Playing Halite IV with Deep Reinforcement Learning

Kim Haapamäki

Jesper Laurell

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6127
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset
Lund 2021

Contents

Abstract

Playing games with reinforcement learning has for years been a target for research and has seen incredible breakthroughs in recent years. Reinforcement learning is a type of machine learning, which can be combined with the concept of deep learning, resulting in what is called deep reinforcement learning. The promise of deep reinforcement learning attracts businesses that aim to get an edge over traditional algorithmic methods. Our work focused on exploring the aspects of deep reinforcement learning with a DQN implementation and the game Halite IV as the environment. We created DQN agents capable of outperforming competitive solutions and tested and evaluated techniques for enhancing the DQN solution. The most insightful results include: individual decision making for a team based environment can simplify the DQN setup drastically, reward function engineering for RL is critical and a sparse reward is not practical for long time frames, self play is advantageous compared to a static opponent and low rates of exploration is beneficial in environments with built in randomness.

Acknowledgements

We would like to thank Johan Eilert for great ideas and in depth discussions, and Sinch for hosting this project making it possible for us to deep dive in reinforcement learning.

We would also like to show our gratitude to Johan Grönqvist who has not only helped us with great discussions and kept us focused on the end goal but also his open mindedness around reinforcement learning research.

Finally we would like to thank Anders Ranzter for his interest in this project and for accepting as examiner of the thesis.

Abbreviations

Adam - Adaptive movement estimation
DL - Deep learning
DQN - Deep Q-network
DRL - Deep reinforcement learning
MARL - Multi agent reinforcement learning
MSE - Mean square error
NN - Neural network
ReLU - Rectified linear unit
RL - Reinforcement learning
TD - Temporal difference

1

Introduction

1.1 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning in which an agent interacts with an environment through an action and is provided with a next state and a reward in return. See figure 1.1. The goal of the training is to learn which actions to take to maximize the reward. This is done by updating a decision policy depending on the outcome of the performed actions, reinforcing successful and discouraging unsuccessful behaviour according to the rewards. RL, and more specifically, deep RL (DRL) — in which the concept of RL is combined with the use of neural networks — has seen a tremendous rise in recent years. DRL has made enormous progress many areas, including playing board games such as the challenging game of Go [Silver et al., 2016][Silver et al., 2017], robotics [Kober et al., 2013], computer vision [Bernstein and Burnaev, 2018] and autonomous driving [Shalev-Shwartz et al., 2016]. RL is not supervised learning, nor is it considered unsupervised learning, but is instead a third fundamental machine learning principle [Luong et al., 2018].

1.2 Halite IV

Halite IV is a computer game designed to be played by bots, initially released as part of a competition hosted by Two Sigma on Kaggle. It is a turn based game, but all players make their moves simultaneously each turn. The goal of the game is to collect and return halite — the only resource in the game — which is initially randomly scattered around a grid map. The game is played by controlling units of two different types; ships and shipyards. Each player starts with a single ship and 5000 halite. Ship units can perform four key tasks: move (4 different directions), farm halite picking it up as cargo (by standing still when occupying a cell containing halite), deposit its currently held cargo (by moving onto a friendly shipyard) and finally convert into shipyards (for a cost of 500 halite). Halite that a ship farms will be contained on that ship and is referred to as *cargo* in this document. Shipyards on

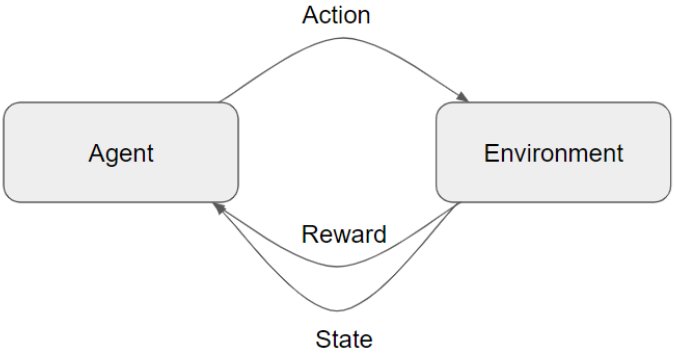


Figure 1.1: The fundamental concept of RL

the other hand cannot move but allows for spawning of new ships for the cost of 500 halite. Units that occupy the same cell collide. If ships collide, the ship with the least amount of cargo destroys the other one (or other ones) and steals its cargo. If a ship collides with an enemy shipyard, both units are destroyed. The winner of the game is the last player remaining (a player is eliminated if it has no ships left and cannot build another) or the player with the most halite after 400 frames of play. The original map of the game is a 21x21 grid and is both rotationally invariant in increments of 90 degrees (all directions in the game behave the same) and translationally invariant (a ship leaving the map will return on the opposite side). Halite IV is *fully observable*, meaning all information about the game is available to all players at every time [Kaggle, 2020]. A screenshot from the game is seen below in 1.2. The information available to the agents is however provided via numbers and matrices.

As for the complexity of Halite IV there is a lot to consider. However, determining the complexity of a game is not trivial and is not the subject for this thesis. In depth analysis of exactly *how* complex the game of Halite is has as such been omitted from this report. At a glance however, one can realize that a grid map of 21x21 and the number of controllable units in a given time frame varying from 0 to around 50 — which each can perform 2 and 6 different actions respectively — and a game length of 400 frames is not a simpler game than other games used for RL research (such as Atari 2600 games, Chess and Go).

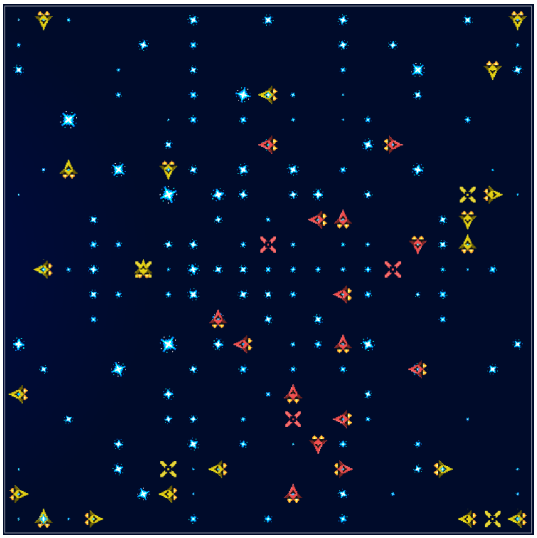


Figure 1.2: Snapshot of a Halite IV game with two players.

1.3 Goal

This project has been done for Sinch, a communications company with interest in machine learning. Sinch's purpose in hosting this master thesis was to learn more about reinforcement learning through a safe environment, which was chosen to be Halite IV. Halite IV is implemented with Python which is used for the reinforcement learning, this allowed the project to be focused on reinforcement learning from the start.

The initial goal was to deepen the understanding of the advantages, limits and properties of reinforcement learning based agents.

However, during the process of our work this vague formulation went through concretizations resulting in both a more limited and more direct goal. Instead the final main goal was formulated as follows:

Evaluate the impact of the RL related techniques and parameters shown in table 1.1 with a DQN approach.

This report serves to describe these different aspects of the RL implementation, with focus on explaining strengths and weaknesses of the additional techniques that were added on top of a baseline agent. By using Halite IV as a sufficiently complex environment, we have investigated a number of additions/adjustments that can be

of interest to explore when using RL as a machine learning tool. Another aim of this thesis is to serve as a basic guide with tips and guidelines for building a DRL agent. The work was performed as a case study around DRL.

The techniques/parameters below in table 1.1 lay the experimental foundation for our thesis. Our goal has been to explore and evaluate the impact each one has when adjusting and improving a DRL agent.

Techniques and parameters
NN layout
Changing reward function
Train on enemy data
Discount factor, γ
Compacting input
Exploration vs. exploitation, ε
Near-sightedness
Choice of opponent during training

Table 1.1: Techniques and parameters that have been investigated

These choices were carefully made and prioritized and is a result of discussions with our supervisors both at the university and at Sinch. The concrete meaning of each and their implementation will be described in depth in the methodology section.

1.4 Scope

Due to limited time and resources of the project a realistic scope for the project had to be set. First and foremost the project is limited to using Halite IV as a medium of evaluation, similar to how previous RL evaluation methods have used the Atari2600 benchmark and other video games for research (see [Mnih et al., 2013] for example). Another limitation that was decided on was to only play against **one opponent** at a time instead of three as in the original version of the game, this was to limit the noise in the results. Also the number of RL techniques had to be limited to the most promising and interesting ones. Analyzing several different aspects means we are taking a wide approach to investigating the huge field of RL.

2

Theory

2.1 Reinforcement Learning

Reinforcement learning is an area of machine learning in which an agent interacts with an environment by performing actions a_t given a state s_t provided by the environment. In response to the action taken the environment will return a new state s_{t+1} and a reward r_t. The goal of the agent is to maximize the reward signal. (The reward signal is usually not only dependent on the immediate reward given by r_t. This will be explained further in the section 2.6 below). This is done by acting according to a policy π, a function that returns an action a_t given an input state s_t,

$$\pi(s) = a_t \tag{2.1}$$

and updating the policy depending on the feedback from the reward signal. By updating its policy according on the outcome of the actions taken, reinforcing successful behaviour and discouraging unsuccessful, the agent can improve over time. There exist several different approaches to implementing an RL agent, one of which, called Deep Q-network (DQN), will be the focus in this report. A key difference from supervised learning is the fact that the agent itself generates the data which is used for training instead of being fed correct answers. For further reading, see [Shao et al., 2019] and [Si et al., 2009].

2.2 State Space

State space refers to the space of all the possible configurations of a given environment. It is easily explained by an example: In 2D pole-balancing task as shown in 2.1 the state can be represented by a 4-tuple $s_t = [x_c, v_c, \alpha_c, \omega_c]$. x_c denotes the x-coordinate of the cart, v_c is the velocity of the cart along the track, α_c is the angle created by the pole and y-axis and ω_c is the angular velocity of the pole around the connection between cart and pole. All the different combinations of these parameters that can arise make up the state space of the system [Nguyen et al., 2018]. The state space of a game is important in determining its complexity which in turn plays

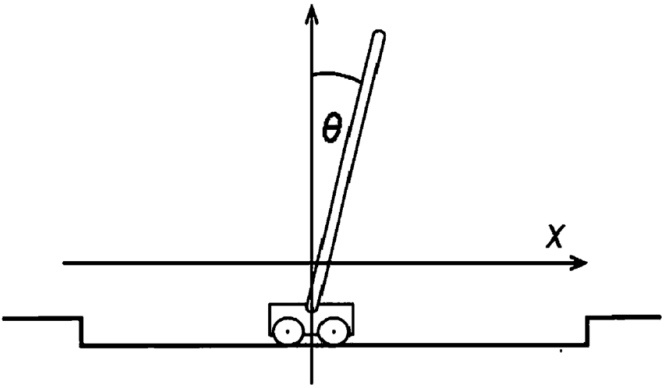


Figure 2.1: Cart pole problem

an important role in how difficult it is to solve the game or create a skilled agent. Traditional games and video games often have state spaces that are easy to describe making these environments ideal for experiments in machine learning. In Halite IV, the state space consists of all possible game board configurations (including the position of all ships and shipyards, and available halite in each grid cell), the amount of deposited halite of each player, and the turn number. For further reading on state space, see [Sutton and Barto, 1998, p.5-7].

2.3 Action Space

Action space in RL refers to the set of possible actions that are available at a certain state in the environment. The action space is different for each problem and can be adjusted according to the solution that one is trying to accomplish. As an example from OpenAI gym LunarLander-v2 [Brockman et al., 2016], the environment consists of space ship that is to be steered to land on a surface. The ship is in each frame able to take one of the four actions, up, left, right or no action (down). An image from this environment can be seen in figure 2.2. Here the action space is easy to determine and the output of the agent will be these 4 actions. However in an environment such as Halite IV, there are a varying number of ships and shipyards in each frame of the game. Each of the units have their own actions that the agent needs to decide in each frame. This adds complexity to the solution and effects the convergence of the resulting policy. For further reading [Sutton and Barto, 1998, p.5-15].

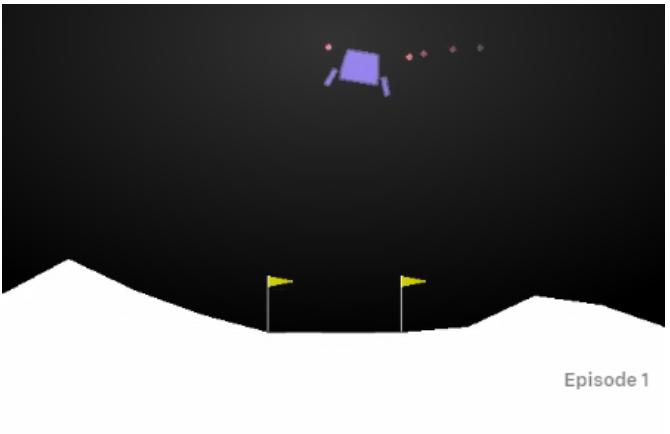


Figure 2.2: Environment LunarLander-v2 in OpenAIgym

2.4 Agent perspective, Multi Agent Reinforcement Learning

Multi Agent Reinforcement Learning (MARL) is a case of reinforcement learning in which there is more than one agent acting in an environment simultaneously. This approach has several advantages but also limitations. The fact that all agents can explore and get feedback from the environment in parallel is a major advantage. Also, if one agent fails there is no need for resetting since the simulation can continue with the remaining agents. It also allows for easy addition and removal of agents into the environment should it be wanted. Furthermore, the action space of an individual agent is often far smaller than the combined joint action space of multiple agents. This means that representing every individual agent by a neural network is a lot easier than representing an unknown number of agents by a single NN [Busoniu et al., 2010].

On the other hand, using an MARL approach also significant drawbacks. With multiple agents acting in parallel, each agent not only have to consider the environment itself but have to consider the other agents as well and have to learn to adapt to their decisions. Another major issue is the problem of goal formulation. In cooperative games the overarching goal is for the agents to collaborate in an effective way but with multiple agents in play each agent is given feedback based on its own decisions. This can give rise to egoistic behavior that benefits only the individual but cripples the collective if the reward function is not carefully designed. For further reading, see [Busoniu et al., 2010] .

2.5 Deep Learning

Deep learning in AI is the concept of using a deep neural network (also called a deep graph) with several layers. It is a subset of machine learning and has great flexibility and a vast area of use cases. This field has seen tremendous rise in recent years and found ways to be useful in computer vision, natural language processing, video games, search engines, finance and more. The recent increase in collection of data and computational power has enabled deep learning to be much more effective. One of the main strengths of deep learning can be described as follows: "Deep learning enables the computer to build complex concepts out of simpler concepts" [Goodfellow et al., 2016, p.5]. This principle allows for computers to generalize input and make decisions based on those generalizations rather than relying on traditional learning and algorithms. See [Goodfellow et al., 2016, p.8- 9] for further reading.

2.6 Temporal Difference and Tabular Q-Learning

Temporal difference learning is a form of reinforcement learning and Q-learning is an implementation of this concept. Temporal difference refers to the difference between the agent's estimation of the value of next state and the value of the current state. An action given a state is referred to as a *state-action pair*. One of the most straight-forward ways to set up an RL agent is with the use of tabular Q-learning. The Q in Q-learning describes the value of a state-action pair (in contrast to the value of a *state*). The tabular Q-learning method utilizes a matrix (called Q-matrix) to keep track of an estimation of how good all possible actions (denoted a) are, given a previously visited state (denoted s). A state that has yet not been visited will yield a value of 0 for standard methods. The idea is to give the agent positive rewards when it succeeds with the given task and negative (or zero) when it fails.

The update rule for Q-learning is presented below (2.2)

$$Q^{new}(s_t, a_t) \longleftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.2}$$

Here Q denotes the matrix value, α the *learning rate*, r_t the *reward* at the current time frame and γ the *discount factor* determining the importance of future rewards compared to the current. Note that the new Q (on the left hand side) is based on its previous value for the given state-action pair with the second term added. The temporal difference is the term inside the square brackets, which is the difference between the current estimate and the updated estimate. By updating the values in the Q-matrix iteratively depending on the outcome of the obtained reward — which is a sum of the current reward r_t and the discounted maximum estimated reward in the next time frame $\gamma \cdot \max_a Q(s_{t+1}, a)$ — the Q-matrix converges towards an optimal mapping from state to action for the problem. A major issue with tabular Q-learning however is that many tasks require a large state space in order to be represented

realistically. Every time an agent using this method encounters an unseen state it has no previous information about it and thus can not make an educated prediction. Recognizing similar states and generalizing is not possible for tabular methods and as such they are only suitable for simple problems with small state spaces. This is where deep learning with neural networks enter the picture. [Sutton and Barto, 1998][p. 157] explains Temporal Difference learning more in depth.

2.7 DQN — Deep Q-network

Similar to tabular Q-learning, the DQN method approaches RL in the same way. By allowing the agent to visit states it tries to update a Q-function that approximates the optimal policy. However, instead of using a table to map states to actions, a neural net is used as a function approximator, see [Hornik et al., 1989]. Essentially the state space for the problem at hand is recognized to be too large to be contained in a table and an approximation of the state suffices. DQN provides a solution for the large state space issue that is apparent in tabular Q-learning by using a NN as a function, mapping a state-action pair to a value, instead of storing each and every state-action pair and their estimated values separately in a table. By having a fixed number of network coefficients, the NN tries to form functions to give correct estimations of states, these functions are able to generalize to similar states. Using this change to the tabular Q-learning and utilizing the same update function in order to connect the explored states to targets in a matrix, these states and targets (stored in the matrix) are then used for updating the weights of the NN by a key function commonly called *fit*. The *fit* function adjusts the weights in the NN, changing the function currently contained in the NN to fit the given inputs to the wanted outputs (called *targets*). The *fit* function can be performed for a large number of inputs and targets. Since the inputs for a given state is already known, the task is reduced to building and updating the targets which is done with the help of rewards. For further reading, see [Nguyen et al., 2018].

2.8 Reward Function

r_t in equation 2.2 is defined by the reward function, which returns a scalar number, $r_t \in \mathbb{R}$, as the agent performs actions in the environment. The rewards can be either sparse or shaped. Sparse means that the only reward given to the agent is if it won or reached its goal at the very end of the episode. Rewards that are introduced during the episode are called shaped. The purpose of the reward function is well summarized in the following quote:

> "A reward signal defines the goal in a reinforcement learning problem. On each time step, the environment sends to the reinforcement learning

> agent a single number, a reward. The agent's sole objective is to maximize the total reward it receives over the long run. The reward signal thus defines what are the good and bad events for the agent." - [Sutton and Barto, 1998, p.7]

As an example an agent might be used to escape a maze and it gains a large reward when exiting the maze, in this scenario it could also be given a small negative reward for each step taken prior to escaping the maze thus giving incentive to escape as fast as possible to maximize the total reward. It is critical that the rewards are set up to indicate the end goal of the agent and not allow it to find ways to maximize the reward without accomplishing the intended end goal. For further reading about reward functions in RL, see [Sutton and Barto, 1998, p.7-9].

2.9 Discount Factor, γ

The discount factor γ that is present in many RL methods, including Q-learning, determines how large of an impact the estimated future state should have compared to the immediate reward. If $\gamma = 1$ it means there is no discount and all future rewards are valued as equal to the current reward. Setting $\gamma = 0$ results in an agent only maximizing the immediate reward and potential future rewards are not considered in the current time frame. Selecting to prioritize the short term reward can result in missing out on a larger total long term reward. A large γ, making the agent more farsighted, means future rewards are more heavily considered but projecting rewards far into the future in an environment that is hard to predict accurately can mean some of the rewards far into the future are actually not obtainable. A γ between 0.9 and 1 is standard practice in RL. For futher reading, see [Sutton and Barto, 1998][p. 60].

2.10 Exploration vs Exploitation, ε

A key dilemma in RL is exploring vs exploiting, i.e. when to choose the optimal action according to the current policy and when to ignore the policy and select another action (for example chosen from a uniformly random distribution over all valid actions). This can in theory help the agent to explore random actions yielding a larger long term reward that would have never been discovered without the random exploration. The most straight-forward way of implementing exploration is with the ε-greedy method. This method works in the following way: When an action (a) is to be selected, choose another random (uniformly distributed) valid action with a probability of ε and choose the action giving the most total reward according to the current policy otherwise (probability of $1-\varepsilon$). See equation 2.3.

$$a = \begin{cases} \text{random} & \text{with probability } \varepsilon \\ \max_a Q(s_t, a) & \text{otherwise} \end{cases} \tag{2.3}$$

A higher value of ε means a higher rate of exploration. The method is greedy because if the random number generator selects a number smaller than ε the agent will take the action that is expected to yield the most total reward. Exploration is essential for solving some RL problems since it enables the agent to discover new strategies and improve performance in the long run. This is due to the fact that if an improvement to the policy can be discovered through exploration, it can later be exploited for all future times where the same (or a similar in the case of DRL) situation arises. For further reading, see [Sutton and Barto, 1998, p. 32-34].

2.11 Action Replay

Using an NN to approximate mapping state-action pairs to targets from one episode is often unstable due to the high variance from episode to episode, these changes between episodes can change the behavior of the agent in a volatile and unstable manner. Action replay is a memory of the previously played games and is used for enabling training on previous data and not only the most recent. This is useful in order to mitigate overtraining on recent data by having the NN fitted on prior and recent data. As the NN is fitted over larger sample size the decisions it adopts will be based on strategies that works consistently. As such the data which the agent uses for training is only partly changed each time the policy is updated which gives a more stable update of the decision policy. For further reading, see [Watkins and Dayan, 1992].

2.12 Off-Policy Learning

Off-policy learning means that the updated policy is different from the current behaviour policy. The perks that come from exploration and action replay explained above can only be utilized through off-policy learning since an agent with exploration will always have a different policy compared to its full-exploit counterpart and action replay will contain data generated from a previous policy. DQN is an off-policy method. The DQN agent can be presented with a state from the action replay and will evaluate its expected total reward if it was allowed to follow the current policy from that state (without exploration). For further reading regarding off-policy learning, see [Geist and Scherrer, 2013].

2.13 Elo-Rating

In order to rank (or sort) players according to their skill level in a game such as Chess, Counter Strike or Halite IV an algorithm has to be chosen. Arpad E. Elo invented the currently widely used Elo rating system. Originally it was created for rating Chess players, but is currently also adopted by Go, competitive E-sports and

many other competitive games. The Elo system uses the players' current rating to predict the probability of each player winning and uses this information to update the players' rank after the game. The Elo update rule follows the equations 2.4 and 2.5,

$$W_e = \frac{1}{(1+10^{\frac{(R_{At}-R_{Bt})}{m}})} \tag{2.4}$$

$$R_{A(t+1)} = R_{At} + K(W - W_e), \tag{2.5}$$

where W_e denotes the probability of player A winning, R_{At} and R_{Bt} the current ranks of player A and B respectively, $R_{A(t+1)}$ the new rank of player A, W the actual outcome of the game being equal to either 1 (=player A wins) or 0 (=player A loses), and finally m and K are manually set constants. The corresponding change in rank update will happen in the same manner for player B, which will result in a rank update in the opposite direction. The essence of this rating system is to take into account the relative strength of the players before the the match and adjust the updated ratings depending on the actual outcome compared to the predicted outcome. A win against a higher ranked opponent yields a higher increase in rank than winning against a player of lower rank (and conversely a loss against a lower ranked player means losing a lot of Elo). With this system, a relative rating order can be established without the need for all players in the pool to face each other. This efficiently creates a ladder ranking for all players in a pool. For further reading, see [Elo, 1978].

2.14 Behavioral Cloning

One way of constructing an agent capable of performing well in a given environment is to look at other agents that are already performing well and imitate their behaviour. This approach, theoretically, enables a NN based agent to learn from hard-coded strategies and incorporate them in its own toolbox. By letting the NN agent review a game played by such an expert and predict which action should be taken in a given state, rewarding it for predicting correctly, the problem essentially turns into a case of supervised learning where the correct labels are given by the expert. However, unless the expert is playing perfectly (and is proven to do so) the moves that are rewarded will obviously not necessarily lead to optimal play for the NN agent itself. Refer to [Torabi et al., 2018] for more in depth reading on this subject.

2.15 Tensorflow and Keras

Tensorflow is an open source machine learning framework. It is widely used and developed continously. Keras is a Python deep learning API built to enable easier

implementations for multiple machine learning libraries like Tensorflow. "Being able to go from idea to result as fast as possible is key to doing good research" - [Chollet et al., 2015].

3

Methodology

3.1 Keras

This project is built with the help of the Keras API for Python, and with Tensorflow 2.0 as underlying machine learning library. Having to implement an efficient deep learning framework from scratch would require a lot of additional time and work.

3.2 Construction of Baseline

This project can roughly be divided into the following two parts:

1. Constructing a baseline model
2. Investigating effects of different techniques/parameters and evaluate their impact on performance.

Since Halite IV requires a bit of basic decision making in order to be played, the first weeks of the project were dedicated to creating a somewhat well performing DQN-agent. A lot of testing and tweaking was done in this stage which ultimately resulted in the model which will be referred to as the *baseline*. The second part revolved around trying to improve upon and experiment with the *baseline* by introducing various adjustments. Due to limited time, and to enable us to focus on RL in general, the NN agents only control the ships (and not the shipyards). Another simplification that was made was to exclude the convert action from the NN output and instead converting ships to shipyards according to a fix algorithm.

Only the ships are controlled by a NN because this proved to be sufficiently difficult to master and still allowed for agents to improve drastically with training. Furthermore, controlling the shipyards with a NN would require a complete new set of models, effectively doubling the number of models and drastically increasing the training times. The convert action was excluded from the output of the NN because this action should be taken much less frequently than the other 5 movement

actions. With a randomly initiated NN however, all actions occur equally frequently and ends up converting all of the ships to shipyards costing 500 halite for each convert. This is problematic during the early stages of the training and requires another reward function taking into account the problems of converting. F

Agent Perspective

One of the very first design choices that were made was to let the NN make decisions from the perspective of a ship as opposed to an overviewing perspective. This idea is central to almost all of the implementations and is a very important concept as it has great implications for the training process and problem formulation. All ships during a game make decisions according to the same NN, which is only updated in between games. This means that the fleet of ships can be seen as several clones acting according to the same policy but depending on their own situation on the board. Taking this approach turns the problem into more of a Multi Agent Reinforcement Learning (MARL) problem which has advantages but also limitations. For playing Halite in an optimal way is is required that the ships collaborate in an effective way, maximizing the collectively gathered halite even while acting according to the individual policy. See section 2.4. For further reading on MARL, see [Busoniu et al., 2010].

By utilizing the fact that ships in world of Halite that leave the map will return on the opposite side, the learning data which the model was trained on could be simplified. This was done by reorganizing the raw state matrix of the game into a corresponding one with currently controlled ship in the center. See 3.1. When letting the ships make decisions on an individual level, it makes sense to let the ship get input information in this format. This design allows for focusing only on the states of the state space that have the ship of interest in the center. Also, no data from the state is lost and the most crucial information will always be close to the ship, i.e. the center of the board after centering.

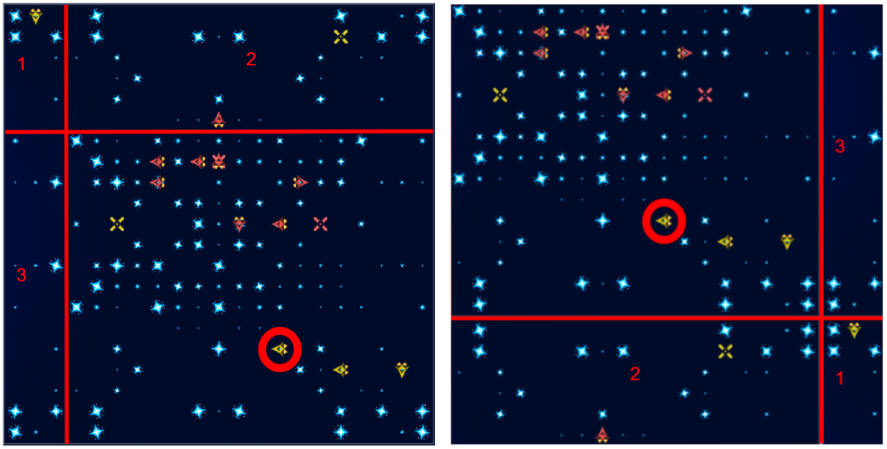


Figure 3.1: Visualization of the center cell preprocessing of a state. It reformats the input from a fixed overview of the map into a map where the current ship is in the center.

State Space Representation

The input for the *baseline* model was set to be 8 matrices of size 21x21. Since there are a few different classes of data a cell can contain, these classes were isolated in their own matrices. For example, the amount of halite on the board is described in the first input matrix and each element's value that ranges from 0 to 500 is normalized by dividing by 500. The matrices describing whether a cell contains a certain unit or not are binary and each cell is described by a 0 or a 1. The time frame and cargo are actually just single digits but contain vital information about the state. In order to pass this information to the model a 21x21 matrix is fully populated with this number. A list with accompanying explanation of all baseline matrices are found in table 3.1. Unless explicitly specified, these are the inputs used for other models than the *baseline* as well

Baseline NN Design

The *baseline* used a NN design shown in figure 3.2. It consists of a simple sequential model using a convolutional layer as the first layer. This enabled some feature extraction from the input images (matrices) of the game state. The use of an initial convolutional layer gives the possibility of learning features with spatial relationships much faster. The convolutional layer is followed by 3 dense layers with 100 units each and a ReLU activation function. The data in the third dense layer is then fed into a final dense layer to give the proper expected value for each action from that state. The final dense layer has the same width as the action space of the model, which for our purposes was 5 (4 directions of movement and standing still).

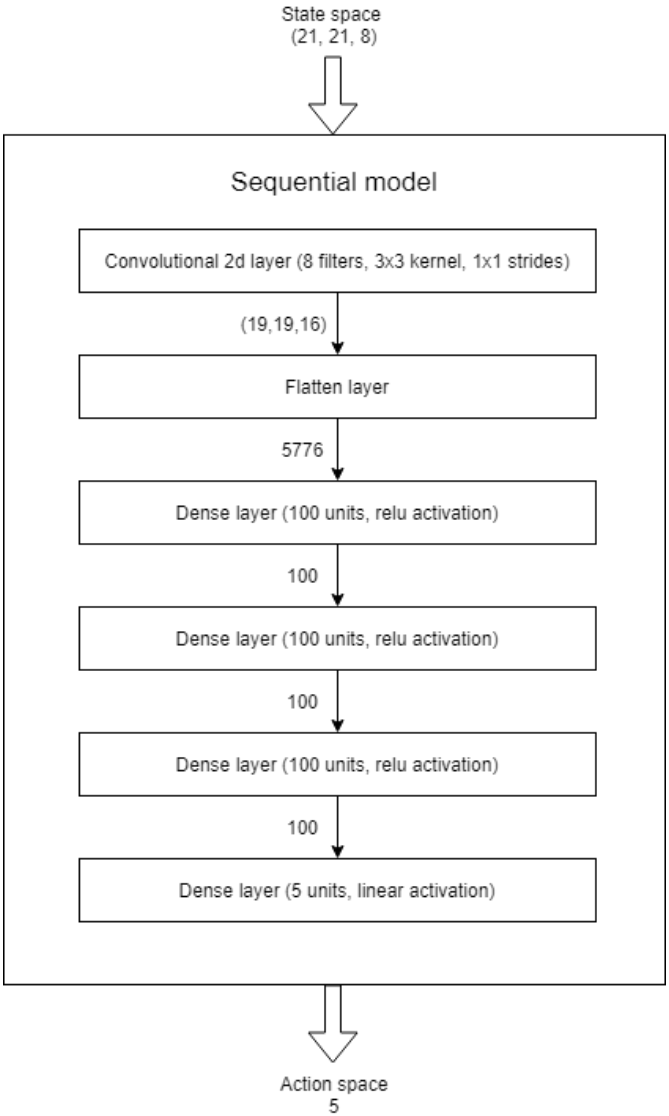


Figure 3.2: Baseline model

Action Replay

Our implementation uses the games played by the agent to learn strategies, these games are saved in an action replay. This means the games are saved but only the 100 most recent are used for training the agent.

Baseline inputs	Explanation
Halite	Amount of halite in each cell, normalized by division by 500
Moved friendly ships	Binary matrix with 1 in each cell containing a ship that has been moved, 0 otherwise
Not yet moved friendly ships	Binary matrix with 1 in each cell containing a not yet moved ship, 0 otherwise
Friendly shipyards	Binary matrix with 1 in each cell containing a friendly shipyard, 0 otherwise
Enemy units	Matrix with 0.5 in each cell containing an enemy ship, 1 containing an enemy shipyard, 0 otherwise
Small enemy units	Binary matrix with 1 in each cell containing an enemy ship with less cargo than the currently controlled ship
Timeframe	Matrix with all elements equal to the current timeframe
Cargo	Matrix with all elements equal to the currently controlled ship's cargo

Table 3.1: Inputs matrices for the baseline model explained

Frame Sampling

"...learning directly from consecutive samples is inefficient, due to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates." - [Mnih et al., 2013].

With this logic in mind, not only random *games* are selected and shuffled when preparing a training iteration, but the actual *frames* of a game are randomly sampled as well. Since a frame is the unit which the agents are actually trained on we performed random sampling for these as well in order to avoid the strong correlations of states within a game.

Baseline Reward Function

The reward function used for the baseline model can be seen in equation 3.1. It should be interpreted from *top to bottom*, setting the resulting reward to be the left hand side if the right hand side condition is met, otherwise moving on to the next condition.

$$r_t = \begin{cases} -50 & \text{if } s_{t+1} \wedge (p_{t+1} \in d_{t+1}) \\ -500 & \text{if } s_{t+1} \wedge (p_{t+1} \in o_{t+1}) \\ c_{t+1} - c_t & \text{if } s_{t+1} \wedge (c_{t+1} > c_t) \\ c_t & \text{if } s_{t+1} \wedge (c_t > 0 \wedge c_{t+1} = 0) \\ -1 & \text{if } s_{t+1} \\ -c_t & \text{if } \beta \\ -(c_t + 500) & \text{if } (p_{t+1} \in d_{t+1} \vee p_{t+1} \in o_{t+1}) \\ Q(s_t, a_t) & \text{otherwise} \end{cases}, \tag{3.1}$$

where t is the time frame, s_t the state at time t, p_t the position of the ship at time t, d_t the positions that are dangerous at time t, o_t the other ships' positions at time t, c_t the cargo of the ship at time t, β_t a boolean telling if it is the last frame of the game at time t and $Q(s_t, a_t)$ the estimation of the current state (at time t). The function is not as complex as it might at first seem, but is rather a translation of basic Halite rules into numbers.

Below the logic of the reward function is briefly explained, since it is a central part to any RL implementation. It outputs negative reward (-50) if the ship makes a move that could have ended up in getting killed by the enemy and a large negative reward (-500) if the move kills another friendly ship. The next two options give a positive reward for farming halite or killing enemy ships with more halite in their cargo, and for returning halite to a shipyard equal to the amount farmed or returned respectively. The fifth possible output is giving a small negative reward (-1) for actions that do not accomplish anything, incentivizing the agent to act. The last three outcomes are for when the ship is not in the next game state, either by ending the game or because it died. If the ship has remaining cargo on the last frame it receives a negative reward equal to the amount of cargo it has at the last frame. If it is not the last frame, however, it either died from an enemy or from poor movement, which yields a large negative reward (-500 - current cargo). If it died because of another friendly ship crashing into it, no penalty is given, which is the last condition in the reward function. The last check was introduced in order not to penalize behaviour which the current ship cannot influence directly. Instead the large negative reward is given to the ship that had information about the ship that moves first, but still moved and destroyed the friendly ship, which is taken care of in the second line of the reward function as explained above.

Opponent for Training

Since there is a significant difference in the way a game is played when there is competition present and not, properly training a model for a competitive environment requires an opponent. For the baseline model, the opponent was chosen to be

the bot placing 26th in the Kaggle competition (referred to as p26) with a modification. The skill level of p26 was modified by forcing it to do random moves with a set probability, for the baseline and most of the trained models, the opponent was set to p26 with a 15% probability of doing a random move, which will be referred to as p26 $\varepsilon = 0.15$ or simply p26_15. p26_15 was chosen for the standard training setup because p26 was ranked the worst out of the ones published and p26_15 proved to be a worthy opponent for the initial phases of the project.

Hyperparameters and Specifications

For setting up a DQN-agent for RL with a Keras NN model, one has to manually set a few hyperparameters and other adjustments. The baseline parameters are presented in table 3.2.

Baseline	
Discount factor, γ	0.9
Exploration rate, ε	0.2
Learning rate, α	0.0001
Batch size	32
Epochs	5
Sample rate	0.1
Loss function	MSE
Optimizer	Adam

Table 3.2: Baseline hyperparameters and properties

Aside from the discount factor and the exploration rate, which have been introduced above, there are a few more settings that were chosen for the baseline model. The learning rate α determines the speed at which the model learns in the sense that larger changes are made to the weights of the NN if α is large. The batch size is a parameter required for the fit function which determines how many input-target pairs there are in a batch. A batch being a subset of the input-target pairs which are also passed to fit. Having 5 epochs means that for each weight update the given input-target pairs are put into batches and trained with 5 times. The sample rate is a custom parameter introduced in order to reduce the number of similar states. A sample rate of 0.1 means only 10% of the frames are added as data to input-target pairs which is later passed to fit. The final two settings in the baseline, loss function MSE (Mean Square Error) and optimizer Adam (Adaptive Movement Estimation) are standard for Keras and determine according to which algorithm the loss and weight update should be carried out. All of the baseline parameters are set in the range of standard values for common practice machine learning.

3.3 Improving and Adjusting the Baseline Model

After establishing a working baseline, the project moved on to the second phase. This second major stage of the project was trying to apply additional techniques and/or change the setup for the baseline agent and analyze the outcome.

Neural Network Layout

A fundamental question when doing any kind of deep learning is to consider the layout of the NN. Both depth and width of the model was changed and tested for a few different combinations. For depth testing we tried models varying between 3 and 10 layers. Width-wise we tried between 10 and 500 units per layer. A full rundown of all combinations that were trained are presented below in table 3.3.

	Dense layers	
NN layouts	Width	Depth
Baseline	100	3
Smaller NN	30	3
Tiny NN	10	3
Tiny Deep NN	10	10
Wider NN	500	3
Deeper NN	100	6

Table 3.3: The different NN layouts covered. Note that all models have an initial 2d convolutional layer which is then followed by a varying setup of dense layers described in the table. After the wide dense layers is the final decision layer with a width equal to the action space of the model (=5).

For all models the width of the different layers were kept constant. Each layer of *baseline* are of width 100, each layer of smaller NN is 30 units wide etc.

Changing Reward Function

Changing the reward function has a great impact on the behaviour of the agent, since it is the only thing deciding what is good or bad. A few different experiments was made around the original baseline reward function, which only contains shaped rewards. Obtaining a sparse reward and utilizing it for effective training was one of the main goals for these experiments, since this in theory enables an agent to train with only the end goal in mind.

Input Change for Sparse Training The baseline did not include status of halite between the players, thus a model that got the scalar

$$\frac{ownhalite}{ownhalite + enemyhalite} \tag{3.2}$$

Pool1
baseline
wider_nn
enemy_data$_5$39
gamma01
gamma05
nearsighted
no_action
p26_5
p26_10
p26_15
p26
tiny_deep_nn
smaller_nn
random

Table 3.4: Pool1 of agents with varying skill level

as an additional input was tried. This trial was done with the same reward function as baseline with the added sparse reward. The reasoning behind adding this additional input was as follows: in order for a model to win and learn from the reward it is given, it will need some input data describing the status of the halite, essentially providing the NN with enough information to realize *why* it wins when it does. All models trained with a sparse reward had this change done to the input.

Switching to Sparse Reward after Training It was also tested if the time spent training a model with shaped reward could be used to achieve a sparse reward. After 500 games of baseline training, the reward function was changed to fully sparse in the test of switching to sparse reward.

Adding Sparse Reward Since the baseline did not have any sparse reward signal, this trial investigated what would happen if one was introduced by giving a 10000 reward if the game was won. This reward was given to all current ships in the last frame of the game if the game was full length, if the game was ended early only the ship that ended the game gained 10000 reward.

Adding Sparse Reward against Pool1 Baseline with the same sparse reward as above was also run against a pool of agents, see table 3.4. This pool consisted of a set of agents ranging from low to high Elo. This was run as an experiment for adjusting the previously described Adding Sparse Reward.

Fully Sparse Reward Baseline was also trained without all other reward signal except the sparse reward of the win condition.

Team Spirit With the previous reward function of baseline, the total reward given to each individual ship was collected before giving the sum to the ships. The expected future reward was also predicted by all ships and the average was given to each ship when the frame was calculated. This would give the ships reward for the team effort and also predict the outcome of the game as a team.

Higher Friendly Kill Penalty This change was made by adjusting the penalty when a ship collides with another friendly ship and kills it. The penalty was adjusted from negative 500 to negative 5000.

Training on Enemy Data

For the enemy perspective, we included the opponents' actions in the action replay, thus enabling our agent to see and make decisions based not only on how the agent played but also from its opponent during training. This was done by adding the enemies' states and actions to the action replay in the same manner that the agent's own state-action pairs are added. Since the opponent during the earlier stages of the training is much better at, for example, collecting and returning halite, the actions yielding the most reward are the enemy ship actions. This type of training is an attempt at behavioral cloning since, aside from the own training process, the agent can clone behavior from the opponent that yields positive rewards.

Discount Factor γ

The hyper-parameter γ is used in the update rule of the policy, and it is the discount factor of future rewards. In addition to the baseline model which uses a standard of $\gamma = 0.9$ three more models with different γ were trained: $\gamma = 0.1$, $\gamma = 0.5$ and $\gamma = 0.99$. This was investigated in order to get a better understanding of the long- versus short term planning aspect of DRL.

Compacting Input

The baseline model receives 8 inputs, each a 21x21 matrix extracted from data on the Halite board. Importantly, two of those inputs are single value filters (where the entire matrix is populated by the same number): the time frame and the ship cargo. In the compacting input trials a few of the input matrices were combined in order to attempt reducing the number of inputs while still maintaining important information. The model called *compact*6 combines the halite data with enemy ships with larger cargo (since they can effectively be attacked and farmed just like the halite) and also omits the data of friendly ships that have yet not made a move for the given frame. Because some environments may have amounts of observable data that is much larger than feasible, it was of interest to research if compacting the available data into less sparse matrices could be done without losing performance. Smaller input size can also speed up training.

Exploration vs Exploitation ε

For adjusting the rate of exploration of the agents, two different trials were done. Firstly, *eps_zero* which has the exploration rate ε set to zero during the entire training. Secondly, *eps_decay* which had ε decay from 50% exploration ($\varepsilon = 0.5$) to 10% ($\varepsilon = 0.1$) linearly over the first 400 games of training. The exploration parameter was investigated in order to analyze its impact in a multi-player competitive environment.

Near-Sightedness

This method was done by reducing the size of the input images to the model. By ignoring the cells that are far away from a ship, and only consider the ones close, the input size can be shrunk. Instead of 21x21x8, the images were set to 11x11x8 which results in 968 data-points instead of the previous 3528 data-points. This also reduces the the state-space since the number of possible configurations is drastically reduced. This removes information far away, making the agents "near-sighted". With the same logic as for compacting input, reducing the input available to the model can be interesting if the available information is too large to take all into account. Only taking into consideration the data in the nearest surrounding to the agent is a straight forward way of limiting the input to the seemingly most relevant data.

Choice of Opponent during Training

The agent's training opponent was altered in a few of the experiments in order to see how this impacted the final outcome. Three different training opponents were tried: p26, p4 and self. p26 as explained above is the agent placed 26th in the Kaggle competition, p4 is the agent placed 4th in that same competition and finally self is the current policy (note: the opponent without exploration). The reason this was tried was to see if there are benefits from battling stronger opponents during the training phase and if this forces the agent to "step up its game" in order to find any rewards. Keep in mind that when facing a more skilled opponent, the amount of halite available is likely to shrink and as such rewards will likely be more difficult to find.

3.4 Measuring Performance

Measuring performance of an agent in a multiplayer game is a bit different compared to doing it for a single player game where some performance measurement is usually built into the game. In a two player setting a lot of the choices depend on the play style of the opponent and the objective of this game in particular is to *win*, by having *more* halite than the opponent at the end, and not simply to have *a lot of* halite. With inspiration from already established methods for determining skill in two player games, an Elo arena was built. Here, our different bots — as well as

a few entries from the Kaggle competition — battled each other forming a ladder ranking based on the Elo ranking system.

4

Results

Determining which results are of interest for this kind of work is not an obvious matter. One issue is the fact that Halite is a multi player game, designed to be played against at least one opponent. Unlike work such as *Playing Atari with Deep Reinforcement Learning* [Mnih et al., 2013] where the performance of an agent is quantified by the score in the single player game of Atari games, Halite IV does not have such a simple measurement of success. Instead we will focus on two ways of presenting the results:

1. Plots with game data generated from full exploit games ($\varepsilon = 0$) versus the training opponent, with the game index on the x-axis. The default opponent is p26_15, which is forced to do 15% random moves.

2. The final 1v1 Elo rating of our agents as well as a few from the Kaggle competition from 2020.

The Kaggle competition bots will act as top-tier player benchmark standard, since for Halite IV there is no human expert to compare against like done in previous RL work with games such as Atari2600 [Mnih et al., 2013], Go [Silver et al., 2016] and Dota 2 [Berner et al., 2019]. Unless specifically mentioned otherwise, *halite* will always refer to the amount of halite collected and deposited to a shipyard, which is what determines the final score in the game. *Cargo* will always refer to the resources carried by a ship, i.e. halite that is aboard a ship and not deposited to a shipyard.

The plots that will be presented contain the following data:

1. Halite and cargo at the end of a game

2. Number of enemy and friendly ships killed

3. Game length

4. Average winrate vs. training opponent

5. Accumulated expected reward

6. Accumulated actual reward

These plots will be presented in the report for the most interesting outcomes and models. The accumulated and expected reward are not directly comparable in absolute values since the expected rewards will for every frame in every game also consider the predicted obtainable reward generated by the discounted future reward. The expected rewards will as such be much larger than the actual rewards in raw numbers, but the shapes and patterns of the two reward-plots can still be of interest to compare. Several of the plots will have a curves that represent moving averages. The window size for these curves are calculated with a window size set to 10 in order to smooth the curves and get a more clear picture of the trends. The non-smoothed raw data will still be presented in the same plots, but less highlighted. Also, for clarification, *friendly kills* refers to the number of own ships that are destroyed by own ships, not including own ships destroyed by enemy ships.

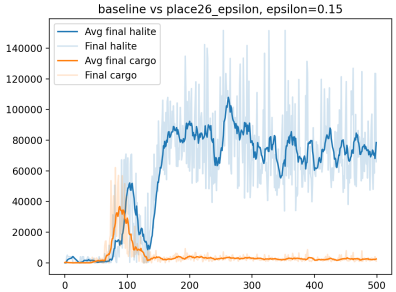
4.1 Naming Convention

The first number succeeding the model name plus an underscore specifies the size of the parameter in question (e.g. *gamma*_01 is the adjustment where the discount factor γ was set to 0.1). For different training opponents the name following the agent name plus an underscore specifies the opponent which the model was trained against.

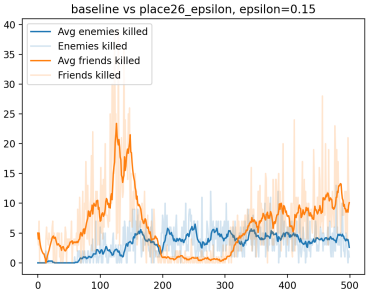
The standard baseline model that was trained for 500 games will be referred to as *baseline500* or simply *baseline*. Models that were trained for some other number of games will marked with a suffix equal to the number of games they were trained for, such as *baseline1000*.

4.2 Baseline results

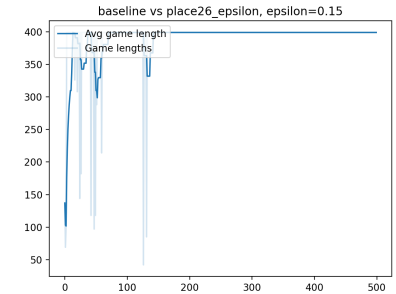
The main results from the *baseline* are presented below in figure 4.1



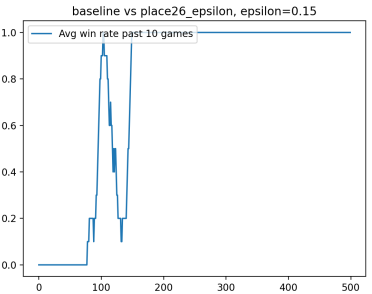
(a) Halite



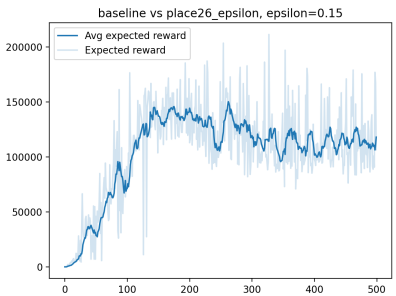
(b) Kills



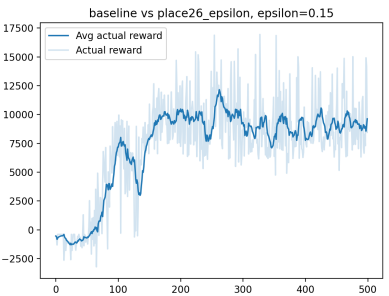
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



(f) Accumulated actual reward

Figure 4.1: Baseline results vs. *p*26_15

The halite in the last frame of each game can be seen in figure 4.1a. In figure 4.1b we can see how the number of enemy and friendly ships killed varies during the training. The length of the games and the average win rate against the training opponent can be seen in figure 4.1c and figure 4.1d respectively. In figure 4.1e

and 4.1f the ships' average accumulated expected and average accumulated actual reward is shown over the course of the training. For these last two plot are mainly important in order to see if there is a trend, and the absolute value on the y-axis is less important and is not directly comparable between models. They provide information about how well the agent manages to utilize the reward function and to what extent reward is found. We note again that the plots are generated from games with no exploration in order to see the results for the agent's current policy without random moves.

As seen in figure 4.1a the amount of halite at the end of the game starts off very close to zero but increases dramatically after about 80 games, hitting a local peak at around 50000 collected halite and 40000 cargo. The final halite then dips down to poor numbers below 20000 at around 120 games played and then climbs fast to reach a level of around 80000 halite while the final cargo approaches zero. The final halite remains fluctuating around this level for the remaining training.

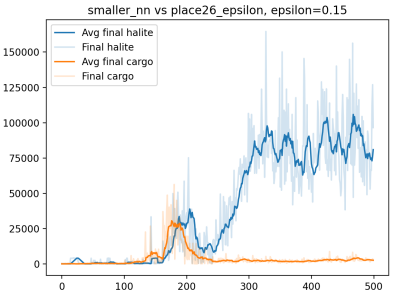
Moving on to figure 4.1b we see a sharp rise in the number of friends killed during the first third of the training process. It then declines to close to zero and after that rises up to killing between 5 and 25 ships per game. Enemy kills increase slowly over the first 200 games and then remains constant at around 5 kills per game.

The accumulated expected and actual reward seen in figure 4.1e and 4.1f respectively both follow the same pattern. They both start off low (note that the actual reward dips down to negative numbers) and then rise over the course of the training, finding a plateau after a couple of hundred iterations. The expected reward is a lot larger in absolute values due to this data containing not only the immediate reward but also the discounted future rewards. The dip in actual reward at around 120 games (figure 4.1f) coincides with the dip in halite collected (figure 4.1a) and the spike in friendly kills (figure 4.1b). This is due to the reward function giving less positive reward for the smaller amount of halite and more negative reward due to the number of friendly kills. This is one way of interpreting interesting patterns from the reward plots.

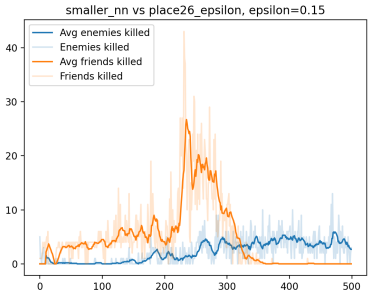
4.3 NN Layout Results

The changing of NN layout had great impact on the outcome of the models. Many of the trends seen in the baseline remain however.

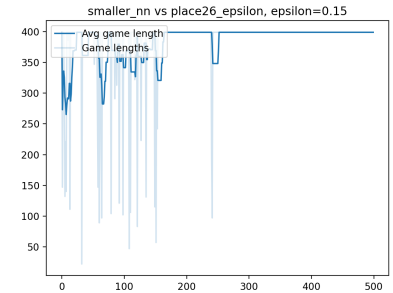
Smaller NN



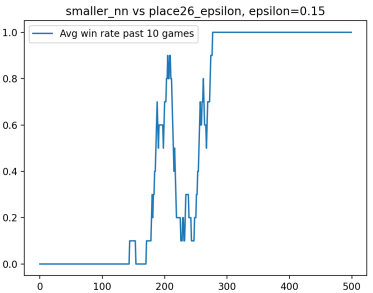
(a) Halite



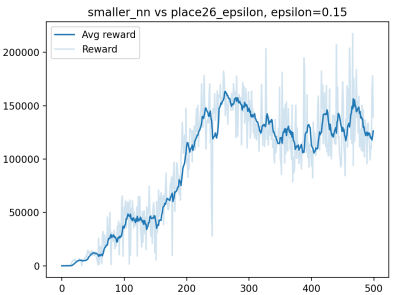
(b) Kills



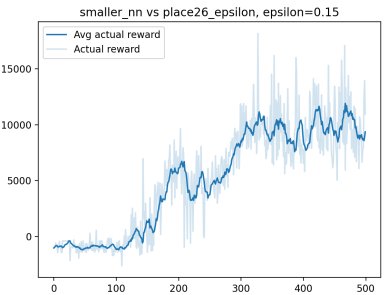
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



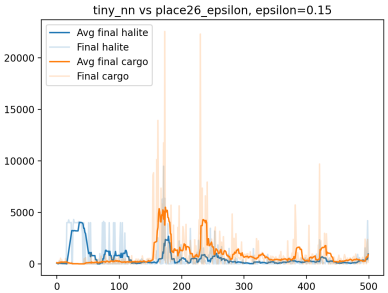
(f) Accumulated expected reward

Figure 4.2: Smaller NN results vs. *p*26_15

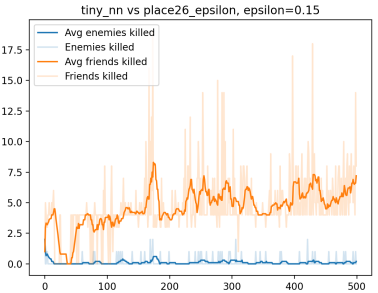
The smaller NN reaches a consistent halite score of around 80000 per game and little to no cargo left, see figure 4.2a. The number of friendly kills spikes during

the middle of the training but declines and reach a constant zero as seen in figure 4.2b. Compared to the baseline plots, these plots are stretched out along the x-axis, indicating that this agent is learning the same patterns but a bit delayed.

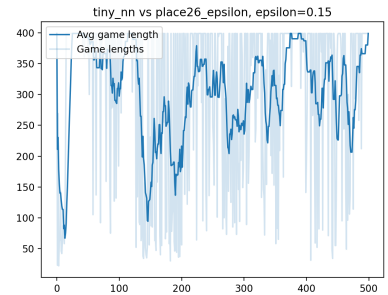
Tiny NN



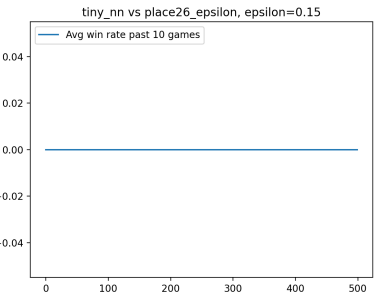
(a) Halite



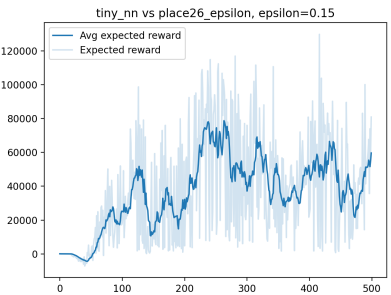
(b) Kills



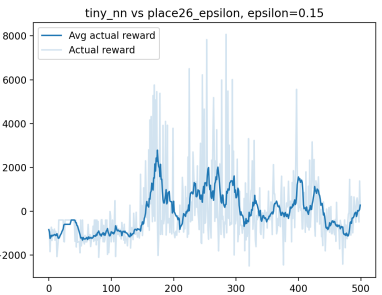
(c) Game length



(d) Average win rate past 10 games



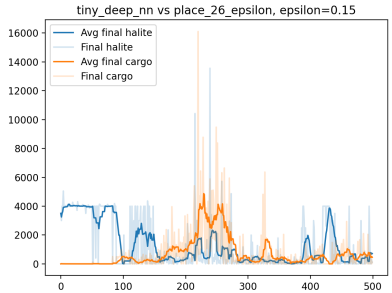
(e) Accumulated expected reward



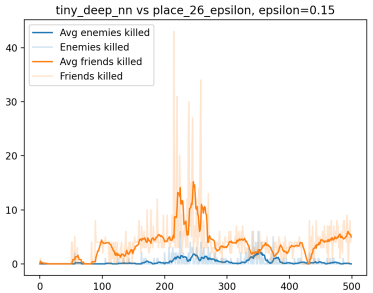
(f) Accumulated actual reward

Figure 4.3: Tiny NN results vs. *p*26_15

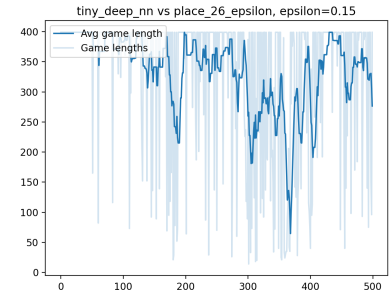
Tiny Deep NN



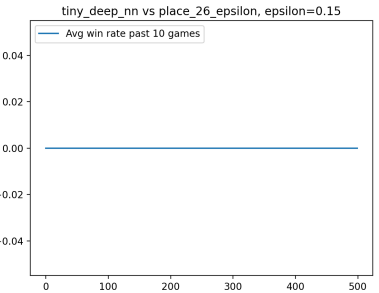
(a) Halite



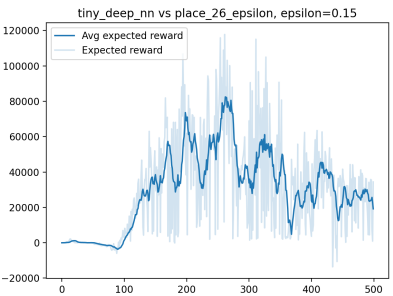
(b) Kills



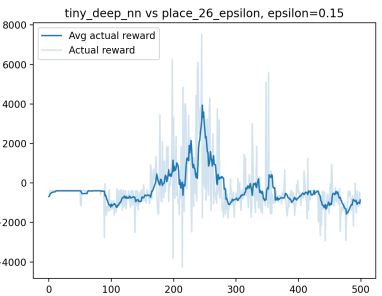
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



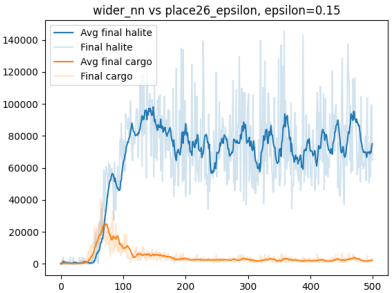
(f) Accumulated actual reward

Figure 4.4: Tiny deep results vs. *p*26_15

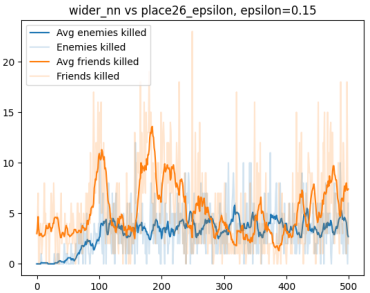
The plots for *tiny NN* and *tiny deep NN* have many similarities. *tiny NN* being only 10 units wide and 3 deep, and *tiny deep NN* being 10 wide and 10 deep. They both

result in disasterous halite scores, almost never surpassing 5000 per game, which is the starting amount. For both of the agents, a large number of the games end before 400 frames, indicating that the agent is being eliminated, see figure 4.3c and 4.4c. Also note that the win rate remains at zero throughout the entire training, see figures 4.3d and 4.4d.

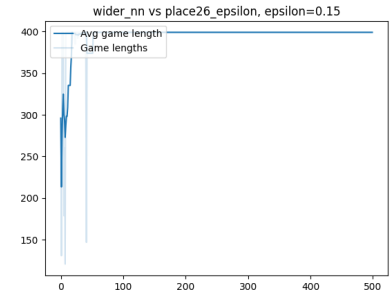
Wider NN



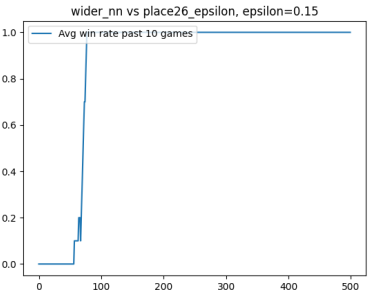
(a) Halite



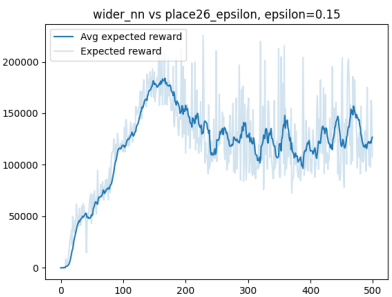
(b) Kills



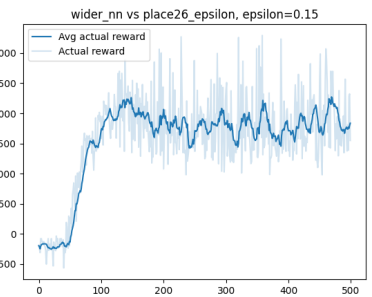
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



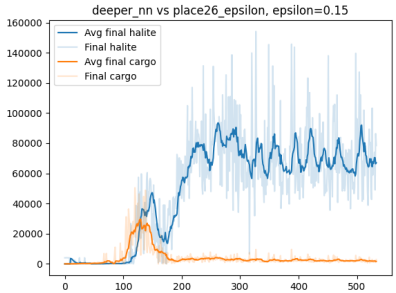
(f) Accumulated actual reward

Figure 4.5: Wider NN results vs. *p*26_15

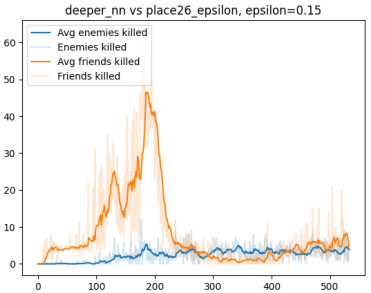
The *wider NN* reaches a consistent halite score of around 80000 in only 120 games as seen in figure 4.5a. The games also reach and stay at full length after only 50

games into the training process, see figure 4.5c, and the win rate reaches 100% before the 100 games played mark, see figure 4.5d. These patterns are found in the baseline as well, but appear faster with *wider NN*.

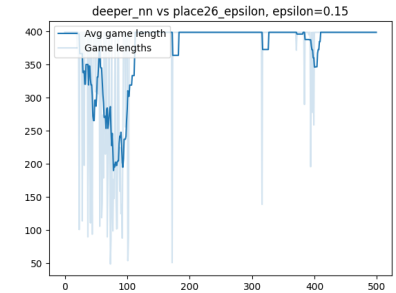
Deeper NN



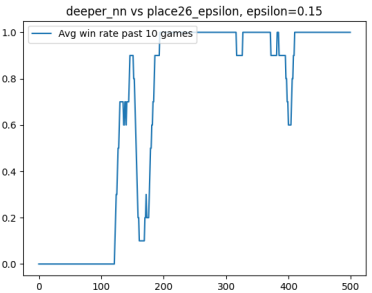
(a) Halite



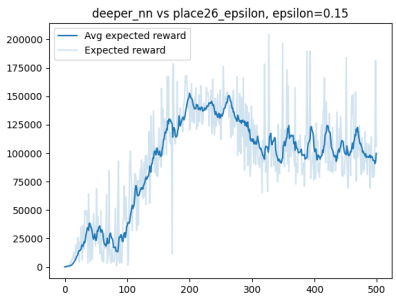
(b) Kills



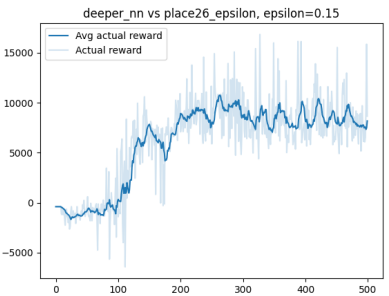
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



(f) Accumulated actual reward

Figure 4.6: Deeper NN results vs. *p*26_15

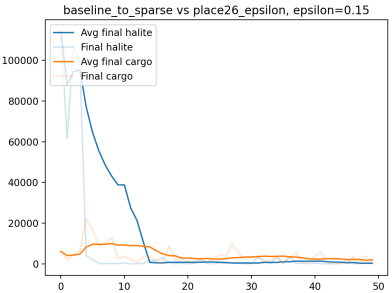
The deeper NN plots resemble the baseline plots very closely. One difference, however, is seen in figure 4.6c where some of the final games end before 400 frames. An-

other difference is the agent's non-consistent win rate as a few losses occur around 300-400 games, see figure 4.6d.

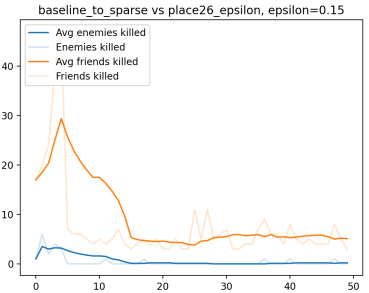
4.4 Changing the Reward Function

The plots showing the actual rewards for this section were removed due to not being able to correctly reflect the actual reward generated from the modified reward functions.

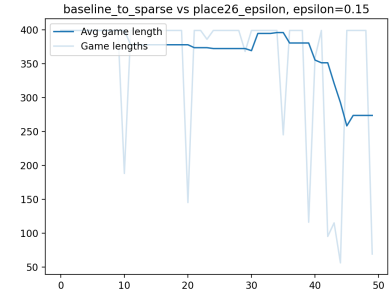
Switching to Sparse Reward after Training



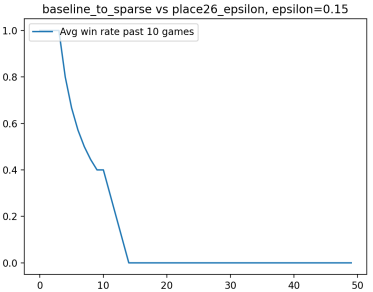
(a) Halite



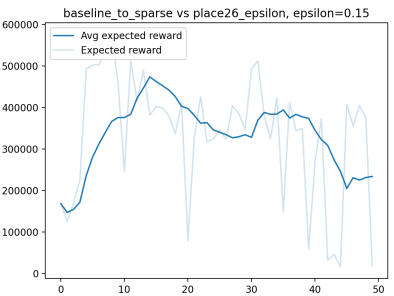
(b) Kills



(c) Game length



(d) Average win rate past 10 games



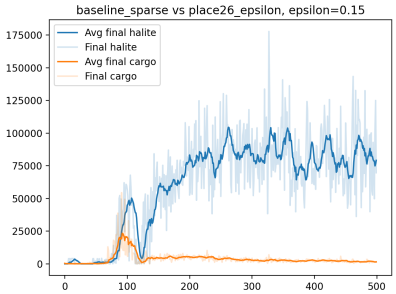
(e) Accumulated expected reward

Figure 4.7: Switching to sparse reward after training results vs. *p*26_15

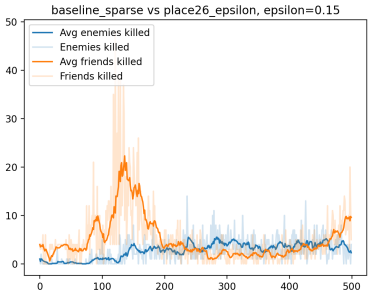
The *baseline to sparse* model is as mentioned a continuation of the baseline model, but with the reward function replaced with only a sparse reward given at the end

of a game, positive for winning and negative for losing. The baseline to sparse was trained only for an additional 50 games after changing the reward function. The halite score rapidly declines and remains close to zero, see figure 4.7a. The win rate plummets from 100% down to 0% in a few games of training, see figure 4.7d.

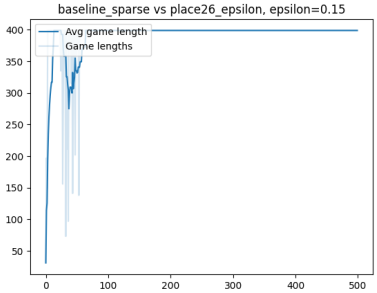
Adding Sparse Reward



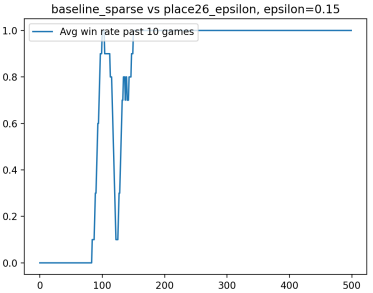
(a) Halite



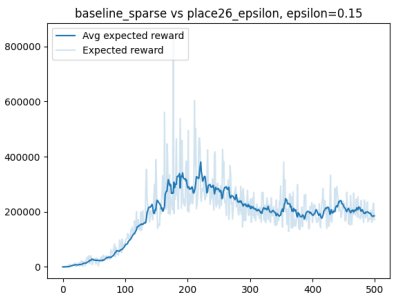
(b) Kills



(c) Game length



(d) Average win rate past 10 games



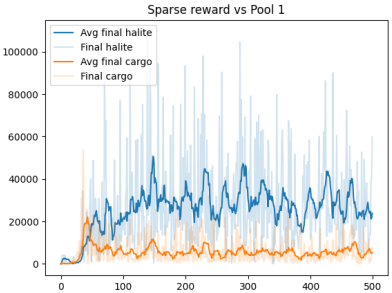
(e) Accumulated expected reward

Figure 4.8: Adding sparse reward results vs. *p*26_15

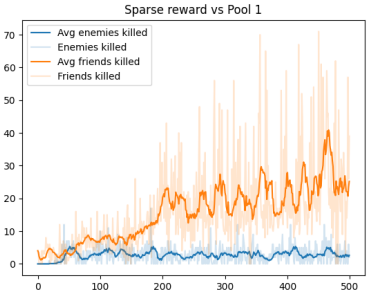
The *baseline with sparse* has a sparse reward included as an addition to the original reward function. These results are very similar to the original baseline plots, but the

expected reward plot, see figure 4.8e, is much larger due to the extra added reward.

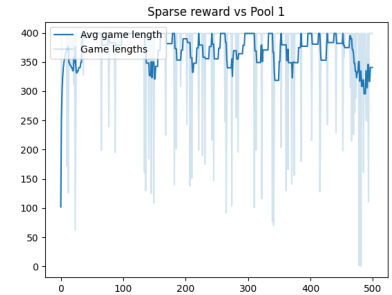
Adding Sparse Reward against Pool 1



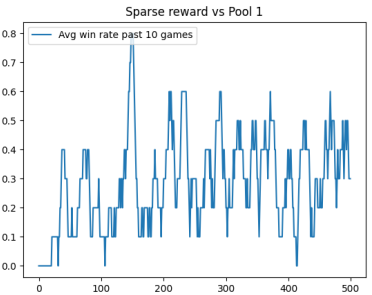
(a) Halite



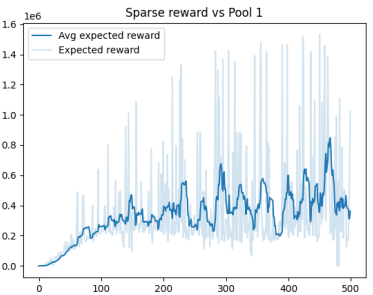
(b) Kills



(c) Game length



(d) Average win rate past 10 games

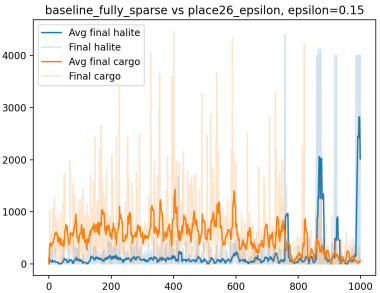


(e) Accumulated expected reward

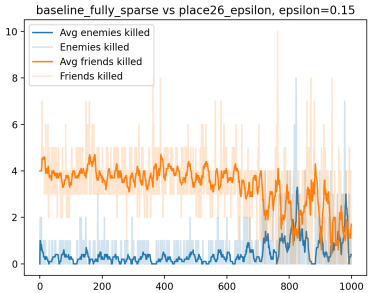
Figure 4.9: Adding Sparse Reward results vs. Pool 1

Facing several different opponents with a sparse reward added resulted in very different outcomes compared to the baseline. The halite score fluctuates around 30000 after 150 games of play. Note that the opponent in this case no longer is only p26_15 but instead a pool of opponents of different skill levels, some of our own DQN agents and some hard coded. As such, the greater fluctuation in the plots is expected and these plots are not fairly compared to other models' plots.

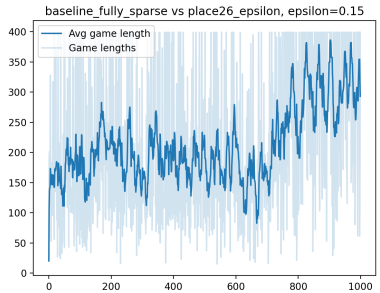
Fully Sparse Reward



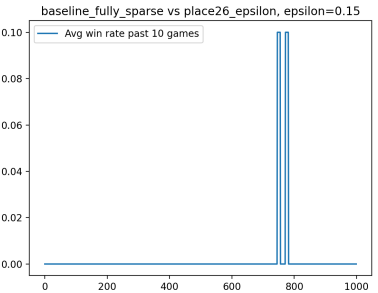
(a) Halite



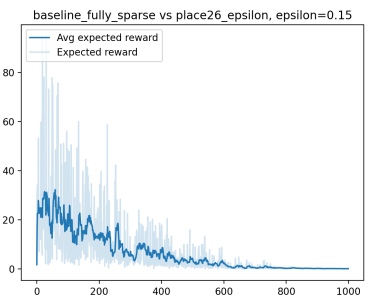
(b) Kills



(c) Game length



(d) Average win rate past 10 games



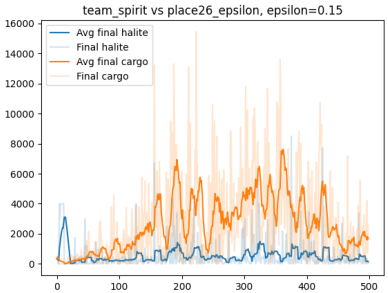
(e) Accumulated expected reward

Figure 4.10: Fully sparse reward results vs. *p*26_15

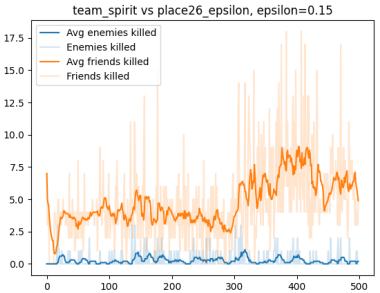
The fully sparse reward function resulted in a halite score only occasionally reaching above 100 halite, after 750 games of training, see figure 4.10a. The game lengths

are on average much shorter than the maximum 400 frames, see figure 4.10c and out of 1000 games the agent managed to win vs p26_15 only twice, see figure 4.10d. This very sparse reward makes it rare for the agent to receive any feedback and as such does not learn anything useful in 1000 games of play. The fact that the expected reward trends down to zero means the agent is not capable of finding the sparse reward and learns to expect not getting any reward at all, see figure 4.10e.

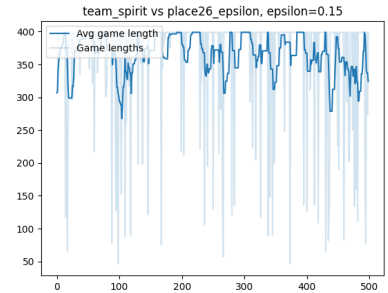
Team Spirit



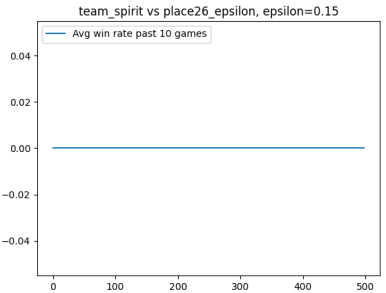
(a) Halite



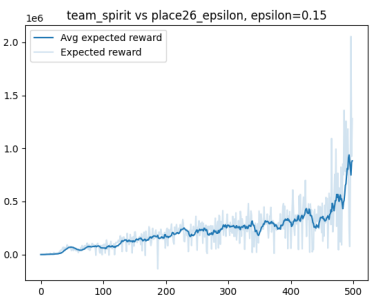
(b) Kills



(c) Game length



(d) Average win rate past 10 games



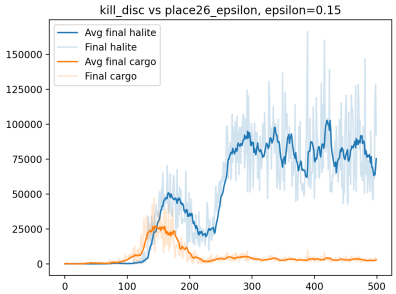
(e) Accumulated expected reward

Figure 4.11: Team based rewards results vs. *p*26_15

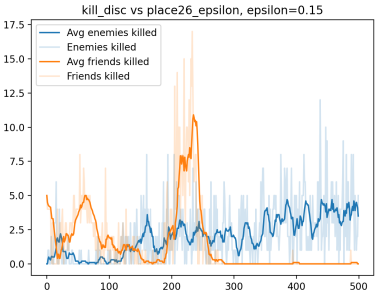
Team spirit with the team based reward function as seen in the figure 4.11 had a hard time converging to any solution for the environment. As seen in figure 4.11a the

agent never learns to achieve a higher end result than the starting 5000 halite. The game length seen in figure 4.11c also shows that the agent does not play well enough to survive the full length of 400 time frames. The previous statement is validated with the 0% win rate across 500 games as seen in figure 4.11d. In expected reward we see it rise toward the end of 1000 games, this seemed to be due to instability as other plots decline, see figure 4.11e.

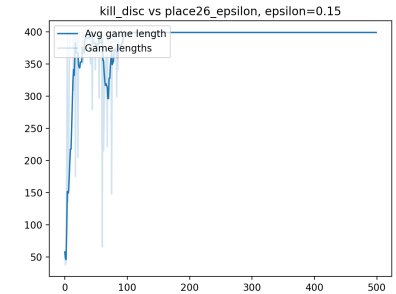
Higher Friendly Kill Penalty



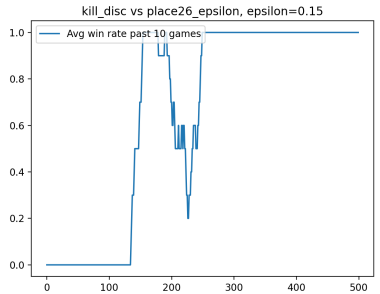
(a) Halite



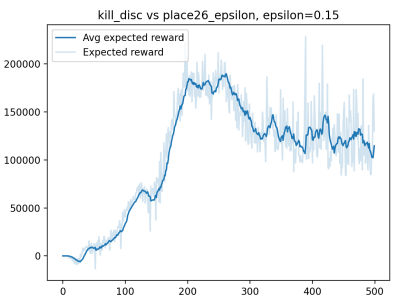
(b) Kills



(c) Game length



(d) Average win rate past 10 games



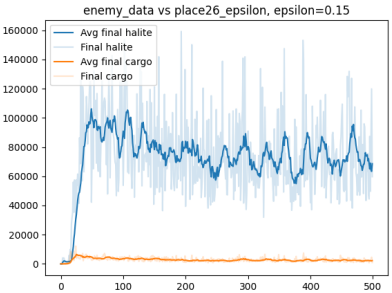
(e) Accumulated expected reward

Figure 4.12: Higher friendly kill penalty results vs. *p*26_15

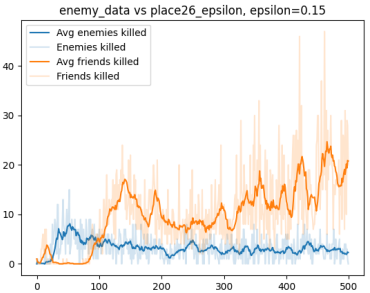
The higher friendly kill penalty showed that the agent indeed avoided killing friendly ships even after the episode count where the baseline model started killing

more friendly ships, see figures 4.12b and 4.1b. Other than the friendly kills the plots are similar to baseline plots without any notable differences.

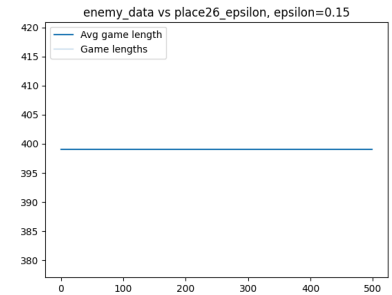
4.5 Training on Enemy Data



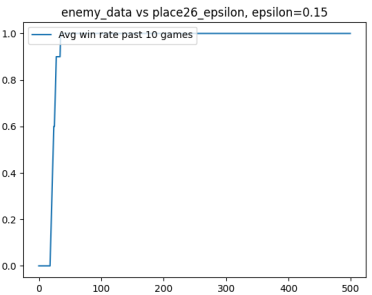
(a) Halite



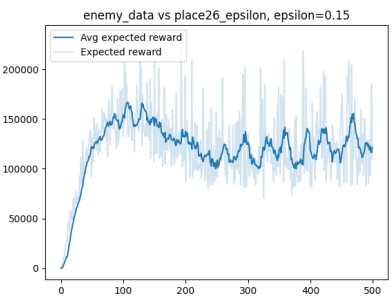
(b) Kills



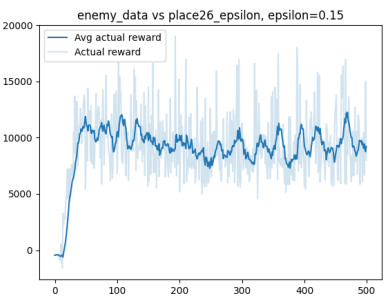
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



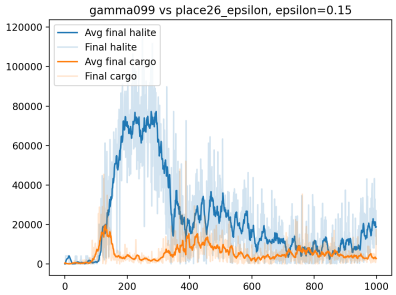
(f) Accumulated actual reward

Figure 4.13: Training on enemy data results vs. *p*26_15

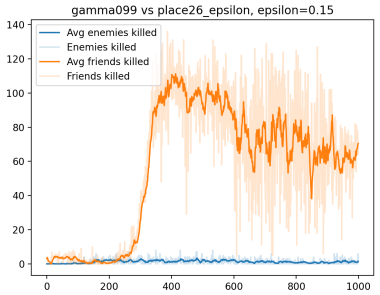
Allowing the agent to train on the enemies state-action pairs as well allowed it to learn initial strategies much faster. The amount of collected halite, the game lengths, win rate, expected and actual reward plots all indicate reaching higher performance levels much earlier in the training than the baseline. This is due to the opponent in the beginning of the training being much more skilled than the agent, allowing the agent to essentially copy moves that yield high rewards. The further training process, however, did not exceed the end result of the baseline, but stagnated around the same level. The number of friendly kills however seem to trend up beyond the baseline levels, see figure 4.13b.

4.6 Discount Factor γ

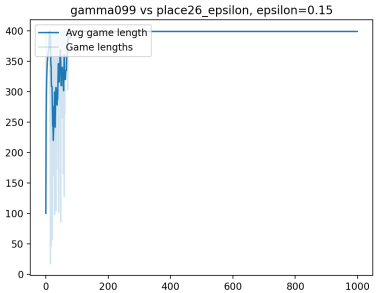
$\gamma = 0.99$ **1000 games**



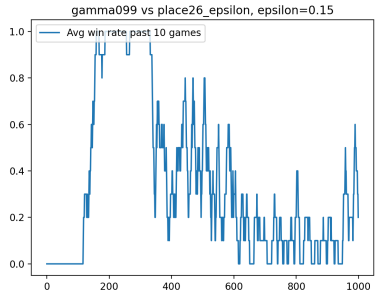
(a) Halite



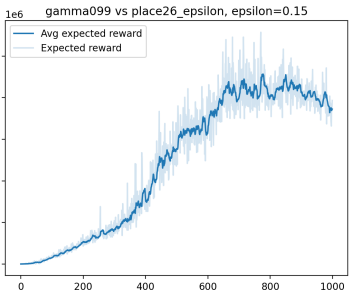
(b) Kills



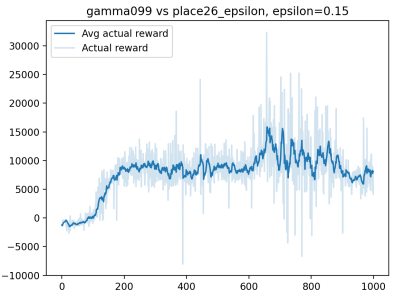
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward

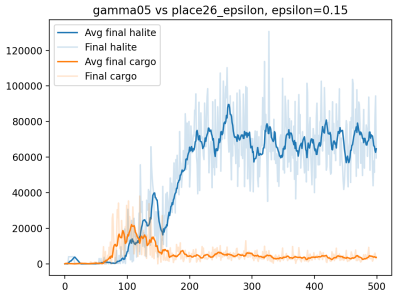


(f) Accumulated actual reward

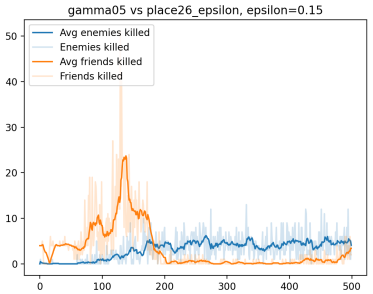
Figure 4.14: $\gamma = 0.99$, 1000 games results vs. *p*26_15

Firstly, note that the plots in 4.14 present data over 1000 games of training, not the default 500. The experiments with $\gamma = 0.99$ showed promise after 500 games of play. After the first 500 games it seemed to have overwritten its previous knowledge and perform worse after 500 games than after 250, see figure 4.14a and figure 4.14d. It was decided to train this model for an additional 500 games since predicting far into the future (which effectively is the case for a larger discount factor) logically will require more experience to predict on and there seemed to be a potential of good performance when looking at the peaks in collected halite and few friendly kills around 250 games, see figures 4.14a and 4.14b respectively. However after 1000 trained games both the halite collected and the win rate remained low and the accumulated expected reward and actual reward stagnated.

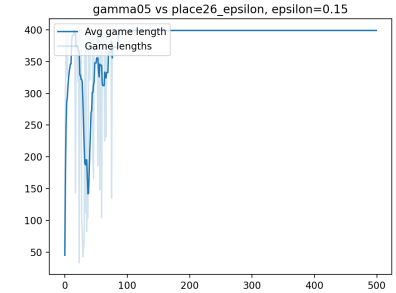
$\gamma = 0.5$



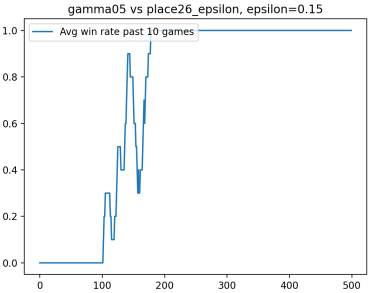
(a) Halite



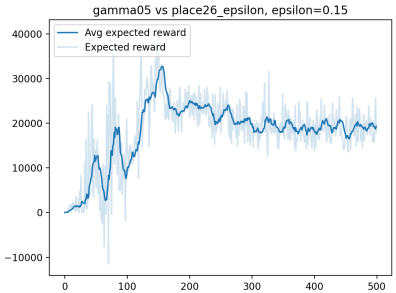
(b) Kills



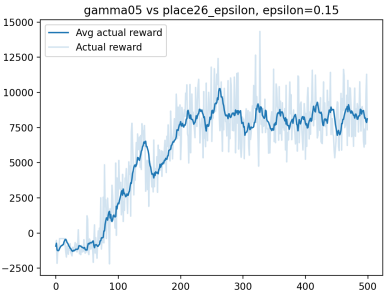
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



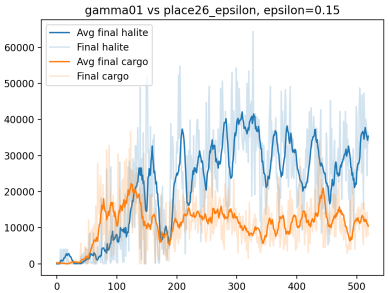
(f) Accumulated actual reward

Figure 4.15: $\gamma = 0.5$ results vs. *p*26_15

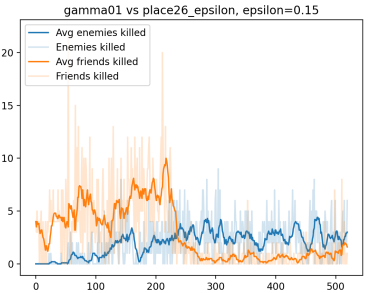
With $\gamma = 0.5$ the outcome is very similar to the baseline model. The expected reward differs greatly however, since the smaller discount factor reduces this value a lot, see

figure 4.15e.

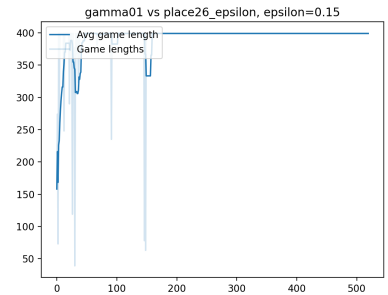
$\gamma = 0.1$



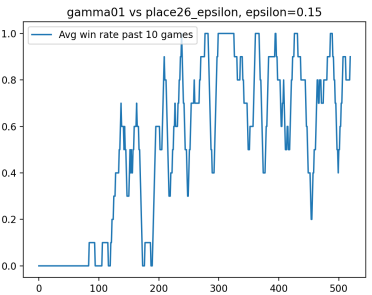
(a) Halite



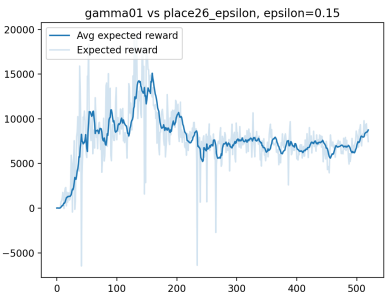
(b) Kills



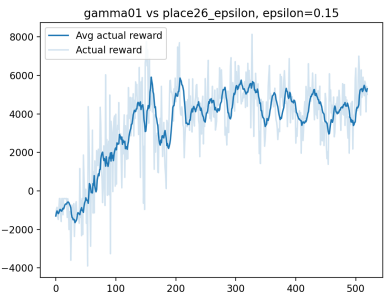
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



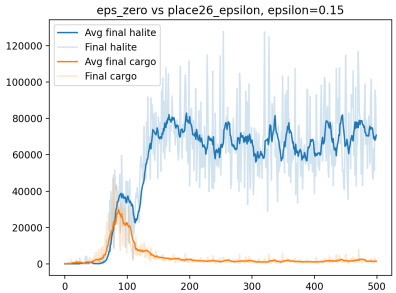
(f) Accumulated actual reward

Figure 4.16: $\gamma = 0.1$ results vs. *p*26_15

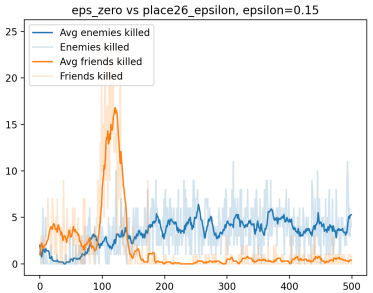
With $\gamma = 0.1$ the collected halite peaked at a level of around only 40000, see figure 4.16a. The number of friendly kills are low, but so is the win rate, see figure 4.16b and figure 4.16d. Being this short-sighted in a game like Halite IV where a bit of planning ahead is useful seems like a poor choice.

4.7 Exploration vs Exploitation, ε

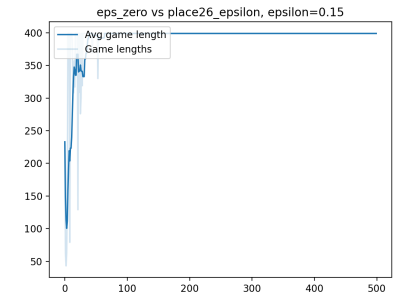
$\varepsilon = 0$



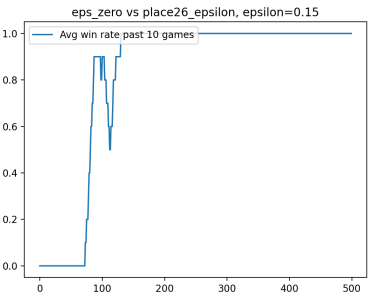
(a) Halite



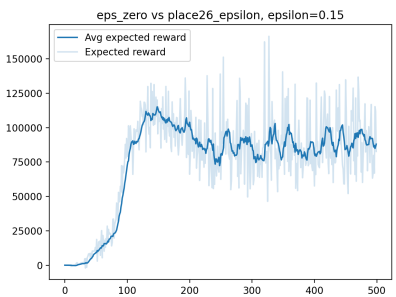
(b) Kills



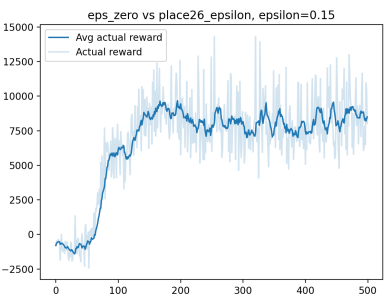
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward

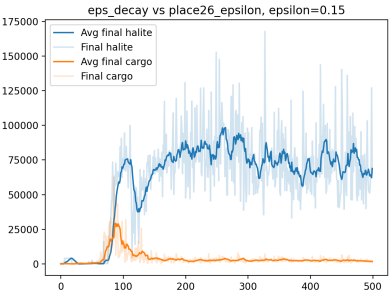


(f) Accumulated actual reward

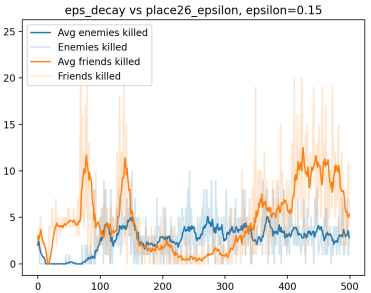
Figure 4.17: $\varepsilon = 0$ results vs. p26_15

Without exploration ($\varepsilon = 0$) the plots are very similar to the baseline, with a halite level around 80000 and the same pattern arising around 100 games of play, see figure 4.17a. The other plots are very similar as well, but the number of friendly kills (see figure 4.17b) end up very close to zero at the end of the training. Ignoring the traditional exploration and instead always selecting the action with the most expected reward in this type of environment seems to have little drawbacks.

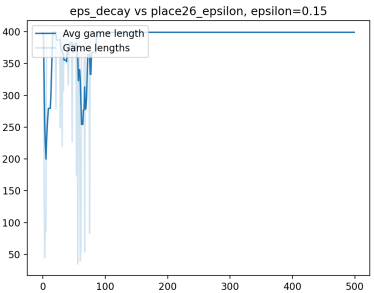
ε decay



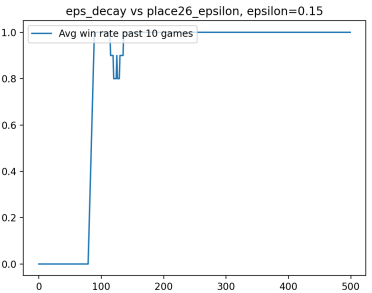
(a) Halite



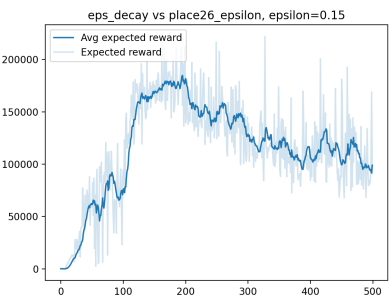
(b) Kills



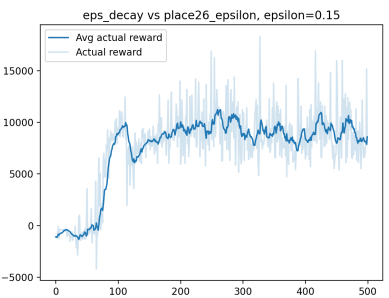
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



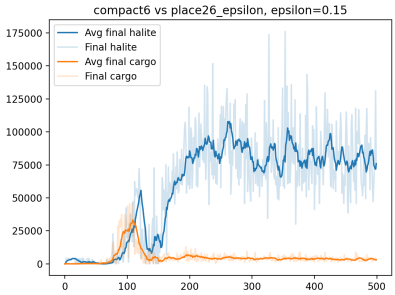
(f) Accumulated actual reward

Figure 4.18: ε decay results vs. *p*26_15

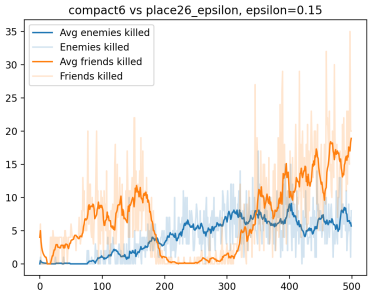
eps_decay which had the exploration rate parameter ε decay from 50% to 10% linearly over the first 400 games of the training also yielded plot results very similar to

the baseline. Note however how the expected reward seen in 4.18e declines steadily from 200 to 500 games played while the actual reward in figure 4.18f remains stable.

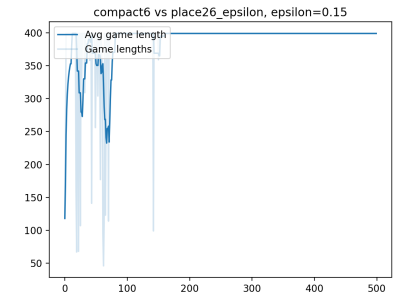
4.8 Compacting Input



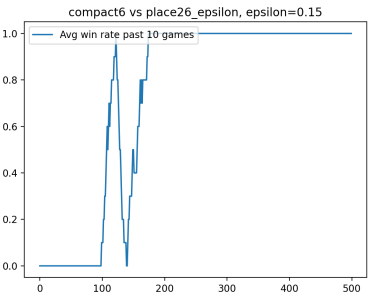
(a) Halite



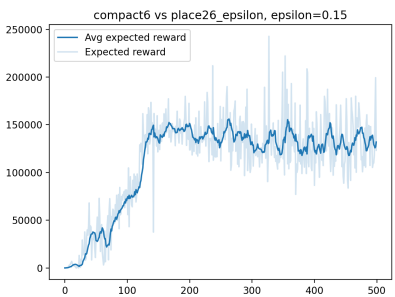
(b) Kills



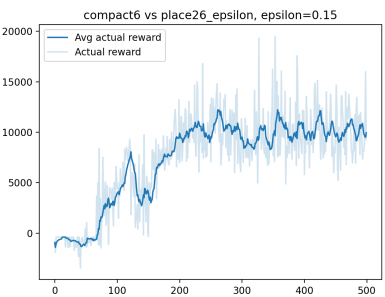
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward

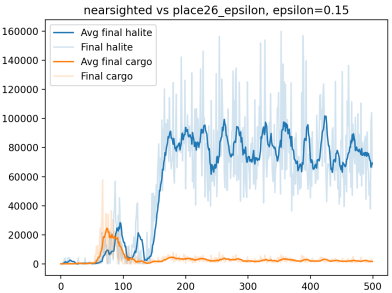


(f) Accumulated actual reward

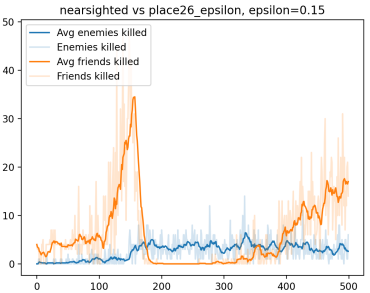
Figure 4.19: compact6 results vs. *p*26_15

The compact6 experiment resulted in plots resembling the baseline with virtually no notable differences. This, however indicates that condensing data when parsing input, making the input less sparse, can (for at least some cases) be done with little loss of performance in the resulting agent.

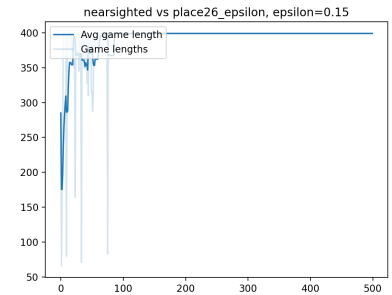
4.9 Nearsighted



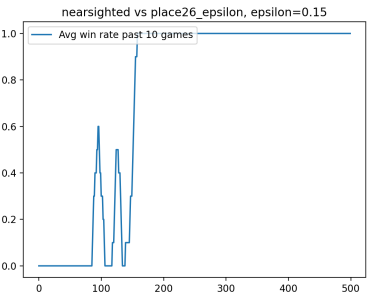
(a) Halite



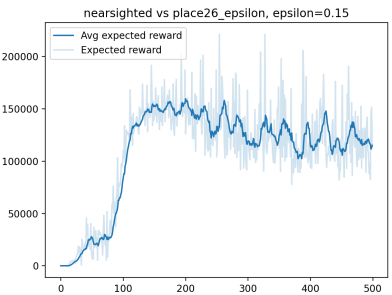
(b) Kills



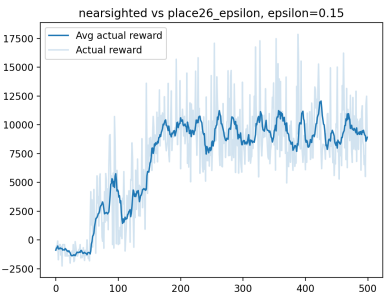
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



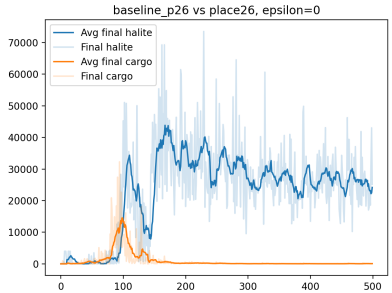
(f) Accumulated actual reward

Figure 4.20: Nearsighted results vs. *p*26_15

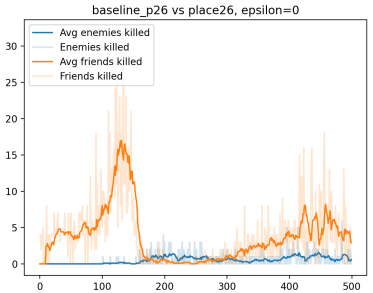
The nearsighted agent shows plot results very similar to baseline. Accumulated actual rewards and expected rewards behave in almost identical fashion, see figures 4.20e and 4.20f. Similar to the compact6 model this version also has access to less information when learning but still manages to reach the same levels of halite collected and has plot patterns that resemble the baseline's closely.

4.10 Changing Training Opponent

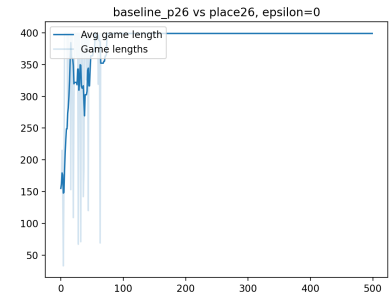
Against P26_0, P26 with $\varepsilon = 0$



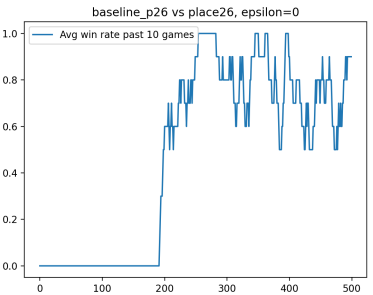
(a) Halite



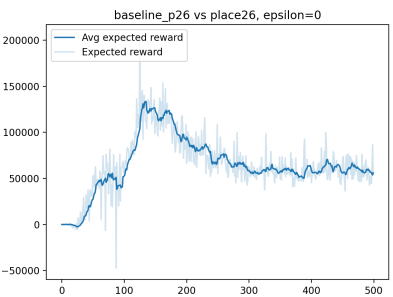
(b) Kills



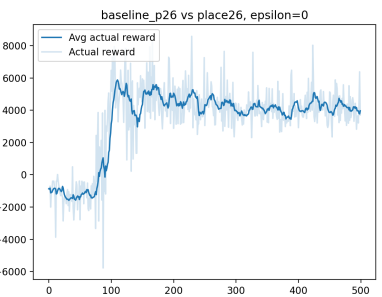
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward

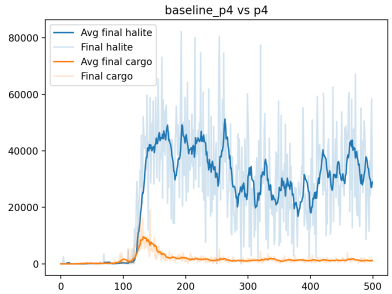


(f) Accumulated actual reward

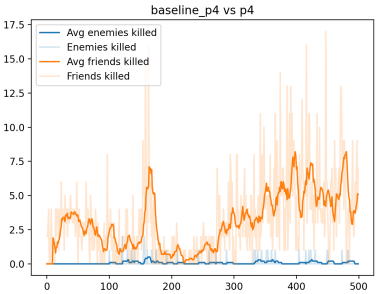
Figure 4.21: Training against p26, $\varepsilon = 0$, results

Training against *p_26* with no forced random moves was initially thought to be a very difficult task. However the agent managed to win games around 80% of the time after 200 games. The amount of halite collected is a lot less, most likely to the fact that *p_26* collects the available halite more efficiently than *p_26_15*, leaving less for its opponent.

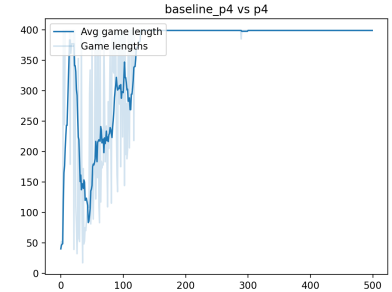
Against P4 with $\varepsilon = 0$



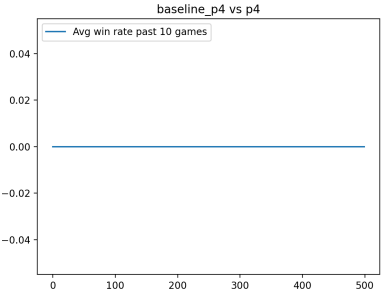
(a) Halite



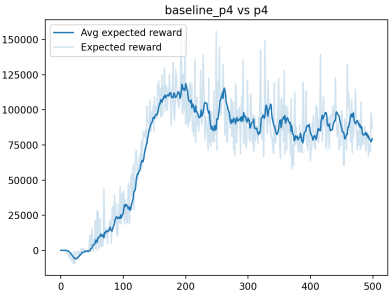
(b) Kills



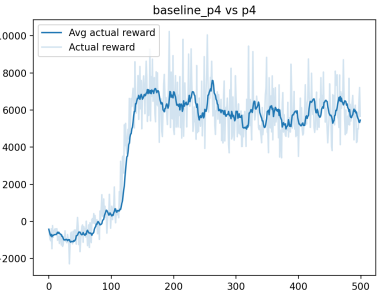
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



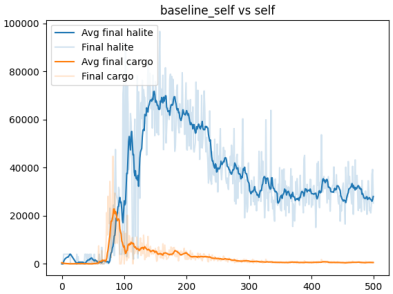
(f) Accumulated actual reward

Figure 4.22: Training against p4, $\varepsilon = 0$, results

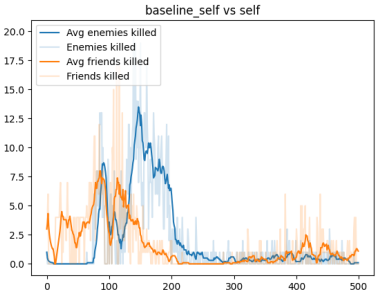
Training against an even more skilled opponent, *p_4* (the 4th placed entry in the Kaggle Halite IV competition), resulted in no wins during the training (see figure

4.22d) and less halite collected than the baseline (see figure 4.22a). However, note again that changing the opponent will make the absolute values in the plots compare unfairly between to models' plots since the game becomes harder with a harder opponent. Instead it is useful to consider the accumulated and actual rewards, see figure 4.22e and figure 4.22f. The positive trend in the rewards indicate that the learning process is effective and the agent is capable of learning to accumulate more reward until around 150 games of training after which the reward remains constant.

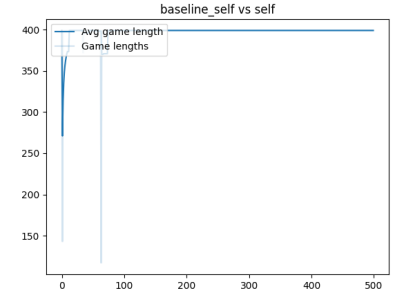
Against Self with $\varepsilon = 0$



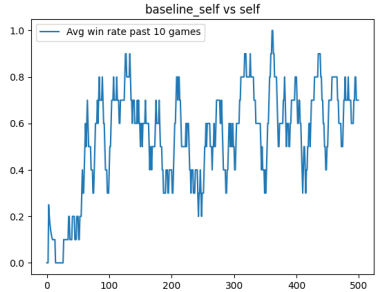
(a) Halite



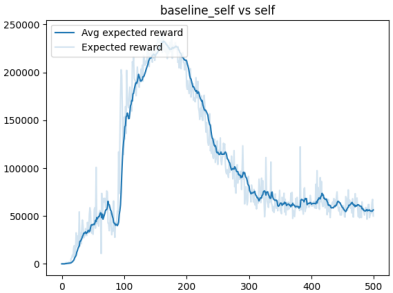
(b) Kills



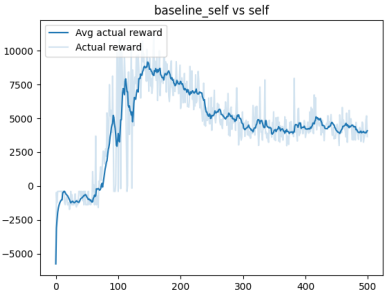
(c) Game length



(d) Average win rate past 10 games



(e) Accumulated expected reward



(f) Accumulated actual reward

Figure 4.23: Training against self, $\varepsilon = 0$, results

The results for training against the model itself are very interesting. The halite results seen in 4.23a indicate that the agents farm slowly which will allow the halite

available on the map to grow and generate more total obtainable halite over the course of the game. However when the agent learns to farm and collect the halite faster, it opponent (itself) will also gather faster which will result in a race to farm the halite depleting the resources and emptying the map faster. This pattern can be seen both in the expected and actual reward as well, as seen in figures 4.23e, 4.23f. The win rate (see figure 4.23d) fluctuates around 50% during the whole process as expected.

4.11 Elo scoreboard

The final Elo scoreboard after 12998 total games played can be seen in table 4.1

	SCOREBOARD	Elo	wins	losses
1.	p6	2475	307	0
2.	p4	2329	291	8
3.	eps_zero	2221	283	22
4.	baseline_self	2082	265	37
5.	deeper_nn	2074	250	50
6.	eps_decay	2038	292	54
7.	kill_penalty	2036	293	51
8.	smaller_nn	1972	251	71
9.	baseline_p4	1960	236	63
10.	wider_nn	1928	221	66
11.	enemy_data	1867	241	110
12.	p26	1842	204	76
13.	baseline_400	1764	213	103
14.	baseline_1000	1734	227	104
15.	baseline_450	1726	198	113
16.	baseline_250	1670	187	133
17.	baseline_300	1662	183	127
18.	baseline_p26	1649	172	128
19.	baseline_500_b	1644	184	133
20.	nearsighted	1618	186	120
21.	compact6	1609	178	153
22.	baseline_500_a	1598	201	141
23.	baseline_350	1581	172	145
24.	with_sparse	1455	133	169
25.	baseline_200	1369	137	211
26.	gamma05	1346	153	196
27.	baseline_150	1247	103	216
28.	with_sparse_pool1	1221	116	201
29.	gamma01	1198	105	244
30.	baseline_100	1198	92	222
31.	gamma099	1071	58	254
32.	p26_15	1061	84	256
33.	gamma099_1000	1004	53	269
34.	no_action	948	55	281
35.	fully_sparse	905	53	240
36.	tiny_nn	788	26	276
37.	to_sparse	737	24	291
38.	baseline_50	736	20	308
39.	random	722	20	296
40.	tiny_deep_nn	717	20	263
41.	team_spirit	677	12	298

Table 4.1: Table showing all agents' Elo ranking after 12998 total games simulated. The number of wins and losses for each agent is also displayed.

5

Discussion

5.1 Overview

As seen in the result section there is a wide range of plots to consider, each containing data interesting for having different discussions. In this chapter we will explain our reasoning when constructing the baseline and discuss the resulting model and its performance. Then we will go into depth about the other implementations built upon the baseline and highlight the most interesting results. Our conclusions will be drawn in relation to how the other methods compare to the baseline and to each other.

5.2 Construction of Baseline

Our first challenge in this project was to realize that our goal was not to understand how to solve a problem using RL but rather how to solve a problem using DQN as a DRL method. DRL has many methods which are used to train models and each of these have their own strengths and weaknesses which would be interesting to compare. However, as each method also has many parts that can be adjusted as hyper-parameters and other methods that can be added it was quickly realized that this project would need to first compare DQN with its hyper-parameters and a few additional methods and try to answer the questions how and why they are set and used. As this became our goal we first needed to develop a baseline which would be used to adjust and gather results that describe how and why each adjustment change the outcome.

Agent Perspective

As stated in the method our solution to train our DRL agent in Halite IV was to take one action for each ship using the same neural network. This was key to make our DQN approach simple and straight forward, DQN predicts the expected reward for each possible action in one state. This means that the player which has a varying number of actions to take from each state would be hard to realize as DQN model

since the size of the of the model output is set when training the model. With predicting the expected reward of each ship with its own actions which is always the same in the environment it is clear how the model will be structured. By also utilizing the fact that each ship can be centered in the environment the model will be able to predict each ship with same logic wherever the ship is located. This also allows for the model to gain more training data since each episode of the environment often contains more ships than one which leads to similarities between our solution and MARL. We also need to regard the drawbacks from MARL, one of which is the fact that each ship that the agent is responsible for should play the game with the aim of winning and not simply maximizing its own reward by sabotaging the team. In short our solution allows us to create and train a DQN solution with relatively small effort but with the drawback that the optimization of each ship individually leads to worse performance for the agent as a whole.

Reward Function

The reward function is a vital part of the RL algorithm. The agent will without any regards to the actual game maximize the reward and nothing else. Since this is the case, it is hard to implement other reward signals than the obvious win condition that is present in competitive games. However when the reward is far in the future it gets harder for the model to find the path to the reward. Especially if the agent is trained against a skilled opponent it might never get any reward.

In figure 4.10 we see results of training our baseline model with only sparse reward signals. Here the model was only able to win 2 games out of 1000, see figure 4.10d. This made the model expect 0 reward in all states, see figure 4.10e. With these results, it became clear for us that the baseline would need more frequent rewards if the agent was to be able to perform any results in our time scope and computational power. Thus, our reward function as stated in equation 3.1 became our baseline for reward. Here we added the rules from the game in the reward function to make rewards much more frequent since a shaped reward is in this case given for every single decision. We also included if the ship survived by luck, that is to say if an enemy ship with less amount of cargo is nearby and our ship moves to a cell where the enemy could have moved we give a negative reward to make the agent realise faster that such enemy ships are to be avoided. Also, an action that does not yield any reward was punished with a small negative reward instead. This encourages the agent to move and explore the environment but also to take the shortest path to each reward. Since moving about and not gathering positive rewards yields a negative reward, the agent will in the long run receive accumulated negative rewards. This will not be an optimal way to act for maximizing the rewards and the only way to avoid these small punishments the agent is required to find ways of finding positive rewards by gathering halite, killing enemies etc.

Our baseline reward function does not include any win condition, this was because our method of making the agent take one action for each ship made constructing rewards for the game hard to formalize. Our logic was that each ship could get a reward for its action based on the general rules for Halite, if the ship farms halite it could be given the amount farmed as a reward, if the ship returns the halite to a shipyard it could get the returned halite as a reward. Each of these was strongly related to what action the ship made in each frame of the game and thus easily provided to the agent making the action. However when the game is won it would be the player that wins and not the individual ships, thus making a reward for an action abstract and hard to design. It is difficult to determine which actions actually did lead to the final outcome and also how valuable a win is compared to the other feedback given in the reward signal. With this logic, it could be hard for the agent to understand what action lead to the potential reward in the terminal state of the game. Logically, the magnitude of the win condition should also be much greater than the shaped rewards since this is the final objective of the game itself, and strategies for hindering the opponent should be rewarded if they exist. Since this proved hard to determine we made the decision that baseline would start without the win condition reward signal, both to make the learning more stable but also to be able to gather results with and without the reward.

5.3 Baseline outcome

The final outcome of the baseline was an overall success. We will go in depth into the results of the baseline itself since many of the traits in the baseline will be found in the results of other models as well since they all build upon the baseline model.

From the halite plot of baseline, see figure 4.1a, several conclusions can be drawn. Firstly, the baseline agent makes great use of the reward function enabling it to learn to play the learn to gather halite after around a hundred episodes. It learns to master the concept of collecting but not returning halite first and afterwards learns to return more and more of the cargo in order to maximize its reward. This is due to two design implementations in the reward function. Firstly, returning halite awards the agent with a reward equal to the amount of halite returned. Secondly, the reward function penalizes a ship containing cargo in the final frame with a negative reward equal to the amount of cargo. Gathering halite is a simpler behaviour than returning it to a shipyard since the former requires only the agent to stand still at a cell containing halite while the latter requires identifying the meaning of the shipyard input, the cargo input and then travel to a shipyard when it has a large amount of cargo. The latter requires learning a sequence of steps in order to achieve. The rise and fall of the cargo curve in figure 4.1a aligns with this explanation.

The final part of the plot seen in figure 4.1b is surprising since the reward function heavily penalizes killing friendly ships. One would expect the agent to minimize its losses by not killing friendly ships, but since the win rate, see figure 4.1d, stays at a stable 100% and the amount of halite the it manages to return does not diminish, the agent seemingly offers up its own ships as part of its strategy for maximizing rewards on other fronts. However, it is hard to draw any definite conclusions about this behavior since the agent is simply trying to maximize the cumulative rewards.

5.4 Elo

The Elo rating table, see table 4.1, offers a very quick overview of the rough outcome of all the adjustments made to the baseline. Unlike the plots the Elo also reflects the performance of an agent against many different opponents, which is of interest when trying to determine if something is good or bad in a competitive environment.

5.5 NN layers

In our work with the baseline we realized feature extraction would be a focal point of our model. Because the input data for the model from Halite IV are images it would be computationally heavy to do a feature extraction with only dense layers. We separate the data in a halite frame into 8 matrices (which can be seen as images), each containing one class of items. One is the halite on the board, another one is enemy ships with less halite than the currently controlled ship and the enemy shipyards, in short all things we should avoid. These could be on the same cell on the board but the separation in the input allows the model to easily understand what classes occupy the position on the board. This is where convolutional layers are strong. They are used in many image processing machine-learning applications and can compress images and output new ones with data from all other images to a new one. For more reading regarding feature extraction from images see [Latif et al., 2019]

Since we were able to greatly reduce the number of parameters to train, and convolution is a widely used layer type in the industry when having images as input, it was decided it would be part of our baseline.

5.6 Simplifications

The simplifications made to the agents, such as excluding the convert action from NN and not letting a NN control the shipyards, does limit an agent's theoretical final

potential. However, these simplifications allowed us to implement a working DQN solution much faster. The agents themselves also adopt initial strategies faster and are still able to learn to perform on a level comparable to completely algorithmic agents. In order to fully explore the full potential of a DQN-agent in the Halite IV environment, these simplifications would need to be removed, letting a NN control both the convert action and the shipyards.

5.7 Techniques and parameters

NN layout

In our work we realised the importance of designing the NN model as a key problem when constructing a DRL solution. The NN layout is tasked with extracting important features from the observation and using this to make a decision which should be optimised for solving the problem. As such, analyzing the consequences of changing the layout of the NN turned into a top priority. Among the five different NN layout variations that were tried (in addition to baseline), the outcomes were vastly different. We will start discussing the NN layouts with fewer units than the baseline, *smaller_nn*, *tiny_nn* and *tiny_deep_nn*.

The *smaller_nn* with 3 layers of 30 units instead of 100 per layer resulted in a greater performance in Elo and some interesting changes to the development during the training. Out of the NN layout experiments, *smaller_nn* was one of the winners. This model manages to limit its friendly kills to zero and keep them close to zero until the end of the training. Compared to *baseline* it has a similar shape on its plots, but seems to lag behind by around 100 games, finding the same patterns a bit later. Interestingly, this lagging can be found in all four of the plots. This outcome is a bit counter intuitive since a smaller net is expected to learn faster but have a worse maximum potential than a larger one. One explanation for this could be the vast size of the input space in relation to the size of the net. The small net is required to make even rougher generalizations from the input and contain the data in fewer units. The process of how this representation is done most effectively might require a few hundred more training epochs which is the reason why it learns the same patterns a bit slower than the larger nets.

Both the *tiny* and the *tiny_deep* NNs resulted in very weak agents, both according to the Elo rating and the plots. All the plots indicate that these agents have not developed any kind of successful policy. They have short game lengths, little halite gathered and returned, increasing friendly kills and few enemy kills over games as well as a 0% win rate during training versus the training opponent *p_26_15*. The tiny width nets also frequently end up at a constant final halite level at exactly 4000 halite during the first 100 games, indicating that the agents have found a local maximum for the reward function by simply standing still. When looking at the videos

of the agents playing A.1 such as *tiny_nn_100* (trained for 100 games), this is in fact the case and the starting ship is standing still on the starting shipyard during the majority of the game. This behaviour is a result of trying to avoid all negative rewards which occur when the ship is destroyed by an enemy or (more probable) trying to avoid colliding with a friendly ship. This is a consequence of the model not having found any reliable strategies that generates more positive reward than performing no action. With the initial weights of the NN being randomly initialized and hence the behaviour of the model also being random, this is nevertheless a learned behaviour. The game of Halite is a bit special in the sense that consistently performing a random action is a lot worse than performing no action, which is the cause of this cowardly model outcome.

Moving on to the models with more NN units than *baseline*: *wider_nn* and *deeper_nn*. The wider NN was one of the NN changes that outperformed the baseline and its plots indicate that it is capable of avoiding the otherwise characteristic win rate dip and the poor halite collection dip, which are related to each other. Otherwise the plots of both *wider_nn* and *deeper_nn* have similar characteristics to baseline, see figures 4.5 and 4.6. However the *deeper_nn* does like *smaller_NN* not kill as many friendly ships which is something we saw as potential reason for the high Elo rating of *deeper_nn*.

The patterns in the NN layout results have some clear implications. One of them is that a very narrow NN, in our case a width of 10, is incapable of learning an effective strategy. Both the *tiny_nn* and the *tiny_deep_nn* had only dense hidden layers with a width of 10 and both dramatically underperformed in Elo rating and performance seen in the plots. This result is in line with previous research suggesting that a NN that is too narrow runs the risk of not being able to "understand" the problem at hand and also does not benefit from adding additional depth. For more information regarding the universal approximation capabilities of NN see [Johnson, 2019].

Since changing the NN layout has such a great impact on the outcome of the agent, and it is not always better to have large deep NN. We believe that it is recommended to try multiple different settings to see exactly what NN works best for the solution that is needed.

Changing Reward Function

Switching to Sparse Reward after Training As a experiment we also tried to change the reward function of a baseline model already trained 500 games, this was in a effort to be able to save time in training the model. Since the model had already hours put in training we hoped that if we changed the reward function to sparse reward of win condition it could find new strategies to become better than

the baseline model. The results of this can be seen in the plots for switching the reward for baseline to sparse, see figures 4.7, and as can be gathered from the plots we see that the reward function was changed in a way where all the weights of the model had to be adjusted. This resulted in the agent not being able to do anything and would need to learn everything from scratch. From this failed attempt of saving time in training by starting with shaped rewards and going for sparse after some time of training we can say that a rather big adjustment in the reward function is not viable in our setting. This idea was taken from DeepMind Dota 2 paper, they describe their "surgery" method that allows them to change their reward function when a new update is released for the game. In their paper it is also stated that the surgery was not always successful and needed to be very precise. [Berner et al., 2019] From their paper and our failure we think it would be needed much more care to go from our shaped reward to a sparse reward, it could be done over 100 games where the amount shaped is adjusted with the sparse taking its place.

Adding Sparse Reward Since our baseline reward function did not have any input which contained information of the player halite amount we decided that it would be included if a sparse reward signal was to be added. As described in the method we added an additional input for the sparse models. This input contained the current halite standings according to equation 3.2. With this change it would become much easier for the agent to learn if a state near the end of a game was going to end with a winning terminal state or a losing one. As discussed in section 5.2 it is not clear how large the reward of a win would be and where it should be introduced, however with our method as described in the method we tried the solution in multiple settings. Both against *p26* but also against a pool of agents, these results can be seen in figure 4.8 and figure 4.9. An interesting part about *baseline with sparse* is that it does not start killing friendly units in the same way the baseline does. This is not something we expected since a win reward signal does not alter any rewards regarding the kills. However if we look closer at figure 4.8a and figure 4.8b it seems the agent is able to learn collecting and returning halite in the same time frame as baseline, around 150 games. The expected reward of the later stages of the training (around game 400-500) in 4.8e has also risen by about 100 000 compared to 4.1e, from 100 000 to about 200 000 in figure 4.8e, with a spike up to 800 000. This should be largely due to the large sparse reward the agent is given for a win at the end of a game, and after around 200 games the agent wins all games against *p26_15*.

It was thought that the agent would not be able to learn *why* it wins if it wins 100 % of the games since the sparse reward would in this case would always be presented, and only be a constant reward. Because of this the baseline with sparse reward was run against a pool of agents as well, this can be seen in figure 4.9. Here the results became messy, with both bad agents like fully random and good agents like *p26* the results become much harder to gain insight to. The halite gathered in 4.9a could be lower due to the fact that many games as seen in 4.9c are not full 400

frames. The win rate in 4.9d is fluctuating and might be somewhere around 30% which could explain the expected reward being up to 1.4m but generally around 200 000.

Fully Sparse Reward As experiment and to have some results with only sparse reward the baseline was run with only the sparse reward signal as described in the method, see section 3.3. From the plots of the fully sparse training, see figures 4.10, we see that the model does not learn anything and in the plot with collected halite we can see that after 1000 games cargo stops moving and the final results seem to go to 4000 which is the halite from the start when building one shipyard and one ship, see figure 4.10a. We can also notice that the expected reward, see figure 4.10e, moves closer to zero as the model starts to expect zero rewards. These results are most likely consequences of training against a better agent meaning finding the win condition reward is a difficult task. Also we can see that the agent actually does win 2 games in the 1000 played, see figure 4.10d. But these wins do not show up as spikes in the expected reward plot, which leads us to believe they where not presented during training, but the final win frames were removed during the sampling of frames. We train with a sample rate of 10% of the last 100 games, which means that these few frames where the agent actually does win are more likely than not missed. From this we draw our conclusions that if the agent is to be trained on fully sparse reward, the opponent needs to be at the same skill level as the training agent in order to achieve around a 50% win rate during the training. This is something we would believe would happen if the agent trained against itself instead of a fix opponent. Also, a priority filtering of frames might need to be introduced so all terminal states are included. If these frames would be prioritized no win reward signal would be missed and in the case of sparse rewards, these frames are essential for the learning.

Team Spirit and Higher Friendly Kill Penalty Since a clear trend arose when looking at our results from baseline and other methods, see figures 4.2 and 4.3, which was that our agent started killing more and more friendly ships it was discussed why and how we could prevent this trend. Due to our solution doing actions for each ship in the environment separately, it was believed that occasionally the individual ship could gain more cumulative reward taking certain actions, even if it meant colliding on the way there. This could be a byproduct of our agent not getting rewarded for the team effort of all ships but only sees the reward the ship can gain in each decision making. One solution to this was to give a combined reward for all ships in each frame. This would lead to a smaller reward if we have fewer ships and very large rewards if all ships gain rewards together. However this was shown to have a much higher complexity for the agent to solve as seen in figure 4.11. This was likely due to the agent having to solve which actions of many that are good and which are bad. A suggested secondary solution we tried was giving a higher friendly kill penalty to the ships. This was motivated by the fact that if we destroy a friendly ship in the start of a game we not only lose the 500 halite re-

quired to create the ship but also all halite that ship would have been able to gather throughout the game. The first results gathered from this is shown in figure 4.12, and as a quite good presentation over how the reward function can be adjusted we can see in figure 4.12b that the friendly kills instead remain at 0 from 280 games and forward. We also notice however that the agent is slower to learn the collection of halite. In figure 4.1a, showing the baseline halite plot, the baseline is able to gather and collect halite around 150 games and forward but with the higher friendly kills penalty it takes around 250 games before it is able to both collect and return halite to the same extent. This could be a consequence of being punished harder during the initial training, thus making the agent play safer which in turn leads to a slower learning.

Train on enemy data

Halite IV is a competitive game with more than one agent in each game, this was discussed as potential data to learn from, especially since we could play against an opponent which was a contender in the competition held in Halite IV. With this we hoped our agent would have access to more valuable data which it could learn from at the start thus reducing the time for it to find better solutions in the environment. This is in a way behavioral cloning, see [Torabi et al., 2018] for more in depth information regarding this. We start by using the enemy to try and learn a good strategy fast. But over time the agent would try to develop its own strategy. This method was inspired by the concept of behavioral cloning and showed success as we can see from the plots for this agent. If we start by having a closer look at figure 4.13d, we can see that only after approximately 30 games our agent starts winning against *p26_15*. This is something the baseline needs more than 100 games to achieve, however when talking about the time it takes to get these results we also need to factor in how many times the weights are updated in the model. Nevertheless it is clear that the agent is able to see and learn competitive solutions for the environment in fewer episodes. If we look the plot over game length with baseline, see figure 4.1c, we see that only after about 150 games our agent manages to play full 400 frame games. When also training with the enemy data we get more states from each game to evaluate. Also since *p26* follows an algorithm to make its action in each game there are no games where it does not play somewhat well. This means that already from the start of training we will get almost the same amount of states the model will be able to evaluate as baseline gets after playing around 150 games. As such it is not entirely fair to compare the methods directly based on number of played games, however even with this difference there is a clear benefit time-wise since playing the Halite games make up approximately 40% of the training time for our implementation.

Another benefit of training on enemy data is that we saw potential in this particular method achieving results in sparse rewards. It is very hard for the agent to

find the path to a reward far into the future, but since these rewards can be shown through the opponent's moves there could be potentially far less need to explore the environment to find a good strategy. We can also see an indication of this in figure 4.13a. In other methods the agent starts by learning to collect halite and after this is learns to return the cargo. But with the implementation of also learning from the opponent it seems like our agent is able to learn both collecting and returning halite at the same time.

Discount factor, γ

The seemingly minor change of adjusting the discount factor γ had a great impact on the behaviour of the agent. Keep in mind that the baseline has a discount factor of $\gamma = 0.9$. Firstly, looking at the 500 games run for $\gamma = 0.99$, see the first half of the plots in figures 4.14. In the halite plot, see figure 4.14a, we can see that the agent learns to farm and collect reaching a local maximum at around 250 games played, but then declined as mentioned in the results for the plots, see figures 4.14. The outcome after 500 games of play gave rise to the following question: will the agent keep playing worse or is it simply a temporary dip in performance which it will outgrow? Similar to how the *smaller_nn* had a small increase in killing friends which then disappeared after further training, maybe as a result of exploring the option of killing friendly ships only to receive a penalty and through many examples learn to avoid that behaviour. A larger γ is also be expected to require more training to reach its maximum potential since projecting more of the reward further into the future means more complex and time requiring strategies could emerge. As such, the $\gamma = 0.99$ agent was run for an additional 500 games. Unfortunately, this experiment was not very successful in the sense that the performance kept declining and no major improvements were seen.

An interesting insight regarding the $\gamma = 0.99$ model in combination with the shaped reward function is that states early during the game will have a tendency to output higher Q-values (i.e. high expected accumulated reward) due to the fact that the future shaped rewards are very lightly discounted. This is assuming the agent's maximum expected reward tends to be positive. This should not be a problem since the interesting aspect of DQN is which action yields the maximum expected reward given a certain state and time frame, and as such the Q-values differing greatly between time frames is not of interest for the outcome of the model.

The $\gamma = 0.5$ and $\gamma = 0.1$ agents turned out mildly interesting. To get a grasp of the impact of gamma, consider the following calculations: a discount factor of $\gamma = 0.9$ as in the baseline means that a reward is discounted to half around 6.5 turns into the future since $0.9^{6.5} = 0.504$. This allows for incentive to plan ahead since it (and all other rewards within 7 turns) is valued more than half of the immediate reward. However, for $\gamma = 0.5$, 6 turns into the future a reward is only worth 1.5% of

its original and for $\gamma = 0.1$ it is reduced by a factor of 10^{-6} which is completely negligible to the agent compared to the immediate reward. This means that these two agents have a very poor sense for planning and are very greedy, especially $\gamma = 0.1$. This becomes apparent when inspecting the actions taken by the agent during a few games. In appendix A.1 *gamma_01* playing a full exploit game after its 500 games of training (agent *gamma_01*, opponent *p26_15*) highlights the previously described problem . The ships simply randomly move around until they occupy a cell of halite which it farms fully and then proceeds to move randomly again. It is also good at avoiding killing its own ships which is explained by the same logic. The inability to plan ahead shows itself in the halite plot, see figure 4.16a, since returning halite to a shipyard (and not only keeping it as cargo) requires planning several steps ahead. It is also displayed in the video, where a lot of halite remains on the field at the end of the game. These cells are the ones the ships did not encounter by their random moving. The agent ends up with much more non-returned cargo in the final frames compared to other agents. The reason why these two agent are not completely useless in the Elo rating, see table 4.1, is probably due to the fact that almost all dangers in Halite IV can be avoided with a reaction to the immediate state, without planning. Farming halite can be done without planning and while returning it to a shipyard is more difficult it is not impossible and with 20 units and a 400 turn game some of the farmed cargo will be returned as collected halite even with poor planning. In conclusion, using a γ below 0.9 seems like a sub optimal choice for tasks where planning is useful. A too small discount factor removes a part of the beauty of reinforcement learning since the built in planning ahead and the trade off between immediate and future reward is one of the central ideas in RL.

Another interesting aspect of the baseline (with $\gamma = 0.9$) versus $\gamma = 0.99$ models is the question whether $\gamma = 0.99$ will learn to accurately predict its rewards after more training. The observations in the results indicate that $\gamma = 0.99$ does not reach the same levels as the baseline even after 1000 games of play and training. We argue that in order for $\gamma = 0.99$ to converge and reach a high and stable performance level in Halite IV it would require a larger NN as well as additional tweaking of the reward function in order to enable the agent to make more precise decisions. Making more precise decisions would in turn lead to the agent being able to make use of the large future rewards present when $\gamma = 0.99$. We do not believe that simply more time spent training with the current setup would enable the $\gamma = 0.99$ model to reach the same skill level as the baseline.

Exploration vs. exploitation, ε

The agents with changed exploration approach ended up in the top tiers of the Elo ranking, see table 4.1. Compared to the baseline, they both used less exploration than the constant 15% of the baseline model. It was very surprising to see the *eps_zero* agent result in the Elo-wise best of all agents that we constructed,

especially since exploration versus exploitation is very a well researched area. The reason for this is probably due to the complex nature of Halite IV and the fact that the next state in a game arises from a lot of decisions not made by the current agent itself. Exploration is more essential when the environment is determined to a higher degree by the decisions of the agent itself. When the environment has "built in" exploration in the sense that new random states arise from external factors. By instead always following the optimal decision and never being forced to perform a random action, as the *eps_zero* agent does, it is enabled to rely on its strategies more and develop more efficient plays. The amount of units present on a Halite board will result in almost all games resulting in unseen states, which can be seen as a built in exploration. The fact that the *eps_zero* agent manages to limit its friendly kills throughout the training process is probably what makes it skilled at playing Halite. The reason for this low friendly kill count is likely a consequence of the agent never exploring the option of killing friendly ships in the short term in order to earn more reward in the long term. This egoistic strategy is never allowed to be explored by this agent, resulting in a constant low friendly kill count, which is evidently good when comparing the Elo rankings. This also highlights the difficulties of constructing a good shaped reward function that minimizes the number of unwanted side effects.

Nearsightedness

In the baseline model the ships make their decision based on the entire board 21x21 size. The input for each frame is categorized into 8 images, each containing different types of data, which are sent into a convolution layer for feature extraction. This means 3528 numbers are sent in as data for the model to compute. This leads to a relatively big input size which could mean slower convergence, see theory 2.2. As a method to solve this we tried having the model only make the decision based in information of the 11x11 cells around the ship, thus reducing the size of the input to 968 which results in a input size smaller than a third of the baseline method. This could mean that a smaller model is able to learn how to play and that overall it could go much faster finding a decent solution.

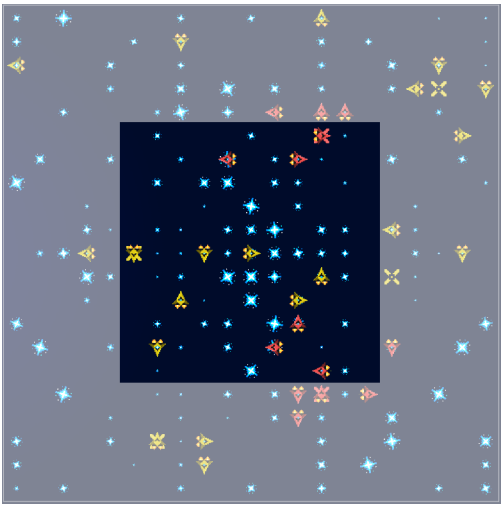


Figure 5.1: The nearsighted agent's field of view visualized

When we compare the results of baseline and the nearsighted agent it is similar in many ways, and there also seems to be some small gain in amount of games before the halite starts rising, see figure 4.20a. However this small difference can not be considered an improvement without more data. What we can say is that the agent is able to learn the environment with smaller amount of data to learn from. This suggest that this type of DQN implementation could be used in situations where the full observation is to large for the computation power available.

Changing training opponent

When training the model against a specific opponent the states are contaminated with data of how the opponent plays. This means that our model might bias its decisions heavily based on how the opponent plays, which might cause it to be over-trained and not play on the same skill level when facing other agents. Also, the agent might not be able to beat better agents if trained on a low skill level opponent since the data it will see will not describe losing states teaching it how to win from a losing position for instance. With this in mind we trained agents against different opponents than the default baseline opponent (*p26_15*). Both changing the training opponent to a single other one, but also a pool of opponent agents might have impact on the final skill level of the agent.

In the experiments we tried having our agent train against 3 other opponents, *p26*, *p4* and against itself. And from these we see that the opponent during training do have a large impact on the performance, it is also noticed that the plots gathered from training these agents can not be compared as easily against the baseline plots.

This due to the fact that if we train against a better opponent there will never be as much reward or halite to gather since the opponent will be better at gathering and deny our agent from doing so. See figures 4.21, 4.22 and 4.23. From training these we can instead see that their skill level that better can be determined from the Elo rating, see table 4.1, improves when playing against better opponents. The agent trained against *p4* ends up in 9th place and have substantial gain against our baseline agents, the agent trained on *p26* without any random moves injected did not have a large impact but did not perform worse than the baseline.

From the training where our agent played against itself we can see the Elo rating going further up to 4th place, see table 4.1, and this is the most distinct run where the plots can be deceiving. Both the expected reward and the actual reward along with the halite gathered during training starts to decline around 150 episodes, see figures 4.23e, 4.23f and 4.23a, and continues to do so until around 300 episodes. This was due to the fact that when our agent starts to play better the opponent would also start playing better, and with this it became a race of farming most halite before everything is gathered. The halite in Halite IV grows with a percentage of what is left in the cell, this means that if the halite is collected and left with close to 0 halite left it will not grow back. And with this it leaves our agent with less reward than if the halite is collected slowly, however this is the only way to gain any rewards since the opponent will collect all available halite from the start. From this we realized that allowing the agent to play against itself allowed for having a good opponent during all 500 games. In this way the opponent will always have a similar skill level to our agent which would allow our agent to always try to improve its strategy.

5.8 Reproducibility

This problem is something that we have thought about and tried to be aware of during training of models and while gathering our results during this project.

> "Reproducibility of modeling is a problem that exists for any machine learning practitioner, whether in industry or academia. The consequences of an irreproducible model can include significant financial costs, lost time, and even loss of personal reputation (if results prove unable to be replicated)" - [Sugimura and Hartl, 2018].

In our work we have found that our methods used in this work have been fairly easy to reproduce, our code has been run on multiple different setups to confirm the results gathered from many of our methods. Our results have been focused on general tendencies and this is what we have been able to reproduce, other than small fluctuations in the overall skill level given from the Elo rating.

5.9 Summary

Our work has proved to be a great insight of where and how RL can be used and where it shows its strengths and weaknesses. To apply DRL on Halite IV with DQN as the main method we have realised the importance of modeling the solution to fit and take advantage of DQN. DQN will predict the expected reward of each action in any given state, during our work we realized that any model that tried to predict all actions for the player (i.e. all the ships at the same time) in each frame would be highly complex and unlikely to be capable to learn in our time frame.

Our DQN solution became capable of finding which action was most likely to achieve high reward for any given ship. Due to the fact that any ship that was targeted for a prediction also could be moved to the center of the environment, allowing our model to see many similar states where friendly ships, enemy ships and halite were spread around the ship where the prediction would be applied. This is what we believe is the single most important part why and how our model is capable of playing the game Halite IV in a competitive level with a relatively short training time, small model and limited computation power, when compared to other RL papers like OpenAI's Dota 2 article [Berner et al., 2019].

Techniques and parameters

Moving on to each of the experiments tried which have allowed us gain a deeper insight in many of the different aspects of both DRL but mostly DQN we start from constructing the model, NN layout. The general insight we found in this section was that the NN is responsible for how much the model is able to take into consideration for each prediction. It was found that in Halite IV with our approach we could achieve high end results by having a rather small NN and it was not necessary to have a big NN which actually could lead to overfitting the NN to each individual state and make training slower. As suggested we realized future implementations of RL can likely benefit from starting with a smaller NN to analyze if it is sufficient for the solution. In the Elo table, see table 4.1, we can also notice that smaller NN and deeper NN have a significant performance increase from baseline. We use the smaller NN as indication that a smaller network can be capable of performing and will hopefully be less prone to overfitting and find more general strategies. In some work the benefits of deeper NN has been studied and proved to be able to approximate functions where more shallow NN needs to be a large magnitude bigger to be able to do the same, for more regarding this see [Telgarsky, 2016]. So the deeper NN might have found a solution more complex than the other NN solutions leading to better performance.

Changing the reward function proved to have high impact and was hard to engineer both for it to avoid introducing bias for any given strategy but also allow the agent to find solutions which lead to rewards both in the short term and the

long term. In the book "Reinforcement Learning: An Introduction" - [Sutton and Barto, 1998], it is stated that the only positive reward an RL agent should gain is if the game is won, to avoid introducing any kind of bias to the environment. However in the case of Halite IV we did not see any success with a fully sparse model, which is why shaped positive rewards are given to any action that can lead to a victory, such as a ship farming or returning halite to a shipyard. The effects of this are hard to prove but our agents do focus on farming and returning halite which might be the best strategy but it could also be that they are not able to find any more offensive strategies due to the bias in the reward function. As previously discussed in the higher friendly kill penalty, it was noticed that many agents after training were likely to disregard the negative reward for killing friendly ships, leading to an increase in the number of friendly ships at the end of the training. As our solution for playing the game was by controlling each ship separately it is likely that the agent is not capable of realizing the total loss of halite which the killed friendly ship would gain during the game. In a solution where prediction for all ships in a frame is done collectively this would probably be an aspect that could be learned, as the end result is impacted a great deal if we are not able to have many ships. But for each individual ship it does not matter to have more ships since the reward is only what it can gain by itself. This is something we find most interesting and shows how each decision in the solution can impact the nature of what problem we are solving. We gain by having a much easier solution for the predictions of expected reward, but as a result the problem moves to being harder to define how to achieve a good end result. By also including the placement in the Elo table, see table 4.1, we can see that the change not only impacted so the agent killed less friendly ships but also gained a substantial performance increase.

Training on enemy data, which means that our agent is allowed to see and learn from the opponent's states and actions. Our findings exploring this possibility shows potential and is something we think would be useful when trying to approach a more sparse reward function. Due to the fact that it is highly unlikely that our agent will ever find a combination of actions which leads to a random victory against a competitive opponent in Halite IV it could gain much by seeing how the opponent was able to defeat the agent. Due to this the sparse reward of winning the game will be shown in all games during training except in the unlikely event of a tie.

The discount factor γ which is the hyper-parameter which adjusts how much impact future expected rewards will have on the prediction. When equal to 1 the model will try to predict all rewards equally, no matter how far into the future, until the terminal state of the episode. It is not clear what it should be set to due to the fact that it is a clear benefit for the agent to be capable of having strategies that lead to a victory late in the game by setting a complex strategy from the start. However the problem becomes more complex and it might not be possible to predict many time frames in the future depending on the problem. In "The Dependence of Effective

Planning Horizon on Model Accuracy" - [Jiang et al., 2015] it is also stated that in many cases it has been proved that a lower discount factor can achieve better end results than a high discount factor [Jiang et al., 2015]. As this is, we realized that each RL problem would need to depend on an individual γ but that it might be ideal to start with a γ near 0.9 and explore the impact it has in the environment presented. In our findings the Elo table, see table 4.1, shows that γ set to 0.9 as in baseline had the highest performance but with 0.5 close behind which could mean there could be a small gain in exploring γ close to 0.9.

Compacting input and near-sightedness are both methods which allows for smaller input data to the model, and in cases where the environment is larger than the computational power is capable of handling could be a necessity for the training to be possible. From our experiments we show that it is possible to achieve learning in Halite IV with smaller input and incomplete information, and if Elo table is included, see table 4.1, both compact and nearsightedness achieved approximately the same rating as our baseline. It is possible to combine these with other of our methods since their skill level is similar to the baseline, but this is for future work.

Changing the rates of ε, exploration vs. exploitation, proved to be an impactful change. Lower rates of exploration seem advantageous for environments such as Halite IV which are complex in nature and where the next state is not only dependent on the action taken by the current unit. Using a greedy approach, where the exploration is zero, to our surprise proved to be the agent with the highest Elo ranking of all our agents.

Choice of opponent during training was tested against two of the competitive agents from the Halite competition and we also tried having the agent play against itself. From training these agents we saw that it had a large impact on how the agent performed in the Elo table, see table 4.1, the agent that trained against p4 was clearly performing better than both the baseline and the agent trained against p26 without epsilon. Also noticeably better was the agent that trained against itself. We believed when training these agents there would be no improvements since our reward system was shaped and it might be hard for the agent to find more strategies than one that achieves high rewards. Also the plots for the agent trained against itself, see figures 4.23, shows that the agent is not able to gather more halite during the game, see figure 4.23a. Since this was the case but the agent still performed better in the Elo rating we watched some of the games and realized that the agent cleared the entire field of halite. Since the halite grows by a percentage of the current halite in the cell each turn it does not recover if farmed empty. This strategy allowed the agent to gain more halite than its opponent fast and after this there was no more halite to be farmed which leading to a victory for the faster agent.

A condensed table version of the final conclusions can be found in table 5.1.

Technique/parameter	Summary
NN layout	Has big impact. Very narrow nets learn nothing. Wider and deeper nets take do not necessarily improve performance.
Changing reward function	Big impact. Needs time and thought to avoid introducing bias. Sparse rewards are very hard to utilize in environments with long time frames.
Train on enemy data	Speeds up initial training drastically. Final outcome is not better than baseline.
Discount factor, γ	For environments that can be navigated with little planning, a large γ can be detrimental. A small γ results in seemingly random behavior except for when finding immediate reward.
Compacting input	Little to no loss in performance when compacting input in a logical way. No increase either however.
Exploration vs. exploitation, ε	Low or decaying exploration rate in environments with built in randomness can be very beneficial.
Near-sightedness	Limiting input information to the most relevant gives no loss in performance.
Change of opponent during training	Big impact. Training against self proved to be the best alternative, even with shaped reward function.

Table 5.1: Final summary of conclusions for the eight techniques and parameters tested.

5.10 Final thoughts

This project has been an overall success. The goal has been met by thorough and strict analysis, isolating the impact of each of the techniques and parameters in table 1.1. With close communications with our supervisor at Sinch, the company has gotten insights in the value, strengths and weaknesses of RL solutions.

At a glance, Halite IV looks like a perfect environment for being dominated by a reinforcement learning agent because of its straightforward rules, small action space for each ship, discrete time frames and the fact that it is designed to be played by computers from the start. However, because of its simplicity, the number of valid strategies are tremendous and therefore makes the game complex. The resulting size of the state space in combination with an episode length of 400 discrete frames results in a problem that is incredibly difficult to solve with RL. However, our study proves that it is possible with limited computational power to obtain a DQN solution that is competitive against algorithmic solutions.

Since the field of RL is huge our work has been limited to DRL with DQN as the main method for gathering results. However even with these limitations DQN has many parts which can be adjusted and our methods do not include all of the parts that can be adjusted, and not all possible variations. Our work has also been done solely with Halite IV as an environment to gather results. With these limitations we were able to go more in depth and analyze our results, but our results will not be general for other environments or solutions. Our results are to be used as guidelines why and when the discussed methods could have potential impacts.

If the project would have continued for three more months we would have used our insights to build solutions for other environments than Halite IV. This continuation would show us which of the methods are more valuable for a general RL solution and which are specific for Halite IV.

References

Berner, C., G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang (2019). "Dota 2 with large scale deep reinforcement learning". *CoRR* **abs/1912.06680**. arXiv: 1912.06680. URL: http://arxiv.org/abs/1912.06680.

Bernstein, A. and E. Burnaev (2018). "Reinforcement learning in computer vision". In: p. 58. DOI: 10.1117/12.2309945.

Brockman, G., V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba (2016). *Openai gym*. eprint: arXiv:1606.01540.

Busoniu, L., R. Babuska, and B. De Schutter (2010). "Multi-agent reinforcement learning: an overview". *Studies in Computational Intelligence* **310**, pp. 183–221. DOI: 10.1007/978-3-642-14435-6_7.

Chollet, F. et al. (2015). *Keras*. https://keras.io.

Elo, A. E. (1978). *The Rating of Chessplayers, Past and Present*. Arco Pub., New York. ISBN: 0668047216 9780668047210.

Geist, M. and B. Scherrer (2013). "Off-policy learning with eligibility traces: A survey". *CoRR* **abs/1304.3999**. arXiv: 1304.3999. URL: http://arxiv.org/abs/1304.3999.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Hornik, K., M. Stinchcombe, and H. White (1989). "Multilayer feedforward networks are universal approximators". *Neural Networks* **2**:5, pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

Jiang, N., A. Kulesza, S. Singh, and R. L. Lewis (2015). "The dependence of effective planning horizon on model accuracy".

Johnson, J. (2019). "Deep, skinny neural networks are not universal approximators". URL: https://arxiv.org/pdf/1810.00393.pdf.

Kaggle (2020). *Halite by two sigma*. Ed. by Kaggle. [Accessed: 2021-03-02]. URL: https://www.kaggle.com/c/halite/overview/halite-rules.

Kober, J., J. Bagnell, and J. Peters (2013). "Reinforcement learning in robotics: a survey". *The International Journal of Robotics Research* **32**, pp. 1238–1274. DOI: 10.1177/0278364913495721.

Latif, A., A. Rasheed, U. Sajid, J. Ahmed, N. Ali, N. I. Ratyal, B. Zafar, and S. H. D. M. S. T. Khalil (2019). "Content-based image retrieval and feature extraction: a comprehensive review". DOI: 10.1155/2019/9658350.

Luong, N. C., D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim (2018). *Applications of deep reinforcement learning in communications and networking: a survey*. arXiv: 1810.07862 [cs.NI].

Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller (2013). "Playing atari with deep reinforcement learning". *CoRR* **abs/1312.5602**. arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

Nguyen, T. T., N. D. Nguyen, and S. Nahavandi (2018). "Deep reinforcement learning for multi-agent systems: A review of challenges, solutions and applications". *CoRR* **abs/1812.11794**. arXiv: 1812.11794. URL: http://arxiv.org/abs/1812.11794.

Shalev-Shwartz, S., S. Shammah, and A. Shashua (2016). "Safe, multi-agent, reinforcement learning for autonomous driving".

Shao, K., Z. Tang, Y. Zhu, N. Li, and D. Zhao (2019). "A survey of deep reinforcement learning in video games". *CoRR* **abs/1912.10944**. arXiv: 1912.10944. URL: http://arxiv.org/abs/1912.10944.

Si, J., A. Barto, W. Powell, and D. Wunsch (2009). "Reinforcement learning and its relationship to supervised learning", pp. 45–63. DOI: 10.1109/9780470544785.ch2.

Silver, D., A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis (2016). "Mastering the game of go with deep neural networks and tree search". *Nature* **529**, pp. 484–489. DOI: 10.1038/nature16961.

Silver, D., J. Schrittwieser, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Driessche, T. Graepel, and D. Hassabis (2017). "Mastering the game of go without human knowledge". *Nature* **550**, pp. 354–359. DOI: 10.1038/nature24270.

Sugimura, P. and F. Hartl (2018). "Building a reproducible machine learning pipeline". *CoRR* **abs/1810.04570**. arXiv: 1810.04570. URL: http://arxiv.org/abs/1810.04570.

Sutton, R. and A. Barto (1998). "Reinforcement learning: an introduction". *IEEE Transactions on Neural Networks* **9**:5. DOI: 10.1109/TNN.1998.712192.

Telgarsky, M. (2016). "Benefits of depth in neural networks". *CoRR* **abs/1602.04485**. arXiv: 1602.04485. URL: http://arxiv.org/abs/1602.04485.

Torabi, F., G. Warnell, and P. Stone (2018). "Behavioral cloning from observation". *CoRR* **abs/1805.01954**. arXiv: 1805.01954. URL: http://arxiv.org/abs/1805.01954.

Watkins, C. J. and P. Dayan (1992). URL: https://www.gatsby.ucl.ac.uk/~dayan/papers/cjch.pdf.

A

Supplementary material

A.1 Videos

Agent	Opponent	Games trained	Link
baseline	p26_15	0	https://youtu.be/2ace3bYcbcU
baseline	p26_15	500	https://youtu.be/FQ7Tm2EH4tA
tiny_nn	p26_15	200	https://youtu.be/RM0kV_ZVv_c
baseline_self	baseline_p26	500	https://youtu.be/CIPkqd5IrRM
gamma01	p26_15	500	https://youtu.be/YWjj9PLCUpE

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
June 2021

Document Number
TFRT-6127

Author(s)
Kim Haapamäki
Jesper Laurell

Supervisor
Johan Eilert, Sinch AB (publ), Sweden
Johan Grönqvist, Dept. of Automatic Control, Lund University, Sweden
Anders Ranzter, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle
Playing Halite IV with Deep Reinforcement Learning

Abstract

Playing games with reinforcement learning has for years been a target for research and has seen incredible breakthroughs in recent years. Reinforcement learning is a type of machine learning, which can be combined with the concept of deep learning, resulting in what is called deep reinforcement learning. The promise of deep reinforcement learning attracts businesses that aim to get an edge over traditional algorithmic methods. Our work focused on exploring the aspects of deep reinforcement learning with a DQN implementation and the game Halite IV as the environment. We created DQN agents capable of outperforming competitive solutions and tested and evaluated techniques for enhancing the DQN solution. The most insightful results include: individual decision making for a team based environment can simplify the DQN setup drastically, reward function engineering for RL is critical and a sparse reward is not practical for long time frames, self play is advantageous compared to a static opponent and low rates of exploration is beneficial in environments with built in randomness.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-101

Recipient's notes

Security classification

http://www.control.lth.se/publications/