Reinforcement Learning

Final Report

Mathias Øgaard & Tobias Verheijen

June 2024

Contents

1 Introduction **1**

2 Overview of the project's chronology **2**

3 Environment **2**

3.1 Othello Description 2

3.2 Environment Description 3

3.3 Environment Agents 3

3.4 Environment Functions 4

4 Description of the algorithms **5**

4.1 Policy 6

4.2 Q-Network Design 7

4.3 Training Phase 7

4.4 Q-Network - Self-Play 7

4.5 Q-Network - Positional Learning 8

5 Experimental Results **8**

6 Future work **9**

7 Conclusion **10**

1 Introduction

The main focus of our project[1] is to create an agent which learns how to play the game of Othello. We use an Othello environment built off of the work presented in [1]. This script comes with eight inherent Artificial Intelligence (AI) playing options for the Othello board game. For further information on the AI agents, please see section 3.

[1] Our GitHub repository is available here

The state space of Othello is far too large to run an exhaustive search for each state. Therefore, we start by implementing one non-learning agent for benchmarking: *The Positional Player* (section 4.1 in [2]). Please refer to section 3.3 of our analysis for a more elaborate description of the positional player. We then implement a Q-learning agent (section 4.3 in [2]) and train two models, a self-play and a positional learning agent. To check the progress of the model's learning, we have our learning agent play against the positional agent. In an ideal world, it improves against each of the 8 AI agents and comprehensively beats the positional player used as a baseline. Our report is outlined as follows: We first provide a description of the Othello board game and the environment. Then, we outline the algorithms we use for our analysis. In the end, we close with a summary of our experimental results.

2 Overview of the project's chronology

```
#TODO (optional): Do the project?
#TODO: Strike out the line above, give or take...
```

1. We started by improving the interface to interact with the agents. This is important for human interpretation of the scenario.

2. We added an option to play and learn on a reduced board size, e.g. 4x4, to allow for faster prototyping and evaluation of our algorithmic implementation.

3. Implemented our own Q-Learning player using code and algorithms from class.

4. Ran experiments on our Q-learning player to evaluate its performance

5. Visualized our results

6. Iteratively compared Q-Learning agent against base-line Positional Player implemented and described in section 3.

3 Environment

3.1 Othello Description

A brief description of the Othello board game follows. Opponents with opposing colored discs alternate moves in an attempt to secure as many positions on an 8x8 board of their respective colors as possible. The winner is the player who has the most pieces when an end state is reached: When neither player can make a valid move or the board has no open spaces. A valid move is a move that flips a number of the opponent's pieces by trapping them between your own pieces, like in the example in figure 1.

The example shows only one tile trapped between the dark discs, however this

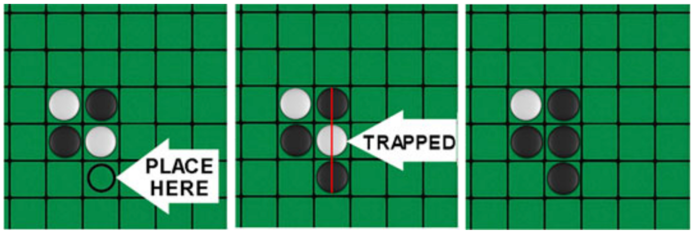


Figure 1: Example of a valid move. Taken from: [3]

can be extended so that several tiles are trapped between one players pieces. In this instance, all of the trapped tiles are flipped before the next player is able to make their move.

3.2 Environment Description

The design of our environment takes into account a number of common pieces: the state space, the set of actions, and the transition function. There are also a number of situationally dependent portions: the reward, the utility, the policy, and the value function. The state is easily represented as the current state of the board, which considers the number of opposing discs, the number of our own discs, and the positions of each disc. The set of actions is a layer which is the size of the Othello board where we simply apply a mask of the list of legal moves at the current state. The transition function is a deterministic function which indicates which discs will be flipped once an action has been played. The reward function for the positional agent is adapted slightly from the van Eck and van Wezel paper, and is given as the difference between the number of discs owned by player 1 minus those of player 2 if the game is in an end state, or 0 if we are not in an end state. We also implement a random starting strategy for the first three moves of the game. During the learning phase we implement an *Epsilon Greedy* policy to make the next move with a decay rate starting after the learning warm up phase.

3.3 Environment Agents

We are using an environment for the Othello game [1], containing one positional and eight different AI algorithms playing Othello:

- EvalBoard *(positional player)*: Returns the action giving the best next state, according to the EvalBoard() function (see section 3.4)
- Minimax: Minimising the maximum loss, a.k.a. maximising the minimum reward.
- Minimax w/ Alpha-Beta Pruning: Aims to decrease the number of potential moves evaluated

- Negamax: Version of Minimax, utilising the fact that:

$$Value(board, player1) = -Value(board, player2)$$

- Negamax w/ Alpha-Beta Pruning
- Negascout (Principal Variation Search): Extension of Negamax which aims to be faster than Negamax with Alpha-Beta Pruning by never considering nodes which wouldn't be considered by Alpha-Beta Pruning and further reducing the search space.
- Minimax w/ Alpha-Beta Pruning w/ Sorted Nodes: Prioritising high-valued moves when searching by making use of the GetSortedNodes() function (see section 3.4).
- Negamax w/ Alpha-Beta Pruning w/ Sorted Nodes
- Negascout (Principal Variation Search) w/ Sorted Nodes

3.4 Environment Functions

The environment provides us with a few native functions that we take advantage of when implementing our playing agent[1]. The environment has been converted into a `Board` class which maintains the current state of the game and handles all of the below functions.

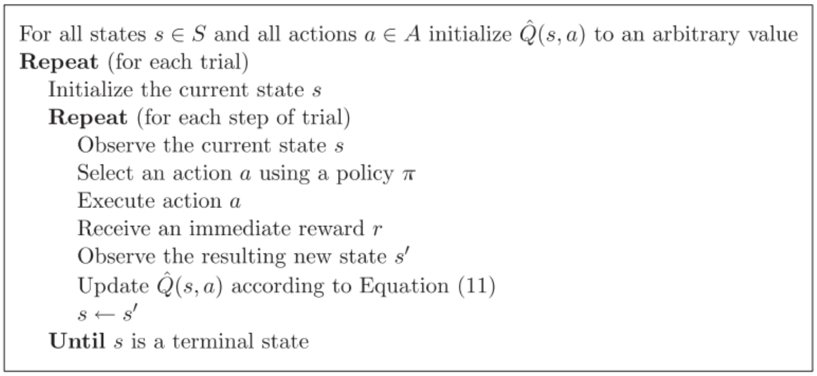
- `InitBoard()`: Initialises the board in Othello starting position.
- `PrintBoard()`: Prints the current state of the board.
- `MakeMove()`: Assuming a valid move is provided, the agent makes the move at the specified position. Returns the board in its updated state and the count of how many tiles were flipped in the move-makers favour.
- `ValidMove()`: Checks if the move provided is valid. Returns boolean result
- `EvalBoard()`: Naively calculates the value of the board assuming corners are worth 4 points, edges are worth 2 points, and interior points are worth one point. The function returns the total board score.
- `IsTerminalNode()`: If there are no valid moves, the function returns true and marks the end of the game.
- `GetSortedNodes()`: Plays all possible moves for the agent and calculates the value of the board for each such move. Then the results are sorted in decreasing order: Highest valued moves first.
- `BestMove()`: Returns the best move according to the indicated playing agent.

We added a few functions to the environment which were necessary for the training phase described in section 4.3.

- `set_board()`: Sets `Board.board` to the passed in board state.
- `board_to_numpy()`: Converts user friendly appearance board to numpy board.
- `count_board()`: Counts the number of discs the provided player has.

4 Description of the algorithms

For our experiments, we planned to implement the Q-learning algorithm, as specified in section 2.1 of [2]:



```
For all states $s \in S$ and all actions $a \in A$ initialize $\hat{Q}(s,a)$ to an arbitrary value
Repeat (for each trial)
    Initialize the current state $s$
    Repeat (for each step of trial)
        Observe the current state $s$
        Select an action $a$ using a policy $\pi$
        Execute action $a$
        Receive an immediate reward $r$
        Observe the resulting new state $s'$
        Update $\hat{Q}(s,a)$ according to Equation (11)
        $s \leftarrow s'$
    Until $s$ is a terminal state
```

Figure 2: Pseudo-code for Q-Learning algorithm

Where Equation (11) is:

$$\hat{Q}(s,a) \leftarrow (1-\alpha)\hat{Q}(s,a) + \alpha\left(r + \gamma \max_{a' \in A}\hat{Q}(s',a')\right) \tag{1}$$

However, Othello has a massive state space, which means that something needs to be done to address this. One such approach is the class of function approximators. In section 2.1 of van Eck and van Wezel [2], they outline an algorithm whose pseudo code can be seen here in figure 3.
For an explanation of the models designed for our experiments, please reference section 4.2. The action is then chosen based on a weighted probability which considers the benefit of exploiting our current knowledge or exploring a new action to gain more knowledge of different moves. We can decrease the exploration rate over time, so that as we gain more information, we are more likely to exploit what we know rather than to explore different actions.

In our case, we define two Neural Networks (NN), a *target* network and an

```
Initialize all neural network (NN) weights to small random numbers
**Repeat** (for each trial)
  Initialize the current state $s$
  **Repeat** (for each step of trial)
    Observe the current state $s$
    For all actions $a'$ in $s$ use the NN to compute $\hat{Q}(s, a')$
    Select an action $a$ using a policy $\pi$
    $Q^{\text{output}} \leftarrow \hat{Q}(s, a)$
    Execute action $a$
    Receive an immediate reward $r$
    Observe the resulting new state $s'$
    For all actions $a'$ in $s'$ use the NN to compute $\hat{Q}(s', a')$
    According to Equation (11) compute $Q^{\text{target}} \leftarrow \hat{Q}(s, a)$
    Adjust the NN by backpropagating the error $(Q^{\text{target}} - Q^{\text{output}})$
    $s \leftarrow s'$
  **Until** $s$ is a terminal state
```

Figure 3: "The Q-learning algorithm with a feedforward neural network as function approximator" [2]

output network. The *target* network maintains the current "best" approximation of the network and the *output* network is the on-policy network which is updated throughout the training process. The training cycle of our project closely follows the *Deep Q-Learning* script provided to us in the class [4] which bears resemblance to the algorithm proposed in figure 3

4.1 Policy

For both the self-play 4.4 and positional 4.5 learned models, we implemented an *Epsilon Greedy* action policy as described in [5].

For both models, we input the state of the game into the learning network for the forward pass and apply a legal action filter to consider only legal actions in the given state. We then take the *argmax* over the actions of the output and choose between this value and a random action from the list of legal actions with probability $[1 - \epsilon, \epsilon]$.

So, with the estimated best action $\mathcal{A}^* \triangleq \underset{a}{argmax}\ \hat{Q}(s, a)$ and $\mathcal{A}$ being defined as the uniform distribution of all other actions, our policy is given by

$$\hat{\pi}^*_\epsilon \triangleq (1 - \epsilon)\mathcal{A}^* + \epsilon\mathcal{A}$$

where ϵ denotes the exploration rate, deciding to which degree we should explore other actions than the one we assume to be the best.

4.2 Q-Network Design

Akin to the NN design in [2], we have an single hidden layer NN with both an input layer and an output layer of size 64x1. The state of the board is passed into the neural network where empty spaces take the value 0, spaces occupied by player 1 take the value 1, and spaces occupied by player 2 take the value 2. The hidden layer has 128 neurons, we did not try to optimize the size of the hidden layer as training times were longer than anticipated.

4.3 Training Phase

Our training phase starts with a warm-up of 1000 games, where our transition buffer is fed with enough game states to initiate our *target network* Q^{target}.

Following the warm-up phase, the agent is trained by playing a certain number of games against a given opponent. After every game, *the agent's Q-network* π is updated by looking back on its actions, and comparing them to the target network.

The computed loss (MSE) between the expected utility of our agent's actions and the "true" Q-values sampled from our target, is used to update both Q^{target} and π.

The update of π's parameters are handled by `torch.optim.Adam`, while the parameters of Q^{target} is updated by

$$Q^{target} = (1 - \tau)Q^{target} + \tau\pi,$$

where τ is the update value.

Our *Positional Learning Agent* is trained by playing against the (non-learning) *Positional Player*, while the *Self-Play Agent* is trained by playing its own accumulated Q-network.

4.4 Q-Network - Self-Play

To avoid dealing with the uncertainty of playing an unknown opponent, we implemented a self-learning agent. Here we define two networks, a policy network and a target network. The policy network is our online learning agent and the target network is the agent which we use to approximate our network. Both the policy network and the target network have the same structure as defined in section 4.2. We follow the training regiment defined in 4.3 and our agent plays as player 1 against the target network. That is to say, for each state of the board, it is either player 1's move, or player 2's move. On player 1's move we take our action from the *Epsilon Greedy* policy defined 4.1. Whereas, on player 2's move, we take our action from the target network's best move with no additional policy adjustments.

4.5 Q-Network - Positional Learning

We also implemented a positional learning player, which learns similarly to the self-playing agent described above. In place of the target network making player 2's move, we have implemented a positional player which makes use of the position value matrix from [2] provided below in figure 4.

	a	b	c	d	e	f	g	h	
1	100	-20	10	5	5	10	-20	100	1
2	-20	-50	-2	-2	-2	-2	-50	-20	2
3	10	-2	-1	-1	-1	-1	-2	10	3
4	5	-2	-1	-1	-1	-1	-2	5	4
5	5	-2	-1	-1	-1	-1	-2	5	5
6	10	-2	-1	-1	-1	-1	-2	10	6
7	-20	-50	-2	-2	-2	-2	-50	-20	7
8	100	-20	10	5	5	10	-20	100	8
	a	b	c	d	e	f	g	h	

Figure 4: Position Value Matrix as outlined in [2].

This takes advantage of some expert knowledge which defines corner squares to be more valuable and positions which give away corner squares to be less valuable. We hypothesized that this would lead to our agent learning the game faster. This will be discussed further in 5

5 Experimental Results

Unfortunately, the results of our experiments did not pan out the way we had expected them to. As detailed in the 'Future work' section of our mid-term report, we implemented a 4x4 Q-learning agent which yielded a promising training curve as seen in figure 5.

This lead to another promising training curve as seen in figure 7.

However, these promising results quickly ground to a halt upon the realization that the model was playing illegal moves and placing disks when searching for legal actions. This meant that the model would "find" legal moves when there were none, which obviously tilts the board in their favor.

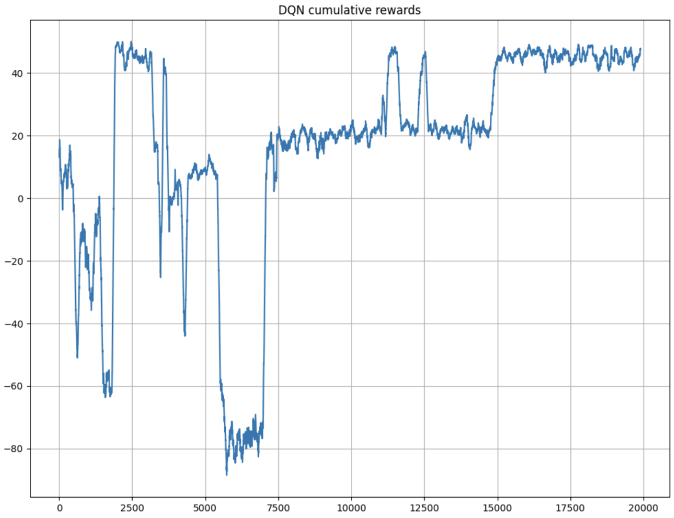


Figure 5: Running mean plot of a 4x4 learning curve.

After fixing this error, we found that the training had taken a turn towards the worst. Following dozens of retrainings, each taking up to three hours to complete, the model's training curves looked something like that in figure **??**.

We still tested these models against the two positional players as base-line quality metrics, here are the two winning proportion graphs in figure 8. As we can see from the graph from [1]'s positional player, there does appear to be a slight up-trend throughout the course of training. We have concluded that the positional player presented in [2] is a significantly stronger player than the positional player from [1]. This notion is indeed confirmed in the two plots presented in figure 8.

6 Future work

Based on our results, we believe that it would be beneficial to increase the number of training iterations. Looking at van Eck & van Wezel's project, they trained their agents on 15,000,000 games, compared to our 15,000. So this would be a natural place to start to try to improve our results.

Additionally, future experiments could involve tuning the hyperparameters in order to get a more accurate and stable model.

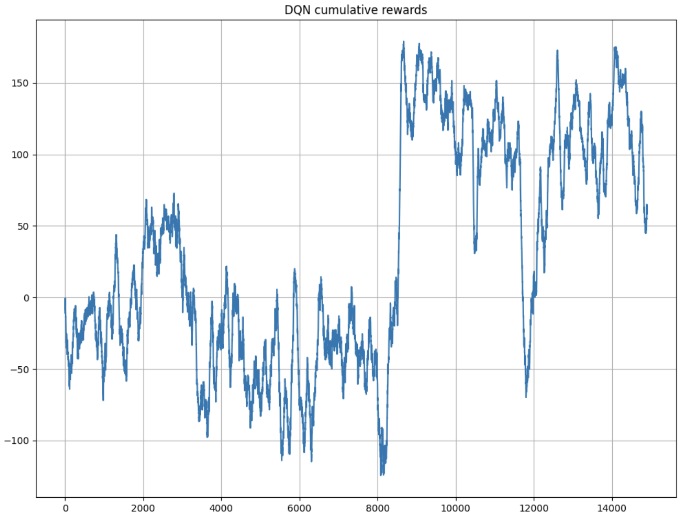


Figure 6: Running mean plot of an 8x8 learning curve.

Finally, we think testing different variations on the hidden layers - both in size and in numbers - could be an interesting follow-up of our work.

7 Conclusion

As discussed in our future work section, there are a number of changes which we believe could bring some stronger results to our project. Despite the slight appearance of learning presented and discussed in figure 8, it seems that our trained model is struggling to learn throughout the training phase. It suffices to say, that we would need to see the quality of a model with more training iterations before completely nixing the approach which we have taken. It is of course possible that there are some errors in our implementation, however it is often difficult to locate these small niggles in the code, especially after seeing a glimmer of hope. All in all, we have been able to produce an agent which learns with a function approximated Q-learning approach, albeit the resulting player is not as strong as we would like it to be.

Our code base is accessible in [6].

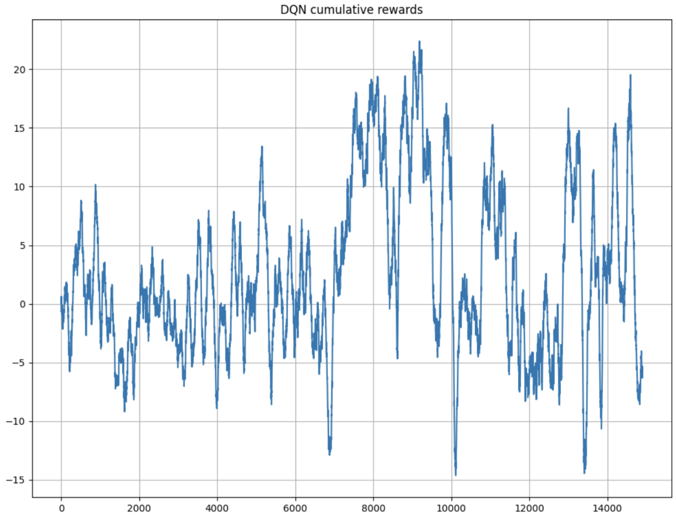


Figure 7: Current running mean plot of an 8x8 learning curve.

References

[1] FB36, "Reversi Othello ⟪ Python recipes ⟪ ActiveState Code," code.activestate.com, 2016, Accessed: Mar. 18, 2024. The MIT License. Copyright (c) 2010 ActiveState Software Inc. [Online]. Available: https://code.activestate.com/recipes/580698-reversi-othello/.

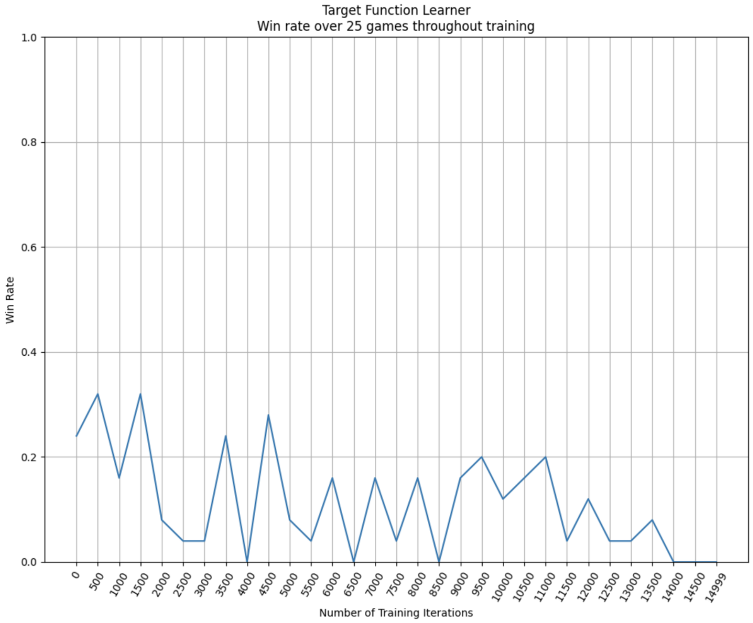
[2] N. J. van Eck and M. van Wezel, "Application of reinforcement learning to the game of Othello," Computers & Operations Research, vol. 35, no. 6, pp. 1999–2017, DOI: https://doi.org/10.1016/j.cor.2006.10.004. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=ceec5e39fdd0c8be0e6270c46488ccfe0254a6b6.

[3] "Othello Reversi Online: Play Reversi Online Free," playpager.com, Accessed: Mar. 18, 2024. [Online]. Available: https://playpager.com/othello-reversi/.

[4] C. Dimitrakakis and J. Tłuczek, "Deep Q Learning Jupyter Notebook," GitHub, [Online]. Available: https://github.com/olethrosdc/rldmuu/blob/main/src/RL/deep_rl/DQN.ipynb.

[5] C. Dimitrakakis and R. Ortner, "Decision Making Under Uncertainty and Reinforcement Learning,"

[6] M. Øgaard and T. Verheijen, "Othello Agent," GitHub, [Online]. Available: https://github.com/Ma10as00/othello_agent.



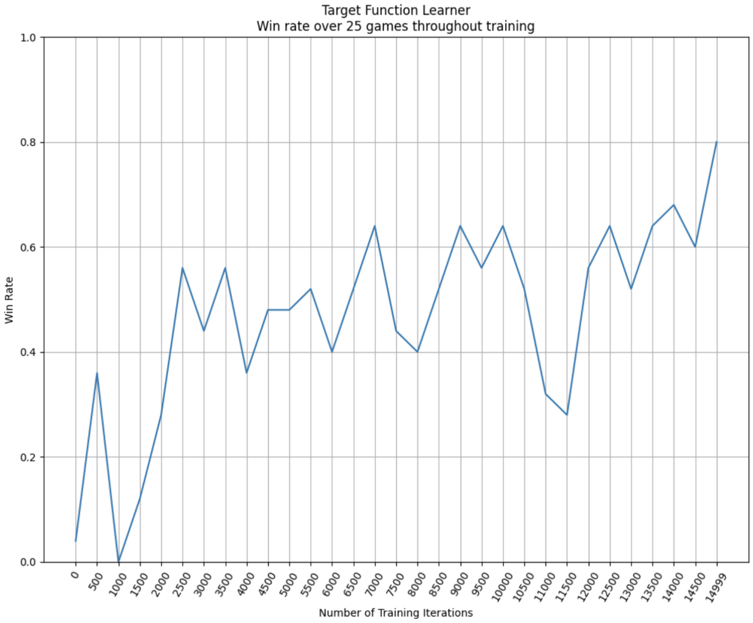


Figure 8: Win-rate graphs, (upper) positional player from [2], (lower) positional player from [1]