****

School of Software Engineering: Intelligent Systems

# **Mastering a zero-sum asymmetric game using deep neural networks**

A project report submitted toward the degree of

Master of Science in Intelligent Systems

**Student name: Eliav Shalelashvili**

**Supervisor: Dr Yehudit Aperstein**

**Date: 08/10/2021**

****

**בית הספר להנדסת תוכנה**

**החוג למערכות תבוניות**

**שם העבודה:**

**התמחות במשחק סכום אפס אסימטרי באמצעות רשתות עצביות עמוקות**

***חיבור על עבודת גמר למילוי חלקי של הדרישות לקבלת***

***תואר M.Sc. במערכות תבוניות***

**שם הסטודנט: אליאב שללאשוילי**

**שם המנחה: דר' יהודית אפרשטיין**

**תאריך הגשה: 08/10/2021**

## Abstract

In this research we are dealing with solutions of a two-player zero-sum game with asymmetric information. The goal of each player is to maximize its reward. The game is a civilian version of a known security problem.

We formulate the problem as a problem in the field of Reinforcement Learning (RL) and we will examine problems and solutions from there.

Our problem is characterized by a particularly large state space in relation to known problems in the field of RL. While the state space complexity of the games Tic-Tac-Toe Chess and Go stands at 4, 43 and 172 respectively, the state space complexity in our game, for the discrete state space only, stands at 270. In addition, considering the game tree complexity, while the number of possible actions in Go stands on 200 on average, in our game the number stands at 900 possible actions per turn.

As part of the project, we conducted a literature review of similar problems and different ways to solve them, a review that formulated our path to action. Which is, the implementation of a basic algorithm from the field of RL, called Double Deep Q-Network (or DDQN), another algorithm from the field of general computer science search methods called Monte Carlo Tree Search (or MCTS) and finally a state-of-the-art algorithm called AlphaZero.

In addition, we built the environment on which we ran and taught the learning algorithms in a way that would be generic and configurable for multiple runs of learning agents for the use of researchers who are dealing with similar problems.

A collaboration was made with one of the aerospace industries in order to prove the ability of algorithms from the field of Reinforcement Learning (RL) in dealing with a problem well known to the mentioned industry. The environment we built is based on ideas that came up to simulate their case.

The learning agents were tested on a variety of games (different board sizes) and in different configurations (hyper-parameter optimization). At first, we tested the learning agents against a group of simple agents with a constant or random strategy.

The DDQN and AlphaZero based agents achieved convergence in a variety of scenarios tested and in contrast the MCTS algorithm did not converge at all due to RAM limitations of the machine.

Additionally, we obtained a stable convergence (similar result for different sets of parameters) of the DDQN agent in all scenarios against the constant agents, all along achieving good results against the random agents only on small boards. In contrast we managed to achieve full convergence of AlphaZero in those random scenarios in the large boards but this convergence is unstable and does not even exist in some of the scenarios against the constant agents.

We then conducted a competition between the learning agents: DDQN and AlphaZero. This competition has led to a significant victory for DDQN in most hits, this victory is not intuitive and did not come as expected but can be explained by the massive amount of AlphaZero parameters in comparison with DDQN and by the insufficient number of episodes we ran during training due to long run time of the implementation we chose.

## Table of Contents

[**Mastering a zero-sum asymmetric game using deep neural networks** 1](#_Toc84096554)

[1. Abstract 2](#_Toc84096555)

[2. Table of Contents 3](#_Toc84096556)

[3. Figure List 5](#_Toc84096557)

[4. Table List 7](#_Toc84096558)

[5. Introduction 8](#_Toc84096559)

[6. Literature Review 9](#_Toc84096560)

[*6.1* *Insights from the GVG-AI Competition* 9](#_Toc84096561)

[*6.2* *Monte Carlo Tree Search* 10](#_Toc84096562)

[*6.3* *From AlphaGo to AlphaZero* 10](#_Toc84096563)

[7. Our Game – Light vs Zombies 14](#_Toc84096564)

[*7.1* *Assumptions* 14](#_Toc84096565)

[*7.2* *Stochastic game* 15](#_Toc84096566)

[*7.3* *Players* 15](#_Toc84096567)

[*7.4* *Game rules* 16](#_Toc84096568)

[8. Building the Simulation 18](#_Toc84096569)

[8.1 Framework Implementation 18](#_Toc84096570)

[8.2 Implemented Agents 18](#_Toc84096571)

[8.3 Run the Simulation 19](#_Toc84096572)

[8.4 Simulation Test 21](#_Toc84096573)

[9. Methodology 28](#_Toc84096574)

[9.1 Double Deep Q-Network Evaluation 28](#_Toc84096575)

[9.2 Learning Based Monte Carlo Tree Search 37](#_Toc84096576)

[9.3 Learning Based AlphaZero Algorithm 41](#_Toc84096577)

[10. Competing AlphaZero and DDQN 52](#_Toc84096578)

[10.1 Constructing the Composite Agent 53](#_Toc84096579)

[10.2 Final Evaluation 53](#_Toc84096580)

[10.3 The Competition 54](#_Toc84096581)

[11. Conclusions and Next Steps 67](#_Toc84096582)

[12. References 69](#_Toc84096583)

[13. Appendix A – Elaboration of Related Literature 70](#_Toc84096584)

[13.1 Reinforcement Learning 70](#_Toc84096585)

[*13.2* *Stochastic Games* 70](#_Toc84096586)

[*13.3* *Nash Equilibrium in SGs* 71](#_Toc84096587)

[*13.4* *Learning in SGs* 72](#_Toc84096588)

## Figure List

[Figure 1 - AlphaGo overview 11](#_Toc83655161)

[Figure 2 - Resnet contribution over traditional CNNs 12](#_Toc83655162)

[Figure 3 - AlphaGo Zero training characteristics 12](#_Toc83655163)

[Figure 4 - Example of grid set-up with light mark of 3x3 and 8 zzombies 15](#_Toc83655164)

[Figure 5- Frameworks' Main configuration example 19](#_Toc83655165)

[Figure 6 - Abstract methods every agent should implement in order to join the Framework 20](#_Toc83655166)

[Figure 7 - Two simple lines of code to run the Framework and get results 20](#_Toc83655167)

[Figure 8 - Summary - three steps to run our framework 20](#_Toc83655168)

[Figure 9 – Double Deep Q-Network architecture and flow 21](#_Toc83655169)

[Figure 10 - Epsilon greedy values 22](#_Toc83655170)

[Figure 11 – environment set-up for zombie Player performance check with optional actions 23](#_Toc83655171)

[Figure 12 – Zombie Player actions distribution along different ranges of episodes 24](#_Toc83655172)

[Figure 13 – Total zombies survived vs. the episodes (blue) with its moving average (orange) 24](#_Toc83655173)

[Figure 14 - environment set-up for light Player performance check with optional actions 25](#_Toc83655174)

[Figure 15 - Light Player actions distribution along different ranges of episodes 26](#_Toc83655175)

[Figure 16 - Total zombies survived vs. the episodes (blue) with its moving average (orange) 27](#_Toc83655176)

[Figure 17 - Example of reward per episode graph, DDQN plays Zombie and Single Action plays Light 29](#_Toc83655177)

[Figure 18 - Scenario Evaluation by Average Test Reward 30](#_Toc83655178)

[Figure 19 - Heat-Map of the Average Test Rewards of all the scenarios that DDQN Agent plays Zombie and Single Action Agent plays Light 30](#_Toc83655179)

[Figure 20 - A summary of all the scenarios that DDQN plays Zombie by Heat-Maps of the Average Test Reward 31](#_Toc83655180)

[Figure 21 - Comparing the results of the DDQN agent as the Zombie Player, with the best parameters over the different four simple competitors 33](#_Toc83655181)

[Figure 22 - Heat-Map of the Average Test Rewards of all the scenarios that DDQN Agent plays Light and Single Action Agent plays Zombie 34](#_Toc83655182)

[Figure 23 - A summary of all the scenarios that DDQN plays Light by Heat-Maps of the Average Test Reward 35](#_Toc83655183)

[Figure 24 - Comparing the results of the DDQN agent as the Light Player, with the best parameters over the different four simple competitors 37](#_Toc83655184)

[Figure 25 - Step of Monte Carlo tree search 38](#_Toc83655185)

[Figure 26 - NN Architecture. Red and blue arrows indicate convolutional and fully Connected layers respectively 43](#_Toc83655186)

[Figure 27 - A summary of all the scenarios that AlphaZero plays Zombie. Heat-Maps of the Average Test Reward 46](#_Toc83655187)

[Figure 28 - Comparing the results of the AlphaZero agent as the Zombie Player, with the best parameters over the different four simple competitors 48](#_Toc83655188)

[Figure 29 - A summary of all the scenarios that AlphaZero plays Light. Heat-Maps of the Average Test Reward 49](#_Toc83655189)

[Figure 30 - Comparing the results of the AlphaZero agent as the Light Player, with the best parameters over the different four simple competitors 51](#_Toc83655190)

[Figure 31 - Action and Rewards distribution graph 54](#_Toc83655191)

[Figure 32 - Rewards over Episodes, board 10x10 55](#_Toc83655192)

[Figure 33 - Action and Reward Distributions, board 10x10 56](#_Toc83655193)

[Figure 34 - Rewards over Episodes, board 20x20 57](#_Toc83655194)

[Figure 35 - Action and Reward Distributions, board 20x20 58](#_Toc83655195)

[Figure 36 - Rewards over Episodes, board 30x30 59](#_Toc83655196)

[Figure 37 - Action and Reward Distributions, board 30x30 60](#_Toc83655197)

[Figure 38 - Rewards over Episodes, board 10x10 61](#_Toc83655198)

[Figure 39 - Action and Reward Distributions, board 10x10 62](#_Toc83655199)

[Figure 40 - Rewards over Episodes, board 20x20 63](#_Toc83655200)

[Figure 41 - Action and Reward Distributions, board 20x20 64](#_Toc83655201)

[Figure 42 - Rewards over Episodes, board 30x30 65](#_Toc83655202)

[Figure 43 - Action and Reward Distributions, board 30x30 66](#_Toc83655203)

[Figure 44 – Game tree when there are two actions by player 71](#_Toc83655204)

## Table List

[Table 1 – Learning parameters while evaluating the zombie Player 22](#_Toc83652787)

[Table 2 – learning parameters while evaluating the light Player 25](#_Toc83652788)

[Table 3 – DDQN Evaluation: Game Scenarios 28](#_Toc83652789)

[Table 4 - Best Configurations of all Scenarios in which the DDQN plays Zombie 31](#_Toc83652790)

[Table 5 - Best Configurations of all Scenarios in which the DDQN plays Light 35](#_Toc83652791)

[Table 6 – DDQN Evaluation: Game Scenarios 44](#_Toc83652792)

[Table 7 - Best Configurations of all Scenarios in which AlphaZero plays Zombie 46](#_Toc83652793)

[Table 8 - Best Configurations of all Scenarios in which AlphaZero plays Light 49](#_Toc83652794)

[Table 9 - Summary of competing DDQN Agent vs. Single, Double and Gaussian Agents 52](#_Toc83652795)

[Table 10 – DDQN trained with Composite agents, evaluation vs. all Simple Agents 53](#_Toc83652796)

[Table 11 - AlphaZero trained with Composite Agent, evaluation vs. all Simple Agents 54](#_Toc83652797)

## Introduction

One of the primary goals of the field of artificial intelligence (AI) is to produce fully autonomous agents that interact with their environments to learn optimal behaviors, improving over time through trial and error. Crafting AI systems that are responsive and can effectively learn has been a long-standing challenge, ranging from robots, which can sense and react to the world around them, to purely software-based agents, which can interact with natural language and multimedia.

Toward this project, we were presented with a problem that concerns one of the defense industries, in which we were asked to build a family of stochastic games which characterized with extreme state-space complexity. Each one of the stochastic games consists of two players with non-symmetric action space and exposed to different types of information.

Our goal is to implement carefully selected algorithms and examine whose performance is superior in each game. We aim to achieve successful results with three types of agents: one that learns using a traditional Reinforcement Learning algorithm, another one from the tree-search area and the last that uses state-of-the-art algorithm from the field of Reinforcement Learning. The combination and analysis of agents from different domains of research will provide us a broad understanding of the field that will produce the most successful agents for our family of games.

In the following document we present the final project of the Intelligent Systems program at Afeka College of Engineering.

During the project, we've built a framework of two-player-games for developing and comparing Reinforcement Learning algorithms. It is able to evaluate the learning process of the agents, compare between learning agents and more. We will discuss the subject in detail in chapter ‎‎8.

Furthermore, we have implemented two algorithms from the Reinforcement Learning domain called: Double-Deep-Q-Network [17] and AlphaZero [14]. Our framework provides each player the ability to play by an algorithm, DDQN, AlphaZero or any other. In chapter ‎8.4 we test the stability and performance of the environment. In chapter ‎9, we provide in-depth analysis of the learning algorithms we implemented, in manners of achieving optimal policy in various scenarios.

At the same time of building the algorithms we mentioned, we implemented another algorithm from the field of search algorithms, called Monte-Carlo-Tree-Search (see Learning Based Monte Carlo Tree Search) in favor of comparison with algorithms from the Reinforcement Learning field. The MCTS algorithm did not lead to any success due to RAM limitations and the fact that the state-space in the game is too large to withstand the search tree it builds.

Finally, after implementing and testing the performance of all the algorithms individually, we trained them in several different board sizes (square boards of 10, 20 and 30) to allow a competition between them (see chapter ‎10).

## Literature Review

The field of agents that master the learning of a particular game, or agents seeking for the highest score over a set of games, significantly increased in the past few years. Since we are not facing with a studied problem nor a known game, we will divide our review into three sections:

1. Insights from the GVG-AI Competition
2. Monte Carlo Tree Search
3. From AlphaGo to AlphaZero

All along with elaboration of the potential contribution of each topic to our research due to the successes of similar problems and previous research of the domain.

* Note that we have extra elaboration on basic RL concepts and ideas we won't apply in the scope of the project, all in ‎14 (Appendix A – Elaboration of Related Literature).

### *Insights from the GVG-AI Competition*

The General Video Game AI competition (GVGAI) was created in order to test these general agents on a multitude of real-time games (both stochastic and deterministic) under the same conditions and constraints. It has received significant international attention in the seven years it has been running and has allowed for many interesting algorithms to be tested on a large number of problems.

The GVG-AI Competition explores the problem of creating controllers for general video game playing, in such platform, researches have an opportunity to test their agents via participating in the competitions.

The past few years have led to some great RL algorithms like 'MaastCTS' and 'OLMCTS' [11], both based on the MCTS [12] algorithm. All of these have proven themselves in the competitions, therefore it might be useful and beneficial to our research.

The platform of GVG-AI lets the competitors test their algorithms on some environments built specially for them (by DeepMind) to challenge and push them to their limits. By going over all the proposed environments, we found some games with a lot of resemblance to us, like in our research, there were two players in a zero-sum stochastic game alongside the fact that the action space of the agents is much like ours, for the illustration in this paper we want to elaborate on two games that might be of our interest:

**Ghostbusters**, a version of the known Atari game with improvements to satisfy the competition demands. The game contains two players: one player is the ghost and the other is the hunter.

The ghost can pass through walls and wraps around the level aiming to either avoid dying or catch the hunter. The hunter's goal is to avoid the ghost which can hurt him, shoots missiles on it while he can move faster than the ghost.

The algorithms that achieved the best results in the competition are called: MCTS and [MaastCTS2](https://github.com/DennisSoemers/MaastCTS2/tree/master/Two-Player/src/MaastCTS2), both are a variant of an agent that learns by building Monte Carlo Tree Search.

The game Ghostbusters briefly described above reminds our game in many manners: the similarity of the actions the agents take, the ghost that is capable of moving around the grid while targeting to catch the hunter, much like our Light Player (described later). The hunter as well - shoots missiles in many directions to try and catch the ghost, just like our Zombie Player (described later).

Another game is called **Upgrade-X**, its environment contains two players who are both located in a two-dimensional square they can't leave. Each player has some laser cannons, which they can move around. Getting hit by the laser cannon of the opponent hurts the player and makes him lose some health points. The winner is the player that survives with the most points at the end of the game.

The algorithms that achieved the best results in the competition are called: OLMCTS, SARSA-UCT and [MaastCTS2](https://github.com/DennisSoemers/MaastCTS2/tree/master/Two-Player/src/MaastCTS2). All of which are also a variant of an agent that learns by building Monte Carlo Tree Search.

Like in the previous game, the Upgrade-X games have a bunch of similarities with our game: two agents moving around with laser cannons are responsible of the canon direction and their goal is to maximize their health/strength points until the end of the game, a process that reminds a lot our Zombie Agent (described later) that is responsible to his zombie's direction and velocities alongside the goal of maximizing their strength throughout the game.

### *Monte Carlo Tree Search*

The GVG-AI competition brought into the RL field brilliant algorithms and ideas. Most of whom have one thing in common – Monte Carlo Tree Search Algorithm.

The focus of MCTS is on the analysis of the most promising moves, expanding the [search tree](https://en.wikipedia.org/wiki/Search_tree) based on [random sampling](https://en.wikipedia.org/wiki/Monte_Carlo_method) of the search space. The application of Monte Carlo tree search in games is based on many playouts, also called rollouts. In each playout, the game is played out to the very end by selecting moves at random. The final game result of each playout is then used to weight the nodes in the game tree so that better nodes are more likely to be chosen in future playouts.

The MCTS algorithm is an algorithm from the search tree family, in classical computer science. Thus, its assessment of state-values is maintained in a structure that develops during processing and playing, which is something that can lead to the requirement of enormous space complexity.

### *From AlphaGo to AlphaZero*

Having briefly explained Monte Carlo Tree Search ideas, we will expand the conversation to a family of outbreaking algorithms that, for today, are considered State of the Art: Known as: Alpha Go, AlphaGo Zero, and Alpha Zero. These are algorithms developed by DeepMind, whom are Google's top artificial intelligence research group.

#### AlphaGo

AlphaGo [16] is the first paper in the series, showing that Deep Neural Networks could play the game of Go by predicting a **policy** (mapping from state to action) and **value estimate** (probability of winning from a given state). These policy and value networks are used to enhance tree-based lookahead search by selecting which actions to take from given states and which states are worth exploring further.

AlphaGo uses 4 Deep Convolutional Neural Networks, 3 policy networks and a value network. 2 of the policy networks are trained with supervised learning on expert moves.

Supervised learning describes loss functions consisting of some kind of . In this case, the is the action the policy network predicted from a given state, and the y is the action the expert human player had taken in that state.

The rollout policy is a smaller neural network that takes in a smaller input state representation as well. As a consequence of this, the rollout policy has a significantly lower modeling accuracy of expert moves than the higher capacity network. However, the rollout policy network’s inference time (time to make a prediction of action given state) is 2 microseconds compared to 3 milliseconds with the larger network, making it useful for Monte Carlo Tree Search simulations.

The SL policy network is used to initialize the 3rd policy network which is trained with self-play and policy gradients. Policy gradients describe the idea of optimizing the policy directly with respect to the resulting rewards, compared to other RL algorithms that learn a value function and then make the policy greedy with respect to the value function. The policy gradient trained policy network plays against previous iterations of its own parameters, optimizing its parameters to select the moves that result in wins. The **self-play dataset** is then used to train a value network to predict the winner of a game from a given state.

The final workhorse of AlphaGo is the combination of policy and value networks in MCTS, depicted below:

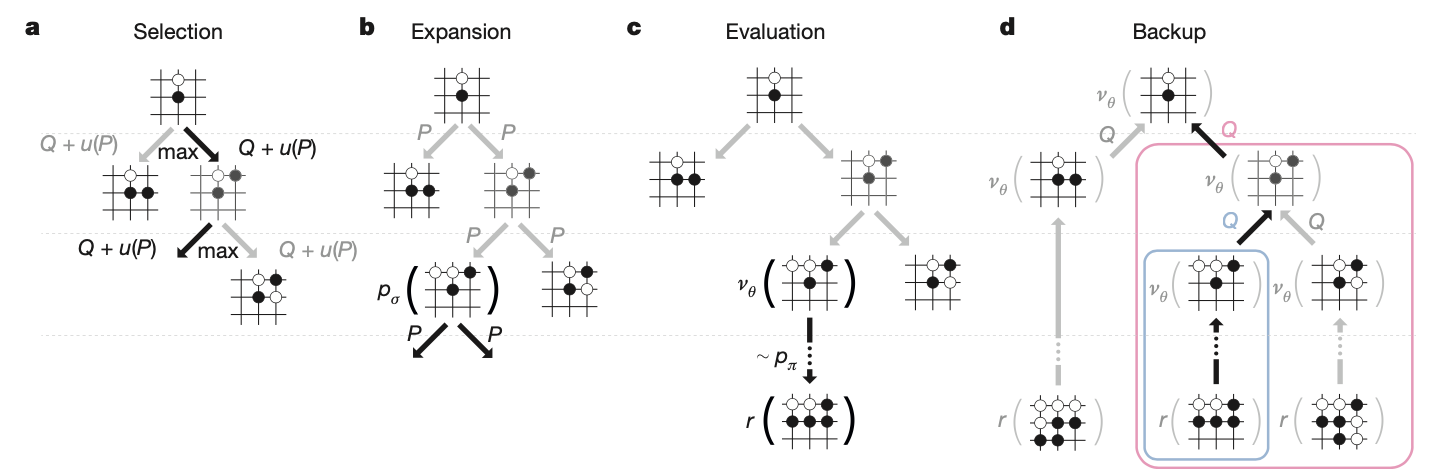


Figure 1 - AlphaGo overview

#### AlphaGo Zero

AlphaGo Zero [15] significantly improves the AlphaGo algorithm by making it more general and starting from **“Zero” human knowledge**. AlphaGo Zero avoids the supervised learning of expert moves initialization and combines the value and policy network into a single neural network. This neural network is scaled up as well to utilize a ResNet compared to a simpler convolutional network in AlphaGo. The contribution of the ResNet performing both value and policy mappings is evident in the diagram below comparing the dual task ResNet to separate task CNNs:



Figure 2 - Resnet contribution over traditional CNNs

One of the most interesting characteristics of AlphaGo Zero is the way it trains its policy network using the action distribution found by MCTS, depicted below:

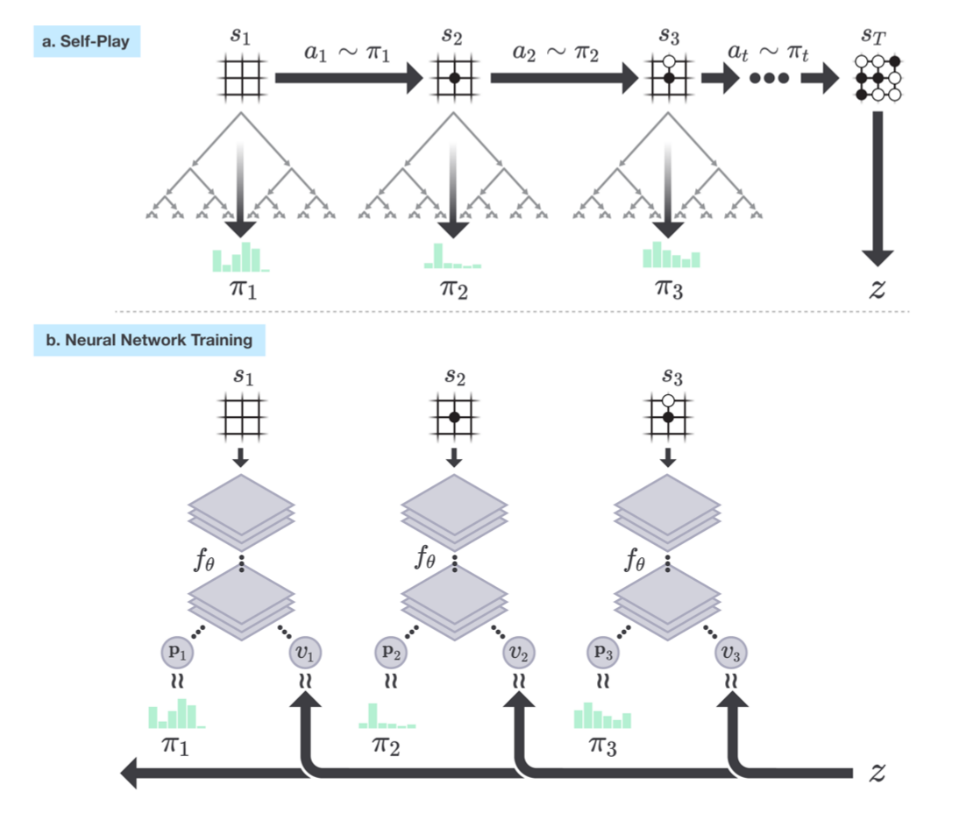


Figure 3 - AlphaGo Zero training characteristics

The MCTS trains the policy network by using it as supervision to update the policy network. This is a clever idea since MCTS produces a better action distribution through lookahead search than the policy network’s instant mapping from state to action.

#### AlphaZero

AlphaZero [14] is the first step towards generalizing the AlphaGo family outside of Go, looking at changes needed to play Chess and Shogi as well. This requires formulating **input state and output action** representations for the residual neural network.

AlphaZero also makes some more subtle changes to the algorithm such as the way the self-play champion is crowned and the eliminations of data augmentation from Go board games such as reflections and rotations.

#### MuZero

MuZero presents a very powerful generalization to the algorithm of AlphaZero that allows it to learn without a perfect simulator. Chess, Shogi, and Go are all examples of games that come with a perfect simulator, if you move your pawn forward 2 positions, you know exactly what the resulting state of the board will be.

Since our case consider a perfect simulator, we do not require an algorithm such as MuZero.

## Our Game – Light vs Zombies

Our learning model is based on a two-dimensional game grid on which the zombies live. In this chapter we'll expand beyond the idea and intuition, we will formally review the rules of the agents, and define the principles of stochastic game in our particular case of two-player zero sum game.

Imagine a board of zombies approaching from some locations in the left side of the board towards the right side of the board. Above all that, there is a light that can be positioned anywhere on the board.  
The two agents will be called 'Zombie Player' and 'Light Player' the Zombie Player is responsible of positioning the zombies in the left side and determine their initial angle and speed that will stay constant for each zombie. On the other hand, the Light Player decides where to project his light in every turn. Each zombie that leaves the left side of the board and goes under the light of the Light Player is damaged and his strength meter is lowered by some value.  
In general, the goal of the Zombie Player/Light Player is to maximize/minimize the strength of the zombies that are reaching the right side of the board.

### *Assumptions*

we'll start with some basic assumptions relating the environment and the agent's movement.

Throughout the game, the time and space will be considered discrete while the system operates in discrete time over a horizon T. The system area is represented by N-by-M grid with integer coordinates

Around the board will revolve two types of agents, light and zombies. A zombie and light marking might take coordinates on the integer grid (cells), while at each time moment a zombie can move one cell in right direction.

Furthermore, the light-mark is represented by a square area A-by-A.

Figure 4 - Example of grid set-up with light mark of 3x3 and 8 zzombies

### *Stochastic game*

As stated above, we deal with two player stochastic game, we will now define the problem we are facing similar to what is stated in the section ‎14.2.

Our stochastic game is defined by the tuple where:

* + 1. is the number of players
    2. is the transition function
    3. is the action set for the player
    4. is the discount factor (for now )
    5. is the reward function for player i

We deal with stochastic two players zero-sum game, i.e.

with limited information on one side (asymmetric information).

### *Players*

#### Zombie Player

The first player is the Zombie Player, its objective is to maximize the average lifetime of zombies. One way of defining the above is the sum lifetime of all zombies and average over all game rounds.  
In each round, the Zombie Player must decide on a coordinate Y, where the next zombie should start. Action is an integer number from 0 to N (N is the size of the board). Therefore, the actions it can take are the set: .

The Zombie Player bases its decisions on the available information which is the matrix of the zombies' locations N-by-M. Each cell of the matrix is 0 (no zombie) or 1 (occupied by zombie). Denote the collection of variables (i.e. observations, actions) available to player 1 at time t by

, where

Denote a subset of all observations until time t and actions until time t-1 by . Therefore contains a set of t N-by-M matrices and a history of choices from the set .

#### Light Player

Another player in the game is the Light Player. Its objective is to minimize the average lifetime of zombies, that is defined similarly to the definition of the Zombie Player: the sum lifetime of all zombies and average over all game rounds.  
In each round of the game, the Light Player has to choose where to put the center of the light (x, y coordinates):

Thus, the space of action in its possession is:

Again, similarly to the Zombie Player, the Light Player must choose an action based on some available information it has like the N-by-M matrix of zombies and zombie's history as defined above (see Zombie Player). In addition, it has information on the strength of the zombies at any given moment and the history of light locations. To sum up, all the information available to him is:

* A tensor (two 2-d matrix) of:
  + A zombie location matrix, such that
  + A zombie strength matrix, with non-empty cells at zombie location, such that

* The mark (light) at time t, , i.e., the player’s action

Therefore, the available information at time t is

Denote a subset of all observations until time t and actions until time t-1 by

### *Game rules*

The game has discrete clock, in each clock tick:

* The Zombie and Light Players decide where a new zombie will appear and where the new light mark is placed, accordingly.  
  All previous existing zombies are moved right cell, i.e. if for   
  In general,
* Each zombie that inside marked region (light) got additional hit of the amount c. Meaning the hit points are increased by c (default value is )
* Each remaining zombie heal itself by multiplying hit point by (1-epsilon) factor. Thus:
* Zombies that go over the right boundary of the board, disappear and provide reward for the Zombie and Light players.
* For each zombie mentioned above (that goes over the right boundary), we calculate a concave and monotonically-increasing utility function , based on its hit points value. produce values between . Each zombie with score that is equal or greater than 1 is considered disabled and both Zombie and Light players get rewards of 0.
* For the rest of the zombies, with U-score between 0 and 1. We calculate their probability to be disabled with Bernoulli distribution with parameter .
* Each Zombie that wasn’t disabled by the process above, gains a positive reward of +1 for the Zombie player and negative reward of -1 for the Light player.
* Note: The healing process and utility function U we mentioned above are responsible for meeting the requirement that each zombie should be hit multiple times.

## Building the Simulation

Building a simulation for Reinforcement learning purposes is mostly a manner of creating an environment and allow a variety of agents to access it take actions in it and gain some rewards from it.

In this section we will elaborate the characteristics of the environment, the main components inside it and provide details about the type of agents and algorithms we implemented.

As stated in ‎5, the project first came up as a requirement from one of the Israeli Air Industries, to build and evaluate learning agents on top of playing a game with certain rules. The basic requirement from the environment was to examine the success of learning agents versus simpler agents (who play by a fixed, uniform strategy, etc.). In addition, as detailed in ‎7, the game is between two players, the Zombie Player and the Light Player, with an opposite reward based on the number of zombies survived at the final line of the board. To satisfy all these requirements, in this project we built the game as defined in ‎7, while we create an independency between the Player type (Zombie or Light) and the algorithm it is playing by.

### Framework Implementation

Building an environment that is able to communicate with an interface of generic type of any agent, turned to be an integrated and significant part of the project:

* More than 40 classes, abstract classes and config files.
* Four simple agents (based on known distribution).
* Three algorithmic agents from two different domains (Reinforcement Learning and Tree Search).

Furthermore, all the results we are going to discuss from now on (chapters ‎0, ‎0 and forth) are automatically generated by the framework after each scenario and batch of scenarios.

### Implemented Agents

As mentioned above, we implemented four simple agents and three algorithmic agents. From a given set of possible actions.

The Simple Agents are:

* Single Action - picks the first action
* Double Action - picks the first and middle actions
* Uniform - picks a uniformly action
* Gaussian - picks an action from the normal distribution with:
  + Mean:
  + Standard deviation:

The algorithmic agents are:

* Double-Deep-Q-Network Agent. More details provided at: "The Algorithm – Double Deep Q Network"
* Monte-Carlo-Tree-Search Agent. More details provided at: "MCTS - Implementation and Config"
* Alpha-Zero Agent. More details provided at: "
* Learning Based AlphaZero Algorithm"

### Run the Simulation

To be able to run the simulation, we first must define our desired configurations:

* Interactive mode – Boolean, whether we want to visualize the environment
* Display width/height of the visualization
* Number of training and validation episodes
* Number of zombies per episode
* Light size
* Board width/height
* Maximum hit points
* Heal ratio

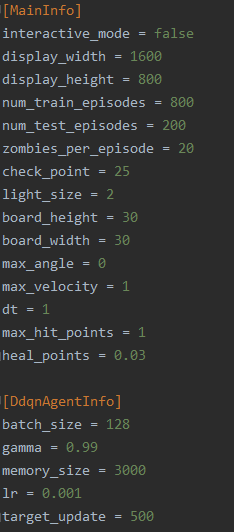


Figure 5- Frameworks' Main configuration example

Next, we must define the minds behind the competitors:

* For example, we want to let the Double Deep Q-Network agent play as the light player against a zombie player that acts according to the uniform distribution
* Each agent must implement some basic methods in order to participate the game:

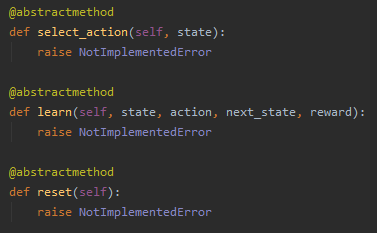


Figure 6 - Abstract methods every agent should implement in order to join the Framework

Finally, we can run the framework with the agents we have built:

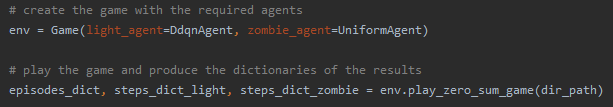


Figure 7 - Two simple lines of code to run the Framework and get results

To sum up:



Figure 8 - Summary - three steps to run our framework

### Simulation Test

Usually in RL projects, we will use some known and tested environment, since that’s not the case, we have to test the performance of the environment with some simple scenarios in order to prove sanity and stability.

Following are the steps of the tests:

* First, we will test the Zombie Player performance with a random-uniform Light Player.
* Therefore, we will test the light Player performance with a random-uniform zombie Player.
* Then, we will test the performance of both the agents trying to learn while playing against each other.

Now, before we start the tests like mentioned, lets introduce the algorithm we are going to use for learning.

#### The Algorithm – Double Deep Q Network

For testing the performance we'll use a model known as DDQN, which stands for Double Deep Q-Network.

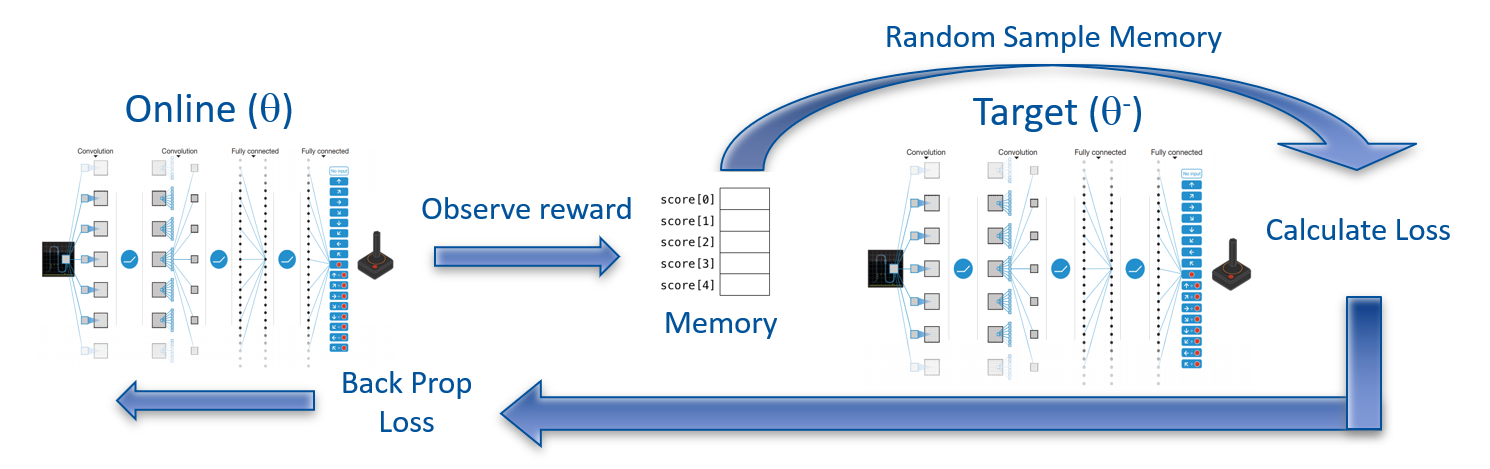


Figure 9 – Double Deep Q-Network architecture and flow

The learning algorithm we used in this project called: "Double Deep Q Network".  
In **Double Deep Q Network**, the agent uses two neural networks to learn and predict what action to take at every step. One network, referred to as the **online network**, is used to predict what to do when the agent encounters a new state. It takes in the state as input and outputs Q values for the possible actions that could be taken.  
The other network, referred to as the **target network**, is used to evaluate what is the best action to take for the next state (the action with the highest Q value).

For the evaluation process we use something called **replay memory**, which holds the last history up to sometime in the past. And eventually, for **loss calculation** we sample a **random batch** (with some size smaller than the memory size) from the replay memory and updating by **back propagation** the online network. After some number of rounds called **replace target frequency**, we **update the target** net weights according to the online net. We can look at Figure 9 that sums up the whole idea.

#### Epsilon Greedy strategy

**Epsilon greedy policy** is a way of selecting random actions with uniform distribution from a set of available actions. Using this policy either we can select random action with epsilon probability and we can select an action with 1-epsilon probability that gives maximum reward in given state.

During the learning process we will use the epsilon greedy strategy with non-linear decrease in epsilon of:

While the 'start' and 'end' parameters stand for the starting value and ending value of the epsilon function. The 'step' parameter represents the current step of an episode and is multiplied by the 'decay' parameter that is equal to , for achieving the start-end values of the epsilon function.

For example, with 10,000 episodes with 2 steps each, we get the convex function :

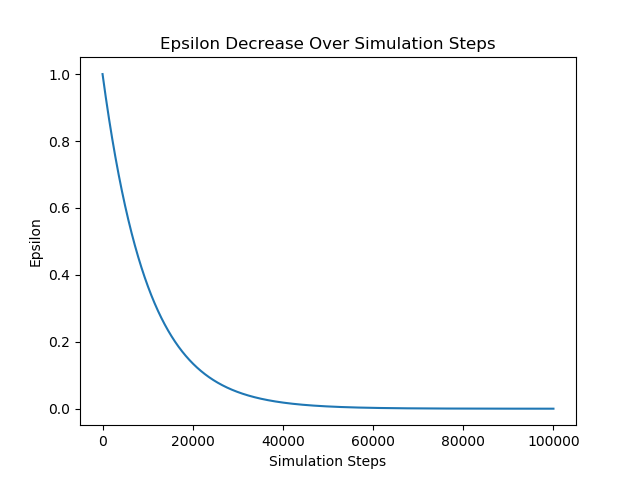
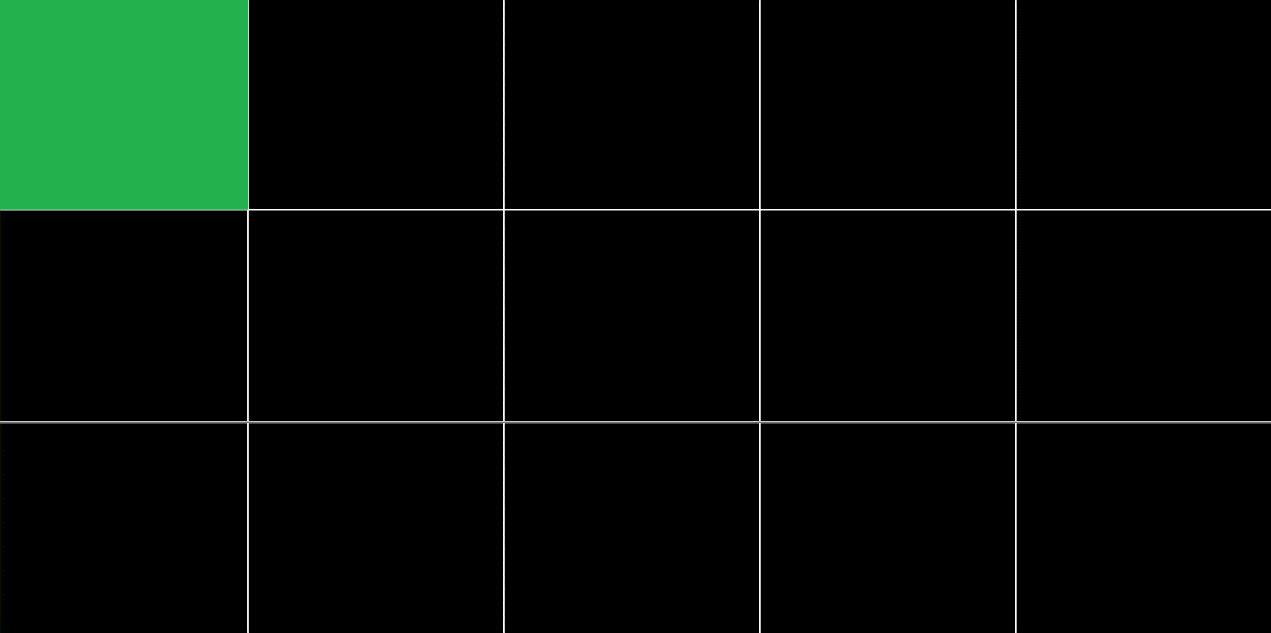


Figure 10 - Epsilon greedy values

#### Zombie Player test on a 3x5 board

As of the first test of learning, consider the Zombie Player that learns alone while the Light Player is forced to take some predetermined action.

At the beginning, we implemented the DDQN algorithm for the zombie agent with grid of 3x5 that looks like:



0

1

2

Figure 11 – environment set-up for zombie Player performance check with optional actions

As we can see in Figure 11 – environment set-up for zombie Player performance check, the light Player is forced to take the top left cell as action in every step as stated above.

Consider the following parameters for the learning process:

|  |  |
| --- | --- |
| Light Player action | 0 |
| Target update | 10 |
| Num episodes | 100 |
| Steps per episode | 100 |
| Batch size | 264 |
| Gamma (discount factor) | 0.999 |
| Epsilon-greedy start | 1 |
| Epsilon-greedy end | 0.05 |
| Epsilon-greedy decay | 0.000222 |
| Replay memory size | 1000 |
| Learning rate | 0.001 |

Table 1 – Learning parameters while evaluating the zombie Player

With a deep NN of three layers, all fully connected (called 'Linear' in pytorch formulation): Linear (15,128), Linear (128,128), Linear (128,3). As we know, the zombie Player has three possible actions to play – the meaning of the '3' in the last layer.

We achieved a convergence in the number of times the zombie Player chose to send a zombie from the top row (the worst decision it could make):

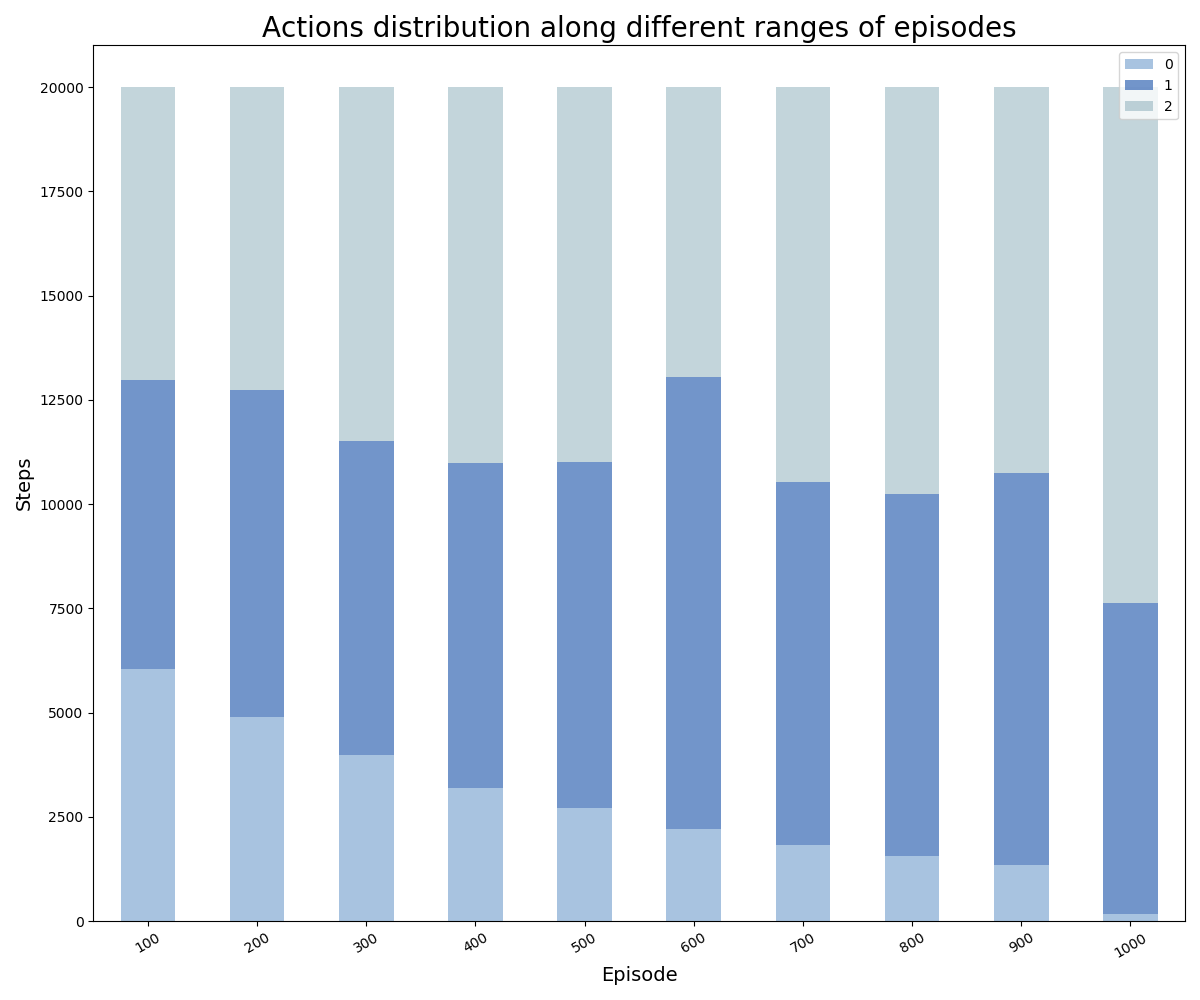


Figure 12 – Zombie Player actions distribution along different ranges of episodes

In Figure 12 – Zombie Player actions distribution along different ranges of episodes we can notice the fading of the lower blue which means that the zombie Player decides to choose action one or two outright as the episodes go on.

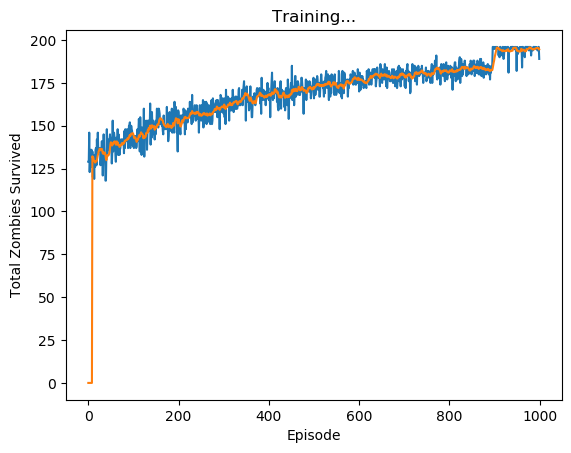
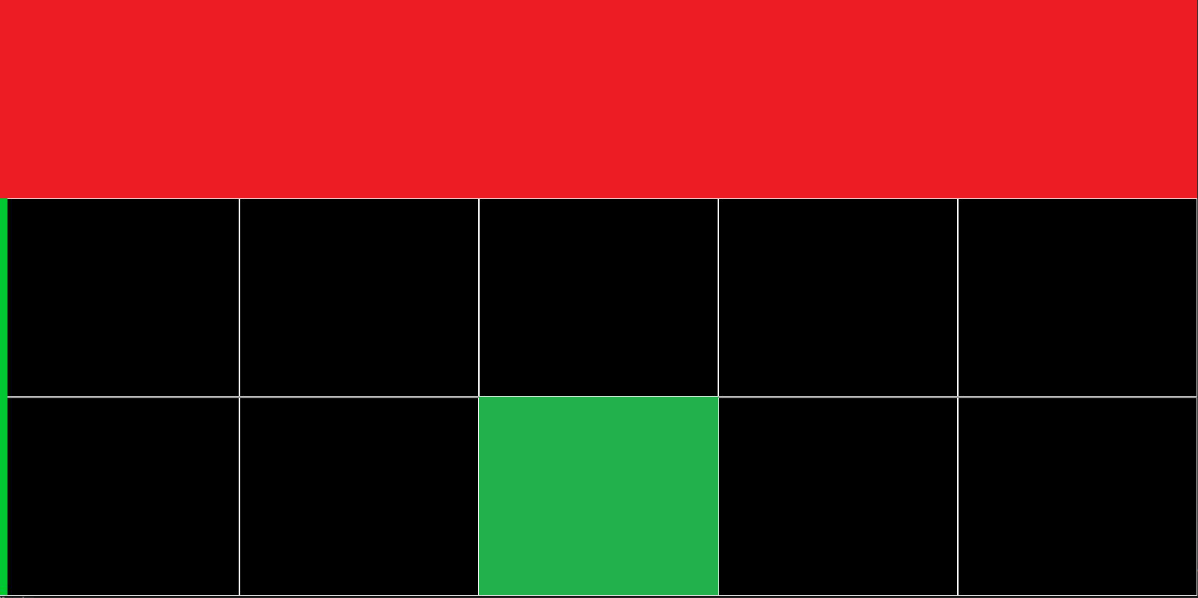


Figure 13 – Total zombies survived vs. the episodes (blue) with its moving average (orange)

Plus, we can tell from Figure 13, the zombie Player reaches the maximum it can get – reaches 196 zombies from possible of 196 (there are 201 steps with grid width of 5), we were able to achieve that thanks to the last 100 episodes with zero epsilon greedy parameter.

#### Light Player test on a 3x5 board



7

8

6

5

9

0

1

2

3

4

10

11

13

14

12

Figure 14 - environment set-up for light Player performance check with optional actions

Figure 14 illustrates the simulation while testing the performance of the light Player.   
As we can see, the zombie Player takes only the action 0 (red cells, predetermined for simplicity) what caused the zombies to exit from the upper cell solely. Which after few steps made the entire top row full of zombies. In addition, the green cell represents the light action in the current step.

Furthermore, in general, we can tell in the first episodes there should survive roughly ~130 zombies since the actions are taken random and there is 33% chance for the light Player to light the top row.

Once again, consider the following parameters:

|  |  |
| --- | --- |
| Zombie Player action | 0 |
| Target update | 10 |
| Num episodes | 1000 |
| Steps per episode | 200 |
| Batch size | 256 |
| Gamma (discount factor) | 0.999 |
| Epsilon-greedy start | 1 |
| Epsilon-greedy end | 0.05 |
| Epsilon-greedy decay | 0.00001 |
| Replay memory size | 1000 |
| Learning rate | 0.001 |

Table 2 – learning parameters while evaluating the light Player

With a deep NN of three layers, all fully connected (called 'Linear' in pytorch formulation): Linear (15,128), Linear (128,128), Linear (128,15).  
In this case we have fifteen outputs, hence the output of the last layer equals to 15. This time we achieve increase in the amount the light Player chooses to light the first row. The phenomenon indicates the light agent's recognition of the fact that the zombies are coming out of the upper cell.

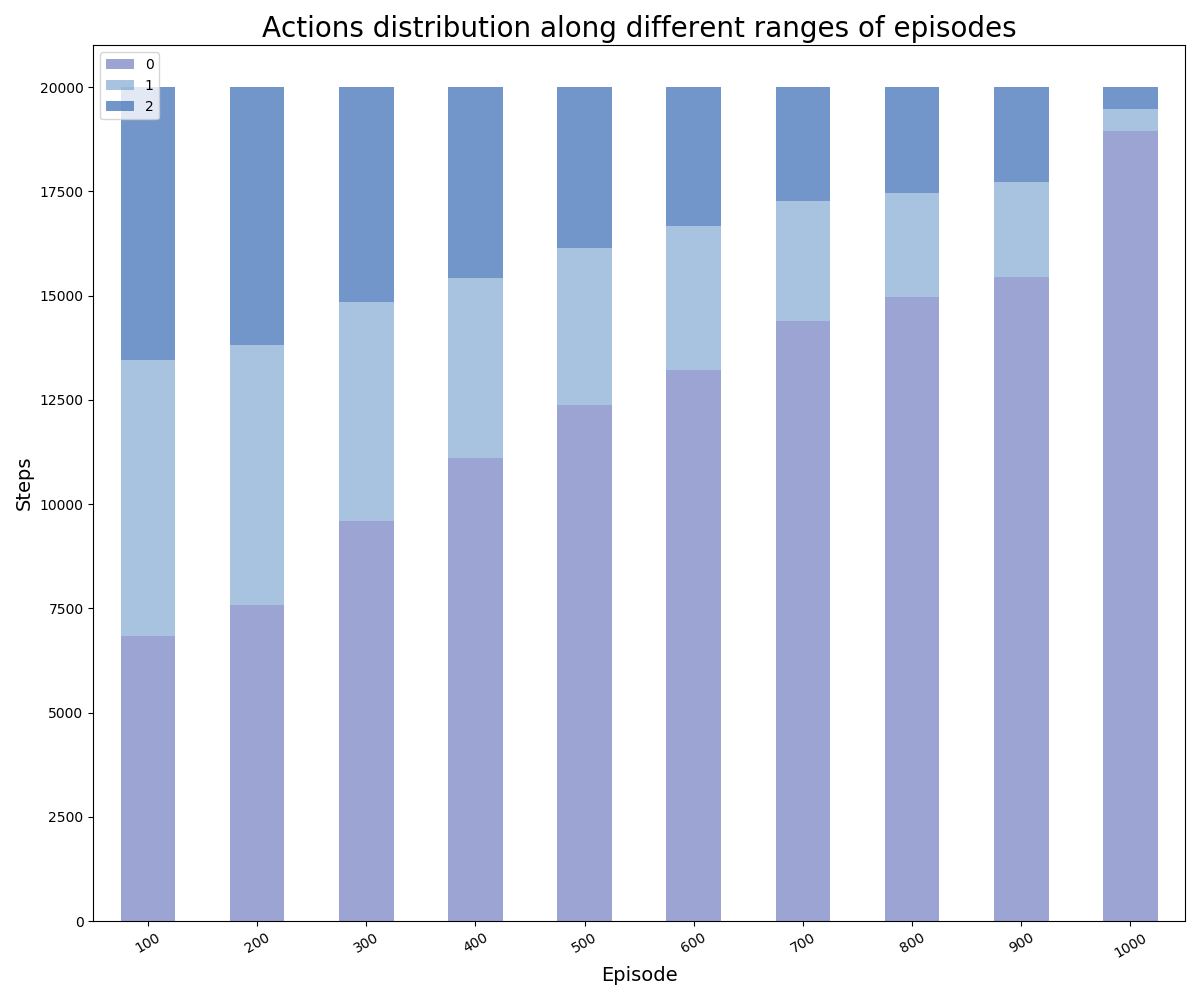


Figure 15 - Light Player actions distribution along different ranges of episodes

In Figure 15 we can see the significant increase in the number of times the light agent selected the top row illumination throughout the simulation progress.  
After 900 episodes, the light Player chooses to light the correct row more than 90% of the time. Note that the value of the greedy epsilon here is decreasing to 0 at the 900th episode.

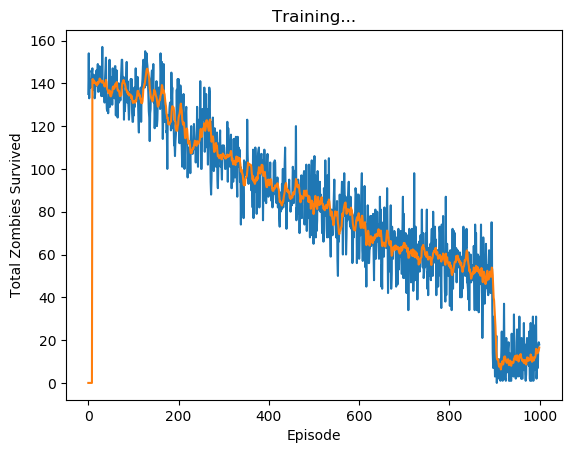


Figure 16 - Total zombies survived vs. the episodes (blue) with its moving average (orange)

In Figure 16 we can clearly see the learning process of the light agent, from the first episodes with 140 zombies survived (out of possible 195 over the episode, it's approximately two thirds), it managed to eliminate over 90% of the zombies by the 1000th episode, note that the noise remains thanks to the stochastic coin we flip every time step which means that zombies can still survive after getting hit.

## Methodology

Once we have built the simulation, in this chapter we will expand the process we will go through in the project, the different methods we have chosen for our research and their results.

In this chapter we will expand on all the ideas we have reviewed in chapter ‎6 and can be seen as a continuation of it.

### Double Deep Q-Network Evaluation

The first method we use called Double Deep Q-Network (or DDQN). The algorithm served us for Simulation Test (see ‎8.4), in this chapter we will examine the performance of a DDQN-based agent against the four simple agents and itself. We seek to find the best parameter sets for each scenario and compare their performances side by side.

We will examine the performance of all scenarios for different board sizes and different values of the parameters: **Target network update frequency** and **Replay memory size** (Those two parameters are most influencing when it comes to DDQN).

The evaluation process of DDQN is done with three phases:

* Estimating Average Test Reward of each scenario of different sets of parameters on various boards
* Choosing the best sets of parameters
* At last, comparing the results of the best agent over the different competitors

#### Evaluation Configuration

For the evaluation process, let's review the hyper-parameters of the model:

* Fixed parameters:
* Number of training episodes: 800
* Number of test episodes: 200
* Zombies per episode: 20
* Light size: 2
  + In case the learning agent plays as Zombie, the light size is a third of the board length - In order to make it less easy
* Minimum hit points of certain death: 1
* Heal ratio: 0.97
* Tuning Parameters:
* Target Policy update frequency – [500, 750, 1000]
* Replay Memory size – [3000, 4000, 5000]

#### Game Scenarios

Each scenario consists of two players on top of a board. The scenarios we are going to run are:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Game Board | Light Player | Zombie Player |  | Zombie Player | Light Player |
| 10x10 | DDQN | Single Action |  | DDQN | Single Action |
| Double Action |  | Double Action |
| Uniform |  | Uniform |
| Gaussian |  | Gaussian |
| 20x20 | DDQN | Single Action |  | DDQN | Single Action |
| Double Action |  | Double Action |
| Uniform |  | Uniform |
| Gaussian |  | Gaussian |
| 30x30 | DDQN | Single Action |  | DDQN | Single Action |
| Double Action |  | Double Action |
| Uniform |  | Uniform |
| Gaussian |  | Gaussian |

Table 3 – DDQN Evaluation: Game Scenarios

We repeat each scenario twice to reduce the bias of a single sample. Which means, in total we execute twice four scenarios (four simple agents), for each board size, for each player type. That sums up to .

We do all the above for every combination of the tuning parameters which adds up to scenarios to process.

Let's start with an example of a single evaluation. Consider a scenario of DDQN as Zombie and Single Action Agent as Light.

First, we repeat the scenario for several times while our Framework produces the following reward per episode graphs:

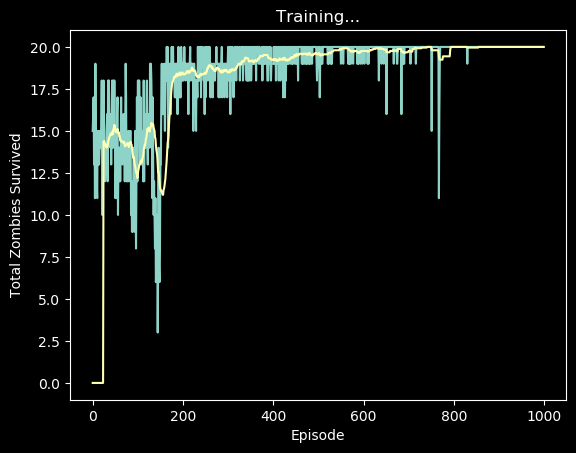


Figure 17 - Example of reward per episode graph, DDQN plays Zombie and Single Action plays Light

Second, we get the average of test rewards (episodes 800 - 1000) from all graphs as in Figure 18:

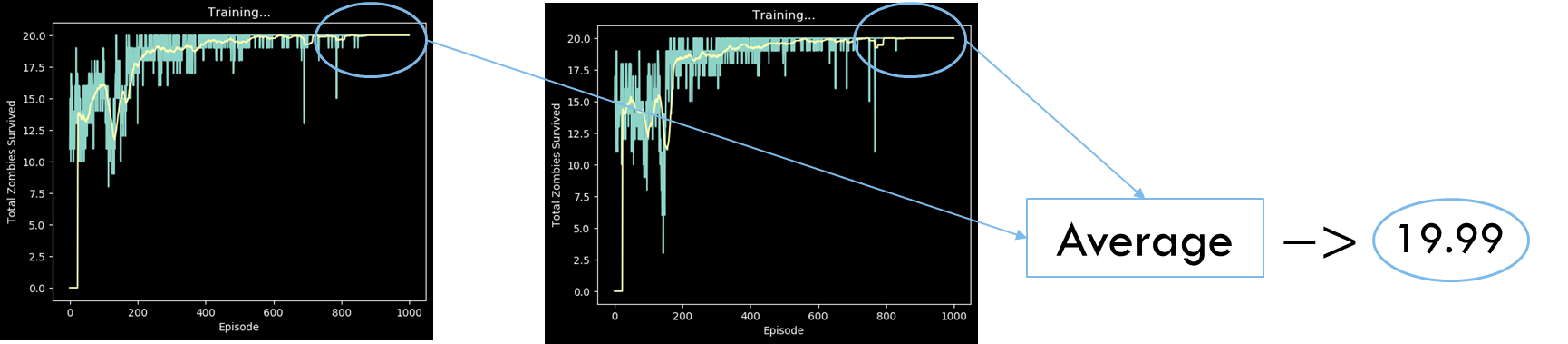


Figure 18 - Scenario Evaluation by Average Test Reward

And finally get an estimate of the performance of the agents in the scenario (the value 19.99 as in Figure 18)

#### Double Deep Q-Network as Zombie Player

Before diving in to the results, lets understand a simple scenario of DDQN Agent as Zombie and Single Action Agent as Light:

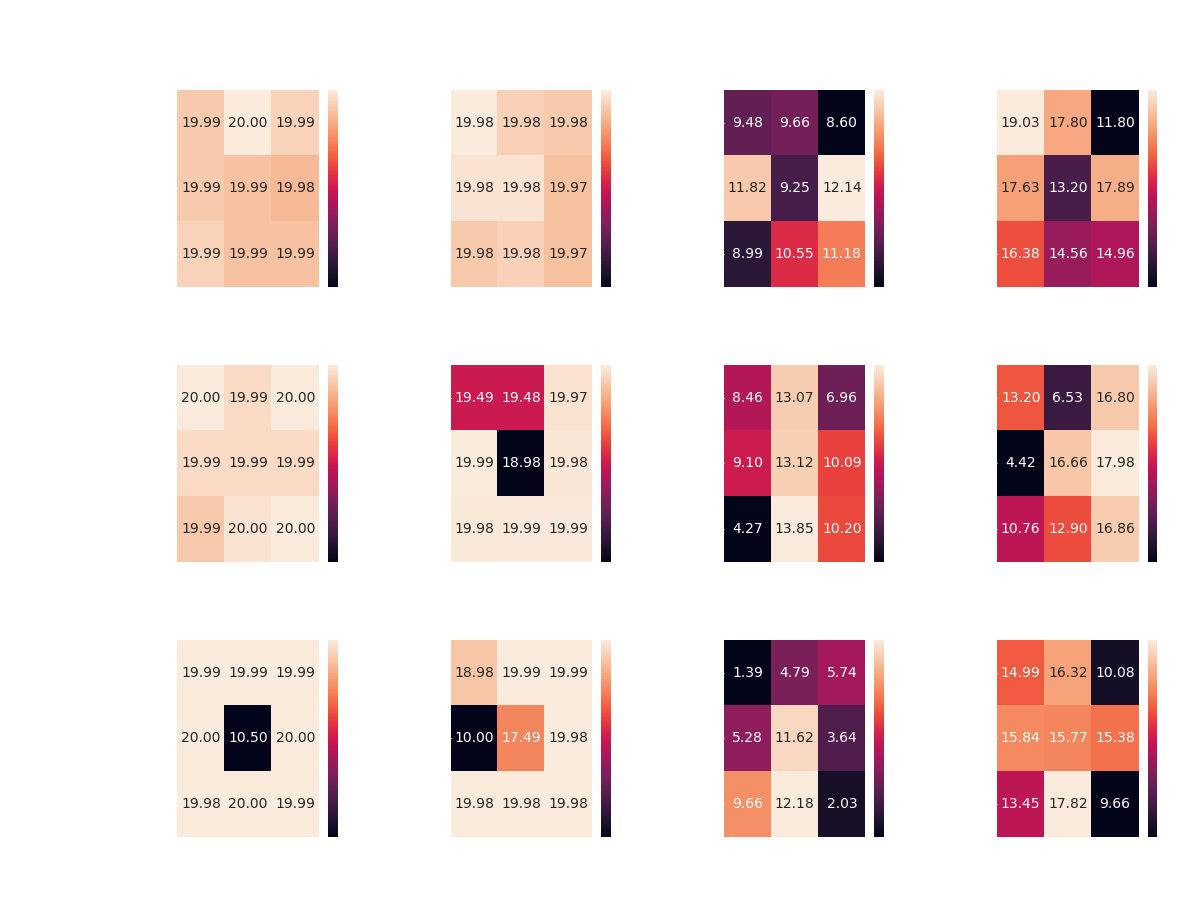


Figure 19 - Heat-Map of the Average Test Rewards of all the scenarios that DDQN Agent plays Zombie and Single Action Agent plays Light

The diagram consists of two axes for each tuning parameter and cells with values of the Average Test Rewards. There is also a matching color axis that describes the absolute value displayed inside each cell.

Since the maximum reward possible in a single episode is 20 (the total number of zombies in episode – [see configuration](#_Evaluation_Configuration)) We can easily conclude that the DDQN agent as Zombie outperformed the Light Player for all parameters.

Next, let's have a look at the rest of the scenarios:

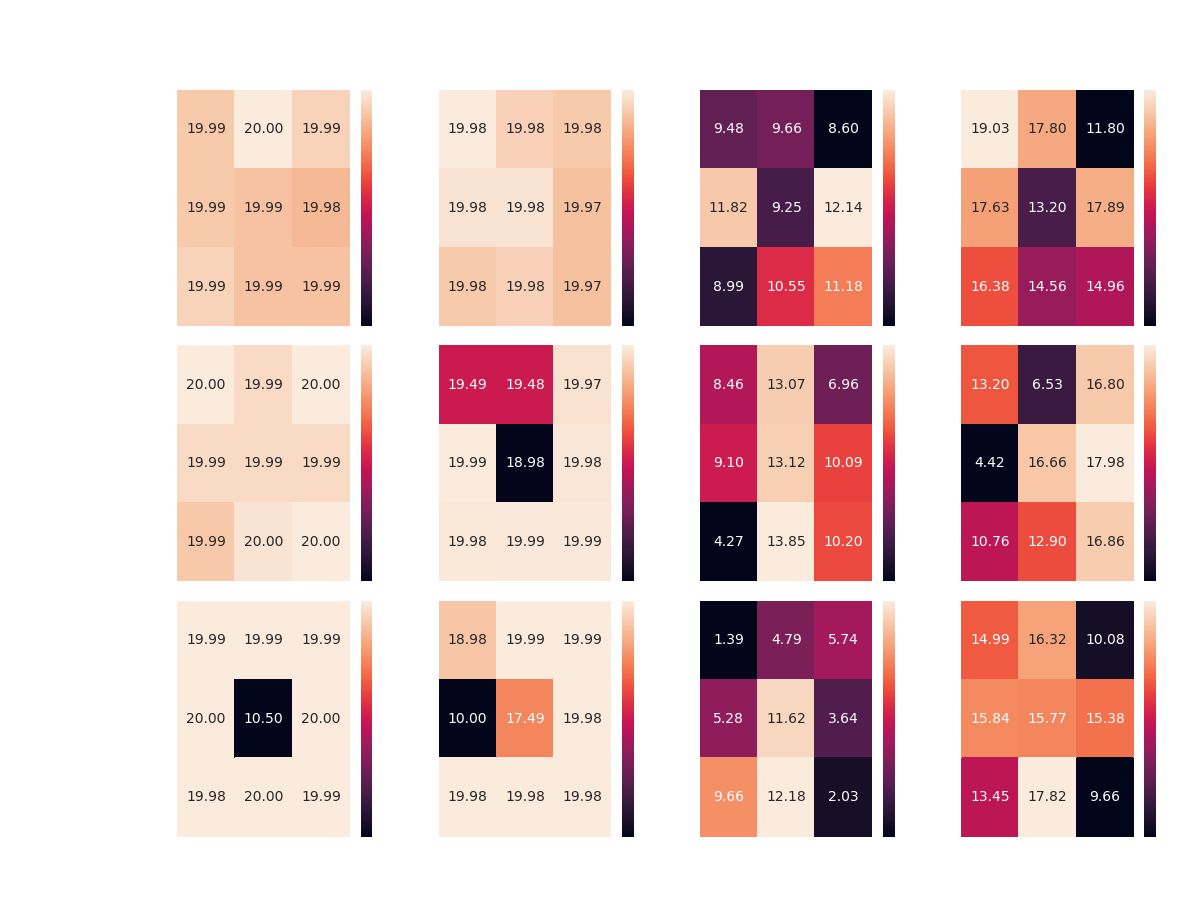


Figure 20 - A summary of all the scenarios that DDQN plays Zombie by Heat-Maps of the Average Test Reward

Overall, it seems that the DDQN agent manages to overcome its competitors in most of the scenarios:

* Achieving optimal reward competing Single and Double Action Agents with most of the configurations, on top all boards
* Achieving 17+ average reward competing the Gaussian Agent on all boards, without achieving optimal reward in any scenario

However, the DDQN agent seems to encounter difficulties when faced with the Uniform Agent – managing to achieve average reward of 12-14.

While competing against both the Random Agents, the DDQN Agent gets approximately same average reward over all three boards. Perhaps more learning episodes would make a change here?

* Keep in mind that the highest reward against the Uniform Agent as light player is 15 – calculated by an agent which picks the best action exclusively (places zombies at the first row)

To settle the issue, we ran multiple scenarios of the same configuration, DDQN Agent vs. Uniform Agent, this time with 1800 learning episodes and 200 test episodes – all cases yield to the same average test reward of around 12, hence, the DDQN Agent can't achieve better results in those scenarios.

Now we can summarize all results of best parameter configurations:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Board size | Competitor | Memory Size | Target Update | Average Test Reward |
| 10x10 | Single Action | 3000 | 750 | 20 |
| Double Action | 3000 | 500 | 19.98 |
| Uniform | 4000 | 1000 | 12.14 |
| Gaussian | 3000 | 500 | 19.03 |
| 20x20 | Single Action | 3000 | 500 | 20 |
| Double Action | 4000 | 500 | 19.99 |
| Uniform | 5000 | 750 | 13.85 |
| Gaussian | 4000 | 1000 | 17.98 |
| 30x30 | Single Action | 5000 | 750 | 20 |
| Double Action | 3000 | 1000 | 19.99 |
| Uniform | 5000 | 750 | 12.18 |
| Gaussian | 5000 | 750 | 17.82 |

Table 4 - Best Configurations of all Scenarios in which the DDQN plays Zombie

* In cases where the same result was obtained for several scenarios, the best params chosen arbitrarily

From the summary of the results (see Table 4), there does not appear to be any clear preference for a particular configuration in the aspects of the boards and agents.

From here, we can present the rewards graph of each scenario (see Figure 17) side-by-side along different board sizes:



Figure 21 - Comparing the results of the DDQN agent as the Zombie Player, with the best parameters over the different four simple competitors

* note that in the case where the algorithm plays the zombie, the reward is positive

In general, it can be noticed that there is learning against all the agents in all the games.  
There is absolute success (convergence to optimal policy) against the constant agents  
On the other hand, in games against the random agents, there is partial success but a significant upward trend in almost all cases. In all cases except against the random agent in the game in the smallest board.  
However, because at this stage we have not examined the results in relation to any optimal/another method, we cannot say if these are the best results the agent could achieve, but these are good and satisfactory results - there is general learning in all board sizes against all agents.

#### Double Deep Q-Network as Light Player

Same of the [last chapter](#_Double_Deep_Q-Network), Before diving in, let's review a simple scenario of DDQN Agent as Light and Single Action Agent as Zombie:

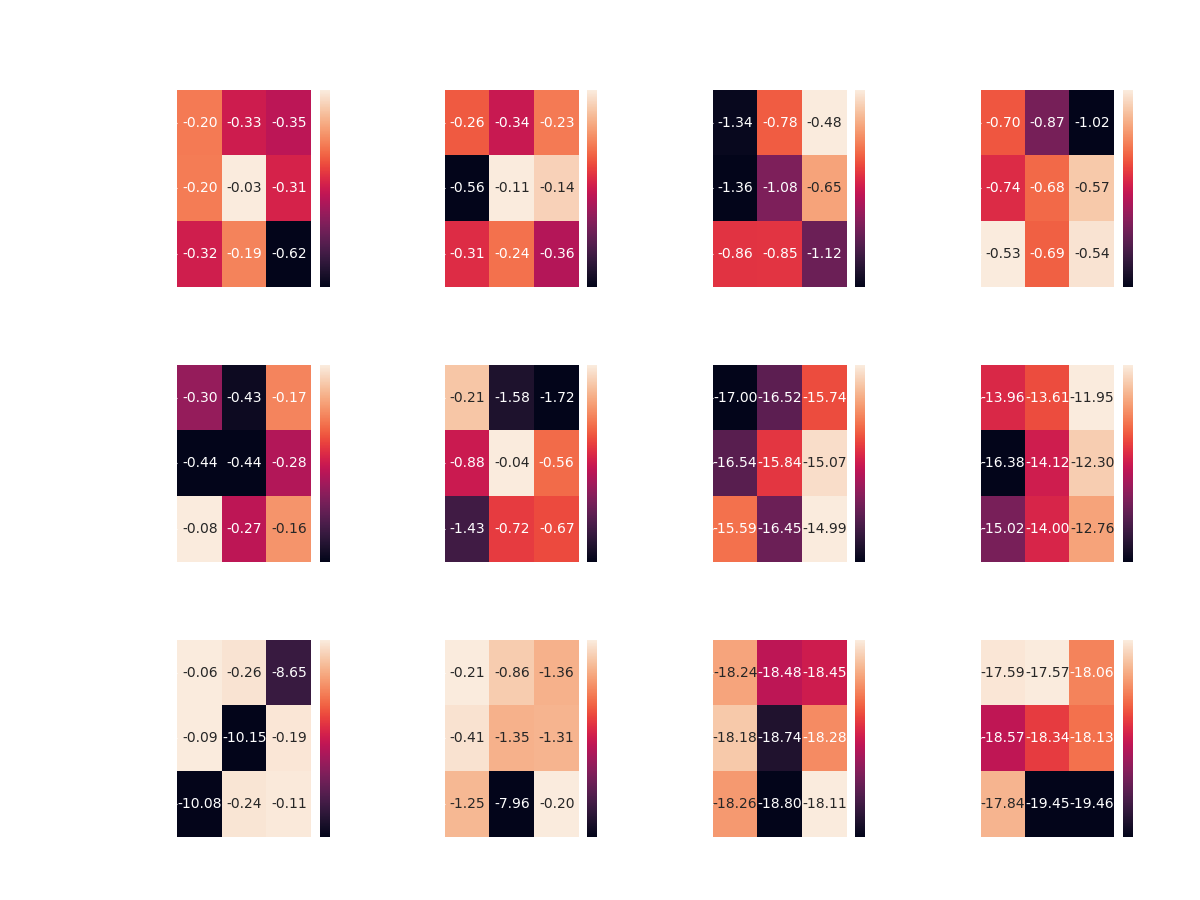


Figure 22 - Heat-Map of the Average Test Rewards of all the scenarios that DDQN Agent plays Light and Single Action Agent plays Zombie

We look at the same diagram with one significant difference – the values inside the cells are negative. Remember that we are playing a zero-sum game in such a way that the light agent will always receive an opposite and necessarily, negative reward. Yet his mission is the same, to maximize it.

Next, let's have a look at the rest of the scenarios:

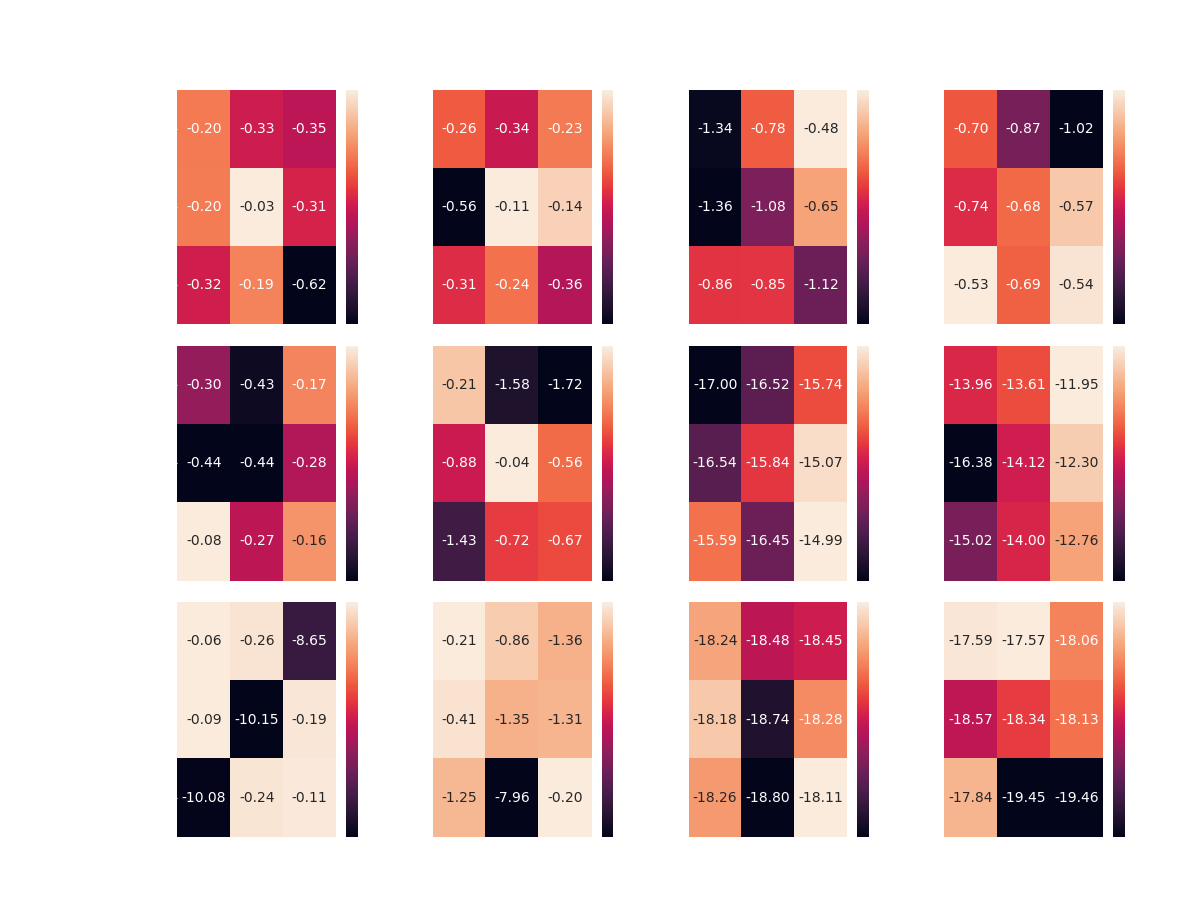


Figure 23 - A summary of all the scenarios that DDQN plays Light by Heat-Maps of the Average Test Reward

This time the DDQN Agent managed to achieve partial success.

Over all boards, the DDQN Agent seems to overcome the two Constant Agents, with some preference of the [750, 4000] configuration in half of the cases.

However, competing the Random Agents didn’t lead to same success. the DDQN Agent was able to reach optimality only in the case of the smallest board, and this time, without any preference of any specific configuration.

In addition, we witness the instability of DDQN Agent while training on large boards, evident in squared board of length 30, the DDQN agent shows that there are combinations of parameters that do not lead to good results against the Constant Agents (Single and Double Agents), yet in most cases it is still a success.

Now we can summarize all the results of best parameter configurations:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Board size | Competitor | Memory Size | Target Update | Average Test Reward |
| 10x10 | Single Action | 4000 | 750 | -0.03 |
| Double Action | 4000 | 750 | -0.11 |
| Uniform | 3000 | 1000 | -0.48 |
| Gaussian | 5000 | 500 | -0.53 |
| 20x20 | Single Action | 5000 | 500 | -0.08 |
| Double Action | 4000 | 750 | -0.04 |
| Uniform | 5000 | 1000 | -14.99 |
| Gaussian | 3000 | 1000 | -11.94 |
| 30x30 | Single Action | 3000 | 500 | -0.06 |
| Double Action | 5000 | 1000 | -0.2 |
| Uniform | 5000 | 1000 | -18.11 |
| Gaussian | 3000 | 750 | -17.57 |

Table 5 - Best Configurations of all Scenarios in which the DDQN plays Light

* In cases where the same result was obtained for several scenarios, the best params chosen arbitrarily

From the summary of above (Table 5), the DDQN Agent seems to prefer the parameters: [750, 4000] while competing against the two Constant Agents (Single and Double Agents).

As for the rest, there is no apparent preference.

From here, we can present the rewards graph of each scenario (see Figure 17) side-by-side along different board sizes:

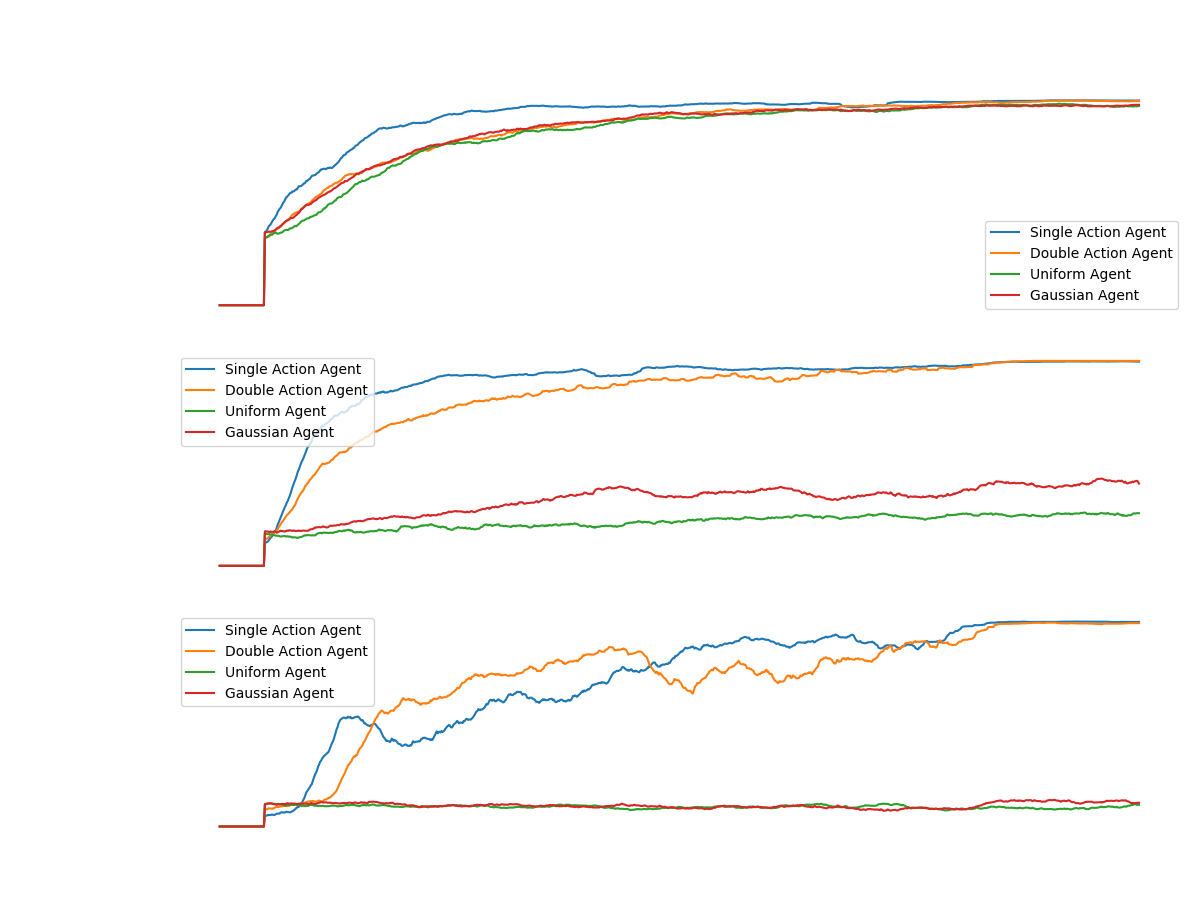


Figure 24 - Comparing the results of the DDQN agent as the Light Player, with the best parameters over the different four simple competitors

* note that in the case where the algorithm plays the zombie, the reward is negative

We can see a beautiful convergence to optimal policy against all agents on a 10x10 board. However, when looking at the success of the light player as the DDQN agent in larger board sizes, one can see controversial success. There is a convergence in all cases in the game against the constant agents. But, if we look at the game against the random agents, we can see a relatively upward trend in the game against the middle Gaussian agent. And in the rest of the games: A complete failure - there is not even a spark of convergence.

### Learning Based Monte Carlo Tree Search

The purpose of this section is to add the Monte Carlo Tree Search algorithm to our project.   
We must take into account that this is the second algorithm we implement during the project, among many more in potential. Therefore, the code we build from now on is fully generic and easily adaptable to absorb any kind of new learning algorithm (see ‎8).

Continuing chapter ‎6.2, each round of Monte Carlo tree search algorithm consists of four steps:

* Selection: Start from root R and select successive child nodes until a leaf node L is reached. The root is the current game state and a leaf is any node that has a potential child from which no simulation (playout) has yet been initiated.
  + We traverse according to the upper confidence bound of Where:
    - is the total reward aggregated in the current node
    - is the number of visits through the current node
    - is the number of visits through parent node
    - is the exploration factor
* *Expansion*: Unless  ends the game decisively (e.g., win/loss/draw) for either player, create one (or more) child nodes and choose node C from one of them. Child nodes are any valid moves from the game position defined by L.
* *Simulation*: Complete one random playout from node C. This step is sometimes also called playout or rollout. A playout may be as simple as choosing uniform random moves until the game is decided (for example in chess, the game is won, lost, or drawn).
* *Backpropagation*: Use the result of the playout to update information in the nodes on the path from C to R.

All the above can be summed up to Figure 25:

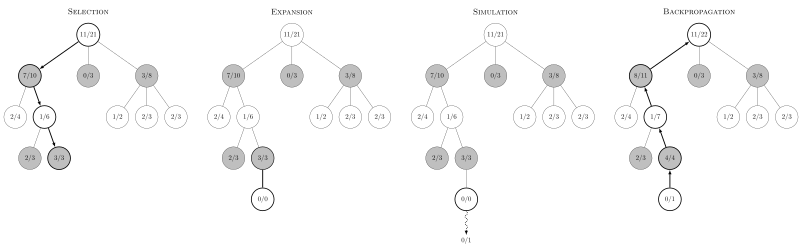


Figure 25 - Step of Monte Carlo tree search

#### MCTS - Implementation and Config

As mentioned in "*Monte Carlo Tree Search*" the whole learning process of the algorithm consists of **four stages**. In this chapter we will elaborate on each stage, while presenting conflicts that arose along the way with proposals and the way of realization.

In the first stage the goal is to find a leaf in the tree which we will move on. To do this we will start traversing the nodes of the tree according to UCT strategy until we reach a leaf node.

Now, we would like to expand its children. We can do this in one of two ways, **one is to create all possible nodes** (as the number of possible actions) **and the other is to create only one**.

The consideration here is, On the one hand, to prevent the explosion of the tree, or with other words, the unnecessary expansion of places that will later be found irrelevant, thing that might cause the explosion of memory and killing the process. while the expansion of nodes that are relevant will eventually will cause a positive effect of faster convergence (in terms of number of steps).

On the other hand, when we expand only one leaf, we delay the expansion of the tree in ways that are interesting and relevant.

Imagine that we have reached a particular node, assuming that this node is very relevant to learning, we would like to expand and explore its children. Something we will not be able to do until its fully expanded. The described situation was examined and we finally came to the conclusion that despite the overloading memory in a large tree, **we would prefer to expand all the leaves of a particular node and avoid slowing down the convergence of the algorithm**.

At the next phase, we perform simulations from the selected leaf, in order to estimate the expected reward that an agent will receive from now on. According to that estimation, we update the statistics upwards. Which raises the question: **Should we update the stats up to the tree root or stop at the current game-state-node?**   
According to the literature I have seen and code-libraries concerns the two approaches. Some argue that updating to the root of the tree might bias the game to random directions and therefore it is advisable to update the stats up to the current game-state-node. Others claim that updating to the root of the tree is necessary for the reason that you do not visit this area much (only at beginning of games), therefore, it will take a lot of time to train the agent to choose wise actions at the first moves of each game.

Back to simulation phase. The most significant parameters here are the number of simulations and the depth of a simulation. In each simulation we traverse the tree for some fixed depth, at each step we select random actions for both agents and lastly promoting the environment accordingly, while receiving reward and accumulating it. We continue so until we reach the required depth of simulation. After the process for number-of-simulations time, we average the rewards and backpropagate them.

At this point another assumption was made: **How will we treat a reward with a numerical value greater than 1 in absolute value?** There are few references to such cases in the literature. Accepting such rewards is indeed problematic since a reward received from the environment is 1 or -1 in our game, while reward obtained from simulation gets values from 1 up to the depth-of-simulation (absolute value). **How can we update the statistics of each node with values from different ranges?** Several options were tested and the solution we chose is to normalize the value obtained from the simulation with the depth-of-simulation, which changes its values range back to [-1,1].

After all the above, the backpropagation phase comes, in which, we take the value obtained from the simulation at any given moment and add it to the reward value of each node up to the root node, while increasing the number of visits of the node by 1.

To sum up, during the implementation of Monte Carlo tree search algorithm, some loose ends were discovered:

1. At the selection phase, what upper-confidence-bound constant should we use?
2. At the expansion phase, once we reach a leaf, should we expand all the possible node or just one?
3. After simulation occur, how do we treat a reward greater than 1? Should we update statistics only after simulations to prevent the inconsistency? And if so, how do we use the reward that comes in real time from the environment, since it is a real and accurate reward – we might want to use it.
4. At the backpropagation phase, should we update the stats up to the tree root or the current-game-state?

#### MCTS Results and Conclusions

For evaluation and test of the algorithm we implemented, we taught it in a game against one of the simple agents. The agent reached an equilibrium in games it played as the zombie agent. Against Simple agents and DDQN agent. While as the Light Player, each learning scenario kept fail duo to the 16 RAM constraint of the machine. It prevented us from playing on any reasonable size of board. The main reason of that is that the action space the light agent has is the number of expanded children each node has.

Therefore, we decided to proceed with our research and leaving the MCTS implementation behind, meaning, we won't use it for the following steps and evaluations of the research.

### Learning Based AlphaZero Algorithm

In this chapter we will expand on the assumptions and steps of the algorithm, we will give a detailed description of the implementation of the algorithm in our case and its initial results.

#### AlphaZero – The Neural Network

Unsurprisingly, there's a neural network at the core of things. The neural network is parameterized by θ and takes as input the state of the board. It has two outputs: a continuous value of the board state from the perspective of the current player, and a policy that is a probability vector over all possible actions. When training the network, at the end of each game of self-play, the neural network is provided training examples of the form . is an estimate of the policy from state (we'll get to how πt is arrived at in the next section), is the final outcome of the game from the perspective of the player at . The neural network is then trained to minimize the following loss function (excluding regularization terms):

The underlying idea is that over time, the network will learn what states eventually lead to wins (or losses). In addition, learning the policy would give a good estimate of what the best action is from a given state. The neural network architecture in general would depend on the game. Most board games such as Go can use a multi-layer CNN architecture. In the paper by DeepMind [14], they use 20 residual blocks, each with 2 convolutional layers.

#### Monte Carlo Tree Search for Policy Improvement

Given a state , the neural network provides an estimate of the policy . During the training phase, we wish to improve these estimates. This is accomplished using a Monte Carlo Tree Search (MCTS). In the search tree, each node represents a board configuration. A directed edge exists between two nodes  if a valid action can cause state transition from state  to . Starting with an empty search tree, we expand the search tree one node (state) at a time. When a new node is encountered, instead of performing a rollout, the value of the new node is obtained from the neural network itself. This value is propagated up the search path. Let's sketch this out in more detail.

For the tree search, we maintain the following:

* : the expected reward for taking action  from state , i.e., the Q values
* : the number of times we took action  from state  across simulations
* : the initial estimate of taking an action from the state  according to the policy returned by the current neural network.

From these, we can calculate , the upper confidence bound on the Q-values as:

Here  is a hyperparameter that controls the degree of exploration. To use MCTS to improve the initial policy returned by the current neural network, we initialize our empty search tree with  as the root. A single simulation proceeds as follows. We compute the action  that maximizes the upper confidence bound . If the next state  (obtained by playing action  on state ) exists in our tree, we recursively call the search on s′. If it does not exist, we add the new state to our tree and initialize  and the value  from the neural network, and initialize  and  to 0 for all . Instead of performing a rollout, we then propagate  up along the path seen in the current simulation and update all  values. On the other hand, if we encounter a terminal state, we propagate the actual reward (+1 if player wins, else -1).

After a few simulations, the  values at the root provide a better approximation for the policy. The improved stochastic policy  is simply the normalized counts . During self-play, we perform MCTS and pick a move by sampling a move from the improved policy .

#### Policy Iteration Through Self-Play

We now have all elements required to train our unsupervised game playing agent. Learning through self-play is essentially a policy iteration algorithm - we play games and compute Q-values using our current policy (the neural network in this case), and then update our policy using the computed statistics.

Here is the complete training algorithm. We initialize our neural network with random weights, thus starting with a random policy and value network. In each iteration of our algorithm, we play a number of games of self-play. In each turn of a game, we perform a fixed number of MCTS simulations starting from the current state . We pick a move by sampling from the improved policy . This gives us a training example . The reward  is filled with value of: +1 if a zombie has passed the board alive, else -1. The search tree is preserved during a game.

At the end of the iteration, the neural network is trained with the obtained training examples. The old and the new networks are pit against each other. If the new network wins more than a set threshold fraction of games (55% in the DeepMind paper), the network is updated to the new network. Otherwise, we conduct another iteration to augment the training examples.

#### Contribution of AlphaZero over AlphaGo-Zero

Our implementation is influenced from the last versions of the algorithm (AlphaZero, AlphaGo Zero). It's important to understand why AlphaZero is more suitable for our problem. Let's dive into the two approaches.

The AlphaZero algorithm described in [14] differs from the original AlphaGo Zero [15] algorithm in several aspects. AlphaGo Zero estimates and optimizes the probability of winning, assuming binary win/loss outcomes. AlphaZero instead estimates and optimizes the expected outcome, **taking account of draws or potentially other outcomes**. The rules of Go are invariant to rotation and reflection. This fact was exploited in AlphaGo and AlphaGo Zero in two ways. First, training data was augmented by generating 8 symmetries for each position. Second, during MCTS, board positions were transformed using a randomly selected rotation or reflection before being evaluated by the neural network, so that the Monte Carlo evaluation is averaged over different biases. **The rules of chess and shogi are asymmetric, and in general symmetries cannot be assumed**. AlphaZero does not augment the training data and does not transform the board position during MCTS. In AlphaGo Zero, self-play games were generated by the best player from all previous iterations. After each iteration of training, the performance of the new player was measured against the best player; if it won by a margin of 55% then it replaced the best player and self-play games were subsequently generated by this new player. **In contrast, AlphaZero simply maintains a single neural network that is updated continually**, rather than waiting for an iteration to complete.

#### Our Network

Following section ‎6.3 (*From AlphaGo to AlphaZero*) and the previous details provided in the chapter, the model we use takes the bold ideas (see ‎9.3.4) that is basically the AlphaZero algorithm and thus actually makes it possible to implement the principles of Multi Agent Reinforcement Learning together with Monte Carlo Tree Search (see ‎9.2: Learning Based Monte Carlo Tree Search), on top of our game environment 'Light vs Zombies'.

There are many network architectures regarding the alpha/Go/Zero implementations (see [15],[16],[17]).

In the paper by DeepMind [14], they use 20 residual blocks, each with 2 convolutional layers. Despite that, we were able to train a 3-layer CNN network followed by a few feedforward layers.

Let's review the architecture of AlphaZero as Light Agent with more details:

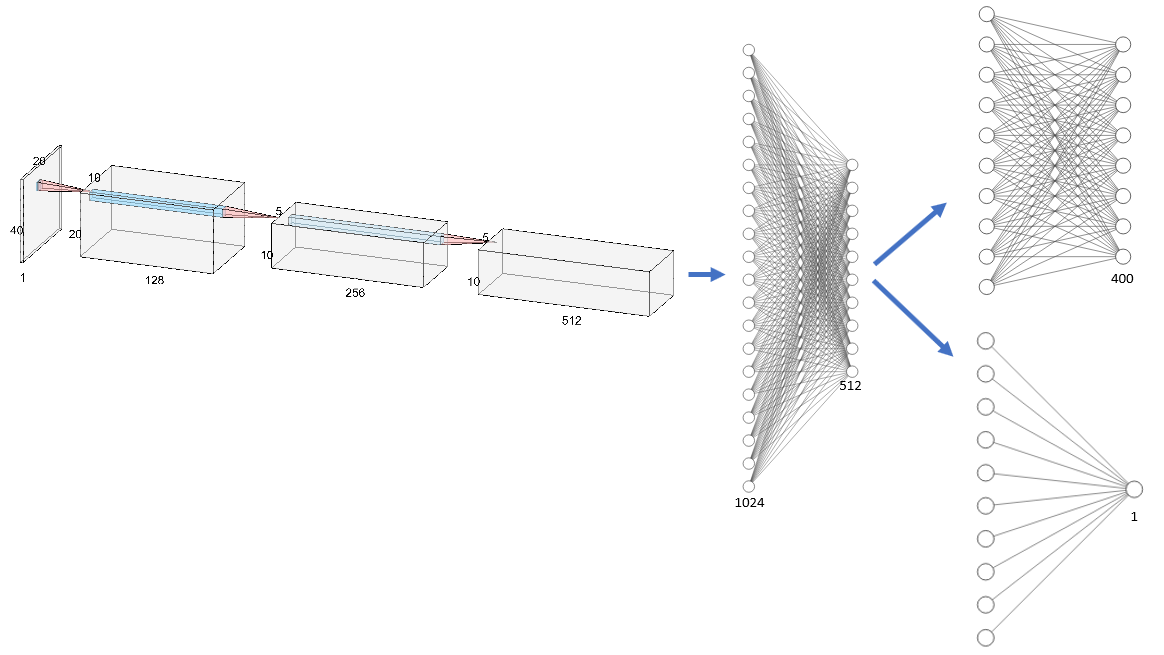


Figure 26 - NN Architecture. Red and blue arrows indicate convolutional and fully Connected layers respectively

Our input of 20 by 40 (the light agent has locations and strength as its state) passing forward the following layers:

* 2D-Convolution layer with 128 filters with size of 3 by 3, stride of 1 and padding of 1 to keep the size of the input.
* 2D-MaxPool of stride 2.
* 2D-BatchNorm of the 128 filters.
* 2D-Convolution layer with 256 filters with size of 3 by 3, stride of 1 and padding of 1 to keep the size of the input.
* 2D-MaxPool of stride 2.
* 2D-BatchNorm of the 256 filters.
* 2D-Convolution layer with 512 filters with size of 3 by 3, stride of 1 and padding of 1 to keep the size of the input.
* 2D-BatchNorm of the 512 filters.
* FC layer – 1024 units.
* FC layer – 512 units.
  + Output layer of 400 units – represents the action probability
  + Output layer of 1 unit – represents the value function approximation.

Or with pytorch view:

=================================================================  
Layer (type:depth-idx)                   Param #  
=================================================================  
├─Conv2d: 1-1                            1,280 = 128x(3x3+1) \*plus 1 for bias unit  
├─MaxPool2d: 1-2                         --  
├─BatchNorm2d: 1-3                       256  
├─Conv2d: 1-4                            295,168 = (128x3x3+1)x256 \*biases as number of out-filters  
├─MaxPool2d: 1-5                         --  
├─BatchNorm2d: 1-6                       512  
├─Conv2d: 1-7                            1,180,160 = (256x3x3+1)x512  
├─BatchNorm2d: 1-8                       1,024  
├─Linear: 1-9                            5,243,904 = 1024x(1+512x(3x3+1))   
├─BatchNorm1d: 1-10                      512  
├─Linear: 1-11                           32,896 = 256x(128 +1)  
├─BatchNorm1d: 1-12                      256  
├─Linear: 1-13                           12,900  
├─Linear: 1-14                           129  
=================================================================  
Total params: 6,768,997  
Trainable params: 6,768,997  
Non-trainable params: 0  
=================================================================

#### Training Process of AlphaZero

Having gone through the rules of the game, the implementation and special assumptions, we can now give an example that will summary the whole training process with a set of default hyper-parameter:

At first, similarly to MCTS, we build a search tree but this time the tree will be initialized as new in each episode of the game. At each step, we will run 100 simulations of searching at a depth of 2 levels down through the tree, such that after each search, stats will be propagated up the tree. At each episode we will store the state-action-reward triplet. Lastly, At the end of an episode, we initiate a new Monte Carlo Tree Search with the new policy network to get ready for the next episode.

After every 40 episodes we will learn our policy network using all data accumulated from the last 200 episodes.

#### AlphaZero Evaluation

Similarly of the evaluation process of DDQN algorithm (described in section ‎0), we let an agent based AlphaZero algorithm play as Light and Zombie players, vs the four simple agents (Constant, Double Constant, Gaussian and Uniform) as its competitor.

##### Evaluation Configuration

For the evaluation and tuning process, we used the following set of hyper parameters of the AlphaZero algorithm:

Fixed parameters:

* Episode's history – 100. The number of recent episodes of which we can sample a batch of training data.
* Number of episodes per training – 20
* Depth of search – 2
* Learning rate – 0.001
* Epochs per training – 10
* Batch size – 128

Tuning Parameters:

* Number of searches – 5, 10, 15
* Exploration rate – 0.5, 1, 1.5

##### Game Scenarios

Similar to ‎9.1.2, The scenarios we are going to run are:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Game Board | Light Player | Zombie Player |  | Zombie Player | Light Player |
| 10x10 | AlphaZero | Single Action |  | AlphaZero | Single Action |
| Double Action |  | Double Action |
| Uniform |  | Uniform |
| Gaussian |  | Gaussian |
| 20x20 | AlphaZero | Single Action |  | AlphaZero | Single Action |
| Double Action |  | Double Action |
| Uniform |  | Uniform |
| Gaussian |  | Gaussian |
| 30x30 | AlphaZero | Single Action |  | AlphaZero | Single Action |
| Double Action |  | Double Action |
| Uniform |  | Uniform |
| Gaussian |  | Gaussian |

Table 6 – DDQN Evaluation: Game Scenarios

For each scenario we calculate the Average Test Reward, again, as we already explained in section ‎9.1.2.

##### Alpha Zero as Zombie Player

Similar to the evaluation process of DDQN algorithm (see: *Double Deep Q-Network as Zombie Player*), we start with the summary of the average test reward of each scenario with heatmaps.

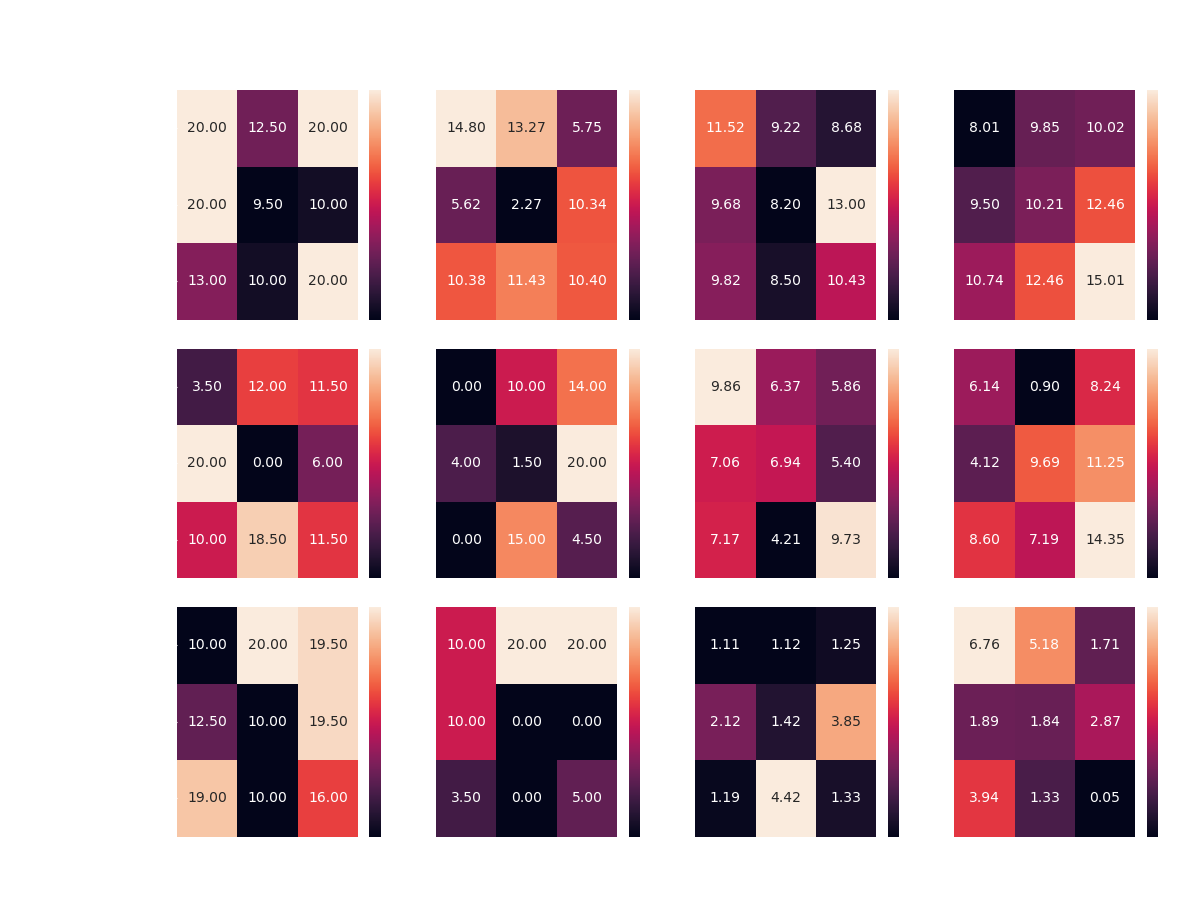


Figure 27 - A summary of all the scenarios that AlphaZero plays Zombie. Heat-Maps of the Average Test Reward

As you can see, unsurprisingly the AlphaZero agent manages to find a way to beat the simple agents in any board configuration.

In contrast to the simple cases, the agent has a hard time coming up with a good strategy (compared to DDQN) against the random agents.

In addition, it is apparent that there is a high variability in the agent's performance across the various configurations, unlike DDQN.

Now, we can summary the results like we did with DDQN (Table 4):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Board size | Competitor | Monte Carlo searches | Exploration rate | Average Test Reward |
| 10x10 | Single Action | 5 | 0.5 | 20 |
| Double Action | 5 | 0.5 | 14.8 |
| Uniform | 10 | 1.5 | 13 |
| Gaussian | 15 | 1.5 | 15 |
| 20x20 | Single Action | 10 | 0.5 | 20 |
| Double Action | 10 | 1.5 | 20 |
| Uniform | 5 | 0.5 | 9.86 |
| Gaussian | 15 | 1.5 | 14.34 |
| 30x30 | Single Action | 5 | 1 | 20 |
| Double Action | 5 | 1 | 20 |
| Uniform | 15 | 1 | 4.42 |
| Gaussian | 5 | 0.5 | 6.76 |

Table 7 - Best Configurations of all Scenarios in which AlphaZero plays Zombie

We can nicely see the tendency of the amount of searches to stick with the lowest possible for most simple agents (Single and Double agents), while the number of searches increases as the complexity of the agent increase (Gaussian and Uniform agents).

* Note that the claim above is relevant for the case of the Uniform agent in the 20x20 board. The optimal number of searches could be 15 as the difference is neglegible (9.73 and 9.86).

For complete closure, the mean reward graph of the best agents is presented:

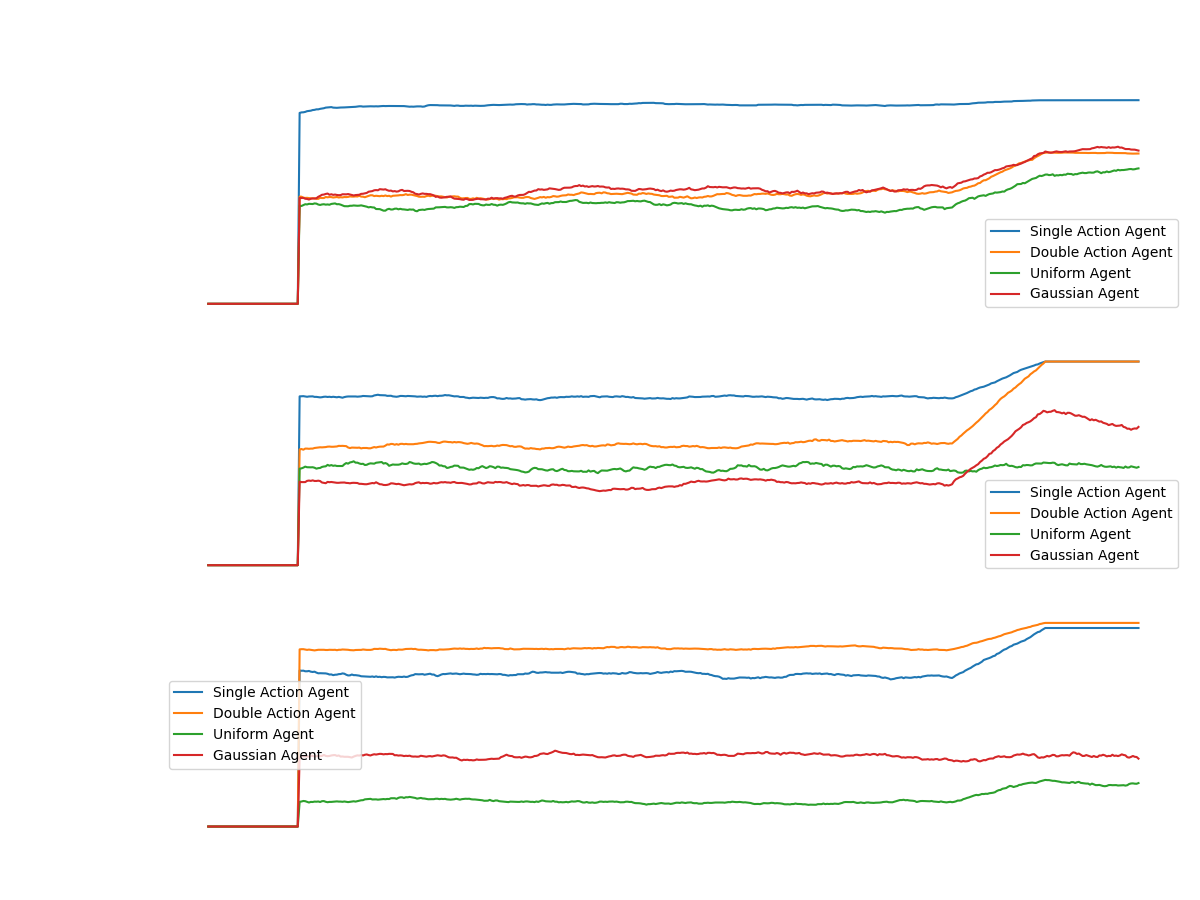


Figure 28 - Comparing the results of the AlphaZero agent as the Zombie Player, with the best parameters over the different four simple competitors

From the graph above, it seems that in cases where the agent fails to converge to successful strategy (compared to DDQN), there is a learning barrier at the beginning of the process.

This barrier could be due to a poor configuration of the AlphaZero algorithm. As well as the partial observability compared to the success the agent reaches as the Light Player (next section).

##### Alpha Zero as Light Player

Now, we begin testing the results of the AlphaZero algorithm, remember that as a Light Player, the agent gets information about the strength of the zombies at each step of the environment - unlike the zombie player.

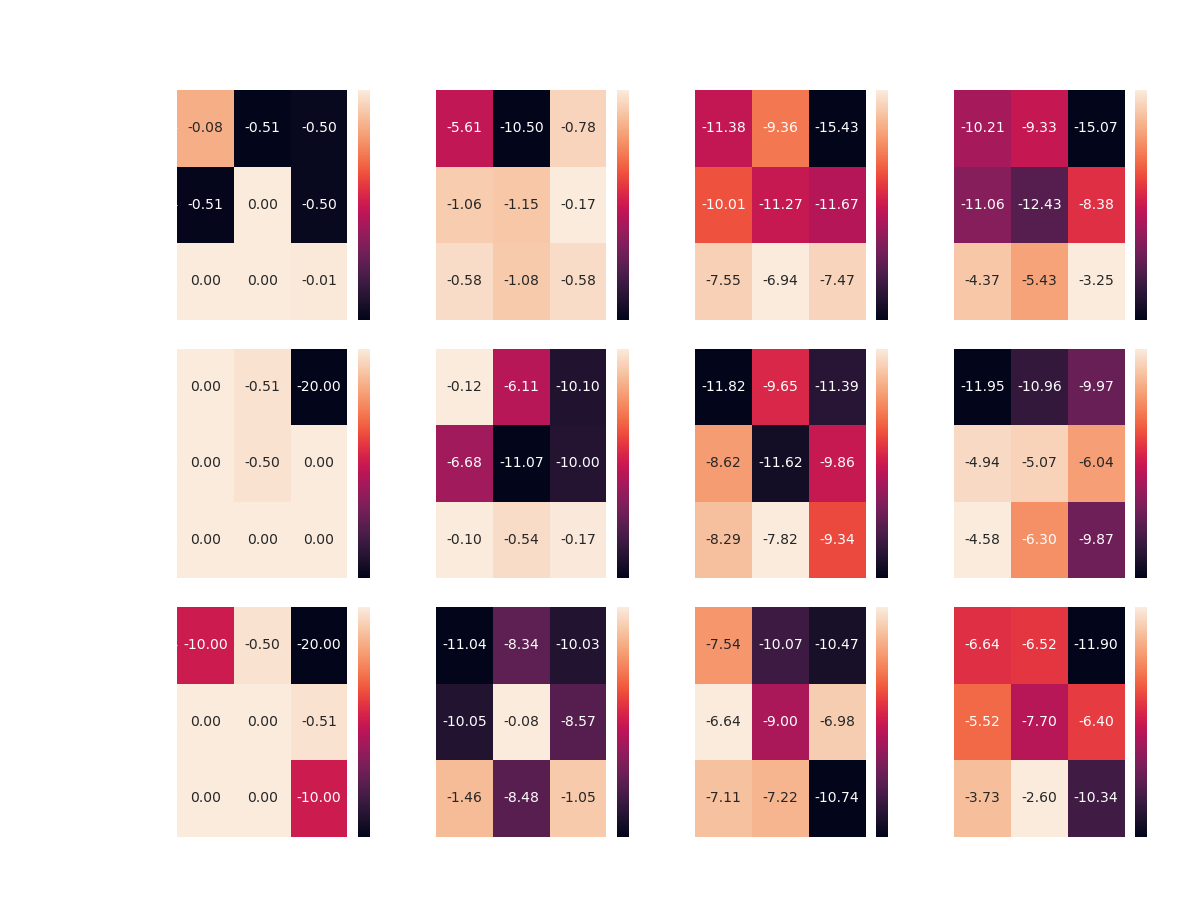


Figure 29 - A summary of all the scenarios that AlphaZero plays Light. Heat-Maps of the Average Test Reward

As expected, the agent comes up with an optimal strategy In addition, in an unprecedented way so far in the project, the agent manages to come up with a successful strategy against the random agents the simple agents.

In addition, in an unprecedented way so far in the project, the agent manages to come up with a successful strategy against the random agents.

In both games with the 20 and 30 square boards, the agent surpassed the DDQN agent so that in the game against the Gaussian agent he gained almost the maximum possible reward.

Again, let's take a look over the summary of best configurations:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Board size | Competitor | Monte Carlo searches | Exploration rate | Average Test Reward |
| 10x10 | Single Action | 15 | 0.5 | 0 |
| Double Action | 10 | 1.5 | -0.17 |
| Uniform | 15 | 1 | -6.94 |
| Gaussian | 15 | 1.5 | -3.25 |
| 20x20 | Single Action | 5 | 0.5 | 0 |
| Double Action | 15 | 0.5 | -0.1 |
| Uniform | 15 | 1 | -7.82 |
| Gaussian | 15 | 0.5 | -4.58 |
| 30x30 | Single Action | 10 | 0.5 | 0 |
| Double Action | 10 | 1 | -0.08 |
| Uniform | 10 | 0.5 | -6.64 |
| Gaussian | 15 | 1 | -2.6 |

Table 8 - Best Configurations of all Scenarios in which AlphaZero plays Light

As the light player, we notice that the algorithm performs well with the greater amounts of tree searches, which it is reasonable since the light player action space is larger (exponential greater compared to zombie player).

At last, the mean of the rewards for the best agents for all boards:

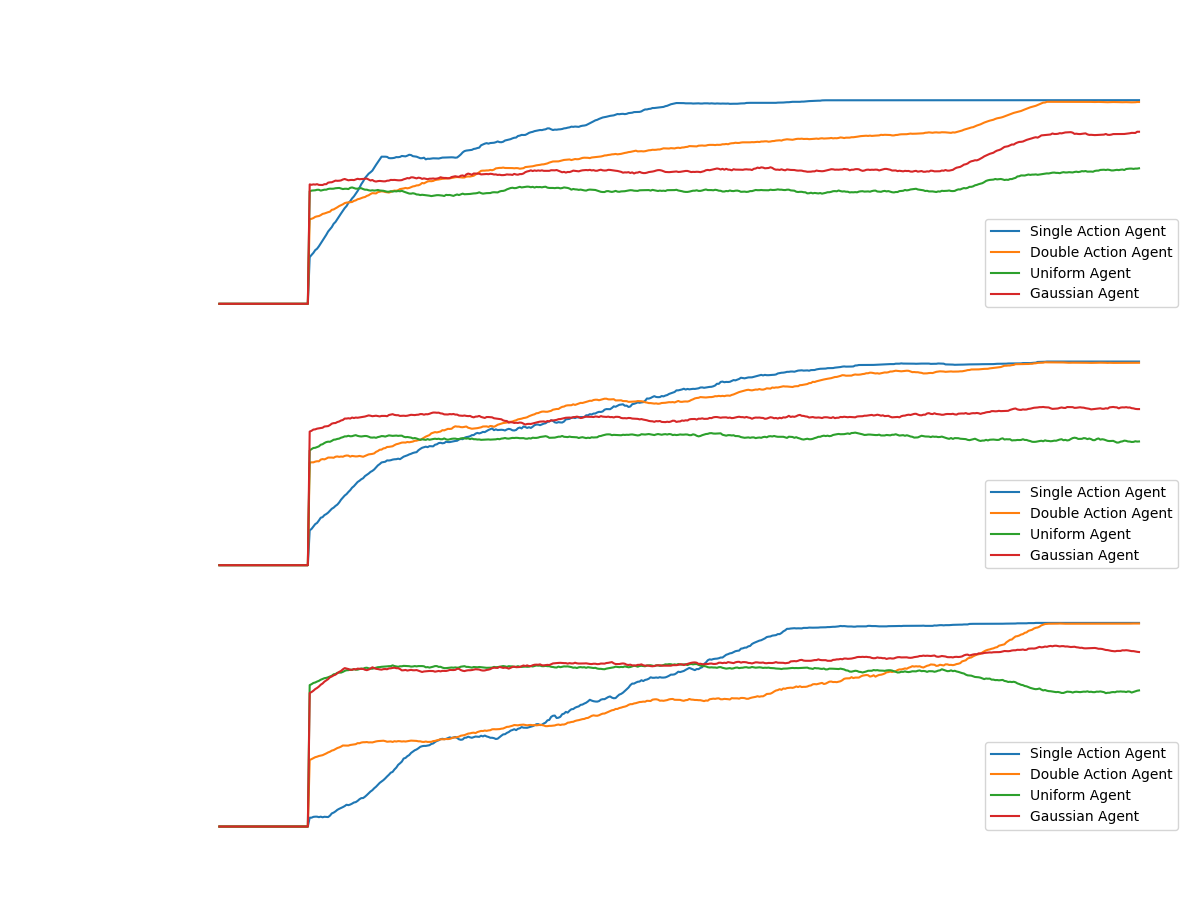


Figure 30 - Comparing the results of the AlphaZero agent as the Light Player, with the best parameters over the different four simple competitors

According to Figure 30, we can see the learning curve of the most successful agents in each game.

The most striking thing that catches our eyes in games against the random agents, is the fact that AlphaZero agent quickly converged into a strategy unlike those against the simple agents.

The main reason for such cases is the fact that the algorithm is based on the Monte Carlo Tree. Which updates the State-Action values ​​according to the rewards from the environment.

In games against the simple agents, AlphaZero agent spends most of the time looking for actions which will reward him positively and finds this only in a very limited number of situations, since the zombies travel in unique places over the board.

In contrast to the games against random agents. Which the rewards are more likely to appear much more frequently and the craft of searching is simpler.

* Note, Once the agent finds a variety of states that are rewarding in different ways, the network can learn. Which happens very quickly (as seen in Figure 30) in games against the random agents.

## Competing AlphaZero and DDQN

We have now reached the final stage of the project, a competition between learning algorithms. After we seen comparisons of each of the learning algorithms against the simple agents, and in this chapter, we will see a comparison between the learning agents.

Continuing from the previous stages where we taught the agents and pushed them to the edge of the ability limit and we looked for the best set of parameters for them in different boards.

Now, it's time to take the best agents in each game board and let them compete with each other, once as the light player and once as the zombie player.

To do so, we will perform the following steps:

1. Save the models (network modes) with the best configurations of the two agents in all boards (from tables: Table 4, Table 5, Table 7 and Table 8), once as the light player and once as the zombie player (12 models in total)
   * Each time we gather the best configuration of a scenario, we give great preference to the configurations of the games against the Uniform Agent (since, this is the most complex) while still considering the others.
2. Prove the convergence of the chosen agent relative to the rest of the simple agents.
3. Build Metrics and graphs to describe the learning process (considering networks parameters and agents' actions).
4. Run and analyze the scenarios of the competition:
   1. AlphaZero as light vs DDQN as Zombie – 3 scenarios
   2. DDQN as light vs AlphaZero as Zombie – 3 scenarios

After gathering the best models of the DDQN agent against the Uniform Agent, we will now compare his performance against the three agents: Single, Double and Gaussian. Below is a summary of the results in binary view, significant success as 1 or not as 0:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Board Size | Model | Single | Double | Gaussian |
| As light | 10x10 | DDQN | 1 | 1 | 1 |
| 20x20 | DDQN | 1 | 1 | 1 |
| 30x30 | DDQN | 0 | 0 | 0 |
| As zombie | 10x10 | DDQN | 0 | 0 | 1 |
| 20x20 | DDQN | 0 | 0 | 1 |
| 30x30 | DDQN | 0 | 0 | 1 |

Table 9 - Summary of competing DDQN Agent vs. Single, Double and Gaussian Agents

As can be seen in Table 9, there is significant success against the Gaussian Agent. This is in contrast to a very limited success against the simplest agents. This is because the best model was trained against the Uniform agent which is very similar to the Gaussian agent.

Hence, we would like to expand the experience of our final model with the goal of defeating all simple agents. Which, as we have seen before (see Figure 21 and Figure 24), is achievable.

### Constructing the Composite Agent

The original idea that stands behind preparing the learning agents for battle, was to train them against the most complex not-learning-agent we have and test their performance against the others, hasn't led to any success.

Therefore, we had to take another approach in order to generify the agent strategy and its total potential against other unknown agents (ex. The battle they are facing).

For that effort, we built a new agent called 'Composite Agent' that is the combination of all simple agents.

Its' strategy set to imitate one of the four simple agents at a time, for full episode period (from the first zombie enters the board until the last exit). After that, the agent will replace the strategy to the next in line, keep it for a full episode and continue so.

The order of strategies loop is: SingleAction -> DoubleAction -> Gaussian -> Uniform -> SingleAction etc...

By that, we get to learn from data samples generated from constant AND random agents, all at the same train process. With the expectation for the Composite-Agent's strategy to out-perform each of the simple agents, separately.

### Final Evaluation

After training the DDQN Agent against the Composite Agent, we will examine his success against the other simple agents:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Board Size | Model | Single | Double | Gaussian | Uniform |
| As light | 10x10 | DDQN | 1 | 1 | 1 | 1 |
| 20x20 | DDQN | 1 | 1 | 1 | 1 |
| 30x30 | DDQN | 0 | 0 | 1 | 1 |
| As zombie | 10x10 | DDQN | 1 | 1 | 1 | 1 |
| 20x20 | DDQN | 1 | 1 | 1 | 0 |
| 30x30 | DDQN | 0 | 0 | 1 | 0 |

Table 10 – DDQN trained with Composite agents, evaluation vs. all Simple Agents

In general, there has been significant success in training against the uniformed agent (see Table 9).

In the same way, we will take the AlphaZero agent and we will train it in games against the Composite agent.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Board Size | Model | SingleAction | DoubleAction | Gaussian | Uniform |
| As light | 10x10 | AlphaZero | 1 | 0 | 0 | 0 |
| 20x20 | AlphaZero | 0 | 0 | 1 | 1 |
| 30x30 | AlphaZero | 0 | 1 | 1 | 1 |
| As zombie | 10x10 | AlphaZero | 0 | 0 | 1 | 1 |
| 20x20 | AlphaZero | 1 | 0 | 1 | 1 |
| 30x30 | AlphaZero | 1 | 0 | 0 | 0 |

Table 11 - AlphaZero trained with Composite Agent, evaluation vs. all Simple Agents

Again, we achieved successful and satisfying results.

### The Competition

Now that we have trained our two learning agents: DDQN and AlphaZero to the best we could, in every game board and on each side of it. We will examine the success of the agents against each other.

#### Action and Reward Distribution

Before we dive into the competition, we need to understand the visualization we are going to use throughout the games. In addition to the reward graphs that we already familiar with (see "Game Scenarios"), we will show the distribution of agents' actions in the following graph (for example):

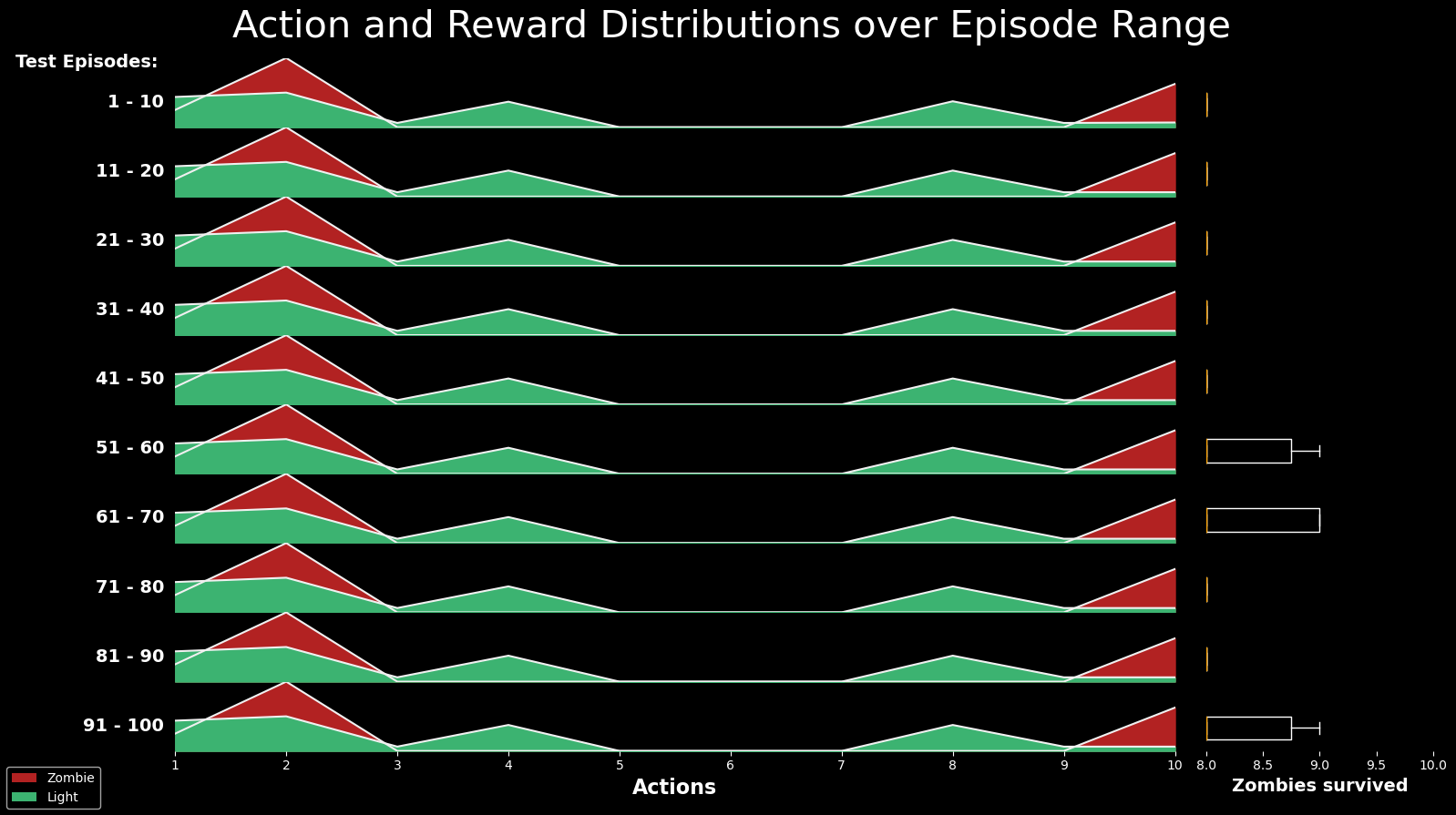


Figure 31 - Action and Rewards distribution graph

For understanding Figure 31, we will elaborate on each of the three graphs separately:

1. In the left graph, we present the range of episodes on which the data is based.
2. The middle graph shows the distribution of each player's actions. The action space of the light player is two-dimensional while the zombie player's action space is one-dimensional - as the number of rows of the board. Therefore, to bridge between these two, we decided to present the row of each selected action, for the two players.  
   In this way, the two-dimensional action of the light player is summed up to the row in which it was taken, and for the zombie player, the action is also the row.
   * Note for the graph in Figure 31, the actions the Zombie Players play are - rows 2 and 10. While the Light Player the actions 1,2,4 and 8 (mainly). The light mark is with size of 3x3, which means that expect from the Light Agent to place its mark on the rows of the zombies and the two above them (row 1 for action 2 and rows 8,9 for action 10 of the Zombie Player), from the reason that it is likely to hit the zombies and eventually profit from it. Conversely, in the absence of light, the zombie agent's profit could be observed with the same logic.
3. The right graph shows the box-plot of rewards for the episodes in the current range.

Finally, we have to say that this is an insightful and yet incomplete graph, with regards to the light agent. Since, it is not possible to understand in which column he chose to place the cursor.

Now, in order to compare the performance of the agents, we will take the converged models and pass them on to the agents, each with its own set of parameters (see Table 4 and Table 5).   
We will start by presenting the reward throughout the game of each game board. Following, we will analyze the actions the agents have chosen.

#### DDQN as Light Player vs AlphaZero as Zombie Player

##### Analysis of 10x10 board

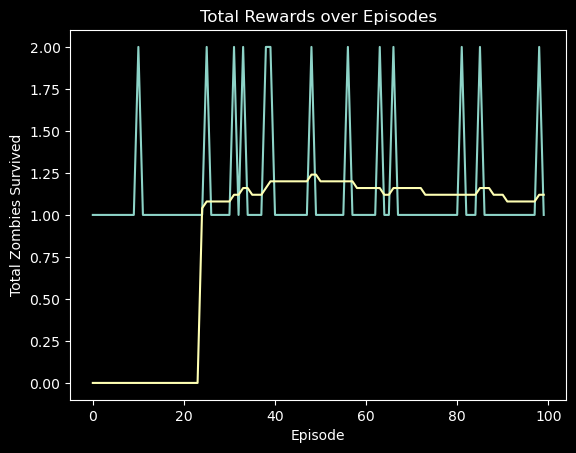


Figure 32 - Rewards over Episodes, board 10x10

There is a significant defeat of the Zombie Player - AlphaZero. With victory of the Light Player - the DDQN, since, the maximum reward that can be received in the game is 20 while on average only one zombie has survived each game.

We will now look at the distribution of agents' actions:

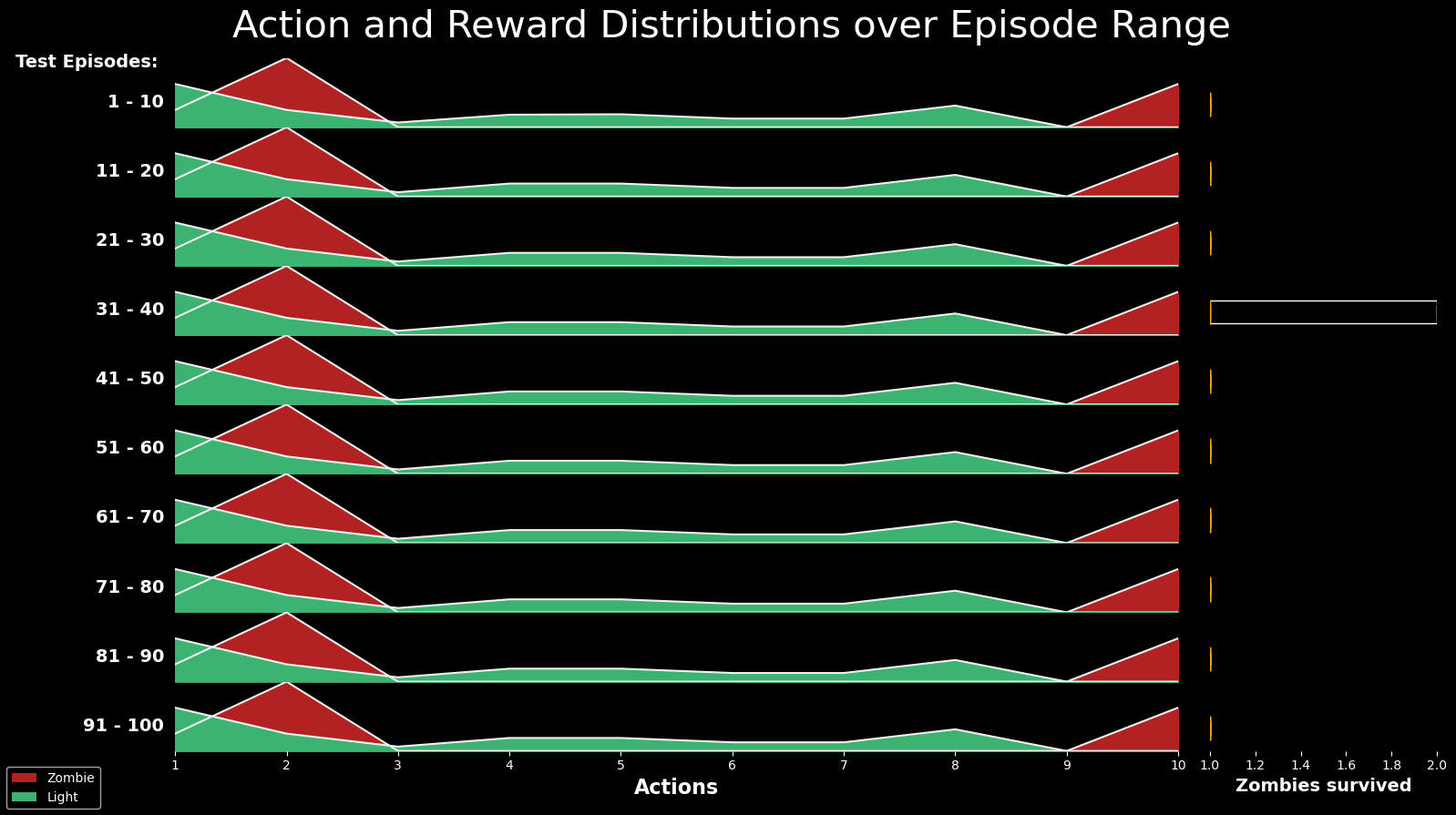


Figure 33 - Action and Reward Distributions, board 10x10

The Zombie Player seems to have chosen to take out zombies from the rows: 2,10 pretty exclusively. In contrast, the light player chose to mark a variety of lines in the range of 1-8, these are smart choices. It is not surprising that the light player has achieved more significant results throughout all games.

In general, it seems that the light player knew how to hit all the lines that would lead to significant damage to the zombies, since the light size is 3X3.

On the right side of the graph, you can see the number of zombies (this value is based on the reward in all games) that survived throughout the episode range, in most runs, the value is stable on one zombie surviving in each episode.

##### Analysis of 20x20 board

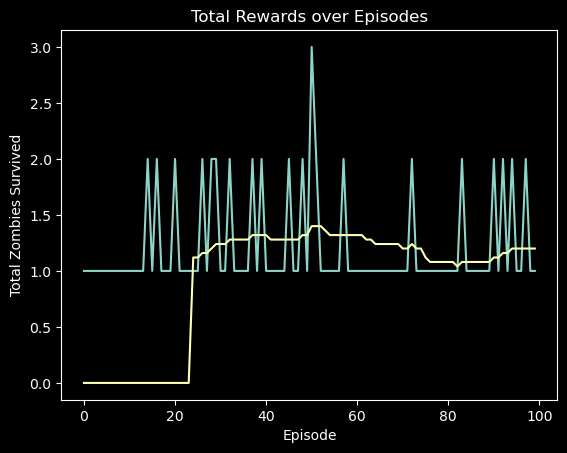


Figure 34 - Rewards over Episodes, board 20x20

This time, the picture looks almost the same, there is a significant victory in favor of the light player – DDQN agent. The players' rewards are stable throughout the scenario on the mean of 1.

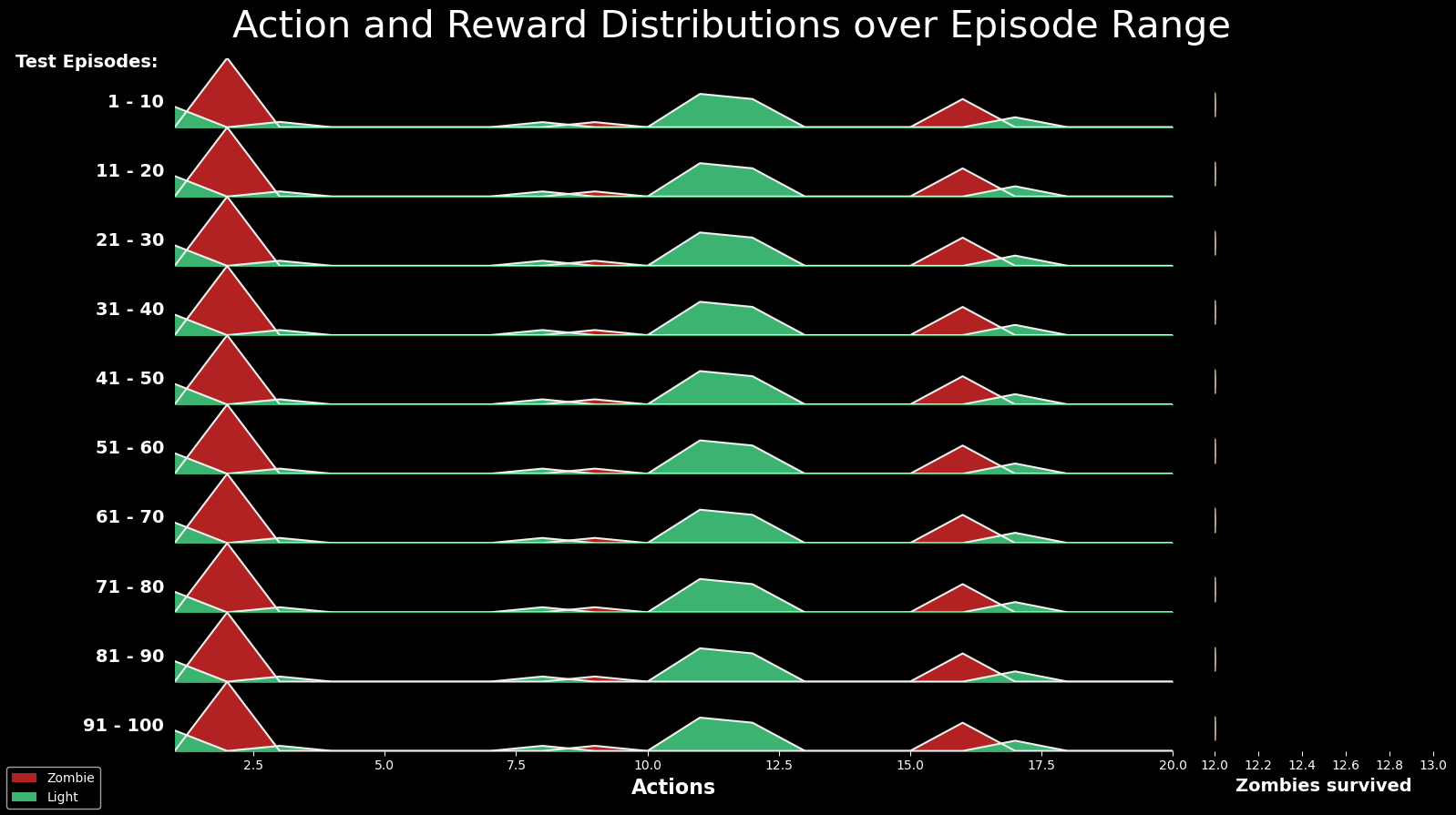
We will now look at the distribution of agents' actions: 

Figure 35 - Action and Reward Distributions, board 20x20

The Zombie Agent (AlphaZero) has chosen two main strategic locations: the second and 16th rows. While the light player has chosen to place the light mainly in rows 1, 11 and 12. These rows have the potential to critically hit zombies, since the light size in this board is 6X6.

##### Analysis of 30x30 board

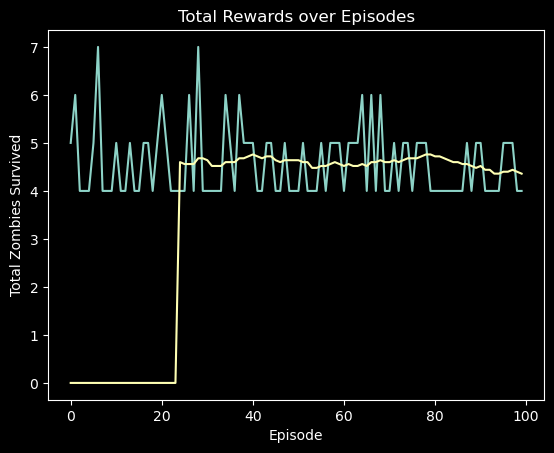


Figure 36 - Rewards over Episodes, board 30x30

The reward graph shows that once again, the scenario tends in favor of the light player – DDQN. The average of the zombies that survive in the episode, throughout the games is stable at 4.5 out of 20.

We will now look at the distribution of agents' actions: 

Figure 37 - Action and Reward Distributions, board 30x30

The Light Player, DDQN, chose to place the light in several rows on the board and significantly in: 4, 8, 16, 18, 22, 30. While the Zombie Player chose to send zombies from row 22 exclusively.

The Light Player's state processing was not good enough in this case. Zombies came out from line 22 and yet the Light Player chose to mark mainly the lines that wouldn't reward him at all (line 8 was chosen the most, with no ability to hit zombies).

Despite this, the light marking hit the zombies when placed in lines 12-18, and that was enough to get a significant advantage and win this scenario.

#### AlphaZero as Light Player vs DDQN as Zombie Player

##### Analysis of 10x10 board

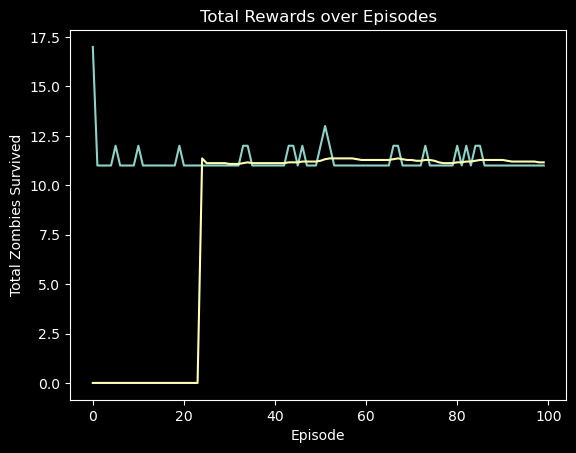


Figure 38 - Rewards over Episodes, board 10x10

Unlike the analyzes in the [previous section](#_DDQN_as_Light), this time one can see the advantage of the zombie player who is none other than - DDQN. The reward value in each episode stabilizes at 11.5 which means that on average there are 11.5 zombies who successfully pass the game board out of 20.

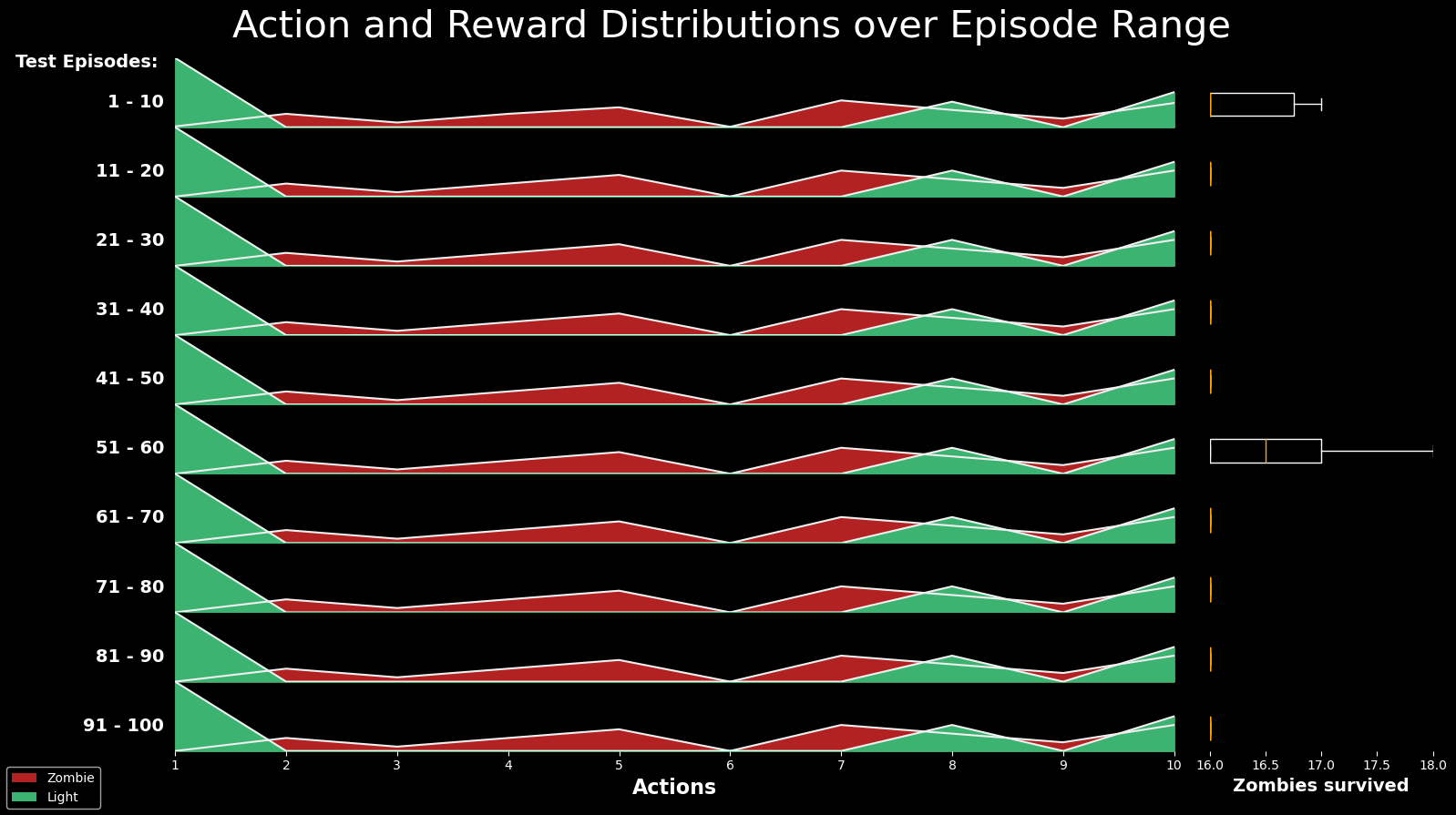
We will now look at the distribution of agents' actions: 

Figure 39 - Action and Reward Distributions, board 10x10

According to Figure 39, the Zombie Player chose to take zombies out of rows 2,4,5,7,9,10 mostly. While the light player chose to mark the lines: 1, 8 and 10 exclusively.

Most of the zombies in the game advanced on lines 5 and 7, the light player did not choose to mark these lines at all for the adjacent ones (remember that the light size in this case is 3X3).

##### Analysis of 20x20 board

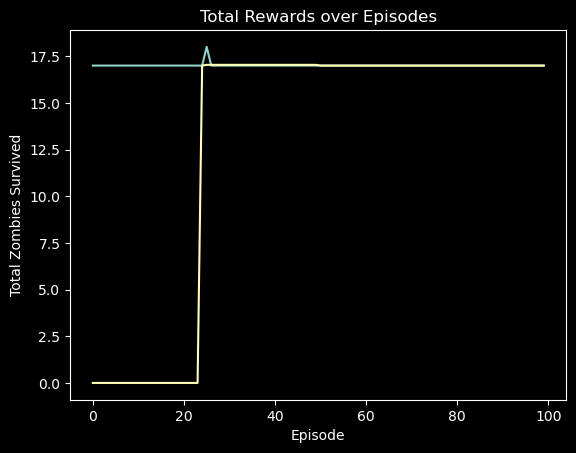


Figure 40 - Rewards over Episodes, board 20x20

A significant victory of the Zombie Player can be seen in the case of a 20X20 board; there is a stabilization of the number of zombies who survived in each game, 17 out of a maximum of 20.

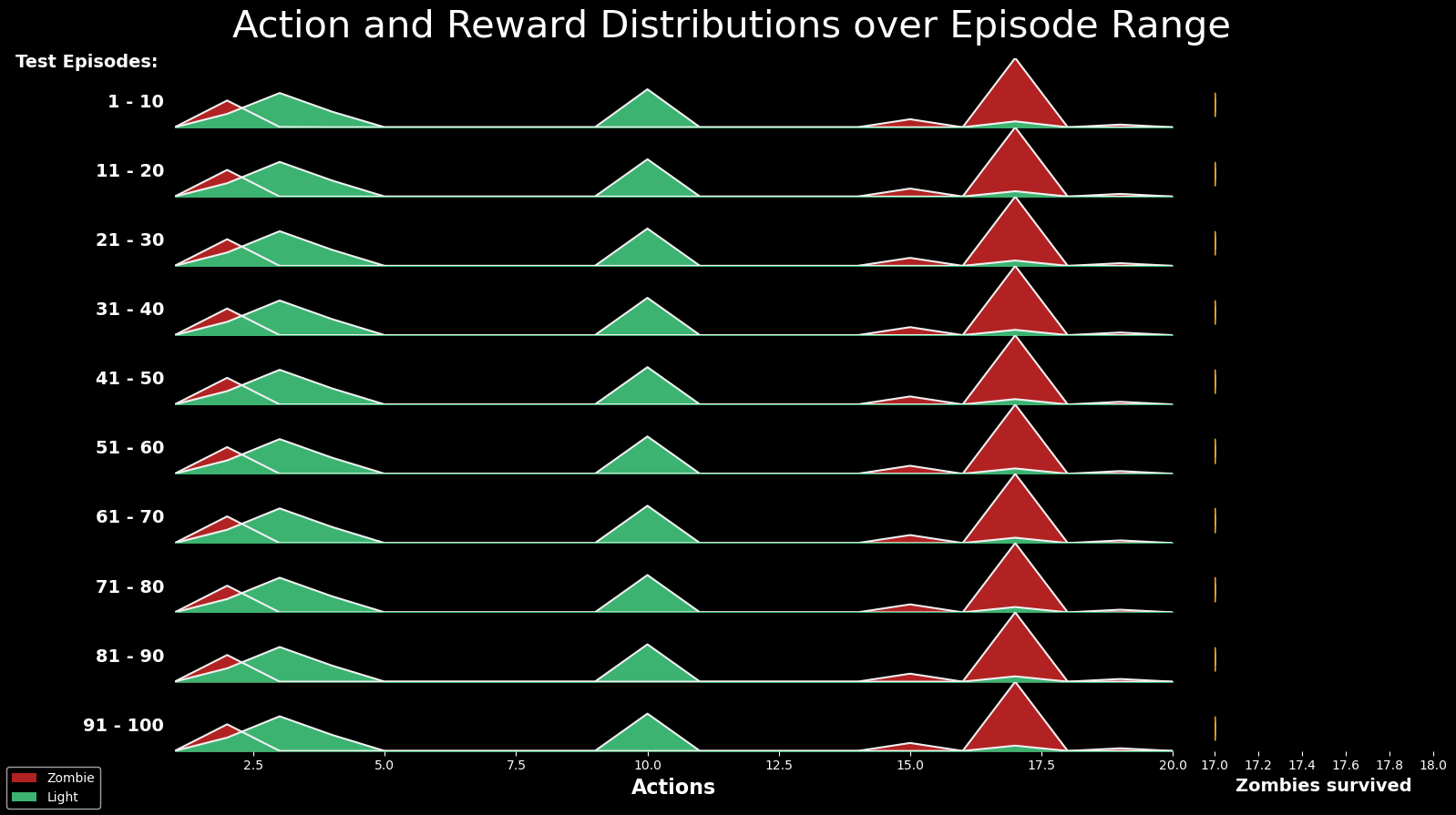
We will now look at the distribution of agents' actions: 

Figure 41 - Action and Reward Distributions, board 20x20

According to Figure 41, The light player chooses to place his cursor in positions 2,3,4 and 10 while the Zombie Player takes an absolute majority of the zombies out of lines 2,15 and especially 17.

Since the Light marker size is 6X6 in the current scenario - Indeed, the Light Player (AlphaZero Agent) fails to find and hasn't harmed zombies almost at all.

##### Analysis of 30x30 board



Figure 42 - Rewards over Episodes, board 30x30

Once again, there is almost a balance between the successes of the agents with a tendency in favor of the Zombie Player (the DDQN agent).

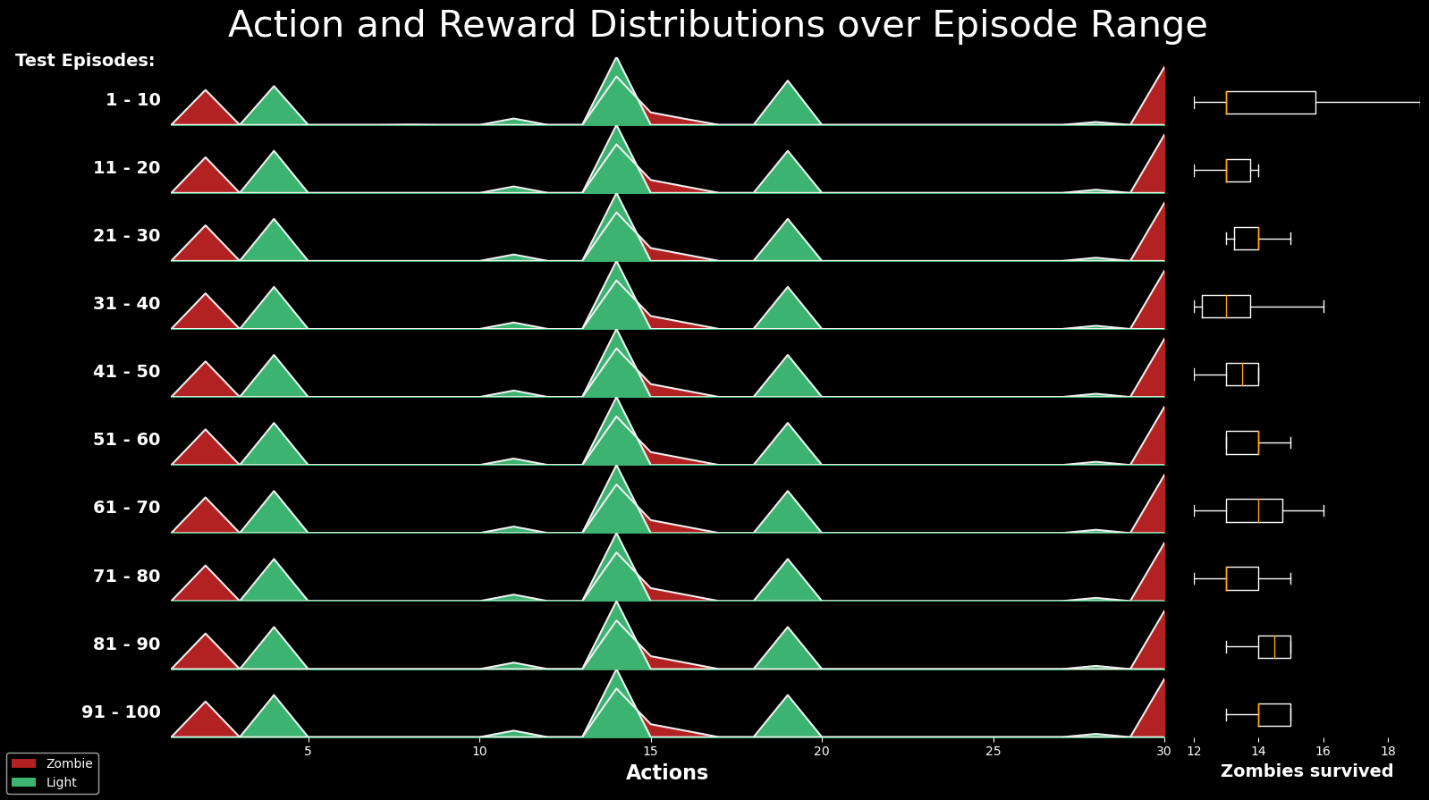
We will now look at the distribution of agents' actions: 

Figure 43 - Action and Reward Distributions, board 30x30

According to Figure 43, the light agent chose to place the cursor in rows 4,14,19,27 while the zombie agent chose to take out the zombies in rows 2,14,15,30.

If we look at the light agent's row selection zones, there seems to be a high overlap with the zombies coming out, in rows 14-15. This overlap led to the same relatively balanced result but the victory of the zombies is still significant, since, the light player did not treat / hit at all two major concentrations of the zombies in rows 2 and 30.

## Conclusions and Next Steps

Throughout the project we implemented three learning algorithms, the Double Deep Q-Network (DDQN), Monte Carlo Tree Search (MCTS) and AlphaZero. The general perception about them is that DDQN is the most basic algorithm that can be implemented to achieve convergence in the least complicated scenarios (smallest board sizes). Next, the MCTS is the best algorithm we have found outside of the field of Reinforcement Learning. An algorithm that will be used for comparison and a supportive measure to success of algorithms in the field of Reinforcement Learning. Finally, the AlphaZero, a state-of-the-art algorithm designed to defeat all the previous ones and achieve unprecedented results on all the scenarios we ran during the project.

During the project we examined the nature of the algorithms and their limits in two types of scenarios, against agents using a pure / mixed strategy based on known distributions, and against learned agents according to the various algorithms we implemented.

After training each of the algorithms we chose against each of the simple agents (constant and random), it was found that the DDQN algorithm achieved a stable and successful convergence against the constant agents in all board sizes, but without any success in complex scenarios (board sizes of 20 and 30).

In practice, as we have seen, the DDQN did converge as we expected, only in the simplest cases but in a stable and reliable manner. In contrast, the other two did not live up to expectations. The MCTS showed only basic capabilities, and did not converge at all the scenarios we chose. Mainly Because of the RAM limitations of the machine on which the scenarios ran, according to the implementation we chose, as explained in ‎9.2.2 - MCTS Results and Conclusions.

Unlike DDQN-based models, AlphaZero did not show any stability on the scenarios it converged on.   
All the above could be due to the specific implementation we chose or the long duration of each episode relative to DDQN which potentially forced us to learn for an insufficient number of episodes of AlphaZero algorithm. (While it takes DDQN about two hours to execute 1000 episodes in a given scenario, we ran AlphaZero on 400 episodes on the same Scenario and it took approximately two days).

However, it must be said that AlphaZero's episode durations being longer and slower than DDQN is understandable mainly due to the reason that during network training (similar to what happens in DDQN), the AlphaZero algorithm builds its MCTS tree, which is probably not as optimal as just training a network like we do in DDQN. (The network training process takes place within an advanced package like TensorFlow (see [18]) that uses machine level language performance. While building and searching on the MCTS happens via Python with the implementation we chose). In addition, it must be said that in this work we have tried to optimize (via parallelism and computation theory) all the processes that take place outside the network and it may be further improved.

Next, the goal before our eyes was to let the two learning agents (DDQN and AlphaZero) compete against each other in all forms of play. To do so, we trained each of the algorithms against the simple composite agent (see ‎10.1: Constructing the Composite Agent), so that they could contain and deal with each of the simple agents' strategies, in order to formulate a stronger strategy, in preparation for the competition.

Each of the learning algorithms was trained against the composite agent, and the learned models were maintained to test their performance against each of the four simple agents as seen in (link to table of zeros and uniformity). Having reached the required convergence (according to the overall success of the algorithms against each of the simple agents separately), we started a competition between them in a way that each of the learning agents takes a particular player and then the other.

In all competition scenarios (see ‎10.3: The Competition), the DDQN algorithm managed to overcome or substantially defeat AlphaZero.

We sum up the conclusions by saying that although the results we obtained are not intuitive, we are satisfied and completed the purpose of the project, since all the agents we implemented showed a positive and significant learning curve, while playing on top of the environment we have built.

Furthermore, we will expand on additional directions for further research, given additional time and computing resources (RAM and processing units).

First, it should be said that training of larger network-based models (we will mention the full implementation of AlphaZero as mentioned in ‎9.3.5) and with more attention to their hyper parameters can make a significant difference in the results.

After examining the performance of the expanded models, one can invest in a more parallel and cost-effective implementation of the Monte-Carlo-Tree-Search algorithm and thus get its results and comparability to the other learning algorithms, against the simple agents and also in competitions against them.

In the aspect of learning algorithms, it is possible to examine the possibility of implementing additional algorithms from the A3C family and more.

Next, the resolution of the simulation (board size) and rest of the simulation parameters (see ‎8.3) could be configured for further analysis. Another idea is to enable the zombie player to choose different angles and velocities for its zombies.

## References

[1] Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994* (pp. 157-163). Morgan Kaufmann.  
[2] Omidshafiei, S., Pazis, J., Amato, C., How, J. P., & Vian, J. (2017, August). Deep decentralized multi-task multi-agent reinforcement learning under partial observability. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (pp. 2681-2690). JMLR. org.

[3] Peshkin, L., Kim, K. E., Meuleau, N., & Kaelbling, L. P. (2001). Learning to cooperate via policy search. arXiv preprint cs/0105032.

[4] Dutech, A., Buffet, O., & Charpillet, F. (2001, August). Multi-agent systems by incremental gradient reinforcement learning. In International Joint Conference on Artificial Intelligence (Vol. 17, No. 1, pp. 833-838). LAWRENCE ERLBAUM ASSOCIATES LTD.

[5] Wu, F., Zilberstein, S., & Chen, X. (2012). Rollout sampling policy iteration for decentralized POMDPs. arXiv preprint arXiv:1203.3528.

[6] Liu, M., Amato, C., Anesta, E. P., Griffith, J. D., & How, J. P. (2016, March). Learning for decentralized control of multiagent systems in large, partially-observable stochastic environments. In Thirtieth AAAI Conference on Artificial Intelligence.

[7] Matignon, L., Laurent, G. J., & Le Fort-Piat, N. (2007, October). Hysteretic q-learning: an algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (pp. 64-69). IEEE.

[8] Matignon, L., Laurent, G. J., & Le Fort-Piat, N. (2007, October). Hysteretic q-learning: an algorithm for decentralized reinforcement learning in cooperative multi-agent teams. In 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (pp. 64-69). IEEE.

[9] Conitzer, V., & Sandholm, T. (2007). AWESOME: A general multiagent learning algorithm that converges in self-play and learns a best response against stationary opponents. Machine Learning, 67(1-2), 23-43.

[10] Gaina, R. D., Couëtoux, A., Soemers, D. J., Winands, M. H., Vodopivec, T., Kirchgeßner, F., ... & Perez-Liebana, D. (2017). The 2016 two-player gvgai competition. IEEE Transactions on Games, 10(2), 209-220.

[11] Vodopivec, T., Samothrakis,dz S., & Ster, B. (2017). On Monte Carlo tree search and reinforcement learning. Journal of Artificial Intelligence Research, 60, 881-936.

[12] T. Imagawa and T. Kaneko, "Enhancements in Monte Carlo tree search algorithms for biased game trees," 2015 IEEE Conference on Computational Intelligence and Games (CIG), Tainan, 2015, pp. 43-50, doi: 10.1109/CIG.2015.7317924.

[13] Silver, David; Hubert, Thomas; Schrittwieser, Julian; Antonoglou, Ioannis; Lai, Matthew; Guez, Arthur; Lanctot, Marc; Sifre, Laurent; Kumaran, Dharshan; Graepel, Thore; Lillicrap, Timothy; Simonyan, Karen; Hassabis, Demis (December 5, 2017). "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm".

[14] Silver, David; Schrittwieser, Julian; Simonyan, Karen; Antonoglou, Ioannis; Huang, Aja; Guez, Arthur; Hubert, Thomas; Baker, Lucas; Lai, Matthew; Bolton, Adrian; Chen, Yutian; Lillicrap, Timothy; Hui, Fan; Sifre, Laurent; Van Den Driessche, George; Graepel, Thore; Hassabis, Demis (2017). "Mastering the game of Go without human knowledge"  
[15] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016). <https://doi.org/10.1038/nature16961> Mastering the game of Go with deep neural networks and tree search

[16] <https://github.com/suragnair/alpha-zero-general>

[17] van Hasselt, Hado, Guez, Arthur and Silver, David. "Deep Reinforcement Learning with Double Q-learning." (2015)  
[18] TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

## Appendix A – Elaboration of Related Literature

This section provides further elaboration on the basic assumptions of modeling in the field of reinforcement learning. All the ideas presented here have been researched but will not be reflected in our project.

### Reinforcement Learning

Reinforcement learning is an area of Machine Learning. It is about taking suitable action to maximize reward in a particular situation. Essentially an agent (or several) is built such that it can perceive and interpret the environment in which is placed, furthermore, it can take actions and interact with it.  
Basic reinforcement learning problems are modeled as a Markov Decision process (MDP) which is a 4-tuple , where:

* is a finite set of states.
* is a finite set of actions.
* is the probability that action  in state at time will lead to state , due to action .
* is the immediate reward (or expected immediate reward) received after transitioning from state to state , due to action .

The goal is to learn a policy that maximizes the cumulative sum of discounted rewards

where are the rewards and is a discount factor, tuning parameter through which we can influence the amount of weight we give to future awards in relation to the immediate reward.

We can split the subject of RL into two main partitions: ***Model-Free*** and ***Model-Based***. In Model-Free RL,  
the agent does not have access to a model of the environment (The agent couldn’t estimate the consequences of his actions). In Model-Based RL, the agent has access to a model of the environment.  
Our focus is on the Model-Free type of learning mainly due to the advantage that it doesn’t require a model of the environment.

The Model-Free learning can be considered as two parts of ***off-policy***learning and ***on-policy***learning. an agent might be acting using one or two control policies. In *on-policy* control the agent is evaluating and simultaneously improving the exact policy that it follows. Conversely, in *off-policy* control, the agent is following one policy, but may be evaluating another – it is following a behavior policy while evaluating a target policy. In our work we will implement some off-policy algorithms alongside an algorithm from the tree search area called MCTS for comparison and evaluation.

### *Stochastic Games*

In this paper, two-player zero-sum Stochastic Games (SGs) are considered. These games proceed like MDPs, with the exception that in each state, both players select their own actions simultaneously, which jointly determine the transition probabilities and their rewards. The zero-sum property restricts that the two players’ payoffs sum to zero.

A *Stochastic Game* (SG) is a tuple , Where:

* N is the number of the players/agents
* T: is the transition function
* is the action set for the player
* is the discount factor
* : is the reward function for player

The objective of the n agents is to find a deterministic joint policy (aka. joint strategy aka. strategy profile) (where ) so as to maximize the expected sum of their discounted payoffs. The Q-function, , is the expected sum of discounted payoffs given that the agents play joint action in state and follow policy thereafter. The optimal -function ­ is the -function for (each) optimal policy . So, captures the game structure. The agents generally do not know in advance. Sometimes, they know neither the payoff structure nor the transition probabilities.

For example, consider a zero-sum game with two players, one player (Player 1) wants to maximize his/her total reward, the other (Player 2) would like to minimize that amount. Similar to the case of MDPs, the reward can be discounted or undiscounted, and the game can be episodic or non-episodic.

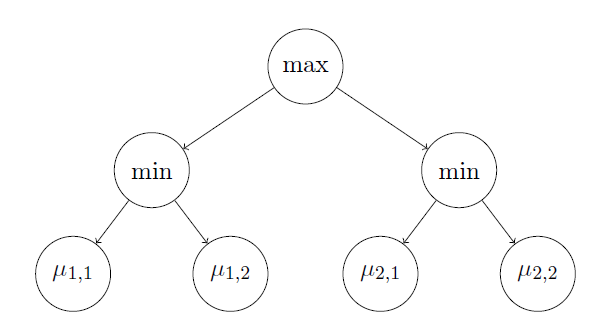


Figure 44 – Game tree when there are two actions by player

We consider a two-player two-round zero-sum game, in which player A has available actions. For each of these actions, indexed by , player B can then choose among possible actions, indexed by . , when player A chooses action and then player B chooses action j, the probability that player A wins is . We investigate the situation (see Figure 44 for an example) from the perspective of Player A, who wants to identify a maximin action

Assuming that Player B is strategic and picks, whatever A’s action , the action minimizing , this is the best choice for A.

### *Nash Equilibrium in SGs*

"In Game Theory, A Nash Equilibrium is a stable state of a system that involves several interacting participants in which no participant can gain by a change of strategy as long as all the other participants remain unchanged"   
Princeton University

A Nash equilibrium is a joint strategy where each agent’s is a best response to the others. For a stochastic game, each agent’s strategy is defined over the entire time horizon of the game.

Given a with players, a Nash Equilibrium is a tuple of strategies such that for all and ,

Where, is the set of strategies available to agent , And,

.

Is the discounted sum of rewards, with discount factor .

A Nash equilibrium is strict if the inequality above is strict. An optimal Nash equilibrium is a Nash equilibrium that gives the agents the maximal expected sum of discounted payoffs.

In the literature, SGs are typically learned under two different settings, and we will call them online and offline settings, respectively. In the offline setting, the learner controls both players in a centralized manner, and the goal is to find the equilibrium of the game [9]. This is also known as finding the worst-case optimality for each player (a.k.a. maximin or minimax policy). In this case, we care about the sample complexity, i.e., how many samples are required to estimate the worst-case optimality such that the error is below some threshold. In the online setting, the learner controls only one of the players, and plays against an arbitrary opponent [10]. In this case, we care about the learner’s regret, i.e., the difference between some benchmark measure and the learner’s total reward earned in the learning process. This benchmark can be defined as the total reward when both players play optimal policies [3], or when Player 1 plays the best stationary response to Player 2. Some of the above online-setting algorithms can find the equilibrium simply through self-playing.

### *Learning in SGs*

Learning in stochastic games can be formalized as a multi-agent reinforcement learning (MARL) problem. we can say that the goal of RL is to learn equilibrium strategies through interaction with the environment.

Our work focuses on competitive settings with partially-observable MARL that has received limited attention throughout the years [2]. There were works include model-free gradient-ascent based method [3][4], simulator-supported methods to improve policies using a series of linear programs [5], Recent scalable methods use Expectation Maximization to learn finite state controller (FSC) policies [6].

The most interesting approach I've found related to our problem of competitive relation between the agents and partial observability framework is described in DEC-HDRQNS [2], that means a Decentralized Hysteretic Deep Recurrent Q-Networks model. Their approach is model-free and decentralized, learning Q-values for each agent. In contrast to policy tables or FSCs, Q-values are amenable to the multi-task distillation process as they inherently measure quality of all actions, rather than just the optimal action.

The proposed approach takes into consideration the concept of Hysteresis (lag) [8].  
Overly-optimistic MARL approaches completely ignore low returns, which are assumed to be caused by teammates’ exploratory actions. This causes severe overestimation of Q-values in stochastic domains.  
Hysteretic Q-learning, instead, uses the insight that low returns may also be caused by domain stochasticity, which should not be ignored. This approach uses two learning rates: nominal learning rate, α, is used when the TD-error is non-negative; a smaller learning rate, β, is used otherwise (where 0 < β < α < 1).