Agent Learning

Alessandro Pio n. 294417

November 19, 2023

Introduction

This report details the implementation of a chess game agent and the creation of a dataset for training machine learning models. The main goal is to maximize the decision-making performance of the agent by employing advanced algorithms and sophisticated strategies.
So, for this assignment, the adaptive min-max was implemented to play the chess game. The following are shown and explained the heuristics implemented, the min-max algorithm with alpha-beta pruning, the Multi-Layer Perceptron repressor with its analysis and finally a final analysis of the games played.

Phases of Alpha-Beta Minimax Algorithm Improvement

Phase 1: Optimization through H0 Static Evaluation
The initial phase focuses on optimizing the Alpha-Beta Minimax search algorithm by introducing an H0 static evaluation tailored for chess. The goal is to improve algorithm efficiency by identifying and selecting the most promising states during the game's decision tree exploration.

Phase 2: Generalization with HL Evaluation
The second phase extends the concept of static evaluation by introducing an HL evaluation specifically designed for chess. The parameter l ranges from 0 to L, where L represents the maximum level of generalization. The trade-off between precision and computational speed is explored through experiments with different l values.

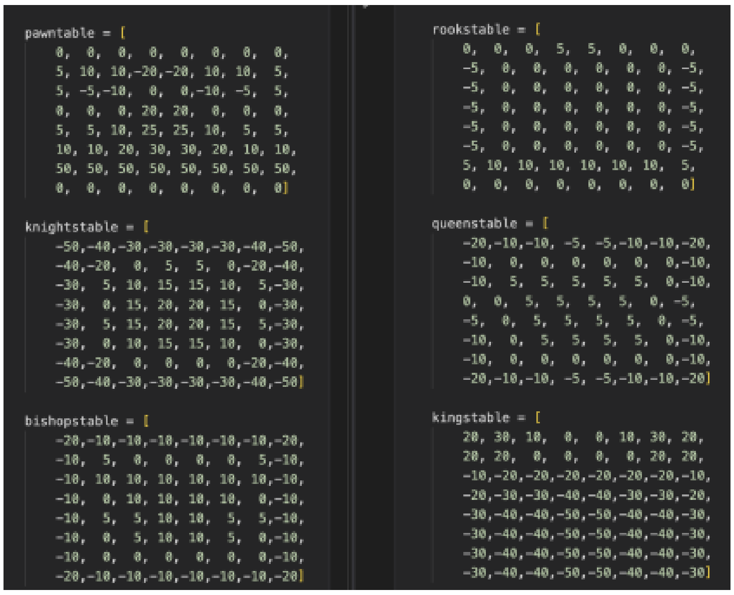
Phase 3: Introduction of H0 based on Regressor R
In the third phase, machine learning is integrated into state evaluations. An H0 function based on specific chess observations is defined, and a regressor (R) is introduced to predict HL values. This regressor replaces static evaluations in the Alpha-Beta Minimax algorithm, enabling more accurate predictions.

Implementation Overview

Heuristics

The implementation introduces a heuristic evaluation strategy through the `HeuristicChess` class. Various static methods compute heuristic values based on different criteria, such as overall board status, piece count, and piece positioning. In particular we have:

- (`Position`) A score is assigned to all the pieces of the chessboard on the basis of the matrices in which the values of the various elements on the chessboard are reported, obviously, this value is calculated by making the value of the player who has the turn minus the value of the opponent.
- (`Pieces Number`) This heuristic is based on counting the pieces of each type for both players on the chessboard. Each type of piece (pawn, knight, bishop, rook, queen) has an associated weight, and the overall evaluation is given by the weighted sum of the differences in piece counts between the players.
  This score takes into account the difference in material strength between the two players on the chessboard. Higher values indicate a favorable position for white, while lower values indicate a favorable position for black.
  As we can se below:



```
pawntable = [
    0,  0,  0,  0,  0,  0,  0,  0,
    5, 10, 10,-20,-20, 10, 10,  5,
    5, -5,-10,  0,  0,-10, -5,  5,
    0,  0,  0, 20, 20,  0,  0,  0,
    5,  5, 10, 25, 25, 10,  5,  5,
    10, 10, 20, 30, 30, 20, 10, 10,
    50, 50, 50, 50, 50, 50, 50, 50,
    0,  0,  0,  0,  0,  0,  0,  0]

knightstable = [
    -50,-40,-30,-30,-30,-30,-40,-50,
    -40,-20,  0,  5,  5,  0,-20,-40,
    -30,  5, 10, 15, 15, 10,  5,-30,
    -30,  0, 15, 20, 20, 15,  0,-30,
    -30,  5, 15, 20, 20, 15,  5,-30,
    -30,  0, 10, 15, 15, 10,  0,-30,
    -40,-20,  0,  0,  0,  0,-20,-40,
    -50,-40,-30,-30,-30,-30,-40,-50]

bishopstable = [
    -20,-10,-10,-10,-10,-10,-10,-20,
    -10,  5,  0,  0,  0,  0,  5,-10,
    -10, 10, 10, 10, 10, 10, 10,-10,
    -10,  0, 10, 10, 10, 10,  0,-10,
    -10,  5,  5, 10, 10,  5,  5,-10,
    -10,  0,  5, 10, 10,  5,  0,-10,
    -10,  0,  0,  0,  0,  0,  0,-10,
    -20,-10,-10,-10,-10,-10,-10,-20]
```

```
rookstable = [
    0,  0,  0,  5,  5,  0,  0,  0,
    -5,  0,  0,  0,  0,  0,  0, -5,
    -5,  0,  0,  0,  0,  0,  0, -5,
    -5,  0,  0,  0,  0,  0,  0, -5,
    -5,  0,  0,  0,  0,  0,  0, -5,
    -5,  0,  0,  0,  0,  0,  0, -5,
    5, 10, 10, 10, 10, 10, 10,  5,
    0,  0,  0,  0,  0,  0,  0,  0]

queenstable = [
    -20,-10,-10, -5, -5,-10,-10,-20,
    -10,  0,  0,  0,  0,  0,  0,-10,
    -10,  5,  5,  5,  5,  5,  0,-10,
    0,  0,  5,  5,  5,  5,  0, -5,
    -5,  0,  5,  5,  5,  5,  0, -5,
    -10,  0,  5,  5,  5,  5,  0,-10,
    -10,  0,  0,  0,  0,  0,  0,-10,
    -20,-10,-10, -5, -5,-10,-10,-20]

kingstable = [
    20, 30, 10,  0,  0, 10, 30, 20,
    20, 20,  0,  0,  0,  0, 20, 20,
    -10,-20,-20,-20,-20,-20,-20,-10,
    -20,-30,-30,-40,-40,-30,-30,-20,
    -30,-40,-40,-50,-50,-40,-40,-30,
    -30,-40,-40,-50,-50,-40,-40,-30,
    -30,-40,-40,-50,-50,-40,-40,-30,
    -30,-40,-40,-50,-50,-40,-40,-30]
```

Min-Max with Alpha-Beta Pruning

This algorithm creates a tree in a dfs fashion, where each node of the tree represents the board in one state. Once the leaves have been reached, the evaluation of the heuristics is carried out and, depending on whose turn it is, the minimum or maximum value of the various leaves is taken into consideration. In fact, in the algorithm, there is a maximizing and a minimizing player, in the specific case of the root the maximum value of the next level is associated.
The maximizing players and the minimizing alternate in minimum and maximum until the leaves.
The alpha-beta pruning algorithm has also been implemented in the min-max algorithm. This algorithm tries to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree and thus saves CPU and memory time.
The min-max algorithm was run several times with various heuristics and with depths from 1 to 3.
Within the implementation we can find the instruction random.shuffle(moves), this instruction shuffles the list of possible moves that can be made every time. This instruction is used to not receive the same move every time the heuristic returns equal values in the roots. In fact, the heuristic returns equal values especially at the beginning when white has to make the first move. This way you will have a different move every time.

Agent and Player Implementation

The code defines generic agents through the `Agent` class, with specific player identifiers ('WHITE' or 'BLACK') and solvers. The `GreedyPlayer` class makes immediate "greedy" moves based on heuristic evaluations, while the `MinMaxPlayer` class employs the Alpha-Beta Minimax algorithm. The `MinMaxPlayerSpeedUp` class optimizes the search process by exploring only promising subtrees.

PredictivePlayer and Variants

The `PredicativePlayer` class introduces predictive modeling for move selection, using regression models. Variants include `PredictivePlayerGreedy` and `PredictivePlayerMinMax`, combining predictive capabilities with the MinMax algorithm.

Regressor

To create an Adaptive Min-Max, a Multi-Layer Perceptron regressor (MLPRegressor henceforth) has been implemented, capable of predicting the Min-Max value with a depth of 3 and corresponding heuristics.
For this purpose, board configurations and Min-Max evaluations were collected to create the dataset and train the MLPRegressor. Specifically, 1000 games were played, with white implementing alpha-beta pruning Min-Max and black being controlled by an agent using the greedy algorithm.
Through the 1000 played games, a dataset of 1620 rows was created. Within the dataset, the first 64 columns represent the chessboard squares with pieces (returned by Min-Max), and the subsequent 4

columns pertain to observations such as the Best move, Piece count, Threat count, and Pawn Structure. The last column, in this case, ”score,” contains the heuristic evaluation (also returned by Min-Max) for that specific state.
Before proceeding with the neural network training, a data conversion was performed to be passed to the regressor (as observed in the code, an example being the numeric mapping of the best move). Subsequently, the MLPRegressor was trained with two hidden layers of 500 neurons each. This configuration achieved a test set score of approximately 0.86, suggesting good training, but a 0.4 in tests indicates a significant discrepancy.
In particular, the MLPRegressor was saved using the Python library ”joblib,” allowing the saving of neural network configurations for import into other notebooks/applications at any time.

Main Class

The `Main` class orchestrates the execution of multiple chess matches between players, providing an overview of the results. It tracks wins for white and black players, draws, total moves, and average moves per game. In each case sets up a simulation with 100 matches (rep) between two players (player1 and player2) using a specific chess game implementation (game). The play method is then called to execute the simulation and print the results of total games.

Statistics and Results

Experiment 1: PredictiveMinMax vs. GreedyPlayer

- Wins: WHITE (3), BLACK (0), Draws (97)
- Total Moves: 4099, Average Moves per Game: 40.99
- Win Percentages: WHITE (3.0%), BLACK (0.0%), Draw Percentage (97.0%)
- Execution Time: Total (4496.16 seconds), Average per Match (44.96 seconds)

As you can see, the Min Max algorithm is definitely better than greedy as the latter never wins, however the predictor’s limits are evident, given that the win ratio is very low, in fact almost all are draws.

Experiment 2: PredictiveMinMax vs. MinMaxPlayer

- Wins: WHITE (21), BLACK (28), Draws (51)
- Total Moves: 4132, Average Moves per Game: 41.32
- Win Percentages: WHITE (21.0%), BLACK (28.0%), Draw Percentage (51.0%)
- Execution Time: Total (4561.65 seconds), Average per Match (45.62 seconds)

The MinMax agent achieved a slightly higher number of victories compared to the predictive agent, which may indicate that, in specific configurations or considered game dynamics, the black agent has a strategic advantage or is better able to capitalize on opportunities,

Experiment 3: PredictiveMinMax vs. PredictiveGreedy

- Wins: WHITE (13), BLACK (0), Draws (87)
- Total Moves: 4087, Average Moves per Game: 40.87
- Win Percentages: WHITE (13.0%), BLACK (0.0%), Draw Percentage (87.0%)
- Execution Time: Total (4490.8 seconds), Average per Match (44.91 seconds)

These results suggest that, in the specific configurations or game dynamics considered, the MinMax approach with pruning adopted by the white player may be more effective in making winning decisions compared to the Greedy method used by the black player. The lack of victories for the black player could indicate a limitation in the effectiveness of the Greedy strategy in this specific context.

Experiment 4: PredictiveGreedy vs PredictiveMinMax

- Wins: WHITE (0), BLACK (57), Draws (46)
- Total Moves: 2855, Average Moves per Game: 28.55
- Win Percentages: WHITE (0.0%), BLACK (54.0%), Draw Percentage (46.0%)
- Execution Time: Total (917.94 seconds), Average per Match (9.18 seconds)

These results suggest that, in the specific configurations or game dynamics considered, the MinMax approach adopted by the black player (MinMax) has been consistently more successful, securing a significant number of victories compared to the Greedy approach utilized by the white player (PredictiveGreedy). The draws indicate a balance in some game scenarios, but the dominance of MinMax in wins implies its strategic advantage in this context.

Conclusions

The results obtained suggest that, in the specific configurations or game dynamics considered, the MinMax approach with pruning adopted by the black player (MinMax) has consistently been more effective, securing a significant number of victories compared to the Greedy approach utilized by the white player (PredictiveGreedy). Draws indicate a balance in some game scenarios, but the dominance of MinMax in wins implies its strategic advantage in this context.
The superior performance of MinMax can be attributed to its ability to conduct a more in-depth exploration of the decision tree, considering a broader range of possible moves and responses. Unlike the immediate and local decision-making of the Greedy approach, MinMax's systematic exploration allows it to anticipate and capitalize on strategic opportunities, especially in complex game scenarios.
Furthermore, the alpha-beta pruning integrated into the MinMax algorithm plays a crucial role in reducing computational complexity. By strategically avoiding the exploration of redundant branches in the decision tree, MinMax optimizes its decision-making process. This becomes particularly advantageous in chess, where the decision tree can become vast, and efficient exploration is paramount.
The incorporation of a Multi-Layer Perceptron (MLP) regressor further enhances MinMax's predictive capabilities. The MLP regressor, trained on a dataset of chessboard configurations and MinMax evaluations, captures intricate patterns and non-linear relationships, enabling more accurate predictions for diverse game situations.