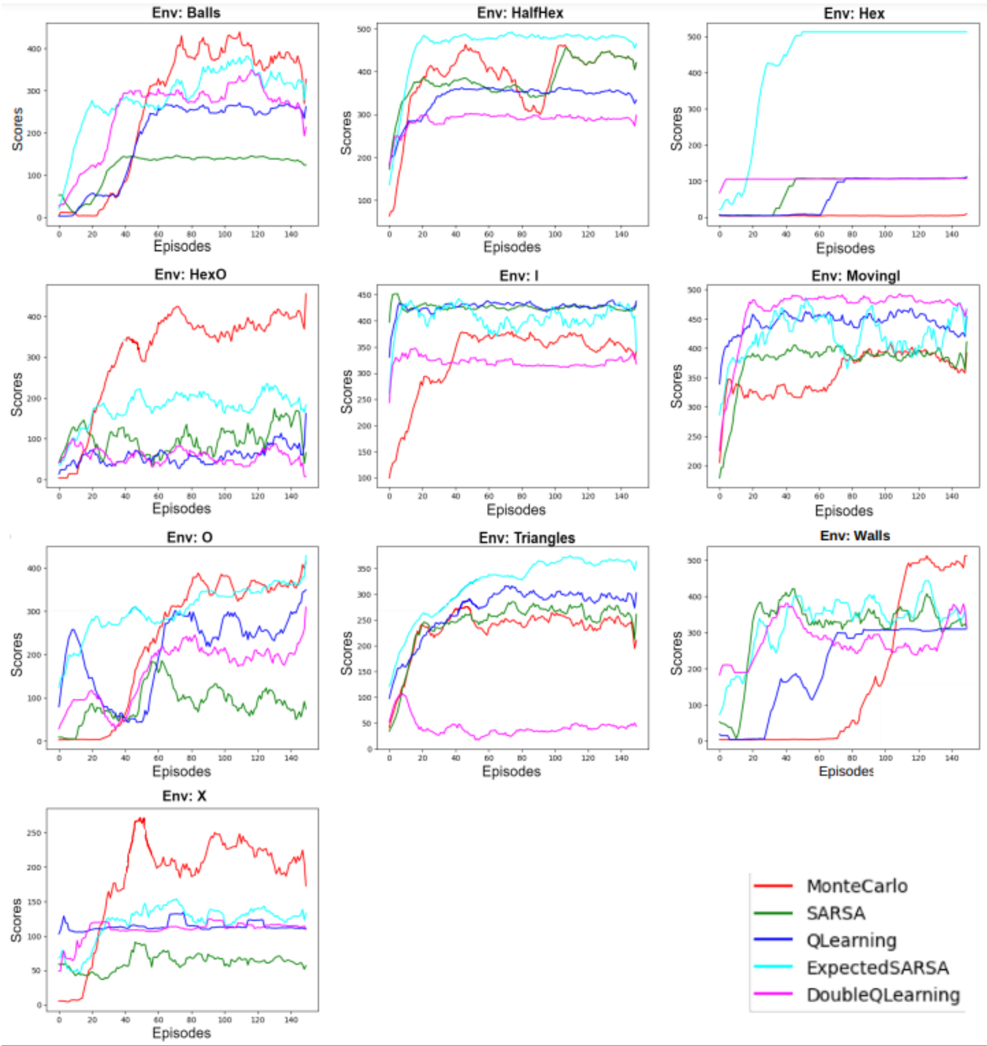


Figure 7.1: Performance of all agents in single-trap environments

In Figure 7.1, for each type of trap, there is a plot with `eps=0.2` and `initOptVal=20` and how each agent performed in an environment containing only that trap type[1]. These values were picked because with most traps the agents

[1]For the purpose of having all of these plots in one figure, they were manually modified.

performed well under these conditions.

It should be noted that all plots within this section have smoothing applied with `window=10`. Thus certain spikes may not be visible. Additionally, unless otherwise suggested, you can assume that the values that were produced are an average of 5 different seeds. The aim is to show a realistic picture on how the agent would perform, and not show the occurrences in which the outcome was satisfactory but rather account for the failures in reproducing the perfect policy as well.

In some cases, of course, the hyperparameters used in the experiments in Figure 7.1 were not ideal. However, for most trap/agent pairs, we managed to find at least one combination of hyperparameters where the agent found an optimal or a policy that won some games but not consecutively, across multiple seeds. The only exceptions were the `MonteCarlo` agent with the `Hex` trap, `DoubleQLearning` with the `HexO`, `Triangles`, and `X` traps, and `SARSA` and `QLearning` with the `HexO` trap. `ExpectedSARSA` was the only agent that produced a good policy on multiple occasions for all individual trap types and even performed exceptionally well with certain hyperparameters for the `Hex`, `I`, `MovingI`, `O`, and `Walls` traps, in which cases it found an optimal policy across multiple seed values.

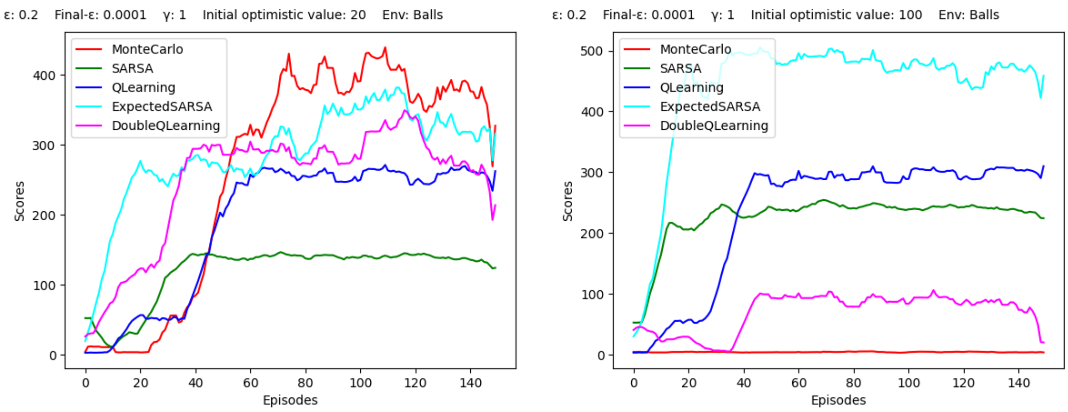


Figure 7.2: `Balls` trap experiments

Choices of the hyperparameters are a very important factor. Most experiments described in this section used the hyperparameter values `eps=0.2` and `initOptVal=20.0` or `initOptVal=100.0`. A good example of how much hyperparameters can influence the outcome is visible in Figure 7.2. Performance in the `Balls` trap environment varies significantly based on the initial optimistic value used. This plot underscores the importance of carefully selecting hyperparameters for reinforcement learning.

In some cases the aforementioned variations in hyperparameter selection lead to highly desirable outcomes. This is exemplified by the results presented in Figures 7.3 and 7.4, which demonstrate the efficacy of the `ExpectedSARSA` agent. Notably, on the right hand side of the both plots, the `ExpectedSARSA` agent was able to achieve optimal performance early on in the game with all seed values. The `ExpectedSARSA` agent, in a very large number of experiments, has outperformed its counterparts, sometimes by a significant amount. This is particularly evident

However, the calculations are still the same as described in the previous chapter.

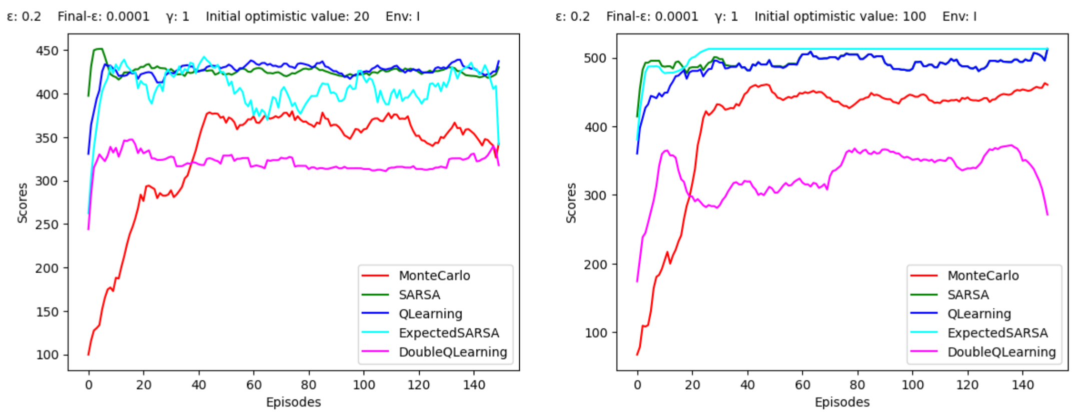


Figure 7.3: `I` trap experiments

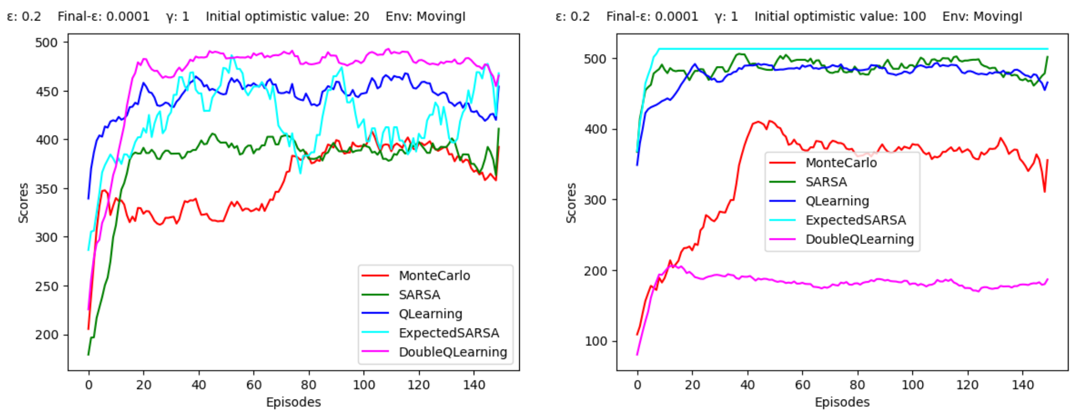


Figure 7.4: `MovingI` trap experiments

when considering its performance in simpler environments such as traps `I` and `MovingI`, where it is apparent that the agent is capable of learning extremely well. While other agents have performed well on these specific traps as well, their success may not be immediately apparent from the averaged results depicted in the plots.

Although performing these experiments with a larger number of seeds would ideally yield even more accurate results, we hope that the picture we presented provides a reasonable representation of the agents' performance in the demonstrated environments.

7.2 Traps environment

This section delves into the exploration of a highly intricate environment of all traps combined, which is the most complex one barring the full game. To clarify, the `Traps` environment contains all 10 trap types mentioned in the previous section, and omits any bugs, virus or token type of obstacles. As depicted in Figure 7.5, the majority of agents were unable to perform optimally in this environment. `ExpectedSARSA` was the only agent able to learn an optimal policy for almost

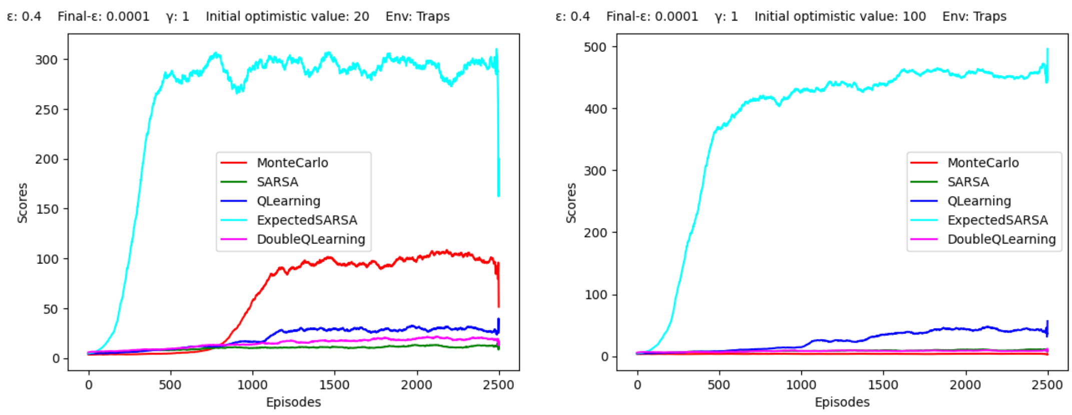


Figure 7.5: All traps experiments

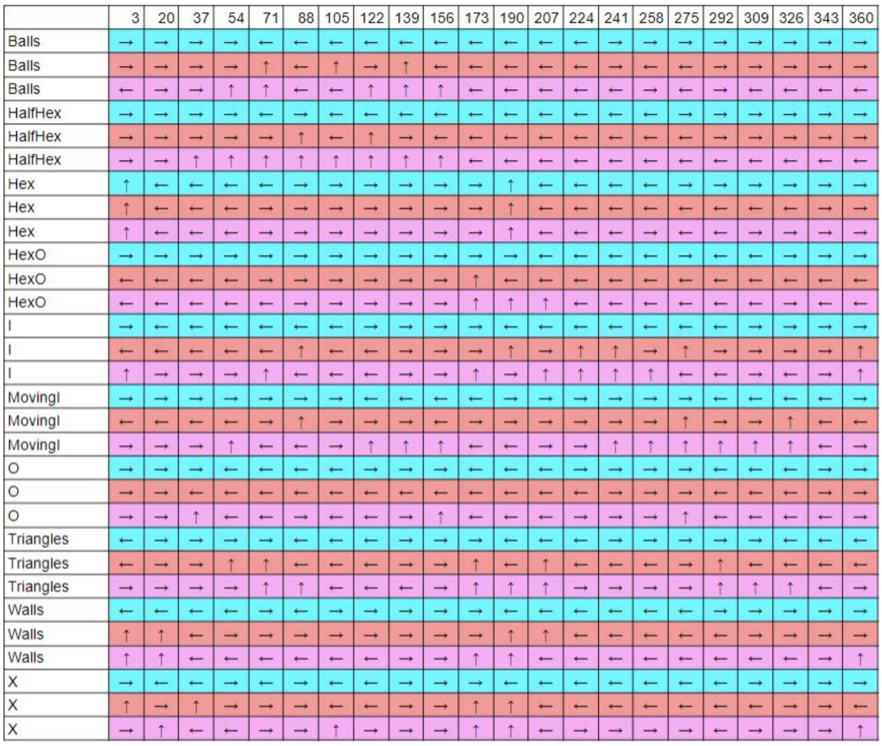
all random seeds tested, as evidenced by its score nearing 500 in the right plot[2], which was the approximate winning score value in all experiments. Moreover, `ExpectedSARSA` not only managed to learn an optimal policy once but did so with different hyperparameters and seed values on multiple occasions, leading to winning streaks of 30 games and early termination of the experiment. This outcome is the most favourable for any environment. The figure displays the averaged value of 9 seeds for all agents under the specified parameters.

The experiments conducted for this environment were systematic and involved matching commonly used `eps` and `initOptVal` to test if the agents could learn. In these experiments, the difference in learning between the `ExpectedSARSA` agent and the others is even more pronounced. However, on the left plot visible in Figure 7.5, where `eps=0.4` and `initOptVal=20`, the `MonteCarlo` agent performed reasonably well, attaining an average score of approximately 100, which is substantially superior to the other agents, except for `ExpectedSARSA`.

When training in the all-traps environment we set the `rots` parameter to 22. That's because this environment contains the `Hex` and `Walls` traps, which require a minimum of 22 rotations for learning to be feasible at all. This means that our experiments in this environment had many more states than in any single-trap environment. Considering that in this case we have 10 different trap types, there are 220 states with `Traps` environment. For that reason we picked `n=2500` for all of the experiments in this section. That's because in our experiments we've generally found that learning is most successful when the number of episodes is at least 10 times the number of states. As a result of having this many `rots` values, with some simpler traps, there can be many safe rotations that the agent can choose from. For that reason, going forward could be viable in multiple adjacent states, when the next trap ahead requires for example `rots=6` when trained individually.

The table in Figure 7.6 provides a comparison of policies from three different experiments, each reproduced with only one seed value, that yielded a policy that managed to win enough times to have an average score between 100 and 200 or more. The purpose is to see how far off the agents were from an optimal policy. The blue rows represent the `ExpectedSARSA` agent, in an experiment performed

[2]The plots for this environment have been smoothed with `window=100`.



	3	20	37	54	71	88	105	122	139	156	173	190	207	224	241	258	275	292	309	326	343	360
Balls	→	→	→	→	←	←	←	←	←	←	←	←	←	←	→	→	→	→	→	→	→	→
Balls	→	→	→	→	↑	←	↑	→	↑	←	←	←	←	←	→	←	←	→	→	→	→	→
Balls	←	→	→	↑	↑	←	←	↑	↑	↑	←	←	←	←	←	→	←	→	←	←	←	←
HalfHex	→	→	→	→	←	→	←	←	←	←	←	←	←	←	←	←	→	→	→	→	→	→
HalfHex	→	→	→	→	→	↑	←	↑	→	←	←	←	←	←	←	←	←	→	→	→	→	→
HalfHex	→	→	↑	↑	↑	↑	↑	↑	↑	↑	←	←	←	←	←	←	←	←	←	←	←	←
Hex	↑	←	←	←	←	→	→	→	→	→	→	↑	←	←	←	←	→	→	→	→	→	→
Hex	↑	←	←	←	→	→	→	→	→	→	→	↑	←	←	←	←	←	←	←	←	→	→
Hex	↑	←	←	←	→	→	→	→	→	→	→	↑	←	←	←	→	←	←	→	→	→	→
HexO	→	→	→	→	→	→	→	→	→	→	→	→	←	←	←	←	→	←	←	→	→	→
HexO	←	←	←	←	→	→	→	→	→	→	↑	←	←	←	←	←	←	←	←	←	←	←
HexO	←	←	←	←	←	→	→	→	→	→	↑	↑	↑	←	←	←	←	←	←	→	←	←
I	→	←	←	←	←	←	←	→	→	→	→	←	←	←	←	←	←	←	→	→	→	→
I	←	←	←	←	←	↑	←	←	→	→	→	↑	→	↑	↑	→	↑	→	→	→	→	↑
I	↑	→	→	→	↑	←	←	←	→	→	↑	→	↑	↑	↑	↑	←	←	→	←	→	↑
MovingI	→	→	→	→	→	→	→	←	←	←	←	→	→	→	→	→	→	←	←	←	←	→
MovingI	←	←	←	←	→	↑	→	→	→	←	→	→	→	→	→	→	↑	→	→	↑	←	←
MovingI	→	→	→	↑	←	←	→	↑	↑	↑	←	←	→	→	↑	↑	↑	↑	↑	↑	←	→
O	→	→	→	←	←	←	←	→	→	→	←	←	←	→	→	→	→	←	←	←	→	→
O	→	→	←	←	←	←	←	←	←	←	←	←	←	←	→	→	→	←	←	←	→	→
O	→	→	↑	←	←	→	←	←	→	↑	←	←	←	→	→	→	↑	←	←	←	←	→
Triangles	←	→	→	→	→	←	←	←	→	→	→	←	←	←	→	→	→	→	→	←	←	←
Triangles	←	→	→	↑	↑	←	←	←	→	→	↑	←	↑	←	←	←	→	↑	←	←	←	←
Triangles	→	→	→	→	↑	↑	←	←	←	→	↑	↑	↑	→	→	→	→	↑	↑	↑	←	→
Walls	←	←	←	←	→	←	→	→	→	→	→	←	←	←	←	←	←	→	→	→	→	→
Walls	↑	↑	←	→	→	→	→	→	→	→	→	↑	↑	←	←	←	←	←	→	→	→	→
Walls	↑	↑	←	←	←	←	←	←	→	→	↑	↑	←	←	←	←	←	←	←	←	→	↑
X	→	←	←	→	←	→	←	←	→	→	→	←	←	←	←	←	←	←	→	→	→	→
X	↑	→	↑	→	→	→	←	←	→	→	↑	↑	←	←	←	←	←	←	←	→	→	←
X	→	↑	←	←	→	→	↑	→	→	→	↑	↑	←	→	→	→	→	←	→	→	→	↑

Figure 7.6: All traps with ExpectedSARSA, MonteCarlo and QLearning agents

with eps=0.2 and initOptVal=100. The agent stopped an experiment early, after 437 games with initial number of games (n) being 2500. Considering it won 30 games in a row, each one lasting 15 levels, we can assume that the learned policy is optimal for this environment. The red rows represent the MonteCarlo agent, which performed reasonably well with eps=0.2 and initOptVal=20, winning 282/2500 games, but the experiment did not terminate early, suggesting a suboptimal policy. The pink rows represent the QLearning agent, which won 142/2500 games with eps=0.4 and initOptVal=20. It should be noted that the combination of a seed value and hyperparameters that yielded the best results were picked for the MonteCarlo and QLearning agents, and in other cases, they won fewer or no games under this environment. Lastly it should be noted that, SARSA and DoubleQLearning performed poorly, with average scores in all experiments under all hyperparameter combinations, being not more than 20.

Multiple instances in the data show the phenomenon discussed in Section 7.7 of this chapter. For instance, upon closer examination of rotation values 54 and 71 in the rows pertaining to the Balls trap, it becomes evident that the ExpectedSARSA agent opted to alternate between those two rotation types, whereas the other two agents, Monte Carlo and QLearning, chose to proceed using only one or both of the rotations. This trend can be observed in several other cases within the data, and it is highly probable that, with so many rotation options available, any of the three methods would lead to the agent safely passing the trap. Nevertheless, it is a fact that ExpectedSARSA learned a better policy than MonteCarlo and QLearning. However, for certain traps, all or at least two of the agents had satisfactory policies (such as the Hex trap), whereas for others, MonteCarlo and/or QLearning were observed taking actions that could not be deemed optimal when compared to the ExpectedSARSA agent. An example of such an instance can be found in rotation values 275 and 292 with trap `X`, where the `QLearning` agent attempted to switch between the two rotations to pass, while both `ExpectedSARSA` and `MonteCarlo` avoided it, suggesting that remaining in that rotation was not safe and that the agent should try to move to another rotation in a timely manner.

7.3 Tokens environment

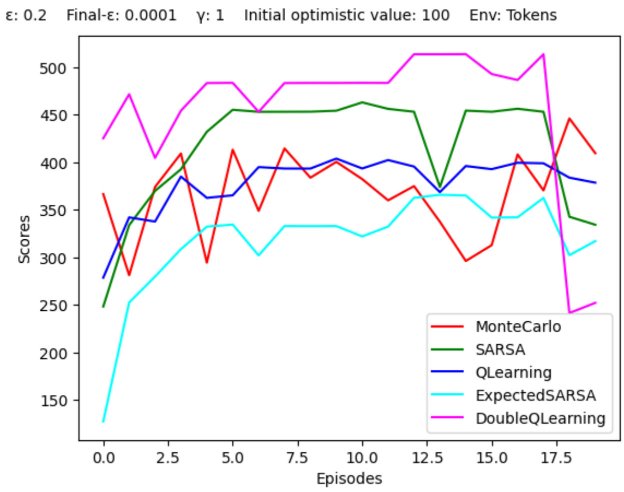


Figure 7.7: `Tokens` experiment

The `Tokens` environment is characterized by its simplicity, as it lacks obstacles that pose a lethal threat to the agent. The agent's task is to collect tokens at regular intervals to prevent its battery from draining completely. The number of rotations needed in this environment is six, making it relatively easy to train. Figure 7.7 illustrates that the average performance of all agents is commendable, even when `n=20`[3]. In subsequent sections, we will delve into more intriguing findings when tokens are incorporated into a larger environment, and explore their impact on the behaviour of the agents.

7.4 Individual Bugs and Viruses environments

This section depicts individual performance of bug and virus type obstacles. They include three (`Worm, LadybugWalking, LadybugFlying`) and two (`Bacteriophage, Rotavirus`) obstacles, respectively, and are different from environments that consist solely of traps. One key difference is that the agent can also use its ability to shoot in `Bugs` and `Viruses` environments. This means that, for the first time in this chapter, our agents have 6 actions to choose from, and we aim to analyse how this affects their behaviour.

We begin by showcasing the performance of all agents in environments containing only a single type of bug or virus. For this purpose, we have evaluated their performance on 50 games, each with `eps=0.2` and `initialOptVal=100.0`.

[3]Note that no smoothing was applied to this plot.

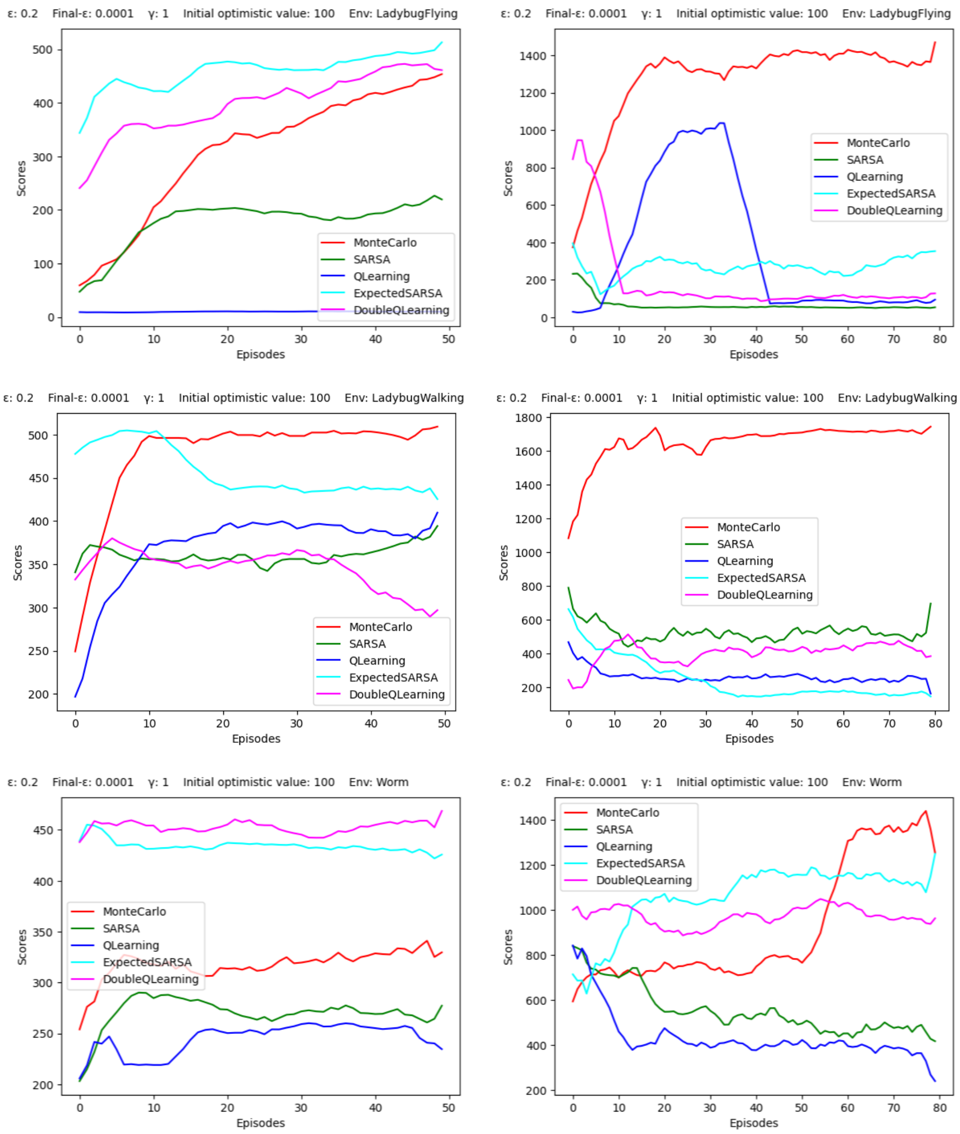


Figure 7.8: `LadybugFlying`, `LadybugWalking` and `Worm` environments experiments

We chose these parameters as they resulted in the best overall performance of the agents. The plots presenting all agents in this section are an average of five different seeds, and each plot has been smoothed with a `window=10`. The left-hand side of both figures displays the agents' performance when shooting was not enabled, whereas the right-hand side depicts the scores when the agents could shoot.

The category of obstacles referred to as bugs in the game environment behaves differently from viruses or traps. This is due to the fact that Hans only loses energy upon contact with any of the bug type obstacles, and only when the battery life is depleted to 0% does the game terminate, or when the game is won. The performance of some agents in this environment is lower than in others, as illustrated in Figure 7.8. However, the `MonteCarlo` agent consistently performs well, particularly with the `LadybugWalking` obstacle. Conversely, the `QLearning` agent appears to perform poorly in this environment, suggesting that it may not be the most suitable method for these seemingly inconsistent bug obstacles. As

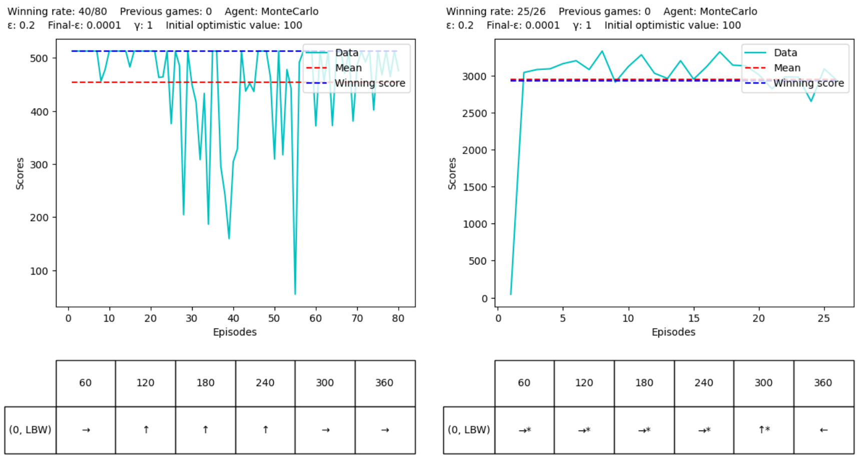


Figure 7.9: `LadybugWalking` with `MonteCarlo` agent experiments

battery life is not part of the state value, the ideal behaviour for the agent would be to either always shoot (if permitted) or always avoid the these obstacles. This behaviour is precisely what is observed in Figure 7.9, where the `MonteCarlo` agent does not shoot at all in the left plot (suggesting that its avoiding the obstacles), while in the right one, it shoots in almost all actions. Although the left plot may not have achieved an optimal policy, it has derived one that allows the agent to win at least 50% of the time.

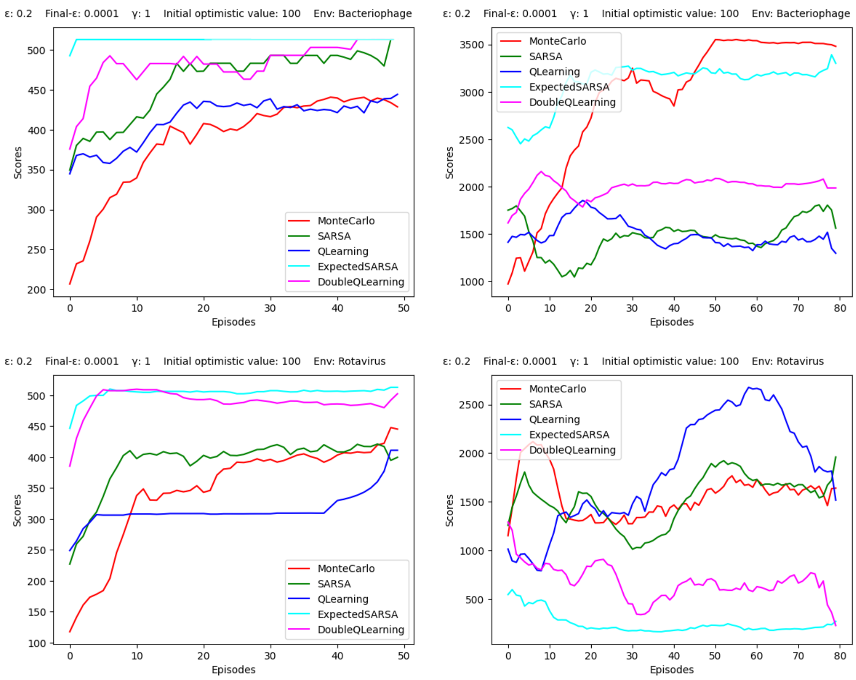


Figure 7.10: `Bacteriophage` and `Rotavirus` environments experiments

Figure 7.10 displays the performance of the agents for each virus type obstacle. Notably, `ExpectedSARSA` exhibited a high level of performance in both cases where shooting was not enabled. However, in the case of `Rotavirus` when the shooting was involved, it performed the worst out of all the agents. It is possible that the agent was confused by the `Rotavirus` behaviour, since when Hans hits a `Rotavirus` for the first time, he becomes sick, and if it is hit again during his sick period, it results in his death. In contrast, the behaviour of `Bacteriophage` is more akin to the traps, as it kills Hans on the spot upon contact.

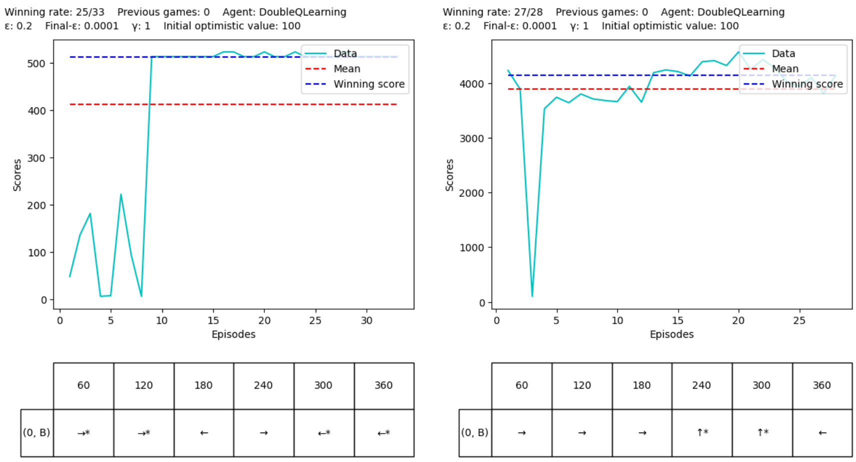


Figure 7.11: `Rotavirus` environment with `DoubleQLearning` agent experiments

To diverge a little from the performance of the agents all together, let's take a look at Figure 7.11 which shows two instances of the `DoubleQLearning` agent with the `Rotavirus` obstacle and `shooting=enabled`. Each plot used only one seed value and the smoothing was not applied. In both of the cases in this figure, the agent learned an optimal policy and thus terminated the game early. However, for the left plot, even though there are some actions in which the agent chooses to shoot, it doesn't actually shoot down any obstacles. This is evident by the fact that the winning score is only around 500, which is the minimum winning score achieved after 15 levels. This might explain why in these conditions, the `DoubleQLearning` agent did not perform as well as some of the others in the overall performance analysis (Figure 7.10). On the other hand, the right plot shows a policy in which that the agent learned to shoot down the `Rotavirus` obstacles and achieved a much higher winning score, demonstrating the importance of properly utilizing the shooting action when it is enabled.

7.5 Bugs and Viruses environments

The current section compares the performance of agents in the full `Bugs` or `Viruses` environments and their combination with `Tokens` when run with the same hyperparameters. Plots that display the performance of all agents in this section are averaged over 10 seeds and smoothed with `window=100`. In Figure 7.12 we see the average performance of the agents when shooting is not allowed. Similar to the traps environment, the `ExpectedSARSA` agent exhibited the best performance. On the other hand, when faced with the full `Bugs` environment, `QLearning` agent showed similar behaviour to that seen on individual bugs obstacles and performed worse than the other agents.

Figure 7.13 shows plots where shooting actions are available to the agent. Here, the `MonteCarlo` agent had the best performance in all four cases. An interesting result is that when faced with `env=[Bugs,Tokens]`, this agent performed

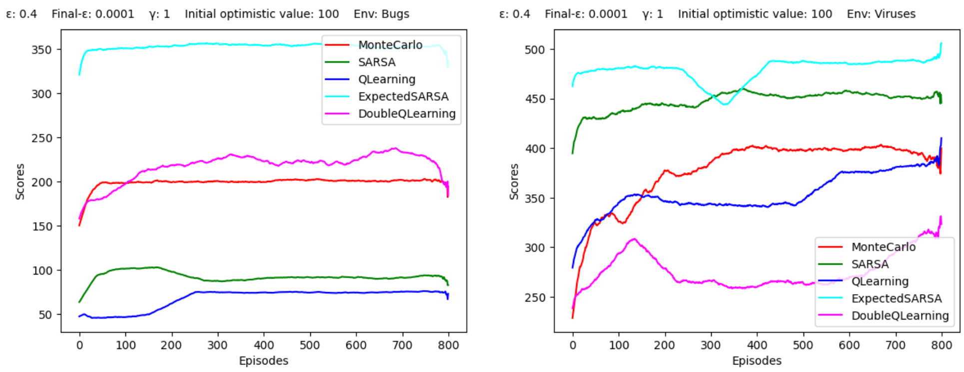


Figure 7.12: `Bugs` and `Viruses` environments experiments

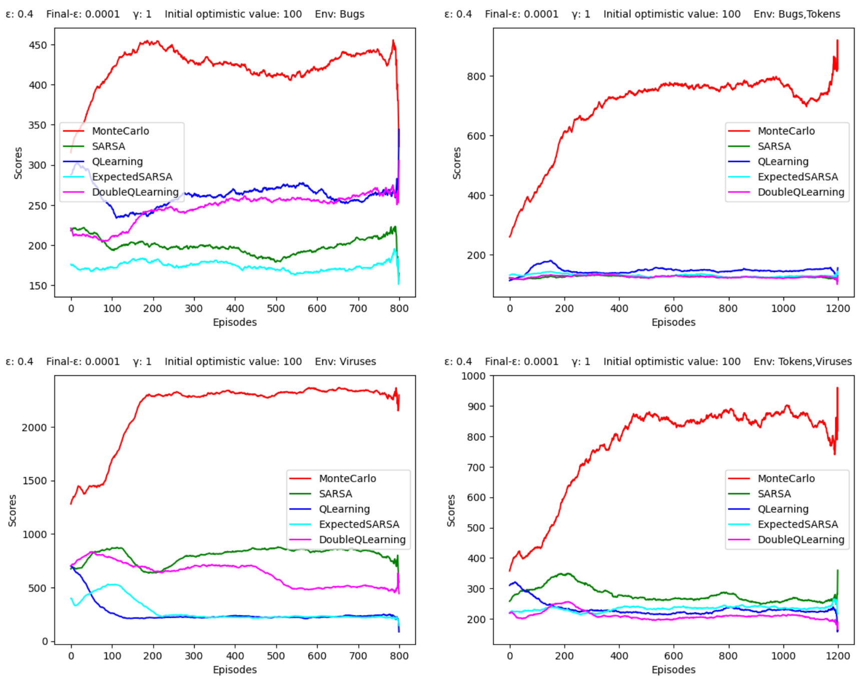
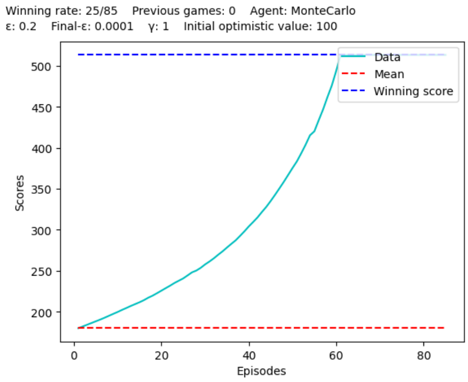


Figure 7.13: `Bugs`, `Viruses` and their combination with `Tokens` experiments

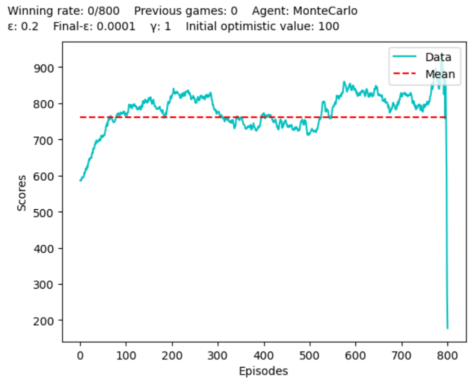
better than in the only `Bugs` environment, while with `env=[Viruses,Tokens]`, it performed worse than with `Viruses` alone. Considering how running into a bug influences Hans, adding `Tokens` to the `Bugs` environment was beneficial as long as the tokens were picked up when possible and the agent was conservative with shooting. In contrast, in the environment containing both `Viruses` and `Tokens`, the agent could still easily lose if Hans ran into `Bacteriophage` once or `Rotavirus` twice in a row. When having unlimited shooting in the `Viruses` environment alone, all agents performed much better than when `Tokens` were part of it as well.

Overall, in environments that include `Tokens`, the ratio of shooting and no shooting actions was approximately even.

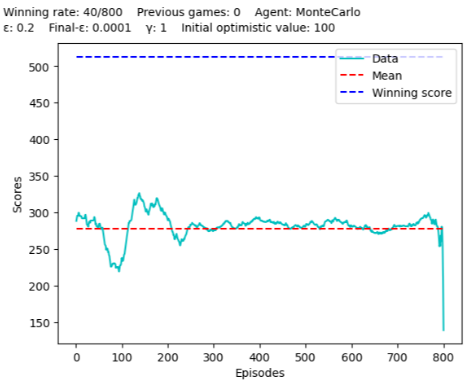
In Figure 7.14, the performance of `MonteCarlo` agent shown environments is displayed for specific hyperparameters and with the same seed value (smoothing window value is 100). For each environment, two plots were created, one where



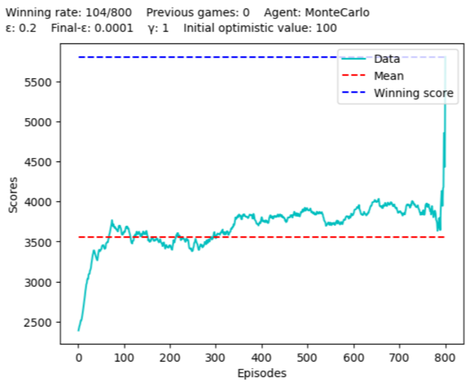
	60	120	180	240	300	360
(0, LBF)	→	→	→	←	←	←
(0, LBW)	←	→	←	→	←	←
(0, W)	→	↑	←	→	←	→



	60	120	180	240	300	360
(0, LBF)	→*	→	↑*	↑*	↑	↑*
(0, LBW)	←	→	→	↑*	←	→*
(0, W)	→*	←	↑	←	→	→



	60	120	180	240	300	360
(0, B)	→	→	↑	↑	←	←
(0, R)	→	→	↑	→	→	→



	60	120	180	240	300	360
(0, B)	→	→	↑*	↑	↩*	↩*
(0, R)	↑	→	→	↑	↩*	↩*

Figure 7.14: `Bugs` and `Viruses` with and without shooting comparison

the agent is allowed to shoot and another where shooting is disabled. The two plots depicting `Bugs` exhibit noticeable differences, as the agent found an optimal policy when shooting was not available. With shooting enabled, even though the direction of the agent's movement for each state is quite similar to the ones in the first plot, it failed to win any games. In contrast, with the `Viruses` environment, the agent's movement is almost identical in both cases, and the plot shows that the agent performed similarly well, with the right plot showing an average and winning scores approximately 10 times higher than on the left, as a consequence of the ability to shoot and due to that gain higher scores.

7.6 Full game environment

In this section, we explore the performance of the agents in the most challenging environment, the full game. The results of these experiments are not surprising, given the complexity of the environment. In the figures presented in this section, we averaged the results of 10 seeds, and each plot was smoothed with a window of

100. The plots on the left side show the performance of the agents in the environment where shooting is disabled, while the right side ones show the environment where shooting is enabled. The right side was trained on 2000 more games than the left one considering that the number of actions possible increased.

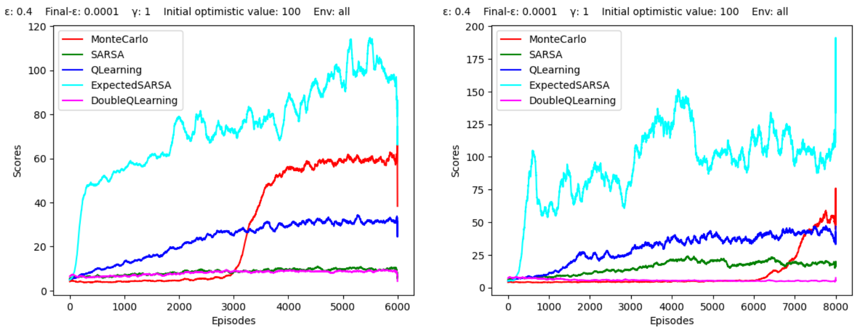


Figure 7.15: Full game experiment

Looking at Figure 7.15, we can see that in the environment without shooting, `ExpectedSARSA` performed the best, as expected. The scores of `MonteCarlo` and `QLearning` agents were not too bad, considering the complexity of the environment and the fact that `QLearning` agent underperformed in the `Bugs` environment. On the right plot, `ExpectedSARSA` is still in the lead compared to the other agents. However, considering that the agents were allowed to shoot in this case, the scores did not improve significantly.

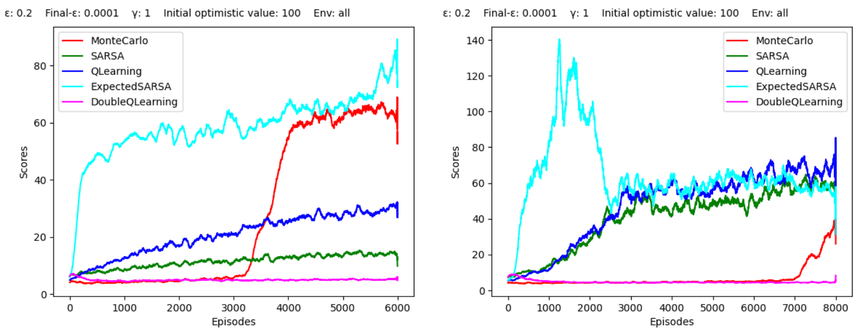
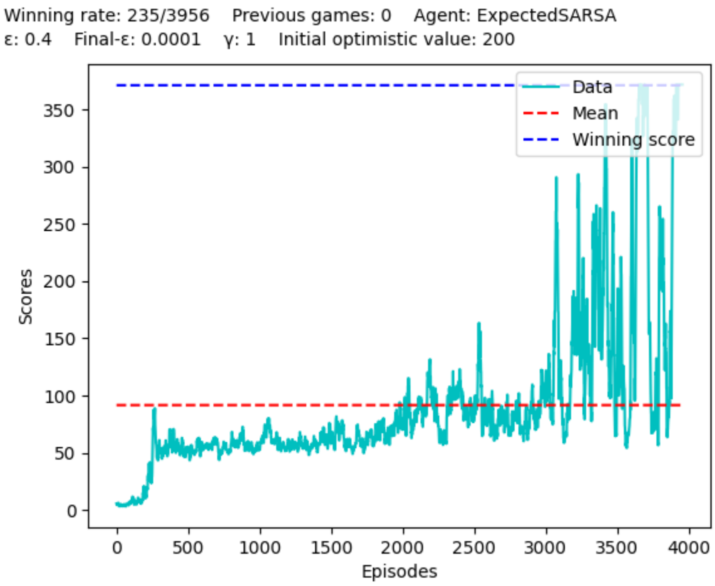


Figure 7.16: Catastrophic forgetting experiment

In Figure 7.16 we can observe an occurrence that has not been discussed before but is present throughout our experiments, namely the concept of *catastrophic forgetting*, Stuart and Peter [2010]. As the name suggests, this is the notion that the agent learns a good policy and due to further exploration of the environment, the policy changes to something suboptimal. Ideally, the agent would come back to the previous policy, but more often than not, this is not the case. In the plot on the right, we can see that this is exactly what happened to the `ExpectedSARSA` agent during this experiment. The policy it learned in the first couple of thousand games was far from optimal, but it achieved a better score than the policies used after the sudden drop at approximately 2500 games.

The highly anticipated encounter detailed in Figure 7.17 showcases a noteworthy outcome, as it portrays the `ExpectedSARSA` agent learning to play a full



	3	20	37	54	71	88	105	122	139	156	173	190	207	224	241	258	275	292	309	326	343	360
(0, B)	→	←	→	→	→	→	←	←	←	↑	→	→	→	→	→	→	→	→	→	←	←	←
(0, Balls)	→	→	→	←	←	→	←	←	←	←	←	←	←	←	←	←	→	→	→	→	→	→
(0, ET)	→	←	←	←	←	←	←	←	←	←	→	→	→	→	→	→	→	→	→	→	→	→
(0, HalfHex)	→	→	→	←	←	←	←	←	←	←	←	←	←	←	←	←	←	→	→	→	→	→
(0, Hex)	↑	←	←	←	←	←	→	→	→	→	→	↑	←	←	←	←	←	→	→	→	→	→
(0, HexO)	←	←	→	→	→	→	→	→	→	→	→	→	←	←	←	←	←	←	←	←	←	←
(0, I)	→	→	→	→	←	←	←	←	→	→	→	→	→	←	←	←	←	←	→	→	→	→
(0, LBF)	→	→	→	→	→	→	→	→	←	←	←	←	←	→	→	←	←	←	→	→	→	→
(0, LBW)	→	→	→	→	←	↑	→	→	←	→	→	←	→	→	→	→	→	→	←	←	←	→
(0, MovingI)	←	→	→	→	→	→	→	→	←	←	←	←	→	→	→	→	→	→	←	←	←	←
(0, O)	→	→	↑	←	←	←	→	→	→	↑	←	←	←	→	→	→	←	←	←	←	←	→
(0, R)	→	→	→	→	→	→	→	→	←	←	←	→	→	→	←	→	←	←	←	←	←	→
(0, Triangles)	→	→	→	→	←	←	→	→	→	→	→	←	←	←	←	→	→	↑	→	←	←	←
(0, W)	→	→	→	→	←	←	←	←	→	←	↑	→	→	→	→	←	→	→	→	←	→	→
(0, Walls)	←	←	←	←	←	←	→	→	→	→	→	→	←	←	←	←	←	→	→	→	→	←
(0, X)	→	←	←	←	←	←	←	→	→	→	→	→	←	←	←	←	←	→	→	→	→	→

Figure 7.17: Full game with `ExpectedSARSA` experiment

game by discovering an optimal policy under specific parameter settings and only one seed value. It is important to highlight that this experiment was conducted while the winning level was 10 and not 15, thus the winning score is lower than in the other examples shown in this chapter. As part of the figure we can see the policy the agent acquired. This example serves as an illustration of how reinforcement learning agents are capable of learning to play this game when given the right conditions. However, it is important to note that this was a singular occurrence, and no similar outcomes were observed when shooting actions were introduced to the experiments. It is reasonable to conclude that while it may not be impossible for the agent to learn such a policy again, it may necessitate obtaining a fortuitous seed value.

7.7 Interesting behaviours

In this chapter, we aim to discuss certain unexpected findings that surfaced during our experimentation. One of the immediate observations can be seen in Figure 7.18[4]. We conducted experiments on two distinct environments, `env=[I]` and

[4]No smoothing was applied to any of the plots in this subsection.

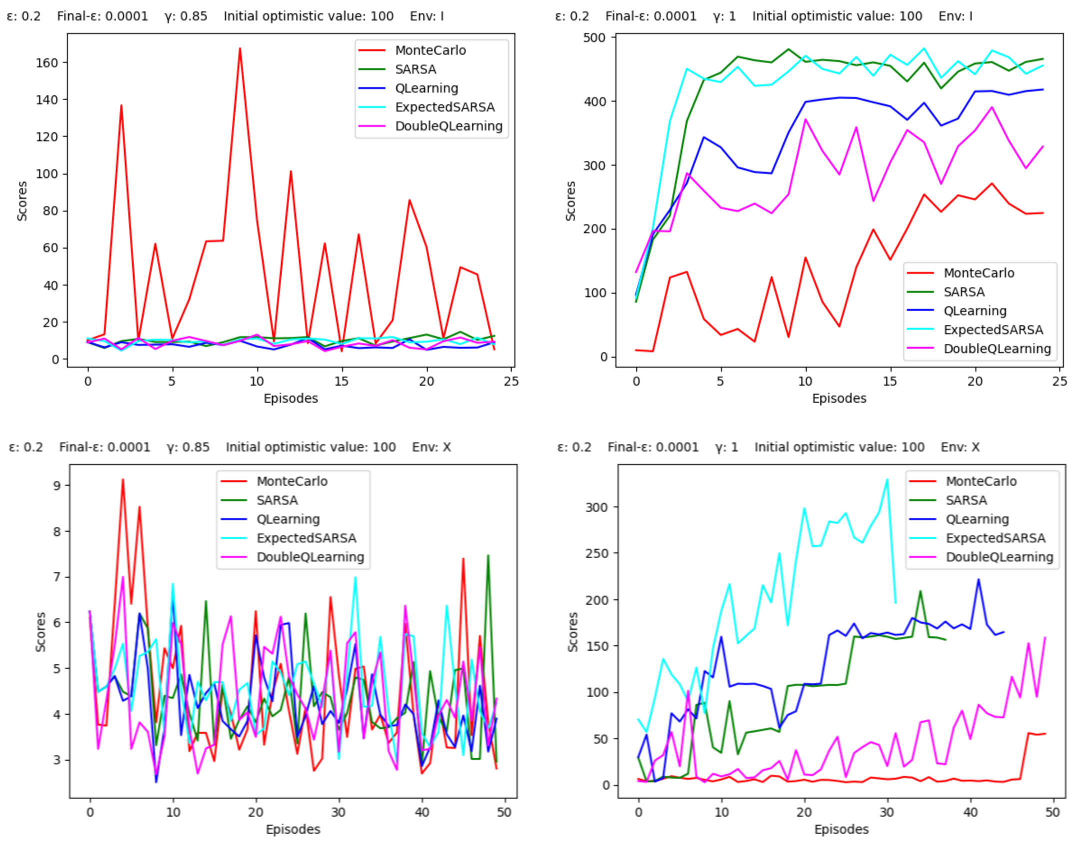


Figure 7.18: Discounting example

`env=[X]`, and for each environment, we carried out experiments with discount rates of `gam=0.85` and `gam=1.0` for all agents. As evident from the plots, the lower gamma value exhibited considerably poorer performance than when no discounting (`gam=1.0`) was applied. This trend is not limited to these specific environments and testing conditions but rather observed consistently across all our experimentation. This result is counterintuitive since it seems logical that penalizing the last action more than previous ones would result in a better policy.

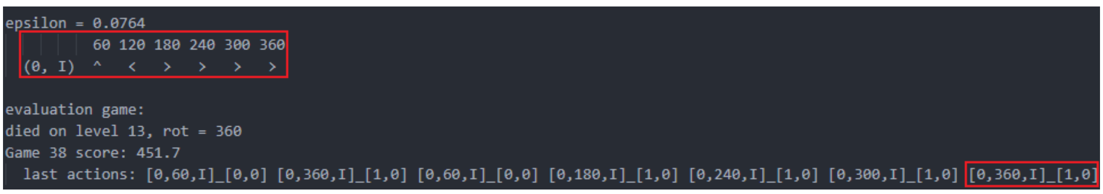
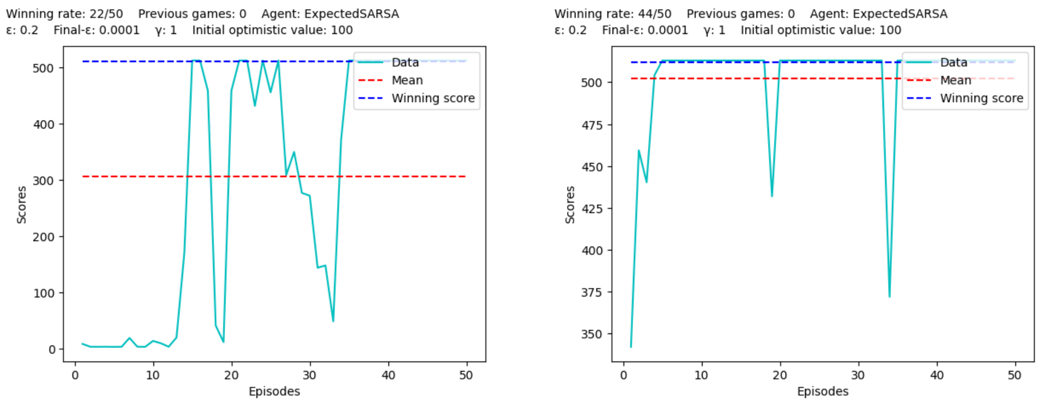


Figure 7.19: Discounting explanation

Upon further investigation, we discovered that in some cases, a lost game for the agent does not result from the last action directly but rather from a chain reaction initiated by a previous bad decision. As seen in Figure 7.19, the agent's last action of going right at rotation `360` to reach a safe one, `60`, is not a poor decision in itself but rather the best possible action in that state[5]. However, analysing the last four actions taken by the agent, it becomes clear

[5]As confirmed by a human player, rotation 60 is safe for trap type I.

that it attempted to reach rotation `60` by going right from the rotation `180`. With a high score of `451.7` (Figure 7.18), the agent undeniably had high speed value, making it move forward very quickly, and while this policy may have been effective in the early game before the agent attained its current speed[6], there is simply not enough time for the agent to rotate at this point in the game. In this case, one could argue that the fourth action from the last was responsible for the agent's loss. For cases like this, we believe that the agent performs better when all actions are penalized equally, i.e. when `gam=1.0`.



	60	120	180	240	300	360
(0, Triangles)	→	←	→	←	→	←

	48	100	152	204	256	308	360
(0, Triangles)	→	→	→	↑	←	←	←

Figure 7.20: `Triangles` example

Another intriguing behaviour emerged as a result of our learning architecture. As previously noted, we chose to have the agent not take a new action every time its `move()` function is called, but rather return the same action until the state changes. This resulted in a substantial reduction in the number of different actions taken by the agent, given that the move function is invoked every game tick, while state changes occur less frequently. This approach led to a behaviour that could not have been predicted at the outset. Figure 7.20 provides an illustration of this behaviour (with some sentiment `env=[Triangles]` was chosen as this was the first environment on which the behaviour was noticed).

Ordinarily, `env=[Triangles]` does not offer any safe rotations unless `rots=7` or more. As shown in the plot, the agent will certainly learn with this rotation value. However, if the agent is given only `6` rotation values to choose from, it will develop a policy that rapidly oscillates between two rotations, keeping the player character, Hans, on the edge of those rotations, allowing him to safely pass through the trap. This behaviour is not confined to situations where the agent is "forced" to make such a decision. During training, in many instances, the agent will learn to stay on the edge rather than advance into a completely safe rotation. There does not appear to be a preference for one or the other; rather, the policy the agent discovers first is determined by other experimental factors. It can be concluded that this behaviour arose solely because the agent did not alter its

[6]It should recalled that after every 3 levels the agent's speed increases.

action until the state changed. In my opinion, this discovery is one of the most exciting outcomes of this project.

Conclusion

In conclusion, this thesis has investigated the efficacy of different reinforcement learning algorithms in various environments in a game implemented in the Godot game engine. The results showed that the `ExpectedSARSA` algorithm performed moderately or exedingly well in all environments, while the performance of other algorithms varied. In particular, `MonteCarlo` demonstrated impressive results in environments featuring bug and virus obstacles. All algorithms displayed adequate performance in environments with individual trap types, while performance in environments with multiple obstacle types was not as consistent. The results underscore the importance of the random actions that the agent receives during training and the balance between exploration and exploitation in reinforcement learning. Future research could explore ways to influence these random actions to potentially achieve better outcomes.

Bibliography

Tunnel Rush, 2023. URL https://tunnelrush2.com/.

Una Adilović. Space Run AI, 2023. URL https://github.com/AdilovicUna/Space-run-AI.

Khronos. glTF - Runtime 3D Asset Delivery, 2023. URL https://www.khronos.org/gltf/.

Juan Linietsky. Godot Engine - Documentation, 2021. URL https://docs.godotengine.org/en/stable/index.html.

Ton Roosendaal. Blender, 2023. URL https://www.blender.org/.

Russell Stuart and Norvig Peter. *Artificial Intelligence: A Modern Approach*. Pearson Education, 4 edition, 2010.

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

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment