

University of Rijeka

*Faculty of Informatics and Digital Technologies*

Marino Linić

# **Application of a Reinforcement Learning Algorithm in the Development of Computer Games**

Bachelor's Thesis

Mentor: Associate Professor Marina Ivašić-Kos, Ph.D.

September 2023 in Rijeka

# Assignment



Rijeka, 15.2.2023.

## Assignment for bachelor's thesis

Applicant: **Marino Linić**

Name of the final thesis: **Application of a Reinforcement Learning Algorithm in the Development of Computer Games**

Name of the final thesis in Croatian language: **Primjena algoritma učenja s podrškom u razvoju računalnih igara**

### Description of the final assignment:

The principles of reinforcement learning and corresponding learning algorithms such as Q-learning should be studied along with the possibility of applying Q-learning in computer games.

Design, choose a programming language for implementation and develop a simple computer game in which Q-learning will be applied. Describe the basic elements of game development and Q-algorithm implementation.

To investigate the influence of different Q-algorithm parameters on the process of learning and finding optimal actions that will lead to the goal of the game with the lowest cost. Show and comment on the effects of changing the parameters of the Q-algorithm on the achievement of goals in the developed game and adjust them so that they provide the most favorable solution.

Mentor:

izv. prof. dr. sc. Marina Ivašić-Kos

Assignments manager:

Doc. dr. sc. Miran Pobar

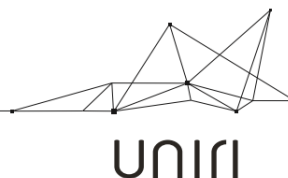
Assignment taken: 15.2.2023.

(Signature of the applicant)

Adresa: Radmile Matejčić 2  
51000 Rijeka, Hrvatska

Tel: **+385(0)51 584 700**  
E-mail: **ured@inf.uniri.hr**

OIB: **64218323816**  
IBAN: **HR1524020061400006966**



# Abstract

In this paper, reinforcement learning was examined by creating a Python puzzle video game and implementing a Q-learning algorithm. Having full control of the development process, it was possible to track, explain, and visualize every step of the algorithm. Along with the theoretical background, this thesis serves as an introductory paper to understand the inner workings of simple machine—or reinforcement—learning algorithms while applying them in video game environments.

Reinforcement learning is an industry standard tool in the development of video games, and having a firm grasp of the methods used to implement such tools is highly pertinent within the field. An analysis of tweaking various parameters of the Q-learning formula is performed and yields different degrees of optimization. An attempt is made to find a more optimal parameter combination, which itself is contextually dependent on the environment the agent is training in.

**Keywords:** Machine Learning, Reinforcement Learning, Video Games, Q-learning, Python, Algorithm, Puzzle, Pathfinding

# Contents

Assignment .....	2
Abstract.....	3
Background.....	5
Reinforcement Learning .....	5
Optimal Control .....	6
Markov Decision Process .....	7
Relevant Machine Learning Terminology .....	9
Episodes .....	9
Hyperparameters .....	9
Q-learning .....	10
Establishing the Methodology .....	11
Development of the <i>Thomas Loops Green</i> video game with a Q-learning agent application .....	12
Setting up the Development Environment .....	12
Gameplay .....	12
Storyline.....	12
PyGame Window and the Map.....	13
Drawing Objects .....	15
Space of Actions (Movement) .....	16
Collision Detection .....	16
Q-learning Agent .....	19
Statistics .....	22
Results.....	25
Comparison and Hyperparameter Tweaking .....	27
The First Run (Default).....	27
Testing Reward Functions .....	28
Testing the Epsilon Value.....	29
Testing the Learning and Discount Rate.....	31
Optimization .....	33
Discussion and Analysis .....	34
Limitations .....	34
Improvements and Alternative Solutions.....	34
Conclusion .....	35
Table of Figures .....	36
Bibliography .....	38

# Background

## Reinforcement Learning

In the early stages of machine learning research, many of the paradigms we recognize today were meshed to the detriment of their development. Reinforcement learning has been recognized as a separate approach to supervised and unsupervised learning—the other two paradigms—since the late 1980s. It utilizes trial-and-error processes characterized both by selection and association, which sets it apart from the other methods (Supervised learning, for example, is only associative). [1] The paradigm fundamentally revolves around balancing the antagonistic relationship between *exploration* and *exploitation*.

The main goal of an agent of a reinforcement learning algorithm is to “learn” via trial-and-error an optimal policy according to user-provided reward reinforcement signals. This is the minimum amount of information that the agent needs before learning or acting. How well the agent can “observe” its environment affects its ability to arrive at an optimal solution, but it is not necessary for the agent to have complete access to the environment: we distinguish between agents having *full observability* and *partial observability*. [1] Reinforcement learning agents are autonomous.

While a more systematic description will be provided later, the process of reinforcement learning can be simplified as the following: an agent takes an action in the environment, the environment is observed by the interpreter, and the interpreter feeds back the reward function and a representation of the state into the agent. This loop occurs in discrete time steps. An illustration is provided in Figure 1.

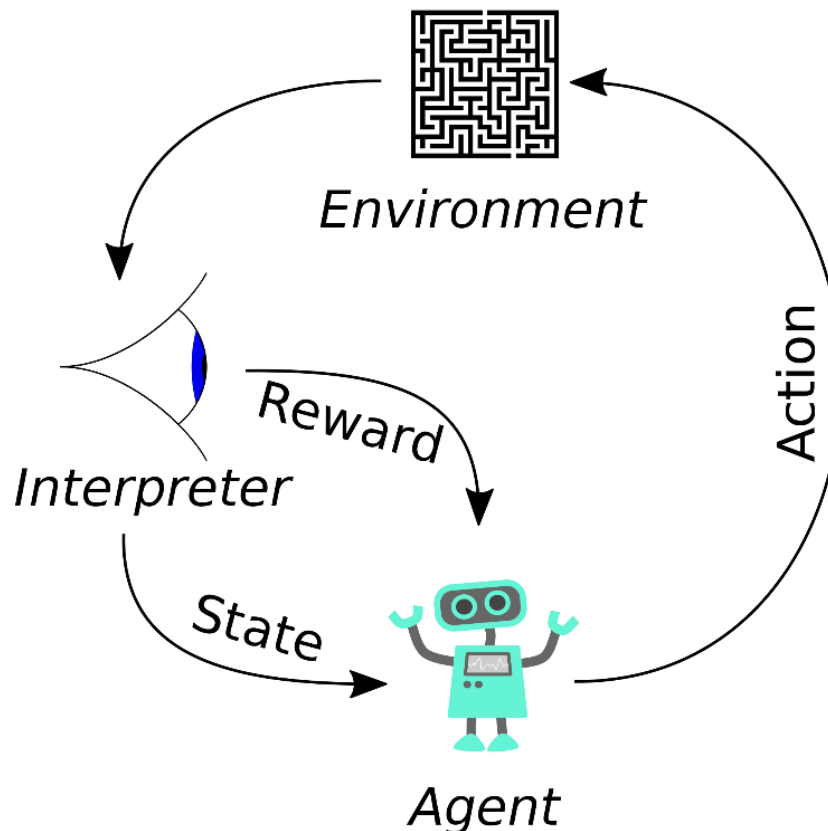


Figure 1: An illustration of reinforcement learning from Wikimedia (CC0). URL: [https://commons.wikimedia.org/wiki/File:Reinforcement\\_learning\\_diagram.svg](https://commons.wikimedia.org/wiki/File:Reinforcement_learning_diagram.svg)

Additionally, reinforcement learning is distinct from other paradigms by not requiring labeled input and output pairs. It tends not to require human intervention beyond defining initial algorithm parameters — which are also specifically called hyperparameters in the context of machine learning. [2] In fact, much of the tuning required in reinforcement learning centers precisely around hyperparameter values, which can significantly improve the speed of learning. Specifically, the exploration aspect of the algorithm — that tends to be very expensive — is minimized, whereas exploitation — “knowledge” that the algorithm possesses — is maximized. Reinforcement learning as such combines *search* and *memory*, the two of which can be clearly mapped to what was previously referred to as selection and association respectively.

That this bears resemblance to many natural and biological processes is not surprising. Reinforcement learning is heavily influenced by psychology and theories on animal learning — where the concept of acquiring behavior by “reinforcement” is common [3]; perhaps the most famous example of this is Pavlovian classical conditioning<sup>1</sup>. Trial-and-error learning is abundant in the natural world as a flexible and general method to build knowledge in an unpredictable environment. It was by studying animal behaviorism that machine learning researchers first began developing reinforcement learning.

Some inspiration came from Edward Thorndike. In 1898, he advanced a principle in behavioral psychology he called the *law of effect*: “responses that produce a satisfying effect in a particular situation become more likely to occur again in that situation, and responses that produce a discomforting effect become less likely to occur again in that situation”. [4]

The techniques used to implement reinforcement learning also strongly resemble *dynamic programming*. In fact, it could be argued that reinforcement learning is a particular type of dynamic programming that deals with problems on scales where it becomes unfeasible to have an exact mathematical model of the environment — which normally takes the form of the *Markov decision process*. In other words, the distinction is ultimately in scale. The reason for this similarity is due to the fact that reinforcement learning also deals with the *optimal control* problem, and dynamic programming is known to be the singular solution to general optimal control problems. [2]

## Optimal Control

Optimal control problems in reinforcement learning are concerned with finding the best policy to maximize cumulative reward in dynamic environments — where it shares characteristics with problems in dynamic programming. [5] The goal is to control dynamic environments so as to maximize a particular function. Ultimately, an optimal policy means that agent’s actions at each state are maximized for expected long-term rewards.

---

<sup>1</sup> Russian physiologist Ivan Pavlov observed that the dogs in his experiments began salivating before being presented with food when subjected to consistent stimuli — such as the same person feeding them. He then associated other stimuli with the dogs’ behavior (like the sound of a metronome) and established the foundation of classical conditioning: the procedure in which organisms automatically develop associations and behavioral patterns with otherwise neutral stimuli. [19]

## Markov Decision Process

The Markov Decision Process (MDP) is a probabilistic (*stochastic*) process: it offers a mathematical approach to modeling decision making in dynamic environments with a significant proportion of random outcomes. [6] It is useful for optimization problems.

MDP functions as *notation* for optimal control problems. Typically, it is a 4-tuple  $(S, A, P_a, R_a)$  where:

$S$  is the **state space** which may be either finite or infinite,

$A$  is the **action space** which may be either finite or infinite,

$P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$  is the probability that action  $a$  in state  $s$  at time  $t$  will lead to state  $s'$  at time  $t + 1$ , in other words, it is **transition probabilities**,

$R_a(s, s')$  is the **immediate reward** following the transition from state  $s$  to state  $s'$  because of action  $a$ .

The policy function, or reward function, is marked with  $\pi$ , and the goal of an MDP is to find a good policy for the agent — the decision maker — where the preferable action  $\pi(s)$  is specified by the function  $\pi$ .  $\pi$  is the probability distribution for every state  $s$ . An example of MDP is illustrated in Figure 2.

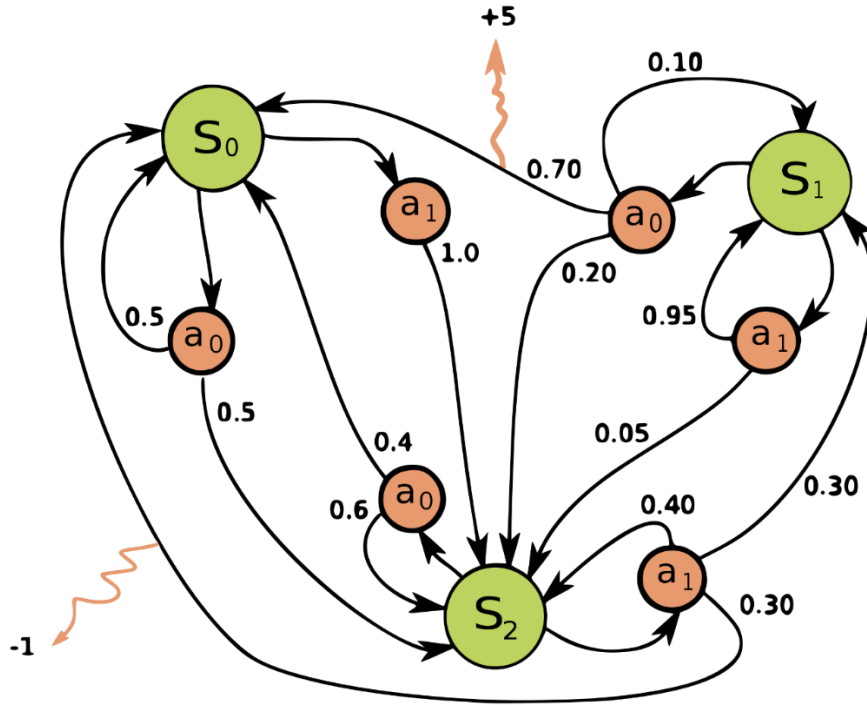


Figure 2: This MDP has three states, two actions for each, and two rewards (the orange arrows).  
URL (Wikimedia, CC0): [https://commons.wikimedia.org/wiki/File:Markov\\_Decision\\_Process.svg](https://commons.wikimedia.org/wiki/File:Markov_Decision_Process.svg)

In the illustration, we may notice that every state has an associated action with a particular outcome — we may call these *state-action pairs*.

## The Bellman Equation

The Bellman equation has come to be the necessary condition for optimality in dynamic programming. It allows for a way to *recursively* express values of state-action pairs. Ultimately, the formula can be summarized as the following [7]:

$$V(s) = \max_a (R(s, a) + \gamma V(s'))$$

where:

$V(s)$  is the **expected return value** at the current state  $s$ ,

$\max_a$  is the **maximum value** of any possible action  $a$ ,

$R(s, a)$  is the **expected reward** for taking **action  $a$  at state  $s$** , and

$\gamma V(s')$  is the **discount factor** multiplied by the value of the **next state**.

In other words, the value is determined by a combination of short term and long term reward (recursive), mediated by the value of the gamma. If gamma is zero, then the value is solely determined by the immediate reward. This simple idea has been immensely influential in the world of reinforcement learning.

To show how this formula works in practice: if we have a game with two choices and one of them includes an unknown probability, such as in betting, then the gamma would be multiplied by the expected value of the next state as influenced by probability, which in turn would be influenced by its own next state, and so on — making the equation recursive up to a certain point when the immediate reward is the better choice.

Figure 3 shows an example of a Bellman flow chart.

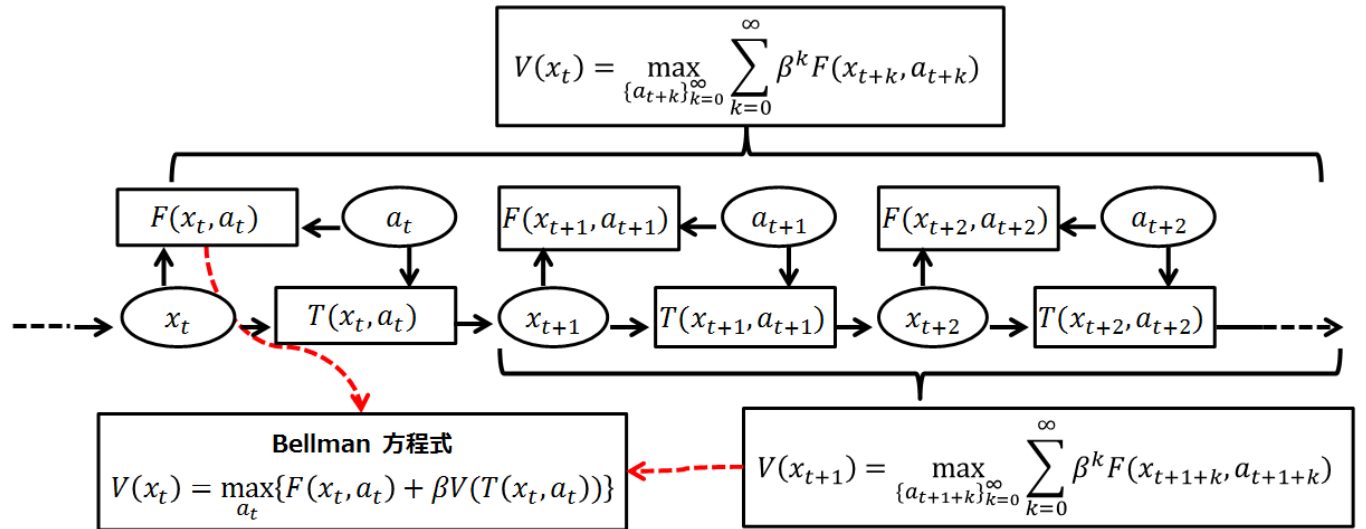


Figure 3: A Bellman flow chart.

URL (Wikimedia, CC0): [https://commons.wikimedia.org/wiki/File:Bellman\\_flow\\_chart.png](https://commons.wikimedia.org/wiki/File:Bellman_flow_chart.png)



## Relevant Machine Learning Terminology

### Episodes

An episode in the context of reinforcement learning is a set of actions and their resulting states that an agent performs until it reaches an end state. Usually, episodes are recorded, and contain information about how the agent dealt with trial-and-error learning. [8]

### Hyperparameters

Hyperparameters are parameters used to fine-tune the training process. Three relevant parameters are listed below.

#### *Learning Rate*

The learning rate determines what the *step size* each iteration will be as the algorithm acquires new information. A higher learning rate means that the algorithm will be quick to learn and adapt, but the downside of that approach is that it can get stuck in suboptimal strategies quickly. [9]

#### *Discount Rate*

The discount rate is used in Q-learning. It determines how future rewards are prioritized. A lower value will result in a more short-term and myopic approach, whereas the inverse is true for a higher value. If the value is equal to one, the agent's time horizon will be infinite. [10]

#### *Epsilon*

In this thesis, epsilon is a number between zero and one that determines how random the agent's actions will be. For instance, an epsilon value of 0.4 will lead to 40% of the agent's actions being random or exploratory.

## Q-learning

Q-learning is a simple reinforcement learning algorithm that does not contain transition probabilities associated with machine learning models. Thus, Q-learning handles problems using state-action pairs only using rewards without adaptations. Q-learning finds optimal policies by maximizing reward and thus expected value via selection. [11]

It was introduced in 1989 [10] and has since become a widely used reinforcement learning algorithm, albeit only in simple use cases.

It creates a table — named Q-table — that it updates based on the Q-learning algorithm formula, which itself is a Bellman equation.

The formula is shown in Figure 4,

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

new value (temporal difference target)

Figure 4: The Q-learning algorithm formula.

URL (Wikipedia): <https://en.wikipedia.org/wiki/Q-learning#Algorithm>

where:

$Q$  is the **value of the state-action pair** which is usually initialized with an arbitrary value like 0,

$Q^{new}$  is the **value of the new state-action pair** that the agent has learned from,

$s_t$  is the **current state**,

$s_{t+1}$  is the **future state**,

$a_t$  is the **current action**,

$\alpha$  is the **learning rate**,

$r_t$  is the **reward for future state**,

$\gamma$  is the **discount factor**,

$\max_a Q(s_{t+1}, a)$  is the **estimate of optimal future value in the state-action pair**.

Usually, episodes end when  $s_{t+1}$  is in the final state, and that will apply to this thesis' program as well.

## Establishing the Methodology

The algorithm chosen to demonstrate reinforcement learning will be Q-learning, due in part to its simplicity — both regarding its implementation and the in-depth understanding of how the algorithm functions. While insufficient for many advanced machine learning tasks, Q-learning is optimal as an introductory method: it is historic, it establishes context for reinforcement learning algorithms, and it shares the *fundamental strategies and thinking behind reinforcement learning*.

Instead of choosing libraries with premade video game environments like OpenAI's *gym*, a custom environment will be made from scratch. The main benefit of this approach is flexibility: How the algorithm functions will be shown and analyzed step-by-step. Furthermore, the game design itself has been adapted to show the flexibility of the approach.

The programming language chosen for the task is *Python*. Python is the most popular language for machine learning [12], however, in this particular instance, it is chosen for:

- a. its video game library *PyGame* — providing a suitable platform for game development [13],
- b. its *NumPy* library — granting powerful support for multidimensional arrays [14],
- c. the *Matplotlib* library — allowing for data visualization [15],
- d. and Python's general ease of use surrounding creating Desktop applications [16].

A simple code editor will be used to build the entire environment. *Visual Studio Code* appears to be the optimal choice, and a VSC extension, *CodeSnap*, will be used to display code snippets in this thesis.

# Development of the *Thomas Loops Green* video game with a Q-learning agent application

The genre and type of video game that will be created is a simple puzzle, path-finding game named *Thomas Loops Green*. The goal of the game is for the player (black square, Thomas) to reach the goal (green square) as fast as possible while avoiding all the enemies (red squares). The game loops the same map of the level every time when either the goal is reached or two hundred moves are made, but spawns Thomas in three different places on the map. A text explaining the rules and the plot is shown to the player ten seconds before the game starts.

## Setting up the Development Environment

Three separate Python files are created (Fig. 5):

1. **main.py**: The file that runs the game and contains most of the game's logic. Functions from other files are imported. Unless explicitly stated otherwise, all figures containing code snippets are assumed to be from main.
2. **stats.py**: The file that visualizes data using the Matplotlib library.
3. **grid\_maker.py**: The file that creates the map.



Figure 5: Included programming libraries in the main.py and stats.py files.

## Gameplay

### Storyline

The game centers around a black square, called Thomas, created as a virtual agent with a single terminal goal: he needs to navigate the map to find the green square while avoiding the red squares, and he wants to reach the green square as quickly as possible (Fig. 7). The latter two are his two instrumental goals.

The game is cut into episodes while running in a loop, but Thomas' memory persists — hence, Thomas eventually gets better at fulfilling his instrumental goals over time.

If Thomas makes two hundred moves without having reached the green square, he is reset at one of the three positions on the map (Fig. 6), and the episode ends without success.

The player is rewarded the sooner they reach the green square and the more they avoid red squares.



Figure 6: Thomas is not placed at different places on the map randomly.

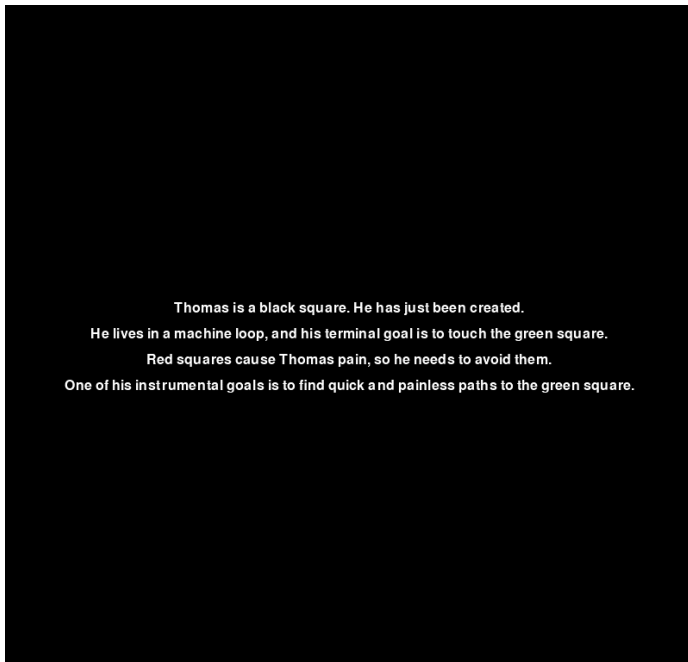


Figure 7: The player is given instructions at the beginning of the game. The right side of the image shows implementation.

## PyGame Window and the Map

PyGame is initialized at the beginning of the main function (Fig. 8). The dimensions of the window are set by the size of the map, which is de facto divided into a 20 x 20 matrix or a *table* where each of the 400 elements or *cells* represents a field of 40 pixels. The window size is thus determined to be 800 x 800.

The colors used in the rest of the code are defined afterwards, and all objects and actors — Thomas (*player*), the red squares (*enemy*) and the green square (*target*) respectively — are the size of one cell, or 40 pixels. This enables us to draw them on the screen so that they fit the imaginary matrix.



Figure 8: Initialization of PyGame.

The exact locations of these three relevant game elements are determined as arrays where the first value represents the index of the row, and the second the index of the column. Only the players' location changes over time (Fig. 9).



Figure 9: Location of actors and objects.

The location and number of enemies dictates the primary look of the map. I have decided to make around half of the matrix cells have an enemy. I generated them randomly and used a function (no longer present in the code) that exported the result as a two-dimensional array, so that it could be visually accessible and easily editable. The function `enemy()` in the `grid_maker.py` file returns this array and defines all enemy locations (Fig. 10).

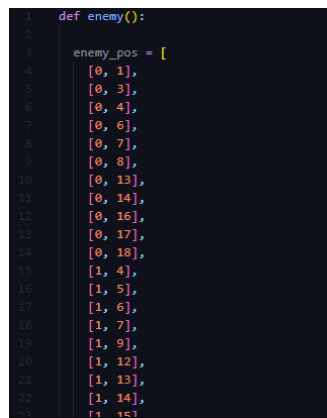


Figure 10: Screenshot of a part of the `grid_maker.py` file.

## Drawing Objects

Once the game loop starts, these characters are drawn on the screen using built-in PyGame functions. Firstly, each object is made into a PyGame rectangle (*pygame.Rect*) to later check for collisions. Afterwards, it is drawn on the screen using *pygame.draw.rect()*. This function takes three primary arguments: the canvas it will be drawn on, the color, and the coordinates, width, and length of the object—in this case a tuple (Fig. 12) [17].

Since there is a dynamic number of red squares, and each is needed to be both drawn on the screen and checked for collisions, a for loop is used to create them as aforementioned entities.

PyGame draws sequentially, hence, the player is drawn last, despite its rectangle being created earlier: firstly, so that the player always stays visible and does not disappear under red squares, and secondly, so that a color can be drawn to indicate a collision (Fig. 13). Fig. 11 shows the look of the game.

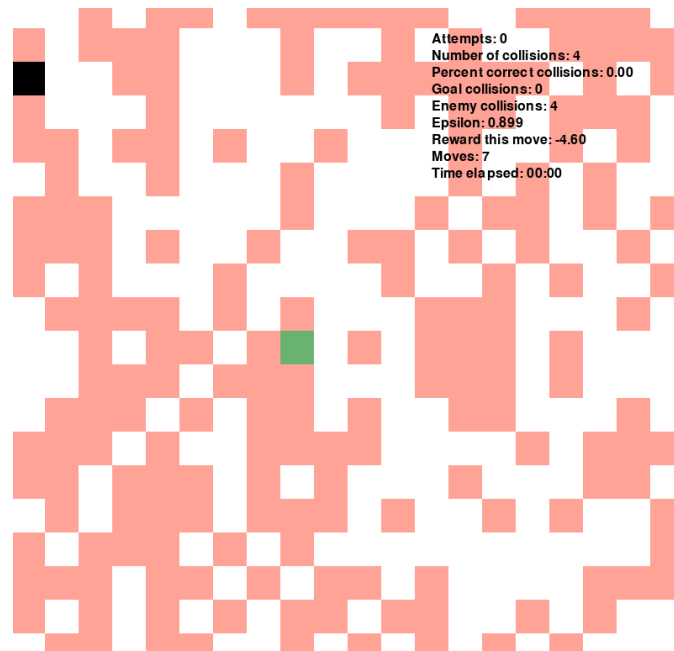


Figure 11: Look of the game.

```
1 player_rect = pygame.Rect(player_pos[0] * cell_size, player_pos[1] * cell_size, player_size, player_size)
2
3 enemy_rects = []
4
5 target_rect = pygame.Rect(target_pos[0] * cell_size, target_pos[1] * cell_size, target_size, target_size)
6 pygame.draw.rect(window, GREEN, (target_pos[0] * cell_size, target_pos[1] * cell_size, target_size, target_size))
7
8 for pos in enemy_pos:
9     enemy_rect = pygame.Rect(pos[0] * cell_size, pos[1] * cell_size, enemy_size, enemy_size)
10    enemy_rects.append(enemy_rect)
11    pygame.draw.rect(window, RED, (pos[0] * cell_size, pos[1] * cell_size, enemy_size, enemy_size))
```

Figure 12: Creation of PyGame Rect objects and drawing them on the screen.

```

1 # Drawing player last
2 if collision_detected == True:
3     pygame.draw.rect(window, COLLISION, (player_pos[0] * cell_size, player_pos[1] * cell_size, player_size, player_size))
4 else:
5     pygame.draw.rect(window, BLACK, (player_pos[0] * cell_size, player_pos[1] * cell_size, player_size, player_size))

```

Figure 13: After performing collision detection, the player's color is determined by the outcome.

## Space of Actions (Movement)

The player can move in four directions: *down*, *up*, *left*, and *right*. There are no restrictions except moving outside the bounds of the window (Fig. 14).

```

1 actions = [0, 1, 2, 3] # Down, Up, Left, Right

```

```

1 # Taking action
2 if action == 0 and player_pos[1] > 0:
3     player_pos[1] -= 1 # Down
4 elif action == 1 and player_pos[1] < grid_height - 1:
5     player_pos[1] += 1 # Up
6 elif action == 2 and player_pos[0] > 0:
7     player_pos[0] -= 1 # Left
8 elif action == 3 and player_pos[0] < grid_width - 1:
9     player_pos[0] += 1 # Right

```

Figure 14: An array of actions is initialized before the game runs. Once an action is chosen, the player moves along the map matrix according to the array index. If an action leads to the player moving out of bounds, no action is taken.

## Collision Detection

Collision detection is primarily performed using PyGame's *Rect* object's method *colliderect()*. [18] The *Rect* object, as we've seen earlier, takes x coordinates, y coordinates, width, and height as arguments. *Colliderect()* returns a boolean after checking whether two *Rect* objects have collided.

Collisions are one of the most pertinent features of the gameplay, so many variables are tied to the outcome of this section of the code. Two Boolean variables are introduced to keep track of these outcomes: *episode\_end* and *collision\_detected*. Both are initialized as *False* (Fig. 15).





Figure 15: Initialization of these variables appears right before collision detection logic.

First, we check whether the player rectangle has collided with the target rectangle (Fig. 16). If so, a host of conditions are met:

1. The agent is rewarded 100 points.
2. A variable that counts collisions is incremented by 1.
3. A variable that counts total collisions with the goal square is incremented by 1.
4. The Boolean variable that detects collision is assigned True.
5. The Boolean variable keeping track of whether the episode will end is also assigned True.
6. A variable that counts collisions with the goal square is incremented by 1.
7. A variable that counts how many collisions happened within an episode is incremented by 1.
8. A variable counting how many goal collisions appeared within the episode is incremented by 1.



Figure 16: Collision check for the goal.

After this check is complete, the same is done for all enemies (Fig. 17). A for loop is used to check whether a collision happened with each individual enemy. If a collision occurs:

1. The agent is awarded -100 points.
2. The variable keeping track of the total number of collisions is incremented by 1.
3. The Boolean variable that keeps track of whether a collision occurred is assigned True.
4. A variable keeping track of total collisions with the enemies is incremented by 1.
5. The variable keeping track of total collisions within an episode is incremented by 1.



Figure 17: Collision check for enemies.

Finally, if `collision_detected` is assigned `False` — if no collision occurred — then the agent is awarded in a more complex manner (Fig. 18).

Every move the agent makes receives a negative reward. However, there is substantial variation in the degree to which this occurs depending on the player's proximity to the goal. A function is used that calculates the distance to the goal by combining the distances via the x and y axes and returns an absolute number. Then, this number is subtracted from 20 — an arbitrary number — and multiplied by 0.1 — also arbitrary — until finally being subtracted by 5 once again. This way, the agent is awarded in a *relative sense* the closer it gets to the goal.



Figure 18: If no collision occurs.

If 200 moves are made within an episode — ie. the goal has not been reached — the episode ends. (Fig. 19)



Figure 19: A prevention measure against getting stuck in a suboptimal state.

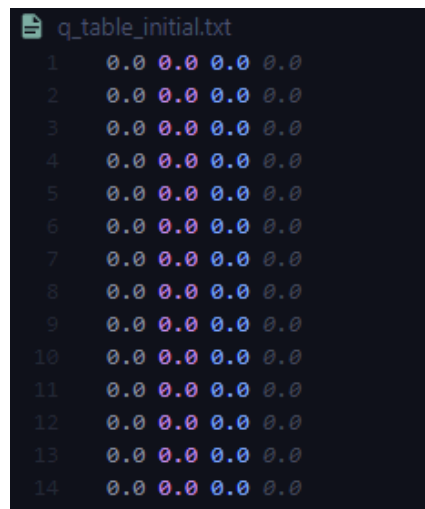
## Q-learning Agent

A Q-table is initialized before the game loop starts (Fig. 20, 21). The NumPy library is used to create the table with zeros. It is made to be the same size as the map matrix, having used *grid\_width* and *grid\_height*, but with an added dimension: actions. This way, each cell in the map has values corresponding to the four available actions, and so the agent can learn to choose the optimal action in each cell.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top left. It contains three lines of Python code:

```
1 q_table = np.zeros((grid_width * grid_height, len(actions)))
2
3 export_q_table(q_table, 'q_table_initial.txt')
```

Figure 20: Initialization of the Q-table.

A text editor window titled 'q\_table\_initial.txt' showing a 14x5 grid of zeros. The first column contains line numbers from 1 to 14. Each of the following four columns contains the value '0.0'.

```
q_table_initial.txt
1 0.0 0.0 0.0 0.0
2 0.0 0.0 0.0 0.0
3 0.0 0.0 0.0 0.0
4 0.0 0.0 0.0 0.0
5 0.0 0.0 0.0 0.0
6 0.0 0.0 0.0 0.0
7 0.0 0.0 0.0 0.0
8 0.0 0.0 0.0 0.0
9 0.0 0.0 0.0 0.0
10 0.0 0.0 0.0 0.0
11 0.0 0.0 0.0 0.0
12 0.0 0.0 0.0 0.0
13 0.0 0.0 0.0 0.0
14 0.0 0.0 0.0 0.0
```

Figure 21: The Q-table is initialized with zeros.

Then, hyperparameters are chosen (Fig. 22, 23). Depending on what values are assigned, the process of learning can be significantly sped up. However, it is not immediately obvious what values are optimal. This part of the code is tested and discussed later in the thesis.

Both the learning and discount rate are initialized with high values: in this case—0.9. Since the map is static, it did not seem necessary to make the agent cautious. The epsilon is initialized with a value of 0.9 also, but a mechanism is implemented to incrementally decrease that value towards 0. The agent can learn fastest by moving almost entirely randomly at the beginning, thus exploring large portions of the map. Then, with each move, it can hover increasingly more around the path towards the goal.



Figure 22: The hyperparameters.



Figure 23: Epsilon decreases with each move as the game loop progresses. When epsilon is 0, the *Q*-table is no longer updated, and it is sent to a file. The speed of the simulation also slows so that the viewer can get a glimpse of how the agent behaves after training.

When it comes to the agent taking action, it can largely be divided into either intentional or random decision-making. The percentage of random actions depends on the value assigned to epsilon. We can conceptually treat epsilon as a percentage of random movements that the agent takes.

The way this is determined is with an *if-else* statement: if a random number in the range between 0 and 1 is smaller than epsilon, then a random action is chosen, and the old state—the *Q*-table row representing the cell that the agent is in before taking action—is preserved as the index number in the matrix in the *old\_state\_index* variable (Fig. 24). If we imagine that epsilon is 0.9 and a random number is chosen in the aforementioned range, then the number will be smaller than epsilon 90% of the time.

If the number is greater than epsilon, the NumPy function *argmax()* is used to return the action in the row with the highest value. That way, optimal actions for the cell are chosen based on the existing *Q*-table. The old state is preserved once again.



Figure 24: Action-taking.

Once the action is chosen by the agent, and the appropriate points are assigned to the reward variable, the Q-table can be updated (Fig. 25). This process can be elucidated by print statements documenting every change (Fig. 26). The feature is an advantage of having built the game from scratch.

After the old state index has been assigned a value, and a move been made, the current state index is also assigned the player's current position. First, the whole row of the cell the agent was in before taking action is stored in the *current\_q* variable.

Subsequently, the best action in the current (new) state is determined by looking at the current state index. The reason for this is to avoid myopic action taking. If every value for every action in every cell is tied to the actions in the next cell, then it follows that the agent can be long-term oriented.

Finally, the formula for Q-learning is implemented and then assigned to the row that the agent was in before having taken the action. The print statements concretize the Q-learning formula in the figure below.

```

1 # Update the Q table
2 state_index = player_pos[0] + player_pos[1] * grid_width
3
4 print("Old state index:", old_state_index)
5 print("New state index:", state_index)
6 print("Actions for old state index:", q_table[old_state_index])
7 print("Actions for new state index:", q_table[state_index])
8
9 current_q = q_table[old_state_index, action]
10 print("Action taken:", current_q)
11
12 max_future_q = np.max(q_table[state_index])
13 print("Best action in new state:", max_future_q)
14
15 new_q = (1 - learning_rate) * current_q + learning_rate * (reward + discount_rate * max_future_q)
16 print("Q formula: (1 - learning_rate) * current_q + learning_rate * (reward + discount_rate * max_future_q)")
17 print("Plugged in: (", "1", "-", learning_rate, ")", "*", current_q, "+", learning_rate, "*", "(", reward, "+", discount_rate, "*", max_future_q, ")")
18 print("New value for action taken:", new_q)
19
20 q_table[old_state_index, action] = new_q
21 print("Actions for old state index:", q_table[old_state_index])
22 print("\n")

```

Figure 25: Updating the Q-table.

```

Old state index: 251
New state index: 231
Actions for old state index: [ 54.3881      36.82149716 -48.2900214   35.90459769]
Actions for new state index: [ -6.30158427  45.39565232  64.209    -59.43382248]
Action taken: 54.38810000000001
Best action in new state: 64.209
Q formula: (1 - learning_rate) * current_q + learning_rate * (reward + discount_rate * max_future_q)
Plugged in: ( 1 - 0.9 ) * 54.38810000000001 + 0.9 * ( -3.4 + 0.9 * 64.209 )
New value for action taken: 54.38810000000001
Actions for old state index: [ 54.3881      36.82149716 -48.2900214   35.90459769]

```

Figure 26: The console prints out every update of the Q-table. This makes debugging simpler and concretizes the formula.

## Statistics

Keeping track of the percentage of correct collisions is done by dividing the number of collisions with the green square with the total number of all collisions, which includes collisions with the red squares. Dividing by zero is not permitted in mathematics, so the calculation only runs if the current loop iteration's collision tracking variable is one or greater.

Various variables are tracked and then displayed on the screen (Fig. 27). The PyGame `draw_text()` function is used to accomplish the drawing.



```
1 # Stats
2 if total_collisions > 0:
3     total_percentage = (goal_collisions / total_collisions) * 100
4
5     draw_text(f"Attempts: {episode}", (500, 45))
6     draw_text(f"Number of collisions: {total_collisions}", (500, 65))
7     draw_text(f"Percent correct collisions: {total_percentage:.2f}", (500, 85))
8     draw_text(f"Goal collisions: {goal_collisions}", (500, 105))
9     draw_text(f"Enemy collisions: {enemy_collisions}", (500, 125))
10    draw_text(f"Epsilon: {epsilon:.3f}", (500, 145))
11    draw_text(f"Reward this move: {reward:.2f}", (500, 165))
12    draw_text(f"Moves: {moves}", (500, 185))
13    draw_text(f"Time elapsed: {minutes:02d}:{seconds:02d}", (500, 205))
```

Figure 27: The player can get access to elapsed time, number of attempts or episodes, rewards, number of total moves, etc.

Once the episode ends, the player is placed at one of three positions on the map (Fig. 28). The first one, on the upper left corner, occurs when an episode is divisible by three. The second position, in the lower right corner, occurs if the episode is divisible by two. Finally, if neither conditions are true, the player is placed on the right end of the map where there is another open path to the goal. Placing the agent on different places of the map allows us to understand the algorithm with greater fidelity.



```
1 if episode_end == True:
2     episode += 1
3
4     if episode % 3 == 0:
5         player_pos = [1, 1]
6     elif episode % 2 == 0:
7         player_pos = [18, 10]
8     else:
9         player_pos = [18, 16]
```

Figure 28: Resetting player's position on the map.

Once again, because numbers can't be divided by zero, `episode_collisions` is given a value of 1 in case the first episode ends without a collision (Fig. 29). We then calculate and restart many of the variables at the end of the episode.



Figure 29: Collision-tracking variables.

The matplotlib graph is remade every tenth episode (Fig. 30). This allows us to visualize the training in real-time and compare parameters.



Figure 30: Sending data every 10<sup>th</sup> episode.

The matplotlib function takes two arguments. The first is an array of the data (*episode\_success*) which represents an array of percent successful collisions per episode. The second argument is an array of average values every tenth episode (*episode\_average*).

A bar chart is used, and a custom color is defined. Afterwards, a trendline is added with *plt.plot*. The image is saved locally. The code is shown in Figure 31.



Figure 31: Bar plot function.

Finally, some statistics are exported to compare various parameters (Fig. 32).



```

1  if episode_percent == 100:
2      with open("hundred.txt", "a") as f:
3          f.write(str(episode) + " " + str(elapsed_time) + "\n")
4
5      if len(episode_success) % 10 == 0:
6          bar_plot_data(episode_success, episode_average)
7
8      if moves >= num_moves:
9          running = False
10         with open("results.txt", "w") as f:
11             f.write("Total correct collisions in percent: " + str(total_percentage) + "\n" +
12                     "Episodes: " + str(episode) + "\n" +
13                     "Moves: " + str(moves) + "\n" +
14                     "Time elapsed in seconds: " + str(elapsed_time) + "\n" +
15                     "Learning rate: " + str(learning_rate) + "\n" +
16                     "Discount rate: " + str(discount_rate) + "\n" +
17                     "Initial epsilon: 0.9" + "\n" +
18                     "Epsilon changed every move by: -0.0001" + "\n" +
19                     "Reward for green square: 100" + "\n" +
20                     "Reward for red square: -100" + "\n" +
21                     "Reward for move: 0.1 * (20 - distance) - 5")
22
23
24     pygame.display.update()

```

Figure 32: Saving parameters.



## Results

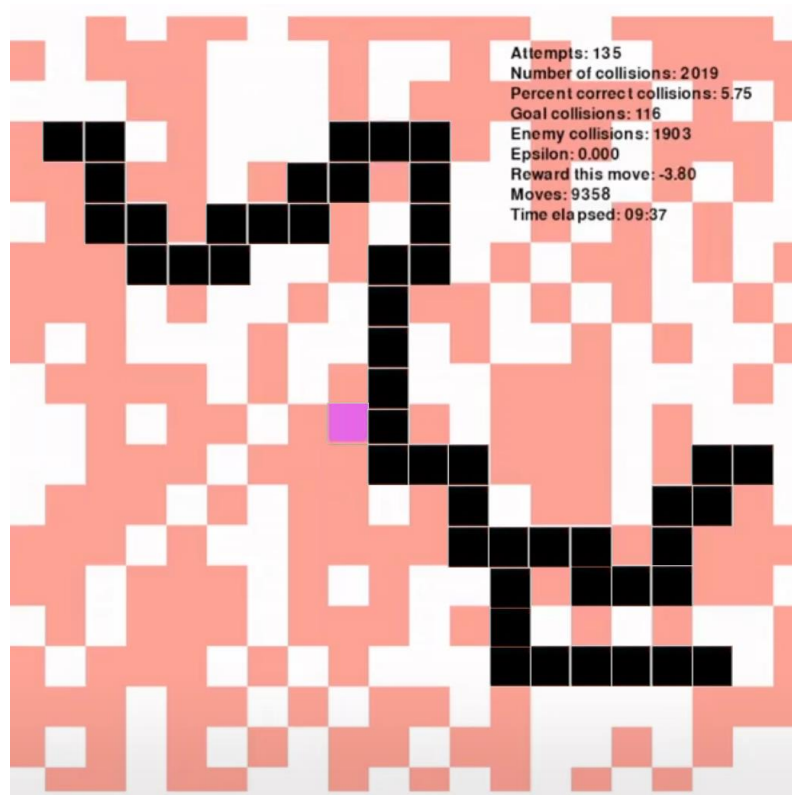


Figure 33: Pathways used by the agent after training is complete.

A three second video of the result can be seen on the following URL: <https://i.imgur.com/XlvAFsQ.mp4>

We see that after approximately nine thousand moves and one hundred and thirty episodes, the agent has learned to play the game quite well (Fig. 33). Not only is there not a single collision with the enemy, but an optimal path is chosen for each position. This is *in spite* of the fact that many of the decisions the agent has to take are long-term oriented. For example, in their first position, the player has to actually *increase the distance* from the goal by going *up* on the map to avoid the obstacles to the right.

This ability can be attributed to the part of Q-learning's function where part of the reward is based on the subsequent move that the agent makes. In effect, each field on the map is tied to all the other fields of the map—granted that they have been explored. That is also why it was so effective to keep the epsilon high at the beginning of the training.

If we look at the exported Q-table that the agent is operating on in this final stage of training (Fig. 34, 35), we notice that most of the map has been explored. Figure 36 visualizes the process on a graph.

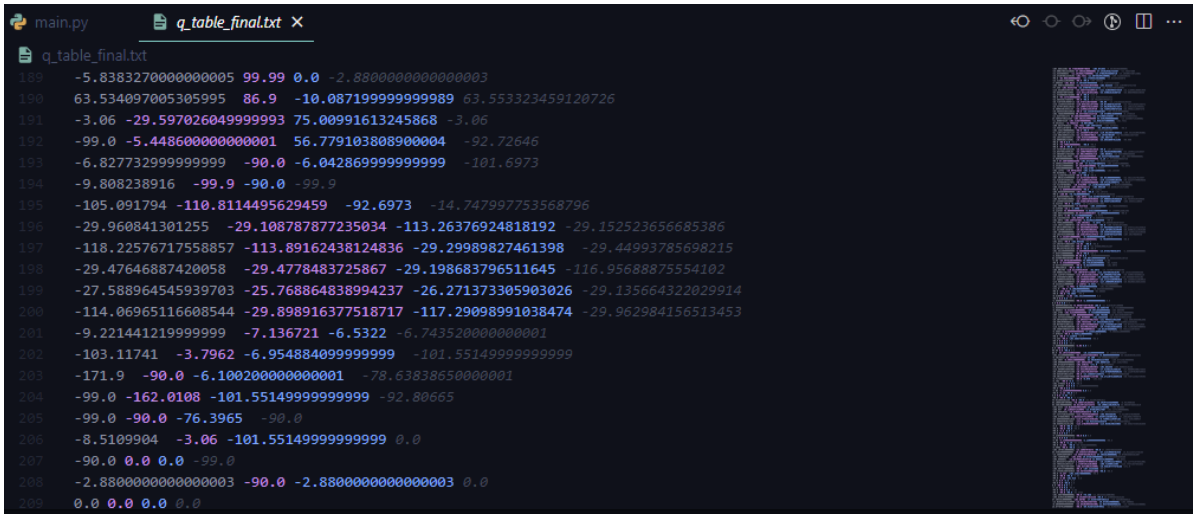


Figure 34: The columns with the highest negative values are actions that lead to enemies.

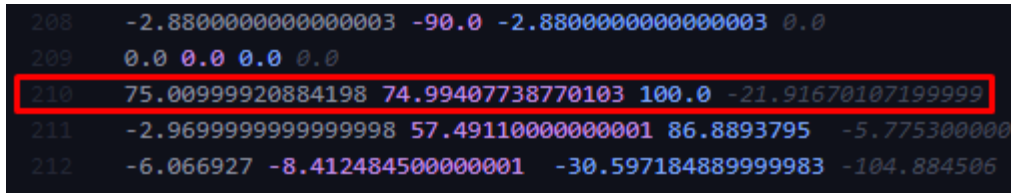


Figure 35: One of the fields that leads directly to the goal via the third action—movement to the left. The agent will always pick the highest value with an epsilon of zero.

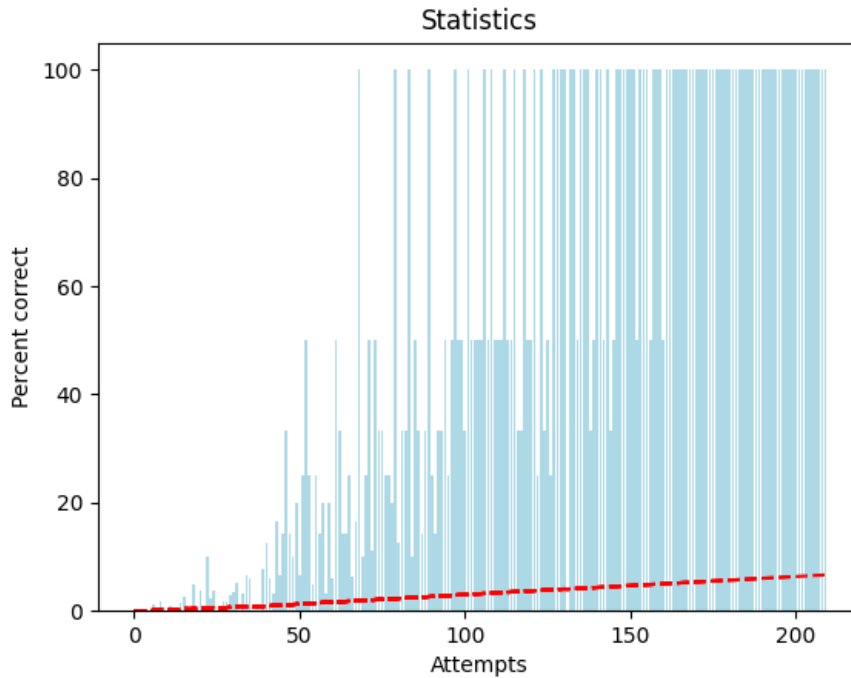


Figure 36: The matplotlib bar chart that was generated for one of the runs. We may notice that the success curve is comparatively linear. One of the reasons for this is that the epsilon value decreases linearly.

## Comparison and Hyperparameter Tweaking

### The First Run (Default)

The first run contains hyperparameters with high values, gives highly contrasting reward functions, and slowly, linearly diminishes the value of epsilon. The number of moves is limited to roughly 10,000. Subsequent runs will be tested to find more optimal values. It should be noted that the low percent of total correct collisions is due to the fact that a majority of the moves in runs occurs while the agent is being trained, and the episode doesn't end if the player touches the enemy.

#### The first run contains the following parameters:

Learning rate: 0.9

Discount rate: 0.9

Initial epsilon: 0.9

Epsilon changed every move by: -0.0001

#### And the following rewards:

Reward for green square: 100

Reward for red square: -100

Reward for move:  $0.1 * (20 - \text{distance}) - 5$

#### The results (Fig. 37):

First 100% correct collision episode: 47

Time elapsed until 100% correct collision episode: 80

Total correct collisions in percent: 6.26%

Episodes: 149

Moves: 10003

Time elapsed in seconds: 122

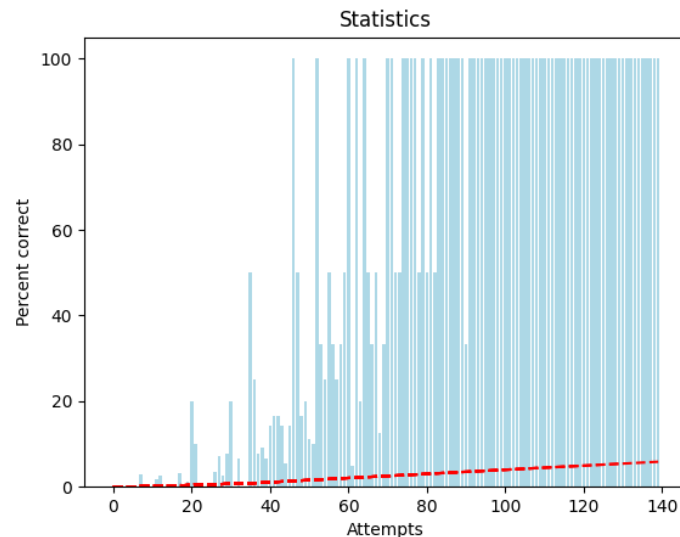


Figure 37: The first run.

It should be noted that Matplotlib graphs have difficulty displaying this many bars in graphs, so places in which it seems like lines are missing are a result of displaying errors. Likewise, it is worth remembering that the x coordinate is not in the time domain, but shows episodes. Episodes are longer if the goal isn't reached (200 moves) as opposed to being reached optimally (a few dozen moves).

## Testing Reward Functions

If we reduce reward functions for goal and enemy squares by two orders of magnitude, we get a dysfunctional agent that doesn't work at all (Fig. 38).

### Rewards:

Reward for green square: 1

Reward for red square: -1

Reward for move:  $0.1 * (20 - \text{distance}) - 5$

### The results:

First 100% correct collision episode: X

Time elapsed until 100% correct collision episode: X

Total correct collisions in percent: **0.3%**

Episodes: 61

Moves: 10048

Time elapsed in seconds: 114

If we instead try only reducing one order of magnitude, the results get better, but ultimately still worse than by default (Fig. 39).

### Rewards:

Reward for green square: 10

Reward for red square: -10

### The results:

First 100% correct collision episode: 62

Time elapsed until 100% correct collision episode: 81

Total correct collisions in percent: **3.56%**

Episodes: 133

Moves: 10008

Time elapsed in seconds: 116

So perhaps the answer lies in *increasing* the reward functions. (It should be noted that what we're testing in this case is merely the relationship between rewards for movement and rewards for collision. The actual rewards in absolute terms don't matter outside this relationship.) We appear to have gotten a more consistent success rate as epsilon approaches zero, while the total success rate is lower because the initial rate was worse—more enemies were run into at the beginning (Fig. 40).

### Rewards:

Reward for green square: 1000

Reward for red square: -1000

### The results:

First 100% correct collision episode: 60

Time elapsed until 100% correct collision episode: 89

Total correct collisions in percent: **5.36% (second try: 5.8%)**

Episodes: 139

Moves: 10002

Time elapsed in seconds: 116

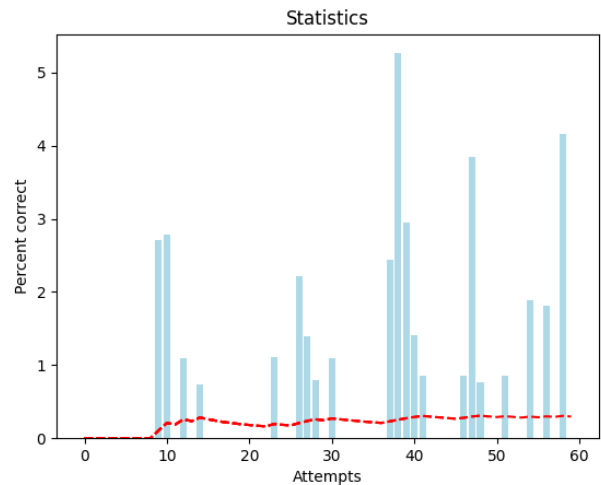


Figure 38: Graph with collision rewards of 1, -1.

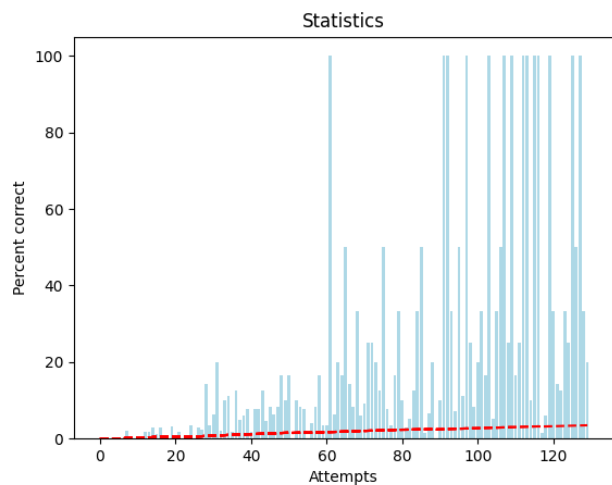


Figure 39: Graph with collision rewards of 10, -10.

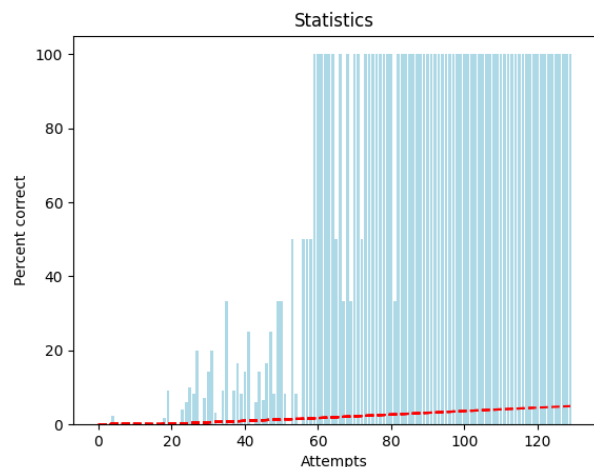


Figure 40: Graph with collision rewards of 1000, -1000.

We decide to keep collision rewards the same (100, -100), and remove the distance part of the formula for movement reward. In other words, the agent is punished for movement regardless if close or far away to the goal square (Fig. 41).

The results are not too distinguishable, but still appear to be worse on average, which makes sense given that factoring distance to the goal square provides a benefit.

#### Rewards:

Reward for green square: 100

Reward for red square: -100

Reward for move: -0.1

#### The results:

Total correct collisions in percent: 5.0%

In conclusion, the difference between collision rewards and move rewards has to be substantial. In the first two cases, we can reason that many cells in the q-table were unreliable due to the fact that the move reward could sometimes be larger than the actual goal reward, leading to an agent that is more capable of avoiding enemies than reaching the goal. If the function reward for movement was different—for instance, if it only contained a simple negative reward for every move—then collision rewards could be substantially smaller.

Additionally, it is clear that adding awareness to the agent regarding its distance to the goal has proven to be useful.

No changes will be made to the default settings when it comes to reward functions.

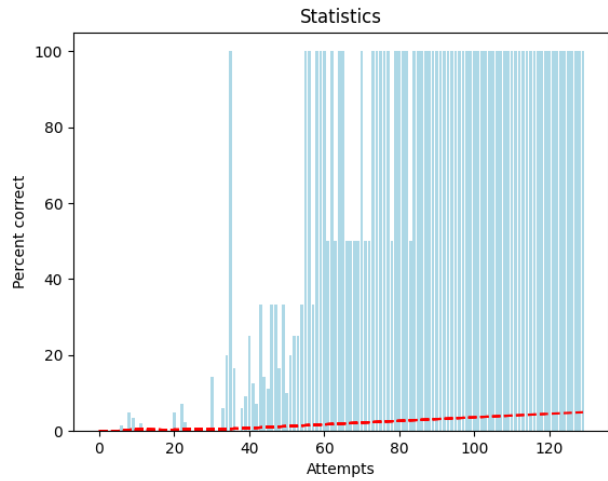


Figure 41: Graph with no goal tracking in the reward function.

## Testing the Epsilon Value

When the function for decreasing epsilon over time is turned off, we find permanent inefficiencies in the agent's strategy. Figure 42 shows the results when epsilon is constant.

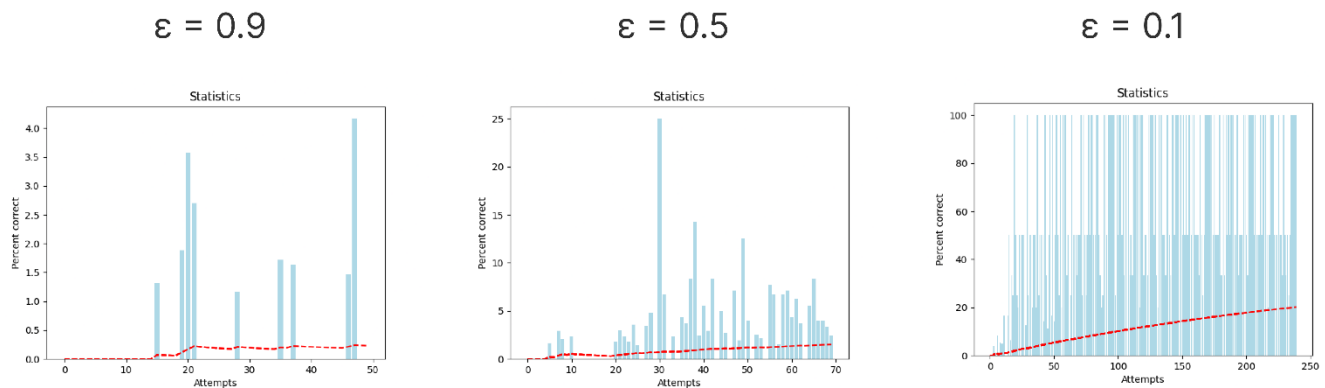


Figure 42: Graphs with decreasing static epsilon values.

This is to be expected. The agent is never able to fully utilize its q-table, and thus always makes some random decisions. We find that the lower the epsilon value, the better the performance.

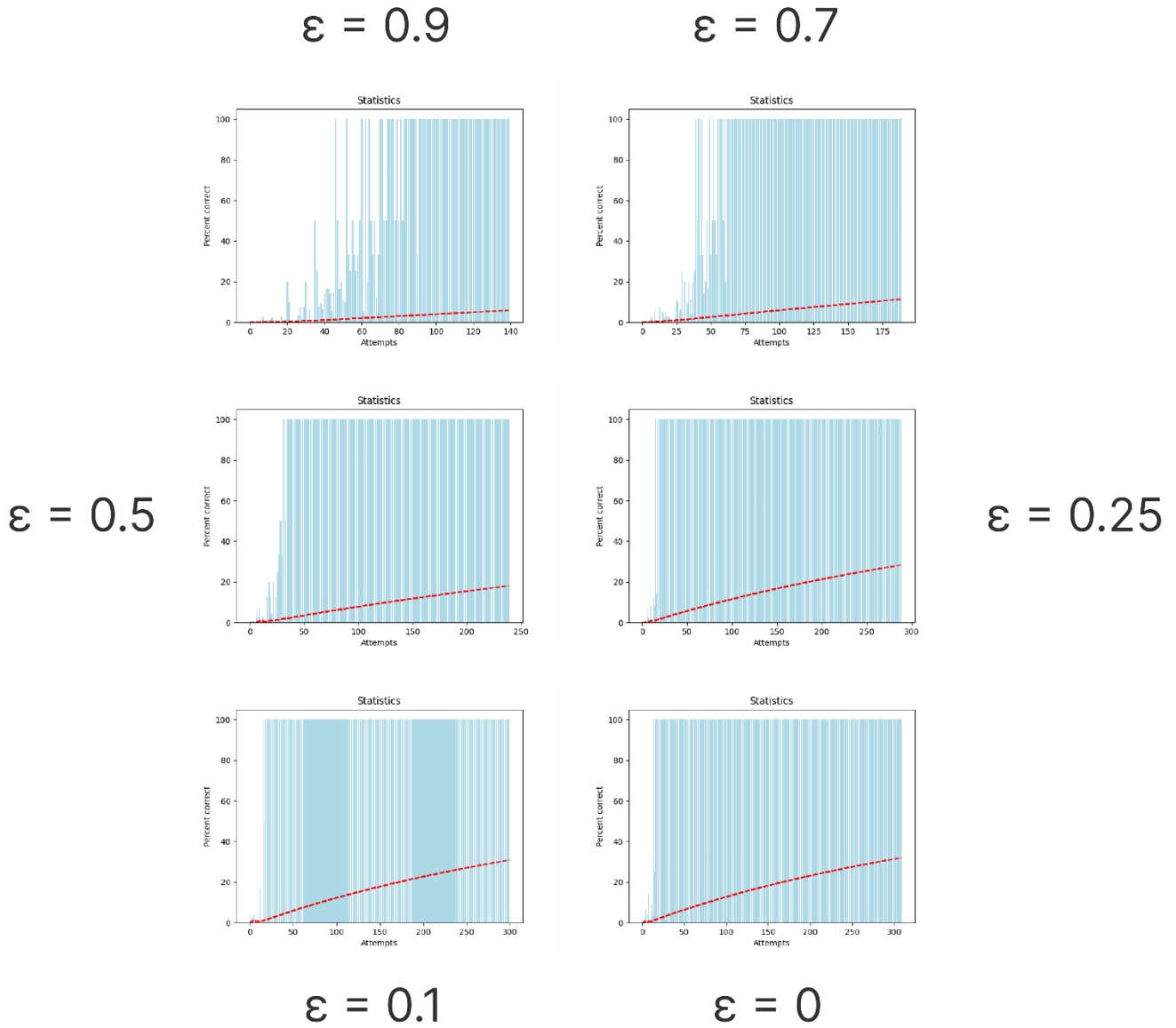


Figure 43: Graphs with decreasing epsilon values.

Furthermore, as shown in Figure 43, we find that the epsilon value is not necessary at all in this simulation. This is because the agent de facto explores the map regardless, especially given the fact that the q-table does not contain any information. The pattern of the agent's exploration will not be random, however, but usually follow some kind of structure: for example, the agent initially explored the map in vertical lines until it reached the goal.

It is far more efficient to allow the agent to explore at a dynamic pace. When epsilon equals 0, overall correct collisions climb to over 32%. The antagonistic relationship between exploration and exploitation continues into this strategy, the only difference is that it is no longer tied to epsilon. The agent always learns so long the q-table is updated.

It should, however, be noted that the agent simply explores more of the map with epsilon. An agent is better prepared for any task when having explored more of the map. The main drawback is that the training will be costlier, and in this case, it is simply not necessary for the agent to have a very in depth understanding of the map.

To optimize the agent, epsilon will now be initialized with the value of zero. Because of a major increase in efficiency, the number of moves will no longer be limited to 10,000 but 5,000 in further testing.

## Testing the Learning and Discount Rate

### Learning Rate

Decreasing the learning rate appears to make the agent much slower and its performance worse (Fig. 44). This is within expectations—given that we are simply decreasing the rate at which new information is absorbed. Learning rate will be increased to 1.

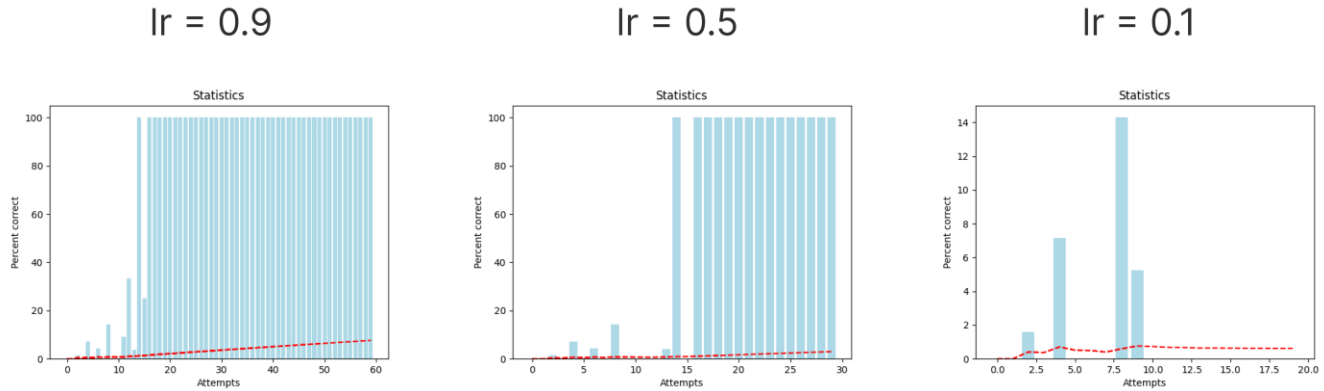


Figure 44: Graphs of decreasing learning rate values.

### Discount Rate

The discount rate likewise shares similarity in effect, but the agent becomes dysfunctional relatively quickly (Fig. 45).

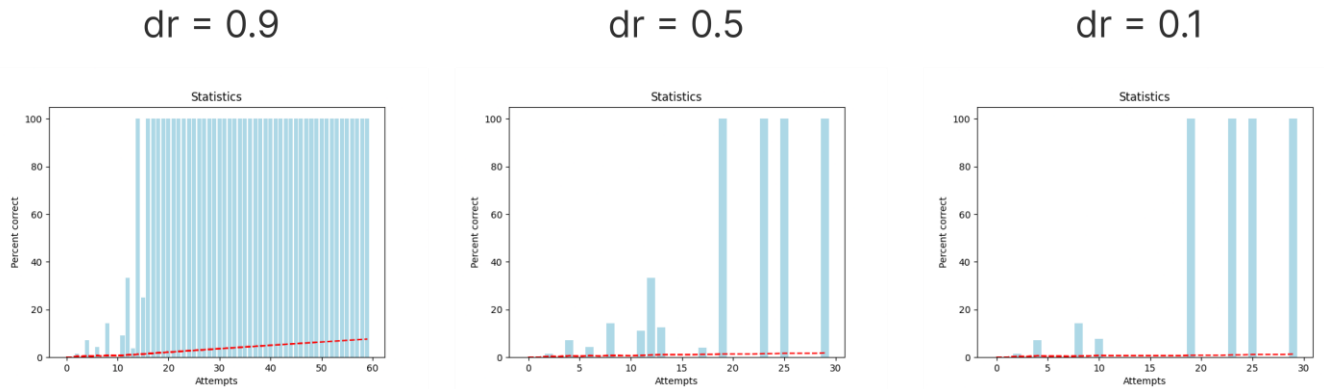


Figure 45: Graphs of decreasing discount rate values.

Nonetheless, the agent also displays strange behavior when the discount rate equals 1—as it enters suboptimal equilibria. For example, as we can see in Figure 46, it will get stuck hitting an enemy before reaching the goal, and its correct collision rate will stay at 50% half of the time. For this reason, discount rate is kept at 0.9.

Because a discount rate of 1 induces an infinite time horizon, we can speculate that the agent is simply incapable of prioritizing any meaningful action. It is unknown how the agent behaves in this circumstance exactly.

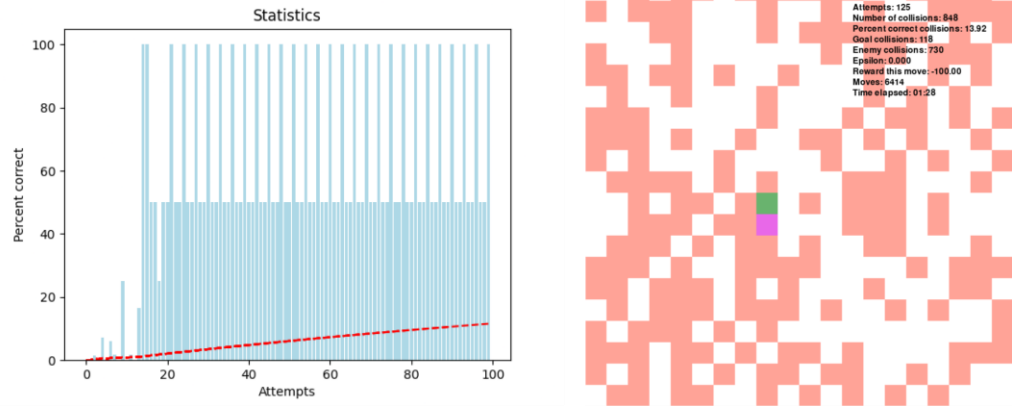


Figure 46: We can observe that the agent decides to reach the goal by going under the square, which causes a collision with an enemy. Hence, the episode ends with a 50% correct collision rate.



## Optimization

### Hyperparameters (first run):

Learning rate: 0.9

Discount rate: 0.9

Initial epsilon: 0.9

Epsilon changed every move by: -0.0001

### Rewards (first run):

Reward for green square: 100

Reward for red square: -100

Reward for move:  $0.1 * (20 - \text{distance}) - 5$

### Results (first run):

First 100% correct collision episode: 47

Time elapsed until 100% correct collision episode: 80

Total correct collisions in percent: 6.26%

Episodes: 149

Moves: 10003

Time elapsed in seconds: 122

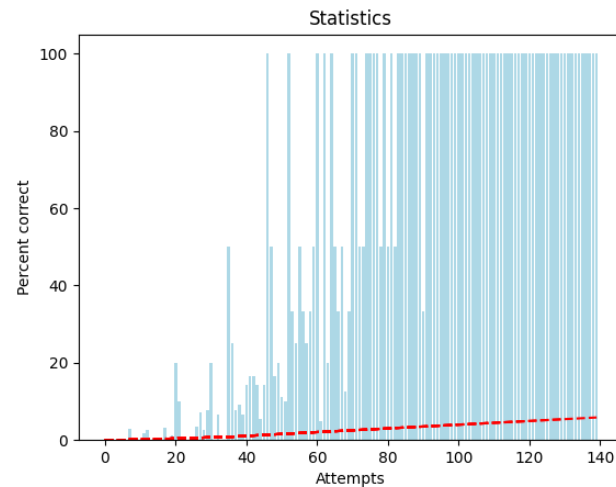


Figure 47: Graph of first run.

### Hyperparameters (optimized):

Learning rate: 1

Discount rate: 0.9

Epsilon: 0

### Rewards (optimized):

Reward for green square: 100

Reward for red square: -100

Reward for move:  $0.1 * (20 - \text{distance}) - 5$

### Results (optimized):

First 100% correct collision episode: 10

Time elapsed until 100% correct collision episode: 21

Total correct collisions in percent: 32.66%

Episodes: 315

Moves: 10017

Time elapsed in seconds: 150

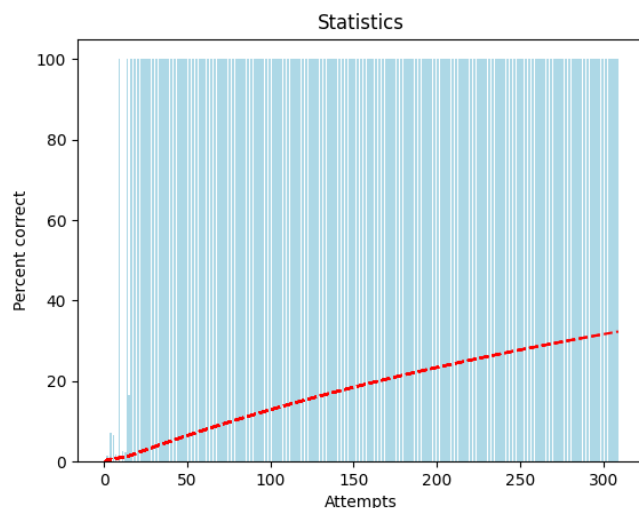


Figure 48: Graph of optimized run.

The algorithm has been successfully optimized in several ways through testing and tweaking hyperparameters (Fig. 47, 48). The percentage of correct collisions in the same time frame has been increased by 422% and stands at 522% the original number. Looking at when 100% correct collision episodes occur, we can conclude that it takes the agent between 20% and 30% as much time to be trained, which would cut training time by 70-80%.

## Discussion and Analysis

This simulation shows how to implement Q-learning in a 2D space (video game) and utilize reinforcement learning's tendency to optimize and solve problems in dynamic programming.

### Limitations

The simulation in this thesis is fairly simplistic. It is not a sophisticated gaming paradigm even within the genre of two-dimensional games. Thus, some features of Q-learning are mostly redundant, as we have been able to observe in the process of fine-tuning the algorithm.

Furthermore, the knowledge that the agent acquires only applies to the map it has been trained in, which means that, if applied to a real video game, the agent could only respond within a game level or room or some other constrained space, and could not be used in foreign contexts within the video game. Nonetheless, assuming training is not expensive, agents can be trained in sections of the game separately.

### Improvements and Alternative Solutions

The environment the agent operates in could be made significantly more complex, and as a result, allow more features to become relevant and create a more interesting gaming experience. Adding more elements other than enemies and goals—such as health items or weapons—would improve the game vertically within the existing paradigm, but adding a third dimension or adding more actions and generalizing the agent would improve it laterally and fundamentally rework the game.

The agent could also be used as a non-playable character (NPC) in the game, especially if the game was story-driven. It would potentially reduce the time needed to animate the character, for example.

The learning could be generalized by training the agent to be sensitive to its distance from all objects rather than simply learning about the map of the level. The Q-table would significantly change shape, and it would also be possible to introduce moving objects—albeit Q-learning is not optimal for very complicated state-action pairs.

## Conclusion

This paper has delved into the world of reinforcement learning via a Python puzzle video game and the implementation of a Q-learning algorithm. By having complete control over the development process, it was possible to track, explain, and visualize many parts of this algorithm. In doing so, an introductory resource for comprehending mechanisms of machine learning has been made, specifically reinforcement learning, when applied to video game environments.

Furthermore, we have explored the optimization potential of the Q-learning formula by experimenting with various hyperparameters. We optimized the setup of the program and improved its training speed up to 80%. Certain features of the Q-formula proved to be redundant due to the simplicity of the simulation. That highlights the importance of adaptability in the realm of machine learning when effectiveness is desired.

This paper sought to illuminate and simplify the entire process for its readership.

## Table of Figures

Figure 1: An illustration of reinforcement learning from Wikimedia (CC0). URL: <a href="https://commons.wikimedia.org/wiki/File:Reinforcement_learning_diagram.svg">https://commons.wikimedia.org/wiki/File:Reinforcement_learning_diagram.svg</a> .....	5
Figure 2: This MDP has three states, two actions for each, and two rewards (the orange arrows). URL (Wikimedia, CC0): <a href="https://commons.wikimedia.org/wiki/File:Markov_Decision_Process.svg">https://commons.wikimedia.org/wiki/File:Markov_Decision_Process.svg</a> .....	7
Figure 3: A Bellman flow chart. URL (Wikimedia, CC0): <a href="https://commons.wikimedia.org/wiki/File:Bellman_flow_chart.png">https://commons.wikimedia.org/wiki/File:Bellman_flow_chart.png</a> .....	8
Figure 4: The Q-learning algorithm formula. URL (Wikipedia): <a href="https://en.wikipedia.org/wiki/Q-learning#Algorithm">https://en.wikipedia.org/wiki/Q-learning#Algorithm</a> .....	10
Figure 5: Included programming libraries in the main.py and stats.py files. ....	12
Figure 6: Thomas is not placed at different places on the map randomly. ....	13
Figure 7: The player is given instructions at the beginning of the game. The right side of the image shows implementation. ....	13
Figure 8: Initialization of PyGame. ....	14
Figure 9: Location of actors and objects.....	14
Figure 10: Screenshot of a part of the grid_maker.py file. ....	14
Figure 11: Look of the game.....	15
Figure 12: Creation of PyGame Rect objects and drawing them on the screen.....	15
Figure 13: After performing collision detection, the player's color is determined by the outcome.....	16
Figure 14: An array of actions is initialized before the game runs. Once an action is chosen, the player moves along the map matrix according to the array index. If an action leads to the player moving out of bounds, no action is taken.....	16
Figure 15: Initialization of these variables appears right before collision detection logic. ....	17
Figure 16: Collision check for the goal.....	17
Figure 17: Collision check for enemies. ....	18
Figure 18: If no collision occurs. ....	18
Figure 19: A prevention measure against getting stuck in a suboptimal state. ....	18
Figure 20: Initialization of the Q-table. ....	19
Figure 21: The Q-table is initialized with zeros.....	19
Figure 22: The hyperparameters. ....	20
Figure 23: Epsilon decreases with each move as the game loop progresses. When epsilon is 0, the Q-table is no longer updated, and it is sent to a file. The speed of the simulation also slows so that the viewer can get a glimpse of how the agent behaves after training. ....	20
Figure 24: Action-taking.....	20
Figure 25: Updating the Q-table. ....	21
Figure 26: The console prints out every update of the Q-table. This makes debugging simpler and concretizes the formula.....	21
Figure 27: The player can get access to elapsed time, number of attempts or episodes, rewards, number of total moves, etc. ....	22
Figure 28: Resetting player's position on the map. ....	22
Figure 29: Collision-tracking variables. ....	23
Figure 30: Sending data every 10 <sup>th</sup> episode. ....	23
Figure 31: Bar plot function.....	23
Figure 32: Saving parameters. ....	24
Figure 33: Pathways used by the agent after training is complete.....	25
Figure 34: The columns with the highest negative values are actions that lead to enemies. ....	26

Figure 35: One of the fields that leads directly to the goal via the third action—movement to the left. The agent will always pick the highest value with an epsilon of zero.....	26
Figure 36: The matplotlib bar chart that was generated for one of the runs. We may notice that the success curve is comparatively linear. One of the reasons for this is that the epsilon value decreases linearly. ....	26
Figure 37: The first run.....	27
Figure 38: Graph with collision rewards of 1, -1.....	28
Figure 39: Graph with collision rewards of 10, -10.....	28
Figure 40: Graph with collision rewards of 1000, -1000.....	28
Figure 41: Graph with no goal tracking in the reward function.....	29
Figure 42: Graphs with decreasing static epsilon values. ....	29
Figure 43: Graphs with decreasing epsilon values. ....	30
Figure 44: Graphs of decreasing learning rate values.....	31
Figure 45: Graphs of decreasing discount rate values. ....	31
Figure 46: We can observe that the agent decides to reach the goal by going under the square, which causes a collision with an enemy. Hence, the episode ends with a 50% correct collision rate.....	32
Figure 47: Graph of first run.....	33
Figure 48: Graph of optimized run. ....	33

# Bibliography

- [1] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, London: The MIT Press, 1992.
- [2] L. P. Kaelbling, M. L. Littman and A. W. Moore, *Reinforcement Learning: A Survey*, Journal of Artificial Intelligence Research, 1996.
- [3] C. M. Bowyer, "A Crude History of Reinforcement Learning (RL)," 2022.
- [4] P. Gray, Psychology, New York: Worth, 2014.
- [5] I. Ross, A primer on Pontryagin's principle in optimal control, San Francisco: Collegiate Publishers, 2015.
- [6] R. Bellman, "A Markovian Decision Process," *Indiana Univ. Math. J.*, pp. 679--684, 1957.
- [7] A. Ye, "A Crash Course in Markov Decision Processes, the Bellman Equation, and Dynamic Programming".
- [8] Glossary of Artificial Intelligence (AI), Machine Learning (ML), and Big Data Terms, "Episode," [Online]. Available: <https://www.aidatatoday.com/glossary/episode>.
- [9] K. P. Murphy, "Logistic Regression," in *Machine Learning: A Probabilistic Perspective*, Cambridge: MIT Press, p. 247.
- [10] "Q-learning," [Online]. Available: <https://en.wikipedia.org/wiki/Q-learning>.
- [11] F. S. Melo, "Convergence of Q-learning: a simple proof," Institute for Systems and Robotics, Lisboa.
- [12] A. Anto, "Machine Learning - Why Python is the most popular language," [Online]. Available: <https://www.zarantech.com/blog/machine-learning-python-popular-language/>.
- [13] "Pygame Front Page," [Online]. Available: <https://www.pygame.org/docs/>.
- [14] C. R. Harris and e. al., "Array programming with NumPy," [Online]. Available: <https://www.nature.com/articles/s41586-020-2649-2.pdf>.
- [15] J. Hunter, D. Dale, E. Firing and M. Droettboom, "Matplotlib 3.7.2 documentation," [Online]. Available: <https://matplotlib.org/3.7.2/index.html>.
- [16] Habr, "Creating desktop applications in Python," [Online]. Available: <https://habr.com/en/sandbox/184708/>.
- [17] "pygame.draw," 1 July 2023. [Online]. Available: <https://www.pygame.org/docs/ref/draw.html#pygame.draw.rect>.
- [18] "pygame.Rect," [Online]. Available: <https://www.pygame.org/docs/ref/rect.html>. [Accessed 1 July 2023].
- [19] I. Rehman, N. Mahabadi, T. Sanvictores and C. I. Rehman, "Classical Conditioning," StatPearls Publishing LLC, 2023.