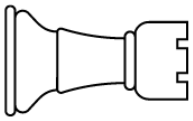
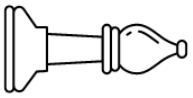
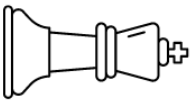


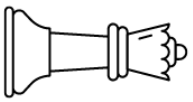
REINFORCEMENT LEARNING

MASTER'S DEGREE PROGRAM IN DATA SCIENCE AND ADVANCED ANALYTICS

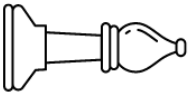




Professor Nuno Alpalhão



Inês Rocha, number: 20220052
Isabel Dias, number: 20191215
Joana Sousa, number: 20191205
Rafael Dinis, number: 20221643



Introduction

The primary objective of this project is to develop an RL agent that can play chess at a high level of proficiency. The agent should be capable of evaluating chess positions and making strategic decisions. However, chess is a complex task to solve, as it has a very high number of potential states, and each state is associated to specific legal actions.

Overall, three distinct approaches were tried to tackle this problem: Monte Carlo Tree Search (MCTS), Deep SARSA and Deep Q-Learning in combination with Imitation Learning. MCTS has a strong theoretical foundation and has been proven to work well in chess. Deep SARSA and Deep Q-Learning allow to handle the computational complexity of the problem, and Imitation Learning allows to leverage knowledge from experts to enhance the agent's decision-making. This allowed for the exploration of different types of reinforcement learning models, on and off policy, and models with and without value function approximation. It was also taken into consideration the complexity and high dimensionality of the space, which would incapacitate the use of certain models without value function approximations.

MCTS

The Monte Carlo Tree Search (MCTS) was implemented as an algorithm designed to be able to play the game of chess. MCTS has been using in similar cases, A Deep Learning enhanced version was used by AlphaGo, by DeepMind, the AI that was able to defeat Lee Sedol [1], the world champion of Go in 2016.

The MCTS is a model-based method, as such, uses a model of the environment for Planning. Typically, it is composed of 4 steps: The Selection, Expansion, Rollout and Backpropagation.

The Selection phase starts model from the root node (current game state) and traverses it until a leaf node is arrived at, where further expansion is possible, the node is selected through the Upper Confidence Bound Policy. After finding the expandable node, the Expansion step starts and a new child node is created, this is done via the expand method of the Node class. This newly created node is the starting point of the Rollout, where the game is played until a terminal state is reached. The moves are randomly chosen until a terminal state is reached. The rewards, that can be 1, -1 or 0 (win, loss, or draw) is then backpropagated, via the rollout(node) function. The Backpropagation step is managed by the backpropagate() function and updates the statistics and values of all nodes visited.

In terms of defined parameters and results, the MCTS was run for 100 evaluations, each composed of 2 games, therefore 200 games total. The iteration size of the MCTS was set to 50. With this conjugation, the algorithm only yielded Losses. One possible improvement while preserving the time spent in computation would be to decrease the number of episodes while increasing the iteration size of the MCTS.

Deep SARSA

SARSA was chosen as an algorithm because although it may not be the most powerful available, it can serve as a solid foundation for initial experimentation with our algorithms. SARSA's temporal difference learning and on-policy nature enable the agent to learn from interactions with the environment, gradually improving its decision-making abilities over time.

As the chess problem involves a high-dimensional state space, it is not feasible to use a traditional Q-table for storing the Q-values. Therefore, an upgrade to deep SARSA is necessary. In deep SARSA, instead of using a Q-table, we utilize a deep neural network to approximate the Q-values.

To make the Deep SARSA, a Q-Network had to be initialized, that is the class Q-Network. After that, the agent class was created, and it has 3 functions. The first function is the function that applies the policy, this is the epsilon-greedy policy, that balances exploration and exploitation by sometimes choosing random actions (based on epsilon) and other times selecting the action with the highest predicted Q-value for a given state. It also has the function of update_Sarsa_Network, that is responsible for update the Q-network based on the target Q-value of the SARSA. Lastly, the update_s creates and prepares the minibatch for the update_Sarsa_Network.

A class called ReplayBuffer was created that helps with storing the transactions and the creation of the minibatches needed to train the Network. This is done the functions add_to_buffer_sarsa and sample_minibatch_sarsa respectively.

Next, it is the functions that plays the games, generate_WHITE_SARSA and generate_BLACK_SARSA. As we want our agent to be able to play in both positions, we created both with slight differences, they work however in the same way. The function resets the environment as we are playing a new game and in case it is the agent turn to play, it gets the next action with the policy of SARSA. After this position is done on the board, the data of the transaction is added to storage if it is not the last play. As it was expected that the algorithm would lose every time, an extra reward was added if the agent was able capture a piece of the opponent. This is added to the final reward that will be returned by this function. In case that it is Stockfish's turn to play, it chooses the best move, plays is and its data is also added to the buffer if it is not the final play. After each play, the state is updated. At the end of a game the network is updated as well as the epsilon value. This function returns the total reward of a game, the number of moves of the agent, the updated epsilon value and the last state.

This algorithm was run for 200 episodes, which returned 400 games, 200 for each colour. The network update was done for 32 rounds with a minibatch size of 64 transactions. The epsilon was started at 0.99.

In our investigation, we discovered some suspicious inconsistencies in the rewards returned by the SARSA algorithm, as well as the outcome of the game. After a thorough examination, it became evident that Stockfish, the chess engine we were utilizing, was not playing on the correct board and it was updating the state wrongly. Despite our efforts, we were not able to correct this error, leading to inaccurate results in the SARSA algorithm. This discrepancy significantly impacted the reliability and validity of our findings. We acknowledge the need for further investigation and resolution of this issue to ensure the accuracy of future experiments.

Deep Q-Learning with Imitation Learning

Deep Q Learning combines deep neural networks with the Q-learning algorithm, allowing to tackle problems with a large number of possible states. This is particularly useful when dealing with complex environments, like chess, where the state-space is very vast, making it unfeasible to store and update the values in a Q-table, which grows exponentially with the number of states.

In Imitation Learning, the agent receives samples of expert behaviour [2] (in our case, chess games between Stockfish and Alphazero[1]) and learns to mimic their policy to maximize return. Given the complexity of the game, it was hypothesized that feeding the model with games between expert players would expedite the convergence of results. This is due to the model having a higher number of samples displaying favourable behaviour in the initial stages. Additionally, pre-training aids in the exploration phase by allowing the model to encounter states that arise from successful policies.

In the article "Reinforcement and Imitation Learning for Diverse Visuomotor Skills" [3] the combination of these two types of learning is explored. And it was found that their combination fared well better than each of them separately. Even though it is not the same problem, the combination of methodologies was deemed valuable for this project.

The model starts by simulating the 110 games imported, and storing them in the replay buffer. The agent then starts to play games randomly (as the epsilon starts at 0.99) against stockfish. All the steps taken are stored in the replay buffer, and at each 10 steps, the model is trained again with a sample of the replenished replay buffer. And the target network updates its weights according to the update rate (5 episodes).

If the action the agent took, from one state to another, results in the opponent losing a piece, the reward is increased by 0.01. The value was chosen so its cumulative sum is quite smaller than the real return. It is so that the agent is encouraged to take the opponents' pieces but should not prioritise that secondary goal over the primary one.

Evaluation

To assess the performance of the three implemented models, a group of features was developed. These were associated with the number of moves per game, the points of material, the remaining number of pieces, as we well as the type of pieces remaining. Metrics of averages, counts and rankings were used to further extract information from the features created.

The Win and Draw Ratios for MCTS and DQN were both 0%, only the Deep SARSA was able to yield a draw ratio of 22.5% as a White Player and 3.0% as a Black Player, while having a 0% win ratio for both (image 1). However, these Draw Ratios might have been caused by the inconsistencies suspected and mentioned in the SARSA chapter.

The variable points of material was created, which maps games according to the values of each pieces in the end of the game:

- Pawn = 1 point
- Knight = 3 points
- Bishop = 3 points
- Rook = 5 points
- Queen = 9 points

The king is the only piece in the game that doesn't have any points associated with it. This is because the king cannot be captured (an attacked king is in check), and also because checkmating the king is the true goal of any chess game.

[1] The pgn file was retrieved from "110 AlphaZero-Stockfish games, starting from the initial board position (.zip file)." Available at: https://www.deepmind.com/open-source/alphazero-resources

By observing both the Points of Material (image 2) [4] or the Number of Pieces (image 3) left of the opponent, we can conclude that the algorithms perform better as the White player (average of around 40 in the MCTS and Deep Q-Learning as opposed to around 20 in the Black player) as there is an inherent slight advantage of starting the game.

Points of Material of the MCTS and Number of Pieces Left of DQN were alternately selected as they are intricately correlated.

Another point of observation is the result that the most common strongest remaining piece of the opponent when using DQN is the rook (image 4), while in the MCTS and Deep SARSA it is the queen (image 5 and image 6). This means that the Deep Q-Learning algorithm was able to, at some level, discern the importance of eliminating the queen.

This can be used as a comparative measure of the quality of the algorithm, as taking the queen can potentially indicate a better strategizing capacity.

Conclusion

All three models yielded no win rates, with only Deep SARSA achieving a draw ratio of 22.5% and 3.0% (white and black player respectively). The advantages in regards to the metrics of White Player could be due to the inherent slight advantage of starting the game, a factor that seemed to benefit the performance of all the algorithms. However, the draw ratios from the Deep SARSA might also have been caused by suspected inconsistencies within the model.

Best Model

We consider the best model to be the DQN as its highest frequency strongest remaining piece of opponent is the Rook (while it is the Queen for the remaining models).

Limitations

Both Deep Learning algorithms would benefit from a higher number of episodes, while the MCTS would mainly benefit from a higher number of iterations.

We could also use a strategy that allows the model to continue the Search tree to be maintained across episodes.

References

[1] Silver, D., Hassabis, D., & Kumaran, D. (2016, March 9). Google artificial intelligence beats expert at Go game. The New York Times. https://www.nytimes.com/2016/03/10/science/google-artificial-intelligence-ai-deepmind-beats-go-champion.html

[2] Stanford AI Lab. (2022). Learning to Imitate. Stanford AI Lab Blog. Retrieved from http://ai.stanford.edu/blog/learning-to-imitate/

[3] Wang, J. X., et. al. (2018). Learning to Reinforcement Learn. Retrieved from https://arxiv.org/pdf/1802.09564.pdf

[4] Chess pieces value. (n.d.). In Chess.com terms. https://www.chess.com/terms/chess-piece-value

Annexes

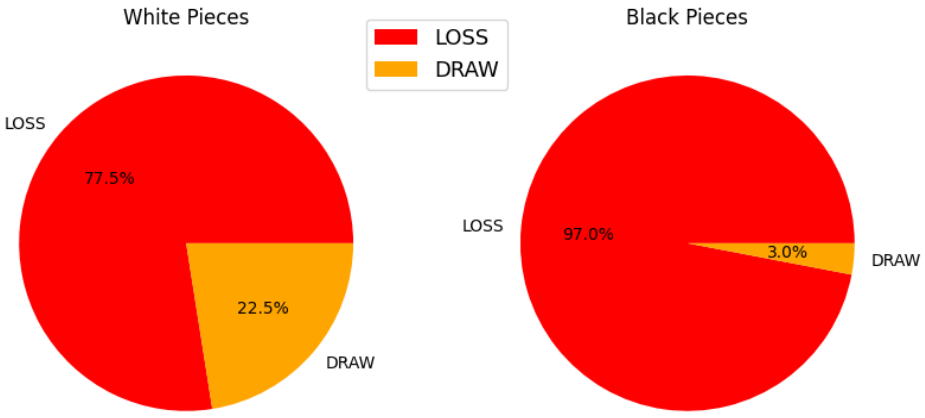


Image 1 - Pie Chart with the Outcome Ratios by White Pieces and Black Pieces

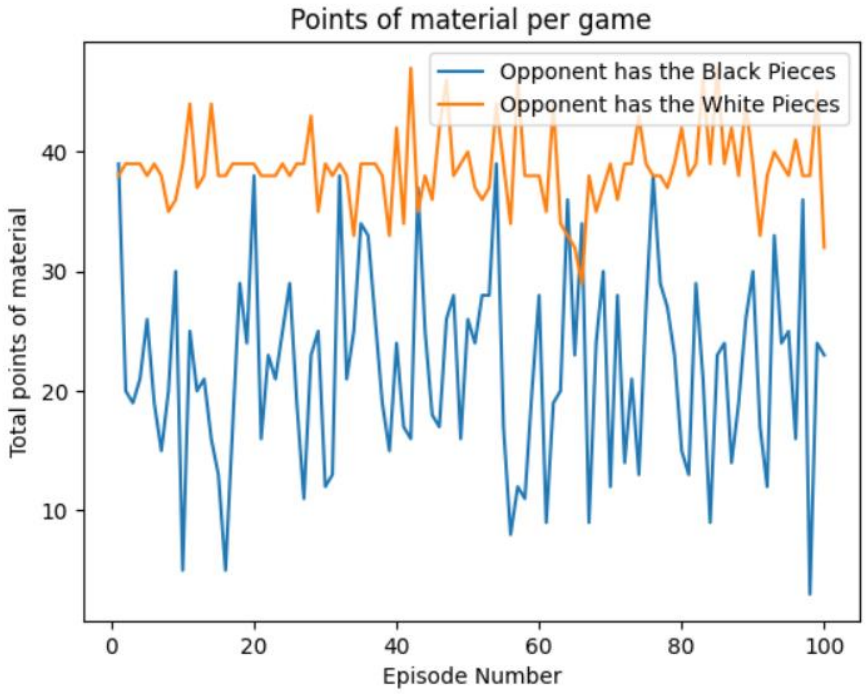


Image 2 - Points of Material per Game by Episode for the MCTS Model

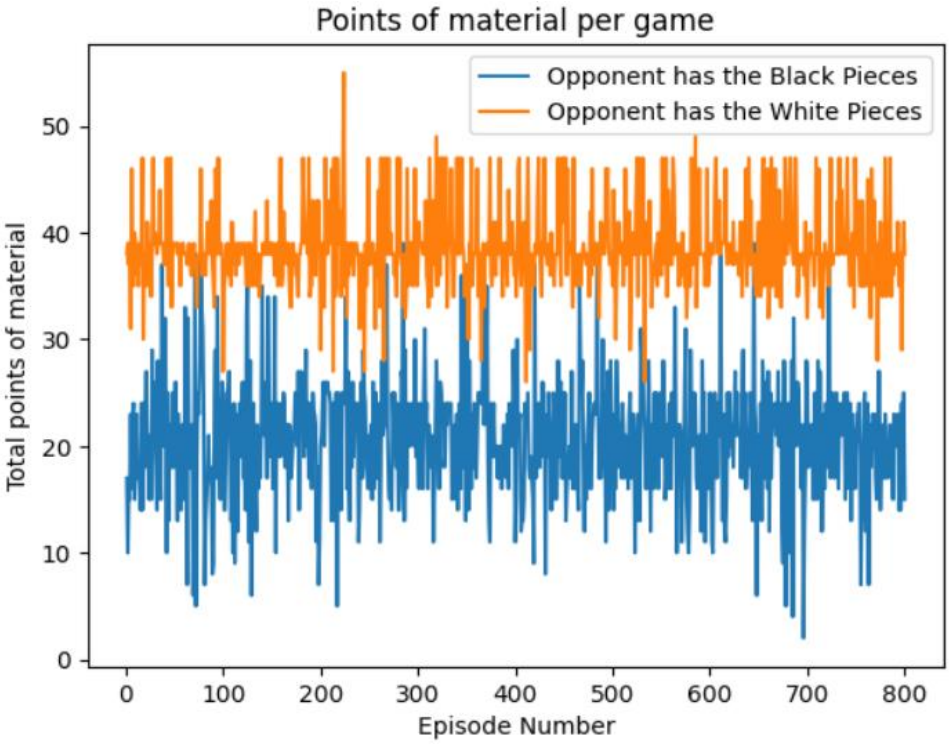


Image 3 - Points of Material per Game by Episode for the DQN Model

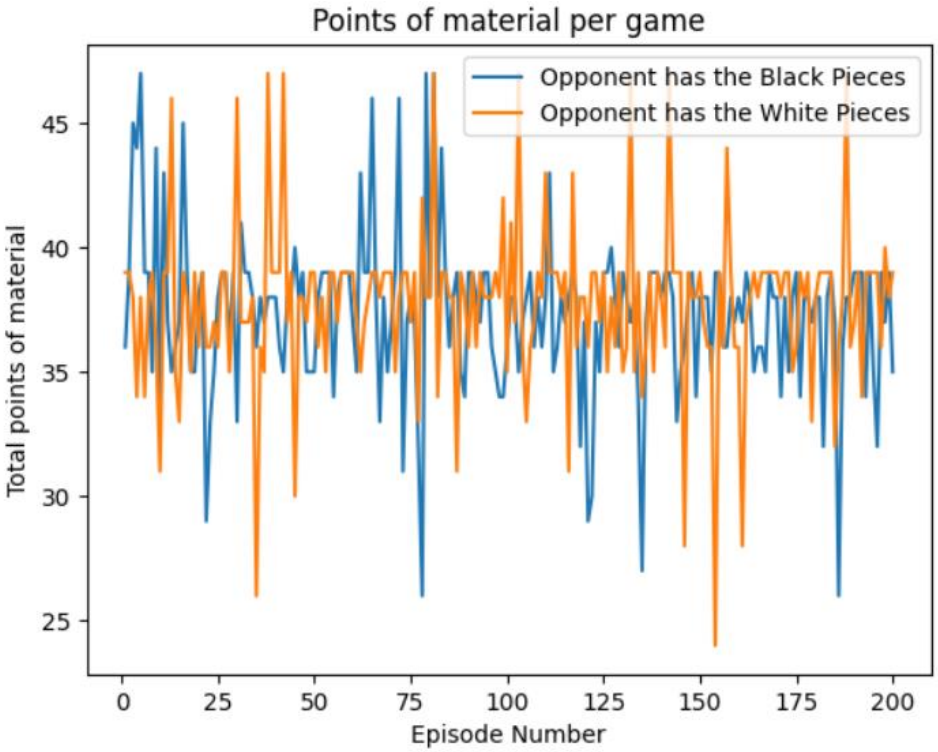


Image 4 -5oints of Material per Game by Episode for the Deep SARSA Model

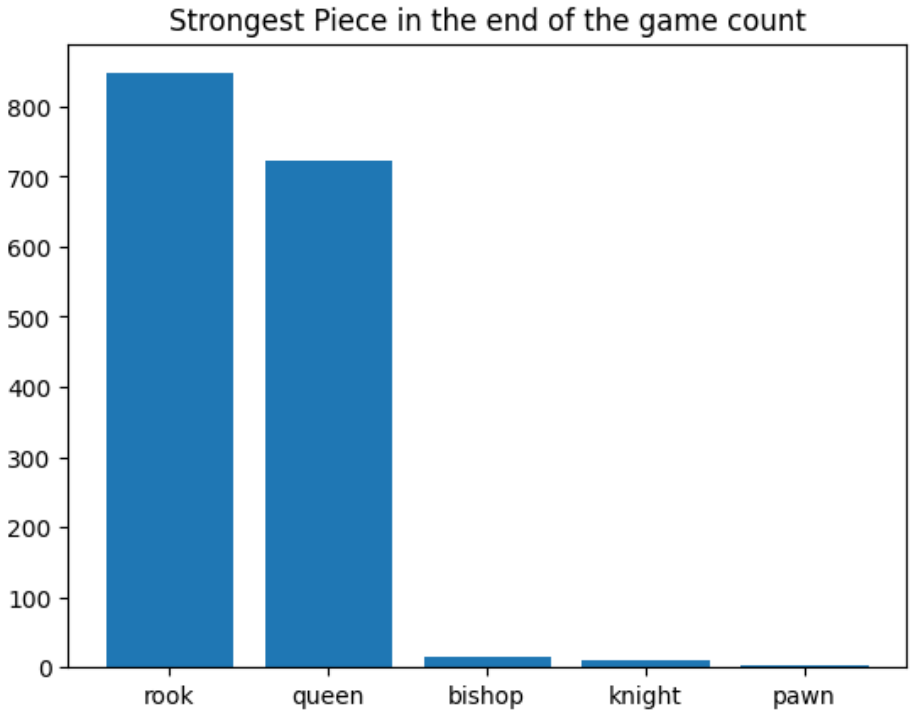


Image 5 – Histogram of Count of Strongest Piece Remaining - DQN

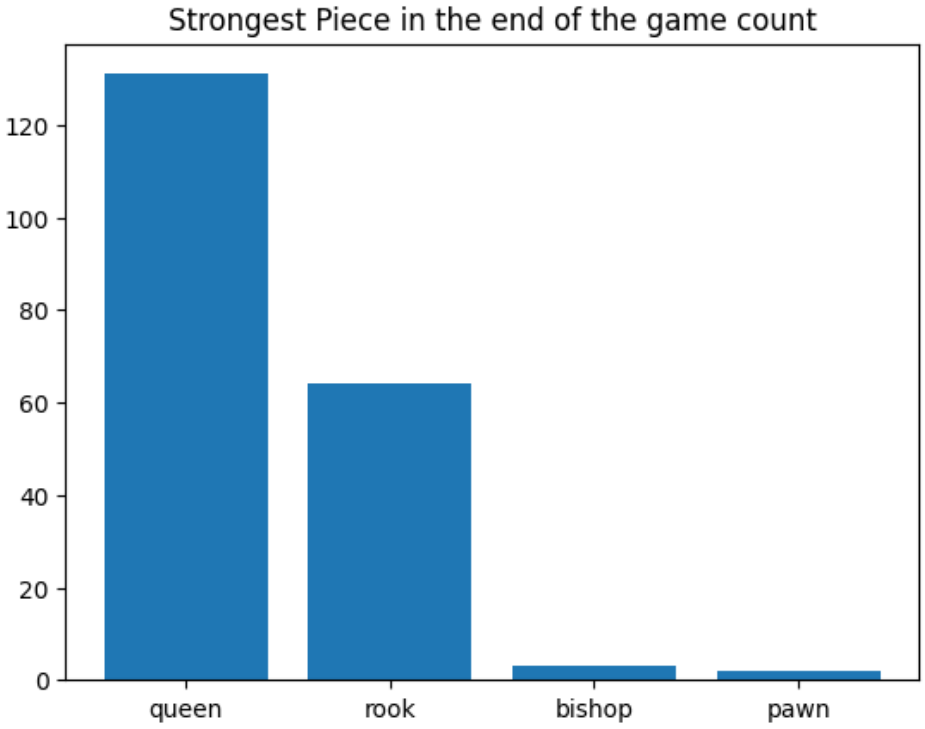


Image 6 – Histogram of Count of Strongest Piece Remaining - MCTS

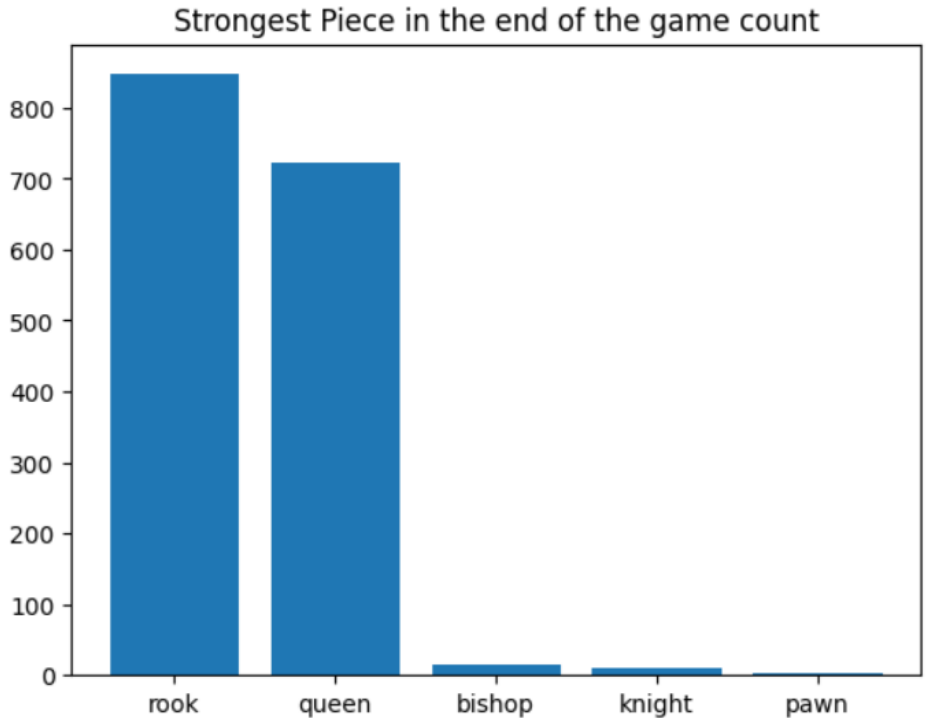


Image 7 – Histogram of Count of Strongest Piece Remaining – Deep SARSA