Reinforcement Learning Algorithm Comparisons For The Game Of Reversi

Cameron Humphreys
Student: 101162528 Email: `CameronHumphreys@cmail.carleton.ca`
Lauris Petlah
Student: 101156789 Email: `laurispetlah@cmail.carleton.ca`
Sukhrobjon Eshmirzaev
Student: 101169793 Email: `SukhrobjonEshmirzaev@cmail.carleton.ca`
Awwab Mahdi
Student: 101225637 Email: `awwabmahdi@cmail.carleton.ca`

1 Introduction

1.1 Purpose

This project focuses on the problem of creating a highly performant agent using Reinforcement learning techniques for the game Reversi. Reversi is a piece-capturing board game where disks are placed adjacent to existing opponent disks, capturing all consecutive opponent pieces between the new disk and the nearest ally disk. Similarily, the game Othello is a variation of this game, and has similar rules to Reversi and strategies within one usually carry over to the other.

Our implementation has a fixed starting state of 4 disks in the middle of the board in a square shape, where players take alternating turns. A player must place a disk such that it captures at minimum one opponent disk, skipping your turn if no legal move exists.

Due to the large variability of the game, with a large number of actions, states and possible ways to win, we decided that this game is a perfect candidate for various reinforcement learning strategies.

Figure 1: Two simple starting moves in Reversi.

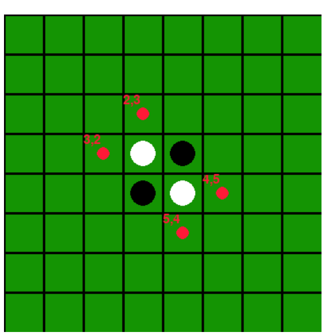

Figure 2: Starting configuration in Reversi. Red dots signify the available moves the Black player is allowed to play

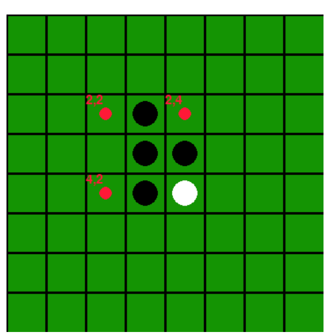

Figure 3: Black player placed a disk at (2,3). Red dots signify the available moves the White player is allowed to play

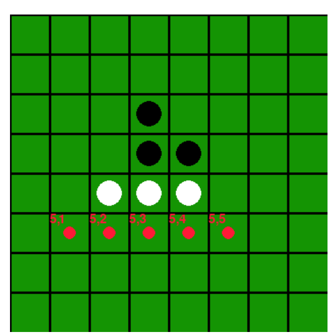

Figure 4: White player placed a disk at (4,2). Red dots signify the available moves the Black player is allowed to play

Preprint. Under review.

The game of Reversi ends when neither player can make a legal move, which typically means the game board is filled, but not always. Once the game has ended, the scoring is simple. The winner of Reversi is the player with the most game pieces on the board at the end of the game.

This problem is made complex by the 8×8 sized board which contains approximately 3^{64} possible states, each with a set of actions ≥ 0. With such a state and action space, tabular solutions are not practical, which makes different function approximation and Deep Reinforcement Learning solutions attractive. Since there are multiple methods to approach this problem, our project implements multiple algorithms and compares them against a baseline (Random) player and against each other. In particular, this project explores the effectiveness of Deep Q-Learning, and Deep SARSA.

1.2 Interests and Insights

Since our problem is so similar to grid-based board games, our project may provide insights into the benefits and drawbacks of different Reinforcment Learning agents in this space.

Our project sources ideas for different function approximation algorithms from similar board games, and applied in insightful ways to find the most optimal application of these algorithms. These insights can be then extended to other board games and possibly applied there as well.

In addition, the selection of multiple function approximation algorithms allows us to learn much more about the optimal learning methods, hyperparameters, and learning times for these different algorithms. Specifically, we can learn how algorithms taught with self-play interact with eachother or with the random agent and evaluate tailored learning methods for each algorithm.

In summary, there is a significant amount of knowledge in the development of this project that provides insights about Reinforcement Learning in Reversi. In addition, the insight gained in this project has the possibility of expanding beyond this specific problem into general problems in board and strategy games.

1.3 Goal

The goal of this report is firstly to detail to the reader the existing knowledge on Reversi in the Reinforcement learning space, then, we wish to impart multiple approaches to Reversi as a Reinforcement Learning problem, and to compare these approaches.

Finally, this report aims to use the empirical results gained through careful experimentation performed on the various methods to gain insights into the optimal algorithms and parameters of each approach.

2 Methodology

2.1 Reinforcement Learning Fundamentals & Reversi

Environment & Markov Decision Processes (MDPs) An MDP is a network of states and actions that result in rewards. We denote all states $s \in S$, all actions $a \in A$, and all rewards as $r(s, a) \mapsto \mathbb{R}$ with . Additionally, each action has a probability of $P(s, a, s')$ of occuring. Finally, The states actions, rewards and probabilities are all set by the environment. In Reversi the state is represented by two parts: the board and the current player. The board consists of 64 grid spaces that can be each occupied by either a Black disk, a White disk, or no disk and can be represented via a 2D-matrix of {-1,0,1} respectively. An action in Reversi can be denoted by a pair ranging from (1,1) to (8,8) denoting the action of placing a disk. The resulting state of an action is one that flips all consecutive opponent disks in between the disk placed, and other surrounding disks. The reward is 0 for a loss, 0.5 for a draw and 1 for a win, with all other states being 0 reward.

Policy The policy for a given state, denoted $\pi(s)$, produces an action dependent on the state for the environment. Many different policies exist with different methods for determining an action.

Agent An agent is an active participant in the environment, with which all reinforcement learning is centered on. That is, the agent progress through the an MDP of an environment with some given policy π. Each state S_t, action A_t, and reward R_t are recorded by each time step t to T where T is the terminal state.

Long-Term Reward For an Agent, the long-term reward of a given state is defined as the cumulative reward for all states in the environment at each time-step. That is, $G_t = R_t + G_{t+1}$.

Q-Function In an ideal setting, the Q-function, denoted $Q(s, a)$, measures the cumulative future reward of the current state-action pair, i.e., G_t. Thus, it reasons that the optimal policy that uses Q-learning is the policy,

$$\pi(s) = \arg\max_a(Q(s, a)) \tag{1}$$

Value-Function In an ideal setting, the Value-function, denoted $V(s)$, measures the longperfectly estimates the best the cumulative future reward of the current state-action pair. Thus, it reasons that the optimal policy that uses Q-learning is the policy,

Reward-Shaping Additionally, in environments with sparse rewards, it may be necessary to introduce an additional hand-crafted reward to the current reward. If the reward from one state following an action to another is given by $R(s, a, s')$ then a shaped reward will be given by, $R'(s, a, s') = R'(s, a, s') + F(s, a, s')$ following some shaping function F.

2.2 Q-Approximation

Q-Learning is a model-free approach to reinforcement learning by learning a policy through an estimated Q-function. Using a combination of the current state and action, Q-Learning attempts to approximate the cumulative future reward of the current state-action pair and update itself accordingly. To do so, we use the Bellman equation,

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r(s, a) + \gamma \max_a Q(s', a) - Q(s, a)] \tag{2}$$

Where α is the step-size or learning rate hyperparameter and γ is the hyperparameter discerning the discount factor, or more simply, the importance of long-term reward

As for policy selection, the ϵ-greedy policy is used which has the benefits of exploring possible states, and also determining the effectiveness of the current $Q(s, a)$ values. That is, choosing some policy based on,

$$\pi_\epsilon(s) = \begin{cases} a \sim Unif(A) & \text{with probability } \epsilon \\ \arg\max_a(Q(s, a)) & \text{with probability } 1 - \epsilon \end{cases} \tag{3}$$

Where $\epsilon \in [0, 1]$.

Deep Q-Learning & Neural Networks We note that Q-learning is often performed using a Q-table of some sort. However, for large state-action spaces this quickly becomes infeasible for the reason finite storage capacity. As a work around, Neural Networks can be used in lieu of this restriction as an abstraction of the Q-table in exchange for a more complex learning algorithm.

Literature on Deep Q-learning have been shown to implement this in several ways. Notably, for discrete action spaces, it is common such as in van der Ree and Wiering [2013] to use a neural network with simply a state as input, and all resulting Q-values for that state and all possible actions (2.2). Others use a neural network with a state-action pair as input, and only the resulting Q-values for output 2.2 as seen in Choudhary [2023].

2.3 Deep Q-Learning and Reversi/Othello

We aim to implement Deep Q-learning within the game space using several methods. Firstly, we intend to use both types of neural networks described in section 2.2. With Q-learning being an off-policy learning approach, we are able to use an experience replay buffer such that past state-action pairs and their resulting state and rewards can be used as another way to train the agent. We further intend to use self-play, a method that uses a previous version of the current model being trained as the opponent policy. Finally, we also intend to explore the feasiblity of using reward shaping.

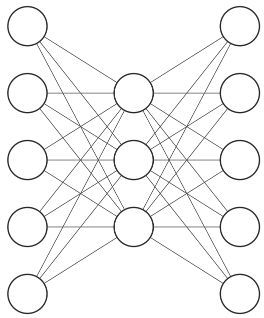

Figure 5: An environment with 5 states and 5 actions with a DQN that maps $S \mapsto Q(s,a) \times A$

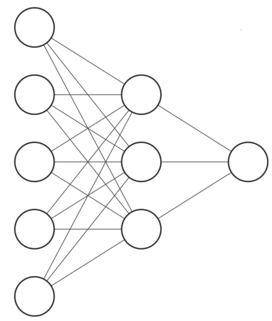

Figure 6: An environment with 3 states and 2 actions with a DQN that maps $S \times A \mapsto Q(s,a)$

Experience Replay A benefit of off-policy learning is the ability to learn from past actions just as well as current ones, thus storing past experiences are beneficial. Inspired by Mnih et al. [2013], we store an experience replay buffer with (s, a, r, s') after each action of the agent. Where s' is the resulting state of performing a at s and the opponent takes their turn. We can then train the agent with the experience replay buffer.

Self-Play With self-play, an agent will play against the same, but earlier iteration of the current model one during training, which has been shown, such as in van der Ree and Wiering [2013] which has been shown to be beneficial. Specifically, we will use the 5th previous iteration of the current model as to introduce some randomness to learning

Reward Shaping While complex reward shaping exists such as in Ng et al. [1999], this requires extensive expert knowledge of the environment as this is a tangible guideline for the agent to follow. Instead, we opt for a simpler reward shaping algorithm,

$$F(s, a, s') = \frac{\text{(\# Of Disks Of Current Agent's Color)}}{64} \tag{4}$$

Neural Network We will implement two networks with the following characteristics,

- Network 1: An input of size 67 consisting of a combination of a state s and action a (64 grid-squares $+1$ current player $+2$ values for grid location of the next action). An output consisting of size 1, a proxy for $Q(s, a)$.
- Network 2: An input of size 65 representing a state s (64 grid-squares $+1$ current player). An output consisting of size 64, a proxy for $Q(s, a)\ \forall a \in A$ given that there are 64 actions ((1,1) to (8,8)) regardless of legality.

Each network will have 3 hidden layers each of size 64. All nodes will use sigmoid activation function, and a loss of MSE will be used for computation of back propogation. Network values are fitted against the updated $Q(s, a)$ values. That is, the model is fitted against,

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a) \tag{5}$$

In regards to network 2, the update is carried out the same, with the additional requirement that the update is a size 64 array of 0's where corresponding action a in the original $Q(s, a)$ is set to the value of (5). Both networks will be fitted after a round has been played with the entirety of the replay buffer, which includes the round that had just been played.

The hyperparameters used for all models was,

- $\alpha = 0.001$
- $\gamma = 0.1$

- $\epsilon = 0.2$ Decaying linearly to $\epsilon = 0$ at the final episode
- A replay buffer of size 4096
- Iteration of learning until no significant improvement in learning has occured for 10 episodes.

Experiments We aim to find out the optimal Deep Q-network through a set of experiments. That is, we wish to find the highest performing network that has generalized the best. In both van der Ree and Wiering [2013] and van Eck and van Wezel [2008], a dueling like structure was used to test all networks involved. We similarily follow this approach and extend it to a tournament structure. That is, we first find the highest performing networks against a random policy opponent, and then rank them each against one another.

3 Experimental Results

We will outline the experimental results gathered from the various experiments and attempt to reach plausible conclusions.

3.1 Deep Q-Learning

The experiments were performed in a tournament style comparision, first comparing against a random policy player, then comparing against the best agents trained to assess performance further. All agents are identified by the following rules,

- dqn1 denotes using network 1 topology
- dqn2 denotes using network 2 topology
- rs denotes using simple reward-shaping (4)
- selfplay denotes using selfplay during training.

Thus, for example, dqn2-rs-selfplay indicates using a network 2 topology, reward-shaping and selfplay, while dqn1-rs denotes a network 1 topology with reward shaping and no selfplay. Next, we define the metrics for the various experiments as Win Rate = $\frac{\#\text{Wins}}{\#\text{Games}}$, Non-Loss Rate = $\frac{\#\text{Wins + \#Draws}}{\#\text{Games}}$ and Non-Win Rate = $\frac{\#\text{Draws + \#Loss}}{\#\text{Games}}$

Table 1: DQN Agent Performance after 50 games against Random Policy player.

Agent	Wins	Draws	Losses	Win Rate	Non-Loss Rate	Non-Win Rate
dqn1	30	2	18	0.6	0.64	0.4
dqn1-rs	19	3	28	0.38	0.44	0.62
dqn1-selfplay	26	1	23	0.52	0.54	0.48
dqn1-rs-selfplay	25	1	24	0.5	0.52	0.5
dqn2	32	12	6	0.64	0.88	0.36
dqn2-rs	23	0	27	0.46	0.46	0.54
dqn2-selfplay	26	2	22	0.52	0.56	0.48
dqn2-rs-selfplay	20	1	29	0.4	0.42	0.6

Observe that the highest performing agent in Table 1 is dqn2 with a win-rate of 0.64 and a non-loss rate of 0.88. We also observe that the highest performing agent of the first network is dqn1 with a win-rate of 0.6 and non-loss rate of 0.64. Next, we play these two agents against every other agent and observe the results.

We observe that the dqn2 agent (Table 2) is more performant than the dqn player (Table 3) against all other players. Verifiably, the total games won or drawn by the dqn2 player against all other plays is 210 Losses against dqn2 + 146 Draws = 356 Non-Losses, while for dqn1, there were 188 Losses against dqn1 + 50 Draws = 238 Non-Losses.This implies that the dqn2 player generalized better than the dqn agent, likely due to the fact that because of the network topology, the extended output of all actions caused undesirable actions to be suppressed at the same time as the desirable action being emphasized.

Table 2: DQN Agent Performance after 50 games against dqn2 player.

Agent	Wins	Draws	Losses	Win Rate	Non-Loss Rate	Non-Win Rate
dqn1	0	25	25	0	0.5	1
dqn1-rs	26	24	0	0.52	1	0.48
dqn1-selfplay	0	25	25	0	0.5	1
dqn1-rs-selfplay	18	0	32	0.36	0.36	0.64
dqn2	0	50	0	0	1	1
dqn2-rs	0	22	28	0	0.44	1
dqn2-selfplay	0	0	50	0	0	1
dqn2-rs-selfplay	0	0	50	0	0	1

Table 3: DQN Agent Performance after 50 games against dqn1 player.

Agent	Wins	Draws	Losses	Win Rate	Non-Loss Rate	Non-Win Rate
dqn1	0	50	0	0	1	1
dqn1-rs	28	0	22	0.56	0.56	0.44
dqn1-selfplay	19	0	31	0.38	0.38	0.62
dqn1-rs-selfplay	34	0	16	0.68	0.68	0.32
dqn2	25	0	25	0.5	0.5	0.5
dqn2-rs	0	0	50	0	0	1
dqn2-selfplay	28	0	22	0.56	0.56	0.44
dqn2-rs-selfplay	28	0	22	0.56	0.56	0.44

Some other important conclusions we can draw is the fact that all reward-shaping agents performed worse against random (Table 1). Notably, dqn1-rs performed better against both dqn1 and dqn2 agents, in fact denying any wins for dqn2, and performing similarily to dqn1. Selfplay agents performed at best equal against random policy (Table 1) as well as dqn1 (Table 3) and dqn2 (Table 2). Notably, dqn1-rs-selfplay outperformed dqn1 with a higher win rate.

References

Ankit Choudhary. A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python — analyticsvidhya.com. `https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/#h-end-notes`, 2023. [Accessed 10-12-2023].

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pages 278–287. Citeseer, 1999.

Michiel van der Ree and Marco Wiering. Reinforcement learning in the game of othello: Learning against a fixed opponent and learning from self-play. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 108–115, 2013. doi: 10.1109/ADPRL.2013.6614996.

Nees Jan van Eck and Michiel van Wezel. Application of reinforcement learning to the game of othello. *Computers and Operations Research*, 35(6):1999–2017, 2008. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2006.10.004. URL `https://www.sciencedirect.com/science/article/pii/S0305054806002553`. Part Special Issue: OR Applications in the Military and in Counter-Terrorism.