Final Project – Progress Report
Building Learning Agents to Play the Game of Light Against Zombies

Eliav Shalelashvili
Project supervised by Dr Yehudit Aperstein
Afeka

1. Table of Contents

2. Figure List

3. Table List

4. Abstract

In the following document we present the progress report of the final project.

During the past months, we've been busy building the framework of the two-player-game described by the partners in the aerospace industry. A Framework 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 9 - Building the Framework9.

Furthermore, we implemented an algorithm from the Reinforcement Learning domain called: Double-Deep-Q-Network. Our framework provides each player the ability to play by the algorithm, DDQN and any other! Our learning agent is used for performance test of the framework in Chapter 10 - Framework Performance Test and in Chapter 11 - Double Deep Q-Network Evaluation we provide in-depth analysis of it, in manners of achieving optimal policy in various scenarios. Additional analysis of the algorithm is mentioned in Appendix A – DDQN Evaluation.

For the next stage of the project, we added theoretical and technical elaboration of the algorithms we want to implement and evaluate: Monte-Carlo-Tree-Search and Alpha-Zero.

To that end, we've refactored Chapter 6 - Literature Review.
We have transferred to Appendix B the basic review and all the extensions without direct relevance to the current project. so that literature review the review remains relevant to our project and the continuation of the work. In that way, Chapter 6 - Literature Review contains the relevant ideas that will be used by us in the next stage of the project.

5. 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.

In this project we are going to build a family of stochastic games that characterized with extreme state-space complexity. Each of our stochastic games consists of two players with non-symmetric action space which are 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 that last that uses state-of-the-art algorithms. The combination and analysis of agents from different areas of research will provide us a broad understanding of the field that will produce the most successful agents for our family of games.

6. Literature Review

The area of learning agents that master a particular game or on the other side, agents that seek the highest score over a set of games, grew to huge scales 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: Appendix A –

6.1 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 the 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 the MCTS [12] algorithm. All of these have proven themselves in the competitions, therefor, might be useful and beneficial to our research.

The platform of GVG-AI letting 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, I 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 I 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 and its aim is to either avoid dying or catch the hunter. While the hunter shoots missiles and moves faster than the ghost and its goal is to avoid the ghost that can hurt him and shoot the ghost.

The algorithms that achieved the best results in the competition are called: MCTS and MaastCTS2 which are both 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). And on the other hand, the hunter that shoot 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 which both are located in a two-dimensional square which 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 make him lose some health points. The winner is the player that survives tor the one 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. 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 like, the 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.

6.2 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 based on random sampling of the search space. The application of Monte Carlo tree search in games is based on many playouts, also called roll-outs. 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.

Thus, each round of Monte Carlo tree search 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 $\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$ Where:
        - w_i is the total reward aggregated in the current node
        - n_i is the number of visits through the current node
        - N_i is the number of visits through parent node
        - C is the exploration factor
- *Expansion*: Unless L 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 1:

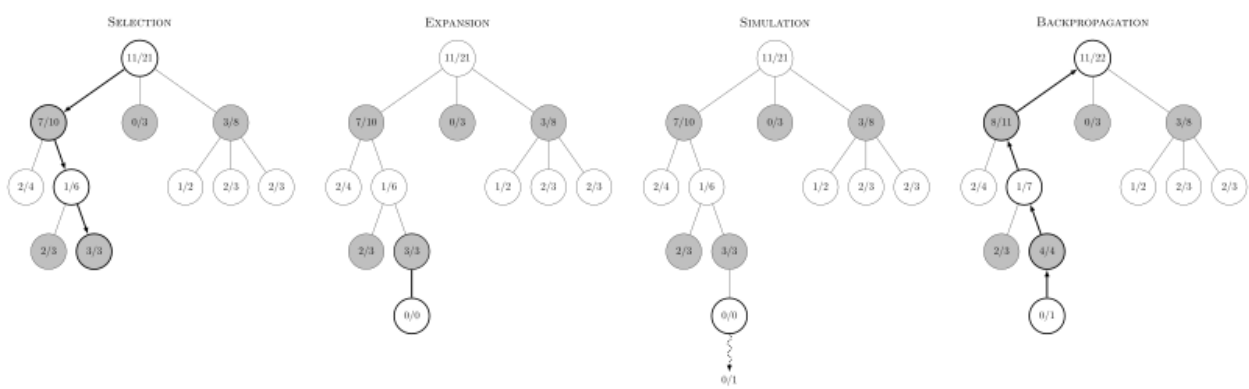

Figure 1 - Step of Monte Carlo tree search

6.3 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.

6.3.1 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 $L(y', y)$. In this case, the y' 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:

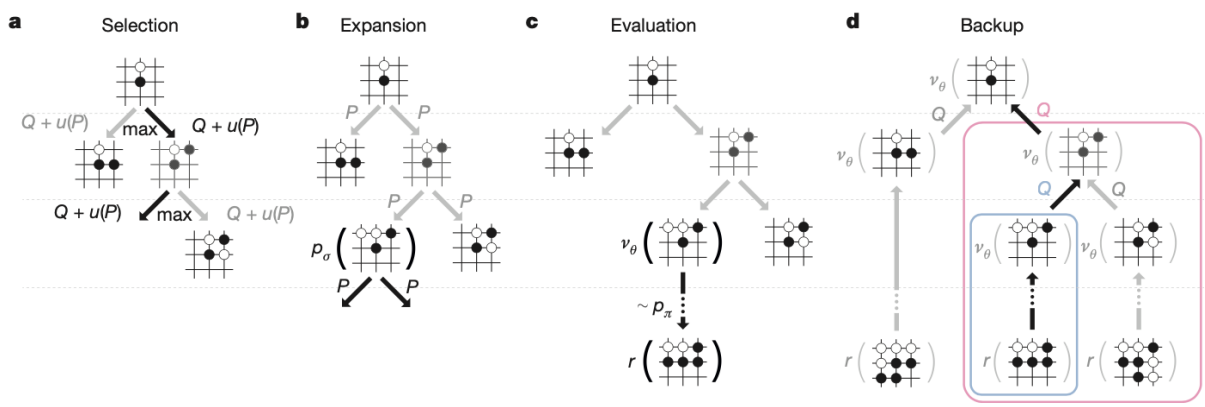

Figure 2 - AlphaGo overview

6.3.2 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:

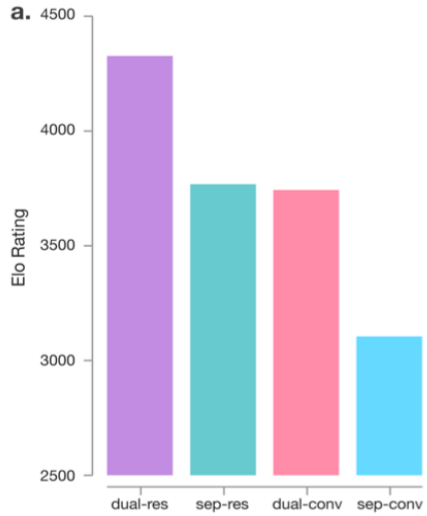

Figure 3 - Resnet contribution over traditional CNNs

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:

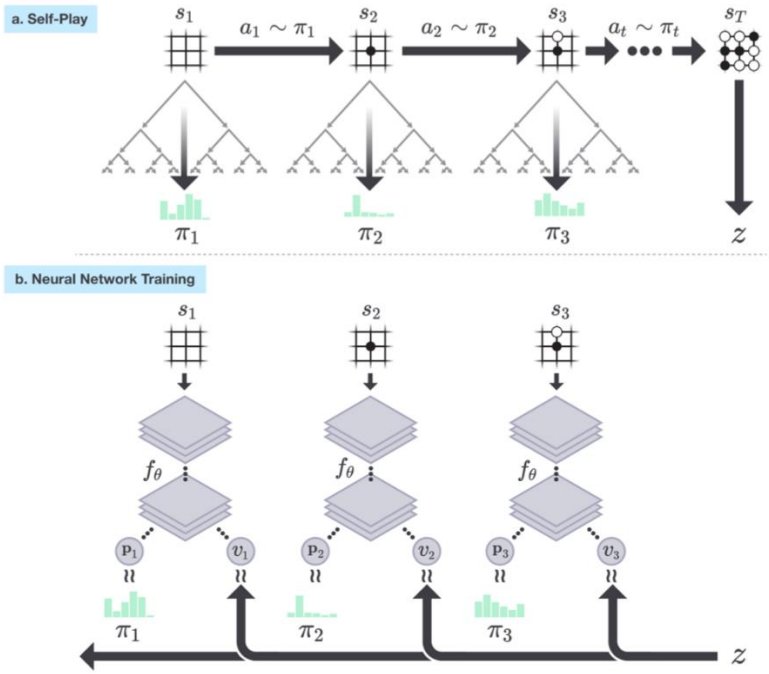

Figure 4 - AlphaGo Zero training characteristics

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.

6.3.3 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.

6.3.3.1. The Neural Network

Unsurprisingly, there's a neural network at the core of things. The neural network f_θ is parameterized by θ and takes as input the state s of the board. It has two outputs: a continuous value of the board state $v_\theta(s) \in [-1,1]$ from the perspective of the current player, and a policy $p_\theta(s)$ 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 (s_t, π_t, z_t). π_t is an estimate of the policy from state s_t (we'll get to how πt is arrived at in the next section), $and\ z_t \in \{-1,1\}$ is the final outcome of the game from the perspective of the player at s_t. The neural network is then trained to minimize the following loss function (excluding regularization terms):

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \pi_t \cdot log(p_\theta(s_t))$$

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.

6.3.3.2. Monte Carlo Tree Search for Policy Improvement

Given a state s, the neural network provides an estimate of the policy p_θ. 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 $i \rightarrow j$ if a valid action can cause state transition from state i to j. 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:

- $Q(s,a)$: the expected reward for taking action aa from state s, i.e., the Q values
- $N(s,a)$: the number of times we took action aa from state s across simulations
- $P(s,\cdot) = p_\theta(s)$: the initial estimate of taking an action from the state ss according to the policy returned by the current neural network.

From these, we can calculate $U(s,a)$, the upper confidence bound on the Q-values as:

$$U(s,a) = Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\Sigma_b N(s,b)}}{1 + N(s,a)}$$

Here c_{puct} 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 ss as the root. A single simulation proceeds as follows. We compute the action a that maximizes the upper confidence bound $U(s,a)$. If the next state s' (obtained by playing action a on state s) 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 $P(s',\cdot) = p_\theta(s')$ and the value $v(s') = v_\theta(s')$ from the neural network, and initialize $Q(s',a)$ and $N(s',a)$ to 0 for all a. Instead of performing a rollout, we then propagate $v(s')$ up along the path seen in the current simulation and update all $Q(s,a)$ 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 $N(s,a)$ values at the root provide a better approximation for the policy. The improved stochastic policy $\pi(s)$ is simply the normalized counts $N(s,\cdot)/\sum_b(N(s,b))$. During self-play, we perform MCTS and pick a move by sampling a move from the improved policy $\pi(s)$.

6.3.3.3. 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 s_t. We pick a move by sampling from the improved policy π_t. This gives us a training example (s_t, π_t, r_t). The reward r_t 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.

6.3.3.4. 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 <u>binary win/loss</u> outcomes. AlphaZero instead estimates and optimizes the expected outcome, **taking account of draws or potentially other outcomes**. The rules of Go are <u>invariant to rotation and reflection</u>. 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 approach takes the bold ideas 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, on top of our game environment: 'Light vs Zombies'.

6.3.4 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.

7. Our Model - Zombie Invasion Problem

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.

7.1 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.

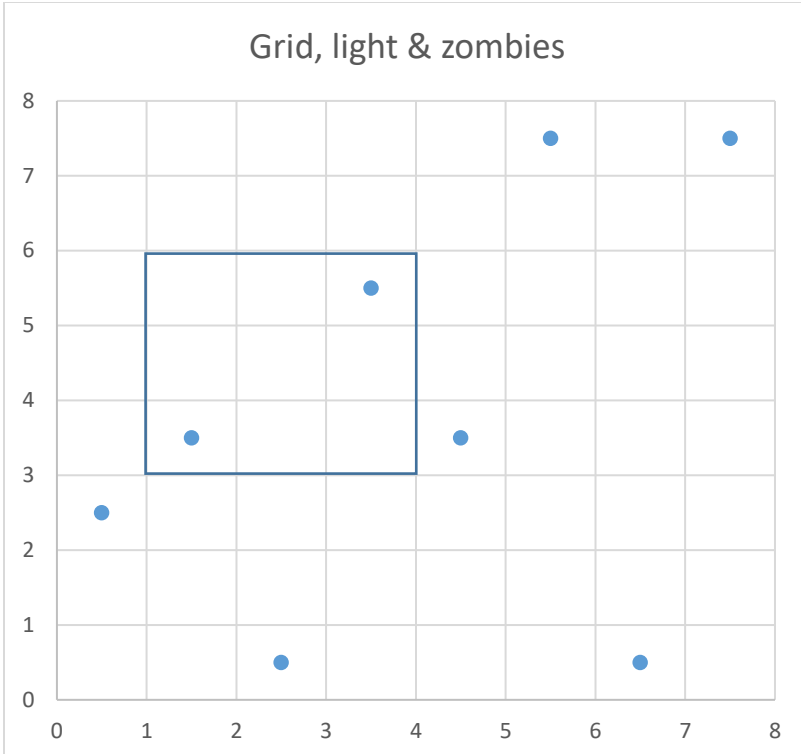

Figure 2: grid set-up for example with light-mark of 3x3 and 8 zombies

7.2 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 $G = (n, S, A_1, A_2, T, \gamma, R_1, R_2)$ where:

6.3.4. $n = 2$ is the number of players
6.3.5. $\mathrm{T}: \mathrm{S} \times \mathrm{A}_1 \times \mathrm{A}_2 \times \mathrm{S} \rightarrow [0,1]$ is the transition function
6.3.6. $\mathrm{A_i}, \mathrm{i} = 1,2$ is the action set for the player i
6.3.7. $\gamma \in [0,1]$ is the discount factor (for now $\gamma = 1$)
6.3.8. $\mathrm{R_i}: \mathrm{S} \times \mathrm{A}_1 \times \mathrm{A}_2 \times \mathrm{S} \rightarrow \mathbb{R}, \mathrm{i} = 1,2$ is the reward function for player i

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

$$R_1(s_t, a_t^1, a_t^2, s_{t+1}) + R_2(s_t, a_t^1, a_t^2, s_{t+1}) = 0, \quad \forall (s_t, a_t^1, a_t^2, s_{t+1}) \in S \times A_1 \times A_2 \times S$$

with limited information on one side (asymmetric information).

7.3 Agents

7.3.1 Zombie Player

The first player is the Zombie Player, its <u>objective</u> 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.
<u>Action</u> is an integer number from 0 to N (N is the size of the board). Therefore, the actions it can take are the set: $A_1 = \{1,2,\ldots,N\}$.

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

$I_t^1 = \{S_t, a_t^1\}$, where $(S_t)_{i,j} = \begin{cases} 1 & there\ is\ a\ zombie\ in\ (i,j)\ cell \\ 0 & otherwise \end{cases}$

Denote a subset of all observations until time t and actions until time t-1 by $I_{1:t}^1$. Therefore $I_{1:t}^1$contains a set of t N-by-M matrices and a history of choices from the set A_1.

7.3.2 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:

$$A_2 = \left\{(x,y): x \in \left\{\frac{A-1}{2}, \frac{A-1}{2}+1, \ldots, M - \frac{A-1}{2}\right\}, y \in \left\{\frac{A-1}{2}, \frac{A-1}{2}+1, \ldots, N - \frac{A-1}{2}\right\}\right\}$$

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$\big(S(t)\big)_{N\times M}$, such that $S_{i,j}(t) = \begin{cases} 1 & there\ is\ a\ zombie\ in\ (i,j)\ cell \\ 0 & otherwise \end{cases}$
  - A zombie strength matrix, with non-empty cells at zombie location$\big(H(t)\big)_{N\times M}$, such that
$$(H_t)_{i,j} = \begin{cases} 0 & (S_t)_{n,m} = 0 \\ h_{i,j}(t) > 0 & othewise \end{cases}$$
- The mark (light) at time t, $A(t)$, i.e., the player's action $a_2(t) \in A_2$

Therefore, the available information at time t is

$$I_t^2 = \{S(t), H(t), A(t)\}$$

Denote a subset of all observations until time t and actions until time t-1 by $I_{1:t}^2$

7.4 Game rules

The game has discrete clock, in each clock tick:

- The Zombie Player decides where a new zombie will appear. The new zombie's hit points is equal to one.
  All previous existing zombies are moved right 1 cell, i.e. $S_{i,j}(t+1) = 1$ if $S_{i-1,j}(t) = 1$ for $1 <$

$i \leq N$

In general, $S_{i,j}(t+1) = \begin{cases} 1 & S_{i-1,j}(t) = 1 \quad or \quad a_1(t+1) = j \\ 0 & otherwise \end{cases}$

- Zombies that go over the right boundary disappear
- Each zombie that inside marked region (light) got additional hit of the amount c. Meaning the hit points are increased by c (default value c=1)
- Each remaining zombie heal itself by multiplying hit point by (1-epsilon) factor. Thus:

$$h_{i,j}(t+1) = \begin{cases} h_{i-1,j}(t) + 1 & (i,j) \in A(t) \\ (1-\varepsilon)h_{i-1,j}(t) & otherwise \end{cases}$$

- Once all hit points are calculated and falling zombies are removed from the board, there is a "kill process" that might remove some zombies from the board
- For each zombie a utility function U (see Figure 5) is calculated based on the hit point. U produce values from [0,1] such that zombie with no hit point get 0 and zombies with large value of hit points get utility close to 1

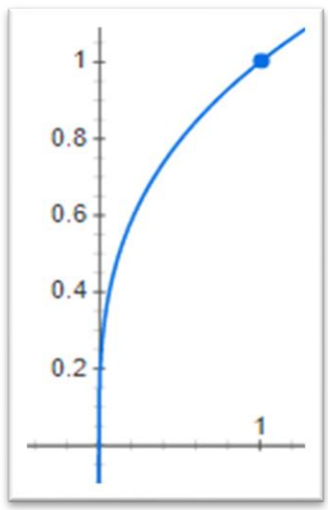

Figure 5 - $\sqrt[3]{x}$ is the utility function of our game.
For some hit-point value we calculate $\sqrt[3]{hit\_point}$, which determines the chances of a zombie to not pass the right border

- If the outcome is positive, zombie is removed from the board
- The reward (for Zombie Player) for the round is computed and its equal to the number of zombies that are still in play
- The round ends by plants selecting a new place for the light

8. Next Steps

- Build the entire environment of the game
  - will contain two agents' possible interactions
  - will be compatible with the OpenAI gym framework to enable potential of wider research.
- Examine the proposed algorithms
  - DQNs – basic and successful
  - MCTS – from the tree search area – will use for reference

    - Combining RL with MCTS – AlphaGo Zero, AlphaZero
- Test the results over a different utility and reward functions
- Increase degree of simulation precision
    - Consider round markings, finer resolution, continuous coordinates etc.

9. Building the Framework

Building a simulation for Reinforcement learning purposes is mostly a manner of creating an environment and throw there some entities that follow certain rules.

In our case the entities thrown into the environment are zombies and light.

The managers of the above entities are the Zombie and Light Players. The Zombie Player has the ability to place a zombie at some starting position as it wishes, and the Light Player has the ability to place the light somewhere on top the board.

Once the Zombie Master ordered to place a zombie, the only rule for the zombie is move straight to the other side of the board.

9.1 Architecture

The two parts of the simulation are: environment and entities.

9.1.1 Environment

As said above, the environment is storage place of all entities that are going to join the simulation. Since we are facing a 2D board game, the environment implemented as a grid with cells. Each cell can accommodate up to one zombie.

On top of the grid, there is the environment class which is able to query and contact the zombies inside the grid. In addition, all the outer communication from the environment is managed by the environment manger. Its purpose is to pass the environment to the learning agents command and process/reshape the environment state before sending forward.

Over view of the above:

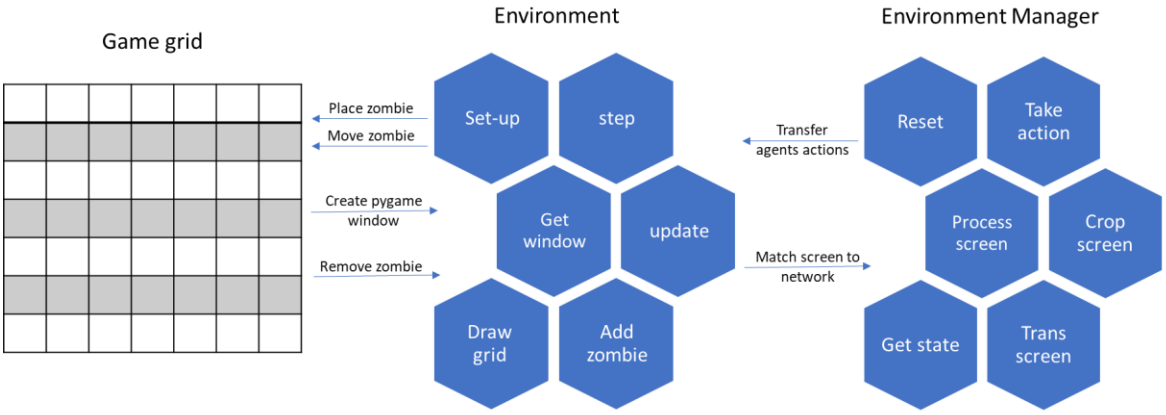

Figure 6 – simulation architecture

9.1.2 Entities

There are four types of entities living in our environment that are:

- Light Player
- Light
- Zombie Player
- Zombie

The Light and Zombie Players are intelligent agents with the ability to create zombies and move the light accordingly. On the other hand, the zombie and light per se are only data entities that are using to visualize the state in some conditions.

9.2 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 20 classes and abstract classes
- Four simple agents
- 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 10, 11 and forth) are automatically generated by the framework after each scenario and batch of scenarios

9.3 Implemented Agents

As mentioned above, we implemented four simple agents and three algorithmic agents. From a given set of N 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 – middle action
    - Standard deviation – $\frac{N}{5}$

And the algorithmic agent is called Double-Deep-Q-Network Agent. More details provided at: "The Algorithm – Double Deep Q Network"

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

- Interactive mode – Boolean, whether or not 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

```
[MainInfo]
interactive_mode = false
display_width = 1600
display_height = 800
num_train_episodes = 800
num_test_episodes = 200
zombies_per_episode = 20
check_point = 25
light_size = 2
board_height = 30
board_width = 30
max_angle = 0
max_velocity = 1
dt = 1
max_hit_points = 1
heal_points = 0.03
```

Figure 7- Frameworks' Main configuration example

Next, we have to 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 the uniform distribution
- Each agent must implement some basic methods in order to participate the game:

```
@abstractmethod
def select_action(self, state):
    raise NotImplementedError

@abstractmethod
def learn(self, state, action, next_state, reward):
    raise NotImplementedError

@abstractmethod
def reset(self):
    raise NotImplementedError
```

Figure 8 - Abstract methods every agent should implement in order to join the Framework

Finally, we are able to run the framework with the agents we have built:

```
# create the game with the required agents
env = Game(light_agent=DdqnAgent, zombie_agent=UniformAgent)

# play the game and produce the dictionaries of the results
episodes_dict, steps_dict_light, steps_dict_zombie = env.play_zero_sum_game(dir_path)
```

Figure 9 - Two simple lines of code to run the Framework and get results

To sum up:

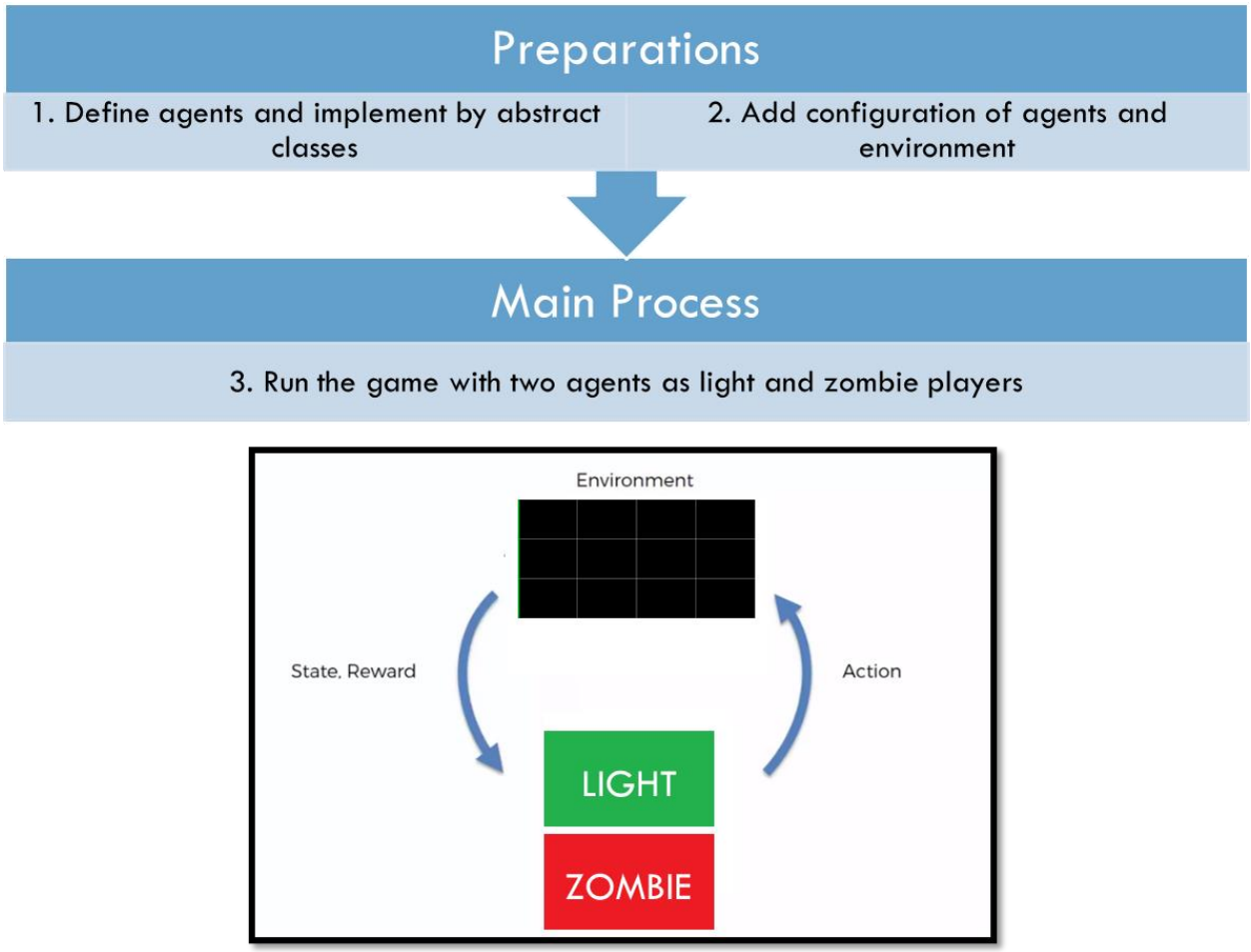

Figure 10 - Summary - three steps for running the Framework

10. Framework Performance 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.

10.1 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.

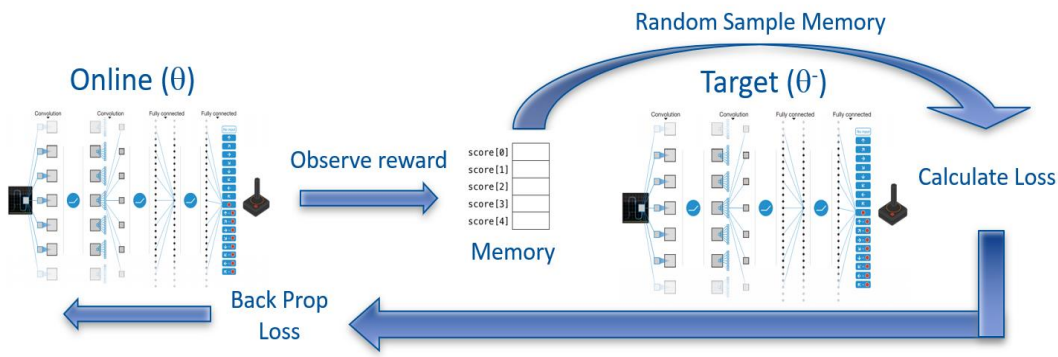

Figure 11 – Double Deep Q-Network architecture and flow

The learning algorithm we used in this project called: 'Double Deep Q Netowrk'.
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 11 that sums up the whole idea.

10.2 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:

$$end + (start - end) * e^{-step \times decay}$$

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 $2/_{(numEpisodes \times stepsPerEpisode)}$, for achieving the start-end values of the epsilon function.

Which with 100,000 steps looks like:

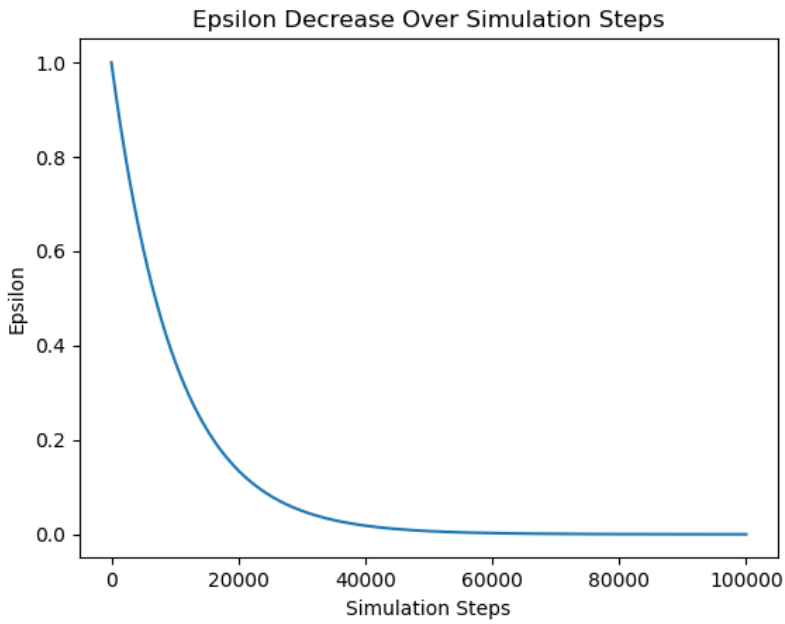

Figure 12 - Epsilon greedy values – convex function of $e^{-0.0001x}$

10.3 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:

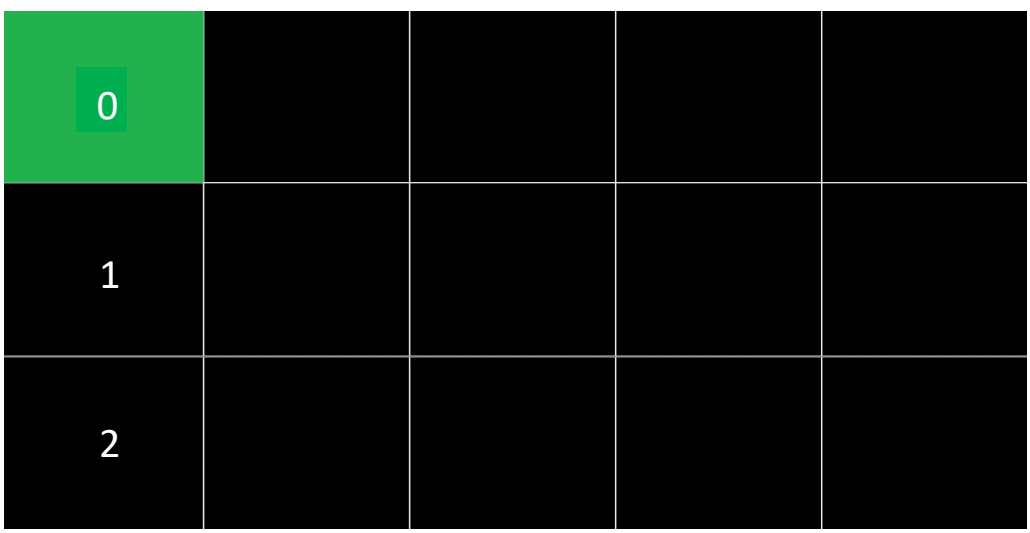

Figure 13 – environment set-up for zombie Player performance check with optional actions

As we can see in Figure 13 – 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:

Table 1 – learning parameters while evaluating the zombie Player

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

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):

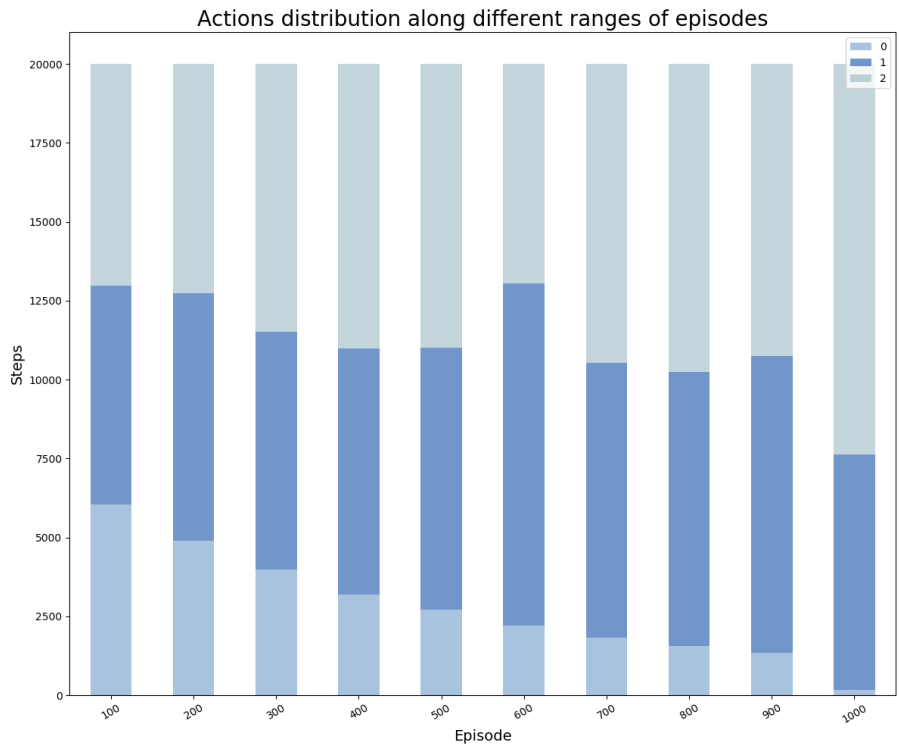

Figure 14 – Zombie Player actions distribution along different ranges of episodes

In Figure 14 – 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.

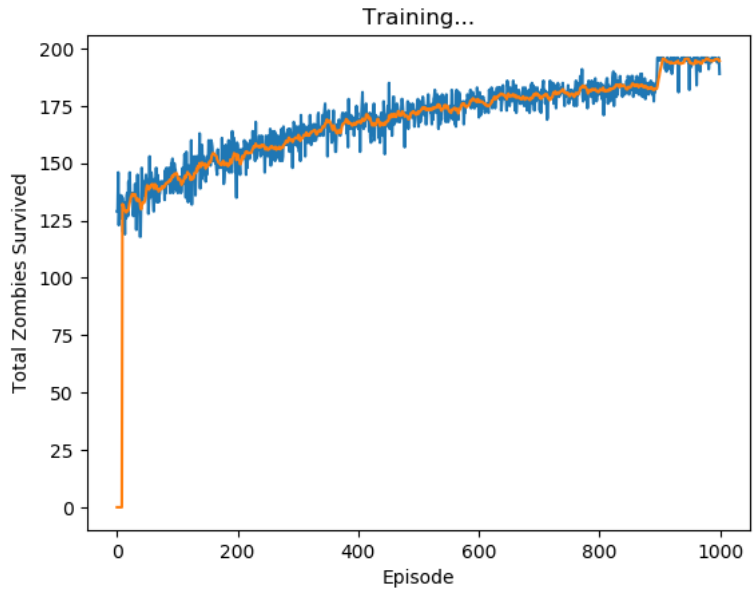

Figure 15 – Total zombies survived vs. the episodes (blue) with its moving average (orange)

Plus, we can tell from Figure 15, 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.

10.4 Light Player test on a 3x5 board

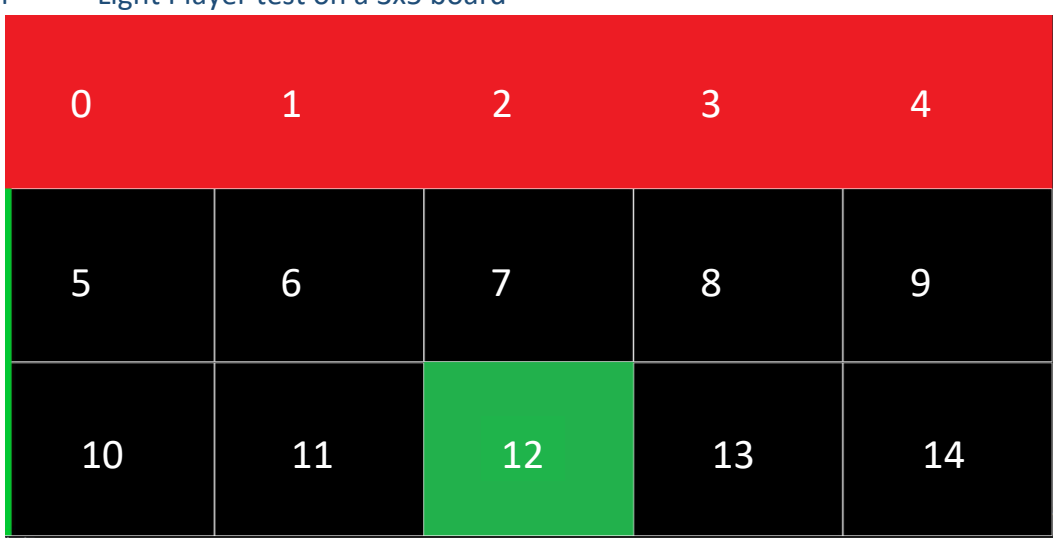

Figure 16 - environment set-up for light Player performance check with optional actions

Figure 16 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:

Table 2 – learning parameters while evaluating the light Player

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

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.

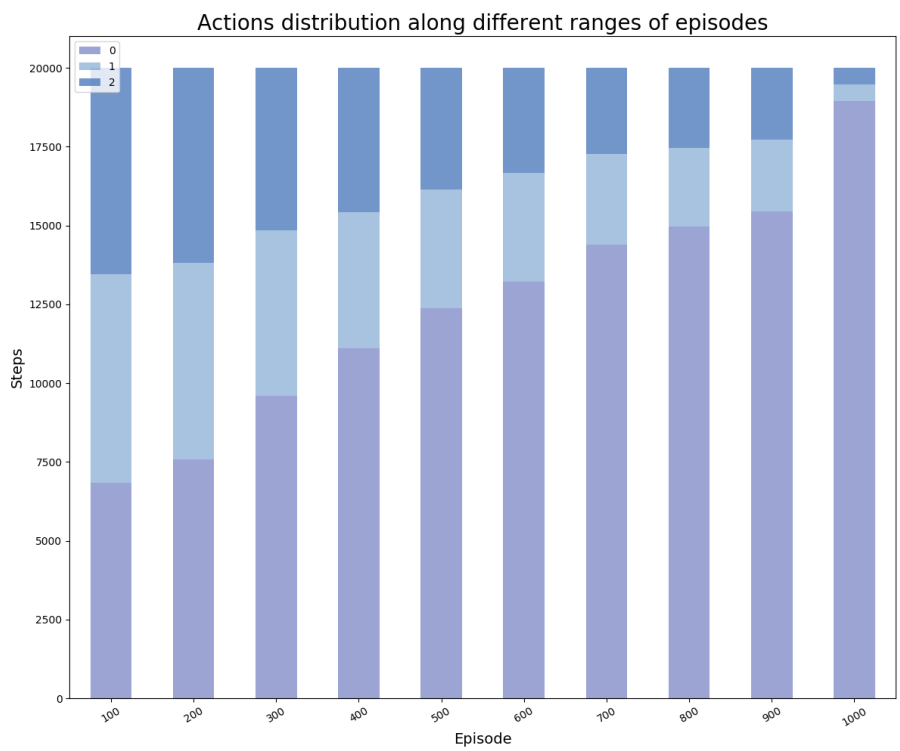

Figure 17 - Light Player actions distribution along different ranges of episodes

In Figure 17 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.

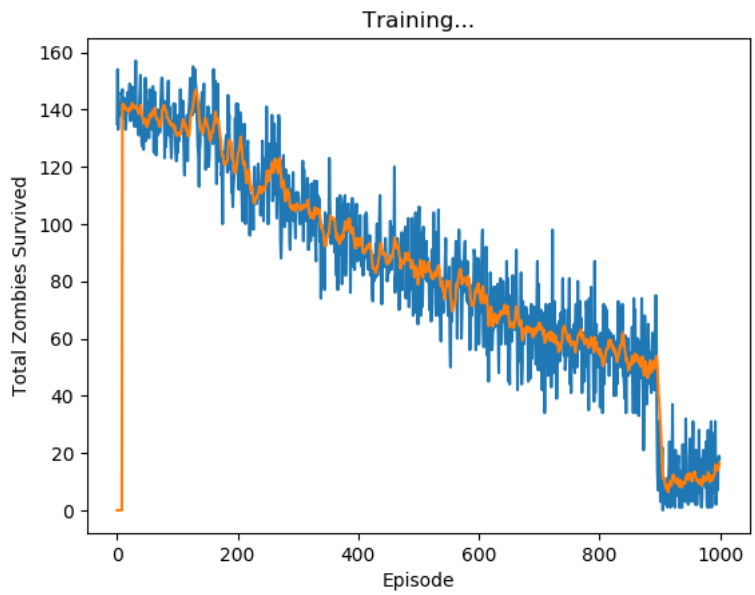

Figure 18 - Total zombies survived vs. the episodes (blue) with its moving average (orange)

In Figure 18 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.

11. Double Deep Q-Network Evaluation

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

11.1 Evaluation Configuration

- 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]

11.2 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 – 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 $2 \times 4 \times 3 \times 2 = 48$.

We do all the above for every combination of the tuning parameters which adds up to $48 \times 9 = 432$ 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:

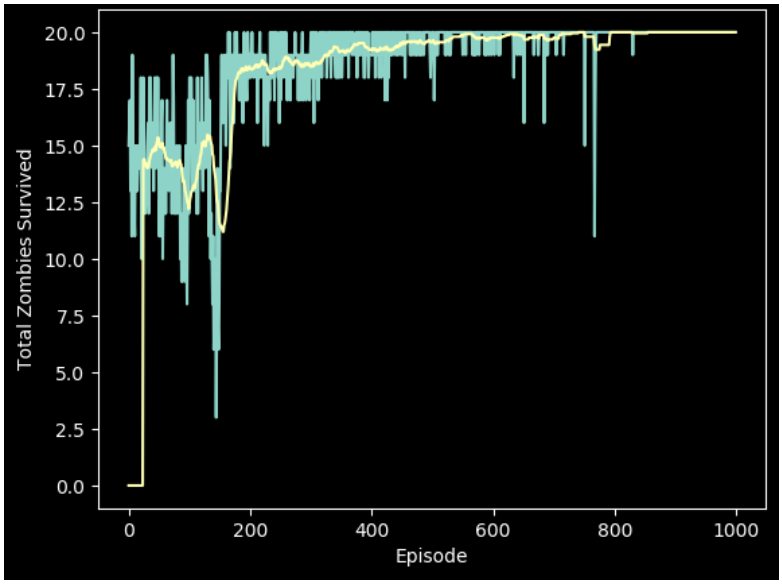

Figure 19 - Example of reward per episode graph, DDQN plays Zombie and Single Action plays Light

Second, we get the average of test rewards (episodes 800 - 1000) from all graphs as in Figure 20:

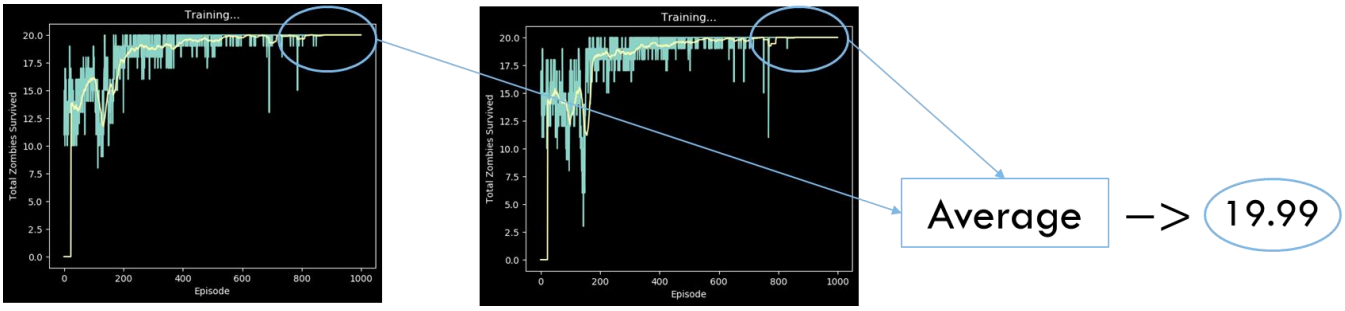

Figure 20 - Scenario Evaluation by Average Test Reward

And finally get an estimate of the performance of the agents in the scenario (the value 19.99 as in Figure 20)

11.3 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:

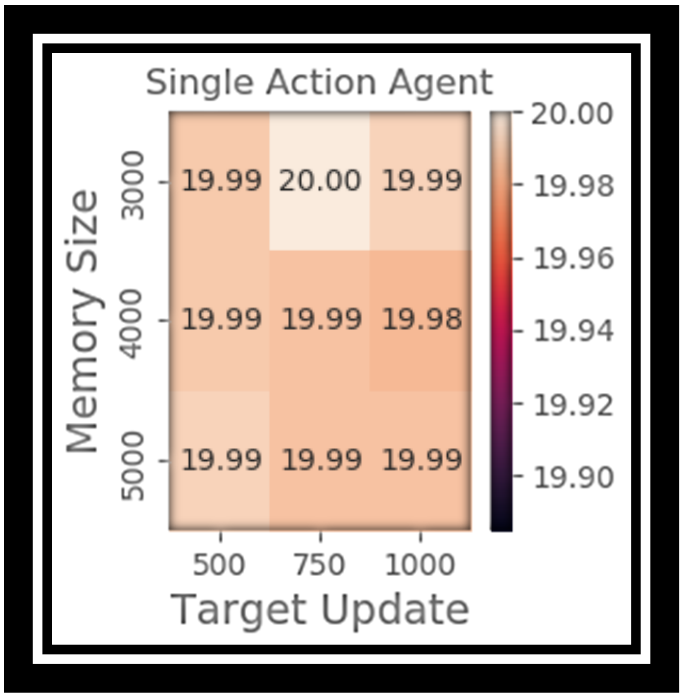

Figure 21 - Heat-Map of the Average Test Rewards of all the scenarios that DDQN Agent plays Zombie and Single Action Agent plays Light

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) 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:

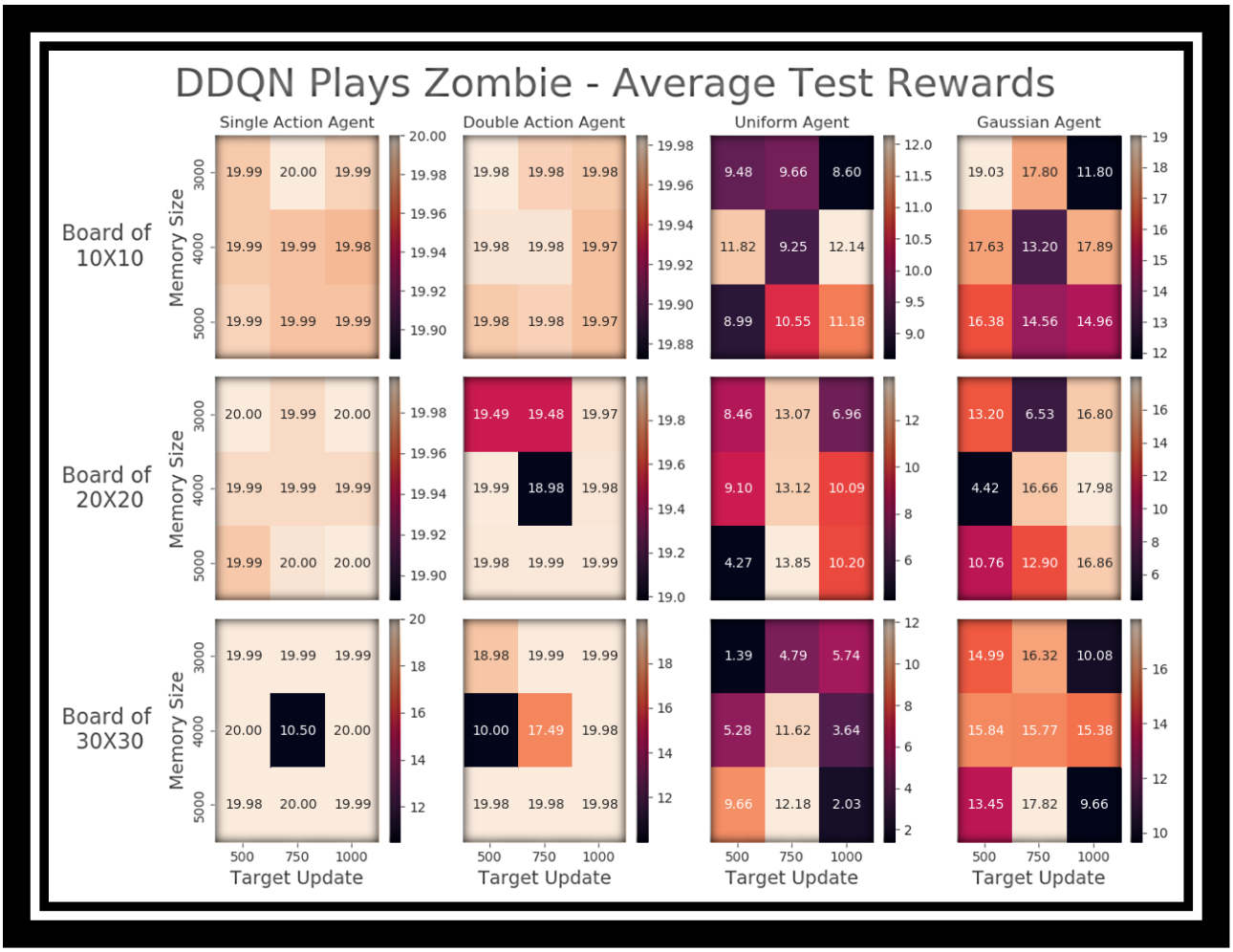

Figure 22 - A summary of all the scenarios that DDQN plays Zombie by Heat-Maps of the Average Test Reward

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, I 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 that 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 19) side-by-side along different board sizes:

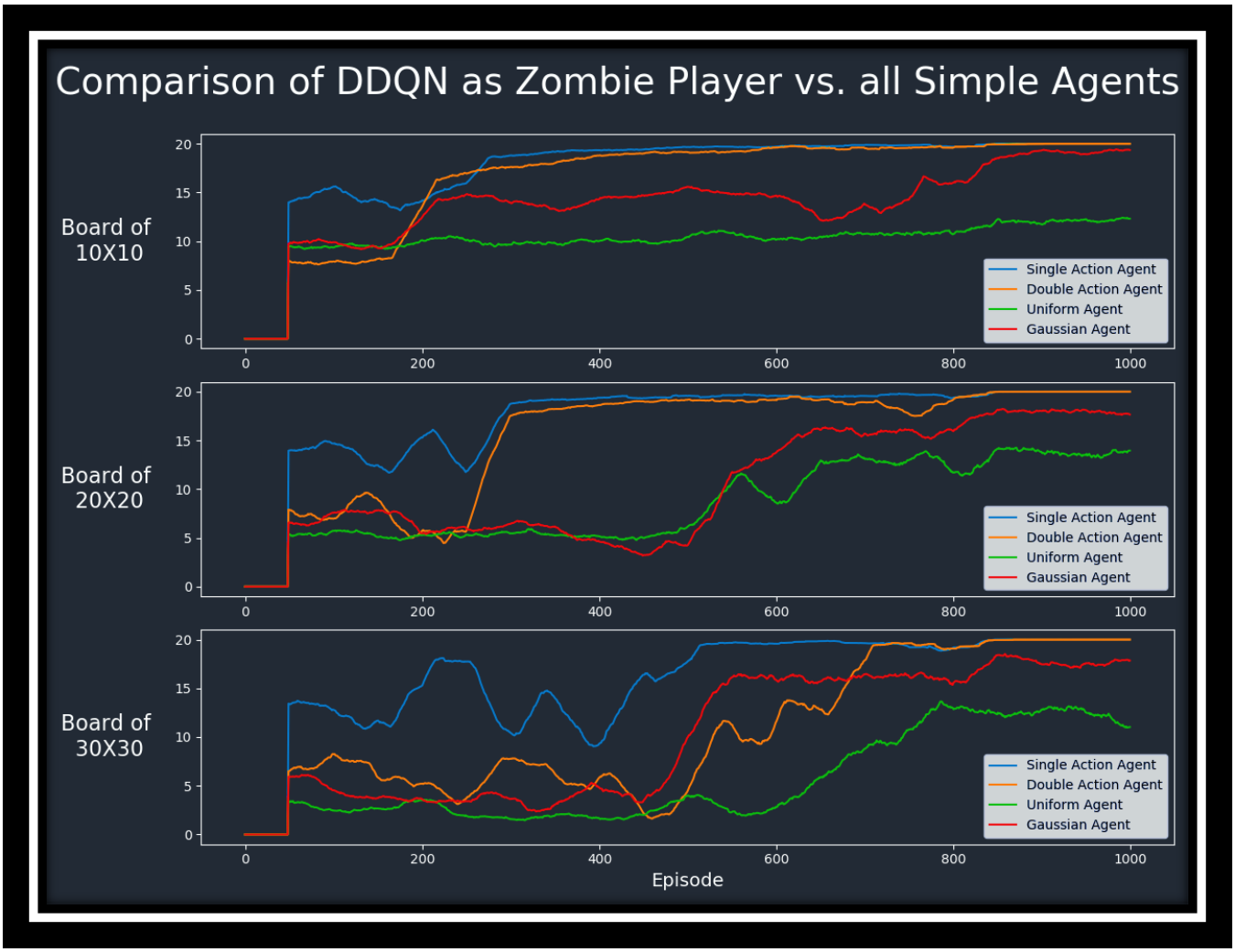

Figure 23 - Comparing the results of the DDQN agent as the Zombie Player, with the best parameters over the different four simple competitors

☒ 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.

11.4 Double Deep Q-Network as Light Player

Same of the last chapter, Before diving in, let's review a simple scenario of DDQN Agent as Light and Single Action Agent as Zombie:

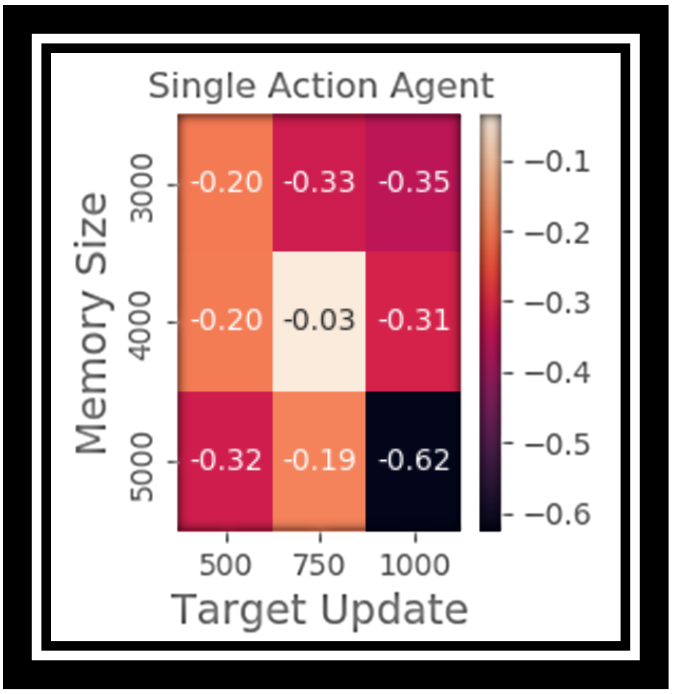

Figure 24 - Heat-Map of the Average Test Rewards of all the scenarios that DDQN Agent plays Light and Single Action Agent plays Zombie

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:

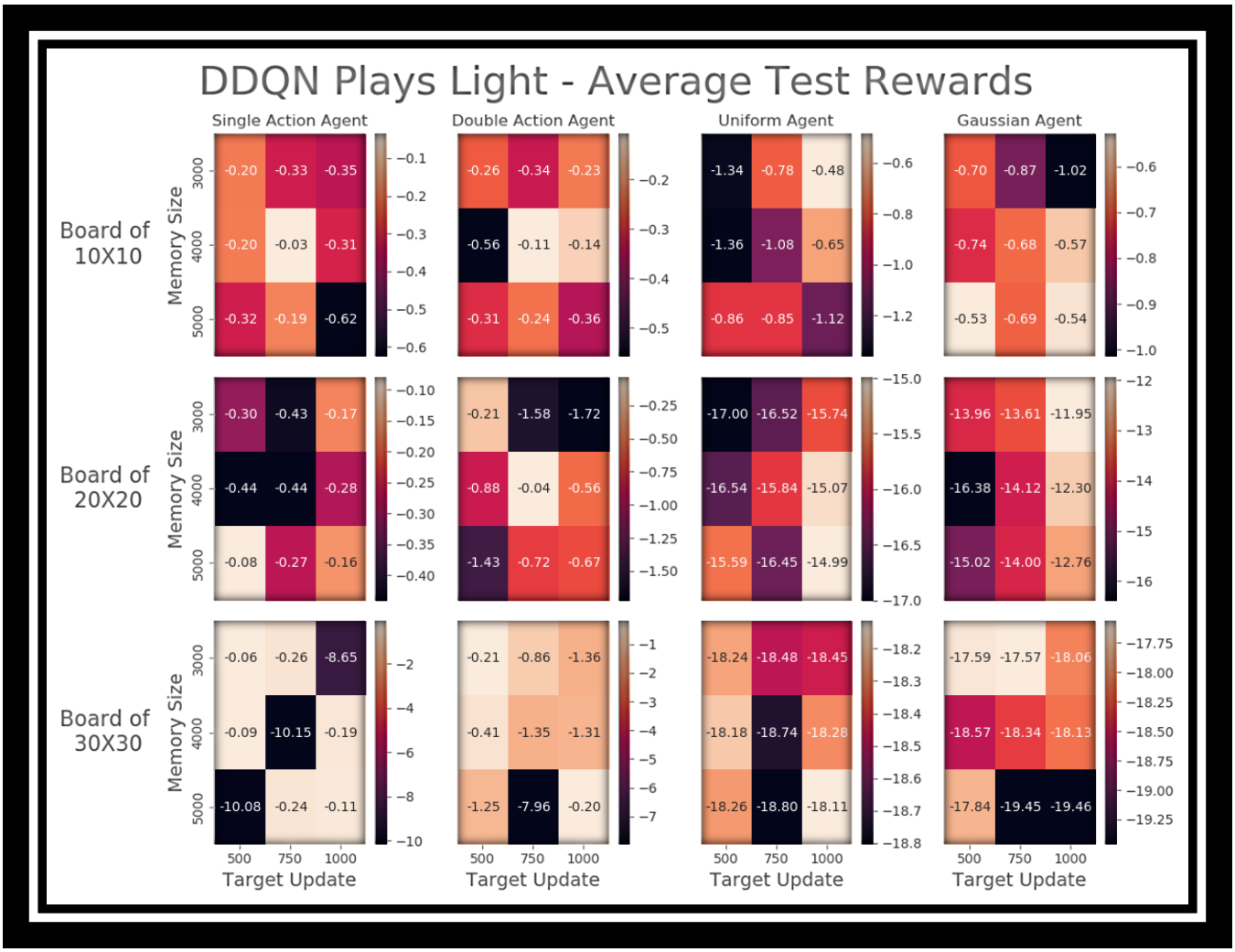

Figure 25 - A summary of all the scenarios that DDQN plays Light by Heat-Maps of the Average Test Reward

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 that 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 19) side-by-side along different board sizes:

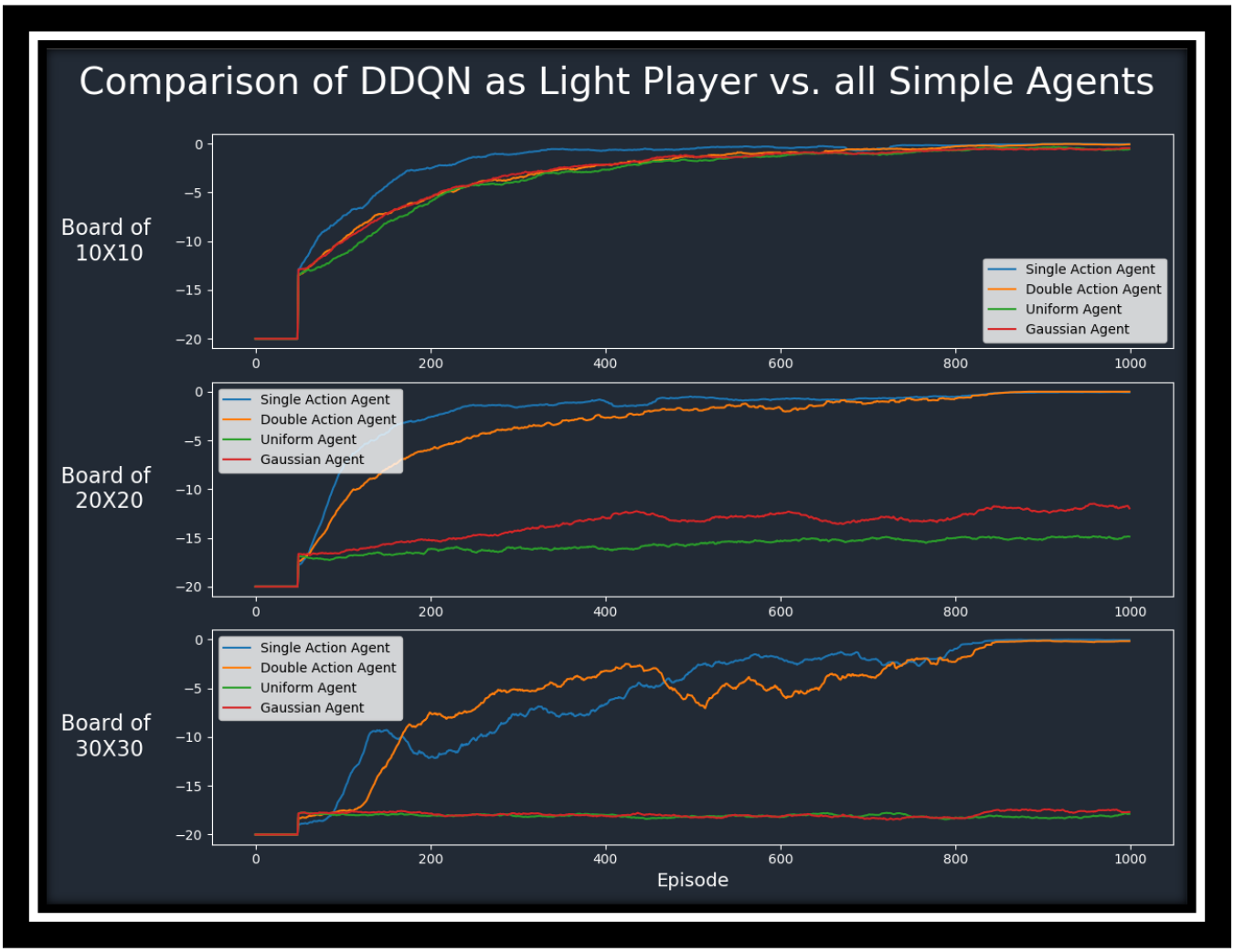

Figure 26 - Comparing the results of the DDQN agent as the Light Player, with the best parameters over the different four simple competitors

☒ 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.

12. Next Steps

- Following the literature review, we will be implementing the algorithms: MCTS, and Alpha Zero in a way that the two players in the game can play against any type of opponent (simple or smart opponent).
- Later, we are going to evaluate the algorithms similarly to what we saw in Chapter 11.

- Finally, after all the algorithms have been evaluated and we gathered their best parameters for each scenario, we are going to compete all learning agents against each other.

13. 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] 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.
[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] 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.
[10] 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.
[11] 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.
[12] Vodopivec, T., Samothrakis,dz S., & Ster, B. (2017). On Monte Carlo tree search and reinforcement learning. Journal of Artificial Intelligence Research, 60, 881-936.
[13] 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.
[14] 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".
[15] 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"
[16] 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

14. 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.

14.1 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 (S, A, P_a, R_a), where:

- S is a finite set of states.
- A is a finite set of actions.
- $P_a(s, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s', due to action a.
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a.

The goal is to learn a policy $\pi^* : S \times A \rightarrow S$ that maximizes the cumulative sum of discounted rewards

$$\pi^* = arg \max_{\pi} \mathbb{E}\left[\sum_k \gamma^k r_k\right]$$

where r_k are the rewards and γ 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.

14.2 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 $(n, S, A_1, \dots, A_n, T, \gamma, R_1, \dots, R_n)$, Where:

- N is the number of the players/agents
- T: $S \times A_1 \times \dots \times A_n \times S \to [0,1]$ is the transition function
- A_i is the action set for the player i
- $\gamma \in [0,1]$ is the discount factor
- $R_i: S \times A_1 \times \dots \times A_n \times S \to \mathbb{R}$ is the reward function for player i

The objective of the n agents is to find a deterministic joint policy (aka. joint strategy aka. strategy profile) $\pi = \{\pi_{i=1\dots n}\}$ (where $\pi: S \to A$ *and* $\pi_i: S \to A_i$) so as to maximize the expected sum of their discounted payoffs. The Q-function, $Q^{\pi}(s, a)$, is the expected sum of discounted payoffs given that the agents play joint action a in state s and follow policy π thereafter. The optimal Q-function $Q^*(s, a)$ is the Q-function for (each) optimal policy π^*. So, Q^* captures the game structure. The agents generally do not know Q^* 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.

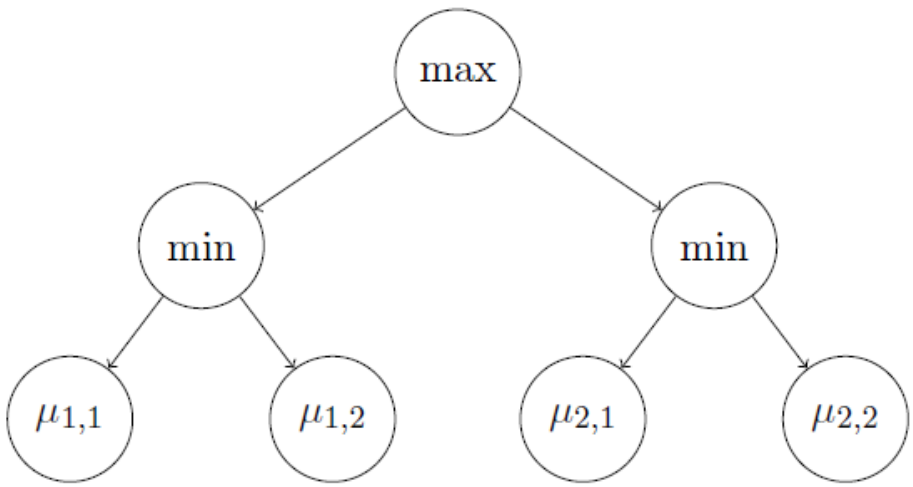

Figure 27 – Game tree when there are two actions by player $(K = K_1 = K_2 = 2)$

We consider a two-player two-round zero-sum game, in which player A has K available actions. For each of these actions, indexed by i, player B can then choose among K_i possible actions, indexed by j. $For\ i \in \{1, \dots, K\}\ and\ j \in \{1, \dots, K_i\}$, when player A chooses action i and then player B chooses action j, the probability that player A wins is $\mu_{i,j}$. We investigate the situation (see Figure 27 for an example) from the perspective of Player A, who wants to identify a maximin action

$$\boxed{i^* \in \operatorname*{argmax}_{i \in \{1,\dots,K\}} \min_{j \in \{1,\dots,K_i\}} \mu_{i,j}}$$

Assuming that Player B is strategic and picks, whatever A's action i, the action j minimizing $\mu_{i,j}$, this is the best choice for A.

14.3 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 SG with n players, a Nash Equilibrium is a tuple of strategies $(\pi_*^1, \dots, \pi_*^n)$ such that for all $s \in S$ and $i = 1, \dots, n$,

$$v^i(s, \pi_*^1, \dots, \pi_*^n) \geq v^i\left(s, \pi_*^1, \dots, \pi_*^{i-1}, \pi^i, \pi_*^{i+1}, \dots, \pi_*^n\right) for\ all\ \pi^i \in \Pi^i$$

Where, Π^i is the set of strategies available to agent i, And,

$$v^i(s, \pi_*^1, \dots, \pi_*^n) = \sum_{t=0}^{\infty} \gamma^t E(r_t^1 | \pi^1, \pi^2, \dots, \pi^n, s_0 = s).$$

Is the discounted sum of rewards, with discount factor $\gamma \in [0,1)$.

A Nash equilibrium is strict if the inequality above is strict. An optimal Nash equilibrium π^* 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.

14.4 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 [7], 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 $0 < \beta < \alpha < 1$).