Homework 3 Adaptive Predictive MinMax

Alessandro Pio 294417
Lorenzo Tucceri Cimini 294425

January 26, 2025

Contents

1 Introduction

Chess has long been an ideal testbed for the development of artificial intelligence algorithms due to its strategic complexity and the vast search space generated by each game position. One of the main challenges for chess engines lies in accurately evaluating positions, identifying the most promising ones for the players. Traditionally, this task is addressed using search algorithms such as *MinMax* with Alpha-Beta pruning.

1.1 MinMax and Alpha-Beta

The *MinMax* algorithm is a recursive technique used to make optimal decisions in zero-sum games, such as chess. Its core principle is to maximize the score of the player in turn (*Max*) while minimizing the opponent's score (*Min*). To improve efficiency, Alpha-Beta pruning eliminates branches from exploration that cannot influence the final outcome, significantly reducing the number of nodes analyzed. Despite this optimization, the computational complexity remains exponential with respect to the search depth, making it impractical to analyze positions at higher levels on limited hardware.

1.2 Limitations of MinMax and Deep Learning

While MinMax with Alpha-Beta pruning is a well-established approach, its computational cost grows rapidly with increasing depth. This necessitates exploring alternative methods that can maintain high-quality evaluations without sacrificing efficiency. In this context, deep learning offers a promising solution: neural network models can be trained to approximate evaluations computed by MinMax, providing accurate results in real time.

1.3 Objectives of the Work

This work proposes a hybrid approach that combines MinMax's exploration capabilities with the speed and flexibility of neural networks. The key idea is to use a deep learning-based regressor, referred to as **RL Regressor**, designed to learn to estimate chess position values. This model takes as input a set of features extracted from the chessboard and outputs a numerical estimate of the position's value.

To progressively enhance the predictive capabilities of the regressor, an iterative bootstrap approach was adopted. In the initial phase, the model is trained on a dataset generated by MinMax. Subsequently, the model's estimates are used to generate new datasets, creating a continuous refinement loop. This process effectively simulates MinMax analysis at different depths while reducing computational costs.

1.4 Main Contributions

The main contributions of this work include:

1. Defining a neural network architecture designed for chess position evaluation.

2. Implementing an iterative bootstrap pipeline for continuous model improvement.

3. Conducting a detailed analysis of the model's performance in terms of accuracy and speed compared to the traditional MinMax algorithm.

The results demonstrate that the proposed approach significantly reduces evaluation times. The theoretical and implementation aspects of the method, along with an empirical analysis of the results, will be discussed in detail in the following sections.

2 Dataset Analysis

This section analyzes the dataset used in the experiments to better understand the distribution of features and labels, as well as to evaluate correlations between variables. The analyses are accompanied by meaningful visualizations.

2.1 Feature Correlations

Figure 1 shows the correlation matrix between the dataset's features. The correlation values are represented using a heatmap, where shades of red indicate stronger positive correlations, while shades of blue indicate stronger negative correlations.

- Most features exhibit weak correlations, with values close to 0, indicating low linear dependence.
- Notable correlations are observed between feature 1 and feature 2 ($r = 0.49$) and between feature 9 and feature 10 ($r = 0.49$).
- The `label` column shows relatively low correlations with all features, with the highest value being 0.39 for feature 0.

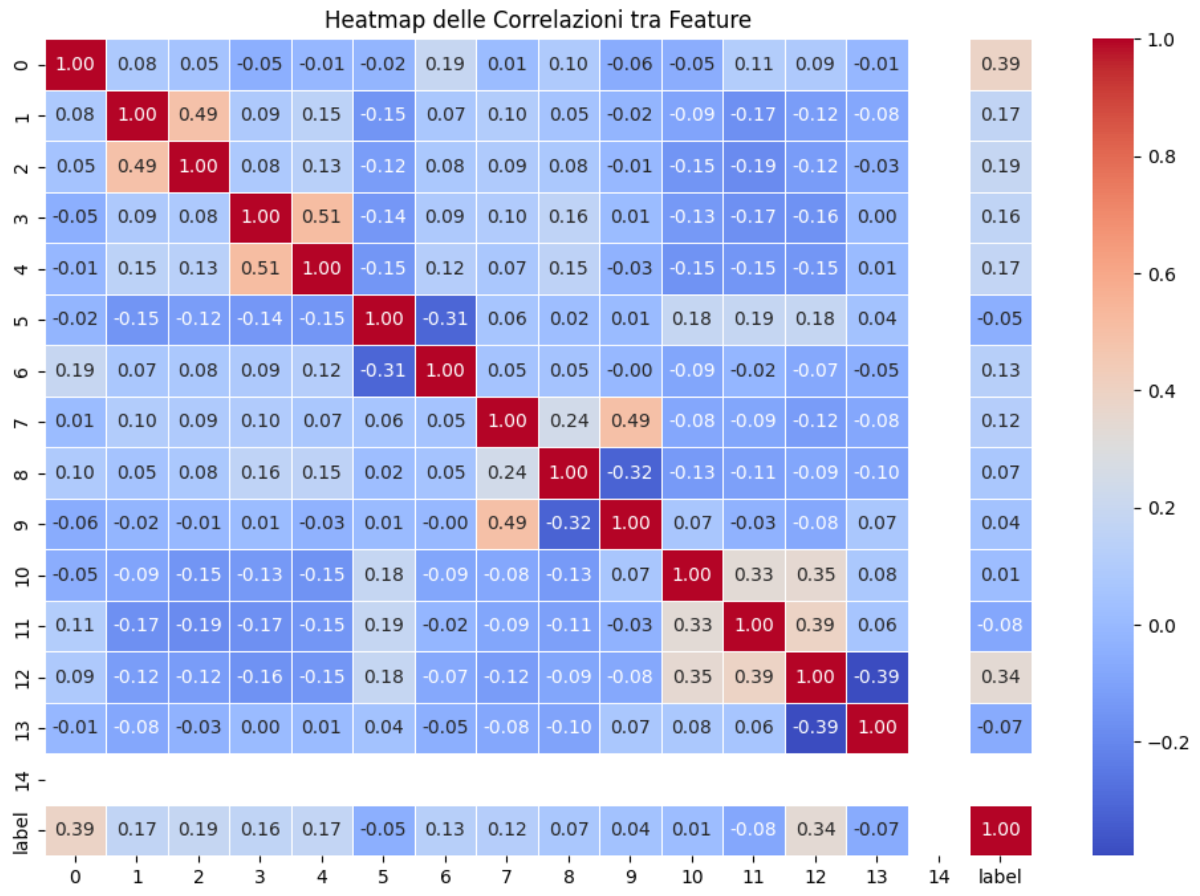

Figure 1: Heatmap of feature correlations in the dataset.

2.2 Label Distribution

Figure 4 shows the histogram of label (y) distribution, representing the evaluation value of the board. The distribution highlights:

- A central peak around the value 8, indicating that most data is concentrated in this range.
- A slight left skew, with a longer tail for values below 8.
- A limited number of extreme values, both lower and higher, which may impact model training.

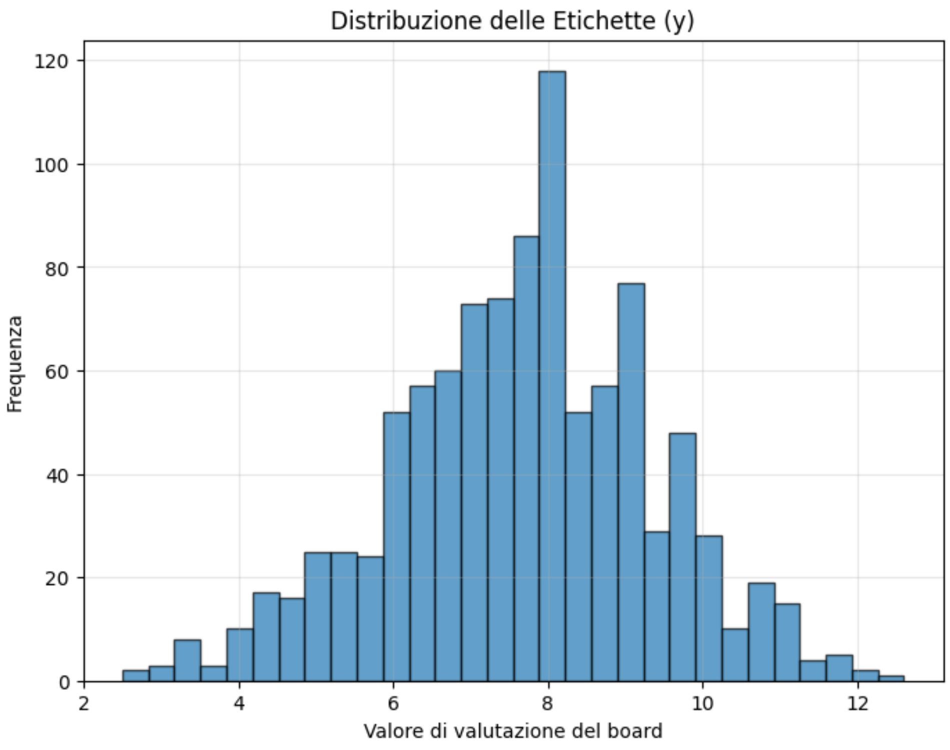

Figure 2: Label (y) distribution in the dataset.

2.3 Turn Distribution

Since the MinMax algorithm alternates players, the dataset's turn distribution is balanced. This means that the analyzed positions include an almost equal number of turns for each player. This balance ensures a fair representation of decisions for both sides during model training.

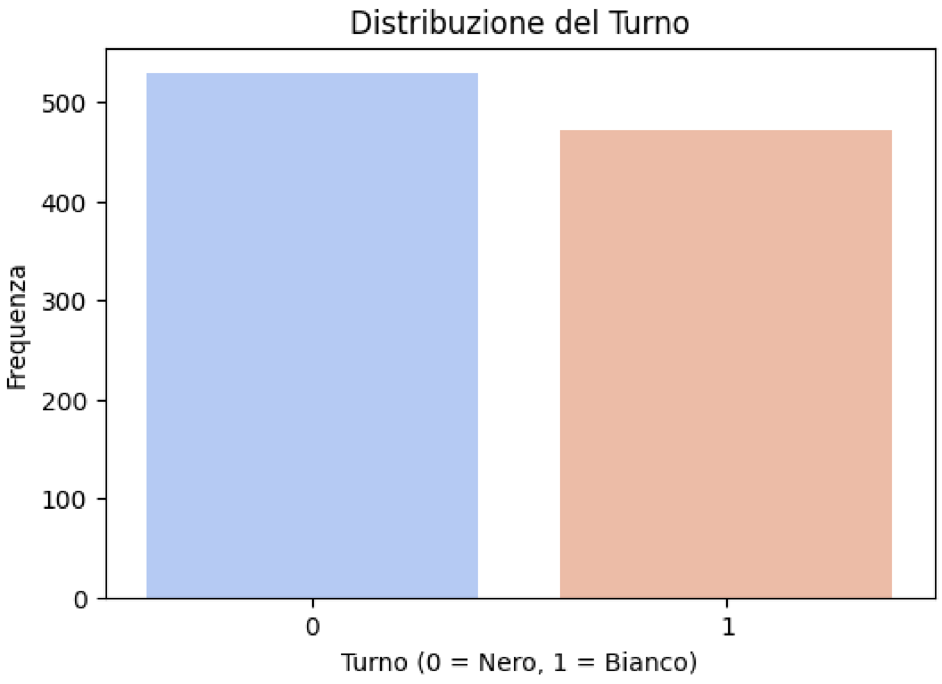

Figure 3: Turn distribution.

2.4 Check Distribution

The distribution of positions where a check occurs is significantly lower than the total analyzed positions. A check represents a less frequent event, as it arises only in specific game situations. This underscores the need to treat such positions carefully, given their potential impact on the overall board evaluation.

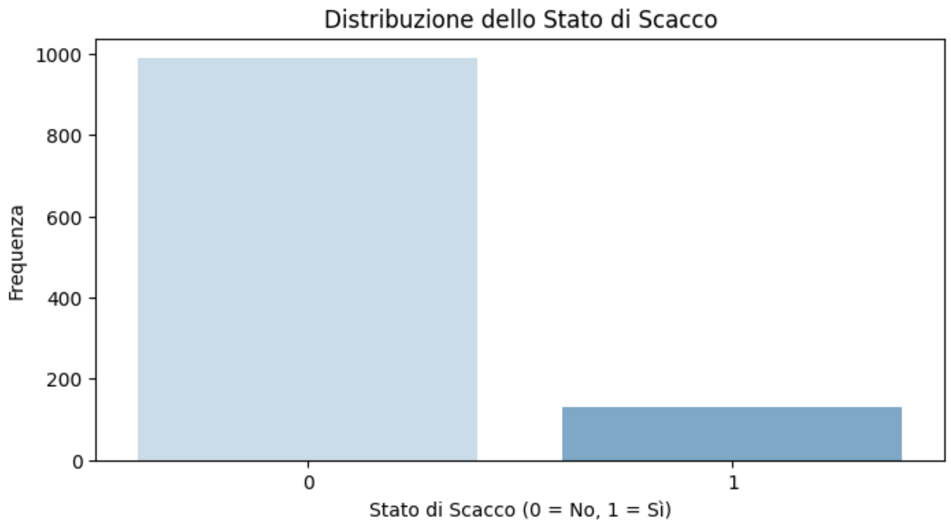

Figure 4: Check distribution.

Similarly, the distribution of positions where a checkmate occurs is even more limited. As shown in the figure below, checkmate (represented with a value of “1”) is an extremely rare event in the dataset. This reflects its exceptional nature and the specific conditions required to achieve it. Given its rarity, it is essential to analyze checkmate positions carefully, as they represent the definitive conclusion of a game and have a significant impact on the evaluation of overall game strategy.

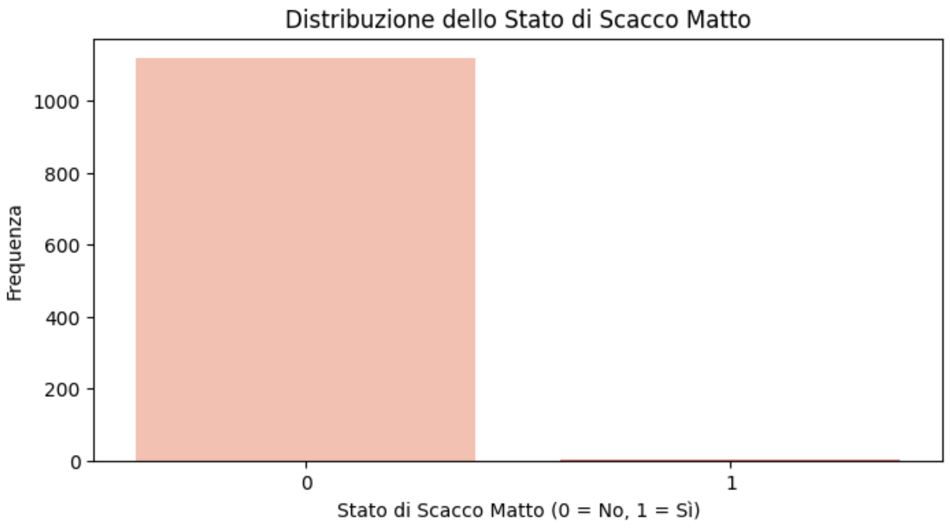

Figure 5: Checkmate State Distribution (0 = No, 1 = Yes).

In summary, the dataset shows limited correlations between features and a well-defined label distribution, with a clear concentration around central values. These aspects may influence the model's performance, particularly in its ability to generalize to extreme values.

3 Comparison with an Evolutionary Approach

3.1 Introduction

In the context of artificial intelligence development for chess, improving the *Alpha-Beta Minimax* algorithm has been central to creating intelligent agents capable of making optimal decisions. An evolutionary approach can be divided into three main phases, including static optimization with H0, generalization with HL, and the integration of *machine learning* through a regressor R. These phases reflect a progressive evolution in evaluation capacity and computational efficiency of the algorithms.

3.2 Phase 1: Optimization with H0 (Static Evaluation)

The first phase involves using a static evaluation, H0, designed to identify the most promising states during the exploration of the move tree. This approach improves efficiency by avoiding the analysis of all possible moves. This strategy relies on the use of static features such as control of the center of the board, material value, and other well-defined metrics. In our code, this phase is comparable to extracting key features from the board, which serve as input to the evaluation model.

3.3 Phase 2: Generalization with HL (Advanced Evaluation)

In the second phase, the evolutionary approach introduces an advanced evaluation, HL, where the parameter l varies from 1 to L, representing the maximum level of generalization. This system allows a trade-off between accuracy and computational speed: increasing l results in more accurate evaluations, but at a greater computational cost. In our method, this concept is represented by using increasingly advanced models to predict more accurate evaluations, such as levels HL1 and beyond, which incorporate growing complexity in the analysis.

3.4 Phase 3: Integration of H0 with Regressor R

The third phase involves the integration of *machine learning* to further improve evaluations. This approach uses a regressor R trained to predict HL values based on features extracted from H0. This represents a step toward greater automation and reduced computational complexity, as the predictive model can simulate higher-level evaluations without direct calculations. In our method, this phase corresponds to training the RL1 regressor using datasets generated from Alpha-Beta Minimax evaluations at increasing levels.

3.5 Differences Between Approaches

The main differences between a traditional evolutionary approach and our regressor-based method can be summarized as follows:

- **Computational Efficiency**: While the evolutionary approach relies on progressive refinement of evaluations with increasingly complex calculations, our method significantly reduces computational costs by using a regressor that effectively simulates HL levels.
- **Automation**: The integration of the regressor in our method allows for automating much of the evaluation process, reducing dependence on static parameters and predefined models.
- **Adaptability**: The regressor-based model can be trained on different datasets and optimized for specific scenarios, while the evolutionary approach often requires manual readjustment for different scenarios.
- **Scalability**: The *machine learning*-based approach is more scalable at higher exploration levels, as the regressor maintains a constant speed once trained, unlike the traditional approach, which exponentially increases costs.

In conclusion, our method represents a significant evolution compared to traditional approaches, combining the advantages of evolutionary techniques with the power of machine learning to provide an efficient and precise solution for evaluating chess positions.

4 Implementation

The code explicitly utilizes the GPU to optimize performance during the training and prediction of neural regressors. This is achieved by specifying `tf.device('/GPU:0')` in the training and inference functions of the model.

The use of the GPU proves particularly advantageous in the following scenarios:

- **Accelerating the training of the RL1 regressor**: Training the RL1 model can require numerous iterations to converge, and the use of the GPU significantly reduces the required time.

- **Speeding up predictions during simulation**: During the dataset simulation process, the GPU reduces the time required for data generation, especially in the later stages of iterative bootstrapping.

If a GPU is not available, TensorFlow automatically uses the CPU, resulting in slower execution times. This trade-off can impact overall performance, particularly for large models or extensive simulations.

The implementation of the evaluation system based on the RL regressor is divided into several key phases. Each phase plays a crucial role in ensuring effective learning and accurate evaluation of chess positions.

4.1 Feature Extraction

The representation of chess positions is critical to the model's effectiveness. The board is encoded into a numerical vector that includes a combination of static and dynamic features. Specifically:

- **Player turn**: A binary value indicating whether it is White's or Black's turn.
- **Castling rights**: Four binary values (two for each side, king-side and queen-side) indicating castling possibilities for White and Black.
- **Fifty-move rule**: A numerical value representing the counter for the fifty-move rule.
- **En passant status**: Specifies whether an en passant capture is possible and, if so, on which column.
- **Total number of White pieces**: A numerical value representing the total count of White pieces on the board.
- **Total number of Black pieces**: A numerical value representing the total count of Black pieces on the board.
- **Material evaluation**: A numerical score representing the overall material value for both players.
- **White's center control**: A metric evaluating how many central squares (*d4*, *d5*, *e4*, *e5*) are under White's control.
- **Black's center control**: A metric evaluating how many central squares (*d4*, *d5*, *e4*, *e5*) are under Black's control.
- **Number of legal moves**: A numerical value representing the total number of legal moves available for the current player.
- **Check status**: A binary indicator representing whether a player's king is in check.
- **Checkmate status**: A binary indicator representing whether a player's king is in checkmate.

This representation captures key aspects of the position without including the entire board configuration, reducing the dimensionality of the problem.

4.2 RL Regressor

The RL model is implemented using a feed-forward neural network. The network structure consists of:

- **Input**: The feature vector extracted from the chess position.
- **Hidden layers**: Two dense layers, each with 128 neurons, activated by the *ReLU* (*Rectified Linear Unit*) function.
- **Output**: A single neuron returning a scalar value representing the position evaluation.

The model is trained using:

- **Optimizer**: Adam (*Adaptive Moment Estimation*) with a learning rate of 0.001.
- **Loss function**: MAE (*Mean Absolute Error*), which measures the difference between the model's predictions and the evaluations provided by the Minimax algorithm.
- **Batch size**: 32 samples per weight update.
- **Number of epochs**: Fixed at 150 based on experimental results.

4.3 Iterative Bootstrapping

The iterative bootstrapping process progressively improves the predictive ability of the RL model:

- In the **first iteration**, the dataset is generated using the Minimax algorithm with a depth limited to 4.
- In each subsequent iteration:
    1. The trained RL regressor is used to simulate new evaluations for unseen positions.
    2. The dataset is enriched with these new evaluations.
    3. A new RL model is trained using the evaluations from the previous regressor.

Regarding the dataset, two experiments were conducted on its size:

- **Constant dataset size**: The dataset size remains fixed across iterations, serving as the baseline approach for this study.
- **Growing dataset size**: The dataset size increases by 25% in each iteration, including more complex positions and higher levels. This approach was considered due to the short generation times with the regressor, providing more data for subsequent iterations.

4.4 Dataset Generation

The creation of a representative dataset is crucial for model effectiveness:

- **Initial generation**: A set of random legal chess positions is generated, ensuring diversity in the configurations.
- Each position is represented by:
  - The feature vector extracted from the board.
  - The evaluation value provided by the Minimax algorithm or the RL regressor, depending on the bootstrap iteration.
- Datasets are split into training (80%) and test (20%) sets.

4.5 Result Visualization

To analyze model performance, several metrics are measured:

- **Dataset size**: The total number of positions included in each iteration.
- **Dataset generation time**: The time required to create the dataset using Minimax or RL simulations.
- **Training time**: The time needed to complete an iteration of RL model training.
- **Mean Absolute Error (MAE)**: The average absolute difference between the regressor's predictions and the evaluations computed by the Minimax algorithm.
- **Convergence graphs**: The loss trends during training, illustrating model stability and effectiveness.

The results are visualized through graphs and tables to identify areas for improvement and evaluate the overall effectiveness of the approach.

5 Comparison between MinMax and RL Regressor

In this section, we compare the traditional MinMax approach with the proposed RL regressor method for evaluating chess positions. The aim is to highlight differences in computational efficiency, accuracy, and scalability.

5.1 Computational Efficiency

The MinMax algorithm with Alpha-Beta pruning requires a recursive analysis of moves and counter-moves up to a defined depth, leading to an exponential computational complexity ($O(b^d)$), where b is the branching factor and d is the depth. In contrast, the RL regressor, once trained, provides constant-time evaluations ($O(1)$) regardless of the position's complexity.

- **Evaluation Time:**
  - MinMax: Time increases drastically with the search depth.

  - RL Regressor: Evaluation is instantaneous, enabling rapid analysis even for a high number of positions.

- **Scalability:**
  - MinMax: Poor scalability for higher depths or limited hardware.
  - RL Regressor: Scalable to larger and more complex datasets.

5.2 Evaluation Accuracy

MinMax is traditionally used for position evaluation, producing highly accurate results for a defined depth. However, the RL regressor is designed to approximate these evaluations, balancing accuracy and efficiency.

- **Quantitative Comparison:**
  - The RL regressor's evaluations compared to MinMax at different depths.
  - The RL regressor's MAE tends to decrease with more bootstrap iterations, approaching MinMax's values.

- **Extreme Value Errors:**
  - The RL regressor struggles to capture extreme values (e.g., imminent checkmate positions), which remain challenging for approximation.

5.3 Robustness and Adaptability

A key advantage of the RL regressor is its adaptability to different scenarios through training. This makes it ideal for dynamic or evolving environments.

- **Adaptability:**
  - MinMax: Not adaptable; evaluations rigidly depend on the evaluation function and depth.
  - RL Regressor: Can be retrained to improve evaluations based on new datasets.

- **Error Robustness:**
  - MinMax: Reliable but vulnerable to the quality of the static evaluation function.
  - RL Regressor: Quality depends on the size and quality of the dataset.

5.4 Performance Comparison

Table 1 summarizes the comparison between the two approaches.

Feature	**MinMax**	**RL Regressor**
Computational Efficiency	Low	High
Accuracy	High	Training-dependent
Scalability	Limited	High
Evaluation Time	Increases with depth	Constant
Adaptability	Limited	High
Robustness	High	Dataset-dependent

Table 1: Comparison between MinMax and RL Regressor.

6 Analysis and Results

This section analyzes experimental results, evaluating the RL model's performance in terms of Mean Absolute Error (MAE), training and generation times, and overall computational efficiency. Each experiment utilized different configurations for the number of chess positions, maximum depth, and depth-increment strategies.

6.1 Experiment 1: Constant Dataset Size ($L = 16$)

Configuration: In this experiment, the model was trained with a constant dataset size throughout all iterations, up to a maximum depth of $L = 16$.

Main Results:

- **Maximum Depth:** 16.
- **Final MAE:** 0.0766.
- **Final Training Time:** 8.25 seconds.

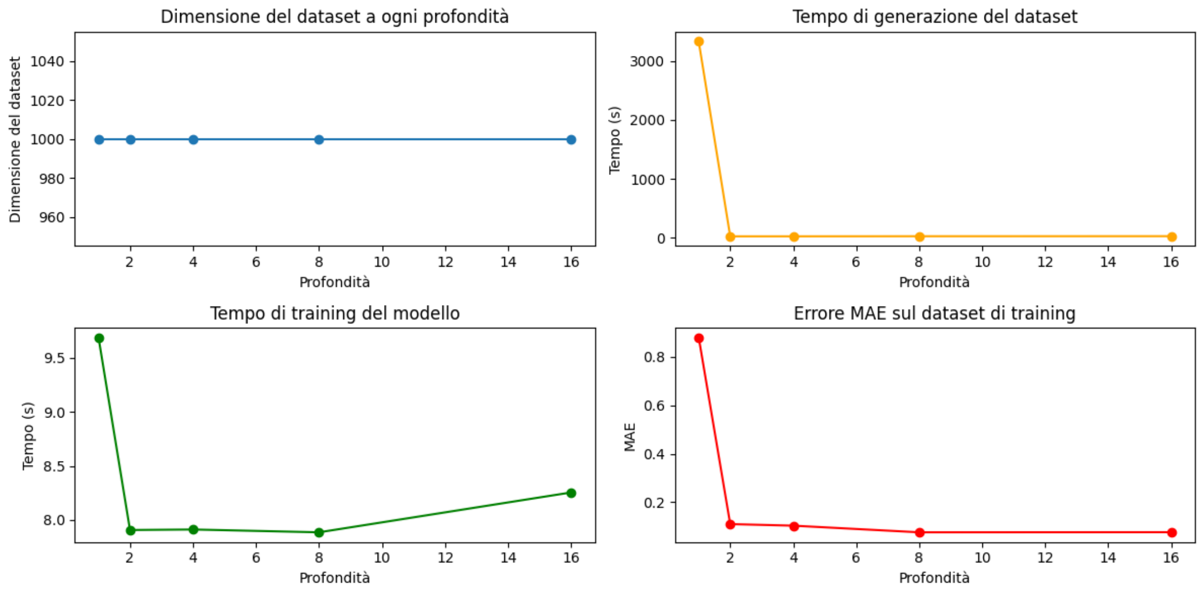

Figure 6: Statistics for the experiment with constant dataset size and maximum depth $L = 16$.

6.2 Experiment 2: Incremental Dataset Size ($L = 32$)

Configuration: In this experiment, the dataset size grew by 25% at each step, starting from an initial depth of 4 and reaching a maximum depth of $L = 32$.

Main Results:

- **Maximum Depth:** 32.
- **Final MAE:** 0.0608.
- **Final Training Time:** 20.26 seconds.

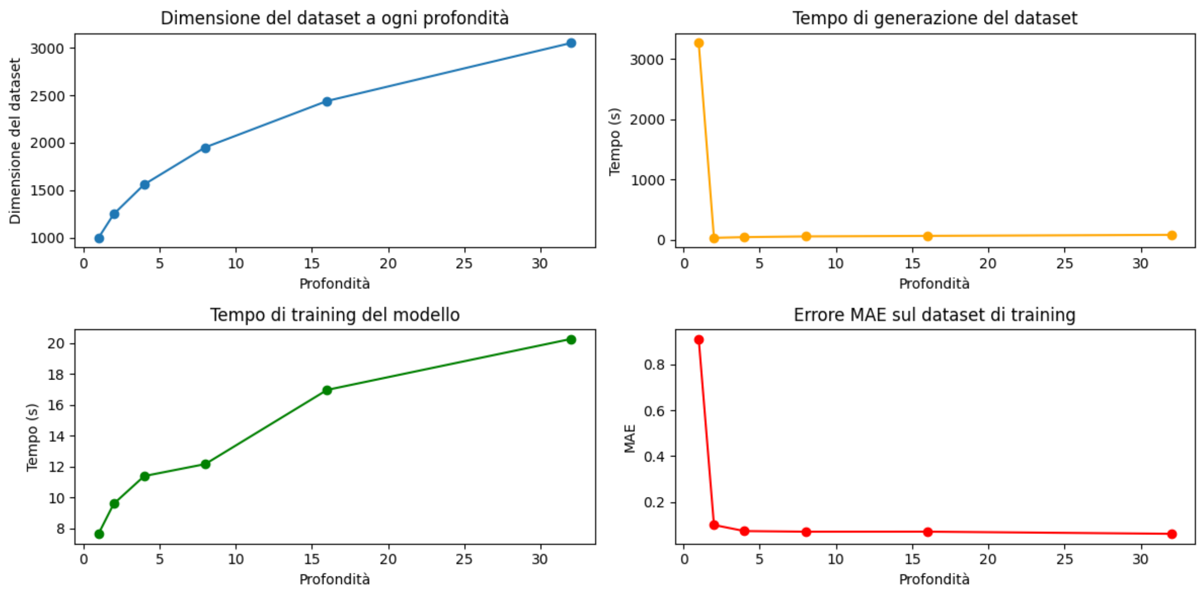

Figure 7: Statistics for the experiment with incremental dataset size and maximum depth $L = 32$.

6.3 Experiment 3: MinMax vs. Regressor

In this experiment, we tested an RL1 model based on a regressor against an opponent controlled by the MinMax algorithm. The goal was to analyze RL1's performance in strategic decision-making using MinMax with a depth of 4. The game was simulated over 100 matches, with a maximum of 100 moves per match.

Figure 8 illustrates match results, divided into three categories:

- RL1 Wins (playing as White).
- MinMax Wins (playing as Black).
- Draws.

From the data, MinMax showed clear superiority, winning approximately 70% of the matches. The RL1 regressor failed to secure significant victories and never directly outperformed MinMax. The remaining matches ended in draws, accounting for about 30

These results highlight some limitations of the RL1 regressor approach in competitive scenarios against a structured opponent like MinMax. Despite leveraging a trained

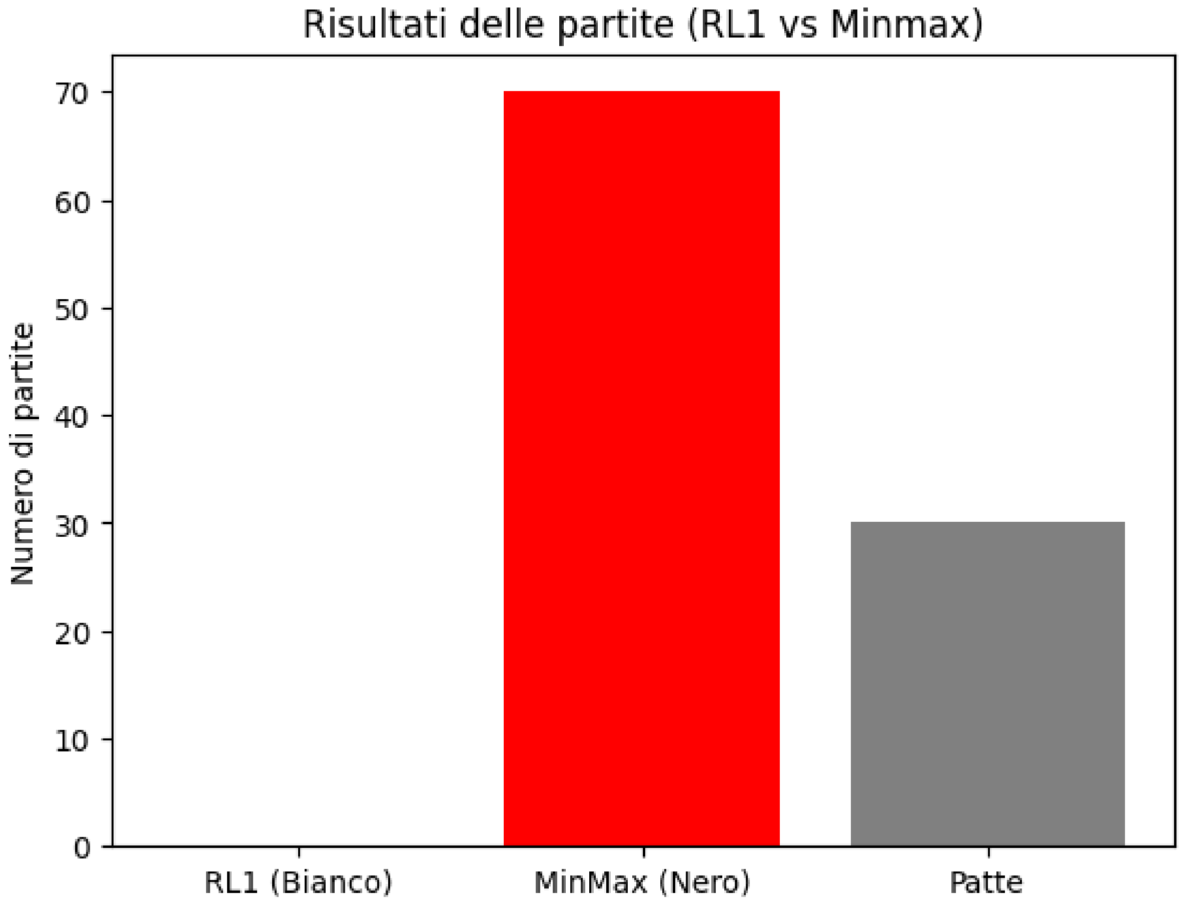

Figure 8: MinMax vs. Regressor

evaluation function, RL1 seems unable to accurately predict long-term moves compared to MinMax, which systematically explores the move space up to a specified depth.

In conclusion, the MinMax algorithm with a depth of 4 proved to be a more robust choice compared to RL1. This suggests that further improvements, such as training on larger datasets and addressing overfitting, are necessary to close the performance gap.

6.4 Key Observations

- **Impact of Maximum Depth:** Increasing the depth from $L = 16$ to $L = 32$ improved accuracy (lower MAE) but significantly increased computational complexity and training time.

- **Dataset Strategy:** Incremental dataset growth proved effective in improving overall accuracy by progressively including more complex positions. However, it requires careful planning to balance resource consumption and processing times.

7 Conclusions

In this work, we developed and analyzed deep learning regression models to estimate chess position evaluations, aiming to enhance the efficiency of the MinMax algorithm with Alpha-Beta pruning. The experiments evaluated model behavior under various configurations, considering accuracy (MAE), training time, and dataset generation time.

7.1 Implications and Future Considerations

The results suggest that a strategy of **progressive complexity increase in the dataset** is effective in improving model performance but must be balanced with computational efficiency. Using a **greater maximum depth** does not always yield accuracy benefits, highlighting that added complexity is not always advantageous without improved training data.

Another key observation concerns the **impact of dataset size**: while increasing the number of chess positions enhances predictive capability, the quality and representativeness of the data are critical. Data quality often outweighs sheer quantity.

Finally, future directions for this work include:

- **Expanding search depth** and integrating advanced pruning algorithms to further enhance efficiency.
- **Training on larger and more diverse datasets**, using data augmentation techniques to increase the variety of chess positions without exponentially increasing training times.
- **Optimizing neural network architecture**, exploring more complex models or regularization techniques to prevent overfitting and improve generalization.

In conclusion, using a deep learning-based regression model to approximate chess position evaluations is a promising strategy for reducing the complexity of search algorithms while maintaining high-quality decision-making. With further development, this approach could lead to significant improvements in the efficiency of intelligent agents, while reducing computational costs in complex game scenarios such as chess.