Rainbow for Abalone

Faraz Maschhur [*1] **Max Reinhard** [*1]

Abstract

Recent research in the field of *Reinforcement Learning* (RL) has led to agents with impressive performance on games and other tasks. One of the biggest breakthroughs in the field of RL is the development of *Deep Q-Networks* (DQN), combining *Convolutional Neural Networks* (CNNs) and *Q-Learning*. The *Rainbow* algorithm integrates DQN and many successful RL techniques and has been used to achieve super-human level of play in Atari games. In order to analyze the abilities of the *Rainbow* algorithm in a task bearing contrasting properties to the Atari games setting – regarding the state- and action-space complexity – and since there is no recent research on the game, we decided to apply *Rainbow* to the strategic board game Abalone. The results of the experiments with the fully trained agents do not provide a basis for comparison in terms of performance, but instead show their inability to master the game. This opens a discussion about the ability of the agents to explore the action-space and learn complex strategies for Abalone.

1. Introduction

The increasing interest in the field of RL during the last years has led to the creation of sophisticated RL agents capable of solving more and more complex tasks. Applying RL agents to complex strategic games presents a metric about the capabilities of the agents during research. One of the most successful appliances of RL agents to strategic board games is *AlphaZero* (Silver et al., 2018) and its variants. In contrast, DQN (Mnih et al., 2013) (Mnih et al., 2015) and *Rainbow* (Hessel et al., 2017) have achieved super-human level of play in Atari games. While Atari games offer a broad state-space and a narrow action-space, complex strategic board games usually feature a more narrow state-space and a broad action-space. In order to analyze the abilities of the *Rainbow* algorithm in a task bearing contrasting properties to the usual Atari games setting regarding the state- and action-space complexity and since there is no recent research on the game, we decided to apply *Rainbow* to the strategic board game Abalone[1]. Understanding the limitations of RL agents contributes to the development of more sophisticated algorithms. This application of *Rainbow* entails potential challenges. Due to the bigger action-space of Abalone, a well-considered choice of an efficient exploration strategy is important. Comparable algorithms, such as *AlphaZero*, use some form of look-ahead functionality, e.g. *Monte-Carlo Tree Search* algorithms (Browne et al., 2012). However, *Rainbow* originally does not use such methods but utilizes *NoisyNets* (Fortunato et al., 2017) instead.

Section 2 provides an overview of the game and the field of RL. In Section 3 we describe our adaptation of *Rainbow* and the game environment. Section 4 explains the experimental setup, presents the empirical results and discusses them. Finally Section 5 concludes our work and gives a look into possible future extensions.

2. Related Work

Abalone is a two-player game, where each player commands either white or black marbles. The game board is a hexagonal shape with 61 evenly distributed recesses (fields) in which the marbles can be placed. Starting on different sides of the board, players move their marbles trying to push out six enemy marbles to win the game. Allowed actions include moving up to 3 marbles in a row at once into any direction. Figure 1 depicts a virtual version of the Abalone game board.

The game of Abalone presents an interesting algorithmic challenge. The complexity of the game is comparable to that of Shogi and Xiangqi (Lemmens, 2005) and therefore higher than the well-explored game of Chess.

The only application of RL methods to Abalone is the use of *TD-Learning* in earlier times of the research field. (Campos & Langlois, 2009)

One of the biggest breakthroughs in the field of RL is the de-

[*]Equal contribution [1]Technical University of Berlin, Germany. Correspondence to: Faraz Maschhur <f.maschhur@campus.tu-berlin.de>, Max Reinhard <max.reinhard@campus.tu-berlin.de>.

Neural Information Processing Projects, TU Berlin, Summer Term 2021.

[1]https://en.wikipedia.org/wiki/Abalone_(board_game)

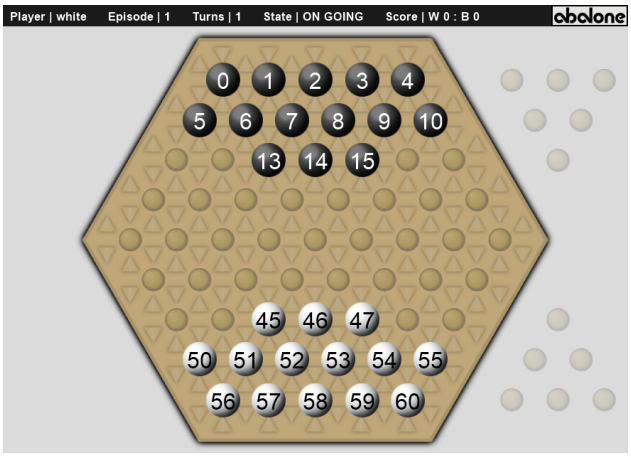

Figure 1. Screenshot of a virtual Abalone game board

velopment of DQNs, which led to super-human level of play in many complex Atari games combining *Convolutional Neural Networks* (CNNs) and *Q-Learning* (a special form of *TD-Learning*) for the first time. The research on DQN built the foundation for modern model-free RL algorithms capable of solving more complex tasks.

Despite its success, there are conceptual problems to DQN, for which according solutions have been proposed. For example, the combination of DQN with *Double Q Learning* (van Hasselt et al., 2015) solves the overestimation bias natural to *Q-Learning* methods and the introduction of *NoisyNets* allows for a more efficient exploration than ϵ-*greedy* in some cases.

Combining the aforementioned two and many other improvements to the initial DQN approach, the *Rainbow* algorithm proposes a sophisticated RL approach to Atari games, which outperforms DQN and every single one of the improvements used by *Rainbow*.

The most successful applications of RL on board games started with the development of *AlphaGo* (Silver et al., 2016), an agent which achieved super-human play in the game of Go. It was trained upon many recordings of professional rounds of Go using knowledge of the games rules and expert domain knowledge. Its successor *AlphaGo Zero* (Silver et al., 2017) learned to play the game of Go and outperform *AlphaGo* only through self-play and by prior knowledge of the games rules. *AlphaGo Zero* has been used to develop *AlphaZero*, which reached super-human level of play in the games Go, Chess and Shogi. Similar to *AlphaGo Zero*, *AlphaZero* knew the rules of the games and learned only through self-play. The most recent development in this particular area of RL agents is *MuZero* (Schrittwieser et al., 2020), an agent capable of playing Go, Chess, Shogi and Atari games on super-human level of play without prior knowledge of the games rules only through self-play.

The use of modern RL approaches is not restricted to board games. Other applications of *AlphaZero* have shown the importance behind the development and evaluation of these methods. (Segler et al., 2018) (Dalgaard et al., 2020)

3. Environment and Agent

Our system consists of two main components: an Abalone game environment and a Rainbow-based reinforcement learning agent which interacts with this environment. For both components we build upon existing implementations that we modified and extended according to our needs.

3.1. Abalone Environment

Since the need for providing a suitable environment is universal to all RL problems, there exist different toolkits which provide a standardized interface to such environments. To our knowledge, among those, the *Open AI Gym* (Brockman et al., 2016) and *PettingZoo* (Terry et al., 2020) are particularly popular and functionally extensive.

As the implementation of the game logic and the choice of representation for e.g. game states and moves itself is quite cumbersome, we build upon an existing environment. We decided on the *gym-abalone* environment (towzeur, 2020), which is publicly available and provides and implementation of the Abalone game based on *Open AI Gym*. This enabled us to leverage other available resources for this toolkit, especially existing implementations of the *Rainbow* algorithm (see section 3.2), which are designed to interface *Open AI Gym* environments.

3.1.1. State Representation

To represent the Abalone board with a quadratic array, an orthogonal basis with length of 11 in both dimensions was chosen in *gym-abalone*. This results in a state representation $s \in S = \{-2, -1, 0, 1\}^{11 \times 11}$, which is implemented as an array. At indices which do not correspond to an actual board field the array has the value of -2. White player marbles are represented by the value 0, black player marbles are represented by the value 1 and empty field are represented by the value -1.

3.1.2. Action Space

An action is encoded as a movement from position $p_1 \in P$ to $p_2 \in P$ with $P = \{0, ..., 60\}$. This results in an action space of $A = P \times P$. It is important to note here that not all of these actions correspond to actual marble movements from p_1 to p_2 as the game only allows certain movements. Among those are simultaneous movements of two or three marbles, which are also encoded in A. For more information about the rules of the game, please refer to the Abalone rule

book.[2] The size of the action space is $61 \cdot 61 = 3721$, while the actual number of technically possible moves is only 1356.[3] A typical Abalone game state actually only allows for about 60 moves (Aichholzer et al.). To be able to determine between valid and invalid moves, the environment provides a functionality to mask invalid moves so that the agent may only select among the valid ones.

3.1.3. Rewards

The *gym-abalone* environment originally implements the following rewards: $+12$ for winning, $+2$ for ejecting an enemy marble, $+0.5$ for pushing an enemy marble (without ejecting it) and -0.1 for moves which do not interact with the enemy. We found those rewards to be heavily biased towards aggressive play and decided against them, although the approach to deincentivize extremely defensive play may enable the agent to learn game-winning policies faster. We tested using only natural rewards, i.e. $+1$ for winning and -1 for loosing, but found that the time needed for training the agent became impractical with regards to our resource constraints. As a middle ground we decided on using the following rewards: $+6$ / -6 for winning / loosing and $+1$ / -1 for ejecting an enemy marble / loosing a marble. The motivation behind the symmetric negative rewards is to increase the amount of information provided by an rewarded experience. Like the rewarded ejections, this should reduce the time needed to learn game-winning policies. We are aware that these rewards also introduce a bias, but bias-free rewards seemed unrealistic due to the mentioned constraints.

3.1.4. Maximum game length

Another modification we made to the *gym-abalone* environment is increasing the maximum number of turns possible in one game. Originally the environment is configured to use a maximum number of turns of 200. An average game of Abalone played by humans takes 87 turns (Chorus, 2009). While testing different game lengths, we found that this number is insufficient for the agent to frequently experience the reward of actually winning the game. The same still holds for a game length of 200, which is why we chose to increase it to 400. We understand that increasing the maximum game length entails the risk of the agent being in situations in which it is not able to drive the game further, therefore loosing valuable training time by generating only experiences with little information. This is why we did not increase the maximum game length any further.

[2] https://cdn.1j1ju.com/medias/c2/b0/3a-abalone-rulebook.pdf

[3] This can be calculated by counting the technically possible moves from each position, starting in the middle and moving outward layer-by-layer: $(1+6)*36+12*29+18*22+24*15 = 1356$

3.2. RL Agent

An RL agent interacts with the provided environment and by selection an action in the action space (legal moves). This action leads to a change in the environment state (game board) and the agent can observe this state transformation. Based on the information provided by these transitions the agent may learn a action-selection policy to maximize the accumulated rewards it receives.

3.2.1. Adaptations to Rainbow

We choose the Rainbow algorithm mentioned in section 2 for our RL agent. Our implementation is heavily based on the *Rainbow is all you need!* tutorial [4], which is publicly available. While the original algorithm used CNNs to process the raw pixel information from Atari games, we omitted those layers since the Abalone environment provides discrete state information. The two-dimensional state representation provided by the *gym-abalone* environment is flattened into a one-dimensional array of size $11 \cdot 11 = 121$. This state representation is then extended to a three dimensional array of size 3×121, where the first layer encodes the positions of the white marbles, the second layer encodes the positions of black marbles and the third layer encode which agents turn it is. This encoding is adapted from the state representation used in *AlphaZero*.

All hyperparameters are adopted from the original paper if not stated otherwise here. For hyperparameters which specify a number of Atari environment frames, we had to choose suitable equivalents, since the Abalone environment is turn-based instead. Additionally we lowered the ϵ decay in the case of ϵ-greedy exploration, to account for the increased exploration need we assumed. Training takes place after each step as soon as the warm-up period has ended. The DQN target network is hard-updated in a certain interval, i.e. target network updating does not use *Polyak Averaging* (Polyak & Juditsky, 1992) (which variant is used originally is not specified). The specific values of all modified hyperparameters can be found in table 1

hyperparameter	value
epsilon decay	$1/100000$
warm-up period	2000
training interval	1
target network update interval	400
memory size	50000

Table 1. General hyperparameters of our RL agent

Our implementation is made publicly available.[5]

[4] https://github.com/Curt-Park/rainbow-is-all-you-need

[5] https://github.com/wuxmax/rl4abalone

4. Experiments

We conducted a number of experiments to assess the performance of our trained RL agents in the game of Abalone.

4.1. Setup

In order to compare the success of different RL agents the use of game engines, such as *Stockfish* (Stockfish) for chess, has become common practice. However, in the case of Abalone the available game engines have not been updated in a long time and offer little practicability. Therefore we decided to benchmark our trained agents against each other and against earlier versions of themselves.

During the benchmarks we track and return the number of turns made by each agent, the winner of each match, the reached scores of both agents, the number of ejected per agent marbles and a ratio specifying the number of unique turns per game per agent. The score an agent reaches is the sum of positive and negative rewards received in a single game and the unique turn ratio is the number of unique moves divided by the number of all moves made by the agent in a single game.

The agents used in the benchmark result from four different training setups. Two of the setups train *NoisyNet* agents, while the other two train ϵ-*greedy* agents. In each case we train the agents for 1M steps with a maximum game length of 400 turns. Every 100k steps during the training we save a version of the current agent for benchmarking purposes. Table 2 shows the hyperparameters specific to the training of *NoisyNet* agents, whereas Table 3 shows hyperparameters specific to the training of ϵ-*greedy* agents. Other hyperparameters are shown in Table 1.

hyperparameter	*NoisyNet 0.5*	*NoisyNet 5.0*
Noise σ init	0.5	5.0

Table 2. NoisyNet specific hyperparameters

hyperparameter	*EpsGreedy 100k*	*EpsGreedy 500k*
epsilon decay	$1/100000$	$1/500000$
initial epsilon	1.0	1.0
minimum epsilon	0.1	0.1

Table 3. ϵ-*greedy* specific hyperparameters

Using this procedure we end up with 10 agents for each of the four training setups. To measure the progress of the training algorithm we benchmark each earlier agent against the next version of itself in their own training setup. To determine the most successful architecture we also benchmark the latest agents of the four different training setups against each other when all four training setups have produced a new agent each.

Each time two agents play against each other, they play a single round only, as there is no randomness during benchmarking and each game between the same two agents always concludes the same way.

4.2. Results

Exemplary results for the iterative benchmark (agent competes against its previous version) of the *NoisyNet 0.5* agent can be found in Table 4. Results of the benchmark of the different version of this agent competing against the corresponding version of the *EpsGreedy 500k* agent are shown in Table 5. All other result tables are located in Appendix A to maintain better readability.

The column names describe the following metrics:

- *S(w)* = Score of white agent
- *S(b)* = Score of black agent
- *#T* = Number of turns in the game
- *S/#T(w)* = Score of white agent divided by number of turns of white agent
- *S/#T(b)* = Score of black agent divided by number of turns of black agent
- *#UT/#T(w)* = Number of unique turns of white agent divided by number of turns of white agent
- *#UT/#T(b)* = Number of unique turns of black agent divided by number of turns of black agent
- *#E(w)* = Number of ejects of white agent
- *#E(b)* = Number of ejects of black agent

The results of the experiments point towards certain problems of our implementation. Not a single game ended by natural means, they all got terminated after the maximum game length of 400 turns. Most games do not contain any eject moves and therefore result in draws. Furthermore, the unique turn ratio lies between 0.01 and 0.3 (i.e. 2 - 60 unique turns per game).

Further inspection of the results shows that no clear subdivision of agents by performance is possible. At first sight, the *NoisyNet* agents seem to outperform the *EpsGreedy* agents, however, given the vast problems described earlier, the data is not suitable to make general statements about the performance of the different architectures.

On further note: comparing the agents performances during training and testing we have found that their performance during training is better than during testing. Games do end in draws less frequently and sometimes are even finished according to the rules (by ejecting six marbles from one player, which results in a score of 11). Also the unique turn ratio of the agents was higher during testing, reaching up

White agent	Black agent	Winner	*S(w)*	*S(b)*	*#T*	*#UT/#T(w)*	*#UT/#T(b)*	*#E(w)*	*#E(b)*
NoisyNet 0.5 [100k]	*NoisyNet 0.5* [200k]	draw	0	0	400	0.035	0.050	0	0
NoisyNet 0.5 [200k]	*NoisyNet 0.5* [300k]	draw	0	0	400	0.115	0.105	0	0
NoisyNet 0.5 [300k]	*NoisyNet 0.5* [400k]	black	-1	1	400	0.120	0.200	0	1
NoisyNet 0.5 [400k]	*NoisyNet 0.5* [500k]	draw	0	0	400	0.075	0.085	0	0
NoisyNet 0.5 [500k]	*NoisyNet 0.5* [600k]	draw	0	0	400	0.040	0.065	0	0
NoisyNet 0.5 [600k]	*NoisyNet 0.5* [700k]	draw	0	0	400	0.040	0.010	0	0
NoisyNet 0.5 [700k]	*NoisyNet 0.5* [800k]	black	-1	1	400	0.095	0.070	0	1
NoisyNet 0.5 [800k]	*NoisyNet 0.5* [900k]	draw	0	0	400	0.085	0.100	0	0
NoisyNet 0.5 [900k]	*NoisyNet 0.5* [1M]	draw	0	0	400	0.105	0.025	0	0
NoisyNet 0.5 [200k]	*NoisyNet 0.5* [100k]	draw	0	0	400	0.100	0.170	0	0
NoisyNet 0.5 [300k]	*NoisyNet 0.5* [200k]	draw	0	0	400	0.105	0.080	0	0
NoisyNet 0.5 [400k]	*NoisyNet 0.5* [300k]	draw	0	0	400	0.125	0.100	0	0
NoisyNet 0.5 [500k]	*NoisyNet 0.5* [400k]	draw	0	0	400	0.040	0.150	1	1
NoisyNet 0.5 [600k]	*NoisyNet 0.5* [500k]	draw	0	0	400	0.040	0.080	0	0
NoisyNet 0.5 [700k]	*NoisyNet 0.5* [600k]	draw	0	0	400	0.095	0.065	0	0
NoisyNet 0.5 [800k]	*NoisyNet 0.5* [700k]	draw	0	0	400	0.065	0.010	0	0
NoisyNet 0.5 [900k]	*NoisyNet 0.5* [800k]	black	-1	1	400	0.115	0.135	1	2
NoisyNet 0.5 [1M]	*NoisyNet 0.5* [900k]	draw	0	0	400	0.110	0.120	0	0

Table 4. Results for *NoisyNet 0.5* agents

White agent	Black agent	Winner	*S(w)*	*S(b)*	*#T*	*#UT/#T(w)*	*#UT/#T(b)*	*#E(w)*	*#E(b)*
NoisyNet 0.5 [100k]	*EpsGreedy 500k* [100k]	draw	0	0	400	0.055	0.060	0	0
NoisyNet 0.5 [200k]	*EpsGreedy 500k* [200k]	draw	0	0	400	0.055	0.120	0	0
NoisyNet 0.5 [300k]	*EpsGreedy 500k* [300k]	draw	0	0	400	0.070	0.095	0	0
NoisyNet 0.5 [400k]	*EpsGreedy 500k* [400k]	draw	0	0	400	0.060	0.035	0	0
NoisyNet 0.5 [500k]	*EpsGreedy 500k* [500k]	draw	0	0	400	0.055	0.170	0	0
NoisyNet 0.5 [600k]	*EpsGreedy 500k* [600k]	draw	0	0	400	0.040	0.075	0	0
NoisyNet 0.5 [700k]	*EpsGreedy 500k* [700k]	draw	0	0	400	0.080	0.050	0	0
NoisyNet 0.5 [800k]	*EpsGreedy 500k* [800k]	draw	0	0	400	0.065	0.045	0	0
NoisyNet 0.5 [900k]	*EpsGreedy 500k* [900k]	white	1	-1	400	0.130	0.135	1	0
NoisyNet 0.5 [1M]	*EpsGreedy 500k* [999k]	draw	0	0	400	0.080	0.050	0	0
EpsGreedy 500k [100k]	*NoisyNet 0.5* [100k]	draw	0	0	400	0.100	0.095	0	0
EpsGreedy 500k [200k]	*NoisyNet 0.5* [200k]	draw	0	0	400	0.195	0.120	0	0
EpsGreedy 500k [300k]	*NoisyNet 0.5* [300k]	draw	0	0	400	0.125	0.130	0	0
EpsGreedy 500k [400k]	*NoisyNet 0.5* [400k]	draw	0	0	400	0.155	0.085	0	0
EpsGreedy 500k [500k]	*NoisyNet 0.5* [500k]	draw	0	0	400	0.110	0.085	0	0
EpsGreedy 500k [600k]	*NoisyNet 0.5* [600k]	draw	0	0	400	0.065	0.065	0	0
EpsGreedy 500k [700k]	*NoisyNet 0.5* [700k]	draw	0	0	400	0.035	0.010	0	0
EpsGreedy 500k [800k]	*NoisyNet 0.5* [800k]	black	-2	2	400	0.250	0.145	1	3
EpsGreedy 500k [900k]	*NoisyNet 0.5* [900k]	draw	0	0	400	0.075	0.080	0	0
EpsGreedy 500k [999k]	*NoisyNet 0.5* [1M]	draw	0	0	400	0.060	0.025	0	0

Table 5. Results for *NoisyNet 0.5* agents vs. *EpsGreedy 500k* agents

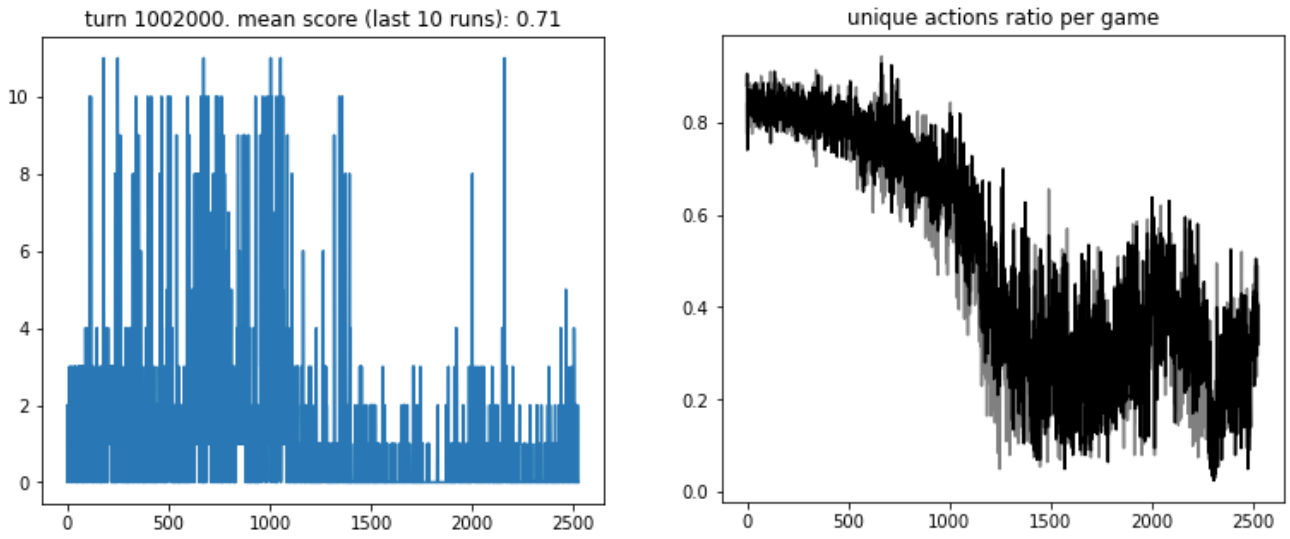

Figure 2. Training metrics of the *EpsGreedy 500k* agent

to 0.9 during training, but plummeted during testing of the same agent. Figure 2 shows two graphs, which visualize the score of the winning agent and the unique turn ratio per game over the course of the training of the *EpsGreedy 500k* agent.

4.3. Discussion

It is clear, that the agents do not achieve super-human or even human-like performance. The results of the experiments hint to an exploration problem, since the agents seem to be stuck in local optima.

The low number of unique turns suggests that the agents execute turns which they repeat over and over again until the environment shuts down the game by force. Stalling the game this way seems more lucrative to the agents than engaging in interaction with the enemy marbles, which results in low eject move counts.

A missing look ahead functionality and the design of the action space in the environment may amplify the tendencies of the agent to stay in local optima. Most of the actions in the action space are actually illegal (only approx. 60 of 3721 are legal). This makes it presumably harder to learn, because in most cases the highest valued action is not the one that is actually executed (we observed this specifically).

The difference in performance between training and testing could be a result of the low training interval and continuous learning of the agents. The permanently changing state-action value attribution may induce a more variational behavior.

5. Conclusion and Future Work

In this work we modified the *Rainbow* algorithm and exposed it to the domain of board games by applying it to the abstract strategy board game Abalone.

One main finding is that agents tend to get stuck in local optima and repeatedly stall the game instead of interacting with enemy marbles. This issue asks for more thorough approaches regarding the trade-off between exploitation and exploration. Look-ahead funtionalities, such as *Monte-Carlo Tree Search* algorithms, could help achieve a better exploration policy.

Additionally the size of the action space defined by *gym-abalone* is tremendous and not easily approximated. Since many of the technically possible actions are impossible or illegal, the algorithm may benefit from small negative rewards for illegal moves instead of masking them away completely.

Another reason for the agents getting stuck in stalling behavior could be the design of rewards. A small reward for moves without enemy interactions (as it is originally implemented in the *gym-abalone* environment) is one modification to consider. Furthermore this problem could be mitigated by decreasing the negative reward for loosing a marble, because it may inhibit the agents to change their behavior. It is important to note here that all modified reward functions (those which are not the natural reward) do induce some form of bias to the policies learned by the agents.

The performance of the agents could benefit from more general extensions of the exploration procedure. Curiosity-driven methods (Pathak et al., 2017) could lead to a more

coordinated exploration of the action-space.[6]

Even though the environment does not work with image input in the form of raw pixels, learning from the array representation of the game board may be accelerated by the use of CNNs. A desirable trait of CNNs is the ability to learn a geometrical understanding of the input. This could possibly be leveraged by keeping the two-dimension shape of the array and processing it with CNNs instead of directly trying to learn from the discrete space representation.

Regarding the specific configurations of the algorithm and the specific design of game environment presented in this article, the *Rainbow* algorithm does not seem to perform well for the game of Abalone. Applying *AlphaZero* (or the most recent successor *MuZero*) presents a different, but promising RL approach to solving Abalone. Both algorithms have been shown to yield outstanding results when applied to board games.

References

Aichholzer, O., Aurenhammer, F., and Werner, T. Algorithmic fun-abalone.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi: 10.1109/TCIAIG.2012.2186810.

Campos, P. and Langlois, T. Abalearn: Ecient self-play learning of the game abalone. 01 2009.

Chorus, P. *Implementing a computer player for abalone using alpha-beta and monte-carlo search*. PhD thesis, Citeseer, 2009.

Dalgaard, M., Motzoi, F., Sørensen, J. J., and Sherson, J. Global optimization of quantum dynamics with AlphaZero deep exploration. *npj Quantum Information*, 6(1), January 2020. doi: 10.1038/s41534-019-0241-0. URL https://doi.org/10.1038/s41534-019-0241-0.

[6]We tried implementing a simple curiosity-driven exploration approach in the form of high reward for unseen states which decreases over a certain number of turns, but to no avail. Finalizing this approach would have required a more detailed examination of possible solutions using curiosity, which was possible due to time constraints.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. Noisy networks for exploration. *CoRR*, abs/1706.10295, 2017. URL http://arxiv.org/abs/1706.10295.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. URL http://arxiv.org/abs/1710.02298.

Lemmens, N. Constructing an abalone game-playing agent. 2005.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning, 2013.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015. ISSN 1476-4687. doi: 10.1038/nature14236. URL https://doi.org/10.1038/nature14236.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. Curiosity-driven exploration by self-supervised prediction. In *ICML*, 2017.

Polyak, B. and Juditsky, A. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30:838–855, 07 1992. doi: 10.1137/0330046.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., and Silver, D. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, December 2020. doi: 10.1038/s41586-020-03051-4. URL https://doi.org/10.1038/s41586-020-03051-4.

Segler, M. H. S., Preuss, M., and Waller, M. P. Planning chemical syntheses with deep neural networks and symbolic AI. *Nature*, 555(7698):604–610, March 2018. doi: 10.1038/nature25978. URL https://doi.org/10.1038/nature25978.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap,

T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan 2016. ISSN 1476-4687. doi: 10.1038/nature16961. URL https://doi.org/10.1038/nature16961.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct 2017. ISSN 1476-4687. doi: 10.1038/nature24270. URL https://doi.org/10.1038/nature24270.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. ISSN 0036-8075. doi: 10.1126/science.aar6404. URL https://science.sciencemag.org/content/362/6419/1140.

Stockfish. Strong open source chess engine. https://github.com/official-stockfish/Stockfish.

Terry, J. K., Black, B., Hari, A., Santos, L., Dieffendahl, C., Williams, N. L., Lokesh, Y., Horsch, C., and Ravi, P. Pettingzoo: Gym for multi-agent reinforcement learning. *CoRR*, abs/2009.14471, 2020. URL https://arxiv.org/abs/2009.14471.

towzeur. Abalone environments for openai gym. https://github.com/towzeur/gym-abalone, 2020.

van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning, 2015.

A. Additional result tables

See next pages.

White agent	Black agent	Winner	*S(w)*	*S(b)*	*#T*	*#UT/#T(w)*	*#UT/#T(b)*	*#E(w)*	*#E(b)*
NoisyNet 5.0 [100k]	*NoisyNet 5.0* [200k]	draw	0	0	400	0.055	0.010	0	0
NoisyNet 5.0 [200k]	*NoisyNet 5.0* [300k]	draw	0	0	400	0.045	0.045	0	0
NoisyNet 5.0 [300k]	*NoisyNet 5.0* [400k]	draw	0	0	400	0.105	0.065	0	0
NoisyNet 5.0 [400k]	*NoisyNet 5.0* [500k]	draw	0	0	400	0.160	0.080	0	0
NoisyNet 5.0 [500k]	*NoisyNet 5.0* [600k]	draw	0	0	400	0.050	0.060	0	0
NoisyNet 5.0 [600k]	*NoisyNet 5.0* [700k]	draw	0	0	400	0.115	0.145	0	0
NoisyNet 5.0 [700k]	*NoisyNet 5.0* [800k]	draw	0	0	400	0.030	0.050	0	0
NoisyNet 5.0 [800k]	*NoisyNet 5.0* [900k]	draw	0	0	400	0.055	0.065	0	0
NoisyNet 5.0 [900k]	*NoisyNet 5.0* [1M]	draw	0	0	400	0.085	0.030	0	0
NoisyNet 5.0 [200k]	*NoisyNet 5.0* [100k]	draw	0	0	400	0.045	0.090	0	0
NoisyNet 5.0 [300k]	*NoisyNet 5.0* [200k]	draw	0	0	400	0.040	0.010	0	0
NoisyNet 5.0 [400k]	*NoisyNet 5.0* [300k]	draw	0	0	400	0.105	0.045	0	0
NoisyNet 5.0 [500k]	*NoisyNet 5.0* [400k]	draw	0	0	400	0.045	0.040	0	0
NoisyNet 5.0 [600k]	*NoisyNet 5.0* [500k]	draw	0	0	400	0.165	0.170	0	0
NoisyNet 5.0 [700k]	*NoisyNet 5.0* [600k]	draw	0	0	400	0.030	0.100	0	0
NoisyNet 5.0 [800k]	*NoisyNet 5.0* [700k]	draw	0	0	400	0.055	0.125	0	0
NoisyNet 5.0 [900k]	*NoisyNet 5.0* [800k]	draw	0	0	400	0.090	0.075	0	0
NoisyNet 5.0 [1M]	*NoisyNet 5.0* [900k]	white	1	-1	400	0.175	0.120	1	0

Table 6. Results for *NoisyNet 5.0* agents

White agent	Black agent	Winner	*S(w)*	*S(b)*	*#T*	*#UT/#T(w)*	*#UT/#T(b)*	*#E(w)*	*#E(b)*
EpsGreedy 100k [100k]	*EpsGreedy 100k* [200k]	white	1	-1	400	0.275	0.245	1	0
EpsGreedy 100k [200k]	*EpsGreedy 100k* [300k]	draw	0	0	400	0.070	0.050	0	0
EpsGreedy 100k [300k]	*EpsGreedy 100k* [400k]	draw	0	0	400	0.105	0.090	0	0
EpsGreedy 100k [400k]	*EpsGreedy 100k* [500k]	draw	0	0	400	0.165	0.175	0	0
EpsGreedy 100k [500k]	*EpsGreedy 100k* [600k]	draw	0	0	400	0.085	0.045	0	0
EpsGreedy 100k [600k]	*EpsGreedy 100k* [700k]	draw	0	0	400	0.035	0.030	0	0
EpsGreedy 100k [700k]	*EpsGreedy 100k* [800k]	draw	0	0	400	0.055	0.045	0	0
EpsGreedy 100k [800k]	*EpsGreedy 100k* [900k]	draw	0	0	400	0.075	0.060	0	0
EpsGreedy 100k [900k]	*EpsGreedy 100k* [1M]	draw	0	0	400	0.030	0.035	0	0
EpsGreedy 100k [200k]	*EpsGreedy 100k* [100k]	draw	0	0	400	0.090	0.080	0	0
EpsGreedy 100k [300k]	*EpsGreedy 100k* [200k]	draw	0	0	400	0.065	0.070	0	0
EpsGreedy 100k [400k]	*EpsGreedy 100k* [300k]	draw	0	0	400	0.035	0.035	0	0
EpsGreedy 100k [500k]	*EpsGreedy 100k* [400k]	draw	0	0	400	0.030	0.040	0	0
EpsGreedy 100k [600k]	*EpsGreedy 100k* [500k]	draw	0	0	400	0.035	0.045	0	0
EpsGreedy 100k [700k]	*EpsGreedy 100k* [600k]	draw	0	0	400	0.040	0.025	0	0
EpsGreedy 100k [800k]	*EpsGreedy 100k* [700k]	draw	0	0	400	0.075	0.090	0	0
EpsGreedy 100k [900k]	*EpsGreedy 100k* [800k]	draw	0	0	400	0.030	0.060	0	0
EpsGreedy 100k [1M]	*EpsGreedy 100k* [900k]	draw	0	0	400	0.025	0.060	0	0

Table 7. Results for *EpsGreedy 100k* agents

White agent	Black agent	Winner	*S(w)*	*S(b)*	*#T*	*#UT/#T(w)*	*#UT/#T(b)*	*#E(w)*	*#E(b)*
EpsGreedy 500k [100k]	*EpsGreedy 500k* [200k]	draw	0	0	400	0.065	0.050	0	0
EpsGreedy 500k [200k]	*EpsGreedy 500k* [300k]	draw	0	0	400	0.075	0.045	0	0
EpsGreedy 500k [300k]	*EpsGreedy 500k* [400k]	draw	0	0	400	0.050	0.035	0	0
EpsGreedy 500k [400k]	*EpsGreedy 500k* [500k]	draw	0	0	400	0.140	0.135	0	0
EpsGreedy 500k [500k]	*EpsGreedy 500k* [600k]	draw	0	0	400	0.030	0.100	0	0
EpsGreedy 500k [600k]	*EpsGreedy 500k* [700k]	draw	0	0	400	0.040	0.035	0	0
EpsGreedy 500k [700k]	*EpsGreedy 500k* [800k]	draw	0	0	400	0.195	0.185	0	0
EpsGreedy 500k [800k]	*EpsGreedy 500k* [900k]	draw	0	0	400	0.145	0.090	0	0
EpsGreedy 500k [900k]	*EpsGreedy 500k* [1M]	draw	0	0	400	0.185	0.195	0	0
EpsGreedy 500k [200k]	*EpsGreedy 500k* [100k]	draw	0	0	400	0.300	0.275	1	1
EpsGreedy 500k [300k]	*EpsGreedy 500k* [200k]	draw	0	0	400	0.025	0.060	0	0
EpsGreedy 500k [400k]	*EpsGreedy 500k* [300k]	draw	0	0	400	0.165	0.195	0	0
EpsGreedy 500k [500k]	*EpsGreedy 500k* [400k]	draw	0	0	400	0.050	0.040	0	0
EpsGreedy 500k [600k]	*EpsGreedy 500k* [500k]	black	-1	1	400	0.235	0.255	0	1
EpsGreedy 500k [700k]	*EpsGreedy 500k* [600k]	draw	0	0	400	0.055	0.080	0	0
EpsGreedy 500k [800k]	*EpsGreedy 500k* [700k]	draw	0	0	400	0.030	0.025	0	0
EpsGreedy 500k [900k]	*EpsGreedy 500k* [800k]	draw	0	0	400	0.020	0.035	0	0
EpsGreedy 500k [1M]	*EpsGreedy 500k* [900k]	draw	0	0	400	0.125	0.090	0	0

Table 8. Results for *EpsGreedy 500k* agents

White agent	Black agent	Winner	*S(w)*	*S(b)*	*#T*	*#UT/#T(w)*	*#UT/#T(b)*	*#E(w)*	*#E(b)*
NoisyNet 0.5 [100k]	*NoisyNet 5.0* [100k]	draw	0	0	400	0.110	0.090	0	0
NoisyNet 0.5 [200k]	*NoisyNet 5.0* [200k]	draw	0	0	400	0.055	0.010	0	0
NoisyNet 0.5 [300k]	*NoisyNet 5.0* [300k]	draw	0	0	400	0.060	0.045	0	0
NoisyNet 0.5 [400k]	*NoisyNet 5.0* [400k]	draw	0	0	400	0.065	0.040	0	0
NoisyNet 0.5 [500k]	*NoisyNet 5.0* [500k]	draw	0	0	400	0.040	0.075	0	0
NoisyNet 0.5 [600k]	*NoisyNet 5.0* [600k]	draw	0	0	400	0.040	0.100	0	0
NoisyNet 0.5 [700k]	*NoisyNet 5.0* [700k]	draw	0	0	400	0.095	0.055	0	0
NoisyNet 0.5 [800k]	*NoisyNet 5.0* [800k]	draw	0	0	400	0.080	0.070	0	0
NoisyNet 0.5 [900k]	*NoisyNet 5.0* [900k]	draw	0	0	400	0.070	0.090	0	0
NoisyNet 0.5 [1M]	*NoisyNet 5.0* [1M]	draw	0	0	400	0.080	0.030	0	0
NoisyNet 5.0 [100k]	*NoisyNet 0.5* [100k]	draw	0	0	400	0.055	0.025	0	0
NoisyNet 5.0 [200k]	*NoisyNet 0.5* [200k]	draw	0	0	400	0.070	0.075	0	0
NoisyNet 5.0 [300k]	*NoisyNet 0.5* [300k]	draw	0	0	400	0.090	0.095	0	0
NoisyNet 5.0 [400k]	*NoisyNet 0.5* [400k]	draw	0	0	400	0.145	0.220	0	0
NoisyNet 5.0 [500k]	*NoisyNet 0.5* [500k]	draw	0	0	400	0.050	0.085	0	0
NoisyNet 5.0 [600k]	*NoisyNet 0.5* [600k]	draw	0	0	400	0.125	0.065	0	0
NoisyNet 5.0 [700k]	*NoisyNet 0.5* [700k]	draw	0	0	400	0.030	0.010	0	0
NoisyNet 5.0 [800k]	*NoisyNet 0.5* [800k]	draw	0	0	400	0.075	0.135	0	0
NoisyNet 5.0 [900k]	*NoisyNet 0.5* [900k]	draw	0	0	400	0.070	0.060	0	0
NoisyNet 5.0 [1M]	*NoisyNet 0.5* [1M]	white	1	-1	400	0.225	0.025	1	0

Table 9. Results for *NoisyNet 0.5* agents vs. *NoisyNet 5.0* agents

White agent	Black agent	Winner	*S(w)*	*S(b)*	*#T*	*#UT/#T(w)*	*#UT/#T(b)*	*#E(w)*	*#E(b)*
NoisyNet 0.5 [100k]	*EpsGreedy 100k* [100k]	draw	0	0	400	0.155	0.075	0	0
NoisyNet 0.5 [200k]	*EpsGreedy 100k* [200k]	draw	0	0	400	0.055	0.090	0	0
NoisyNet 0.5 [300k]	*EpsGreedy 100k* [300k]	white	2	-2	400	0.125	0.205	2	0
NoisyNet 0.5 [400k]	*EpsGreedy 100k* [400k]	draw	0	0	400	0.095	0.175	0	0
NoisyNet 0.5 [500k]	*EpsGreedy 100k* [500k]	draw	0	0	400	0.040	0.065	0	0
NoisyNet 0.5 [600k]	*EpsGreedy 100k* [600k]	draw	0	0	400	0.040	0.095	0	0
NoisyNet 0.5 [700k]	*EpsGreedy 100k* [700k]	draw	0	0	400	0.095	0.135	0	0
NoisyNet 0.5 [800k]	*EpsGreedy 100k* [800k]	draw	0	0	400	0.090	0.120	0	0
NoisyNet 0.5 [900k]	*EpsGreedy 100k* [900k]	draw	0	0	400	0.110	0.125	0	0
NoisyNet 0.5 [1M]	*EpsGreedy 100k* [999k]	draw	0	0	400	0.095	0.185	0	0
EpsGreedy 100k [100k]	*NoisyNet 0.5* [100k]	draw	0	0	400	0.125	0.070	0	0
EpsGreedy 100k [200k]	*NoisyNet 0.5* [200k]	draw	0	0	400	0.105	0.065	0	0
EpsGreedy 100k [300k]	*NoisyNet 0.5* [300k]	draw	0	0	400	0.150	0.130	0	0
EpsGreedy 100k [400k]	*NoisyNet 0.5* [400k]	draw	0	0	400	0.205	0.120	0	0
EpsGreedy 100k [500k]	*NoisyNet 0.5* [500k]	draw	0	0	400	0.055	0.085	0	0
EpsGreedy 100k [600k]	*NoisyNet 0.5* [600k]	draw	0	0	400	0.050	0.065	0	0
EpsGreedy 100k [700k]	*NoisyNet 0.5* [700k]	draw	0	0	400	0.115	0.010	0	0
EpsGreedy 100k [800k]	*NoisyNet 0.5* [800k]	draw	0	0	400	0.110	0.070	0	0
EpsGreedy 100k [900k]	*NoisyNet 0.5* [900k]	draw	0	0	400	0.030	0.070	0	0
EpsGreedy 100k [1M]	*NoisyNet 0.5* [1M]	draw	0	0	400	0.025	0.025	0	0

Table 10. Results for *NoisyNet 0.5* agents vs. *EpsGreedy 100k* agents

White agent	Black agent	Winner	*S(w)*	*S(b)*	*#T*	*#UT/#T(w)*	*#UT/#T(b)*	*#E(w)*	*#E(b)*
NoisyNet 5.0 [100k]	*EpsGreedy 100k* [100k]	draw	0	0	400	0.055	0.080	0	0
NoisyNet 5.0 [200k]	*EpsGreedy 100k* [200k]	draw	0	0	400	0.045	0.070	0	0
NoisyNet 5.0 [300k]	*EpsGreedy 100k* [300k]	draw	0	0	400	0.100	0.105	0	0
NoisyNet 5.0 [400k]	*EpsGreedy 100k* [400k]	draw	0	0	400	0.110	0.150	0	0
NoisyNet 5.0 [500k]	*EpsGreedy 100k* [500k]	draw	0	0	400	0.050	0.055	0	0
NoisyNet 5.0 [600k]	*EpsGreedy 100k* [600k]	white	2	-2	400	0.225	0.200	2	0
NoisyNet 5.0 [700k]	*EpsGreedy 100k* [700k]	draw	0	0	400	0.030	0.035	0	0
NoisyNet 5.0 [800k]	*EpsGreedy 100k* [800k]	draw	0	0	400	0.065	0.075	0	0
NoisyNet 5.0 [900k]	*EpsGreedy 100k* [900k]	draw	0	0	400	0.085	0.140	0	0
NoisyNet 5.0 [1M]	*EpsGreedy 100k* [1M]	draw	0	0	400	0.110	0.035	0	0
EpsGreedy 100k [100k]	*NoisyNet 5.0* [100k]	black	-1	1	400	0.095	0.110	0	1
EpsGreedy 100k [200k]	*NoisyNet 5.0* [200k]	draw	0	0	400	0.025	0.010	0	0
EpsGreedy 100k [300k]	*NoisyNet 5.0* [300k]	draw	0	0	400	0.095	0.045	0	0
EpsGreedy 100k [400k]	*NoisyNet 5.0* [400k]	draw	0	0	400	0.075	0.040	0	0
EpsGreedy 100k [500k]	*NoisyNet 5.0* [500k]	draw	0	0	400	0.125	0.075	0	0
EpsGreedy 100k [600k]	*NoisyNet 5.0* [600k]	draw	0	0	400	0.115	0.095	0	0
EpsGreedy 100k [700k]	*NoisyNet 5.0* [700k]	draw	0	0	400	0.075	0.075	0	0
EpsGreedy 100k [800k]	*NoisyNet 5.0* [800k]	draw	0	0	400	0.115	0.080	0	0
EpsGreedy 100k [900k]	*NoisyNet 5.0* [900k]	draw	0	0	400	0.030	0.065	0	0
EpsGreedy 100k [1M]	*NoisyNet 5.0* [1M]	draw	0	0	400	0.025	0.030	0	0

Table 11. Results for *NoisyNet 5.0* agents vs. *EpsGreedy 100k* agents

White agent	Black agent	Winner	*S(w)*	*S(b)*	*#T*	*#UT/#T(w)*	*#UT/#T(b)*	*#E(w)*	*#E(b)*
NoisyNet 5.0 [100k]	*EpsGreedy 500k* [100k]	draw	0	0	400	0.055	0.175	0	0
NoisyNet 5.0 [200k]	*EpsGreedy 500k* [200k]	draw	0	0	400	0.045	0.075	0	0
NoisyNet 5.0 [300k]	*EpsGreedy 500k* [300k]	black	-2	2	400	0.180	0.240	0	2
NoisyNet 5.0 [400k]	*EpsGreedy 500k* [400k]	draw	0	0	400	0.180	0.195	0	0
NoisyNet 5.0 [500k]	*EpsGreedy 500k* [500k]	draw	0	0	400	0.045	0.095	0	0
NoisyNet 5.0 [600k]	*EpsGreedy 500k* [600k]	draw	0	0	400	0.130	0.085	0	0
NoisyNet 5.0 [700k]	*EpsGreedy 500k* [700k]	draw	0	0	400	0.030	0.035	0	0
NoisyNet 5.0 [800k]	*EpsGreedy 500k* [800k]	draw	0	0	400	0.055	0.070	0	0
NoisyNet 5.0 [900k]	*EpsGreedy 500k* [900k]	draw	0	0	400	0.085	0.065	0	0
NoisyNet 5.0 [1M]	*EpsGreedy 500k* [999k]	draw	0	0	400	0.130	0.105	0	0
EpsGreedy 500k [100k]	*NoisyNet 5.0* [100k]	draw	0	0	400	0.075	0.090	0	0
EpsGreedy 500k [200k]	*NoisyNet 5.0* [200k]	draw	0	0	400	0.090	0.010	0	0
EpsGreedy 500k [300k]	*NoisyNet 5.0* [300k]	draw	0	0	400	0.030	0.045	0	0
EpsGreedy 500k [400k]	*NoisyNet 5.0* [400k]	draw	0	0	400	0.145	0.040	0	0
EpsGreedy 500k [500k]	*NoisyNet 5.0* [500k]	draw	0	0	400	0.105	0.075	0	0
EpsGreedy 500k [600k]	*NoisyNet 5.0* [600k]	draw	0	0	400	0.075	0.100	0	0
EpsGreedy 500k [700k]	*NoisyNet 5.0* [700k]	draw	0	0	400	0.060	0.075	0	0
EpsGreedy 500k [800k]	*NoisyNet 5.0* [800k]	draw	0	0	400	0.040	0.050	0	0
EpsGreedy 500k [900k]	*NoisyNet 5.0* [900k]	draw	0	0	400	0.105	0.065	0	0
EpsGreedy 500k [999k]	*NoisyNet 5.0* [1M]	draw	0	0	400	0.045	0.030	0	0

Table 12. Results for *NoisyNet 5.0* agents vs. *EpsGreedy 500k* agents

White agent	Black agent	Winner	*S(w)*	*S(b)*	*#T*	*#UT/#T(w)*	*#UT/#T(b)*	*#E(w)*	*#E(b)*
EpsGreedy 100k [100k]	*EpsGreedy 500k* [100k]	draw	0	0	400	0.080	0.090	0	0
EpsGreedy 100k [200k]	*EpsGreedy 500k* [200k]	draw	0	0	400	0.080	0.060	0	0
EpsGreedy 100k [300k]	*EpsGreedy 500k* [300k]	draw	0	0	400	0.060	0.025	0	0
EpsGreedy 100k [400k]	*EpsGreedy 500k* [400k]	draw	0	0	400	0.055	0.035	0	0
EpsGreedy 100k [500k]	*EpsGreedy 500k* [500k]	draw	0	0	400	0.155	0.115	0	0
EpsGreedy 100k [600k]	*EpsGreedy 500k* [600k]	draw	0	0	400	0.035	0.025	0	0
EpsGreedy 100k [700k]	*EpsGreedy 500k* [700k]	draw	0	0	400	0.060	0.050	0	0
EpsGreedy 100k [800k]	*EpsGreedy 500k* [800k]	draw	0	0	400	0.135	0.140	0	0
EpsGreedy 100k [900k]	*EpsGreedy 500k* [900k]	draw	0	0	400	0.030	0.010	0	0
EpsGreedy 100k [1M]	*EpsGreedy 500k* [1M]	draw	0	0	400	0.025	0.025	0	0
EpsGreedy 500k [100k]	*EpsGreedy 100k* [100k]	draw	0	0	400	0.100	0.105	0	0
EpsGreedy 500k [200k]	*EpsGreedy 100k* [200k]	draw	0	0	400	0.210	0.170	1	1
EpsGreedy 500k [300k]	*EpsGreedy 100k* [300k]	draw	0	0	400	0.025	0.105	0	0
EpsGreedy 500k [400k]	*EpsGreedy 100k* [400k]	draw	0	0	400	0.165	0.150	0	0
EpsGreedy 500k [500k]	*EpsGreedy 100k* [500k]	draw	0	0	400	0.040	0.070	0	0
EpsGreedy 500k [600k]	*EpsGreedy 100k* [600k]	draw	0	0	400	0.030	0.035	0	0
EpsGreedy 500k [700k]	*EpsGreedy 100k* [700k]	draw	0	0	400	0.065	0.110	0	0
EpsGreedy 500k [800k]	*EpsGreedy 100k* [800k]	draw	0	0	400	0.120	0.125	0	0
EpsGreedy 500k [900k]	*EpsGreedy 100k* [900k]	draw	0	0	400	0.020	0.060	0	0
EpsGreedy 500k [1M]	*EpsGreedy 100k* [1M]	draw	0	0	400	0.095	0.040	0	0

Table 13. Results for *EpsGreedy 100k* agents vs. *EpsGreedy 500k* agents