



**NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS**

**SCHOOL OF SCIENCE  
DEPARTMENT OF PHYSICS**

**INTERFACULTY PROGRAM OF POSTGRADUATE STUDIES IN ELECTRONICS-  
RADIOELECTROLOGY**

**DIPLOMA THESIS**

**Reinforcement Learning for Rogue-like Games**

**Ilias K Simakos**

**Supervisor: Dionysios Reisis, Professor**

**ATHENS**

**NOVEMBER 2022**

# DIPLOMA THESIS

Reinforcement Learning for Rogue-like Games

**Ilias K. Simakos**

**R.N.: 2019109**

**SUPERVISOR:** **Dionisios Reisis**, Professor

**EXAMINATION  
COMMITTEE:**

**Dionysios Reisis**, Professor  
**Hector Nistazakis**, Professor  
**Nikolaos Vlassopoulos**, Visiting Faculty

November 2022

## ΠΕΡΙΛΗΨΗ

Η Τεχνητή Νοημοσύνη artificial intelligence (AI) είναι ένα σημαντικό μέρος της τέταρτης βιομηχανικής επανάστασης και υπόσχεται να αλλάξει δραματικά τη ζωή μας, αναλαμβάνοντας κουραστικές ή επικίνδυνες εργασίες για εμάς. Ωστόσο, πρέπει να γίνει ακόμα πολλή δουλειά. Μια αποτελεσματική οντότητα τεχνητής νοημοσύνης που θα εκτελέσει δύσκολες εργασίες, που μόνο ένας άνθρωπος μπορεί, είναι ακόμα μακριά. Έτσι, καταβάλλονται συνεχείς προσπάθειες για τη δημιουργία επιτυχημένων οντοτήτων AI. Σε αυτή τη διαδικασία τα βιντεοπαιχνίδια είναι ένα σημαντικό πεδίο δοκιμών, είναι εργασίες που περιλαμβάνουν εκμάθηση και συλλογισμό για έννοιες σε πολλαπλά επίπεδα αφαίρεσης, έναν τομέα στον οποίο η τεχνητή νοημοσύνη υστερεί σε σχέση με την ανθρώπινη. Για το λόγο αυτό, οι αλγόριθμοι τεχνητής νοημοσύνης που καταφέρνουν να παίξουν ένα παιχνίδι σε ανθρώπινο επίπεδο θεωρούνται επιτυχία και ένα βήμα μπροστά για τον κλάδο.

Μεταξύ των τομέων της τεχνητής νοημοσύνης, η Ενισχυτική Μάθηση Reinforcement Learning (RL) χρησιμοποιείται συχνότερα για την εκπαίδευση ενός υπολογιστή, που συνήθως ονομάζεται πράκτορας (agent), στο πώς να παίζει βιντεοπαιχνίδια. Αυτή η κατηγορία AI είναι πιο στοχευμένη από τις άλλες, συγκεκριμένα η RL βασίζεται στην αλληλεπίδραση με ένα αβέβαιο περιβάλλον για την επίτευξη ενός τελικού στόχου, μια συμπεριφορά που ταιριάζει με αυτή ενός ανθρώπου που προσπαθεί να παίξει ένα παιχνίδι. Τα τελευταία χρόνια, ένας πράκτορας που εκπαιδεύτηκε με αλγόριθμους RL χρησιμοποιώντας βαθιά νευρωνικά δίκτυα, deep neural networks (DNN), για την προσέγγιση των συναρτήσεων που χρησιμοποιεί, μπόρεσε να παίξει σε ικανοποιητικό βαθμό σχετικά απλά παιχνίδια της κονσόλας ATARI2600 [1], ανοίγοντας το δρόμο για τους πράκτορες AI έτσι ώστε να καταφέρουν να παίξουν ακόμα πιο περίπλοκα παιχνίδια.

Στην παρούσα εργασία, ένα rogue-like παιχνίδι δημιουργείται για να χρησιμεύσει ως βάση δοκιμών για έναν αλγόριθμο RL που συνδυάζει πρόσφατες προηγμένες τεχνικές, όπως η ασύγχρονη εκπαίδευση με χρήση πολυεπεξεργαστή, συνελκτικά νευρωνικά δίκτυα, convolutional neural networks (CNN), και δίκτυα μακράς βραχύχρονης μνήμης long short-term Memory (LSTM). Επίσης έγινε χρήση και κάποιων μοναδικών τεχνικών. Οι τεχνικές αφορούν στην χρήση των γραπτών μηνμάτων του παιχνιδιού και της καταγραφής της κατάστασης του ήρωα του παιχνιδιού ως εισόδους στα DNNs. Οι τεχνικές αυτές επιλέχθηκαν λόγω της σημασίας που έχουν αυτές οι πληροφορίες για έναν άνθρωπο που προσπαθεί να παίξει το παιχνίδι της διπλωματικής. Ο τύπος παιχνιδιού rogue-like επιλέγεται λόγω της δυσκολίας του, που οφείλεται στην έλλειψη ντετερμινισμού (κάθε σετ παιχνιδιού δημιουργείται τυχαία), στην ανάγκη χρήσης χρονικά εκτεταμένων στρατηγικών και στην μερική αναπαράσταση της κατάστασης του στον παίκτη.

Στο πρώτο κεφάλαιο της διπλωματικής εργασίας, θα γίνει μια σύντομη αναφορά στα rogue-like παιχνίδια και συγκεκριμένα σε αυτό που χρησιμοποιείται στην εργασία, ενώ ακολουθεί μια εισαγωγή στο RL και μια σύντομη περιγραφή των DNNs, δίνοντας έμφαση στα δίκτυα που χρησιμοποιούνται στη διπλωματική. Στο δεύτερο κεφάλαιο, θα παρουσιαστεί η υλοποίηση του αλγορίθμου RL που χρησιμοποιείται, συμπεριλαμβανομένης της αρχιτεκτονικής του δικτύου. Θα δοθεί περαιτέρω μια συλλογιστική ως προς το ποιες τεχνικές χρησιμοποιούνται για την αντιμετώπιση των προκλήσεων ενός rogue-like παιχνιδιού. Στο τρίτο κεφάλαιο, θα παρουσιαστούν και θα αξιολογηθούν τα αποτελέσματα της εκπαίδευσης ενός πράκτορα χρησιμοποιώντας τις τεχνικές που περιγράφονται στο δεύτερο κεφάλαιο. Το τελευταίο κεφάλαιο περιλαμβάνει τα συμπεράσματα που προκύπτουν από τη διπλωματική εργασία και μερικές νεότερες τεχνικές που φαίνονται πολλά υποσχόμενες και μπορούν να βοηθήσουν στη βελτίωση της απόδοσης των μελλοντικών πρακτόρων RL στο παιχνίδι της διπλωματικής.

**ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ:** Ενισχυτική Μάθηση

**ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ:** ενισχυτική μάθηση πολιτικής, ασύγχρονη ενισχυτική μάθηση, βελτιστοποίηση, συνελκτικά νευρωνικά δίκτυα, δίκτυα μακράς βραχύχρονης μνήμης

## ABSTRACT

Artificial Intelligence (AI) is an important part of the fourth industrial revolution, and it promises to dramatically change our lives by taking over tedious or dangerous tasks for us. Although, a lot of work still needs to be done. An effective artificial intelligence entity that will perform difficult tasks, which only a human can, is still far away. Thus, continuous efforts are being made to create successful AI entities. In this process, video games are an important testing ground, since they are tasks that involve learning and reasoning about concepts over multiple levels of abstraction, an area in which artificial intelligence lags behind human intelligence. For this reason, AI algorithms that manage to play a game at a human level are considered a success and a step forward in the industry.

Among the fields of artificial intelligence, Reinforcement Learning (RL) is most often used to train a computer, usually called an agent, how to play video games. This category of AI is more goal directed than the others, specifically, the RL is based on interacting with an uncertain environment and achieving a final goal, a behavior that matches a human trying to play a game. In recent years, an agent trained with RL algorithms using deep neural networks (DNN) to approximate functions, was able to play relatively simple ATARI2600 games at a human level [1] paving the way for AI agents to master more complex games.

In this thesis, a Rogue-like game is created to serve as a testbed for an RL algorithm that combines recent advanced techniques such as, asynchronous training using multiprocessors, convolutional neural networks (CNNs), and long-short-term memory (LSTM) networks. Also, some unique techniques were used. The techniques are the use of the game hero status information and the game text messages as inputs to the DNNs. This choice was made because of the importance of this information to a human playing the game. The Rogue-like type of game is chosen because of its difficulty stemming from their lack of determinism (each game set is randomly generated), the need for time-extended strategies, and the partial representation of their state to the player.

The first chapter of the thesis will introduce the rogue-like games and specifically the one used in the thesis, followed by an introduction to RL and a brief description of DNNs, emphasizing the networks used in the thesis. The second chapter will present the implementation of the RL algorithm used including the network architecture. A reasoning will be further given as to what techniques are used to address the challenges of a rogue-like game. In the third chapter, the results of the training of an agent using the techniques, described in chapter two, will be presented and evaluated. The final chapter will include the conclusions drawn from the thesis and some newer techniques that look promising and can help improve the performance of future RL agents in the thesis game.

**SUBJECT AREA:** Reinforcement Learning

**KEYWORDS:** policy gradient, asynchronous reinforcement learning, optimization, convolutional neural networks, long short-term memory network

## **ACKNOWLEDGEMENTS**

I would like to thank my supervisor professor Dionysios Reisis for making this work possible and Mr.Nikolaos Vlassopoulos for his advice and guidance. Also, my family and partner for their support and patience.

# CONTENTS

<b>ΠΕΡΙΛΗΨΗ .....</b>	<b>3</b>
<b>ABSTRACT .....</b>	<b>5</b>
<b>ACKNOWLEDGEMENTS.....</b>	<b>6</b>
<b>1. INTRODUCTION.....</b>	<b>12</b>
<b>1.1 Description of Rogue-like games.....</b>	<b>12</b>
1.1.1 Description of the game used in the thesis.....	12
<b>1.2 Introduction to Artificial Intelligence with focus on Reinforcement Learning.....</b>	<b>20</b>
1.2.1 Machine Learning categories .....	20
1.2.2 Reinforcement Learning.....	21
<b>1.3 Function approximation and Neural-Networks. ....</b>	<b>35</b>
1.3.1 RL optimization and gradient descent methods.....	36
1.3.2 Artificial Neural Networks.....	38
<b>1.4 Challenges that Rogue-like games have .....</b>	<b>43</b>
<b>2. IMPLEMENTATION .....</b>	<b>44</b>
<b>2.1 Publications and related works that influenced the thesis .....</b>	<b>44</b>
<b>2.2 Algorithm used.....</b>	<b>44</b>
2.2.1 Parameterized functions and gradient calculation.....	44
2.2.2 Adam optimizer [21].....	45
2.2.3 Data correlation and asynchronous methods for deep reinforcement learning.....	46
2.2.4 Algorithm explanation and its pseudocode .....	47
<b>2.3 Network architecture and inputs .....</b>	<b>49</b>
2.3.1 Input preprocessing.....	49
2.3.2 Network architecture.....	50
<b>2.4 Reward Signal .....</b>	<b>51</b>
<b>2.5 Trying to deal with the challenges of a Roguelike game.....</b>	<b>52</b>

<b>2.6 Code used in the thesis .....</b>	<b>52</b>
<b>3. RESULTS .....</b>	<b>54</b>
<b>4. CONCLUSIONS.....</b>	<b>58</b>
<b>APPENDIX I: HYPERPARAMETERS.....</b>	<b>60</b>
<b>APPENDIX II: NETWORKS BLOCK DIAGRAM.....</b>	<b>61</b>
<b>APPENDIX III: AGENT TRAINING CODE .....</b>	<b>63</b>
<b>ABBREVIATIONS – ACRONYMS .....</b>	<b>73</b>
<b>REFERENCES .....</b>	<b>74</b>



## FIGURES

Figure 1: A cave consist of 80×60 tiles with 46% Floor percentage.....	13
Figure 2: The ring of the Wizard Werdna, introduction screen.....	13
Figure 3: The ring of the Wizard Werdna, name selection screen.....	14
Figure 4:Hero Depiction.....	14
Figure 5: A depiction of the game after some movements of the hero is given above. The bright red for the tiles means that are inside the hero's visibility and the les opaque red means that the tiles visibility type is fogged.....	15
Figure 6: Enemy rendering example (a giant rat and a Goblin) .....	17
Figure 7: Health potion illustration (left), Mana potion illustration (center) weapon illustration (right) .....	19
Figure 8: Hero Status.....	19
Figure 9: An example of the game log .....	20
Figure 10: The agent–environment interaction in a Markov decision process. [4] .....	22
Figure 11: GPI procedure [4] .....	27
Figure 12: Optimization of the value and policy function through iteration of policy evaluation and policy improvement.[4].....	27
Figure 13: Basic feedforward ANN representation [4].....	38
Figure 14:Examples of activation function[9] .....	38
Figure 15: Architecture of LeNet-5. Each plane is a feature map, i.e., a set of units whose weights are constrained to be identical.[11].....	39
Figure 16: A recurrent network: a network with a loop [13].....	40
Figure 17: A simple RNN, unrolled over time.....	41
Figure 18: Block diagram of the LSTM memory cell. The inputs for the input, input gate, forget gate and the output gate are the output of the previous cell $hi(t)$ and the observed state $xi(t)$ . The black square indicates a delay of a single time step [15] .....	42
Figure 19: Typical computer screen during training .....	54

Figure 20: The blue line indicates the average episode rewards achieved during the agent training, and the red depicts the average episode rewards of an agent acting randomly for one thousand (1K) episodes.....	54
Figure 21:Sequence of events when the agent faces an enemy .....	55
Figure 22:Agent recovers after the encounter with the enemy. ....	56
Figure 23: General orientation of the agent during exploration.....	56
Figure 24: The exploration problem faced by the agent .....	57

## TABLES

Table 1: Warrior level and corresponding attributes .....	16
Table 2: Wizard level and corresponding attributes .....	16
Table 3: Enemies of the Game .....	17
Table 4: Weapon total boost per Hero level.....	18
Table 5:The reward system used to train the agent.....	51

# 1. INTRODUCTION

## 1.1 Description of Rogue-like games

Even though the name of the genre came from the game Rogue, which was developed during the 1980 by Michael Toy and Glenn Wichman with the contribution of Ken Arnold, the first game of this kind is considered the Beneath Apple Manor developed by Don Worth for the Apple II and published by the Software Factory in 1978.

The rogue like games is a subgenre of the Role-Playing Games (RPG). Main characteristic of the RPG is that the player assumes the role of a character in a fictional setting. The rogue-like game narrative is inspired from tabletop role playing games such as Dungeons & Dragons. The main features that characterize a game as rogue-like are:

**Dungeon Crawl:** The hero of the game navigates thru a labyrinth environment (the environment can be caves, castles, forests etc.) confronting monsters, avoiding traps, solving puzzles, and looting treasures. The game ends with the hero finding a precious item (like the Amulet of Yendor in Rogue) or kill a specific enemy.

**Procedural Generated Levels:** The levels are created randomly through procedural generation.

**Permanent Death:** If the hero dies any progress in the game is lost. There isn't any save function.

**Turn based action:** The hero actions work as the game clock. The environment waits the hero to move and only then the world moves forward.

The Classic rogue-like games were based on text/console interfaces, where players, opponents, objects, walls, etc. are represented by letters/ASCII characters (graphics were introduced much later and at the beginning it was simply a direct "translation" of ASCII characters into sprites).

### 1.1.1 Description of the game used in the thesis

The game's name is "The ring of the Wizard Werdna". The game is written in python, and it is used as the game environment that the agent, on this thesis, tries to learn to play.

#### 1.1.1.1 Scope of the game

Main goal of the game is the hero to find the ring of the Wizard Werdna. To achieve this the hero searches a cave complex. Each cave is inhabited by hostile creatures ready to attack our hero. The ring is located at the tenth and final cave.

#### 1.1.1.2 Environment of the Game

The environment of the game consists of caves, each cave is a 2d space of tiles. The tiles occupy 12x12 pixels on the screen and can be floor type (where the hero and the enemies can walk), wall type or stair type. The stair type tile is the entrance for the next cave. The color for each tile is red for the floor, black for the wall and pink for the stair. Each cave is randomly created every time the hero enters in it. The procedure used is a simple (in implementation) algorithm for creating dungeons of a rogue-like game. The algorithm name is "random-walk", and it has been used in many rogue-like games. The way it works is the following, let assume that the cave consists of an  $N \times M$  grid of tiles and all the tiles are characterized as walls. Then a percentage of tiles that will be the floor is set, the algorithm starts from a random point and performs a random walk characterizing each grid it "steps" as floor until, the desired percentage is reached. An example of a cave depiction can be seen in Figure 1: A cave consist of 80x60 tiles with 46% Floor percentage.

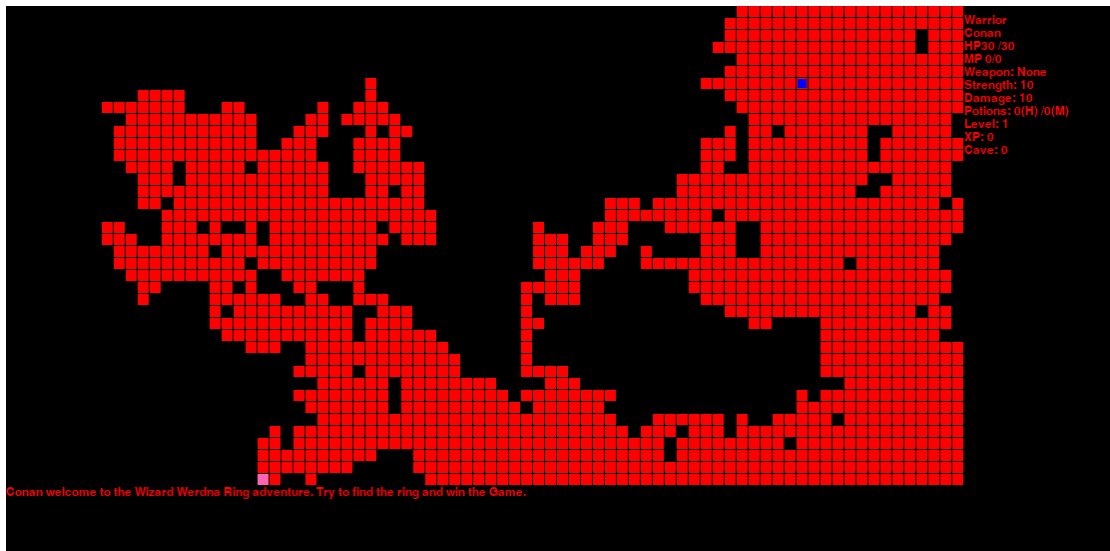


Figure 1: A cave consist of 80×60 tiles with 46% Floor percentage.

### 1.1.1.3 Hero of the game

For the hero of the game the player can choose between two types of characters warrior and wizard. The hero character is chosen at the first screen of the game by pressing 1 for the warrior type and 2 for the wizard. A screenshot of the introduction screen is given in Figure 2.

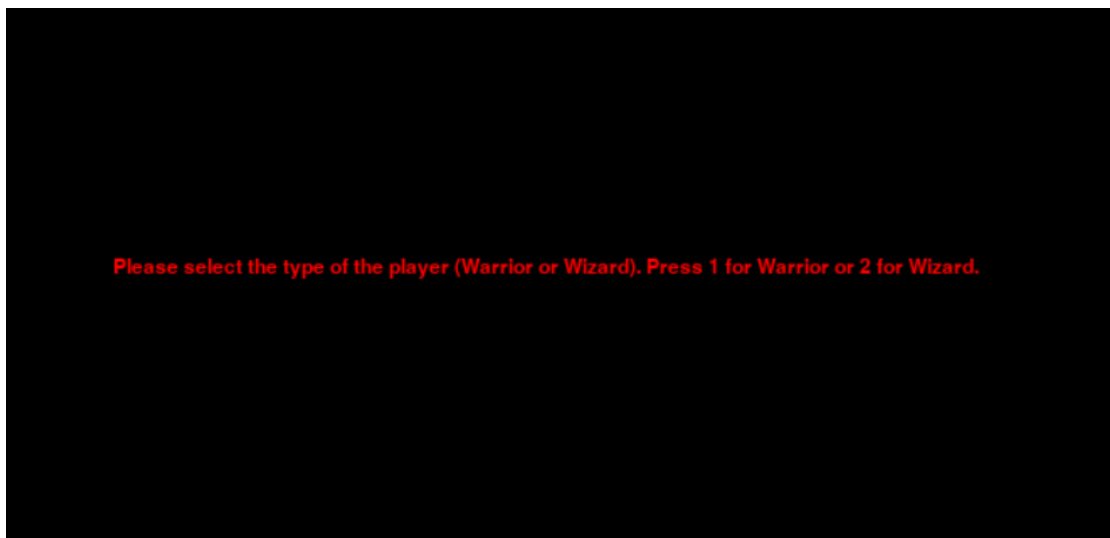


Figure 2: The ring of the Wizard Werdna, introduction screen

The player can choose their hero's name by typing the desired name on the second screen of the game and pressing enter. Figure 3 shows the second screen of the game.



Figure 3: The ring of the Wizard Werdna, name selection screen.

### 1.1.1.3.1 Hero Representation

The hero in the game is shown as a blue square, as can be, for example, seen in Figure 4.

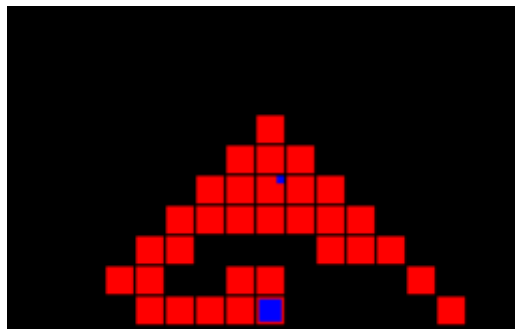
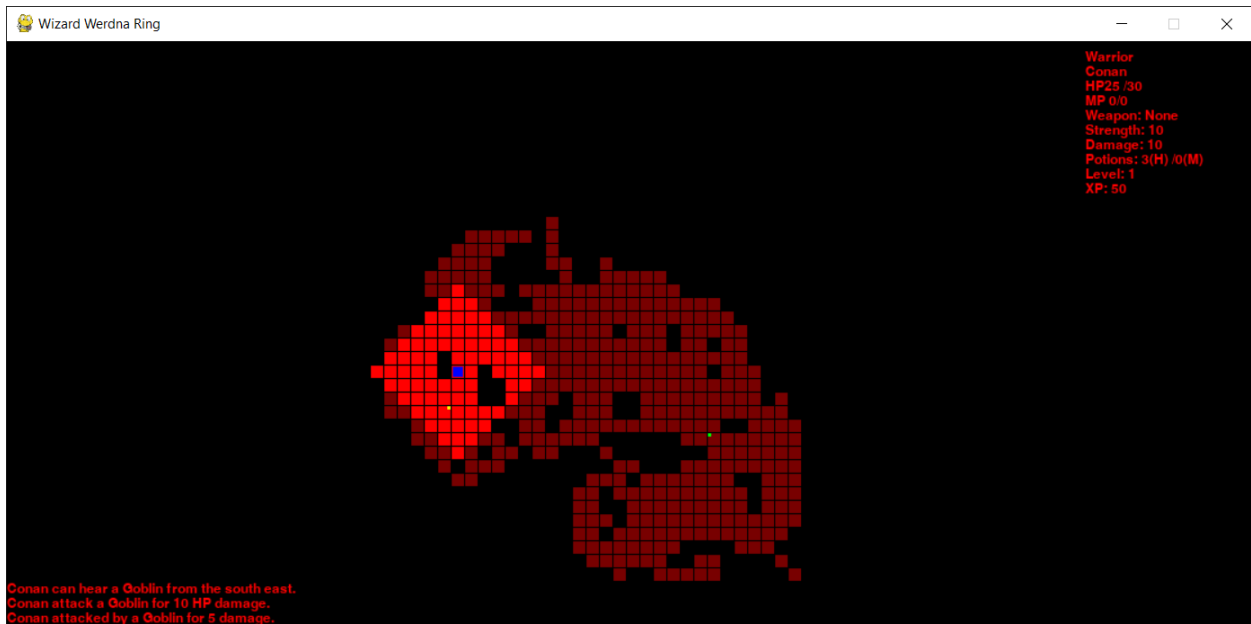


Figure 4:Hero Depiction

### 1.1.1.3.2 Hero Visibility

The hero can see all tiles, tile types, items and enemies that are within distance of 6 map tiles. For the tiles that are outside this range there are two visibility categories, the "UNKNOWN" which means that the hero has not visited the region yet and doesn't know anything for this tile and the "FOGGY" which means the hero has visited the tile and knows if it is a wall, stair or floor and if the tile has any item (the enemies are invisible in this case). Figure 5 depicts a typical game screen after the hero has explored part of the map.



**Figure 5: A depiction of the game after some movements of the hero is given above. The bright red for the tiles means that are inside the hero's visibility and the less opaque red means that the tiles visibility type is fogged.**

### 1.1.1.3.3 Hero Actions (Gameplay)

The type of actions the hero can perform are presented below. Keep in mind that only after the action of the hero the environment of the game reacts, in simple words first the hero performs an action and in response the environment moves.

**Hero basic movements:** The player can move the hero in the four directions using the keys **w** to move up, **a** to move left, **s** to move down and **d** to move right. For each movement the hero moves for one tile in the map/cave. Every time the hero moves there is a possibility that an enemy will be placed on the map. The probability comes from the formula  $p_0/e^{\max\_hero\_hp/hero\_hp}$  where  $p_0$  is between 0.1 and 0.25

**Attack:** Attack is performed by pressing the **SPACE** button. Each type of hero performs different type of attack. The Warrior uses sword as weapon and only attacks if the enemy is on the next tile, if there is more than one enemy nearby, the hero attacks the one with the least hp. The wizard uses the staff as weapon and use spells to attack. Because of this, the wizard attacks any visible enemy and if there is more than one enemy, attack priority is range, then enemy hp. Each time the wizard attacks 5 mana points is consumed. The damage done by the hero depends on his/her level and the weapon the hero possesses. The attribute that characterizes the damage the hero can do is strength for the warrior and intelligence for the wizard.

**Rest:** When the **r** key is pressed both hero types gain 4 HP. The wizard type hero also gains 4MP. When the hero is resting, there is a 25% chance that an enemy will be placed on the map. To increase the game difficulty, when the hero has seen the stair and the distance to the stairs is less than 35 tiles the rest function cannot be used.

**Health Potion Consumption:** The player can press the **h** button and the hero will consume one Health Potion and gain up to 20 HP.

**Mana potion Consumption:** The player can press the **m** button and the hero will consume one Mana Potion and gain up to 20 MP. The Mana potion can only be acquired by the wizard hero type.

**Changing weapons:** When the player moves to a tile that has a weapon, the hero can select that weapon by pressing the **p** key. The sword weapon that the warrior can acquire adds two boosts, one to the maximum HP the hero can have and one to the hero's strength. The staff is the weapon the wizard can acquire and adds three boosts, one each to max HP, max MP, and intelligence.

#### 1.1.1.3.4 Hero Level

The hero starts at level 1 and may reach up to level 5 (if he/her survives long enough). Each level determines the maximum HP for both hero types, maximum MP and intelligence for the wizard type, and strength for the warrior type. To level up, the hero collects XP by killing enemies. The following tables show the level of the hero and the corresponding XP required for each level and the attributes that the hero has at each level. Level also determines the enemies the hero will encounter and weapon bonuses.

**Table 1: Warrior level and corresponding attributes**

Level	XP	HP	Strength
1	0-299	30	10
2	300-899	60	20
3	900-2699	80	25
4	2700 - 6499	90	30
5	6500 - 13999	100	35

**Table 2: Wizard level and corresponding attributes**

Level	XP	HP	MP	Intelligence
1	0-299	20	30	10
2	300-899	40	50	20
3	900-2699	50	70	30
4	2700 - 6499	55	90	40
5	6500 - 13999	60	110	50

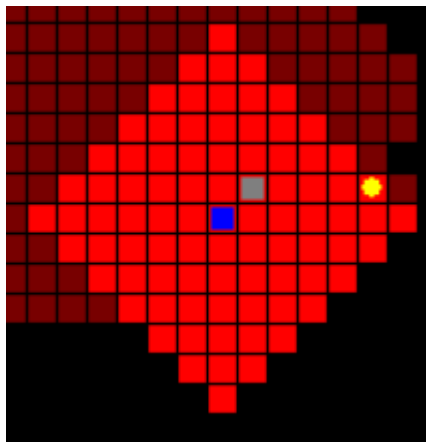
#### 1.1.1.4 Enemies of the game

Enemies in the game spawn during the hero's movement and when he/she is resting. The type of enemy depends on the level of the hero. Enemies have specific visibility, HP, strength, and XP. When killed they give their XP to the hero. They stand still until the hero is in their line of sight, then move towards the hero trying to close the distance and finally attack and reduce the hero's hit points by the amount of power they have. A table with the name of the enemy, the corresponding level the hero must have for that enemy to spawn, how it is shown in-game, and its characteristics are given below. Figure 6 gives an example of how enemies are depicted in the game.



**Table 3: Enemies of the Game**

Hero Level	Enemy name	HP	Strength	XP	Visibility	Depiction
1-2	Giant Rat	5	2	30	4	Gray Square
1-3	Goblin	15	5	50	7	Yellow Circle
3-4	Gray Slime	30	8	80	2	White Square
4-5	Orc Grunt	40	10	100	6	Green Square
3-5	Orc Warlord	50	12	120	7	Dark Green Square
4-5	Ettin	60	20	150	9	Dark Grey Circle
3-5	Skeleton	20	30	100	4	White Rectangle
5	Wyrn	80	20	200	5	Magenta Square
5	Vampire	50	30	400	10	Black Circle

**Figure 6: Enemy rendering example (a giant rat and a Goblin)**

### 1.1.1.5 Items of the game

In the game, tiles can have items stored on them, the two main item categories are potions and weapons. There is also a special item the ring of the Wizard Werdna, which will be detailed below.

#### 1.1.1.5.1 Potions

Potions are used to restore the hero's HP and MP. The health potion restores the hero's HP up to 20 HP. The mana potion is only used by a wizard type hero and restores the hero's MP up to 20 MP. Potions are acquired by the hero automatically when he/she passes a tile on which they are stored. The maximum number of potions a hero can have is 30 potions.

#### 1.1.1.5.2 Weapons

There are two types of weapons in the game, the sword and the staff. The sword can be wielded by a warrior and the staff by a wizard type of hero. Each weapon enhances specific characteristics of the Hero. The weapon and the total boost it provides depends on the hero's level when the weapon is created. The total boost is split between the attributes of the weapon randomly. The table below gives the hero level and the corresponding total boost.

Table 4: Weapon total boost per Hero level

Hero Level	Total Boost
1	10
2	20
3	30
4	40
5	60

##### 1.1.1.5.2.1 Sword

The sword provides a boost to the maximum HP a warrior can have and boosts the hero's strength and thus the damage he/she can deal.

##### 1.1.1.5.2.2 Staff

The staff boosts the maximum HP and MP a wizard can have and boosts the hero's intelligence and thus the damage she/he can deal.

##### 1.1.1.5.2.3 The ring of the Wizard Werdna

This is a special item. It is the item for which our hero begins his adventure. When the item is found and picked by the hero the game ends. The depiction of the ring is the same with the stairs (a pink tile).

### 1.1.1.5.3 Depiction of objects

Items are shown as a small square at the corner of the tile they are stored on. The potion types are shown in the upper right corner of the tile and the color is blue for the health potion and yellow for the mana potion. Weapons are depicted in the upper left corner with one color (yellow) for both. An illustration of the game items is given in Figure 7.

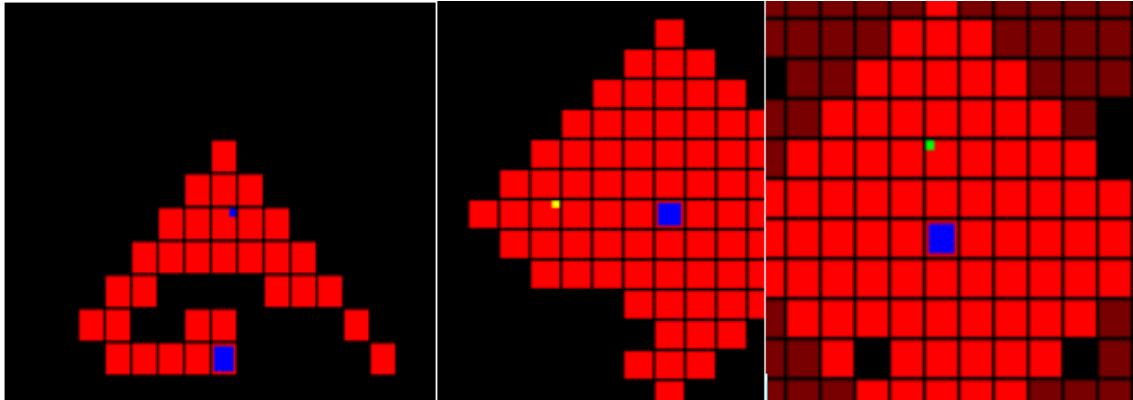


Figure 7: Health potion illustration (left), Mana potion illustration (center) weapon illustration (right)

### 1.1.1.6 Hero Status

Hero status is displayed in the upper right corner of the game screen outside of the map display. The information presented is the hero's type, name, HP, MP, his/her weapon name, warrior strength or wizard intelligence, damage the hero can deal (hero's strength or intelligence plus the boost from the weapon held), the potions the hero has, the hero's level and finally his/her current experience. Figure 8 illustrates an example of the hero status.

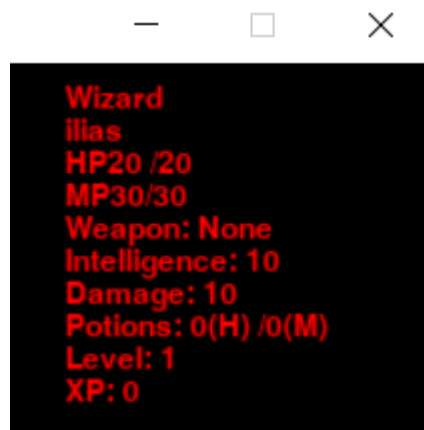
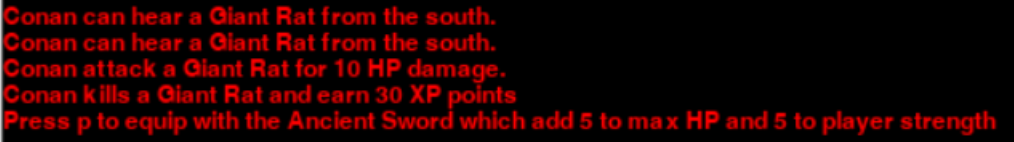


Figure 8: Hero Status

### 1.1.1.7 Game log

The game log is displayed in the lower left corner of the screen below the map screen and prints the last 5 game messages. The messages are related to various aspects of the game, such as the spawn location of an enemy relative to the hero's location, HP/MP restored after using a potion, the name of the weapon and its boosts, etc. Figure 9 illustrates an example of the game log.



```

Conan can hear a Giant Rat from the south.
Conan can hear a Giant Rat from the south.
Conan attack a Giant Rat for 10 HP damage.
Conan kills a Giant Rat and earn 30 XP points
Press p to equip with the Ancient Sword which add 5 to max HP and 5 to player strength

```

Figure 9: An example of the game log

## 1.2 Introduction to Artificial Intelligence with focus on Reinforcement Learning.

The term AI refers to the ability of a computer or computer-controlled robot to perform tasks that usually are done by humans or other intelligent beings. The field through which a computer learns how to perform tasks that require intelligence without human assistance is called Machine Learning (ML). There are many examples of ML algorithms that are very successful in performing competitive tasks (and sometimes surpassing human performance), such text classification, word prediction, voice and face recognition, play Atari games at human-level [1] and even beat human champions at the game of Go [2].

### 1.2.1 Machine Learning categories

ML is categorized in four main categories Supervised Learning, Unsupervised learning, Semi-Supervised learning, and Reinforcement Learning [3].

#### 1.2.1.1 Supervised Learning

Supervised Learning is used mostly on classification problems like text classification, voice and face recognition etc. A simple example will be used for the understanding of this category. Suppose our goal is to build a model that will take a record of three values, petal length, petal width, and color, and classify that record into a flower class. The model training will be based on records that have already been labeled as a specific flower. The records will be divided on training and test data. The training data will then be used to train the model. The training process starts with the model taking as input the values of a record and making a prediction about the flower class, if the prediction is different from the label the record has, the model will be modified, and the process repeated until the model predicts the classes of the training data records with high accuracy. At the end the test data will be used to evaluate the model. Using data that has already been labeled by humans is why this class of ML gets the name supervised.

#### 1.2.1.2 Unsupervised Learning

In Unsupervised Learning the data is unlabeled. The goal of this category of algorithm is to train a model to take, as input, a record with  $x$  number of values and output either a record with a different number of values or a single value that can be used to solve a specific problem. For example, in clustering the model takes as input the values of a record and returns the identity of the cluster to which the records belong, the cluster categories are created during the training based on the similarities of the records and not by human intervention. One application of clustering is in sampling, where it can help the researcher have a smaller and more efficient sample. We can use unsupervised learning to automatically divide the data into different groups and intelligently sample from these groups, avoiding large sample from the general population and thus entries with similar values and biased sample.

Another area where unsupervised learning can be used is dimensionality reduction of the features of elements, this technique can be used to make data easier to present, specifically if the researcher has data with records of more than three values, in which

case the values cannot be graphically presented, unsupervised learning can be used to reduce dimensionality and make graphs feasible.

Outlier detection is another area where unsupervised learning can be used. This technique can be applied to network intrusion problem, helping to detect abnormal packets, and the discovery of unique records, like the discovery of a document that is different from a collection of documents. The way Unsupervised Learning helps in these fields is by building models that can output a value that says how different a record is from the set of data.

### 1.2.1.3 Reinforcement Learning

This category of algorithms is based on an interaction between an agent and an environment. Usually, and as is in our case, the agent is software running on a computer, observes the state of the environment, and takes actions, the actions cause changes in the environment and bring rewards for the agent. The scope of the algorithm is to develop an optimal policy, a function that will take the environmental states and output actions that will yield the maximum reward in the long run.

This class of algorithms is the one that will be used for this thesis, so further analysis will be presented in the next paragraphs.

### 1.2.1.4 Semi-Supervised learning

This category of algorithms is based on a training of a model with the usage of a small set of labeled data and a much larger set of unlabeled data. The objective of the technique remains the same as supervised learning, but unlabeled data is used to improve model performance.

## 1.2.2 Reinforcement Learning

A key difference of RL with the other ML techniques is that RL is more goal directed than them. The other techniques are most often used to solve subproblems without considering how these efforts will help and fit into the solution of a larger problem. RL is based on interacting with an uncertain environment and achieving a final goal.

### 1.2.2.1 RL Characteristics

The key characteristics of the RL are an agent, an environment, a policy, a reward, a value function, and a model of the environment (the latter is optional). Definition of these characteristics is given below:

**The agent** as already mentioned is the AI model we are trying to train.

**Environment** is the space in which the agent moves. In general, as environment we characterize everything outside the agent. The agent cannot control the environment, although the agent interacts with it and cause changes to it, in simple words the agent observes the environment state and takes actions that cause changes in the environment and yield rewards for the agent. The mathematical framework of this interaction is based on the Markov Decision Process (MDP).

**Policy** is the function that will map the state of the environment detected by the agent to the appropriate action to maximize rewards in the long run.

**Reward** is the only value on which training is based. The agent's goal is to maximize the reward in the long run. Thus, the reward drives the training of the agent. For example, if an agent observes a state and takes an action that yields a poor reward it may cause the agent to choose differently the next time it faces the same state.

**The value function** determines the value of a state. By this we mean the rewards that the agent will yield starting from this state and follow a specific policy. Taking this into account, the value function affects the agent's long-term effort more than the reward, and for this reason we are more interested in the value function.

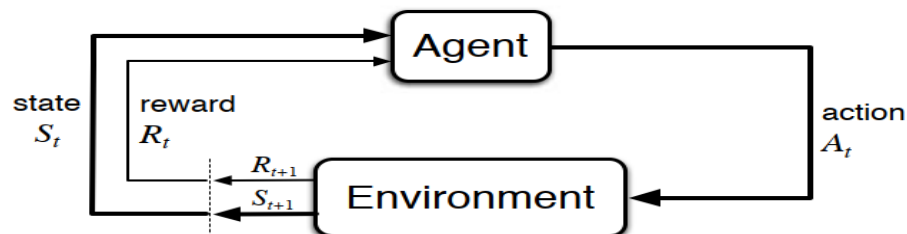
**The environment model** is used in some cases of RL. This characteristic is used when we have complete knowledge of the environment or when we want to simulate the environment. For the first case, an example of a task with a known environment is the game of chess, all states and actions are known, and a model of the environment can be used to plan the agent's next actions. For the second case the model is used to simulate the environment behavior, usually this is very helpful when the trial-and-error approach can have disastrous results or is economical inefficient, consider the training of an agent learning to land a spacecraft on the moon. On the other hand, there are cases where the agent simply learns through actual experience with the environment. Thus, we end up with two main categories of RL algorithms, model-based and model-free.

There are two more basic characteristics the exploration and the exploitation which will be understood easier after the following paragraph. Definition of the exploration and exploitation will be given in paragraph 1.2.2.3.1.

### 1.2.2.2 Basic Mathematical formulation of RL

As we already know the RL concept is based on finite MDPs. The basic mathematical formulation as described here applies to model-based RL algorithms, although with some changes the same framework is used when the exact model of the environment is not known, details for these cases will be discussed in the following paragraphs. A well-known algorithm class which is model-based RL is Dynamic Programming DP and is the class that will be used for the explanation of how a policy is evaluated and improved. These types of algorithms assume a perfect model of the environment and are computationally expensive.

Figure 10 depicts the agent-environment interaction in a Markov decision process.



**Figure 10: The agent–environment interaction in a Markov decision process. [4]**

The agent and the environment interact at discrete time steps. Specifically, the agent observes at each time step a state of the environment denoted by  $S_t$ , then performs an action  $A_t$  and a  $R_{t+1}$  reward is returned. Due to the agent's action the environment transitions to a new state  $S_{t+1}$  and the process continues creating a sequence of states, actions and rewards.

In the case of the finite MDP the transition to the  $R_{t+1}$  and the new state  $S_{t+1}$  is decided by a well-defined discrete probability distribution, which is affected only by the preceding state  $S_t$  and action  $A_t$ . The probability is a four-value function

$$p: (S_{t+1}R_{t+1} | S_t A_t) = [0,1] \quad (1.1)$$

which for each  $S_t$  and  $A_t$  couple holds

$$\sum_{S_{t+1}} \sum_{R_{t+1}} p(S_{t+1} R_{t+1} | S_t A_t) = 1 \quad (1.2)$$

### 1.2.2.2.1 Expected return

Note that the agent each time performs the action  $A_t$  seeking to maximize the sequence of rewards. The term is called expected return and the simplest form is defined as

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4} \dots R_T \quad (1.3)$$

Where  $T$  is the last time step. As a final time step, we can define the end of an episode if the task the agent is trying to solve can be separated into discrete episodes, or infinity if the task continues for ever. An example of a task that has episodes is a video game, the episode ends when the player loses or wins the game, and an example of a task that never ends is operating a robot.

For tasks that continue for many steps or take forever, it is obvious that the expected return will deviate to large or infinite values for each state of the environment, making it impossible for the agent to choose the action that will yield the most rewards in the long run. For this reason, the concept of discounted return is introduced. Discounted return is defined as

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \dots \gamma^{T-1} R_T = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (1.4)$$

and parameter  $\gamma$  is a value between 0 and 1 and is called discount rate. The discounted reward converges, even for infinite  $T$ , to a finite value if the reward sequence is bounded.

The relationship between successive discounted rewards is used very often in RL algorithms, so a definition will be given:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} \dots \gamma^{T-1} R_T \\ G_t &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} \dots \gamma^{T-2} R_T) \\ G_t &= R_{t+1} + \gamma G_{t+1} \end{aligned} \quad (1.5)$$

### 1.2.2.2.2 Bellman Equation (value and action-value function)

The scope of RL algorithms is to define a policy that will derive the maximum rewards in the long run. A policy is the function that maps the states of an environment into probabilities for the agent to choose each possible action. For example, if an agent follows the policy  $\pi$  at specific time  $t$  the probability of observing the state  $s$  and take the action  $a$  is define as  $\pi(a|s)$ .

For a state  $s$  the value function under policy  $\pi$  is the expected return if the agent starts from state  $s$  and follows  $\pi$  thereafter. The value function for policy  $\pi$  is denoted as  $u_\pi$  and for MDPs is defined as

$$u_\pi(s) = E_\pi[G_t | S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s] \text{ for all } s \quad (1.6)$$

The  $E_\pi[\cdot]$  denotes the expected value of random variable, given that the agent follows a policy  $\pi$ , and  $t$  denotes any time step.  $u_\pi(s)$  is called the value function for policy  $\pi$ .

In the same way the action-value function is the expected return if the agent starts from state  $s$  performs the action  $a$  and then follows the policy  $\pi$ . It is denoted as  $q_\pi(s, a)$  and defined as

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a] \quad (1.7)$$

$q_\pi(s, a)$  is called the action-value function for policy  $\pi$ .

$u_\pi$  or the  $q_\pi$  for a particular policy can be estimated using experience or a model of the environment.

Like the case of the discounted rewards the relationship of successive states and states action pairs on  $u_\pi$  and  $q_\pi$  are very important. The equation which expresses this relationship for the  $u_\pi$  and the  $q_\pi$  is called Bellman and is named after Richard E. Bellman [5]. The expression of the equation for the  $u_\pi$  is:

$$\begin{aligned}
 u_\pi(s) &= E_\pi[G_t | S_t = s] \\
 &= E_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\
 &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma E_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s']] \\
 &= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) [r + \gamma u_\pi(s')], \text{ for all } s
 \end{aligned} \tag{1.8}$$

The  $s'$  is the state that follows after the agent observes state  $s$  and takes the action  $a$ . Similarly, the bellman equation for the action-value function  $q_\pi$  is

$$q_\pi(s, a) = \sum_{s', r} p(s', r | s, a) \left[ r + \gamma \sum_{a'} \pi(a', s') q_\pi(s', a') \right] \tag{1.9}$$

In this case  $s'$  again is the state that follows the agent observes state  $s$  and takes the action  $a$  and  $a'$  is the action the agent takes for state  $s'$ .

### 1.2.2.2.3 Optimization and policy value functions

Finding the policy that yields the maximum rewards in the long run remains the main application area of an RL algorithm. This policy is called optimal. There may be more than one optimal policy. All policies have the same state value function, which is denoted as  $u_*$  and expressed in the following form  $u_*(s) = \max_\pi u_\pi(s)$  for all  $s$ . The optimal policies also share the same action-value function which is denoted as  $q_*$  and is expressed in the form  $q_*(s, a) = \max_\pi q_\pi(s, a)$  for all  $s$  and  $a$ . Notice that the  $q_*$  can be written in terms of the  $u_*$  as follows

$$q_*(s, a) = E[R_{t+1} + \gamma u_*(S_{t+1}) | S_t = s, A_t = a] \tag{1.10}$$

Again, it is very useful to define the equation defining the relationship between successive states or state-action pairs and for the optimal policies. The equation which expresses this relationship is called the Bellman optimality equation and, for the  $u_*$ , is defined as

$$u_*(s) = \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma u_*(s')] \tag{1.11}$$

For the  $q_*$  the Bellman optimality equation is

$$\begin{aligned}
 q_*(s, a) &= E \left[ R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a' | S_t = s, A_t = a) \right] = \\
 &\quad \sum_{s', r} p(s', r | s, a) \left[ r + \gamma \max_{a'} q_*(s', a') \right]
 \end{aligned} \tag{1.12}$$



### 1.2.2.2.4 Policy improvement through iteration

In this subparagraph DP will be the referred class of algorithms. The main object will be to understand how the Bellman equation (see paragraph 1.2.2.2.2) and the Bellman optimality equation (see paragraph 1.2.2.2.3) are used in evaluating and improving the policy  $\pi$  followed by the agent.

#### 1.2.2.2.4.1 Policy Evaluation

Firstly, the evaluation of a policy  $\pi$  is considered. The value function will be used as example. The procedure starts by setting an arbitrary value for the value function for all states  $s$ , which is denoted as  $u_0(s)$ . Then the  $u_1(s)$  is updated with the following equation  $u_1(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma u_0(s')]$  for all  $s$  where  $\pi(a|s)$  come from the policy  $\pi$  under evaluation, the procedure is repeated  $k$  times and if the  $k \rightarrow \infty$  then the  $u_k = u_\pi$  applies. The algorithm is described above is called iterative policy evaluation and the formulation is

$$u_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma u_k(s')], \text{ for all } s \quad (1.13)$$

Similarly, the formulation of the iterative policy evaluation algorithm for the action-value function is

$$q_{k+1}(s, a) = \sum_{s',r} p(s',r|s,a) [r + \gamma \sum_{a'} \pi(a',s') q_k(s', a')] \quad (1.14)$$

#### 1.2.2.2.4.2 Policy Improvement

Evaluation of a policy is usually used to improve that policy. For example, consider a deterministic policy  $\pi$  for which the value function  $u_\pi(s)$  has been evaluated and we want to test whether there is a better policy. For this reason, we choose for a state  $s$  an action  $a$  outside this policy  $a \neq \pi(s)$ , then calculate the  $q_\pi(s, a)$ , which is the expected return starting from state  $s$ , choosing the action  $a \neq \pi(s)$  and then following the policy  $\pi$ . If the  $q_\pi(s, a)$  is greater than the  $u_\pi(s)$  then we can conclude that we have a new policy  $\pi'$  which is better than the old one  $\pi$ . This is considered a case of the policy improvement theorem [4], which states that for two deterministic policies  $\pi$  and  $\pi'$ , if  $q_\pi(s, \pi'(s)) \geq u_\pi(s)$  holds for all states  $s$  then  $\pi'$  is assumed to be at least equal with  $\pi$  and for the expected return holds  $u_{\pi'}(s) \geq u_\pi(s)$  for all  $s$ .

Generalizing for all the states  $s$  and performing the same behavior as described above we have the greedy policy,  $\pi'$ , which select for each state the action which seems better according to  $q_\pi(s, a)$ . The definition of the greedy policy is

$$\pi'(s) = \underset{a}{\operatorname{argmax}} q_\pi(s, a) = \underset{a}{\operatorname{argmax}} \sum_{s',r} p(s',r|s,a) [r + \gamma u_\pi(s')] \quad (1.15)$$

$\operatorname{Argmax}_a$  denotes the action  $a$  which maximize the expression that follows. This procedure of creating a new improved policy based on greedy selection with respect of the original value function is called policy improvement.

If the policy,  $\pi'$ , is not better than the original one then  $u_{\pi'} = u_\pi$  and for all  $s$  holds  $u_{\pi'}(s) = \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma u_{\pi'}(s')]$ , which is the bellman optimality equation (see paragraph 1.2.2.2.3) and we can tell that both policies are optimal, concluding that policy improvement must give us a better policy except when the original policy is already optimal.

Our analysis, so far, presupposes that deterministic policies are followed, although the same holds and for stochastic policies. If, for example, there is more than one action that yields the maximum expected return, we do not choose one action, but we assign the probability to each of these actions to be selected in the new greedy policy. We distribute the probability in any way with the condition that non-optimal actions are given zero probability.

#### 1.2.2.2.4.3 Policy Iteration

The combination of the policy evaluation and policy improvement is called policy iteration. The procedure is simple, firstly we evaluate a policy,  $\pi_0$ , using the value function  $u_{\pi_0}$  and based on that value function we create the new improved policy  $\pi_1$ . This procedure continues until we reach the optimal policy  $\pi_*$ , in the case of finite MDP where the number of policies is finite the procedure always converges to the optimal policy and optimal value function in a finite number of iterations.

#### 1.2.2.2.4.4 Policy Iteration restriction and Value Iteration.

The main limitation of policy iteration as described above is the computation required to evaluate the policy. Each policy evaluation requires many iterations over all environments states to converge to the value function of the policy being evaluated. This requirement raises the question of whether it is necessary to wait until the policy evaluation converges to achieve the policy improvement or is it sufficient to do a policy evaluation to some extent and do the policy improvement after this truncated policy evaluation.

The answer on the previous question is that in many cases the optimal policy can be found with the truncated policy evaluation. For this reason, several methods of truncating the policy evaluation have been developed, one method worth mention is the value iteration, which truncates the policy evaluation right after one sweep of the environment states. It is easy to write it as a combination of policy improvement and truncated policy evaluation steps

$$u_{k+1}(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma u_k(s')] \text{ for all } s \quad (1.16)$$

#### 1.2.2.2.4.5 Generalized Policy Iteration

The interaction of policy evaluation and policy improvement is described by the term Generalized Policy Iteration (GPI). Most of the RL algorithms are labeled as GPI. This means they have policies and value functions that interact, i.e., the policy improves based on the value function and the value function changes trying to approximate the policy value function. When the process stops producing changes in the policy and value function, optimization has been achieved. In this case the policy is greedy in its value function, so the Bellman optimality equation ( $u_*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma u_*(s')]$  for all  $s$ ) holds, and policy and value functions are optimal. Figure 11 shows a visualization of the process.

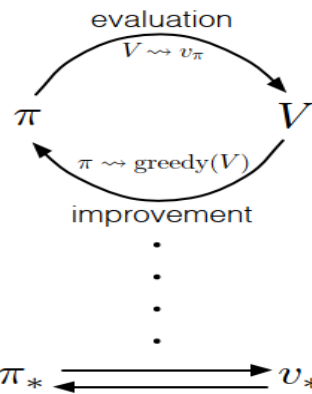


Figure 11: GPI procedure [4]

The two processes compete and cooperate simultaneously. They compete during the procedure, policy improvement makes the value function imprecise for the changed policy, and policy evaluation makes the policy non-greedy anymore, and they cooperate in the long run when the interaction leads to a common solution, which is the optimal value function and the optimal policy. Figure 12 shows how the two processes although have different goals, one is to evaluate the policy and the other is to improve the policy, in the end they achieve the main goal of optimizing the value function and the policy function.

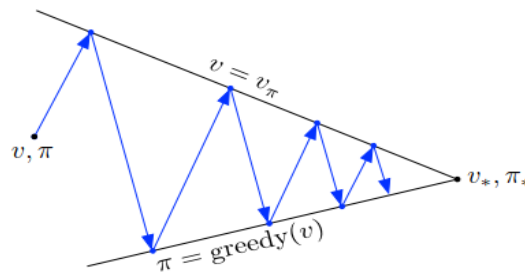


Figure 12: Optimization of the value and policy function through iteration of policy evaluation and policy improvement.[4]

### 1.2.2.3 Basic categories of RL Algorithms

In this paragraph we will present a brief description of the main classes of RL algorithms, including some basic examples. The main difference with the procedures presented in the previous paragraph, the DP algorithms, is that for the RL algorithms presented here, either we do not have a complete knowledge of the environment and cannot evaluate and improve the policy by sweeping through all the states, or even and if we have full knowledge of the environment, it is computationally expensive to implement the DP algorithms. Instead, we rely on the experience accumulated by the agent's interaction with the environment. Because of the way the RL algorithms presented here work; we need to introduce two new characteristics which are added to the characteristics presented in paragraph 1.2.2.1. The characteristics are the exploration and the exploitation of a policy. The way each RL algorithm tries to solve the problem further categorize them as on-policy and off-policy. In the following paragraphs, 1.2.2.3.1 and 1.2.2.3.2, a definition of the two characteristics and the two categories of algorithms will be given.

#### 1.2.2.3.1 Exploration and Exploitation

As already described, RL algorithms without complete knowledge of the environment are trained by the experience they accumulate from interacting with the environment. This

behavior can cause serious problems on the training procedure. Imagine an agent following a policy  $\pi$ , and for a given state  $s$  the policy gives zero probability of choosing an action  $a$ , the agent will never choose this action and will never discover the rewards that this action brings, for this reason the agent should try as many actions as possible in as many situations as possible. Taking random action outside the policy the agent follows is called exploration. On the other hand, when the currently considered best policy is used to select the action to be evaluated, the process is called exploitation. Balancing these two processes is very important. Too much exploitation could lead to the wrong policy, which is equivalent to quickly reaching a local optimum, and in the other hand too much exploration could lead to slow learning.

There are many ways to balance exploration and exploitation for a given policy, the two most common being the  $\epsilon$ -greedy method and using policy gradient RL algorithms.

#### **1.2.2.3.1.1 $\epsilon$ -greedy method**

In this method the algorithm performs random actions on a certain percentage of the actions performed. The way it works is simple, first a number less than 1 is defined and takes the role of the number  $\epsilon$ , then the agent chooses action based on the current policy with probability  $1 - \epsilon$  and random action with probability  $\epsilon$ . After that the action is evaluated and the policy is changed. It is very common to use a variable  $\epsilon$ , which starts with a value close to 1, when the agent starts training and the knowledge of the environment is insufficient, and gradually decreases to a minimum value.

#### **1.2.2.3.1.2 Policy Gradient Algorithms**

In this case the exploration of the environment comes from the way the agent chooses the action to perform. So far, we saw that the agent observes the environment and depending on the state of the environment chooses the action that will yield the best rewards in the long run, with the policy gradient the agent chooses actions based on the preference for those actions. More specifically, the agent observes the state and then calculates the probability for each action, the agent chooses the action to perform based on these probabilities. The action yields the reward, then the reward is used to change the probability-preference for each action, in simple words the agent's policy, for the next time the agent encounters the same situation. Exploration takes place because each time all actions have some probability of being selected.

#### **1.2.2.3.2 On-policy and Off-policy algorithms**

The scope of RL algorithms is to define an optimal policy, see paragraph 1.2.2.2.3, if the same policy, which the agent tries to optimize, is used to gather experience from the environment, then the algorithm belongs to the category of on-policy algorithms. In this case, the policy must maintain some degree of exploration during the agent training. If the algorithm uses one policy to gather experience and tries to optimize a different one, then the algorithm is marked as off-policy. In this case, exploration should only be performed by the policy that collects experience.

#### **1.2.2.3.3 Monte Carlo algorithms**

In this class of RL algorithms, policy evaluation and improvement are based on experience gathered during an episode. The concept of GPI, as described in paragraph 1.2.2.2.4.5, is used by Monte Carlo (MC) algorithms. The evaluation process is, however, different: instead of using a model to calculate the value of a state, we use an average of many returns, collected starting from that state. Because the state value is the expected

return, this average can approximate the value of the state successfully. All that is required for correct convergence is that all states are visited infinite times. For the policy improvement we focus on the approximation of the action-value function, the reason for doing so is that this function can be used for policy improvement without having a model of the dynamic transition of the environment. MC algorithms perform the GPI on an episode-by-episode basis.

There are many differences between DP and MC algorithms. Firstly, MC algorithms learn with interaction with the environment, simulated or not, and they do not require precise model of the environment's dynamics. Because of this, MC algorithms can focus on a small subset of states that are most likely to be encountered and do not need to evaluate all states. Another important difference, which is a unique feature of the MC algorithm, is that they do not bootstrap, meaning that they update the value estimate of a state based on the actual returns that follow rather than the value estimate of a successive state.

Balancing exploration and exploitation, see par 1.2.2.3.1, is very important for any RL algorithm, in the case of MC a commonly used concept is the exploring start. Exploring presupposes that the algorithm starts from state–action pairs that are randomly selected to cover all possibilities. It is worth to mention that exploring starts works in simulated environment, where the initial state of the episode can be controlled.

#### 1.2.2.3.4 Temporal Difference (TD)

This class of RL algorithms combines features from MC and DP algorithms, gaining advantages from both. Like MC, TD relies on interacting with the environment to gather experience, so no model of the environment's dynamics is required. In the other hand TD bootstraps like the DP, it updates its estimates using estimates of successive states and does not need to wait for a final result. Like the other two methods TD relies on the concept of GPI, see paragraph 1.2.2.2.4.5, to obtain an optimal policy. Again, the way in which experience is collected gives rise to the problem of balancing exploration and exploitation, see paragraph 1.2.2.3.1, and how each algorithm deals with this problem further categorizes the algorithm as on-policy or off-policy, see paragraph 1.2.2.3.2.

The simplest implementation of a TD algorithm for estimating the value function will be used to better understand how TD algorithms work, followed by a description of basic TD algorithms.

##### 1.2.2.3.4.1 Estimation of the value function $u_\pi$ using one-step TD algorithm

Suppose we collect experience following a policy  $\pi$  and want to update the estimate  $V$  of the value function  $u_\pi$  for an arbitrary state  $S_t$ , to do this we use the equation

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \quad (1.17)$$

where the  $S_{t+1}$  is the next state and  $R_{t+1}$  is the received reward. The target for the update is the term  $R_{t+1} + \gamma V(S_{t+1})$ , the difference between the target and the initial estimate  $V(S_t)$  gives the name TD. The  $\alpha$  is the step size parameter which defines the percentage of TD to be used to update the initial estimate  $V(S_t)$ . A pseudocode for the procedure described above is following:

##### **One-step TD algorithm for estimating $u_\pi$ [4]**

Input: the policy  $\pi$  to be evaluated

Algorithm parameter: step size  $a \in (0,1]$

Initialize  $V(s)$ , for all  $s \in S^+$ , arbitrarily except that  $V_{(terminal)} = 0$

**Loop** for each episode:

Initialize  $S$

**Loop** for each step of episode:

$A \leftarrow$  action given by  $\pi$  for  $S$

Take action  $A$ , observe  $R, S'$

$V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$

$S \leftarrow S'$

**Until**  $S$  is terminal

#### 1.2.2.3.4.2 Basic TD algorithms

The main TD algorithms classes are SARSA, Q-learning, Expected SARSA, Double Q-learning and Actor-Critic. In the following paragraphs a brief description of the algorithms is given.

#### 1.2.2.3.4.3 SARSA

SARSA is an on-policy, see paragraph 1.2.2.3.2, TD algorithm. This algorithm uses the concept of the GPI, see paragraph 1.2.2.2.4.5, to reach to an optimal policy. The difference from the methods presented already (MC and DP), is that the evaluation of the prediction is performed by the TD method.

In the case of the SARSA algorithm we use TD for the action-value function in the same way as we use it for the value function, see paragraph 1.2.2.3.4.1, the main equation is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (1.18)$$

The use of the quintuple of state, action, reward, next state, and next action gives the name SARSA to the algorithm.

A SARSA pseudocode algorithm, which uses  $\epsilon$ -greedy (see paragraph 1.2.2.3.1.1) policy to balance exploration and exploitation, is given below.

#### **SARSA (on-policy TD control) for estimating $Q \approx q_*$ [4]**

Algorithm parameter: step size  $a \in (0,1]$ , small  $\epsilon > 0$

Initialize  $Q_{(s,a)}$ , for all  $s \in S^+$ ,  $a \in A(s)$ , arbitrarily except that  $Q(\text{terminal}, \cdot) = 0$

**Loop** for each episode:

Initialize  $S$

Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

**Loop** for each step of episode:

Take action  $A$ , observe  $R, S'$

Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$

$S \leftarrow S'; A \leftarrow A';$

**Until**  $S$  is terminal

### 1.2.2.3.4.3.1 Q-Learning

Q-learning is an off-policy, see paragraph 1.2.2.3.2, TD algorithm. The algorithm was introduced by Watkins [6] and the equation which defines is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \quad (1.19)$$

The main difference is that this algorithm directly approximates the optimal action-value function  $q_*$ , the term  $\max_a Q(S_{t+1}, a)$  instead the  $Q(S_{t+1}, A_{t+1})$ , regardless of the policy followed. This approach is simpler and converge faster. Like all other algorithms based on real experience with the environment, all that is required for proper convergence is that all pairs of actions states keep updating.

A Q-Learning pseudocode algorithm, which uses  $\epsilon$ -greedy (see paragraph 1.2.2.3.1.1) policy to balance exploration and exploitation, is given below.

#### Q-Learning (off-policy TD control) for estimating $\pi \approx \pi_*$ [4]

Algorithm parameter: step size  $a \in (0,1]$ , small  $\epsilon > 0$

Initialize  $Q_{(s,a)}$ , for all  $s \in S^+$ ,  $a \in A(s)$ , arbitrarily except that  $Q(\text{terminal}, \cdot) = 0$

**Loop** for each episode:

    Initialize  $S$

**Loop** for each step of episode:

        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)

        Take action  $A$ , observe  $R, S'$

$Q(S, A) \leftarrow Q(S, A) + \alpha \left[ R + \gamma \max_a Q(S', a) - Q(S, A) \right]$

$S \leftarrow S'$

**Until**  $S$  is terminal

### 1.2.2.3.4.3.2 Expected SARSA

This algorithm results if we change the maximum over next state–action pairs used in Q-learning with the expected value, and the main equation is

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \sum_a \pi(a, |S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right] \quad (1.20)$$

Expected SARSA is more computationally complex than SARSA, although it eliminates the variance due to the random choice of  $A_{t+1}$ . In general, it performs better than SARSA and can be used as on-policy or off-policy algorithm.

### 1.2.2.3.4.3.3 Maximization bias and double Q-Learning

The Q-Learning algorithm is prone to the maximization bias. A simple example to understand the maximization bias is the following case, for a single state  $s$  there are many actions  $a$  with true values  $q(s, a)$  equal to zero but their estimated values  $Q(s, a)$  are distributed around zero. The maximum of the estimated values is a positive number although the true maximum is zero. This error is called maximization bias.

One algorithm which effectively deal with the maximization bias is the off-policy Double Q-Learning. In this algorithm two independent estimations for the true action-value  $q(a)$  function is used, let's say  $Q_1(a)$  and  $Q_2(a)$ . The one estimate, assume  $Q_1$ , is used to

determine the  $A = \underset{a}{\operatorname{argmax}} Q_1(a)$  and the other,  $Q_2$ , to provide the estimation for its value  $Q_2(A) = Q_2(\underset{a}{\operatorname{argmax}} Q_1(a))$ . Due to the equality  $E[Q_2(A)] = q(A)$  the estimation will be unbiased. To make the two estimates independent, in Double Q-Learning the time steps, during training, are split in half, for example with probability 0.5 for each step. Half steps are using the equation

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha [R_{t+1} + \gamma Q_2(S_{t+1}, \underset{a}{\operatorname{argmax}} Q_1(S_{t+1}, a)) - Q_1(S_t, A_t)] \quad (1.21)$$

and the other half the equation

$$Q_2(S_t, A_t) \leftarrow Q_2(S_t, A_t) + \alpha [R_{t+1} + \gamma Q_1(S_{t+1}, \underset{a}{\operatorname{argmax}} Q_2(S_{t+1}, a)) - Q_2(S_t, A_t)] \quad (1.22)$$

#### **Double Q-Learning, for estimating $Q_1 \approx Q_2 \approx q_*$ [4]**

Algorithm parameter: step size  $\alpha \in (0,1]$ , small  $\varepsilon > 0$

Initialize  $Q_{1(s,a)}$ , and  $Q_{2(s,a)}$ , for all  $s \in S^+$ ,  $a \in A(s)$ , such that  $Q(\text{terminal}, \cdot) = 0$

**Loop** for each episode:

Initialize  $S$

**Loop** for each step of episode:

Choose  $A$  from  $S$  using the policy  $\varepsilon$ -greedy in  $Q_1 + Q_2$

Take action  $A$ , observe  $R, S'$

With 0.5 probability:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha [R + \gamma Q_2(S', \underset{a}{\operatorname{argmax}} Q_1(S', a)) - Q_1(S, A)]$$

**else:**

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha [R + \gamma Q_1(S', \underset{a}{\operatorname{argmax}} Q_2(S', a)) - Q_2(S, A)]$$

$S \leftarrow S'$

**Until**  $S$  is terminal

#### **1.2.2.3.4.3.4 Actor-critic.**

This algorithm belongs to the family of policy gradient algorithms and is an on-policy method, see paragraph 1.2.2.3.2. Some general information about policy gradient methods and how they perform exploration of the environment is included in paragraph 1.2.2.3.1.2. The main characteristic of the actor-critic algorithm and the policy gradient method in general is that the agent chooses an action using an equation that does not consider the value function or the action-value function and is simply used to generate the agent's policy. This part of the algorithm is called actor. In the case of actor-critic algorithms the value function is used to critique the action the actor takes. The TD error of the value function, see paragraph 1.2.2.3.4.1, is used to critique the actor's actions and the agent policy in general. The part of the algorithm that estimates the value function is called the critic. The TD error is used for the training of both algorithm parts, the actor and the critic.

In its simplest form, the actor-critic algorithm works as follows. The critic part is the state value function. After the action  $a_t$  is chosen a new state  $s_{t+1}$  is observed, then the critic



checks whether the estimation of the value function for the initial state  $s_t$  is accurate using the equation

$$TD_{error} = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \quad (1.23)$$

This error is then used to evaluate the action  $a_t$  just selected. If the error is positive then the preference for the action  $a_t$  is strengthened, if it is negative it is weakened. Assume that we use Gibbs softmax method to generate the action, the equation, which produces the probability for the action selection, is  $\pi_t(a|s) = \frac{e^{p(s,a)}}{\sum_b e^{p(s,b)}}$ , where  $p(s,a)$  denotes the actor's preference for action  $a$  when in state  $s$ . Changing the preference for each action, described above, can be done using a simple equation of the form

$$p(s_t, a_t) \leftarrow p(s_t, a_t) + \beta TD_{error} \quad (1.24)$$

where  $\beta$  is a positive step-size parameter.

The policy gradient methods have many advantages over action-value methods. Some of them are:

**Use in stochastic policies:** They use probabilities to take actions, making them suitable for tasks that require stochastic policy such as card games.

**Environment exploration:** They can explore the environment because of the way they operate (generating probabilities for their action) and can arrive at a deterministic policy. In contrast,  $\epsilon$ -greedy policies will always have some randomness in their decision.

**Continues actions spaces:** They can handle continuous actions spaces.

Since this is the main algorithm that we will use in this thesis, the following chapter explains it further.

### 1.2.2.3.5 n-step TD algorithms.

Both algorithms TD, see paragraph 1.2.2.3.4, and MC, see paragraph 1.2.2.3.3 have advantages and disadvantages.

MC methods have low variance and are less biased as they are not based on an estimate of expected rewards but on an average of actual returns. Although they must wait for the episode to end and then update the value function or the action-value function. This behavior creates problems when the episode is too long, resulting in slow learning, and when the task is not episodic.

In contrast TD methods are implemented in an online, fully incremental manner, overcoming the MC problems described above, specifically they do not have to wait the episode to end to begin training. On the other hand, they have more variance and are more biased, they calculate the future expected rewards bootstrapping from their estimation of value function or action-value function for the next state, see paragraph 1.2.2.3.4.1.

For this reason, methods have been developed that combine features from MC and TD algorithms. These methods tend to perform better than the MC and TD methods. The methods are called n-step TD algorithms. They work in a similar way to TD algorithms. The difference is that in the case of the n-step algorithm more real rewards are used when updating the value function  $V(s_t)$  of an arbitrary state  $s_t$  or the action-value function  $Q(s_t, a_t)$  of an arbitrary pair of state  $s_t$  and action  $a_t$ . Specifically n-1 rewards are used (discounted as described in paragraph 1.2.2.2), and the remaining expected returns are calculated from the algorithm's estimate of the value function  $V(s_{t+n})$  for the environment state  $s_{t+n}$  or the action-value function  $Q(s_{t+n}, a_{t+n})$  for the pair of environment state

$s_{t+n}$  and action  $a_{t+n}$ , encountered  $n$ -step after the state or state action pair for which the update is performed.

We must keep in mind that for  $n$ -step TD algorithms with  $n \geq 1$  the estimation of the discounted expected returns involves rewards and states that are not known at the time of transition from  $t$  to  $t + 1$ . For this reason, no feasible algorithm can use  $n$ -step return until it has seen  $R_{t+n}$  and estimate  $V_{t+n-1}$  or  $Q_{t+n-1}$ . This condition is satisfied at  $t + n$ , and the equation used by the  $n$ -step TD algorithm to learn the value function takes the form

$$V_{t+n}(s_t) = V_{t+n-1}(s_t) + \alpha[G_{t:t+n} - V_{t+n-1}(s_t)], 0 \leq t \leq T \quad (1.25)$$

while the values of all other states remain unchanged, and the  $G_{t:t+n}$  denotes the discounted expected rewards and is equal with

$$R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(s_{t+n}) \quad (1.26)$$

Similarly, the equation used by the  $n$ -step TD algorithm to learn the action-value function is equal to

$$Q_{t+n}(s_t, a_t) = Q_{t+n-1}(s_t, a_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(s_t, a_t)], 0 \leq t \leq T \quad (1.27)$$

while the values of all other pairs of states and actions remain unchanged, and the  $G_{t:t+n}$  symbolize the discounted expected rewards and is equal to

$$R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(s_{t+n}, a_{t+n}) \quad (1.28)$$

All the algorithms presented in paragraph 1.2.2.3.4.2 can now be implemented with the  $n$ -step TD method. For better understanding a pseudocode for the  $n$ -step SARSA algorithm will be given.

#### **n-step SARSA for estimating $Q \approx q_*$ [4]**

Initialize  $Q_{(s,a)}$  arbitrarily, for all  $s \in S, a \in A$

Initialize  $\pi$  to be  $\epsilon$ -greedy with respect to  $Q$ , or to a fixed given policy

Algorithm parameters: step size  $\alpha \in (0,1]$ , small  $\epsilon > 0$ , a positive integer  $n$

All store and access operations (for  $S_t, A_t$  and  $R_t$ ) can take their index mod  $n + 1$

**Loop** for each episode:

Initialize and store  $S_0 \neq \text{terminal}$

Select and store an action  $A_0 \sim \pi(\cdot | S_0)$

$T \leftarrow \infty$

**Loop** for  $t = 0, 1, 2, \dots$ :

**If**  $t < T$ , then:

Take action  $A_t$

Observe and store the next reward as  $R_{t+1}$  and the next state as  $S_{t+1}$

```

If  $S_{t+1}$  is terminal, then:
     $T \leftarrow t + 1$ 
else:
    Select and store an action  $A_{t+1} \sim \pi(\cdot | S_{t+1})$ 
     $\tau \leftarrow t - n + 1$  ( $\tau$  is the time whose estimate is being updated)
If  $\tau \geq 0$ :
     $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$ 
    If  $\tau + n < T$ , then  $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$   $(G_{\tau:\tau+n})$ 
     $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[G - Q(S_\tau, A_\tau)]$ 
    If  $\pi$  is being learned, then ensure that  $\pi(\cdot | S_\tau)$  is  $\epsilon$ -greedy wrt  $Q$ 
Until  $\tau = T - 1$ 

```

### 1.3 Function approximation and Neural-Networks.

So far, we studied algorithms that try to find the optimal policy for a problem based on a value function, an action-value function, or a policy function. The value and the action-value functions are represented as tables that map each state or state-action pair to specific rewards expected to be earned if a policy  $\pi$  is followed after that state or state-action pair, in the case of policy function states are mapped to action probabilities. This way of finding an exact mapping of state or state action pair to expected rewards or to action probabilities is very effective for cases with limited state and action spaces. Although, it is impractical, in terms of memory and time, to do it for problems with big state and action spaces. One such case is the game of this thesis, see paragraph 1.1.1. There can be almost infinite number of states, such as different maps, different enemies and items placed on the map and different cases of hero status. One solution to problems like the game mentioned earlier is to create functions that can generalize. By generalization we mean the ability to make a reasonable estimate of the value function, the action-value or the policy function for states that appear for the first time based on training done for similar situations.

In our case the generalization needed is often called function approximation because it takes examples from a desired function (e.g., a value function) and tries to generalize from them to create an approximation of the entire function.

Henceforth, the three functions used in the algorithms that described in paragraph 1.2.2.3 will be not represented as tables, instead they will be represented as parametrized functions and the parameters will be in the form of a weight vector  $\mathbf{w} \in \mathbb{R}^d$ . So, for the approximate value of the state  $s$  given the weight vector  $\mathbf{w}$  we will write  $\hat{u}(s, \mathbf{w}) \approx u_\pi(s)$ , for the approximate action-value of the state  $s$  and action  $a$ , given the weight vector  $\mathbf{w}$ , we will write  $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$  and for the approximate policy function of the state  $s$ , given the weight vector  $\mathbf{w}$ , we will write  $\pi(s|a, \mathbf{w}) \approx \pi(s|a)$ .

In recent years most of the RL algorithms, including some of the most successful implementations [1],[2], use Neural Networks, see paragraph 1.3.2, to approximate the functions. In that case, which is also used in this thesis,  $\mathbf{w}$  denotes the connection weights in all the layers of the Neural Networks. Typically, the number of weights (the dimensionality of  $\mathbf{w}$ ) is much less than the number of states ( $d \ll |S|$ ) and changing one weight changes the estimated results of many states or state action pairs. Consequently, when a single state is updated, the change generalizes from that state or state action pair

to affect the values, action-values, or action selection probabilities of many other states or state action pairs. Such generalization makes the learning potentially more powerful but also potentially more difficult to manage and understand.

Given the introduction of the weight parameters  $\mathbf{w}$  and how they determine the results the algorithms produce, we need to define how these parameters will be updated after the agent interacts with the environment. For this reason, we need to introduce the concept of optimization and especially gradient descent methods.

### 1.3.1 RL optimization and gradient descent methods

The introduction of the approximation brings the error to the system. In the previous chapter we tried to find the exact values of the following functions  $u_\pi(s)$  and  $q_\pi(s, a)$ , now by parameterizing functions we come to terms with the idea that approximate functions  $\hat{u}(s, \mathbf{w})$  and  $\hat{q}(s, a, \mathbf{w})$  will differ from the actual values by an amount. Our goal is to reduce this amount to a minimum. One class of methods used to do this are gradient descent methods.

#### 1.3.1.1 Stochastic gradient descent.

In general gradient descent methods are used when the parameters of a parameterized function must be changed to achieve the minimum output for that function for given inputs. A simple gradient descent pseudocode algorithm for a parametrized real-valued function  $E: \mathbb{R}^N \times \mathbb{R}^P \rightarrow \mathbb{R}$  is given below.

#### Stochastic gradient descent method [7]

**Input:** differentiable function  $E: \mathbb{R}^N \times \mathbb{R}^P \rightarrow \mathbb{R}$  to be minimized

Step size sequence  $a_t \in [0, 1]$ , initial parameters  $\mathbf{w}_0 \in \mathbb{R}^P$

**Output:** a parameter vector  $\mathbf{w}$  such that  $E$  is small

**For all**  $t \in \{1, 2, \dots\}$  **do**

    Observe  $x_t, E(x_t, \mathbf{w}_t)$

    Calculate gradient:

$$\nabla_{\mathbf{w}} E(x_t, \mathbf{w}_t) = \left( \frac{\partial}{\partial w_{t[1]}} E(x_t, \mathbf{w}_t), \dots, \frac{\partial}{\partial w_{t[P]}} E(x_t, \mathbf{w}_t) \right)^T$$

    Update parameters:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha_t \nabla_{\mathbf{w}} E(x_t, \mathbf{w}_t)$$

To update the parameter vector  $\mathbf{w}_t$ , the derivatives of function  $E$  with respect to the elements of that vector, calculated for the input  $x$ , are used. Because these updates are following the negative gradient the output of the function becomes smaller. Gradient descent methods are called “stochastic” when the update is done on only a single example, which might have been selected randomly, like the algorithm presented previously. We should mention that, since the gradient only describes the local shape of the function, this algorithm may end up in a local minimum.

To achieve the convergence to a local minimum a careful choice of the step size parameter  $\alpha$  must be made. The parameter must be small enough. Remember that we neither seek nor expect to find a value function that has zero error for all states, but only an approximation that balances the errors in different states. If we were to fully correct each example in one step, then we would not find such a balance. In fact, the convergence results for SGD methods assume that  $\alpha$  decreases over time.

In RL a common example of a function we are trying to minimize is the TD function, and very often the mean squared error of the TD.

### 1.3.1.2 RL optimization algorithms in conjunction with SGD and semi-gradient descent

In this paragraph we will present a RL algorithm that tries to find an optimal policy for a problem and like the cases presented in the previous paragraphs, follow the concept of GPI, see paragraph 1.2.2.2.4.5. What is new about this algorithm is that it uses parameterized functions and gradient descent methods. The example that will be presented is on-policy, see paragraph 1.2.2.3.2, semi-gradient n-step SARSA. A pseudo code for the algorithm is following.

#### semi-gradient n-step SARSA for estimating $\hat{q} \approx q_*$ or $q_\pi$ [4]

Input: a differentiable action-value function parameterization  $\hat{q}: S \times A \times \mathbb{R}^d \rightarrow \mathbb{R}$

Input: a policy  $\pi$  (if estimating  $q_\pi$ )

Algorithm parameters: step size  $\alpha > 0$ , small  $\varepsilon > 0$ , a positive integer  $n$

Initialize value-function weights  $\mathbf{w} \in \mathbb{R}^d$  arbitrarily (e.g.,  $\mathbf{w} = 0$ )

All store and access operations ( $S_t, A_t$ , and  $R_t$ ) can take their index mod  $n + 1$

**Loop** for each episode:

    Initialize and store  $S_0 \neq \text{terminal}$

    Select and store an action  $A_0 \sim \pi(\cdot | S_0)$  or  $\varepsilon$ -greedy wrt  $\hat{q}(S_0, \cdot, \mathbf{w})$

$T \leftarrow \infty$

**Loop** for  $t = 0, 1, 2, \dots$ :

**If**  $t < T$ , then:

            Take action  $A_t$

            Observe and store the next reward as  $R_{t+1}$  and the next state as  $S_{t+1}$

**If**  $S_{t+1}$  is terminal, then:

$T \leftarrow t + 1$

**else:**

                Select and store  $A_{t+1} \sim \pi(\cdot | S_{t+1})$  or  $\varepsilon$ -greedy wrt  $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$

$\tau \leftarrow t - n + 1$  ( $\tau$  is the time whose estimate is being updated)

**If**  $\tau \geq 0$ :

$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$

**If**  $\tau + n < T$ , then  $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$       ( $G_{\tau:\tau+n}$ )

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{q}(S_\tau, A_\tau, \mathbf{w})] \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$

**Until**  $\tau = T - 1$

Notice the term *semi-gradient* in the name of this algorithm. The term is introduced because of the bootstrapping we use to create the target for our update

$$G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w}_\tau) \quad (1.29)$$

The bootstrapping part of the algorithm, in this case the  $\hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w}_\tau)$  depends on the current value of the weight vector  $\mathbf{w}_\tau$ , which means that it will be biased and that it will not produce a true gradient-descent method. Therefore, we only take the gradient for the part of the algorithm that contains the state we update  $\hat{q}(S_\tau, A_\tau, \mathbf{w}_\tau)$  and not for the target one, concluding that bootstrap methods are not in actual cases of true gradient descent [8] and as result we call them semi-gradient methods.

Although semi-gradient methods converge less strongly than SGD, we prefer to use them because of the same advantages that bootstrap methods offer, they learn faster and perform online updates, without waiting for the end of an episode.

### 1.3.2 Artificial Neural Networks

ANNs artificial neural networks try to simulate the behavior of the mammalian nervous system and are very often used to approximate nonlinear function. A simple representation of a feedforward ANN is given in Figure 13.

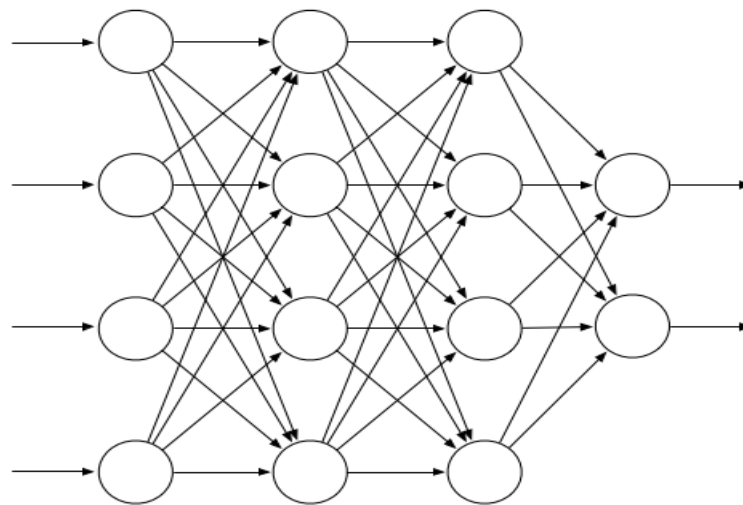


Figure 13: Basic feedforward ANN representation [4]

As we can see the network consists of different layers, the first layer is the input layer, the other two are the hidden layers and the last one is the output layer. Each connection between network units is characterized by a value called weight. Each unit takes the weighted output of the previous units sums it up and then applies to the results a non-linear function, called an activation function, the product of this function is the output of the module, this output can be the input for the others level units or the output of the ANN. Some examples of activation function are given in Figure 14.

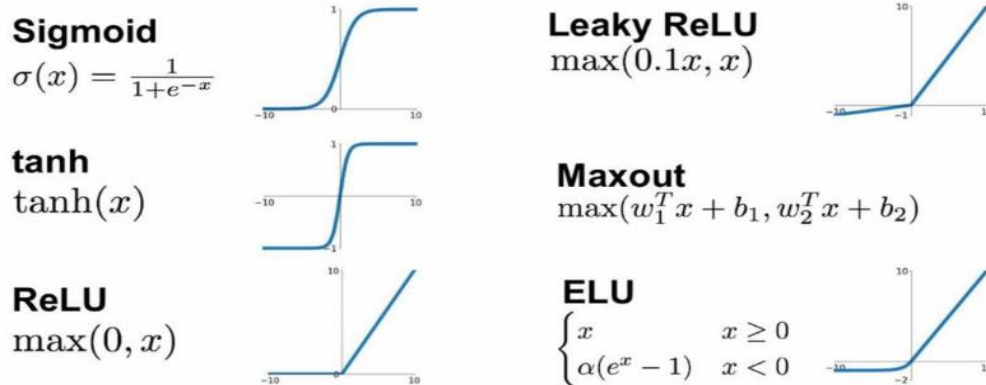


Figure 14: Examples of activation function[9]

The non-linearity of the activation function gives the ability to the ANNs to approximate any non-linear function. Specifically, an ANN with a single hidden layer containing a large enough finite number of sigmoid units can approximate any continuous function on a compact region of the network's input space to any degree of accuracy [10].

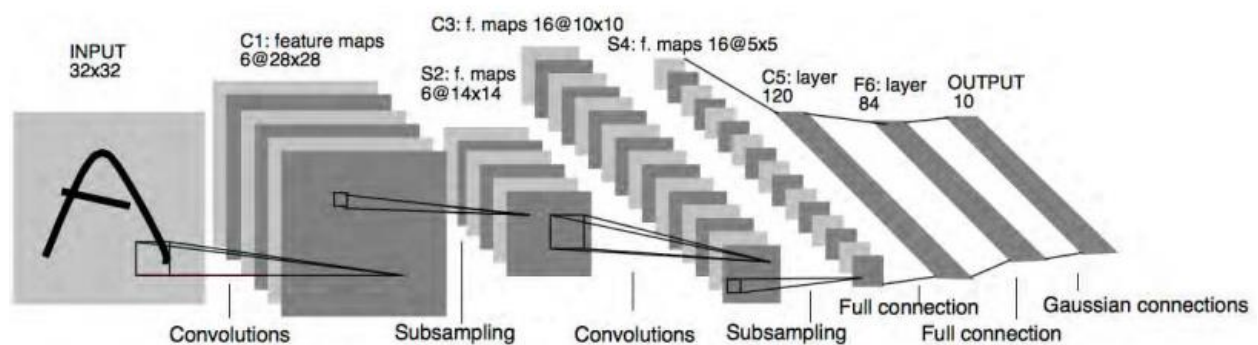
Although, a single hidden layer is not enough, especially for the AI tasks, the approximation of the complex function of the AI requires abstractions that are hierarchical compositions of many layers of lower-level abstractions like the ones produced by ANNs with many hidden layers. Each layer computes a more abstract representation of the network input with each unit providing a feature that contributes to the hierarchical representation of the network input-output function. These types of ANNs are called deep neural networks (DNN).

For this reason, training ANN hidden layers can generate features capable of hierarchically representing a given problem without relying solely on hand-crafted features. In the case of RL, the weights between units are the values used to approximate a function such as TD. Considering this, it is important to train the weights and the algorithm we use is similar with SGD, see paragraph 1.3.1.1. The most common way to implement SGD is through backpropagation. This method uses the chain rule and the layer structure of networks to efficiently calculate the derivatives of the network's output on its parameters. More specifically, the ANN alternates between forward and backward passes through the network. In the forward pass the activation of each unit is calculated, for a given network input. Then in the backward pass we take the results we want to minimize, for example the TD function, and then we calculate a partial derivative of the results with respect to each weight, finally we use these derivatives to update the weights.

There are many types of neural networks. In the following paragraphs we will describe the categories used in this thesis. The categories are Convolutional Neural Networks CNN and the Long Short-Term Memory LSTM.

### 1.3.2.1 Convolutional Neural Networks

CNN is built to process high-dimensional data arranged in spatial arrays, such as images. The CNN was inspired by how early visual processing works in the brain, and one of its first applications was the LeNet-5 network, which was created for hand-written character recognition [11]. The architecture of the LeNet-5 network is shown in Figure 15.



**Figure 15: Architecture of LeNet-5. Each plane is a feature map, i.e., a set of units whose weights are constrained to be identical.[11]**

The composition of the LeNet-5 network is created by combining convolutional and subsampling layers, followed by several fully connected final layers. Several feature maps are produced by each convolutional layer. The map represents a pattern of activity over an array of units. Each unit operates the same way on the data it "sees" from the previous layer (or from the input for the first convolutional layer). So, the units of each feature map

are the same and therefore share the same weights and are trained to recognize a particular feature from the data, the difference comes from the part of the data that each unit processes, the part of the data that each unit sees is shifted by a constant. For example, in the LeNet-5, figure 15, the first convolutional layer produces 6 feature maps each consist of  $28 \times 28$  units. Each unit sees a  $5 \times 5$  region of the original data, and these regions overlap for four columns and four rows. As result, each feature map is characterized by just 25 adjustable weights.

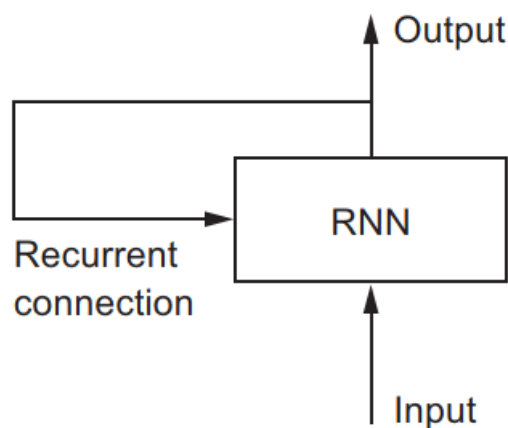
Subsampling layers are used to reduce the spatial resolution of the feature maps and make the network responses spatially invariant. These layers produce the same number of feature maps as the previous convolutional layer. Although this time the units of the new feature maps average over a portion of the units from the previous convolutional layer feature maps. For example, each unit of the 6 feature maps in the first subsampling layer of the LeNet-5 calculates the average for the units of a  $2 \times 2$  non-overlapping part of the feature maps produced by the first convolutional layer, having as results a six  $14 \times 14$  feature maps.

This is the ANN class that we will use in the image processing part of this thesis.

### 1.3.2.2 Recurrent Neural Networks and Long Short-Term Memory.

The ANN description at the outset considers a network that processes its inputs separately, consequently each network output is not affected from previous states. This behavior is effective for tasks where no prior data is needed, such as image recognition, but is less effective for tasks that require a sequence of data such as text prediction. For this reason, recurrent neural networks (RNNs) were introduced [12].

Actually, what we are aiming for by using RNN is to give memory capability to our network. The way this is done is simple, the RNN is a neural network that has a loop, see Figure 16.



**Figure 16: A recurrent network: a network with a loop [13]**

If we unfold a simple RNN over time, we get the sequence of Figure 17. As we can see, the network output  $h_t$  each time depends on the previous output  $h_{t-1}$  and the current observed state  $s_t$  a behavior that allows our network to produce an output considering an arbitrary number of previous states.

Although RNNs appear to be powerful in theory, they suffer from an inability to capture large dependencies due to two problems. Both are associated with error signals which as they flow backwards in time tend to either blow up or vanish. The first cases can cause weights to oscillate and for the second learning to bridge long time delays takes prohibitive time or doesn't work at all [14].



One solution which counter the two problems defined above is the Long Short-Term Memory (LSTM) [14], a special RNN architecture capable of capturing long term time dependencies.

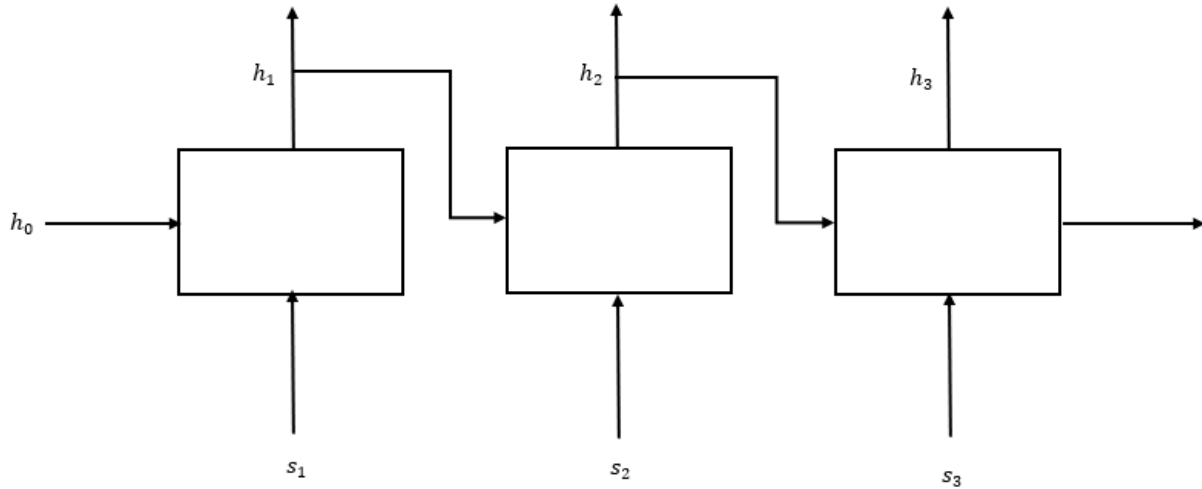


Figure 17: A simple RNN, unrolled over time.

The LSTM networks consist of several memory cells. The concept that enables the LSTM to address the problem of capturing long term time dependencies is that these memory cells have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN, and also use gates that learn to open or close depending on the observed state, the previous internal state (self-loop) and the output from the previous cell.

A block diagram of an LSTM cell is given in Figure 18. Each of these cells connect recurrently to each other in a similar manner like figure 17. A description of the feedforward flow of the block diagram is given below.

A regular artificial neuron unit computes the input feature, then its value is accumulated into the state provided that the sigmoidal input gate allows it. Then the state unit has a linear self-loop whose weight is controlled by the forget gate and the final output of the cell is controlled by the output gate. The gating units use a sigmoid nonlinear function like that in Figure 14 and the input unit can have any squashing nonlinearity.

The equations describing the LSTM behavior are given below. For these equations we will use the notation from [15] where the input state is represented as  $x_i^{(t)}$ , the internal state as  $s_i^{(t)}$  and the cell output as  $h_i^{(t)}$  ( $t$  represents the time step and  $i$  the cell).

The forget gate:

$$f_i^{(t)} = \sigma \left( b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)} \right) \quad (1.30)$$

Where  $x_i^{(t)}$  and  $h_i^{(t)}$  are the current input and the input from the previous cell respectively and the terms  $b_i^f$ ,  $U_{i,j}^f$  and  $W_{i,j}^f$  are the biases and weights of the forget gate.

The internal state update equation:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \left( b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)} \right) \quad (1.31)$$

Where  $b, U$  and  $W$  respectively denote the biases and weights in the LSTM cell. The external input gate is computed like the forget gate

$$g_i^{(t)} = \sigma(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}) \quad (1.32)$$

The output gate equation:

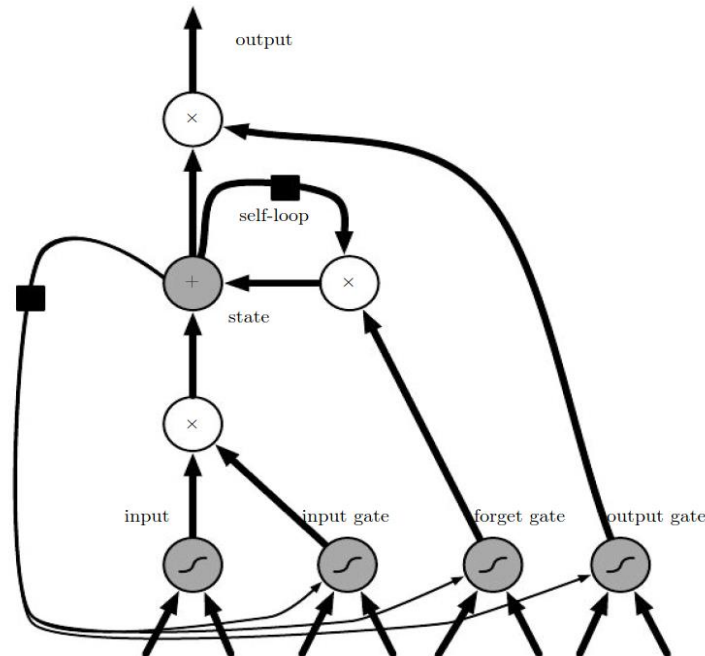
$$q_i^{(t)} = \sigma(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}) \quad (1.33)$$

Where terms  $b_i^o, U_i^o$  and  $W_i^o$  are the biases and weights of the output gate.

The output equation of the LSTM:

$$h_i^{(t)} = \tanh(s_i^{(t)}) q_i^{(t)} \quad (1.34)$$

we must keep in mind that if the internal state  $s_t$  is used as an extra input for the gates of the LSTM, three additional parameters would be required.



**Figure 18: Block diagram of the LSTM memory cell. The inputs for the input, input gate, forget gate and the output gate are the output of the previous cell  $h_i^{(t)}$  and the observed state  $x_i^{(t)}$ . The black square indicates a delay of a single time step [15]**

The error backpropagates through the same mechanism and the gates control the error flow in each cell.

This network class is used to introduce the memory element in the ANN of this thesis. Memory is necessary when the state of the environment is not full observable by the agent. Further details will be given in the next paragraph.

### 1.3.2.2.1 Partial observability and LSTM networks

Until now we suppose that the state of the environment is fully observable from the agent we try to train, unfortunately this is not the case for most of the problems the RL algorithms try to solve, and this holds for the game of the thesis too. The ability to deal with such problems, came from the function approximation, this may seem odd although consider that the parameterized form of a function does not allow its output to depend on certain

aspects of the state, then it is just as if those aspects are not observed. In fact, all the theoretical results for methods using function approximation presented so far hold equally well in cases of partial observability [4]. Although, the problem is that approximation does not augment the state representation with memories of past observations.

RNN and especially LSTM helps to deal with the problem of partial observability and past observations [16]. To explain how this is done, we need to introduce a different mathematical framework than the one we have been using so far. The framework is the Partially Observable Markov Decision Process (POMDP), POMDP recognizes that the observation the agent receives is a glimpse of the state, for this reason POMDP is described as a 6-tuple  $(S, A, P, R, \Omega, O)$  where the first four are the states, actions, transitions probability distributions, and rewards values already discussed, see paragraph 1.2.2.2, and the other two refer to the observation that the agent sees  $o \in \Omega$ . A probability distribution generates the observation based on the environment state  $o \sim O(s)$ . A non-recurrent ANN can produce good results only when the observation reflects the environment state, otherwise the approximate function, for example the value function may be inaccurate since  $\hat{u}(o, w) \neq \hat{u}(s, w)$ . The introduction of recurrency adds the memory element and helps narrowing the gap between  $o$  and  $s$  by expansion to  $\hat{u}(o, w)$  and  $\hat{u}(s, w)$ , resulting in a better approximation of the functions we are interested in and better training algorithms in general.

#### 1.4 Challenges that Rogue-like games have

Challenges refer to the difficulty an AI agent faces in learning how to play a Rogue-like game.

The main challenge of the game is that it requires more time-extended planning strategies. How difficult this is for an AI agent can be seen through the poor performance of the deep Q-network (DQN) [1] algorithm in games such as Montezuma Revenge. The general problem that drives such behavior is the sparse rewards. The term means that the agent only receives reward signals after completing specific series of actions over extended periods of time. An example of extended time planning strategies a player should use for the thesis game, is the use of health potions, these items must be collected and then used after several time steps when the player's HP is low. The same applies to weapon items, each weapon must be evaluated, collected and after some step the benefits it offers can be visible to the player.

Other challenge is the randomness of the game. Each game map is randomly generated, as is the generation of items and enemies. This results in a very large state space which means that an agent that generalizes and explores effectively is required.

The graphics of this game are poor. This makes game entities less observable compared to the Atari 2600 games where the DQN algorithm [1] was tested.

In addition to the screen where the map, items, enemies and the player are displayed, the game provides information to the player through game log, see paragraph 1.1.1.7 and the hero status 1.1.1.6. In order for the agent to observe the game as a human player, all of the above information must be entered into the agent network.

Map and game entities are not fully visible to our hero, see paragraph 1.1.1.3.2. The same screen is given to the agent, so the game state used as input to the network is partially observable.

All the above challenges make the rogue-like game a competitive task for RL algorithms. In this thesis we try to address these challenges using different techniques. The analysis of the approach to the problem will be given in the next chapter.

## 2. Implementation

### 2.1 Publications and related works that influenced the thesis

Three publications were the main influence for the thesis, a brief presentation of the innovations they introduce are given below:

**“Human-level control through deep reinforcement learning”** [1] is considered a pioneering work in the field of game-playing AI agents and has introduced a new RL deep Q-network (DQN) agent that can combine reinforcement learning with a class of artificial neural networks known as deep neural networks and manage to play classic Atari 2600 games in human level.

**“Asynchronous Methods for Deep Reinforcement Learning”** [17] the authors try to solve the same problem, although the innovation of this publication is the use of different instances of the RL algorithm, which train the same (global) network asynchronously, running on a multi-core central processing unit CPU.

**“Playing FPS Games with Deep Reinforcement Learning”** [18] describes an RL algorithm built to play the game Doom a first-person shooter (FPS), a class of games more complex than Atari games due to the partial observability of the state and the 3D environment. The RL algorithm to handle this task, trains a model that performs two tasks simultaneously, recognizes the features the agent sees and takes actions based on a DQN, and also introduces the recurrence to the DQN using LSTM memory cells.

Elements from the three publications are used in the model of this thesis, how this was done will be explained in the next paragraph.

### 2.2 Algorithm used

The RL algorithm used is a model-free asynchronous multi-threaded variant of an actor-critic method, see paragraph 1.2.2.3.4.3.4. To be able to describe the algorithm we use, we need to define the following:

**The actor-critic method with parameterized functions:** Specifically, the functions used and how the gradients of these function are calculated are described in paragraph 2.2.1.

**Optimizer:** The optimization algorithm used for the weights-parameters update, see paragraph 2.2.2.

**Data correlation treatment:** The method used to deal with the correlation of observed data when using on-policy RL algorithms, see paragraph 2.2.3.

In the last subsection, a pseudocode for the algorithm used will be given along with a brief explanation of how it works.

#### 2.2.1 Parameterized functions and gradient calculation

The parametrization of the algorithm is performed thru ANN, see paragraph 1.3.2. The parametrized functions used by the algorithm are the value function  $\hat{u}(s, \mathbf{w}_{critic})$ , the preference  $p(s, a, \mathbf{w}_{actor})$  and policy function  $\pi(a|s, \mathbf{w}_{actor}) = \frac{e^{p(s,a,\mathbf{w}_{actor})}}{\sum_b e^{p(s,b,\mathbf{w}_{actor})}}$ .

The algorithm uses the n-step TD of the value function, see paragraph 1.2.2.3.5, to update the actor and the critic.

For the critic part of the algorithm, which computes the state value function, we seek to minimize the mean squared error of the n-step TD value function, so a gradient descent method is used for the weights update and the equation for the gradient calculation is

$$d\mathbf{w}_{critic} = -[(R_t + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots \gamma^n \hat{u}(s_{t+n}, \mathbf{w}_{critic}) - \hat{u}(s_t, \mathbf{w}_{critic})] \nabla_{\mathbf{w}_{critic}} \hat{u}(s_t, \mathbf{w}_{critic}) \quad (2.1)$$

Keep in mind that you calculate the semi-gradient, see paragraph 1.3.1.2.

For the actor part of the algorithm, we use a gradient ascent method. Recall that we seek to increase the preference of the action that moves the agent to high-valued states. The equation used for the gradient of the actor part was derived from the REINFORCE family of algorithm [19] and especially the REINFORCE with a baseline [19]. The REINFORCE algorithm updates the weights with the equation  $\nabla_{\mathbf{w}_{actor}} \ln \pi(a_t | s_t, \mathbf{w}_{actor}) G_t$ , with  $G_t$  being the discounted rewards for an episode, like Monte Carlo methods 1.2.2.3.3. To reduce variance, the REINFORCE with a baseline subtracts the discounted rewards  $G_t$  by an amount  $b_t(s_t)$  which is called baseline. The actor-critic algorithm uses as a baseline the estimation for the value function  $\hat{u}(s_{t+n}, \mathbf{w}_{critic})$  and the discounted reward  $G_t$  of the n-step TD algorithms, which is equal with  $R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots \gamma^n \hat{u}(s_{t+n}, \mathbf{w}_{critic})$  [4]. So, the equation for the gradient calculation of the actor part of the algorithm is

$$d\mathbf{w}_{actor} = [(R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} \dots \gamma^n \hat{u}(s_{t+n}, \mathbf{w}_{critic}) - \hat{u}(s_t, \mathbf{w}_{critic})] \nabla_{\mathbf{w}_{actor}} \ln \hat{\pi}(s_t | a_t, \mathbf{w}_{actor}) \quad (2.2)$$

We should mention that like [17] the entropy of the policy  $\pi$  is added to the above equation, in order to improve exploration and discourage premature convergence to suboptimal deterministic policies [20]. The equation for the entropy is

$$H(\pi(\cdot | s_t; \mathbf{w}_{critic})) = -\sum_{\alpha \in A} \pi(\alpha | s_t, \mathbf{w}_{critic}) \ln \pi(\alpha | s_t, \mathbf{w}_{critic}) \quad (2.3)$$

and the concluding equation for the gradient calculation of the actor part is

$$d\mathbf{w}_{actor} = [(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots \gamma^n \hat{u}(s_{t+n}, \mathbf{w}_{critic}) - \hat{u}(s_t, \mathbf{w}_{critic})) \nabla_{\mathbf{w}_{actor}} \ln \pi(s_t | a_t, \mathbf{w}_{actor}) + \beta \nabla_{\mathbf{w}_{actor}} H(\pi(|s_t; \mathbf{w}_{actor}))] \quad (2.4)$$

The hyperparameter  $\beta$  controls the strength of the entropy regularization term and for our case was set to 0.0001.

Knowing the gradient of the functions we seek to optimize, it is the time to find the best method to apply those gradients to our model weights. We use the Adam optimization algorithm [21]. A brief introduction to the Adam algorithm is given in the next paragraph.

## 2.2.2 Adam optimizer [21]

The Adam optimizer is an efficient stochastic optimization method that requires only first-order gradients. The name came from the term adaptive momentum estimation, which refers to the way the optimizer works. Specifically, the optimizer calculates individual adaptive learning rates for different parameters from estimates of the first and second moments of the gradients. The Adam optimizer was inspired from the AdaGrad [22] and RMSpro [23] combining benefits from both. From the first the ability to perform well with sparse gradients and from the second one the ability to work well in online and non-stationary settings. A pseudocode of the Adam algorithm is given below.

### Implementation of Adam algorithm for a stochastic objective function $f(w)$ [21]

**Require:**  $\alpha$ : Stepsize

**Require:**  $\beta_1, \beta_2 \in [0,1)$ : Exponential decay rates for the moments estimates

**Require:**  $f(w)$ : Stochastic objective function with parameters  $w$

**Require:**  $w_0$ : initial parameter vector

$m_0 \leftarrow 0$  (Initialize 1<sup>st</sup> moment vector)

$u_0 \leftarrow 0$  (Initialize 2<sup>nd</sup> moment vector)

$t \leftarrow 0$  (Initialize timestep)

**While**  $w_t$  not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_w f_t(w_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)

$u_t \leftarrow \beta_2 \cdot u_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate,  $\beta_1^t$  denotes  $\beta_1$  to the power  $t$ )

$\hat{u}_t \leftarrow u_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate,  $\beta_2^t$  denotes  $\beta_2$  to the power  $t$ )

$w_t \leftarrow w_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{u}_t} + \varepsilon)$  (Update parameters)

**end while**

In our case the hyperparameters was set at  $\alpha = 0.00001$ ,  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$  and  $\varepsilon = 10^{-8}$ .

Many of the benefits offered by the Adam method are that the magnitudes of parameter updates are invariant to rescaling of the gradient, its step-sizes are approximately bounded by the step-size hyperparameter, it does not require a stationary objective, it works with sparse gradients, and it naturally performs a form of step size annealing [21].

We can apply the Adam optimization combined with the actor-critic algorithm to train our agent to play the thesis game. Although first we must consider a problem that arises for RL algorithms, which is the correlation of the data (states) observed by the agent. An explanation of the strategy introduced, in this thesis, to deal with the correlation problem will be given below.

### 2.2.3 Data correlation and asynchronous methods for deep reinforcement learning

The problem with correlating observed data is overfitting and instability. Instability can be caused also due to the reason that small updates to policy function  $\pi(a_t | s_t, \mathbf{w}_{actor})$  may significantly change the policy and therefore change the data distribution. In order to address these problems, the work in [1] introduces experience replay memory. The idea is simple, the observed states are stored in a buffer, and then random batches from that buffer are used to train the network, resulting in decorrelated updates. Although this method is restricted to the off-policy RL algorithms, our current parameters are different to those used to generate the sample, like the Q-learning method, see paragraph 1.2.2.3.4.3.1.

For this reason, in order to use on-policy RL method like the policy gradient algorithm and especially the actor-critic and benefit from the advantage they offer, see paragraph 1.2.2.3.4.3.4, a different approach must be taken.

The approach taken for this thesis is similar with [17]. Where instead of the experience replay memory method, multiple agents run different instances of the game in parallel,

resulting in uncorrelated observed data. Agents running on the same machine make it possible to perform updates like Hogwild! [24], where updates are performed asynchronously and without locks. Although this seems wrong, different agents can overwrite each other's updates, in practice, due to the sparsity of updates relative to the number of parameters of an ANN, correct updating of ANN parameters is possible. We should mention that a form of periodic update is used, like the case of [17] for better results, the way it is performed will be explained in the next subparagraph.

We must mention that the Adam optimizer, see paragraph 2.2.2, is used with shared statistics like the RMSpro optimizer used in [17]. Vectors  $m_t$  and  $u_t$  are shared among threads and are updated asynchronously and without locking. The shared version of optimizer was chosen because of its robustness [17]. Also, sharing statistics among threads reduces memory requirements by using one fewer copy of the parameter vector per thread.

Except the benefit of uncorrelation of the observed data the actor-critic of this thesis runs on a general-purpose CPU instead of a specialized hardware like GPU.

In the next subparagraph, an explanation of how the algorithm is implemented will be given, and a pseudocode of the algorithm will be included.

## 2.2.4 Algorithm explanation and its pseudocode

The algorithm is similar to the asynchronous advantage actor-critic (A3C) [17] and works in the forward view. Specifically, the agent selects actions using a policy  $\pi(a|s, \mathbf{w}_{actor})$  for  $t_{max}$  steps or until the end of an episode. At the same time calculates the value for each state observed  $\hat{u}(s, \mathbf{w}_{critic})$  and collect the rewards  $R_t$ . Then uses the rewards and the value of the states to calculate the n-step discounted reward for each state  $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots \gamma^n \hat{u}(s_{t+n}, \mathbf{w}_{critic})$ . Keep in mind that, for the last steps in the sequence where the n-step discounted rewards cannot be calculated the discounted rewards  $G_t$  for the largest available number of steps is used. Then the gradients for the actor part and the gradients for the critic part for each state are calculated using the equations of the subparagraph 2.2.1. The accumulated gradients are applied in a single step.

The updates the agents perform have a kind of periodicity. To achieve this a local and a global model are used. Each agent has its local model. The local model is synchronized with the global model, which is unique and is the one we actually seek to train, at the start of training, after  $t_{max}$  or at the end of an episode. During training each agent uses its local model, which is an asynchronous copy of the global model, and computes gradients based on the outputs of the local model. The gradients are accumulated and then used to update the global model.

A pseudocode of the algorithm for each agent is given below.

### Asynchronous advantage actor-critic - pseudocode for each actor-learner thread

```
// Assume global shared parameter vectors  $w_{actor}$  and  $w_{critic}$  and global shared counter  $T = 0$ 
```

```
// Assume thread-specific parameter vectors  $w'_{actor}$  and  $w'_{critic}$ 
```

```
Initialize thread step counter  $t \leftarrow 1$ 
```

```
Set  $n$  number of steps to calculate the n-step discounted reward
```

**Repeat**

Reset gradients:  $dw_{actor} \leftarrow 0$  and  $dw_{critic} \leftarrow 0$

Synchronize thread-specific parameters  $w'_{actor} = w_{actor}$  and  $w'_{critic} = w_{critic}$

$t_{start} = t$

Get state  $s_t$

**Repeat**

Perform  $a_t$  according to policy  $\pi(a_t|s_t, w'_{actor})$

Estimate the value function  $\hat{u}(s_t, w'_{critic})$

Receive reward  $r_t$  and new state  $s_{t+1}$

$t \leftarrow t + 1$

$T \leftarrow T + 1$

**Until** terminal  $s_t$  or  $t - t_{start} == t_{max}$

**If**  $s_t$  is terminal:

$\hat{u}(s_t, w_{critic}) = 0$

**For**  $i \in \{t_{start}, \dots, t - 1\}$  do

**If**  $i + n < t_{start} + t_{max}$  do

$power = 0$

$G_i = 0$

**For**  $z \in \{i \dots i + n\}$  do

$G_i \leftarrow G_i + \gamma^{power} r_z$

$power \leftarrow power + 1$

**End for**

$G_i \leftarrow G_i + \gamma^n \hat{u}(s_{i+n}, w_{critic})$

**If**  $i + n \geq t_{start} + t_{max}$  do

$power = 0$

$G_i = 0$

**For**  $z \in \{i \dots t - 1\}$  do

$G_i \leftarrow G_i + \gamma^{power} r_z$

$power \leftarrow power + 1$

**End for**

$G_i \leftarrow G_i + \gamma^{power} \hat{u}(s_t, w_{critic})$

**End for**

**For**  $i \in \{t_{start}, \dots, t - 1\}$  do



$$\text{Accumulate gradients wrt } w'_{actor}: dw_{actor} \leftarrow dw_{actor} + G_i \nabla_{w_{actor}} \ln \pi(a_i | s_i, w'_{actor}) + \beta \nabla_{w_{actor}} H(\pi(a_i | s_i, w'_{actor}))$$

$$\text{Accumulate gradient wrt } w'_{critic}: dw_{critic} \leftarrow dw_{critic} - [G_i - \hat{u}(s_i, w'_{critic})] \nabla_{w'_{critic}} \hat{u}(s_i, w'_{critic})$$
**End for**

Perform asynchronous update of  $w_{actor}$  using  $dw_{actor}$  and of  $w_{critic}$  using  $dw_{critic}$

**Until**  $T > T_{max}$ 

In our case, the algorithm uses a  $n$  equal with four and a  $t_{max}$  equal with eight. Thus, agents perform eight actions and then perform seven updates, four of which are four-step, one three-step, one two-step, and one one-step.

The parameters that define the way the algorithm perform the update, like  $n$  and  $t_{max}$  are called hyperparameters. All the hyperparameters used in the thesis algorithm can be found on “APPENDIX I: Hyperparameters I: Hyperparameters”.

We are using two separate model for the actor and the critic. The architecture of the network is given in the next paragraph.

### 2.3 Network architecture and inputs

For the parametrization of the functions, see paragraph 1.3 and 2.2.1, we will use ANNs, par 1.3.2. The type of the ANNs we will use depends on the inputs we want our agent to see. These inputs are the state of the environment  $s_t$  used on the algorithm described in the previous paragraph.

As we mention in the first chapter the game of the thesis provides to the player three type of information, the screen which depicts the environment of the game (see paragraph 1.1.1.2), the hero status (see paragraph 1.1.1.6) and the game log (see paragraph 1.1.1.7). Because of the importance of all this information for a human to play the game, it was decided to use all of them as input to our network. Although the different nature of the information and its size, especially the image size, requires the preprocessing of the inputs.

#### 2.3.1 Input preprocessing

The screen of the game has been rescaled by a factor of three. So, for a 444x444 screen the input will be a 148x148 screen. The same applies to the display of tiles, items, enemies and the player, for example an item that on the initial screen is displayed as a 3x3 square will now be a 1x1 square. An RGB channel is used to color each pixel. RGB channel values are in the range [0, 255]. This is not ideal for a neural network. we generally seek to keep input values small. For this reason, we rescale the values to be between zero and one dividing the channel values with the maximum value (255). The resulting screen used as an input to our network is an array with dimension of 148x148x3 and values between zero and one.

The status of the hero, see paragraph 1.1.1.6 is represented by a vector of eight values. The values give information regarding the hero including the weapon, the HPs, the maxHPs, the strength, the damage, the potions e.t.c. All the values are rescaled between zero and one dividing each value with its maximum. For example, the hero's current level is divided by five which is the maximum level the player can reach.

The game log contains the text messages of the game, see paragraph 1.1.1.7. To be able to use the information, the words were coded into numbers. The coding was very simple, each word in the game was mapped to an integer. The numbers are then entered into a  $5 \times 25$  table. Where the first dimension represents the rows of the game log and the second dimension the maximum words a message can have. When the words of a message are less than 25, the remaining values are set to zero. Values are rescaled again by dividing the integer representing a word by the maximum integer value used during encoding. Finally, the array reshaped into a vector of 125 values which is the input to our network. This coding was chosen to keep the ANN dimensionality low, reducing at the same time the textual input sensitivity, since using, for example, a one hot encoding for each word would result in more complex networks.

### 2.3.2 Network architecture

Two independent networks are used, one for the actor and one for the critic. The networks differ only on their output part. For this reason, we will present the common part and then the output part for each network.

**Common part:** Each input, see paragraph 2.3.1, is processed separately. First, we will describe the network used for the image processing. It is a CNN and has a similar structure to the network presented in paragraph 1.3.2.1. Although our network does not have the subsampling layers, remember that subsampling is used to reduce spatial sensitivity of a CNN, which we do not want for an agent playing our game, for example the position of an enemy is an essential information for the agent. So, the concluding CNN is composed of three convolutional hidden layers, the first takes as input the preprocessed image of shape  $148 \times 148 \times 3$  and convolves it using 32 filters of  $4 \times 4$  with stride 1 and applies a rectifier nonlinearity ReLU (Figure 14). The second hidden layer convolves 16 filters of  $5 \times 5$  with stride 2, again followed by a ReLU. This is followed by a third convolutional layer that convolves 8 filters of  $11 \times 11$  with stride 2 followed by a ReLU. The output is flattened and then is fed to a LSTM network, see paragraph 1.3.2.2. The output of the LSTM is a vector of 256 values.

The hero status input, an eight-value vector, is fed directly to an LSTM network which output a vector of 64 values.

The part of the network for the game log input, a vector of 125 values, works similar with the hero status network part and consists of a LSTM network with an output vector of 128 values.

The three LSTM networks outputs are combined to create a vector of 448 values, which in turn is fully connected to a linear network. This is the point where the actor and the critic networks differ.

**Actor linear network output:** The network has 9 units (neurons) one for each game action. The output of these units is used in a softmax equation  $(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$  and the action probability distribution is generated from the result.

**Critic linear network output:** The network has a unit (neuron) that takes the 448 input values and generates a single value, which is the output of the value function we are trying to approximate.

A block diagram of the two networks is given in “APPENDIX II: Networks Block Diagram”. The Netron application <https://github.com/lutzroeder/netron> was used for the visualization.

## 2.4 Reward Signal

Rewards are the only input to our network that drives training. For this reason, careful planning is needed. In our case the goal of the agent is to find the stairs leading to the next cave and successively to the ring of the wizard Werdna. During this process the agent must learn to explore a randomly generated environment, collect and use items and face enemies. For this reason, a subgoal reward system was used. For example, a positive reward is given when a new area is explored and a negative one when the agent moves towards a wall. This behavior, although not directly related to entering the new cave, we don't wait for the agent to find the stair tile and then give them a reward, it intends to lead the agent to explore the environment and consequently find the cave entrance. The reward system used is given in the following table.

**Table 5: The reward system used to train the agent**

Reward Condition	Reward
New area is discovered	20
The agent moves to already explored area	1
The agent moves towards a wall area	-1
The stair tile is visible when the agent moves	20
The agent enters a new cave or finds Werdna's ring.	500
A potion item is collected	100
The agent uses a potion and gain 20 HP	100
The agent has no potion and is trying to use one.	-0.1
A weapon is picked by the agent	100
The agent tries to pick a weapon when none is stored on the tile.	-0.1
The agent rests and gains 4 HP	10
The agent rests and gains 0 HP	-0.1
The agent attacks an enemy.	10
An enemy is killed by the agent	100
An enemy attacks the hero	-1
The agent attacks when no enemy is near him	-0.1
The HP of the hero increased	10

All rewards are added after each agent action. For example, if the agent moves and explores new territory and at the same time is attacked by an enemy, then the two rewards are added, and the resulting reward is equal to 19.

## 2.5 Trying to deal with the challenges of a Roguelike game

Rogue-like games have many challenges for an RL agent, a discussion of these challenges, and especially those of the thesis game, can be found in paragraph 1.4. In this paragraph we will present the methods used to address these challenges.

**Sparse rewards and time-extended planning strategies:** We attempt to address this problem by rewarding the agent for achieving subgoals that, if the agent completes, will lead to the fulfillment of the overall goal.

**Randomness of the game and huge state space:** The game, unlike many other games, is not deterministic. The map, items and enemies are randomly generated. For this reason, the state space is huge. To deal with this problem we set a relatively small learning rate (0.00001) to avoid converging to a false local optimum and in addition we use three methods. First, policy entropy is added to actor losses, see paragraph 2.2.1, aiding exploration, and second, we use multiple asynchronous agents, see paragraph 2.2.3, to prevent data correlation and hence overfitting. Finally, again the reward signal is shaped into a form capable of aiding exploration of the environment. These choices help the agent to visit the state space of the game as much as possible.

**Low level graphics:** Because of this problem we use CNN layers differently from DQN [1]. We use a first layer with a 4×4 filter shape, we do this because the game tiles where all game entities are located are the same size and our goal is to map all different game entities as features of the first convolutional layer. We then use two larger filters one with a 5×5 shape and one with an 11×11 shape to find more complex features.

**Information other than the game screen:** The game gives valuable information to the player through game log and hero status. We use the information as different inputs, from the game screen to our network and after some processing we combine it with the screen for the output, for details see paragraph 2.3.

**Partial observability:** The agent sees only a part of the game state. For this reason, we try to close the gap between the agent observation and the environment state by using the LSTM network and thereby adding memory to our agent, see paragraph 1.3.2.2.1. In agent training we use the last eight observations as input sequence to the LSTM network. For this reason, training starts after 16 steps from the beginning of each episode, 8 steps until the first sequence of observations is ready, and 8 steps until the first training batch is ready.

## 2.6 Code used in the thesis

The Python programming language is used for the game of the thesis and for training the agent. The game code can be found on GitHub <https://github.com/IliasSim/Wizard-Werdna-Ring-Adventure-game-only> and the code for the networks, agent and its training can be found in "APPENDIX III: Agent training code". The code for the Adam optimizer par.2.2.2 with shared statistics, see paragraph 2.2.3 can be found on GitHub [https://github.com/ikostrikov/pytorch-a3c/blob/master/my\\_optim.py](https://github.com/ikostrikov/pytorch-a3c/blob/master/my_optim.py) from a PyTorch implementation of the code used in [17].

The framework used to train the agent is PyTorch "An open source machine learning framework that accelerates the path from research prototyping to production development" An attempt was made to use the other most popular platform, Tensorflow,

although PyTorch is easier to implement for asynchronous training methods, it includes a module dedicated to multiprocessing, and for this reason it was chosen.

### 3. Results

The network described in paragraph 2.3 was trained using the algorithm described in paragraph 2.2 and the reward signal in paragraph 2.4. The CPU used during training was an Intel(R) Core (TM) i7-7700HQ CPU @ 2.80GHz with 4 cores and 8 logical processors. The type of the hero during the training was the warrior and the training was performed for about four thousand (4000) episodes, which corresponds to about seven million steps (7M) and roughly to nine hundred thousand (900) updates. A typical computer screen during the training is depicted in Figure 19

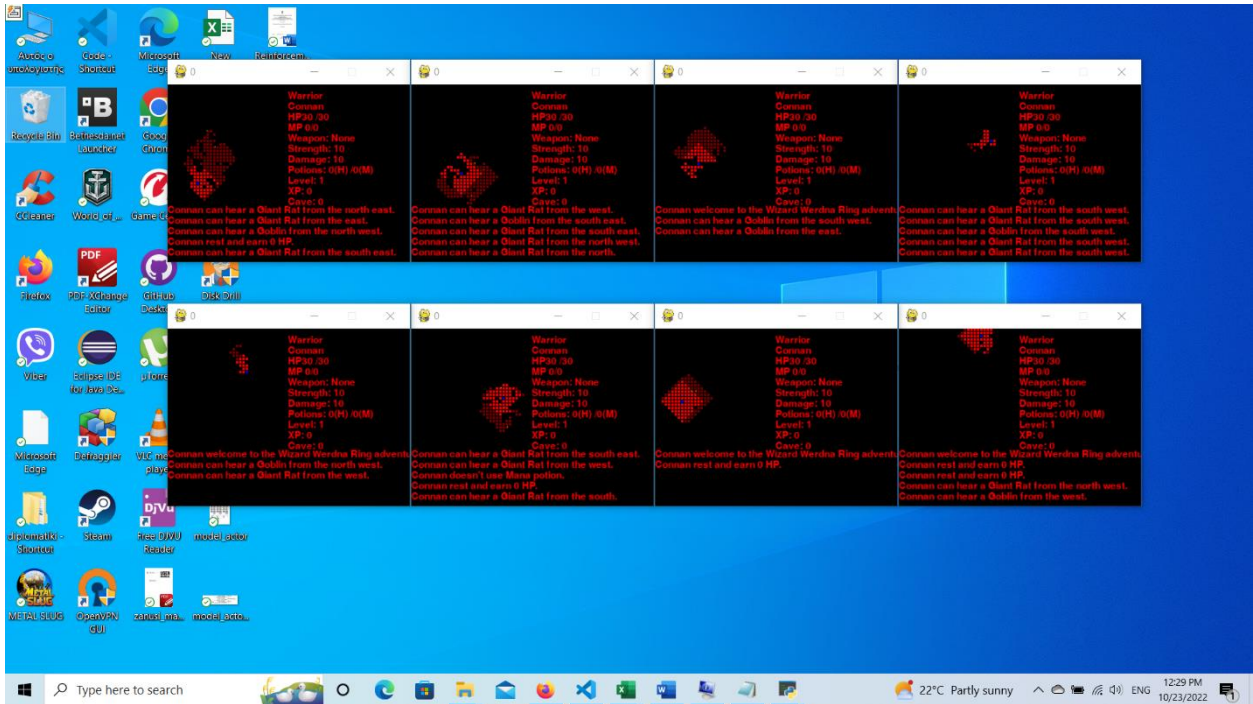


Figure 19: Typical computer screen during training

Training results in the agent accumulating more average rewards per episode compared to an agent acting randomly. The progress is depicted at Figure 20

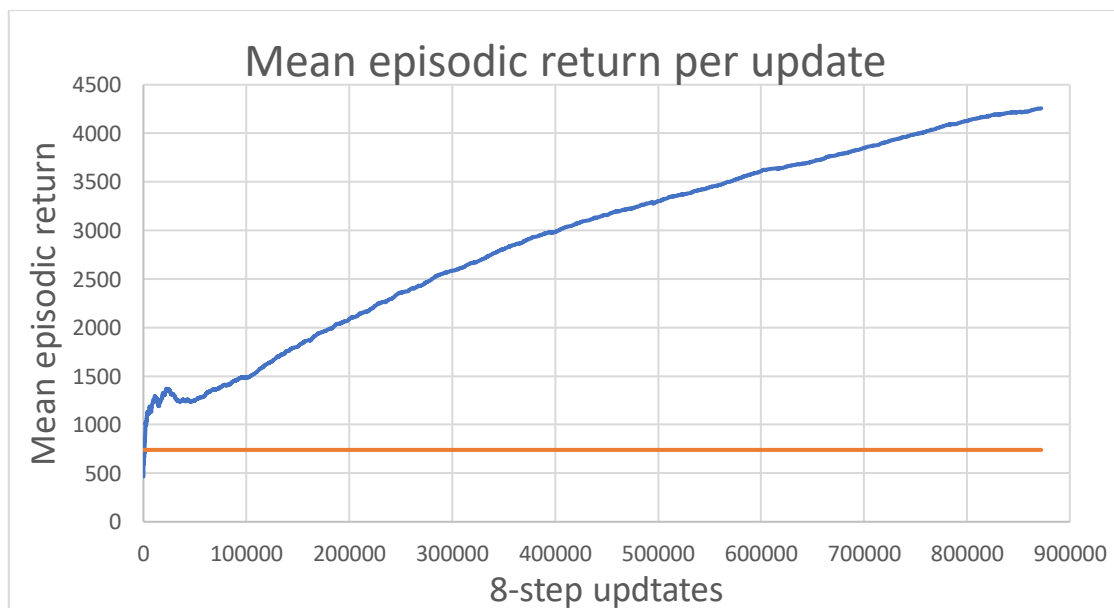
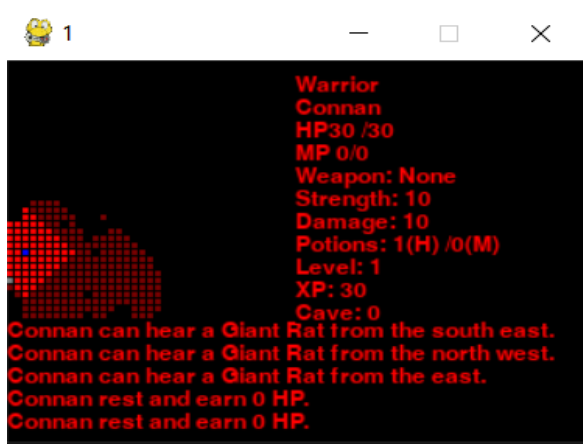


Figure 20: The blue line indicates the average episode rewards achieved during the agent training, and the red depicts the average episode rewards of an agent acting randomly for one thousand (1K) episodes.

Where the blue line shows the average episode rewards achieved during agent training and the red line depicts the average episode rewards achieved by an agent acting randomly over one thousand (1K) episodes.

As we can see, the agent successfully learns to gradually accumulate more rewards. Although we must keep in mind that these rewards come from a system created to help train the agent and are not a scoring system that characterizes the agent's success in the game. For this reason, we need to evaluate the training of the agent through its behavior and its ability to reach the final goal of the game which is the entrance to the next cave.

Results in this scope are mixed, with the agent certainly achieving some good behavior while some other goals are not. Specifically, the agent learns to react to enemy attacks and as a result survives longer during a game episode. For example, the next figures will describe the behavior of the agent when facing enemies, each figure caption will give the probability that an action is chosen, as well as the action the agent ultimately chooses.



Probabilities for the next hero action are [moves up:0.1998, moves right:0.1645, moves down 0.2082, moves left 0.2199, rests 0.0522, uses hp potion 0.0012, uses mp potion 0.0004, attacks 0.1535, picks a weapon 0.0004], next action of the hero is down



Probabilities for the next hero action are [moves up:0.1977, moves right:0.1711, moves down 0.2064, moves left 0.2236, rests 0.0485, uses hp potion 0.0011, uses mp potion 0.0003, attacks 0.1509, picks a weapon 0.0004], next action of the hero is down



Probabilities for the next hero action are [moves up:0.1299, moves right:0.0996, moves down 0.1199, moves left 0.1334, rests 0.0682, uses hp potion 0.0006, uses mp potion 0.0002, attacks 0.4480, picks a weapon 0.0002], next action of the hero is attack



Probabilities for the next hero action are [moves up:0.0853, moves right:0.0596, moves down 0.0739, moves left 0.0812, rests 0.0801, uses hp potion 0.0005, uses mp potion 0.0001, attacks 0.6193, picks a weapon 0.0001], next action of the hero is attack

Figure 21:Sequence of events when the agent faces an enemy



During the sequence of events, the agent senses the presence of an enemy then tries to attack and kill that enemy. In the next series of figures, we will see how the agent tries to recover from the battle.

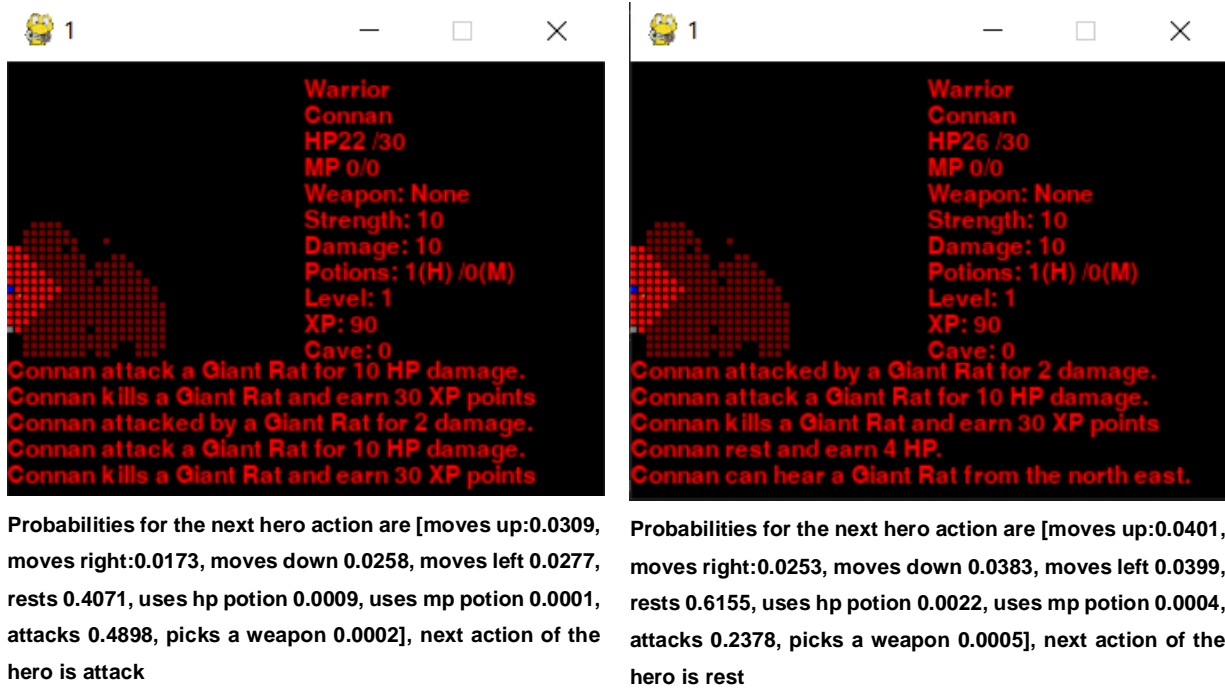


Figure 22: Agent recovers after the encounter with the enemy.

The agent also manages to have a good general orientation about which direction to move in order to fully explore the map, in the next series of figures different game starts are used to show the correct general orientation of the agent during exploration.

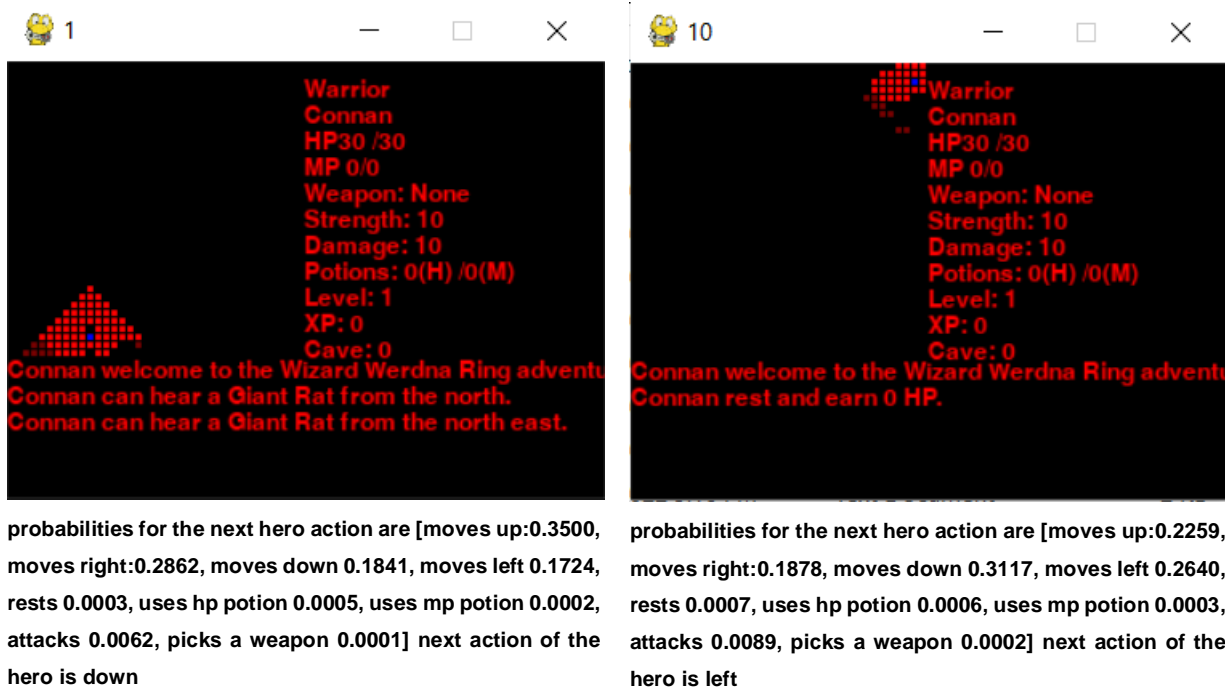


Figure 23: General orientation of the agent during exploration.



One of the things the agent failed to learn is how to properly use the health potion. The agent seems to prefer the rest function instead. This is due to the problem of sparse rewards, see paragraph 1.4, because it is easier for the agent to understand the immediate rewards given by the rest action than the longer-delayed rewards from the health potions, even if the health potions rewards are significantly larger.

Another area where the agent failed to develop the right strategy is acquiring weapons, again the problem of sparse rewards seems to be the main reason. The agent did not understand how acquiring a weapon in the present would benefit the future, either in the ability to kill enemies more easily or in the ability to withstand enemy attacks.

In general, the agent seems to learn how to survive and kill enemies and do some basic map exploration. Although the agent failed to effectively achieve the ultimate goal of the game, which is to enter the next cave. As we can see in Figure 24, the agent managed to enter the next cave and to explore a large part of the map and in the meantime kill many enemies, notice the level of the hero, on the other hand the agent was stuck in parts of the map unable to understand that more rewards can be collected by re-crossing the already discovered part of the map and exploring new areas, where the entrance to the next cave is located. This problem occurs again due to sparse rewards, although one more challenge is added. The challenge is the randomness of the game, each time the game map is generated is different and the location of the entrance to the next cave is unknown, so the agent cannot memorize a specific pattern of how to find the next cave entrance and it is necessary to explore each cave to find it. This leads to an agent unable to link the exploration with the final goal, which is the stair type tile, see paragraph 1.1.1.2.



The agent after 5380 moves has explore the most part of the map and manage to reach level 3.

The agent after 9669 steps is stuck in the same area and cannot conclude that traversing the already discovered areas of the map will lead to more rewards and the stair to the next cave.

Figure 24: The exploration problem faced by the agent

## 4. Conclusions

Most of the techniques, used in training the thesis agent, are well-proven. For example, CNNs are able to capture images and extract features from them [1],[11], LSTM can help in closing the gap between agent observation and game state [16],[18], asynchronous parallel training can deal with data correlation and speed-up training [17], and the Adam optimizer has the ability to perform well with sparse gradients and to work well in online and non-stationary settings [21]. Although, the challenges presented by the game of the thesis (par. 1.4) create a more challenging environment than those for which the aforementioned techniques have been used to demonstrate an RL agent capable of playing video games at human level.

In particular, the two challenges that increase the difficulty are the sparse rewards and the randomness of the game (see paragraph 1.4), which in turn creates a huge state space. As a result, the agent must learn to effectively explore the game environment and devise extended time strategies. In order to achieve this, three tools were used, the entropy (see paragraph 2.2.1), the small learning rate (see paragraph 2.5) and the rewards signal shaping (see paragraph 2.5). The process of tuning the reward signal proved to be a difficult task, requiring many trials, where small changes lead to very different results. For example, when the rewards used did not include punishment for pointless actions, like attacking without the presence of an enemy, led to an agent that explored less and stuck to specific actions. Similarly, when the punishment for moving toward a wall was equal to the punishment for taking pointless actions, the agent moved aimlessly near walls and was reluctant to move away from them. Thus, the rewards system described in paragraph 2.4 is a result of many tests and although is not perfect creates the agent with the most promising behavior.

Another important factor determining the behavior of the agent, was the inputs to the network. When games log (see paragraph 1.1.1.7) and hero status (see paragraph 1.1.1.6) were not included as network inputs, the agent failed to detect changes in the game state, such as hit point reduction, and react accordingly. For this reason, the use of this information was critical to the performance of the agent presented in this thesis.

The performance of the agent trained for this thesis has mixed results. The agent was able to learn how to survive on a game by attacking enemies and recovering lost hit points. Although, the agent fails to learn how to implement extended strategies (see paragraph 1.4) regarding the correct usage of items and the proper exploration of the game maps. This behavior exposes the limitation of the techniques used in this thesis. Simple optimization techniques like the one used in [1] can, with enough computational ability, solve naïve problems like some of the ATARI 2600 games. This problem arises for tasks that requires more extended time strategies like the game of the thesis and the Montezuma's Revenge, an ATARI game famously difficult for deep reinforcement learning methods.

Probably the efforts made to solve Montezuma's Revenge can be the solution for the ring of the Wizard Werdna game. In particular, two recent publications use advanced exploration techniques to perform well in Montezuma's Revenge. The first [25] introduce bootstrapped DQN, a simple algorithm that explores in a computationally and statistically efficient manner through use of randomized value functions. The second [26] uses random network distillation (RND) as an exploration bonus for deep reinforcement learning methods. The bonus is the error of a neural network predicting features of the observations given by a fixed randomly initialized neural network.

A future effort for creating an agent capable of playing the ring of the Wizard Werdna game effectively, should consider the use of these two advanced exploration techniques described above.

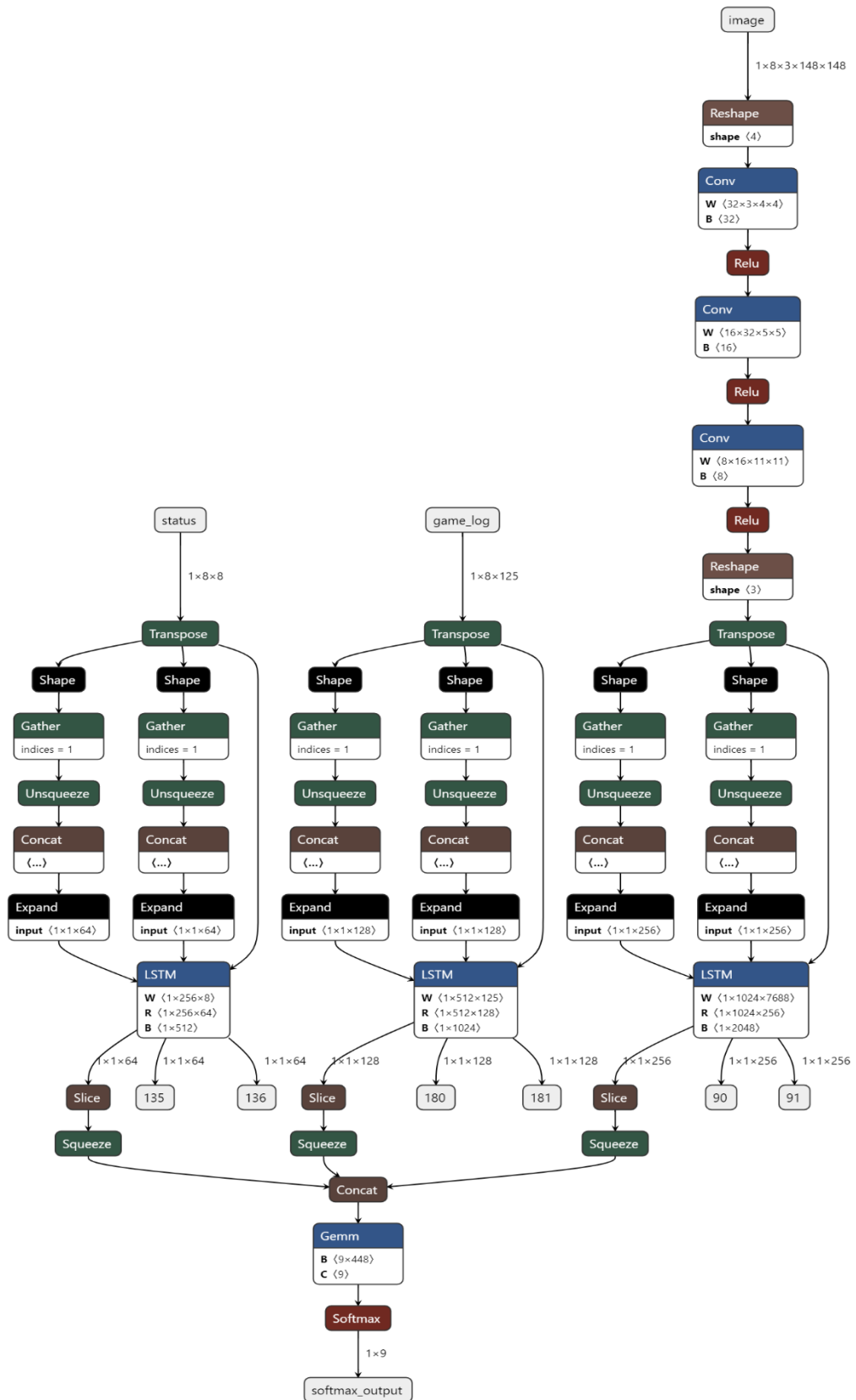
## APPENDIX I: Hyperparameters

List of hyperparameters and their values.

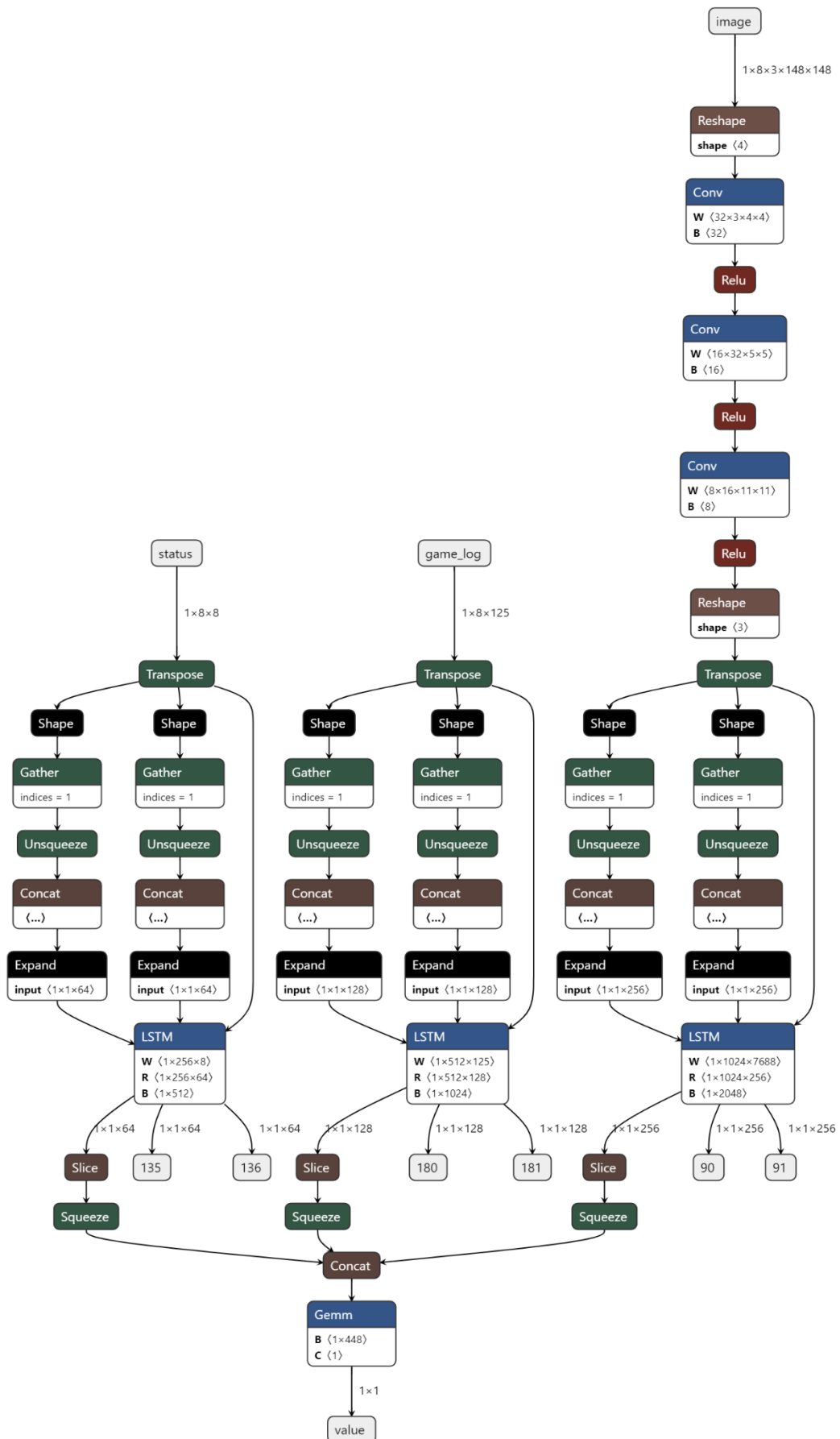
Hyperparameter	Value	Description
Training Batch	8	The number of training case over which the update is performed
History length	8	Number of consecutive states used as input to the LSTM network.
Discount factor	0.99	Discount factor $\gamma$ used in the calculation of the discounted reward $G_t$
Update frequency	8	The number of actions selected by the agent between successive updates.
$n$ -step	4	The maximum number of consecutive rewards used for the calculation of the discounted reward $G_t$
Entropy regularization $\beta$	0.0001	The parameter controls the amount of entropy $H(\pi(\cdot   s_t; \mathbf{w}_{critic}))$ added at the policy losses.
Learning rate $\alpha$	0.00001	The learning rate used by the Adam optimizer
Exponential decay rate $\beta_1$	0.9	The coefficient which controls the moving average of the gradient $m_t$ used by the Adam optimizer
Exponential decay rate $\beta_2$	0.999	The coefficient which controls the moving average of the squared gradient $u_t$ used by the Adam optimizer
Epsilon $\varepsilon$	$10^{-8}$	Term added to the denominator of the Adam update equation to improve numerical stability.

## APPENDIX II: Networks Block Diagram

### Actor network block diagram



### Critic network block diagram



**APPENDIX III: Agent training code**

```

import torch.cuda
import torch.nn.functional as F
from torch import nn
import torch.multiprocessing as mp
from GamePAI import GamePAI
import my_optim
import sys
import pygame
import numpy as np
import pandas as pd
from os.path import exists

featuresCNN1 = 32
CNN1Shape = 4
CNN1Step = 1
featuresCNN2 = 16
CNN2Shape = 5
CNN2Step = 2
featuresCNN3 = 8
CNN3Shape = 11
CNN3Step = 2
denseLayerN = 256
denseLayerNL_21 = 64
denseLayerNL_31 = 128
action_number = 9
h_step = 8
n_step = 4
batch = 8
screenfactor = 1
decay_steps = 10000
seeded = False
input = (148,148,3)
record = False
torch_g = True
dir = 'D:\ekpa\diplomatiiki\old_versions\Wizard-Werdna-Ring-Adventure-Problem-
fixed-float-instead-integer-for-playerstatus-and-game-text-also-better-approach-
for-the-LSTM-usage\playerActorLSTM_23_8_22_1_a3c_11'
input1 = (1, 1, 148, 148, 3)
input2 = (1,1,8)
input3 = (1,1,125)

class actor(nn.Module):
    '''This class creates the model of the actor part of the reinforcement learning
    algorithm'''
    def __init__(self):
        super(actor,self).__init__()
        self.cnn1 = nn.Conv2d(3,featuresCNN1,CNN1Shape,stroke=CNN1Step)

```

```

        self.cnn2 =
nn.Conv2d(featuresCNN1,featuresCNN2,CNN2Shape,stroke=CNN2Step)
        self.cnn3 =
nn.Conv2d(featuresCNN2,featuresCNN3,CNN3Shape,stroke=CNN3Step)
        self.lstmCnn = nn.LSTM(7688,denseLayerN,batch_first=True)
        self.lstmStatus = nn.LSTM(8,denseLayerNL_21,batch_first=True)
        self.lstmText = nn.LSTM(125,denseLayerNL_31,batch_first=True)
        self.act_prob =
nn.Linear(denseLayerN+denseLayerNL_21+denseLayerNL_31,action_number)

    def forward(self,input1,input2,input3,hiddenCnn = None,hiddenl1 =
None,hiddenl2 = None):
        batch_size, timesteps, C, H, W = input1.size()
        c1_in = input1.view(batch_size * timesteps, C, H, W)
        x = self.cnn1(c1_in)
        c2_in = F.relu(x)
        c3_in = F.relu(self.cnn2(c2_in))
        c_out = F.relu(self.cnn3(c3_in))
        lstmCnn_in = c_out.view(batch_size, timesteps, -1)
        if hiddenCnn is not None:
            lstmCnn_out, hiddenCnn_out = self.lstmCnn(lstmCnn_in, hiddenCnn)
        else:
            lstmCnn_out, hiddenCnn_out = self.lstmCnn(lstmCnn_in)
        if hiddenl1 is not None:
            lstm11_out, hiddenl1_out = self.lstmStatus(input2, hiddenl1)
        else:
            lstm11_out, hiddenl1_out = self.lstmStatus(input2)
        if hiddenl2 is not None:
            lstm12_out, hiddenl2_out = self.lstmText(input3, hiddenl2)
        else:
            lstm12_out, hiddenl2_out = self.lstmText(input3)
        a = F.softmax(self.act_prob(torch.cat((hiddenCnn_out[0][-
1],hiddenl1_out[0][-1],hiddenl2_out[0][-1])),dim=1)),dim=1)
        return a,hiddenCnn_out,hiddenl1_out,hiddenl2_out

class critic(nn.Module):
    '''This class creates the model of the critic part of the reinforcement
learning algorithm'''
    def __init__(self):
        super(critic,self).__init__()
        self.cnn1 = nn.Conv2d(3,featuresCNN1,CNN1Shape,stroke=CNN1Step)
        self.cnn2 =
nn.Conv2d(featuresCNN1,featuresCNN2,CNN2Shape,stroke=CNN2Step)
        self.cnn3 =
nn.Conv2d(featuresCNN2,featuresCNN3,CNN3Shape,stroke=CNN3Step)
        self.lstmCnn = nn.LSTM(7688,denseLayerN,batch_first=True)
        self.lstmStatus = nn.LSTM(8,denseLayerNL_21,batch_first=True)
        self.lstmText = nn.LSTM(125,denseLayerNL_31,batch_first=True)
        self.value = nn.Linear(denseLayerN+denseLayerNL_21+denseLayerNL_31,1)

```



```

def forward(self,input1,input2,input3,hiddenCnn = None,hiddenl1 =
None,hiddenl2 = None):
    batch_size, timesteps, C, H, W = input1.size()
    c1_in = input1.view(batch_size * timesteps, C, H, W)
    x = self.cnn1(c1_in)
    c2_in = F.relu(x)
    c3_in = F.relu(self.cnn2(c2_in))
    c_out = F.relu(self.cnn3(c3_in))
    lstmCnn_in = c_out.view(batch_size, timesteps, -1)
    if hiddenCnn is not None:
        lstmCnn_out, hiddenCnn_out = self.lstmCnn(lstmCnn_in, hiddenCnn)
    else:
        lstmCnn_out, hiddenCnn_out = self.lstmCnn(lstmCnn_in)
    if hiddenl1 is not None:
        lstm11_out, hiddenl1_out = self.lstmStatus(input2, hiddenl1)
    else:
        lstm11_out, hiddenl1_out = self.lstmStatus(input2)
    if hiddenl2 is not None:
        lstm12_out, hiddenl2_out = self.lstmText(input3, hiddenl2)
    else:
        lstm12_out, hiddenl2_out = self.lstmText(input3)
    v = self.value(torch.cat((hiddenCnn_out[0][-1],hiddenl1_out[0][-
1],hiddenl2_out[0][-1]),dim=1))
    return v,hiddenCnn_out,hiddenl1_out,hiddenl2_out

class agent():
    '''This class creates the agent, which interacts with the game and tries to
learn how to play following actor-critic RL algorithm'''
    def __init__(self,actor,critic, gamma = 0.99):
        self.gamma = gamma
        self.actor = actor
        self.critic = critic
        self.log_prob = None
        self.buffer_State = []
        self.buffer_playerStatus = []
        self.buffer_text = []
        self.buffer_reward = []
        self.buffer_size = 7

    def act(self,state,playerstatus,gameText,agent):
        '''This function returns the action the agent will perform based on the
enviroment state'''
        prob,_,_,_ =
self.actor.forward(torch.tensor(state).float(),torch.tensor(playerstatus).float()
, torch.tensor(gameText).float())
        print(prob,agent)
        dist = torch.distributions.Categorical(prob)
        action = dist.sample()
        return action

    def value(self,state,playerstatus,gameText):

```

```

        '''This function returns the agent estimation for the value of the
        enviroment state'''
        v,_,_,_ =
self.critic.forward(torch.tensor(state).float(),torch.tensor(playerstatus).float(
), torch.tensor(gameText).float())
        return v

def preprocess0(self, state,playerstatus,gameText,game_start):
    '''This function prepares the environment observations as input for the
    two networks. Creates packet of 8 consecutive observations as input for the lstm
    networks'''
    if game_start:
        self.buffer_State = []
        self.buffer_playerStatus = []
        self.buffer_text = []
    if len(self.buffer_State) > self.buffer_size:
        del self.buffer_State[0]
    self.buffer_State.append(state)
    state = np.array(self.buffer_State)
    state = np.expand_dims(state, axis=0)

    if len(self.buffer_playerStatus) > self.buffer_size:
        del self.buffer_playerStatus[0]
    self.buffer_playerStatus.append(playerstatus)
    playerstatus = np.array(self.buffer_playerStatus, dtype=np.float32)
    playerstatus = np.expand_dims(playerstatus, axis=0)

    if len(self.buffer_text) > self.buffer_size:
        del self.buffer_text[0]
    self.buffer_text.append(gameText)
    gameText = np.array(self.buffer_text, dtype=np.float32)
    gameText = np.expand_dims(gameText, axis=0)
    return state,playerstatus,gameText

def preprocess1(self, state,playerstatus,gameText,
rewards,values,actions,done, gamma):
    '''This function prepares the environment observations as input for
    training the networks. Creates 7 packs of 8 consecutive environmental
    observations and
    also calculates the discounted rewards for each case
    using the equation  $Vs_0=R_0+\gamma R_1+\gamma^2 R_2+\dots+\gamma^n Vs_4$ '''
    discnt_rewards = []
    if done:
        values[-1]=0
    for i in range(len(rewards)):
        if i + n_step < len(rewards):
            power = 0
            disc_rew = 0
            for z in range(i,i+n_step):
                disc_rew += rewards[z]*(gamma**power) #+
                power += 1

```

```

        disc_rew += values[i+n_step]*(gamma**n_step)
        discnt_rewards.append(disc_rew)
    if i + n_step >= len(rewards):
        power = 0
        disc_rew = 0
        for z in range(i,len(rewards)-1):
            disc_rew += rewards[z]*(gamma**power) #+
            power += 1
        if i < len(rewards)-1:
            disc_rew += values[-1]*(gamma**power)
            discnt_rewards.append(disc_rew)
    state = state[:-1]
    playerstatus = playerstatus[:-1]
    gameText = gameText[:-1]
    actions = actions[:-1]
    state = np.array(state)
    state = np.squeeze(state,axis = 1)
    playerstatus = np.array(playerstatus)
    playerstatus = np.squeeze(playerstatus,axis = 1)
    gameText = np.array(gameText)
    gameText = np.squeeze(gameText,axis = 1)
    return state,playerstatus,gameText,discnt_rewards,actions

def actor_loss(self, probs, actions, td):
    '''This function calculate the actor loss using the equation Policy Loss:
 $L = -\log(\pi(a | s)) * A(s) - \beta * H(\pi)$ 
    where  $H(\pi) = -\sum(P(x) \log(P(x)))$ '''
    e_loss = []
    p_loss= []
    log_probabilities = torch.log(probs)
    for pb,lb,a,t in zip(probs,log_probabilities,actions,td):
        policy_loss = torch.mul(torch.squeeze(lb[a]),t)
        entropy_loss = torch.negative(torch.sum(torch.multiply(pb,lb)))
        e_loss.append(entropy_loss)
        p_loss.append(policy_loss)
    p_loss = torch.stack(p_loss)
    e_loss = torch.stack(e_loss)
    p_loss = torch.mean(p_loss)
    e_loss = torch.mean(e_loss)
    loss = -p_loss - 0.0001 * e_loss
    return loss

def losses(self, states,playerstatus,gameTexts, actions, discnt_rewards):
    '''Critic and actor losses are calculated and refunded'''
    discnt_rewards = torch.tensor(discnt_rewards)
    p,_,_,_ =
self.actor.forward(torch.tensor(states).float(),torch.tensor(playerstatus).float(
), torch.tensor(gameTexts).float())
    v,_,_,_
= self.critic.forward(torch.tensor(states).float(),torch.tensor(playerstatus).fl
oat(), torch.tensor(gameTexts).float())

```

```

    v = torch.reshape(v, (len(v),))
    td = torch.subtract(discnt_rewards, v)
    a_loss = self.actor_loss(p, actions, td.detach())
    c_loss = 0.5*torch.mean(torch.pow(torch.subtract(discnt_rewards, v),2))
    return a_loss, c_loss

def share_grads_a(self, global_actor):
    '''Sets the grads of the global model equal to those of the local
model'''
    for param_a_l,param_a_g in
zip(self.actor.parameters(),global_actor.parameters()):
        param_a_g._grad = param_a_l.grad
    def share_grads_c(self,global_critic):
        '''Sets the grads of the global model equal to those of the local
model'''
        for param_c_l,param_c_g in
zip(self.critic.parameters(),global_critic.parameters()):
            param_c_g._grad = param_c_l.grad

def
Agent_Runner(total_steps,global_actor,global_critic,dfrewards,agents_reward,lock,
agents,total_episodes,optimizer_actor,optimizer_critic):
    '''A function that runs the training for 10000 episodes'''
    local_actor = actor()
    local_critic = critic()
    local_actor.load_state_dict(global_actor.state_dict())
    local_critic.load_state_dict(global_critic.state_dict())
    agentoo7 = agent(local_actor,local_critic)
    episode = 10000
    total_avgr = []
    game =
GamePAI(1,'Connan',444,444,screenfactor,True,0,False,seeded,torch_g,agents)
    game_No = 0
    s = 0
    for s in range(episode):
        s += 1
        game_No = game_No + 1
        done = False
        game_start = True
        state,playerStatus, gameText = game.initialGameState()
        state,playerStatus, gameText = agentoo7.preprocess0(state,playerStatus,
gameText,game_start)
        game_start = False
        total_reward = 0
        train_actions = []
        train_states = []
        train_playerstatus = []
        train_gametexts = []
        train_rewards = []
        train_values = []

```

```

steps = 0
while not done:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            lock.acquire()
            total_steps[0] += steps
            agents_reward[0] += total_reward
            total_episodes[0] += 1
            total_avgr = agents_reward[0]/total_episodes[0]
            episodeStat =
[total_steps[0],total_episodes[0],agents_reward[0],total_avgr]
            dfrewards.append(episodeStat)
            lock.release()
            pygame.quit()
            sys.exit()

    steps += 1
    action = agentoo7.act(state,playerStatus,gameText,agents)
    values = agentoo7.value(state,playerStatus,gameText)
    next_state,reward, next_playerStatus, next_gameText,done =
game.playerAction(action)
    action_name =
{'0:'up',1:'right',2:'down',3:'left',4:'rest',5:'hp',6:'mp',7:'attack',8:'pick'}
    print(agents,action_name[action.item()],reward,game.cave,steps)

    total_reward += reward
    if len(train_playerstatus) >= batch:
        del train_actions[0]
        del train_states[0]
        del train_playerstatus[0]
        del train_gametexts[0]
        del train_values[0]
        del train_rewards[0]
        train_actions.append(action)
        train_states.append(state)
        train_playerstatus.append(playerStatus)
        train_gametexts.append(gameText)
        train_rewards.append(reward)
        train_values.append(values)
        next_state,next_playerStatus,next_gameText =
agentoo7.preprocess0(next_state,next_playerStatus,next_gameText,game_start)
        state = next_state
        playerStatus = next_playerStatus
        gameText = next_gameText
        if done:
            game.__init__(1,'Connan',444,444,screenfactor,True,game_No,False,
seeded,torch_g,agents)
            if steps > batch+h_step:
                if steps%batch == 0 or done:
                    train_states_1,train_playerstatus_1,train_gametexts_1,discont_
rewards_1,train_actions_1=

```

```

agentoo7.preprocess1(train_states,train_playerstatus,train_gametexts,
train_rewards,train_values,train_actions,done, 0.99)
    a1,c1 =
agentoo7.losses(train_states_1,train_playerstatus_1,train_gametexts_1,
train_actions_1, discnt_rewards_1)
    print(a1,c1)
    optimizer_actor.zero_grad()
    optimizer_critic.zero_grad()
    a1.backward()
    c1.backward()
    agentoo7.share_grads_a(global_actor)
    agentoo7.share_grads_c(global_critic)
    optimizer_actor.step()
    optimizer_critic.step()
    local_actor.load_state_dict(global_actor.state_dict())
    local_critic.load_state_dict(global_critic.state_dict())

    if done:
        lock.acquire()
        agents_reward[0] += total_reward
        total_steps[0] += steps
        total_episodes[0] += 1
        print('I work '+str(agents))
        total_avgr = agents_reward[0]/total_episodes[0]
        episodeStat =
[total_steps[0],total_episodes[0],agents_reward[0],total_avgr]
        dfrewards.append(episodeStat)
        if total_episodes[0]%200 == 0 and total_episodes[0] !=0:
            d = list(dfrewards)
            d = pd.DataFrame(d, columns=['steps','episodes', 'rewards',
'average_rewards'])
            d.to_excel(dir+'\statistics_'+
str(total_episodes[0])+'.xlsx')
            torch.save(global_actor.state_dict(), dir + '\ ' +
str(total_episodes[0]) + 'actor_model.pt')
            torch.save(global_critic.state_dict(), dir + '\ ' +
str(total_episodes[0]) + 'critic_model.pt')
            lock.release()

if __name__ == '__main__':
    '''The multiprocessing part of the script. This code starts 8 Agent_Runner
functions, one for each processor core'''
    if exists(dir + '\steps.txt'):
        f = open(dir+'\steps.txt','r')
        total_steps_int = int(f.read())
        f.close()
    if exists(dir + '\Total_rewards.txt'):
        f = open(dir+'\Total_rewards.txt','r')
        total_rewards = float(f.read())
        f.close()

```

```

if exists(dir + '\episodes.txt'):
    f = open(dir+'\episodes.txt','r')
    episodes = int(f.read())
    f.close()
num_processes = mp.cpu_count()
total_steps = mp.Manager().list([0])
total_steps[0]=total_steps_int
dfrewards = mp.Manager().list()
agents_reward = mp.Manager().list([0])
agents_reward[0]= total_rewards
total_episodes = mp.Manager().list([0])
total_episodes[0] = episodes
lock = mp.Lock()
global_actor = actor()
global_critic = critic()
print(exists(dir + '\ ' + str(total_episodes[0]) + 'critic_model.pt'))
if exists(dir + '\ ' + str(total_episodes[0]) + 'actor_model.pt'):
    global_actor.load_state_dict(torch.load(dir + '\ ' +
str(total_episodes[0]) + 'actor_model.pt'))
    global_actor.train()
    print('Global actor model is loaded')
if exists(dir + '\ ' + str(total_episodes[0]) + 'critic_model.pt'):
    global_critic.load_state_dict(torch.load(dir + '\ ' +
str(total_episodes[0]) + 'critic_model.pt'))
    global_critic.train()
    print('Global critic model is loaded')
global_actor.share_memory()
global_critic.share_memory()
optimizer_actor = my_optim.SharedAdam(global_actor.parameters(), lr=1e-5)
optimizer_critic = my_optim.SharedAdam(global_critic.parameters(), lr=1e-5)
processes = []
for agents in range(num_processes):
    p = mp.Process(target=Agent_Runner,
args=(total_steps,global_actor,global_critic,dfrewards,agents_reward,lock,agents,
total_episodes,optimizer_actor,optimizer_critic))
    p.start()
    processes.append(p)
for p in processes:
    p.join()
torch.save(global_actor.state_dict(), dir + '\ ' + str(total_episodes[0]) +
'actor_model.pt')
torch.save(global_critic.state_dict(), dir + '\ ' + str(total_episodes[0])
+ 'critic_model.pt')
d = list(dfrewards)
d = pd.DataFrame(d, columns=['steps','episodes', 'rewards',
'average_rewards'])
d.to_excel(dir+'\statistics_'+ str(total_episodes[0]) + '.xlsx')
f = open(dir+'\steps.txt','w')
f.write(str(total_steps[0]))
f.close()
f = open(dir+'\Total_rewards.txt','w')

```

```
f.write(str(agents_reward[0]))  
f.close()  
f = open(dir+'\episodes.txt','w')  
f.write(str(total_episodes[0]))  
f.close()
```



**ABBREVIATIONS – ACRONYMS**

A3C	Asynchronous Advantage Actor-Critic
AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
DP	Dynamic Programing
DQN	deep Q-network
FPS	First Person Shooter
GPI	General Policy Iteration
GPU	Graphic Processing Unit
HP	Hit Points
ID	Identity
LSTM	Long Short-Term Memory
MC	Monte Carlo
MDP	Markov Decision Process
ML	Machine Learning
MP	Mana Points
NN	Neural Networks
POMDP	Partially Observable Markov Decision Process
ReLU	Rectifier Nonlinearity
RND	Random Network Distillation
RNN	Recurrent Neural Network
RPG	Role Playing Games
SARSA	Sate Action Reward State Action
SGD	Stochastic Gradient Descent
TD	Temporal Difference
XP	Experience Points

## REFERENCES

- [1] Mnih, Volodymyr & Kavukcuoglu, Koray & Silver, David & Rusu, Andrei & Veness, Joel & Bellemare, Marc & Graves, Alex & Riedmiller, Martin & Fidjeland, Andreas & Ostrovski, Georg & Petersen, Stig & Beattie, Charles & Sadik, Amir & Antonoglou, Ioannis & King, Helen & Kumaran, Dharshan & Wierstra, Daan & Legg, Shane & Hassabis, Demis. (2015). Human-level control through deep reinforcement learning. *Nature*. 518. 529-33. [10.1038/nature14236](https://doi.org/10.1038/nature14236).
- [2] Silver, David & Huang, Aja & Maddison, Christopher & Guez, Arthur & Sifre, Laurent & Driessche, George & Schrittwieser, Julian & Antonoglou, Ioannis & Panneershelvam, Veda & Lanctot, Marc & Dieleman, Sander & Grewe, Dominik & Nham, John & Kalchbrenner, Nal & Sutskever, Ilya & Lillicrap, Timothy & Leach, Madeleine & Kavukcuoglu, Koray & Graepel, Thore & Hassabis, Demis. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*. 529. 484-489. [10.1038/nature16961](https://doi.org/10.1038/nature16961).
- [3] Andriy Burkov., *The Hundred-Page Machine Learning Book*.
- [4] Sutton, R. & Barto, A. *Reinforcement Learning: An Introduction* (MIT Press, 2018).
- [5] Bellman, R. E. (1957a). *Dynamic Programming*. Princeton University Press, Princeton
- [6] Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. Ph.D. thesis, University of Cambridge
- [7] Wiering, M., & Van Otterlo, M. (2012). *Reinforcement Learning: State of the Art*. Springer. <https://doi.org/10.1007/978-3-642-27645-3>
- [8] E. Barnard, "Temporal-difference methods and Markov models," in *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 23, no. 2, pp. 357-365, March-April 1993, doi: 10.1109/21.229449..
- [9] Jayawardana, Rahul & Bandaranayake, Thusitha. (2021). ANALYSIS OF OPTIMIZING NEURAL NETWORKS AND ARTIFICIAL INTELLIGENT MODELS FOR GUIDANCE, CONTROL, AND NAVIGATION SYSTEMS.
- [10] Cybenko, G. Approximation by superpositions of a sigmoidal function. *Math. Control Signal Systems* 2, 303–314 (1989). <https://doi.org/10.1007/BF02551274>
- [11] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.
- [12] Rumelhart, D., Hinton, G. & Williams, R. Learning representations by back-propagating errors. *Nature* 323, 533–536 (1986). <https://doi.org/10.1038/323533a0>
- [13] Chollet, Francois. *Deep Learning with Python*. Manning Publications, 2017.
- [14] Hochreiter, Sepp & Schmidhuber, Jürgen. (1997). Long Short-term Memory. *Neural computation*. 9. 1735-80. [10.1162/neco.1997.9.8.1735](https://doi.org/10.1162/neco.1997.9.8.1735).
- [15] I. Goodfellow, Y. Bengio, A. Courville, and F. Bach, *Deep Learning*. Cambridge, MA: MIT Press, 2016
- [16] Hausknecht, Matthew & Stone, Peter. (2015). Deep Recurrent Q-Learning for Partially Observable MDPs.
- [17] Mnih, Volodymyr & Badia, Adrià & Mirza, Mehdi & Graves, Alex & Lillicrap, Timothy & Harley, Tim & Silver, David & Kavukcuoglu, Koray. (2016). Asynchronous Methods for Deep Reinforcement Learning.
- [18] Lample, Guillaume & Chaplot, Devendra. (2016). Playing FPS Games with Deep Reinforcement Learning.
- [19] Williams, R.J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach Learn* 8, 229–256 (1992). <https://doi.org/10.1007/BF00992696>
- [20] Williams, Ronald & Peng, Jing. (1991). Function Optimization Using Connectionist Reinforcement Learning Algorithms. *Connection Science*. 3. 241-. [10.1080/09540099108946587](https://doi.org/10.1080/09540099108946587).
- [21] Kingma, D.P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980.
- [22] Duchi, John & Hazan, Elad & Singer, Yoram. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*. 12. 2121-2159.
- [23] Tieleman, T. and Hinton, G. (2012) Lecture 6.5-rmsprop: Divide the Gradient by a Running Average of Its Recent Magnitude. *COURSERA: Neural Networks for Machine Learning*, 4, 26-31.
- [24] Niu, Feng & Recht, Benjamin & Ré, Christopher & Wright, Stephen. (2011). HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. *NIPS*. 24.
- [25] Osband, Ian & Blundell, Charles & Pritzel, Alexander & Roy, Benjamin. (2016). Deep Exploration via Bootstrapped DQN.
- [26] Burda, Yuri & Edwards, Harrison & Storkey, Amos & Klimov, Oleg. (2018). Exploration by Random Network Distillation.