Reinforcement learning: Project 4

Florence Hugh, August Jonasson, Amanda Tisell

March 21, 2025

Introduction

In this project our goal is to train an agent to play the game Othello, which is a modern version of Reversi[1]. Othello is a turn-based strategy board game for two players played on an 8×8 grid. Players take turns placing one piece on an empty spot on the board with their assigned color. After a play is made, the pieces of the opponent's color that lie bounded by the current player's color are turned over. The game ends when the board is filled or when none of the players can make a move. The player who controls the most pieces at the end wins the game. Figure 1 shows an example of the first few moves in a game.

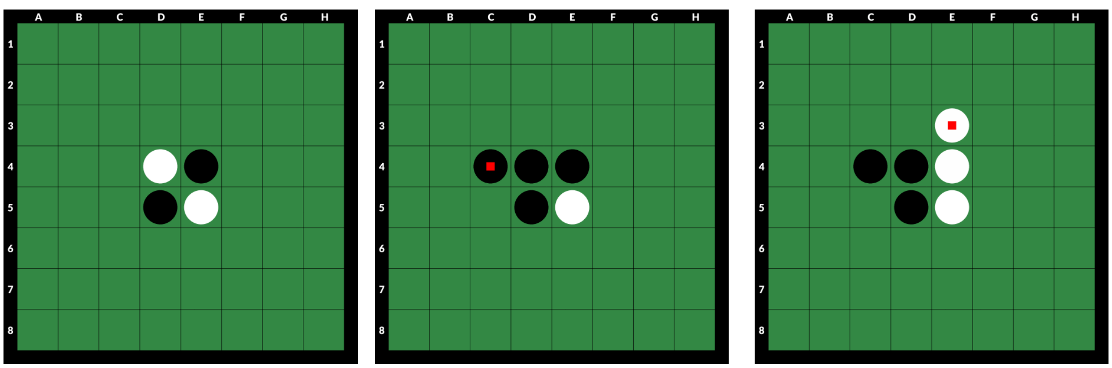


Figure 1: Starting state and first two moves of a game.

Some challenges and how we address them

Since Othello is a two-player game, we need to consider the agent/opponent setting. A natural idea would be to have one set of values for black and a completely different set of values for white, thus yielding two separate policies. When dealing with a two-player setting such as this, however, it turns out that we can avoid having to train two agents simultaneously by using self-play in combination with a minimax algorithm, thus allowing for both the black and white pieces to play against the same set of values.

Othello, similarly to Chess and AlphaGo, is unsolved and has a huge state space (around 10^{28} possible board positions on an 8×8 grid). Handling this size would require a large amount of compute which we do not have. Therefore, we decide to cut down the size of the board to a 6×6 grid instead. This results in a state space of around 10^{12}, which is still large enough to prevent accidentally brute-forcing the problem, but much more manageable within the scope of this project. A possibly even more important

[1] https://en.wikipedia.org/wiki/Reversi

advantage of this simplification is that 6×6 Othello is solved, with the white pieces having a clear advantage[2]. This means that it will now be much easier to assess the results of the training, since we now know what behavior to look for. Ideally, we would like to see the agent converge towards white winning every game.

As in most reinforcement learning systems, we have a setting of sparse rewards (only at the end of games) and the issue of balancing exploration/exploitation properly. We want to be able to learn meaningful patterns in a reasonable amount of time without having to go through the whole decision tree of possible games. Our attempt at addressing both of these problems (which is heavily inspired by the AlphaGo article Silver et al. 2016) is to use a Monte Carlo tree search algorithm in combination with a neural network-approximated policy that, simply put, favors deep over wide exploration such that rewards become more frequent, thus speeding up the learning.

MDP framing

The environment is a 6×6 grid where the dynamics are completely deterministic. The number of unique board positions in 6×6 Othello has been approximated to 10^{12} ($3^{32} \cdot 2^4 \approx 10^{16}$ and then divided by some number that accounts for what proportion of these actually represent legal board positions). A state will be represented as a 6×6 matrix with entries being

$$s_{ij} = \begin{cases} -1 & \text{if square } (i,j) \text{ is occupied by a white piece,} \\ 0 & \text{if square } (i,j) \text{ is not occupied by any piece,} \\ 1 & \text{if square } (i,j) \text{ is occupied by a black piece.} \end{cases} \tag{1}$$

The action space of size 32 includes all squares on the board, excluding the four starting pieces. However, most of these actions will be illegal. According to a paper that claims to have weakly solved Othello, there are on average 10 legal actions per move on a 8×8 grid (Takizawa 2023). This leads us to believe that a 6×6 grid may have on average 5-8 legal actions from a given state. An action a is represented as a tuple (i, j), which is the position where the piece is placed on the board.

A reward is only given at the end of each game and will be defined as

$$R(s,a) = \begin{cases} 1 & \text{if black wins the game} \\ -1 & \text{if white wins the game} \\ 0 & \text{if the game is drawn} \end{cases} \tag{2}$$

[2] https://en.wikipedia.org/wiki/Computer_Othello

Monte Carlo tree search (MCTS)

Monte Carlo tree search (Sutton, Barto, et al. 2018, ch. 8.11) is a type of decision-time planning algorithm based on Monte Carlo control applied to simulations starting from the root state, that is, a type of rollout algorithm.

Decision-time planning (Sutton, Barto, et al. 2018, ch. 8.8) is a type of planning that focuses on a particular state. That is, instead of simulating experience for the whole state space, it only plans ahead from the current state s_t. Using simulated experience to gradually improve a policy or value function or to select an action for the current state. The values and policy are specified to the current state and the action choices available there. In decision-time planning there is a tradeoff between better policies and the time that is needed to simulate enough trajectories to obtain good value estimates.

A rollout algorithm (Sutton, Barto, et al. 2018, ch. 8.10) is a decision-time planning algorithm based on Monte Carlo control applied to simulated trajectories that all begin in the current state. Each rollout produce Monte Carlo estimates of action-values for each current state and for a given policy, usually called the rollout policy. The aim of a rollout algorithm is to improve the rollout policy and not find the optimal policy like in regular Monte Carlo methods.

In MCTS, as in a rollout algorithm, each execution is an iterative process that simulates many trajectories, each trajectory starting from the current state s_t, running to a terminal state s_T. This iterative process consists of four steps; selection, expansion, rollout (simulation) and backpropagation.

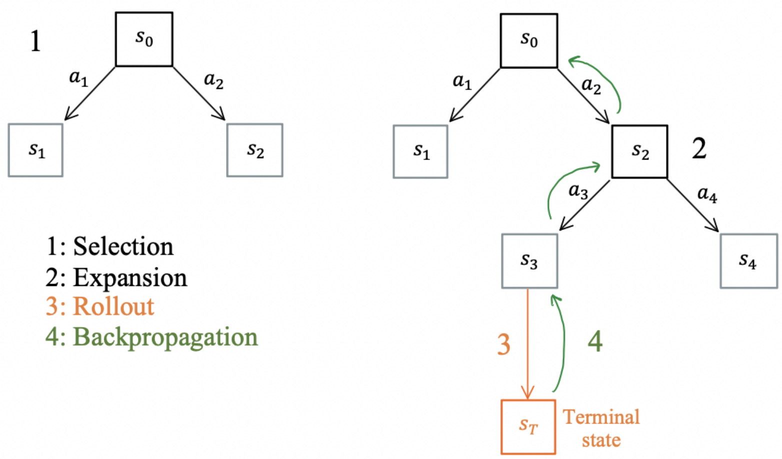


Figure 2: Monte Carlo tree search

These four steps continues to execute, starting each time at the tree's root node (initial state), until some criterion is fulfilled. This criterion can for instance be iterating for a fixed number of iterations (fixed number of terminal states reached with the tree policy).

1. Selection

In the selection part, a node (state) is chosen according to a tree policy. The tree policy balances exploration and exploitation, where it's action selection can for instance be based on ϵ-greedy or an upper confidence bound (UCB) selection formula

$$a_t = \arg\max_a \left(Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \right). \tag{3}$$

The $Q(s,a)$ is an average action-value (updated by the backpropagation, explained in part 4), c is a constant that controls the greediness of the action selection, $N(s)$ is the number of visits in current state s and $N(s,a)$ is the number of visits in the next state given action a.

The MCTS traverses down the decision tree and whenever a state is visited, the next action is selected among the possible actions from this current state according to equation 3. This process continues until a leaf node (unexplored and not terminal state) is reached in which an expansion is applied.

2. Expansion

The decision tree is expanded from the current state by connecting all possible actions to their respectively next state. From these new states one is selected based on the tree policy used in the selection step. However, all these newly added states will have a $N(s,a)$ count equal to zero. Since the MCTS priorities untried actions, one of these states will simply be picked randomly as a zero count will be seen as a a_t value of $\pm\infty$. Whenever a zero counted state-action is chosen we proceed to the next step, the rollout.

3. Rollout

During the rollout, a simulation is applied from the current state. This simulates a whole episode starting from the current state until a terminal state is reached. The actions selected during the simulation are generated using a rollout policy and does not necessarily follow the chosen tree policy. The rollout policy is usually a simpler policy allowing for faster computation. It could for example follow a policy where actions are chosen uniformly. When the terminal state is reached, a rollout reward R_r is received. The reward will be backpropagated and finally, all values generated by the rollout will be discarded.

4. Backpropagation

In the final phase, called the backpropagation, the reward observed from doing the rollout is backpropagated through the tree, updating all state-action values from where we started the rollout (the leaf) up to the root. The update rule is the in-place update of the average, given by

$$Q_{n+1}(s,a) \leftarrow Q_n(s,a) + \frac{1}{N(s,a)}\left[R_r - Q_n(s,a)\right], \tag{4}$$

(Sutton, Barto, et al. 2018, eq. (2.5)) where $Q_n(s,a)$ is the old state-action-value, $N(s,a)$ is the number of visits to state-action pair s,a, R_r is the reward yielded from the rollout, and $Q_{n+1}(s,a)$ is the updated value.

Modifying MCTS

In order to speed up the tree traversal and gain better control of the exploration (such that rewards become more frequent) we now make some modifications to regular MCTS.

First, to address the agent/opponent setting of the system, we solve this problem by using a minimax[3] algorithm in combination with self-play. As such, we reframe the system as an agent learning how to play Othello against itself, rather than an agent playing Othello against an opponent. The minimax algorithm is a decision rule strategy used in game theory in order to find optimal moves in zero-sum games (when a player's gain is balanced out by the other player's loss). The algorithm assumes that one player tries to maximize its score, while the opponent tries to minimize it. This fits well into our model as our reward structure results in a zero-sum game.

We have incorporated this algorithm into the Monte Carlo tree structure by adding additional information about which color's move it is at each state. By doing this the MCTS has an indicator of when to minimize/maximize the next state-action value. The action selection part of the tree will handle the minimax situation by using formula

$$a_t = \underset{a}{\overset{(\arg\min)}{\arg\max}} \left(Q(s,a) \underset{(-)}{+} c \frac{P(s,a)\sqrt{N(s)}}{N(s,a)} \right), \tag{5}$$

where the black (white) player will try to maximize (minimize) the return.

In addition, a policy $P(s,a)$ is added in order to make the agent more selective toward states that have already seen a lot of visits, thus pushing the actual exploration down along a few branches instead of higher up among many branches. This will lead to rewards being less sparse in the training, and as such, will speed up learning. The policy itself is retrieved through a neural network, which we will present in its own section.

Lastly, the rollout policy is replaced by function approximation of state-action values, as constantly letting the games play out to their end is too slow. Now, instead of playing them out, we simply input the leaf state into a function and get an approximation of the value back immediately. This function approximation is also handled by a neural network which we will present in its entirety in the next section. The backpropagation still remains as before. The only difference now is that instead of the actual reward, it is the approximated value that gets backpropagated, unless we have actually reached a terminal state - then the real reward is backpropagated.

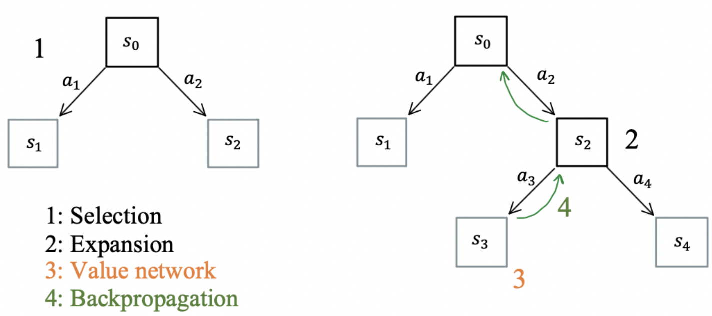


Figure 3: Modified Monte Carlo tree search

[3]https://en.wikipedia.org/wiki/Minimax#Minimax

Value approximation network

The function approximation of the action-values is handled by a deep convolutional neural network. In this section we will briefly present the architecture of the network and also explain how, and on what data, the initial training was done. For specific hyperparameter settings (such as batch-sizes, learning rates, etc.) we refer to the code appendix (line 749 in the code).

The architecture of the value neural network is heavily inspired by resnet-18 (He et al. 2016). Figure 4 depicts the network in its entirety.

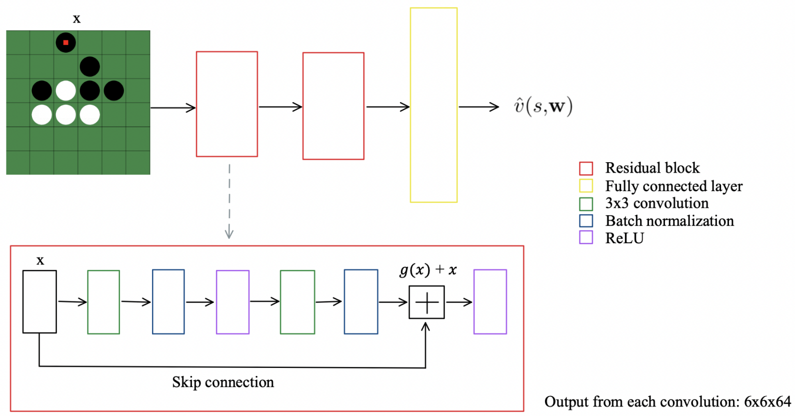
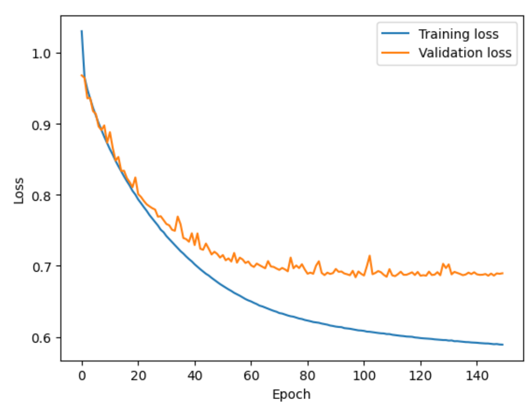


Figure 4: Value network architecture.

The input is a 6×6 matrix with elements according to (1) in the MDP framing, the targets are the Monte Carlo returns associated with the game from which that particular board position was retrieved, i.e. -1, 0 or 1, the output is the full-gradient action-value approximation, and finally, the loss function is the Mean Squared Value Error with $\mu(s)$ set to constant $1/|\text{data}|$ (Sutton, Barto, et al. 2018, p. 199).

The data used for the initial training of the value network consists of unique board positions retrieved from one hundred thousand randomly simulated games, with each one labeled according to the reward that was given at the end of its respective game. In total, this yields a dataset of size $\approx 2 \cdot 10^6$, which is then divided into a training dataset and a validation dataset according to a 80-20 split. Figure 5 depicts the training curve for 150 epochs. From the validation curve, we see no indication of overfitting, and the training curve seems to converge. As such, we deduce that the training is healthy. Looking at the actual loss, however, which is squared, we notice that for returns in the range $[-1, 1]$ it seems relatively large. This might be an indication that our network is not deep enough to successfully capture the complex relationships between board positions and returns. Given more time and resources, adding more residual blocks is a potential improvement for the project. See code appendix for the full `tensorflow` implementation.

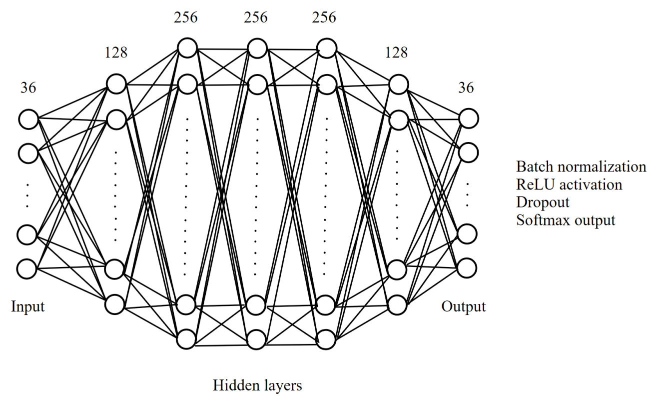
Figure 5: Initial training of the value network.



Policy approximation network

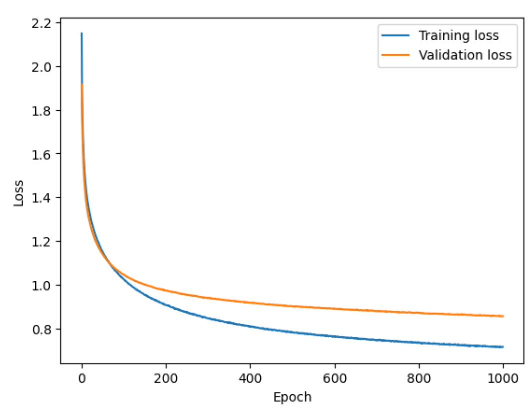
The policy approximation is handled by a plain, fully connected, feedforward neural network with linear inputs up until and including the output layer, which uses a softmax function in order to model the probability distribution over the legal actions in the given state. The architecture is summarized in Figure 6. Again, refer to the code appendix for specific hyperparameter settings (line 802 in the code).

Figure 6: Initial training of the value network.



Since this network only requires the states as a sort of identifier to know which actions are legal, the board position matrices can be flattened to a 36-dimensional vector before being input into the network, thus avoiding the need for convolutions to preserve more complex relations. Most likely, the states can be mapped to an even lower dimension and still act as unique identifiers. This is a potential improvement for the network, in terms of lowering the necessary complexity. The targets here are the normalized number of times each action had been taken from the given state in that particular tree, thus forming an approximate probability distribution over the actions. Since the number of actions can vary depending on the state, the output dimension is kept at a constant 36. This means that the target will be of this dimension as well, but with zeros on all positions that are not legal actions from the given state.

Figure 7: Initial training of the value network.



This means that the predicted probabilities will not form a perfect distribution over the legal actions, but the hope is that the bulk of the probability mass will still fall here, as the remaining actions are fitted to zeros. Dealing with this is another potential improvement. Finally, the Kullback-Leibler divergence[4] was used as loss function (minimization of distance between probability distributions).

The dataset used for the initial training of the policy network is a set of one thousand games simulated using the regular *upper-confidence bound* formula (eq. (3)) but with the value network instead of the rollout policy, thus forming a sort of intermediary step before transitioning completely to the model in Figure 3. Ideally, this is where we would want to use prerecorded "expert" experience, but due to not being able find any such databases for 6×6 Othello, we settle for this solution, and assume that the MCTS with the value network alone is sufficient. The training can be seen in Figure 7 and looks healthy. Since the targets represent unique state-action visits, we are ensured that there are no identical matches between the training and validation set. This indicates that the model generalizes well to unseen data.

[4] https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

Training cycle

With the modified MCTS, the value network and the policy network, we are now ready to define a *training cycle*, which is shown in Figure 8. Using the initially trained neural networks, we begin with simulating games from the Monte Carlo tree with a predefined number of episodes before termination. These games are then stored in a replay buffer which upon termination of the tree is sent into training of the neural networks for a number of epochs. Finally the updated networks are sent back to the tree, where a new step of the cycle can begin.

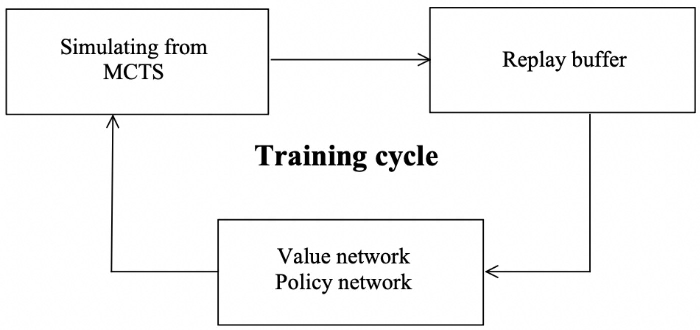


Figure 8: The agent training cycle.

In the actual implementation of the model, we let one tree iterate until it had reached terminal states 25 times, for 30 cycles. Even a single cycle often took as long as 30 minutes to complete due to the tree exploring along several different branches before reaching terminal states. In total, with the initial training of the value and policy networks included, approximately 15 hours was spent on training the model (on a CPU).

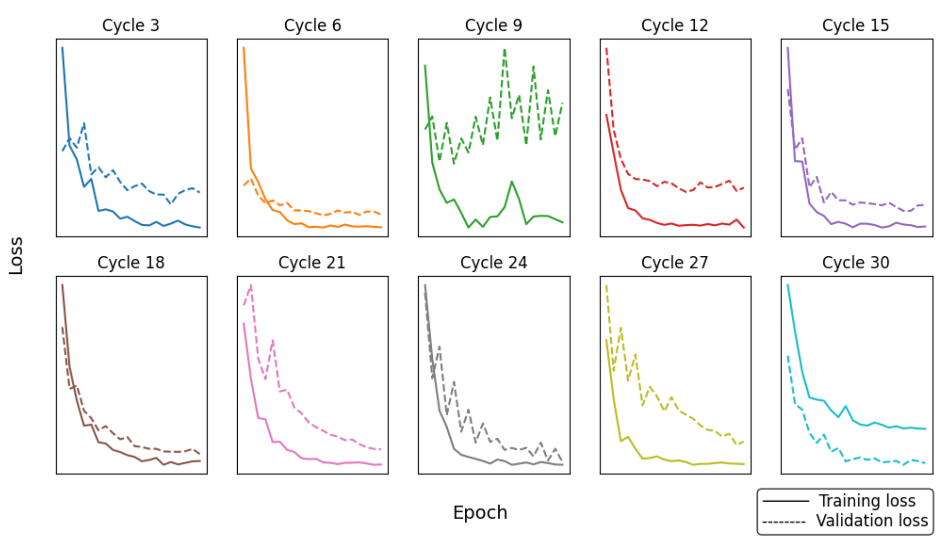


Figure 9: Value network training from ten selected cycles.

Figures 9 and 10 depict the training curves of the neural networks within the 30 training cycles. Overall the training looks very healthy in both networks. Cycle 9 within the value network has a validation

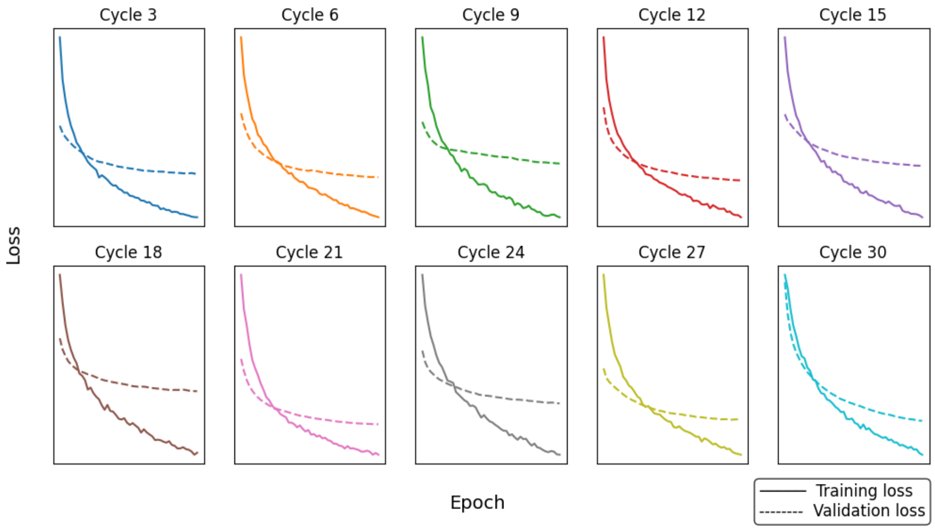


Figure 10: Policy network training from ten selected cycles.

curve that indicates a poor fit, which would be interesting to examine further. Cycle 30 from the value network has a validation curve that lies entirely below the training curve. This is actually because the training has converged such that the training and validation datasets are perfectly correlated. The regularization in the training is what causes the validation curve to fall below. In the policy network training we don't see any anomalies, and we can now move on to the results of the agent.

Results

Figure 11 shows the result after training the model on 30 training cycles with 25 episodes each. As seen in Figure 11, the white piece player becomes increasingly more dominant and, at the end, completely dominating. This is a promising result as 6×6 Othello is solved with the result that white has an advantage.

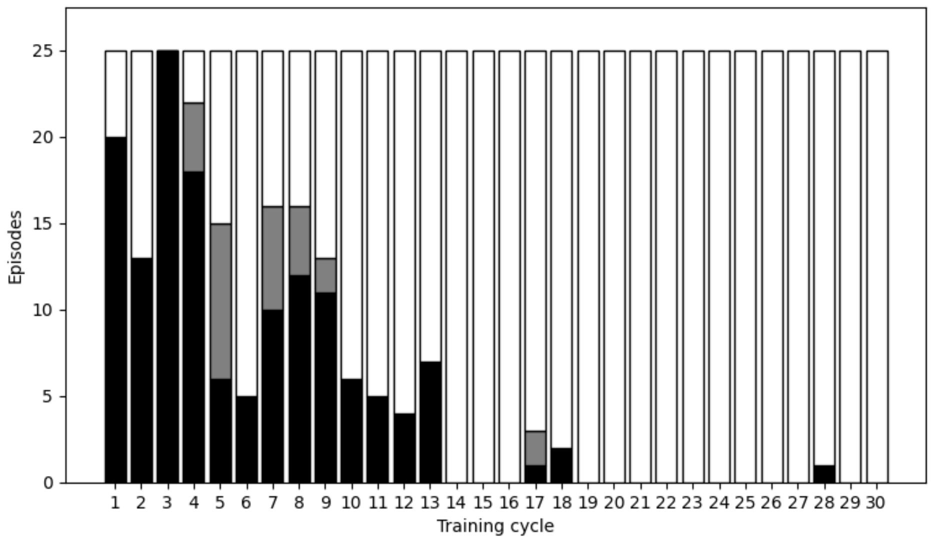


Figure 11: Reward distribution of each training cycle.

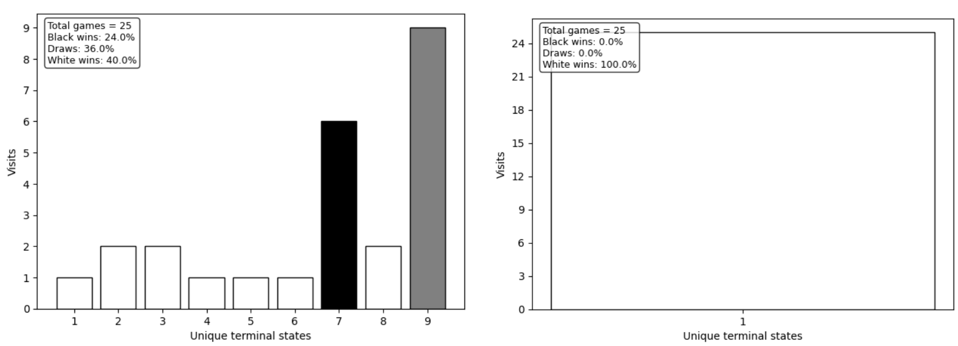


Figure 12: Terminal state distribution of cycle 5 (left) and cycle 30 (right).

Delving into some of the specific cycles now, we can see from Figure 12, for example, that even though white wins the first 6 episodes in the cycle, the fact that it wins in 6 different ways indicates that black is fighting back. Eventually, black even manages to get back with 6 wins in the same terminal state. Why white allows for this same terminal state to be reached so many times in a row could have something to do with the fact that white previously has gotten high rewards along this branch and therefore requires a couple of iterations before it realizes that better options exist.

Finally, though, which can be seen from Figure 12, white takes over completely and manages to force the game towards a single terminal state each game in the whole cycle. This happened in four other cycles towards the end as well, further strengthening the hypothesis that our training converged.

Discussion

The final result indicates that our agent has trained well. However, to gain more insight into the agents behavior, a deeper analysis of the trained decision tree is required. This involves examining the structure of the tree to find meaningful patterns, such as frequently visited paths and areas where the agent shows uncertainty or bias, and map out specific trajectories and compare them to solutions of 6×6 Othello. Since the game is solved, it would actually be possible to see if an optimal policy has been reached.

We mention that initially we worked on a 4×4 grid where a large amount of time was spent on implementing the methods. This turned out to be too simple as we ended up brute-forcing the whole tree, resulting in an optimal solution during the first cycle. Then we jumped straight to the 8×8 grid in which we realized we did not have enough computational power. Since our code was written for the general case, we seemlessly ended up with the 6×6 grid eventually.

Possible Improvements

Neural Networks

The policy network needs further improvements seeing how the probability mass of the output did not match well with the legal actions deeper down the tree. Meaning that in some cases, as little as 50 % of the probability mass was distributed to the legal actions. One way to solve it could be by adding a masking part which zeros out the illegal actions and weighted up the legal actions so they sum to one. There is probably a better way to improve this but we would need more time to figure it out.

Another improvement could be to simply train one neural network instead of having two separate ones. This network would then have as output both the actions and the values. By only training one network, we may speed up the training cycle slightly. It was, however, the tree traversal that ate up most of the training time, due to the size of the state space. So, any gains in the traversal would be even more valuable.

Parallel Processing

To train the tree, it needed to be traversed many times. Instead of just using one agent to traverse down the tree, we could have trained several agents simultaneously to enhance the process. There should also be a way for parallelizing the networks and the tree.

Handle Symmetry

We know for sure that the first four positions in which black can put its pieces are completely symmetric. With this knowledge, we could have fixed the first black piece to one position and then only trained that part of the tree to optimize it performance on one branch. Then the result in this branch could somehow map the other starting positions to equivalent actions.

Play against the agent

Ideally, it would be nice to be able to play with the agent. This way we could see if it actually wins all the games (given that it plays the white pieces) when playing against a human. We started building a graphical user interface (GUI) to play against the agent. However, because our agent had not explored enough state-action pairs, it could not make an action fast enough as it had to explore the tree from this unseen state. A way to solve this could be to either build a larger tree with the already trained weights from the networks (but this takes quite a time) or we could have added a time horizon threshold such that the agent only had a set amount of time to think. As it was implemented now, it spent a long time thinking (planning ahead) on each move which made the games very long. We are unfortunately not entirely sure how well it performs against a human, as of now.

References

He, Kaiming et al. (2016). “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Silver, David et al. (2016). “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587, pp. 484–489.

Sutton, Richard S, Andrew G Barto, et al. (2018). *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge.

Takizawa, Hiroki (2023). “Othello is solved”. In: *arXiv preprint arXiv:2310.19387*.

Appendix A: Python code

```
import copy
import numpy as np
import pickle
import random
import tensorflow as tf
from MCT_Othello_classes import *
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt
import matplotlib.ticker as mticker
import keras
from keras import layers, regularizers, Model
from othello_rl_helper_fcts import *

# 0=blank, 1=tot_black, -1=white
class OthelloBoard():
    dirx = [-1, 0, 1, -1, 1, -1, 0, 1]
    diry = [-1, -1, -1, 0, 0, 1, 1, 1]

    def __init__(self, n):
        self.n = n
        self.board = [[0 for _ in range(n)] for _ in range(n)]
        self.to_play = 1 #keep track of players turn 1=tot_black -1=white
        # self.pass_counter = 0
        self.reset_game()

    def reset_game(self):
        n = self.n
        # self.pass_counter = 0
        self.to_play = 1
        self.board = [[0 for _ in range(n)] for _ in range(n)]
        board = self.board

        # set up initial bricks
        z = (n - 2) // 2
        board[z][z] = -1
        board[n - 1 - z][z] = 1
        board[z][n - 1 - z] = 1
        board[n - 1 - z][n - 1 - z] = -1

        return board

    def print_board(self):
        n = self.n
        board = self.board
        m = len(str(n - 1))
        for y in range(n):
            row = ''
            for x in range(n):
                row += str(board[y][x])
                row += ' ' * m
            print(row + ' ' + str(y))
        print("")
        row = ''
        for x in range(n):
            row += str(x).zfill(m) + ' '
        print(row + '\n')

    def make_move(self, curr_state, action, to_play):
        x = action[0]
        y = action[1]
        if self.check_valid_move(curr_state, x, y, to_play):
            n = self.n
            bricks_taken = 0 # total number of opponent pieces taken
```

```
            curr_state[y][x] = to_play
            for d in range(len(self.dirx)): # 8 directions
                bricks = 0
                for i in range(n):
                    dx = x + self.dirx[d] * (i + 1)
                    dy = y + self.diry[d] * (i + 1)
                    if dx < 0 or dx > n - 1 or dy < 0 or dy > n - 1:
                        bricks = 0; break
                    elif curr_state[dy][dx] == to_play:
                        break
                    elif curr_state[dy][dx] == 0:
                        bricks = 0; break
                    else:
                        bricks += 1
                for i in range(bricks):
                    dx = x + self.dirx[d] * (i + 1)
                    dy = y + self.diry[d] * (i + 1)
                    curr_state[dy][dx] = to_play
                bricks_taken += bricks
            return (curr_state, bricks_taken)
        else:
            return print("Not valid move, retry")

    def check_valid_move(self, curr_state, x, y, to_play):
        """
        Function checks playable moves. First if the agent is within the board, then checks
        if the spot is occupied by a tot_black or white brick and finally, if the player do not
        take any of the opponents bricks, then it is not a legal move.
        """
        if x < 0 or x > self.n - 1 or y < 0 or y > self.n - 1:
            return False
        if curr_state[y][x] != 0:
            return False
        (_, totctr) = self._check_valid_move(copy.deepcopy(curr_state), x, y, to_play)
        if totctr == 0:
            return False
        return True

    def _check_valid_move(self, board, x, y, to_play):
        """
        Helper function to check_valid_move function to not overwrite the playing board if move is illegal
        """
        n = self.n
        bricks_taken = 0 # total number of opponent pieces taken

        board[y][x] = to_play
        for d in range(len(self.dirx)): # 8 directions
            bricks = 0
            for i in range(n):
                dx = x + self.dirx[d] * (i + 1)
                dy = y + self.diry[d] * (i + 1)
                if dx < 0 or dx > n - 1 or dy < 0 or dy > n - 1:
                    bricks = 0; break
                elif board[dy][dx] == to_play:
                    break
                elif board[dy][dx] == 0:
                    bricks = 0; break
                else:
                    bricks += 1
            for i in range(bricks):
                dx = x + self.dirx[d] * (i + 1)
```

```
                dy = y + self.diry[d] * (i + 1)
                board[dy][dx] = to_play
            bricks_taken += bricks
        return (board, bricks_taken)

    def move_generator(self, curr_state, to_play):
        possibleMoves = []
        for i in range(self.n):
            for j in range(self.n):
                if(self.check_valid_move(curr_state, i, j, to_play)):
                    possibleMoves.append((i, j))
        return possibleMoves

    def find_winner(self, curr_state):
        tot_black = 0
        tot_whites = 0

        for i in range(self.n):
            for j in range(self.n):
                if (curr_state[i][j] == -1):
                    tot_whites += 1
                elif (curr_state[i][j] == 1):
                    tot_black += 1

        if (tot_black == tot_whites):
            return 0
        elif (tot_black > tot_whites):
            return 1
        else:
            return -1



class MCTSNode(OthelloBoard):
    def __init__(self, n, state, to_play, parent=None, parent_action=None):
        super().__init__(n)
        self.terminal_visits = 0
        self.state = state
        self.parent = parent
        self.parent_action = parent_action
        self.child_nodes = []
        self._nof_visits = 0
        self.player_turn = to_play
        self.q_value = 0
        self.p_action = 0
        self.avg_q_value = 0
        self.pass_counter = 0
        self._untried_actions = self.untried_actions()
        self.all_visited = False


    def untried_actions(self):
        self._untried_actions = self.move_generator(self.state, self.player_turn)

        if len(self._untried_actions) == 0 and self.pass_counter != 2:
            self.pass_counter += 1
            self.player_turn *= -1
            self.untried_actions()

        return self._untried_actions

    def expand(self):
        action = self._untried_actions.pop()
        next_state, _ = self.make_move(copy.deepcopy(self.state), action, self.player_
    turn)
```

```
        next_player = self.player_turn*-1
        child_node = MCTSNode(self.n, next_state, next_player, parent=self, parent_action=action)
        self.child_nodes.append(child_node)
        if len(self._untried_actions) == 0:
            self.all_visited = True
        return child_node

    def update_q(self, val):
        self.q_value = val
        # return self.q_value

    # generalize this function such that it works for something
    def uniform_policy(self):
        """
        Initializes a policy uniformly over all legal actions.
        """
        nof_actions = len(self.child_nodes)
        return 1 / nof_actions

    def backpropagate(self, q_NN):
        # self.acum_q_value += q_NN
        self._nof_visits += 1
        self.avg_q_value += (q_NN - self.avg_q_value)/self._nof_visits # Q_{n+1}
        if self.parent: # Check if list is empty
            self.parent.backpropagate(q_NN)

    def best_child(self, c):
        """
        Minimax for training two agents
        """
        if self.player_turn == 1: # max
            UCB_values = [child.avg_q_value + c * child.p_action * np.sqrt(self._nof_visits) / child._nof_visits
                          for child in self.child_nodes]
            #  + c_paramnp.sqrt(np.log(self._nof_visits)/child._nof_visits)
            return self.child_nodes[np.argmax(UCB_values)]
        elif self.player_turn == -1: # min
            UCB_values = [child.avg_q_value - c * child.p_action * np.sqrt(self._nof_visits) / child._nof_visits
                          for child in self.child_nodes]
            #  - c_param*np.sqrt(np.log(self._nof_visits)/child._nof_visits)
            return self.child_nodes[np.argmin(UCB_values)]

def unpack_positions_returns(games):
    """
    Takes in games in the form of a list of tuples where the first element of the
    tuple is a list of board positions representing one full game, and the second
    element in the tuple is the reward (-1, 0, or 1) from that game.

    Returns a list of tuples where each tuple consists of a single board position
    and the reward associated with the game where that board position came from.
    This is a (state s_t, return G_t) tuple in RL terms.
    """
    data = []
    for game in games:
        for position in game[0]:
            data.append((position, game[1]))
    return data

def unique_board_positions(state_return_tuples):
    """
    Takes in list of tuples (state, return) where state is a board position and returns
    a list of only the unique tuples.
```

```
    This function will most likely be used only once, on the purely random simulated
    data.
    """
    unique_dict = {}
    for state, target in state_return_tuples:
        # Convert the matrix to a hashable representation using .tobytes()
        key = (state.tobytes(), target)

        # Only add unique tuples
        if key not in unique_dict:
            unique_dict[key] = (state, target)

    # Saving the unique (board, reward) tuples as data list
    return list(unique_dict.values())

def value_predictors_targets(state_return_tuples):
    """
    Takes in list of tuples (state, return) where state is board position and return
    is
    observed reward from game.

    And returns them in the form of X, y, ready to be used in a model.
    """
    # orders data into predictros and targets
    X = np.array([x for x, _ in state_return_tuples])
    y = np.array([y for _, y in state_return_tuples])

    # expanding X to include #channels=1
    X = np.expand_dims(X, axis=-1)

    return X, y

# Unpickling and saving the games to a list
def read_files(path):
    with open(path, "rb") as fp:
        boards = pickle.load(fp)
    return boards

def create_dataset(X, y, batch_size=128, shuffle_buffer_size=10000):
    """
    X and y should already be processed according to above preprocessing
    functions before being passed to this function.

    This function is simply to make training more efficient from memory.
    """
    # Create dataset from tensor slices
    dataset = tf.data.Dataset.from_tensor_slices((X, y))

    # Shuffle the dataset (reshuffles each epoch)
    dataset = dataset.shuffle(buffer_size=shuffle_buffer_size, reshuffle_each_
    iteration=True)

    # Batch the dataset
    dataset = dataset.batch(batch_size)

    # Prefetch for performance optimization
    dataset = dataset.prefetch(tf.data.AUTOTUNE)

    return dataset

def map_states_to_action_visits(sav):
    state_to_action_visits = {}
    for state, action, count in sav:
        key = (state.tobytes())
```

```
        if key not in state_to_action_visits:
            state_to_action_visits[key] = [(action, count)]
        else:
            state_to_action_visits[key].append((action, count))

    return state_to_action_visits

def map_actions_to_integers(n):
    actions = [(i,j) for i in range(n) for j in range(n)]
    map_actions = dict(zip(actions, [i for i in range(n*n)]))
    return map_actions


def policy_predictors_targets(sav, n):
    """
    Takes in a list of tuples (state, action, count) where state is board position,
    action is the action taken in this state and the count is the number of visits
    to the next state when taking the action.

    It should be the N(s,a) count from the UCB algorithm.

    The predictors are the states and the target will be the N(s,a) count
    """
    map_sav = map_states_to_action_visits(sav)
    mapped_actions = map_actions_to_integers(n)
    X = []
    y = []

    for state, value in map_sav.items():
        x = np.frombuffer(state, dtype=int)
        X.append(x)
        y_i = np.zeros(n*n)
        total_visits = 0
        for action, visits in value:
            total_visits += visits
            y_i[mapped_actions[action]] = visits

        y_i = y_i/total_visits
        y.append(y_i)

    X = np.array(X)
    y = np.array(y)

    return X, y

def episode(node):
    episode_list = [np.array(node.state)]
    if node.parent:
        episode_list.extend(episode(node.parent))
    return episode_list

def treetraversal(node, res):
    """
    Recursive helper function to state_action_visits.
    """
    if not node:
        return

    if node.parent != None:
        tup = (np.array(node.parent.state), node.parent_action, node._nof_visits)
        res.append(tup)

    for child in node.child_nodes:
        treetraversal(child, res)

```

```

def state_action_visits(root):
    """
    Given MCTS, recursively goes through the tree and returns a list with tuples
    containing state s, action a and N(s,a).
    """
    res = []
    treetraversal(root, res)
    return res


def results_distribution(episodes):
    """
    Given episodes, return a tuple with win, draw and loss.
    """
    win = 0
    loss = 0
    draw = 0

    for i in range(len(episodes)):
        if episodes[i][1] == -1:
            loss += 1

        elif episodes[i][1] == 0:
            draw += 1

        elif episodes[i][1] == 1:
            win += 1

    return (win, draw, loss)


def terminal_state_visits(episodes):
    """
    Given episodes, returns a dictionary with terminal state as key
    and number of visits in that terminal state as value.
    """
    term_state_visits = {}

    for i in range(len(episodes)):
        term_state = (episodes[i][0][0].tobytes(),)   # Make terminal state hashable
        term_state_visits[term_state] = term_state_visits.get(term_state, 0) + 1

    return term_state_visits


def bar_plot_term_states(episodes):
    """
    Given episodes, plot the distribution over how often a terminal state is visited.
    """
    win, draw, loss = results_distribution(episodes)
    tot_games = len(episodes)
    win_proc = np.round(win/tot_games*100, decimals=4)
    draw_proc = np.round(draw/tot_games*100, decimals=4)
    loss_proc = np.round(loss/tot_games*100, decimals=4)

    term_state_visits = terminal_state_visits(episodes)
    x_vals = [i for i in range(len(term_state_visits))]
    plt.bar(x_vals, term_state_visits.values())

    # Add a text box with game results
    textstr = f"Total games = {tot_games}\nWin: {win_proc}%\nDraw: {draw_proc}%\nLoss:
     {loss_proc}%"

    # Position the text box
```

```
    props = dict(boxstyle='round', facecolor='white', alpha=0.8)
    plt.text(0.68, 0.9, textstr, transform=plt.gca().transAxes, fontsize=9,
            verticalalignment='top', bbox=props)

    plt.xlabel("Terminal states")
    plt.ylabel("Number of visits")
    plt.grid()


def online_value_training(value_model, replay_buffer, epochs=20, batch_size=64):
    """
    Given a replay buffer in the form of list of tuples, (where each tuple consists of
    a list of board positions from one game, and the reward from said game) and a value model,
    trains this model using the replay buffer through given number of epochs.

    Returns the trained value_model
    """
    state_return_tuples = unpack_positions_returns(replay_buffer)
    X_buffer, y_buffer = value_predictors_targets(state_return_tuples)
    X_train_onl, X_test_onl, y_train_onl, y_test_onl = train_test_split(X_buffer, y_buffer,
                                                                        test_size=0.2,
                                                                        random_state=80085)

    train_dataset = create_dataset(X_train_onl, y_train_onl)
    val_dataset = create_dataset(X_test_onl, y_test_onl)

    history_onl = value_model.fit(
        train_dataset,
        batch_size=batch_size,
        epochs=epochs,
        validation_data=val_dataset
    )

    return value_model, history_onl

def online_policy_training(policy_model, tree, epochs=20, n=6, batch_size=64):
    """
    Given a replay buffer in the form of list of state-actions and their visits and a policy model,
    trains this model using the replay buffer through given number of epochs.

    Returns the trained policy_model
    """
    savs = state_action_visits(tree)
    X_buffer, y_buffer = policy_predictors_targets(savs, n)
    X_train_onl, X_test_onl, y_train_onl, y_test_onl = train_test_split(X_buffer, y_buffer,
                                                                        test_size=0.2,
                                                                        random_state=80085)

    train_dataset = create_dataset(X_train_onl, y_train_onl)
    val_dataset = create_dataset(X_test_onl, y_test_onl)

    history_onl = policy_model.fit(
        train_dataset,
        batch_size=batch_size,
        epochs=epochs,
        validation_data=val_dataset
    )

    return policy_model, history_onl
```

```python

def simulate_random_games(n):
    game_boards = []
    game = OthelloBoard(n)

    while True:
        moves = game.move_generator()
        if moves == []:
            game.pass_counter += 1
            game.change_turn()

        else:
            game.pass_counter = 0 # reset counter
            action = random.choice(moves)
            board = game.make_move(action)[0]
            game_boards.append(np.array(board))
            game.change_turn()

        if game.pass_counter == 2:
            reward = game.find_winner()
            break

    return (game_boards, reward)

def save_games(n, nr_simulations):
    all_game_boards = []

    for _ in range(nr_simulations):
        boards, actions = simulate_random_games(n)
        all_game_boards.append(boards)

    with open(f"othello_sim_boards_{nr_simulations}", "wb") as fp:   #Pickling
        pickle.dump(all_game_boards, fp)

def build_tree(n, value_model, policy_model, c, nr_simulations=1000):

    game = OthelloBoard(n)
    root = MCTSNode(n, game.board, game.to_play)
    episodes = []
    mapped_actions = map_actions_to_integers(n)

    while True:
        terminal_state, reward = expand_tree_iteratively(root, value_model, policy_model, mapped_actions, c)
        episodes.append((episode(terminal_state)[:-1], reward))
        print('Episode done!')

        if len(episodes) == nr_simulations:
            break

    return root, episodes

def expand_tree_iteratively(root, value_model, policy_model, mapped_actions, c):
    stack = [(root, root)]  # Stack holds pairs of (root, root)

    while stack:
        current_root, current_node = stack.pop()  # Get the last node to process

        if current_node._untried_actions:  # If the node has untried actions, expand it
            next_node = current_node.expand()
            # Instead of using recursion, use a random value for the q_val
            q_val = np.array(value_model(np.expand_dims(np.array(next_node.state), axis=(0, -1)))[0][0])
            next_node.update_q(q_val)
```

```
            next_node.backpropagate(next_node.q_value)

            # Push the root back into the stack to continue processing from the root
            stack.append((current_root, current_root))

        else:  # If no untried actions, move to the best child or terminal node
            if current_node.pass_counter != 2:

                flatten_state = np.ndarray.flatten(np.array(current_node.state))
                policy_dist = np.array(policy_model(np.expand_dims(flatten_state, axis
    =0))[0])
                for child in current_node.child_nodes:
                    child.p_action = policy_dist[mapped_actions[child.parent_action]]

                best_child = current_node.best_child(c)
                stack.append((current_root, best_child))  # Continue with the best
    child

            else:  # Terminal node/state
                reward = current_node.find_winner(current_node.state)
                current_node.update_q(reward)
                current_node.backpropagate(reward)
                current_node.terminal_visits += 1
                return current_node, reward

def unpack_plot_history(history_list):
    for i, history in enumerate(history_list):
        if history != []:
            label = f"C{i}"
            plt.plot(history.history['loss'], label=f'Training loss', color = label)
            plt.plot(history.history['val_loss'], linestyle="dashed", label = '
    Validation loss', color = label)

    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend(loc='upper right')
    plt.show()

def cycle_results_histogram(episodes_list):
    """
    Takes in an ordered (from first to last training cycle) episodes list where the
    length of the
    list denotes the number of training cycles that were needed to create it (each
    element in the
    list should be a list of episodes).

    In return, plots a histogram with sections colored according to the number of
    white wins, black
    wins, and draws (grey).

    As white should win at 6x6 othello, we expect to see the bars further to the right
     to be
    increasingly dominated by the white sections.
    """
    results = []
    for episodes in episodes_list:
        results.append(results_distribution(episodes))

    # Generate x-axis indices for the number of tuples in the list
    x = range(len(results))
    labels = list(range(1, len(results) + 1))

    # Unpack each tuple into separate segments
    segment1 = [t[0] for t in results]  # Black section
    segment2 = [t[1] for t in results]  # Grey section
```

```
    segment3 = [t[2] for t in results]  # White section

    plt.figure(figsize=(11, 5))

    # Plot the first segment (black)
    plt.bar(x, segment1, color='black', edgecolor='black')

    # Plot the second segment (grey), stacked on top of the first
    plt.bar(x, segment2, bottom=segment1, color='grey', edgecolor='black')

    # Compute the bottom for the third segment (segment1 + segment2)
    bottom3 = [a + b for a, b in zip(segment1, segment2)]

    # Plot the third segment (white)
    plt.bar(x, segment3, bottom=bottom3, color='white', edgecolor='black')

    # # Add a text annotation above each bar showing the white percentage.
    # for i, t in enumerate(results):
    #     total = sum(t)
    #     # Calculate the white percentage (t[2] is the white segment)
    #     white_percentage = (t[2] / total) * 100 if total else 0
    #     # Use .3g to format the number with a maximum of 3 significant digits
    #     plt.text(i, total + 1, f'{white_percentage:.3g}%', ha='center', va='bottom',
     fontsize=5)

    plt.xlabel('Training cycle')
    plt.ylabel('Episodes')
    plt.ylim(0, max([sum(t) for t in results]) * 1.1)  # Slightly higher than max for
    visual clarity
    plt.xticks(x, labels)
    plt.show()

def terminal_visits_histogram_colored(episodes):
    """
    Takes in a list of episodes and returns a histogram with as many bars as
    unique terminal states that were visited in those episodes, with each bar
    colored according to who wins (or draw) in that position. White bars for
    white wins, black bars for black wins, and grey for draws.
    """

    nr_games = len(episodes)
    win, draw, loss = results_distribution(episodes)
    w_rate, d_rate, l_rate = np.round(np.array([win, draw, loss])*(100/nr_games),
    decimals=0)

    terminal_states = {}
    for episode in episodes:
        key = (episode[0][0].tobytes(), episode[1])
        if key not in terminal_states:
            terminal_states[key] = 1
        else:
            terminal_states[key] += 1

    color_mapping = {
        -1: 'white',  # white for -1
        0: 'grey',   # grey for 0
        1: 'black'   # black for 1
    }

    # Extract rewards, visits, and colors
    identifiers = []
    heights = []
    colors = []
    for (identifier, y_value), height in terminal_states.items():
        if isinstance(identifier, bytes):
```

```python
            identifier = identifier.decode('latin-1')
        identifiers.append(identifier)
        heights.append(height)
        colors.append(color_mapping[y_value])

    x = range(len(terminal_states))

    plt.figure(figsize=(7.2, 5))
    plt.bar(identifiers, heights, color=colors, edgecolor='black')

    # Add a text box with game results
    textstr = f"Total games = {nr_games}\nBlack wins: {w_rate}%\nDraws: {d_rate}%\
    nWhite wins: {l_rate}%"

    # Position the text box
    props = dict(boxstyle='round', facecolor='white', alpha=0.8)
    plt.text(0.025, 0.975, textstr, transform=plt.gca().transAxes, fontsize=9,
            verticalalignment='top', bbox=props)


    plt.xticks(x, [str(i + 1) for i in x])

    plt.gca().yaxis.set_major_locator(mticker.MaxNLocator(integer=True))

    plt.xlabel('Unique terminal states')
    plt.ylabel('Visits')

    plt.show()

def remove_empty_lists(list):
    # remove all empty lists from the list
    list = [x for x in list if x]
    return list

def unpack_plot_train_val_curves(history_list, cycle_labels):
    """
    All empty lists have to removed from the input list here in order for it to work.
    See above helper function.

    Cycle_labels should be a list of of labels for the cycles, i.e. [3,6,9,...] if
    every third cycle is passed in the history list
    """

    fig = plt.figure(figsize=(12, 6))

    for i, history in enumerate(history_list):
        plt.subplot(2,5,i+1)
        color = f"C{i}"

        plt.plot(history.history['loss'], label=f'Training loss', color = color)
        plt.plot(history.history['val_loss'], linestyle="dashed", label = 'Validation
    loss', color = color)
        plt.title(f'Cycle {cycle_labels[i]}')
        plt.tick_params(axis='x', which='both', bottom=False,
            top=False, labelbottom=False)
        plt.tick_params(axis='y', which='both', right=False,
            left=False, labelleft=False)

    textstr = '\u2500' * 5 + '  Training loss \n--------  Validation loss'
    props = dict(boxstyle='round', facecolor='white', alpha=0.8)
    plt.text(-0.125, -0.10, textstr, transform=plt.gca().transAxes, fontsize=12,
            verticalalignment='top', bbox=props)

    fig.text(0.5, 0.04, 'Epoch', va='center', ha='center', fontsize=14)
```

```
    fig.text(0.09, 0.5, 'Loss', va='center', ha='center', rotation='vertical', fontsize=14)
    plt.show()

###########################################################################
# ------------------ Beginning of VALUE NN architecture ------------------
###########################################################################

def residual_block(x, channels=64, kernel_size=(3,3), weight_decay=0.001):
    shortcut = x  # No projection needed if dimensions already match.

    x = layers.Conv2D(channels, kernel_size=kernel_size,
                      padding='same', use_bias=False,
                      kernel_regularizer=regularizers.l2(weight_decay))(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    x = layers.Conv2D(channels, kernel_size=kernel_size,
                      padding='same', use_bias=False,
                      kernel_regularizer=regularizers.l2(weight_decay))(x)
    x = layers.BatchNormalization()(x)

    # Direct addition is fine here.
    x = layers.Add()([x, shortcut])
    x = layers.ReLU()(x)

    return x

# Build the model
inputs = layers.Input(shape=(6, 6, 1)) # 6x6 Othello

# Initial convolution block (producing 16 channels)
x = layers.Conv2D(64, kernel_size=(3,3),
                  padding='same', use_bias=False,
                  kernel_regularizer=regularizers.l2(0.001))(inputs)
x = layers.BatchNormalization()(x)
x = layers.ReLU()(x)

# Add a number of residual blocks with constant 64 channels
x = residual_block(x, channels=64, kernel_size=(3,3), weight_decay=0.001)
x = residual_block(x, channels=64, kernel_size=(3,3), weight_decay=0.001)

# Flatten the features and output a single scalar value (for a value network)
x = layers.Flatten()(x)
outputs = layers.Dense(1, name="value_output")(x)

# Create the model
value_model = Model(inputs=inputs, outputs=outputs)
value_model.summary()

# Optimizer for the value model
value_model.compile(
    optimizer=keras.optimizers.SGD(learning_rate=1e-4, momentum=0.9),
    loss = keras.losses.MeanSquaredError()
    # metrics = [keras.metrics.RootMeanSquaredError]
)

##########################################################################
#  ---------------- Beginning of Policy NN architecture ------------------
##########################################################################

# Policy model architecture
policy_model = keras.Sequential([

    layers.Dense(128, input_shape=(36,)),
```

```
    layers.BatchNormalization(),
    layers.Activation("relu"),
    layers.Dropout(0.025),

    layers.Dense(256),
    layers.BatchNormalization(),
    layers.Activation("relu"),
    layers.Dropout(0.025),

    layers.Dense(256),
    layers.BatchNormalization(),
    layers.Activation('relu'),
    layers.Dropout(0.025),

    layers.Dense(256),
    layers.BatchNormalization(),
    layers.Activation("relu"),
    layers.Dropout(0.025),

    layers.Dense(128),
    layers.BatchNormalization(),
    layers.Activation('relu'),
    layers.Dropout(0.025),

    layers.Dense(36),
    layers.Activation('softmax')
])

policy_model.summary()

policy_model.compile(optimizer=keras.optimizers.SGD(learning_rate=1e-3, momentum=0.9),
                    loss=keras.losses.KLDivergence())

# %%
# Loading the model weights
# Load the model weights
# value_model.load_weights('zeroth_value_nn.weights.h5')
# policy_model.load_weights('zeroth_policy_nn.weights.h5')


###########################################################################
# ----------------------- Online training section -----------------------
###########################################################################

"""
Given a replay buffer, we want to be able to continously feed new game
information into the value network in the form of mini batches. This
section aims to prepare for thats.

Replay buffer will be in the form of a list of tuples, where the first
elements in each tuple is a game consisting of a sequence of board positions
and the second element is the observed reward from that game.

Prerequisites for the following function is to already have pre-trained
value and policy networks.
"""

# %%
def online_simulation(n, c, value_model, policy_model, nr_cycles, nr_episodes_per_tree
     = 160, history_val = [], history_pol = []):
    ''' hehiha '''

    trees = []
    episodes = []

```

```
    value_model_history = [history_val]
    policy_model_history = [history_pol]

    for _ in range(nr_cycles):

        ##### Build tree with updated model #####
        tree, replay_buffer = build_tree(n, value_model, policy_model, c, nr_episodes_per_tree)

        trees.append(tree)
        episodes.append(replay_buffer)

        ##### Update value network #####
        value_model, value_history_temp = online_value_training(value_model, replay_buffer,
                                                                 epochs=20, batch_size=64)
        policy_model, policy_history_temp = online_policy_training(policy_model, tree,
                                                                 epochs=50, batch_size=128)
        value_model_history.append(value_history_temp)
        policy_model_history.append(policy_history_temp)

        print('Cycle done!')


    return trees, episodes, value_model_history, policy_model_history


# %%
trees, episodes, val_hist, pol_hist = online_simulation(n=6, c=0.5,
                                                        value_model=value_model,
                                                        policy_model=policy_model,
                                                        nr_cycles=10,
                                                        nr_episodes_per_tree=25)

# %%
# saving the training from the online training
second_10cycles25episodes_c05_data = [trees, episodes, val_hist, pol_hist]
with open('second_10cycles25episodes_c05_data', 'wb') as handle:
    pickle.dump(second_10cycles25episodes_c05_data,
                handle, protocol=pickle.HIGHEST_PROTOCOL)

value_model.save_weights('value_second_10cycles25episodes_c05.weights.h5')
policy_model.save_weights('policy_second_10cycles25episodes_c05.weights.h5')

##########################################################################
# ------------------- Initial training of VALUE NN ----------------------
##########################################################################

# Initial value network data - from completely random games
# Unique board positions from first 100k random games
path = './othello_random_simulations/othello_sim_boards_100000_6x6'
games = read_files(path)

# %%
state_return_tuples = unpack_positions_returns(games)
state_return_tuples = unique_board_positions(state_return_tuples)

X, y = value_predictors_targets(state_return_tuples)

# splitting data into training and validation
X_train_val, X_test_val, y_train_val, y_test_val = train_test_split(X, y, test_size=0.2,
```

```
                                                                    random_state
    =80085)

train_dataset_value_model = create_dataset(X_train_val, y_train_val, batch_size=256,
                                           shuffle_buffer_size=10000)
val_dataset_value_model = create_dataset(X_test_val, y_test_val, batch_size=256,
                                         shuffle_buffer_size=10000)

# Training the value model
history_value = value_model.fit(
    train_dataset_value_model,
    epochs = 50,
    validation_data=val_dataset_value_model
    # callbacks=[keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)]
)

# with open('initial_value_training_history', 'wb') as handle:
#     pickle.dump(history_value, handle, protocol=pickle.HIGHEST_PROTOCOL)

# # Saving the weights from the first good network
# value_model.save_weights('zeroth_value_nn.weights.h5')

# %%
##########################################################################
#  ------------------- Initial training of Policy NN  ---------------------
##########################################################################

sav = read_files("./othello_random_simulations/othello_sim_sa_visits")
X, y = policy_predictors_targets(sav, 6)

X_train_pol, X_test_pol, y_train_pol, y_test_pol = train_test_split(X, y, test_size
    =0.2,
                                                     random_state=80085)

train_dataset_policy_model = create_dataset(X_train_pol, y_train_pol, batch_size=128)
val_dataset_policy_model = create_dataset(X_test_pol, y_test_pol, batch_size=128)

history_policy = policy_model.fit(train_dataset_policy_model,
                        validation_data=val_dataset_policy_model,
                        epochs=500)

# with open('initial_policy_training_history_2', 'wb') as handle:
#     pickle.dump(history_policy, handle, protocol=pickle.HIGHEST_PROTOCOL)

# saving weights from first good network
# policy_model.save_weights('zeroth_policy_nn_2.weights.h5')


##########################################################################
#  -------------- Concatenating history from different runs -------------
##########################################################################

# concatenating and plotting history from first and second iteration of VALUE network
initial_value_training_history_1 = read_files('initial_value_training_history')
initial_value_training_history_2 = read_files('initial_value_training_history_2')

complete_value_history_train_loss = initial_value_training_history_1.history['loss'] +
     initial_value_training_history_2.history['loss']
complete_value_history_val_loss = initial_value_training_history_1.history['val_loss']
     + initial_value_training_history_2.history['val_loss']

plt.plot(complete_value_history_train_loss, label='Training loss')
plt.plot(complete_value_history_val_loss, label = 'Validation loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
```

```
plt.title('value network')
plt.legend(loc='upper right')
plt.show()

# %%
# concatenating and plotting history from first and second iteration of POLICY network
initial_policy_training_history_1 = read_files('initial_policy_training_history')
initial_policy_training_history_2 = read_files('initial_policy_training_history_2')

complete_policy_history_train_loss = initial_policy_training_history_1.history['loss']
     + initial_policy_training_history_2.history['loss']
complete_policy_history_val_loss = initial_policy_training_history_1.history['val_loss
    '] + initial_policy_training_history_2.history['val_loss']

plt.plot(complete_policy_history_train_loss, label='Training loss')
plt.plot(complete_policy_history_val_loss, label = 'Validation loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('policy network')
plt.legend(loc='upper right')
plt.show()
```