Ryan Vasios
Comp 135 Project Paper

Reinforcement Learning in the Game of Othello:

Q-learning with Neural Networks in Self-Play and Fixed-Play

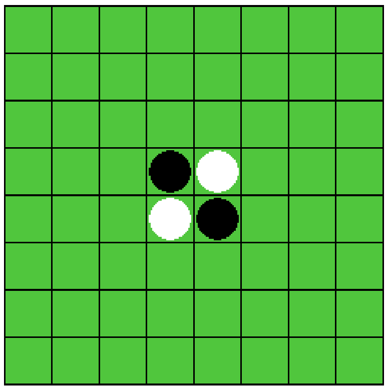
Intro:

While most of the methods of machine learning we have studied this semester have focused on supervised and unsupervised methods of learning, I have focused my project on the area of reinforcement learning and its application in developing artificial intelligences for 2 player board games. Particularly, I was curious to see how it could be applied to the game Othello, also known as Reversi. Fortunately there was already published material on this problem and its solutions through reinforcement learning. Those papers I found most useful demonstrated two distinct strategies. The first is learning through self play which consists of taking two malleable, learning agents that compete against and learn from one another in their interactions. The second method is learning from a fixed strategy. This method, on the other hand, involves only a single learning agent that adapts facing off against an adversary that executes a determined but effective strategy based on principles of the game.

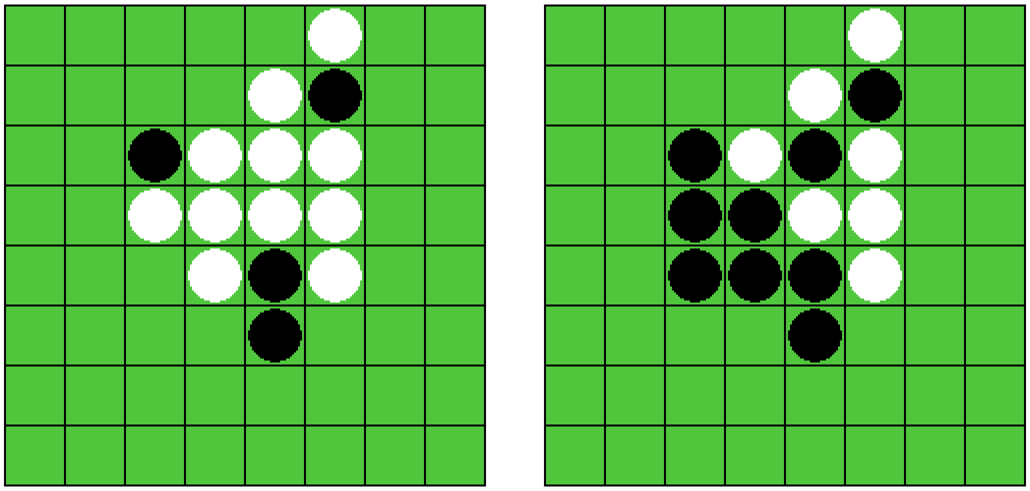
The Game:

In formal terms Othello is a two-player, deterministic zero-sum game. What this means is that the two players are in direct opposition such that one player's gain is another player's loss. The two players can not both benefit or both be hurt by the same board move. It is also deterministic in that a given move, performed in a given state maps to a single, distinct result. It is also worth noting that Othello is a perfect information game because the state of the game is disclosed fully to both players at all times in the form of the game board.

The rules of the game are simple. It is played on an 8x8 board, using 64 two-sided discs. Each disc is white on one side and black on another with each color corresponding to one of the two players. The board's starting state begins with a simple, centered, 4-disc square, consisting of two white and two black tiles



Beginning with black, players then alternate making valid moves on the board until neither play has any available moves. A game has at most 60 moves, corresponding to the 60 empty tiles at the start position. A legal move consists of placing a disc on an empty tile such that in at least one direction (horizontally, vertically, diagonally) from the played square, there is a sequence of one or more of the opponent's discs followed by one's own disc. The bounded opponent's discs in this sequence are then turned over. Below I've included a simple visual demonstration:

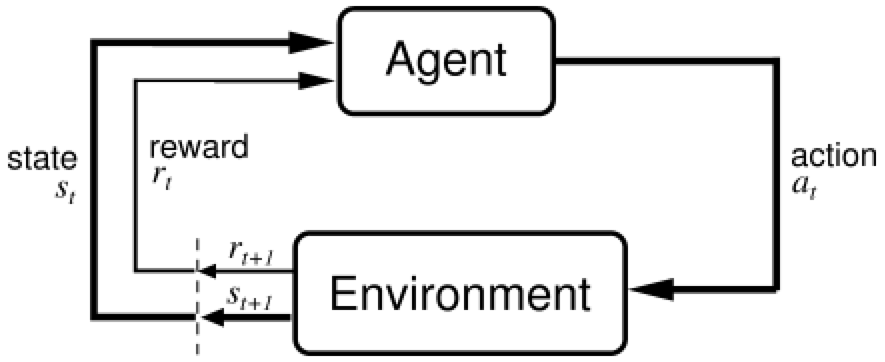


In this example, the black player has placed a disc in the cluster's corner of the third column. From this position, there are three sequences in three directions of white discs bounded by black discs that are then turned over. One sequence is vertical and above the placed disc. Another is horizontal and extends to the right of the placed disc. The last sequence is diagonal and extends above-right of the placed disc.

At the game's end the player with the most discs represented on the board is the winner.

Reinforcement Learning:

Studying and developing autonomous strategies for this game suggests an approach suited to the agent-environment interface of reinforcement learning. In this framework, a learner or decision maker interacts with a larger environment consisting of everything outside that player. The agent interacts with the environment over a series of discrete timesteps. At each timestep, ***t***, the agent receives some representation of the environment, ***s***. On the basis of this state, the agent will select an action, ***a***, from those actions available to the agent in this state. One time step later, the agent receives a reward, rt, and a new state, st+1. This series of interactions can be demonstrated with this common, cyclical diagram:



Another important reason that this problem is especially suited to reinforcement learning is its exhibition of the Markov property. Defined formally, the term 'Markov property' refers to the memoryless property of a stochastic process. In more intuitive and contextual terms, the state of our game at any point is analyzed and reacted to in isolation from its previous states. Many board configurations can be reached through multiple series of actions but are evaluated the same regardless of how we got there.

Because of this, the essential task of reinforcement learning is to learn a mapping from our set of possible states to the actions we would take in each one. This is also described as the policy of a game, **π** . The value of a policy is the expected rewards of adhering to its actions over time:

$$V(\boldsymbol{\pi}) = E\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \right].$$

In the above expression, the lambda represents the discount factor of future rewards. In reinforcement learning there is an important distinction between *continuous* and *episodic* learning tasks. With lambda set to a value less than one in continuous learning tasks, a learning agent is taught to value immediate and sooner rewards than those further down the line. In the maze solving problem for instance, an agent is seeking a shortest path and should depreciate the final reward based on how long it takes. Alternatively, in *episodic* learning tasks, agents receive rewards on the basis of an episode's completion and outcome no matter how long it takes. As mentioned, a game of Othello is at most 60 moves with a typical game have ~55+ moves. Even if there was a greater breadth of possible game lengths, players are still concerned solely with either winning or losing an entire game. This leads to two fundamental properties we must apply in modeling Othello as a reinforcement learning task. Firstly, our lambda is set to 1 since we only care about the final outcome of a game. Similarly, the second implication is that learning agents do not receive reward payouts until the end of the game.

Q-Learning:

Q-learning is a reinforcement learning algorithm that learns the value of a function Q(s,a) to find an optimal policy π^*. This Q-function is defined as the reward received upon immediately executing action a in state s plus the discounted value of the rewards obtained by continuing to follow the optimal policy, **V***.

$$Q(s,a) = E[r(s,a) + \gamma V^*(\sigma(s,a))]$$

It is easy to see that if the Q-function is known, an optimal policy is constructed by simply selecting the action that maximizes Q(s,a) in any state.

$$V^*(s) = \max_a Q(s,a)$$

Substituting this expression into the former definition of Q(s,a) we obtain the following recursive definition

$$Q(s,a) = E[r(s,a) + \gamma \max_{a'} Q(s', a')]$$

Our learning algorithm is derived from this definition of the Q-function wherein our agent iteratively approximates the Q-function. Per each iteration, the agent observes the current state, chooses an action, receives a reward and new state. It then updates its estimate of the approximated Q-function according to the rule:

$$Q(s,a) \leftarrow (1 - \alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s', a')\alpha$$

Here ή represents the learning rate of our algorithm or the rate at which observations and adjustments are incorporated.

Approximation with NN:
Q-learning offers us both benefits and detractions as a learning method. Unlike such methods as dynamic programming, we can attempt to learn our Markov Decision Process without having access to the reward function or state transition function. Our learning agent is not required to posses a model of its environment. However, our estimated Q-values need to be stored somewhere during the estimation process and afterwards. If we were to rely on the simplest form of storage we could use a standard lookup table for every state-action pair. Though a basic approach, the space-complexity required for this can be unfeasible. In Othello, the state space of the game is over 10^{28}, exceeding the capacities of any computer!
We can avoid the problem of large state-action spaces by seeking a function approximation method instead. The idea is that instead of storing the values for every state-action pair, but instead to store the Q-values as a function of the state and action.

Following the experiments of Eck and Wezel, we approximate our Q-function using a feedforward neural network. More specifically, our neural network is a multilevel perceptron consisting of an input layer, a hidden layer, and an output layer. Each node in a given layer is connected by a series of weighted edges to all nodes in the next layer. The input layer is where input data representing the game state is fed into the network. The input layer feeds into our hidden layer, which in turn feeds to the output layer.

Our input layer consists of 65 nodes corresponding to the 64 squares of the board along with 1 bias node. The inputs of correspond are either 1, 0, or -1 depending on whether a square is occupied by the player at hand, is empty, or is occupied by the opponent. The hidden layer consists of 44 tanh units that output to another 64 nodes again correspond to the squares of the board. Each of these output nodes corresponds to an action, a', with a value approximating Q(s,a'). We only compare the Q-values of those output nodes that represent valid board moves.

We then select a move based on some policy, and retain its approximated value as our 'output' Q-value. we receive a new board state resulting from both our move and that of the opponent. Given this new state, we compute what our 'target' Q-value is according to the aforementioned equation:

$$\mathbf{Q(s,a) \leftarrow (1 - \alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s', a')}$$

Finally, we backpropagate the error through our output node from the difference of our output and target Q-values, $(Q_{target} - Q_{output})$.

As mentioned earlier, r, the reward value, is set to zero until the end of the game whereupon it returns the value 1 for a win, -1 for a loss, or 0 for a draw. It is also worth pointing out that both our Q-learning algorithm and NN each have a learning rate of .1 in the following experiments. The initial weight values for our players' NNs are initialized to arbitrary values between -.1 and .1.

<u>Implementation and Modules:</u>

Modules used:

othello.py - contains class 'Othello_game_state'. This is the 'board' of a game that takes in and places moves. It also outputs the game state for our NN, prints the board, and scores games.

player.py - contains the class 'RL-player' which is the basis for our learning agent. Each RL-player contains their own NN which they use to evaluate, select, and learn from moves that they give back to board class.

rand.py- contain the random_player class which selects moves at random in games. Also contains the

ge.py- contains the functions used for training RL_players and

mob.py , pos.py, greedy.py - contains the classes used for the fixed players

NN.py- contains the class for our neural network. This was used courtesy of a Mr. Florian Muellerklein's open source implementation (see sources). Though there are many packages available, this was simple and compatible with the optimizing interpreter used.

Action selection: Like in many other reinforcement learning problems, there is a common tradeoff between exploration and exploitation. In my program, the player's probability of choosing a move is proportional to that move's estimated Q-value out of the sum Q-values of all available moves. This seemed like a practical and intuitive method of balancing these two values.

When competing for evaluations, Q-learning players switch to the method of choosing those actions with the highest Q-values. In other words, they no longer explore but only exploit what they 'know'.

It is also worth pointing out that since the 'fixed' players described below work in deterministic fashion, games played against them must be arbitrarily started several moves to diverse the outcome of their games.

PyPy: Reinforcement learning problems can take especially long period of times for computation. In the papers I read, players would play up to 15 million training games! Fortunately, most significant results occurred far before this point. In order to run my program effectively, I used the python interpreter, pypy to execute the code. It reduces runtime by approximately a thousandfold and made the following experiments feasible!

Experiments:

Experiment One: Learning through Self-Play:

In my initial series of experiments, I iteratively trained two players for two million games, extracting copies of their weights at 250,000 games, 1,000,000 games, and 2,000,000 games. For the sake of convenience, I will refer to these three copies as versions 1,2 and 3 for both black(B) trained player and white(W) trained player.

Our initial table shows the number of wins between a player and different time versions of itself over 4 rounds of 100 games. Players alternate their roles as the starting player in each game. Please remember though that the number of wins may not add to 100 because of draws.

CHART ONE: PLAYING AGAINST DIFFERENT TIME VERSIONS OF SELF

	B1/B2	B1/B3	B2/B3	W1/W2	W1/W3	W2/W3
R1	14-83	8-92	27-73	55-40	49-45	46-49
R2	25-70	10-90	25-75	53-42	49-49	42-53
R3	14-83	10-89	35-64	57-42	51-44	44-52
R4	28-68	7-92	24-70	55-38	47-43	44-50
Avg:	20-76	9-91	27-70	55-40	49-45	44-51

Observations: The black players shows vast improvements over prior versions of itself with win margins increasing with its training times. The white player on the other hand does not seem to improve or change too drastically. It is worth observing from chart three, that white (by sheer luck of its starting arbitrary weights) was originally a much better player than black. Because black's "environment" consisted of a superior player, its opportunities for more useful feedback were increased and probably prompted greater improvement.

CHART TWO: PLAYING AGAINST THE RANDOM PLAYER:
Players are assessed in their performance against the random player

Player	Random as black	Random as white
B1	30	40
B2	46	46
B3	61	63
W1	57	56
W2	55	56
W3	54	59

Observations: With the exception of the lower left quadrant, we can see that successive versions of our players on average fare better against the random player, suggesting that they are learning better strategies for playing the game.

CHART THREE: BLACK PLAYER VS WHITE PLAYER
Those players trained as black and white face off in their respective roles in a hundred games. Cells show number of black wins

	W1	W2	W3
B1	10	13	11
B2	18	23	29

B3	18	58	45

Observations: I noticed in comparing these players that the White player had significant advantages over all versions of the Black player. Despite this, it seems the the black player learned more effectively how to improve its performance over time. In all three match versions, the player significantly improved its performance with successive versions of itself.

Experiment Two: Learning through a Fixed-opponent:
In these series of experiments I trained learning agents from their interactions with 'fixed strategies'. In short it appears that agents learn much faster and effectively when trained against players that represent already cogent strategies. These strategies consist of:
The positional player: This player emphasizes the relative importance of different positions on the board. Positions such as edges and corners are highly valued while other positions more liable to being "flipped" have lower values. Corners are especially valued because they can never be reclaimed by the opponent. This player considers every available move and chooses the action that will result in the board with the highest value. This player consults an internal 2d array that scores each square on the board.
The mobility player: Another important strategy in Othello is attempting to limit the mobility of the opponent while maximizing one's own. Doing this, a player will always have access to more actions and more opportunities throughout gameplay. This player considers all possible actions and outcomes for its state and chooses that one that results in the greatest difference between its own mobility and that of the opponent.
The greedy player: The simplest of all 'fixed players' , the greedy player simply chooses whatever action will result in the greatest number of its own pieces on the board. This strategy is typically preferred during the end phases of the game but can also be exercised throughout a round as it does tend to incorporate some of the advantages of the positional and mobility player as it both accumulates more positions and could limit an opponent's mobility.
For all three players, I created learning agents that trained for 1,000,000 and 2,000,000 games with each. I will refer to these players as P1,P2,M1,M2,G1 and

G2 after the initials of their training opponent and the number of rounds they played. All players were trained as the ‘black’ player (ie- going first).

CHART FOUR: COMPARING TO SELF-LEARNING PLAYERS.
Lists the “fixed learners” wins out of a 100 games vs earlier players.
Self-learners move first

	B1	B2	B3	W1	W2	W3
M1	96	69	61	59	63	62
M2	89	54	36	21	37	43
P1	44	32	15	8	7	9
P2	55	47	17	8	18	14
G1	89	51	32	22	36	54
G2	91	54	31	22	35	56

Observations: This chart demonstrates a trend that persists in the other following experiments. This is that those fixed players that provide the best basis for reinforcement learning are the mobility player, followed by the greedy player, followed by the positional player. It also upholds the findings of the first series of experiments in that it shows how B1, B2 and B3 demonstrates a successive series of improvements in performance with the W players slightly depreciating. It is also worth observing that M1 outperforms M2 suggesting that in the additional 1,000,000 rounds of training between them, our agent ‘overfit’ its strategy to dealing with the mobility opponent.

CHART FIVE: PLAYING AGAINST THE RANDOM PLAYER

	Random as black	**Random as White**
M1	**76**	**66**
M2	**57**	**45**
P1	**15**	**14**

P2	**17**	**18**
G1	**52**	**53**
G2	**55**	**56**

Observations: Again we have the M players performing best on average (with M1 outperforming M2) followed by G's and P's.   Also would like to point out that the random player is surprisingly robust. The positional learner loses severely and consistently to the random player.

CHART SIX: COMPARING FIXED PLAYERS:
Demonstrates the number of wins for the row player out of 100 games. Row player moves first.

	M1	**M2**	**P1**	**P2**	**G1**	**G2**
M1	**X**	**65**	**84**	**77**	**62**	**59**
M2	**22**	**X**	**79**	**74**	**61**	**62**
P1	**17**	**17**	**X**	**42**	**19**	**18**
P2	**18**	**22**	**47**	**X**	**29**	**31**
G1	**30**	**38**	**82**	**69**	**X**	**45**
G2	**39**	**43**	**85**	**71**	**55**	**X**

Observations: appears to maintain the observed relative performance and ranking : m1>m2>= g2>g1p2 > p1

<u>**Conclusions/Future Work:**</u>

Self-learning agents show effective strategies can emerge through cross-play between themselves. In my trials, it appears that self-learners stand to improve their performance more quickly when they face off against superior players. Although the W players beat the B players extensively, the B players demonstrate drastic improvements in their performances over the trials.

Fixed players also provide a useful basis for learning agents to acquire effective strategies in Othello. It is clear, however, that certain strategies are easier for agents to learn quickly. Both the mobility and greedy strategies are more straightforward for a learning agent to learn from and don't have the extra layer of abstraction and evaluation that the positional player uses to select moves with its board matrix. Although the positionally trained players perform the worst, they do show steady increases in performance between P1 and P2 suggesting that if further training was conducted with several other million rounds then we could possibly obtain really great players and policies.

Provided the opportunity, I would like to see what results could emerge if the scale of training could be expanded for both series of learners. Time played a considerable constraint in conducting timely experiments even after the introduction of our optimizing interpreter with a million games between self-learners taking approximately 10 hours. I would also like to experiment more freely with tuning the parameters of both the Q-learning algorithm and our NN, possibly through an additional series of algorithms designed for this purpose.

Finally, I'm curious if self-play and fixed play can be hybridized for training learning agents. This would give them the ability to learn from specific strategies without overfitting by playing against other learners as well with more apparently random behavior. I imagine that this could be conducted with alternate training rounds between the two methods or alternate series of rounds.

Sources:

Michiel van der Ree .**Reinforcement learning in the game of Othello: Learning against a fixed opponent and learning from self-play.**Inst. of Artificial Intell. & Cognitive Eng., Univ. of Groningen, Groningen, Netherlands

- Introduced the two methods tested: self-play and fixed opponent learning

Nees Jan van Eck∗, Michiel van Wezel**,Application of reinforcement learning to the game of Othello .**Computers & Operations Research 35 (2008) 1999 – 2017

- Introduced NN as a form of Q-learning and described several fixed strategies I used

Sutton and Baro. **Reinforcement Learning: An Introduction**. MIT Press, 1991

- Laid groundwork in theory of MDP's and Reinforcement Learning

D. Moriarty and R. Miikkulainen, "Discovering complex othello strategies through evolutionary neural networks," Connection Science, vol. 7, no. 3, pp. 195-210, 1995.

- More examples of NN's as a basis for learning othello strategies

https://github.com/FlorianMuellerklein/Machine-Learning/blob/master/Old/BackPropagationNN.py (Code I used for the NN I implemented)