# 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.

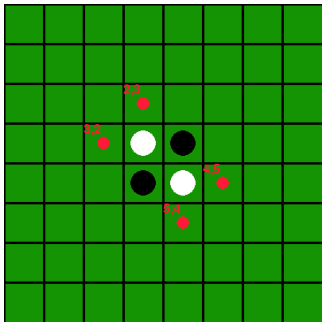


Figure 2: Starting configuration in Reversi. Red dots signify the available moves the Black player is allowed to play

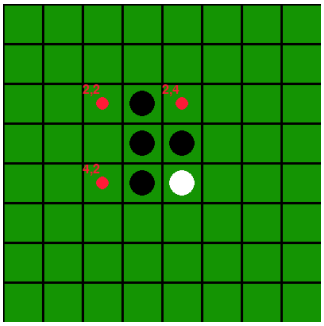


Figure 3: Black player placed a disk at (2,3). Observe that a white disk was converted to a black disk

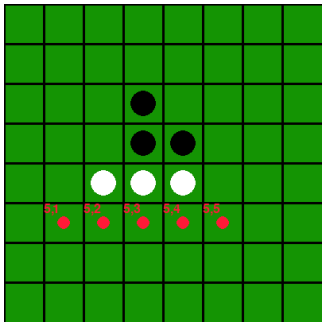


Figure 4: White player placed a disk at (4,2). Observe that a black disk was converted to a white disk

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 $8 \times 8$ sized board which contains approximately $3^{64}$ possible states, each with a set of actions $\geq 0$. 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 $\pi$. 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 cumulative future reward of the current state. Thus, it reasons that the optimal policy chosses an action according to:

$$V(s) = \max_a(\sum_{s,a} p(s', r|s, a)[r + \gamma V(s')]) \tag{2}$$

**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,

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r(s, a) + \gamma \max_a Q(s', a) - Q(s, a)] \tag{3}$$

Where $\alpha$ is the step-size or learning rate hyperparameter and $\gamma$ is the hyperparameter discerning the discount factor, or more simply, the importance of long-term reward

As for policy selection, the $\epsilon$-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,

$$\pi_\epsilon(s) = \begin{cases} a \sim Unif(A) & \text{with probability } \epsilon \\ \arg\max_a(Q(s, a)) & \text{with probability } 1 - \epsilon \end{cases} \tag{4}$$

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 **?** 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 **?**.

### 2.3 Deep Q-Learning and Reversi

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.

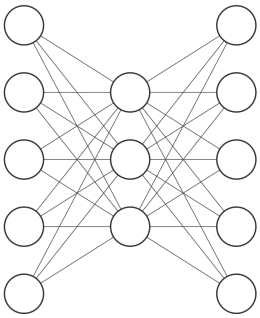

Figure 5: An environment with 5 states and 5 actions with a DQN that maps $S \mapsto Q(s,a) \times A$

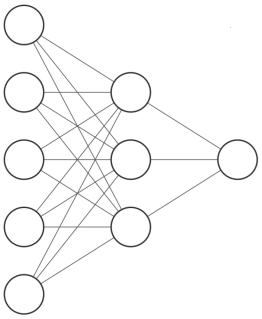

Figure 6: An environment with 3 states and 2 actions with a DQN that maps $S \times A \mapsto Q(s,a)$

**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 **?**, 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 **?** 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 **?**, 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,

$$F(s, a, s') = \frac{\text{(\# Of Disks Of Current Agent's Color)}}{64} \tag{5}$$

**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,

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a) \tag{6}$$

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 (6). 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.

### 2.4 Deep Reinforcement Learning with Experience Replay Based on SARSA (Deep SARSA) and Reversi

Deep SARSA (State-Action-Reward-State-Action) is a reinforcement learning algorithm that extends the SARSA algorithm by incorporating deep neural networks. Inspired by the application of this algorithm to solve video games control problem as in **?**, we aim to implement this algorithm to learn an optimal policy to play Reversi game.

**SARSA** SARSA is an on-policy reinforcement learning algorithm that updates its Q-values (action values) based on the current policy. The Q-values represent the expected cumulative rewards for taking a particular action in a specific state.

**Deep SARSA** In Deep SARSA, the Q-values are estimated using a deep neural network. This allows the algorithm to handle high-dimensional state spaces like the board configuration in Reversi.The neural network takes the current state as input and outputs Q-values for all possible actions. We use the same netwrok as Network 1 used in Deep Q-Learning implementation mentioned above.

**Experience Replay** Deep SARSA uses experience replay to improve sample efficiency and stability. It stores experiences (state, action, reward, next state) in a replay buffer and samples random batches during training using different sampling methods such as consecutive sampling, uniform sampling, and weighted sampling method by rewards **?**. We use uniform sampling method in this implementation.

**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 **?** which has been shown to be beneficial.

The hyperparameters used for all models was,

- $\gamma = 0.1$
- $\epsilon = 0.1$
- A replay buffer of size 256
- Sample 64 experiences uniformly random for experience replay
- Iteration of learning over 100 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 **?** and **?**, 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.

### 2.5 Monte Carlo Tree Search and Reversi/Othello

We aim to implement a Monte Carlo Tree Search that explores the Game Tree of Reversi. The Monte Carlo Tree search method uses four fundamental steps, Selection, Rollout, Expansion and Backpropagation that we will describe below. The MCTS will repeat traversal, which is made up of these four steps, an amount specified by the parameter $n_iter$ every time the MCTS must select an action during play.

**Selection** The selection portion of the Monte Carlo Tree Search determines the expansion or rollout of a node, $N$ with children, $C$. Specifically, the node is only expanded or rolled out if $|C| = 0$. In the case that $N$ has children, the algorithm will check the child with the highest $UCT$ score, $c$ with children, $C'$. Selection on $c$ occurs similarily, if $|c'| = 0$, and selects the child, $c'$ with the highest $UCT$ score.

**Rollout** Rollout is the fundamental part that seperates the Monte Carlo Tree Search from other tree searches like $Minimax$. Rollout will simulate a game between two random agents until the game ends and then use Backpropegation for the results of the game.

**Expansion** The Expansion algorithm expands a node, $N$ by generating child nodes $C$ and adding them as the children of $N$. The node is expanded using each action, $a$ from the list of it's legal actions, $A$. $C$ is made up of states that occur after an action $a$ has occured and after an opponent response $o$ of the list of legal responses, $O$ has been selected. Thus, after expansion, $|C| = \sum_A |(O|a)|$.

**Backpropegation** Backpropegation is the simplest step of the Monte Carlo Tree Search. At the end of a rollout, the game is scored, and the reward, which is +1 for a win, 0.5 for a draw, and -1 for a loss, is backpropegated through all parents of the node that was rolled out. In our implementation, the reward is saved in terms of $n_w$ and $n_s$ in $N$, these variables are used to determine the win rate of $N$.

**UCT score** The UCT score, as mentioned in Selection is the score that determines the priority of expansion or rollout when choosing $c$ from $C$. the equation is as follows:

$$UCT = w/n + c\sqrt{\ln t/n} \tag{7}$$

with $w = n_w, n = n_s$ for $N$, $c = 2$ for our implementation, and $t$ as $\sum_{n_w ins}(N)$ for all $N$

## 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 (5)
- 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.

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 |

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.

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.

### 3.2 Deep SARSA (DSQN)

Table 4: DSQN Agent Performance after 50 games.

| Player 1 VS Player 2 | Player 1 win | Draws | Player 2 loss | Win Rate | Non-Loss Rate | Non-Win Rate |
|---|---|---|---|---|---|---|
| dsqn VS random | 25 | 0 | 25 | 0.5 | 0.5 | 0.5 |
| dsqn-selfplay VS random | 31 | 2 | 17 | 0.62 | 0.66 | 0.38 |
| dsqn VS dsqn-selfplay | 0 | 0 | 50 | 0 | 0 | 1 |
| dsqn VS dqn2 | 27 | 0 | 23 | 0.54 | 0.54 | 0.46 |
| dsqn-selfplay VS dqn2 | 28 | 0 | 22 | 0.56 | 0.56 | 0.44 |

We can observere from the table (Table 4) that dsqn is as performant as random player while dsqn-selfplay performs better than random player. Furthermore, It can be seen from the table (Table 4) that dsqn and dsqn-selfplay permorm as good as dqn2. Moreover, dsqn and dsqn-selfplay play draw against each other.

### 3.3 Monte Carlo Tree Search

The MCTS agent in the following table is the same agent with different simulation paramaters, MCTS 5 will do 5 game simulations per turn, and MCTS 10 will do 10 simulations.

Table 5: MCTS Agent Performance Vs Random after 20 games.

| | Player 1 win | Draws | Player 2 loss | Win Rate | Non-Loss Rate | Non-Win Rate |
|---|---|---|---|---|---|---|
| MCTS 5 | 12 | 1 | 7 | 0.6 | 0.65 | 0.4 |
| MCTS 10 | 16 | 1 | 3 | 0.8 | 0.85 | 0.2 |

Due to time constraints and slow performance, the MCTS could only be tested against the random agent for 20 games per run. The data collected does show, however that the MCTS is better than the Random agent, and get's better at larger game simulation amounts.

## 4 Conclusion

Our empirical research has proven several different key points, the efficacy of different reinforcement learning methods, as well as how useful and effective Neural Networks were on the generalization of an agent in an MDP environment.

### 4.1 Deep Q-Learning

Deep Q-Learning was shown to be effective against many agents, indicating that it generalized, likely due to the prescence of a Neural Network. This goes against the conclusion in **?** that Deep Q-Networks were not ideal at generalization. The only indication that this lack of generalization occurred is during selfplay training, as this trained the network with a moving target, rather than random moves.

It is noted that Deep Q-Networks appear to be no less viable than Deep-SARSA and the minimal winrate of Deep-SARSA can be attributed to randomness. However, this may not be the case for larger games played, though this difference appears to be at most a factor of around 5%.

Limitations of simple Deep Q-Networks is likely do to overestimation bias as stated in **?** with solution being through the use of double Q-learning, where an additional Q-network models is used to reduce this bias. Other important Q-networks to explore could have been Dueling Q-networks such as in **?**, where a network represent the components of the Q-function, specifically the value function $V(s)$ and the advantage $A * ()$ more accurately, increasing the stability of the algorithm.

### 4.2 Monte Carlo Tree Search

The Monte Carlo Tree Search suffered from severely poor performance compared to alternate algorithms. This is a result of the complexity difference. Where the other algorithms would estimate the value of a state action, then update it, MCTS would effectively go and simulate a game to find something more analageous to the actual value of that state. The accuracy differences between these two ways of value approximation is debatable, but the performance differences stand.

Through this project, We've learned much about Reversi and board games in RL, different ways to approach Reversi as a problem, and we've gleaned insights on how the differences between DQN, Deep SARSA, and MCTS. We learned specific tuning for these algorithms in terms of hyperparameters, and in terms of strategies like the different neural network topologies, replay, and selfplay.

We would like to improve the MCTS implementation using more training time, optimization for the performance, and different methods of heuristics for play, like deliberately trying to choose corners or edges, or atleast avoiding giving those high valued board positions up. Alternatively, teaching the MCTS using selfplay, or using a different policy like epsilon greedy or softmax for the expansion.

### 4.3 Deep SARSA

In this project we experimented with an on-policy method, Deep Reinforcement Learning based on SARSA (Deep SARSA). SARSA learning has some advantages when being applied to decision making problems. It makes learning process more stable and is more suitable to some complicated systems. Given these facts, Deep SARSA can be used to solve the control problems of video games **?** , and for board games like Reversi. Our experiments have shown that Deep SARSA agents performs better that random player, and it as performant as Deep Q-Learning. If we had more time we would work on improving the run-time complexity of the Deep SARSA implementation.