NOVA IMS Information Management School

Reinforcement Learning Project

MASTER DEGREE PROGRAM IN DATA SCIENCE AND ADVANCED ANALYTICS

Chess: A Reinforcement Learning Approach

Felix Gaber, number: 20221385

Jannik Kickler, number: 20220630

Rita Soares, number: 20220616

June, 2023

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

TABLE OF CONTENT

1. Introduction

Chess, one of the oldest and most popular board games, requires a logical approach that relies on strategy and the current state of the match. Known for being one of the most challenging games, it is understandable the interest in building an algorithm that could not only play but defeat a human as well. For many years, the approach was based on hard-coding the rules and the strategies one can take, given the opponent's move and the current board. However, for a good approach, one should consider the complexity of chess, a board of size 9x9 with a total of 16 pieces per player, six distinct ones, each with their own movement rules, summarizing around 10^8 strategies available. It's easily understood that traditional approaches are not efficient in this case.

The Reinforcement Learning (RL) field englobes algorithms that learn from experience. An agent takes a decision and learns from it through cumulative reward in a way that its main objective is to maximize it; this offers a way to make "thoughtful" decisions without resorting to brute force coding. The application of this field to chess has already been proven beyond successful, as one would expect.

In this project, different RL algorithms were built to learn how to play chess; the final purpose is to understand better how reinforcement learning works and its promising perspectives for the future.

2. Model Deployment and Evaluation

We thought about the best method to train the agent. We quickly encountered methods such as Deep Q-Learning and Monte Carlo Tree Search (Silver et al., 2017). Considering that the state space is far too large for methods like Monte Carlo or Q-learning, we still started with these methods to learn the practical basics of Reinforcement Learning and the chess environment of OpenAI. Afterward, we will use a Deep-Q Neural Network to train an agent. In all different methods, the agent will be tested and evaluated against two opponents: a Random player who chooses only random actions out of the action space and the Stockfish algorithm with the lowest ELO rating of 1.

We also considered whether it would make sense which data we would best use for learning our agent. We came up with different possibilities: Depending on whether our agent plays against itself, against random, or Stockfish during training, we could either use one player side, the other player side, or both at the same time to update our Q-table or DQN. The method with the best results would probably have been updating our agent based on the actions and rewards of the Stockfish. However, we did not consider "copying" the Stockfish as a clean way to train our agent, so we decided only to use the information to train our agent that it learns through its exploration of the environment.

In the given chess environment, there are 119 chess boards with dimensions of 8x8. Among these boards, the first 12 represent the current state of the environment, with the positions of all the current pieces on the board. The remaining boards represent past states and contain additional information, such as player color. Upon exploring the environment, we discovered that boards 0 to 5 always represent the enemy's perspective. On the other hand, boards 6 to 11 always depict the current player's pieces in every state. This behavior of the environment allowed us to easily define tracking variables to evaluate the learning progress of our agents, which will be discussed in detail later. When implementing the training algorithms for our agents, we needed to store information about the states and associated actions. Considering the large number of boards and our decision only to utilize the first 12 boards, as they contain all the necessary information for our approaches, we sliced the states to include only these 12 boards that accurately represent the current board state and piece positions.

Regarding the policy we used for all our agent approaches, the epsilon greedy policy, so that our agent can balance between exploration and exploitation. We restricted the policy only to return legal actions to ensure the agent's actions are valid within the current environment. To configure the epsilon decay, we implemented a calculation method to divide the value one by the total number of episodes. This way, the decay will always be about the number of games in one run.

Considering the reward system provided by the environment, where a winning game results in 1 point, a draw gives 0 points, and losing leads to -1 point. Recognizing that winning against Stockfish would be impossible with the provided resources, we aimed to enhance the learning process of our agent with additional rewards. To achieve this, we introduced a reward for each agent's step. By doing so, we intended to encourage the agent to focus on staying as long in a game as possible, thereby increasing its learning opportunities. Additionally, we implemented a reward for capturing the opponent's pieces. To pick the appropriate points to assign for each captured piece, we used the official point system from chess, where pawns are valued at 1 point, bishops at 3 points, rooks at 5 points, and queens at 9 points. As the agent already receives a reward of 1 when capturing the opponent's king, we assign it a quantified value of 0 in this point dictionary. With this approach, we hope the agent will learn as it receives additional rewards for progressing in the game and strategically capturing the opponent's pieces.

For evaluation purposes, we initialize multiple tracking lists and dictionaries in all methods to store the data obtained over time. We use these in the following visualizations:

- The **plot_steps** function shows the steps achieved in each episode in blue and the binned average in red, giving us better information on whether the steps increase over time.
- The **plot_points** function shows how many total points were scored in the episodes. Again, the same bucket size mechanism is built in for the average, with the number of bins fixed at 10. The total score is calculated by summing the win bonus (whether we win/draw/lose with a value of 1/0/-1), the step bonus, and the points accumulated by beating the opponent's pieces.
- The **plot_losses** function displays a stacked bar chart consisting of two sub-charts - one for the agent's losses and one for the opponents'. Each column in these charts represents a period or group of episodes. The height of each column indicates the total number of tokens lost during that period. The different colors within each column represent the different types of tiles that were lost, and the height of each colored segment within the column shows the number of tiles of that type that were lost.

Note: Due to the many episodes and variance within episodes, average bins are built into all three visualizations to determine better learning over time. The number of buckets is hardcoded to 10, so the bucket size is automatically calculated from the number of episodes divided by 10.

Before going into the various agent approaches, it is essential to familiarize the reader with a list of global variables that are important in understanding the basis of our algorithms:

- *STEP_REWARDS* (=0.00001): This variable represents the reward value given to the agent for each step it remains in the game. It is intentionally set to a small value to make it relatively insignificant compared to the reward for winning the game.
- *PIECES_REWARDS_MULTIPLIER* (=0.001): This variable is multiplied by the respective points defined in *POINTS_DICT* for the game pieces. It helps keep the reward relatively small compared to the reward for winning the game.
- *MAX_EPSILON* (=1): This variable represents the initial value of epsilon used for exploration in a reinforcement learning algorithm. Epsilon balances exploration (random actions) and exploitation (optimal actions).
- *MIN_EPSILON* (=0.01): This variable represents the end value for epsilon used in exploitation. As the algorithm progresses, epsilon decreases gradually to prioritize exploitation over exploration.
- *POINTS_DICT*: This dictionary assigns points to the figures (Pawn: 1, Bishop: 3, Rook: 5, Queen: 9). It allows an assignment of specific rewards to the agent depending on the type of opponent's game piece it successfully defeats.
- *PIECES_QUANTITY*: This dictionary defines the initial quantities of all tiles. It allows for better evaluation and obtaining rewards for the agent when the quantities of the tiles change, either for the agent or the opponent.
- *PIECES_NAMES*: We have also defined a dictionary called *PIECES_NAMES*. It contains the names of the game pieces, which are useful for subsequent visualization purposes. This dictionary allows us to accurately display and identify the game pieces that the agent or the enemy has defeated during gameplay.

Lastly, it is to mention that after every 50 episodes, the Q-table is saved for Monte Carlo and Q-learning, and the weights of the main model are saved for DQN. We employ a backup mechanism to safeguard against data loss while importing and updating the file.

2.1 Monte Carlo

The Monte Carlo with Every Visit algorithm consists of an episodic-based model-free approach, whose learning is based on a trial and error approach, playing multiple games with an epsilon probability of taking a random action and at the end of each episode building an average of the cumulative reward per state given an action visited.

For the development of this logic, lists were applied to store the following information after each step played by our agent: state, action, and total reward. At the end of each episode, the update of the Monte Carlo is done by accessing the stored information, performing the average accordingly, and updating the Q-table dictionary.

Despite, for this study, only the Every Visit being deployed, the code is generalized to also perform the First Visit method with a simple change in the input parameters when calling the function.

As it's possible to observe in Figures 1 and 2, playing against Stockfish, our model is not performing well; this result is expected since Monte Carlo is a slow learner and a relatively simple algorithm. Despite that, this method is an excellent base to understand the field better and evolve to more complex models.

Since beating Stockfish is a big challenge, another round of episodes was played against random by Figures 3, 4, and 5. It’s possible to conclude our model is just too weak to go against Stockfish.

2.2 Q-Learning

Q-Learning uses the Markov decision process, despite not knowing how the environment behaves until the learning process starts with the goal of reaching the optimal policy, which is an excellent approach since model-based approaches cannot perform on such a large state and action space. For the deployment of this algorithm, a learning rate of 0.7 and a discount factor of 0.95 were considered.

In Q-Learning, it is important to note that we used the epsilon greedy policy to prevent the overestimation problem of Classical Q-Learning.

We use structures like dictionaries and lists to store the needed variables through the steps. After each step, we update the Q-table and some evaluation information with the new values by state and action.

Despite the high expectations that Q-Learning, a more complex algorithm than Monte Carlo, would yield improved performance, the obtained results, as depicted in Figures 6, 7, and 8, showed otherwise. By comparing the outcomes of the plays against the Random agent - Figures 9, 10, and 11 - and Stockfish, it became evident that the learning process did not improve. Even after playing 3,000 games against Stockfish, the average step size remained consistently around 38 steps, and the total points acquired did not show any notable increase throughout the games.

2.3 Deep-Q Neural Network

In this approach, we employ a standard Deep-Q Neural Network consisting of a main agent and a target agent. The DQN utilizes two neural networks, each serving a specific purpose. The main agent makes decisions based on the current state and is trained to produce the optimal state-action value. The target agent, on the other hand, is a copy of the main agent, which creates target Q-values. These target Q-values are used to calculate the loss of the main network to update the weights during the training process.

To implement the approach, we developed a customized class. For the architecture of the DQN, we decided to use a straightforward deep structure, which consists of a sequential model with linear layers and incorporates a rectified linear unit (ReLU) activation function between each layer. As an optimizer, we chose Adam and Mean Squared Error as the loss calculation for training the main agent.

We evaluated our agent against the random opponent and Stockfish as in the previous approaches. After conducting 1,000 games against a Random opponent - Figures 15, 16, and 17, we saw a notable increase in our agent's steps taken per episode by our agent. This outcome indicated that our strategy of rewarding the agent for taking more steps had been successful. Furthermore, the cumulative positive points across all episodes also displayed a consistent upward trend. Encouraged by these achievements, we tested our DQN against the Stockfish agent - Figures 12, 13, and 14, hoping to witness similar progress. In the initial games' results, we resembled those obtained from Monte Carlo and Q-learning approaches. Consequently, we decided to continue running the neural network for additional games, anticipating that the learning process might eventually converge. Unfortunately, even after completing 8,000 games, we observed no significant learning effects. On average, the agent maintained consistent steps throughout all episodes and could only marginally increase its positive points by capturing enemy pieces. Our agent has yet to achieve a single draw or victory against Stockfish.

4. Conclusion

Although the initial results of the learning process were not as robust as we had hoped, we observed some promising signs of progress when the system played against random opponents. Although we were aware of the high sophistication of the Stockfish algorithm, which can outperform even the best chess players, we hoped for a win in at least one of the 8,000 games with our best agent (DQN) against an ELO level 1 opponent.

As we continue to explore this project, we are excited about the potential to test different DQN architectures, such as Convolutional Neural Networks. In addition, we believe that including the not used states of the environment could provide valuable insights to improve the learning process.

The implementation of Monte Carlo Tree Search, having more time, would have been our next step. Looking at the success of the Deepmind agent AlphaZero, we are optimistic that this method could be an extremely effective strategy for playing against Stockfish. In addition, we are very interested in experimenting with new forms of rewards and testing different strategies. We are confident these approaches will allow us to progress significantly on our project.

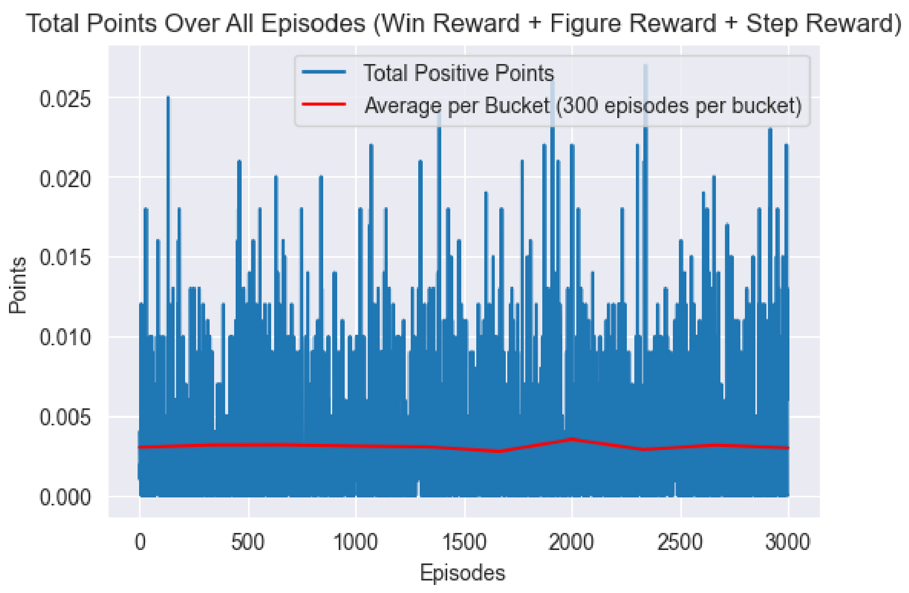
5. References

[1] Richard S. Sutton and Andrew G. Barto, "Reinforcement learning: An Introduction", Second Edition, MIT Press, 2018

[2] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

6. Appendix

Monte Carlo Against Stockfish:



7

Figure 1: Total points over all episodes on a total of 3000 for the Monte Carlo algorithm playing against Stockfish.

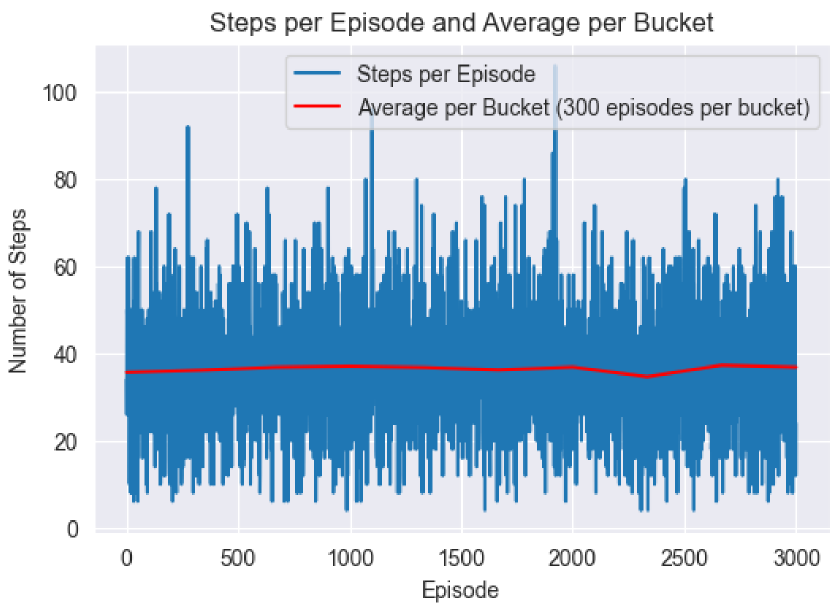


Figure 2: Steps per Episode and average per bucket on a total of 3000 for the Monte Carlo algorithm playing against Stockfish and 300 episodes per bucket.

Monte Carlo Against Random:

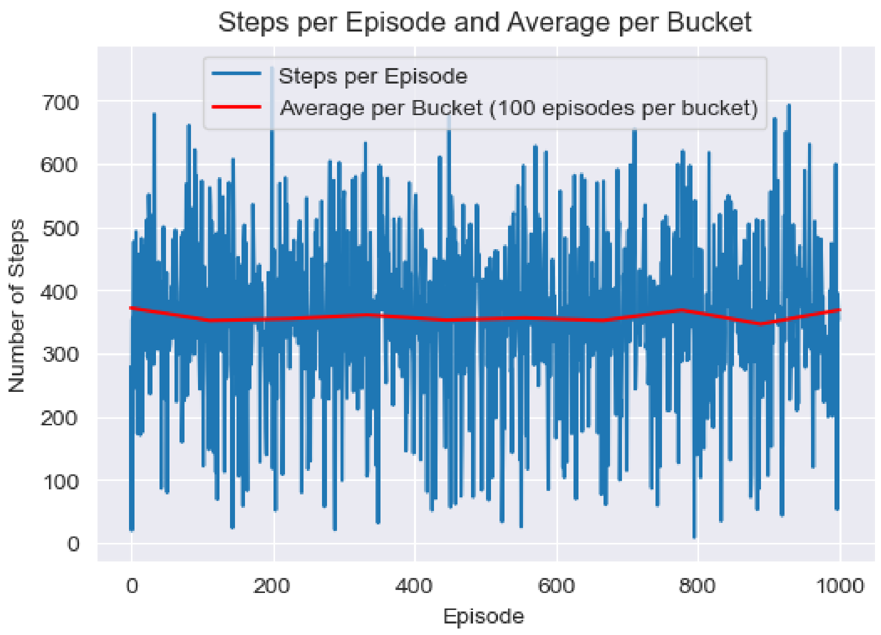


Figure 3: Steps per Episode and average per bucket on a total of 1000 for the Monte Carlo algorithm playing against Random and 100 episodes per bucket.

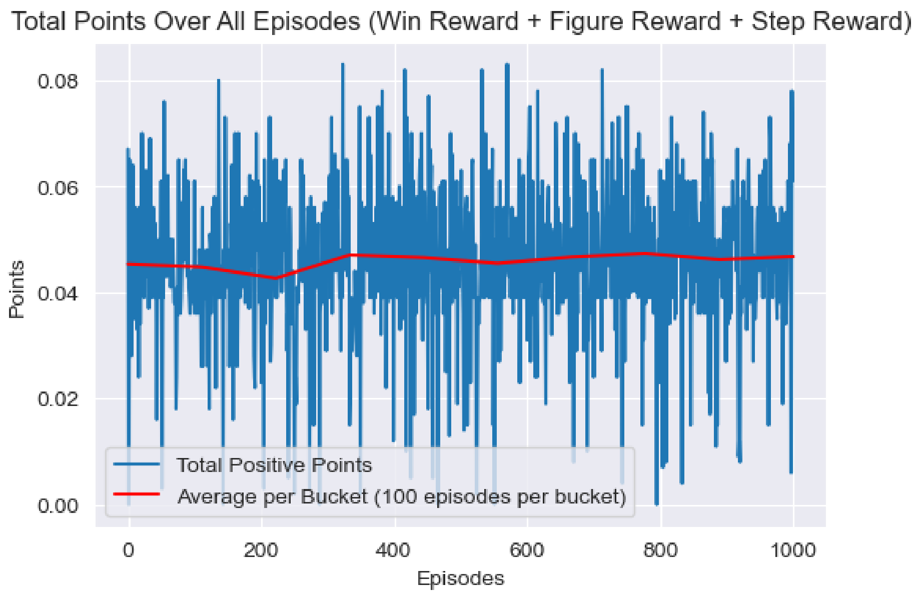


Figure 4: Total points over all episodes on a total of 1000 for the Monte Carlo algorithm playing against Random.

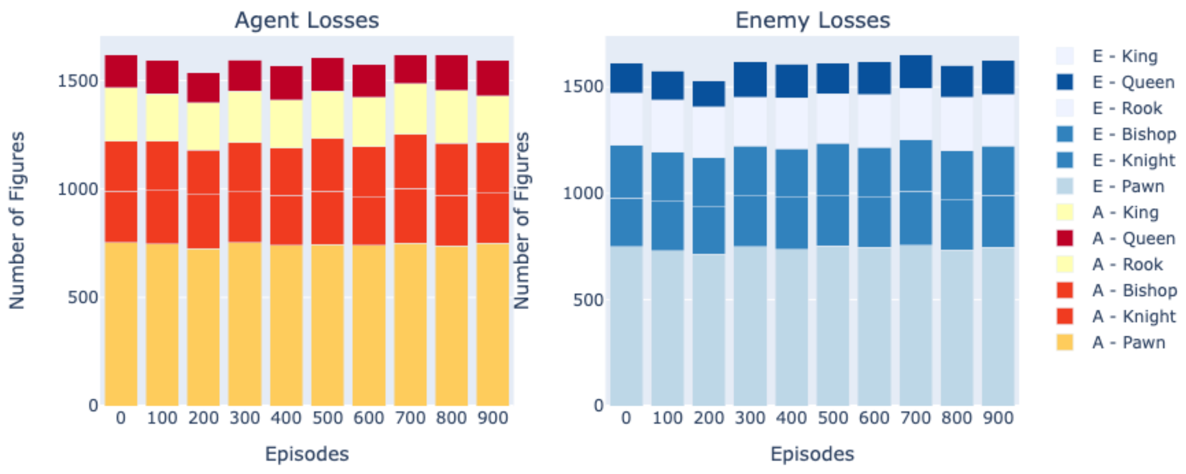


Figure 5: Number of figures lost per Agent and Enemy on Monte Carlo against Random on a total of a 1000 episodes batched in 100.

Q-Learning Against Stockfish:

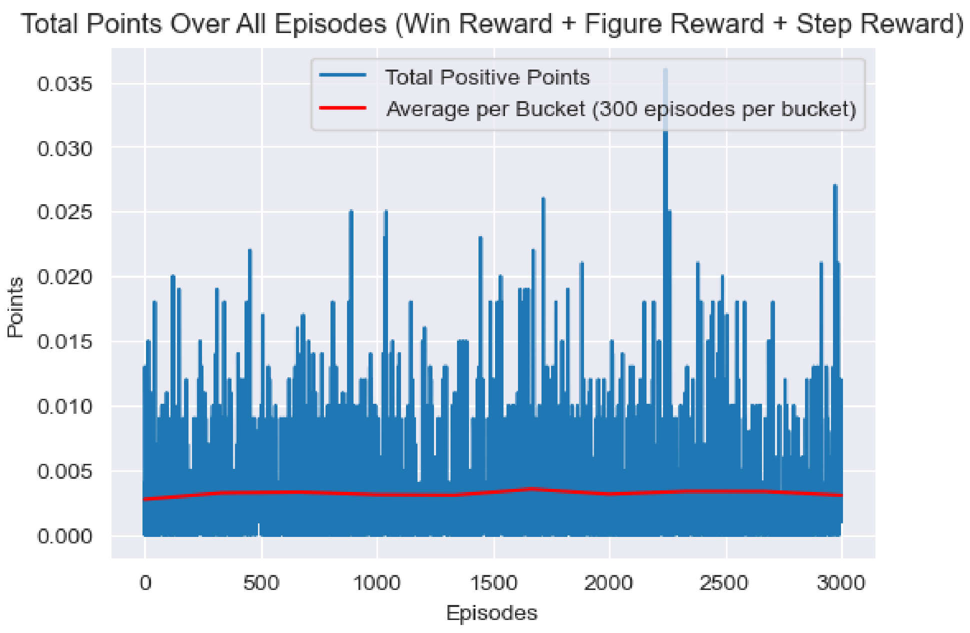


Figure 6: Total points over all episodes on a total of 3000 for the Q-Learning algorithm playing against Stockfish.

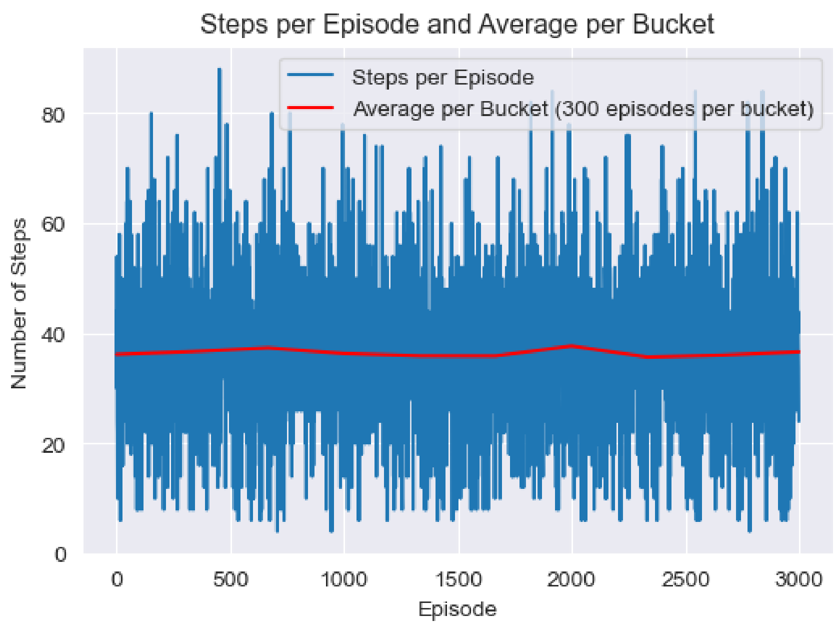


Figure 7: Steps per Episode and average per bucket on a total of 3000 for the Q-Learning playing against Stockfish algorithm and 300 episodes per bucket.

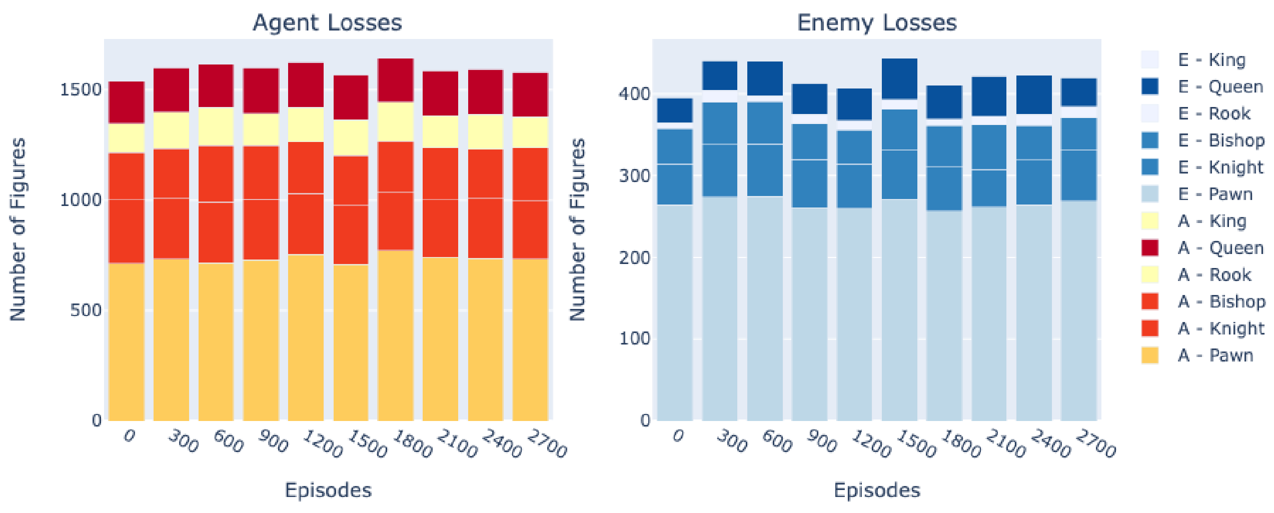


Figure 8: Number of figures lost per Agent and Enemy on Q-Learning against Stockfish on a total of a 3000 episodes batched in 300.

Q-Learning Against Random:

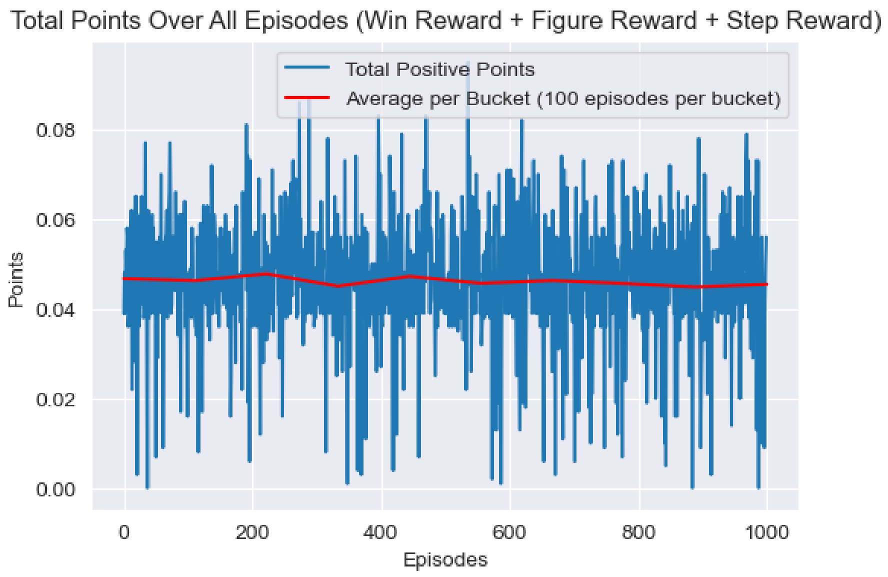


Figure 9: Total points over all episodes on a total of 1000 for the Q-Learning algorithm playing against Random.

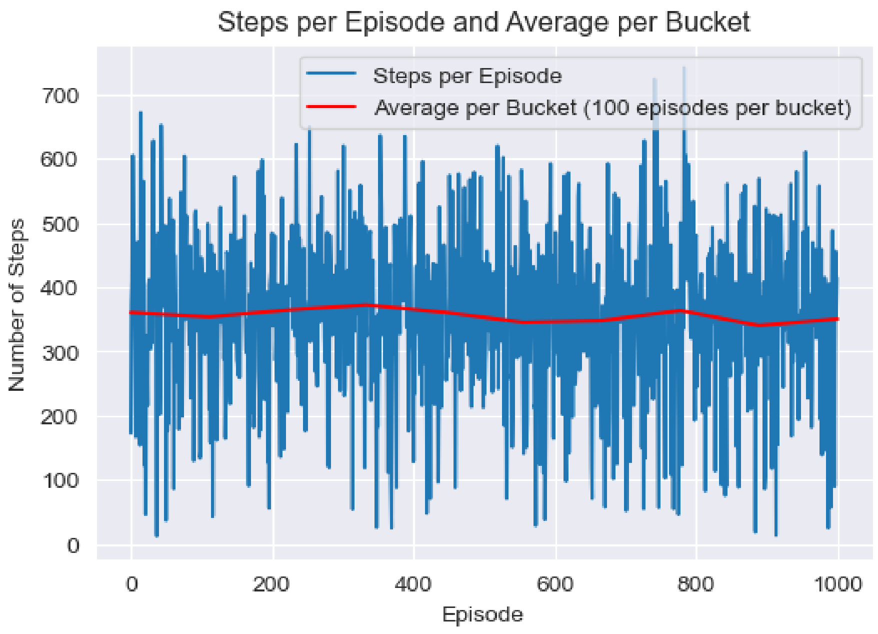


Figure 10: Steps per Episode and average per bucket on a total of 1000 for the Q-Learning playing against Random algorithm and 100 episodes per bucket.

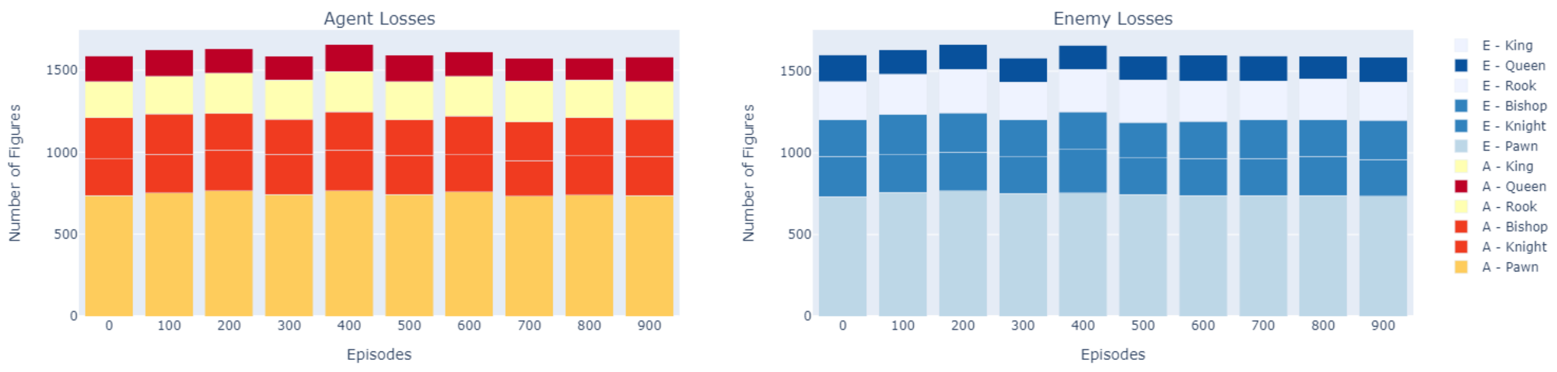


Figure 11: Number of figures lost per Agent and Enemy on Q-Learning against Random on a total of a 1000 episodes batched in 100.

Deep Q-Network Against Stockfish:

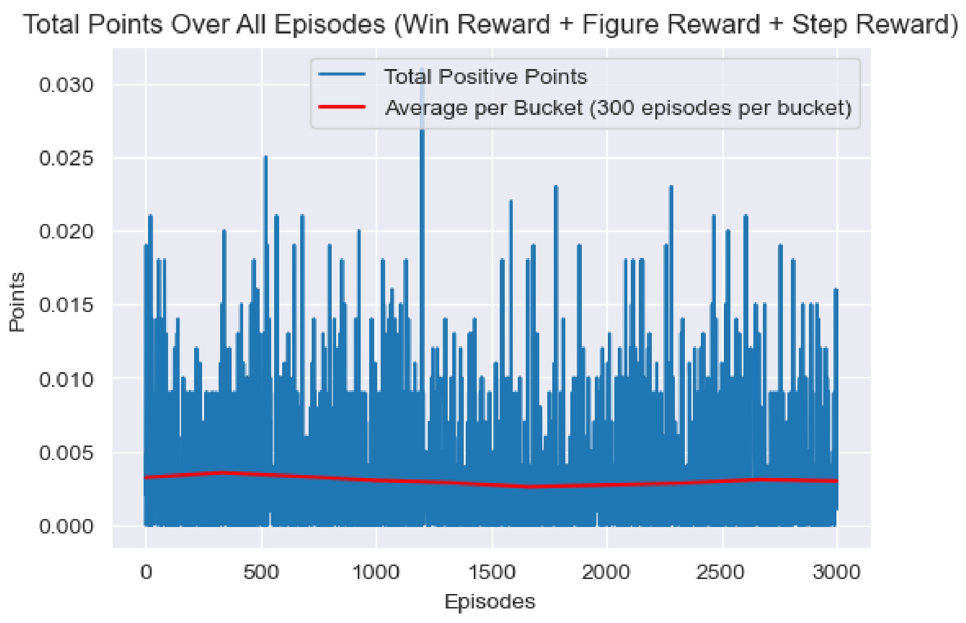


Figure 12: Total points over all episodes on a total of 3000 for the Deep Q-Network algorithm playing against Stockfish.

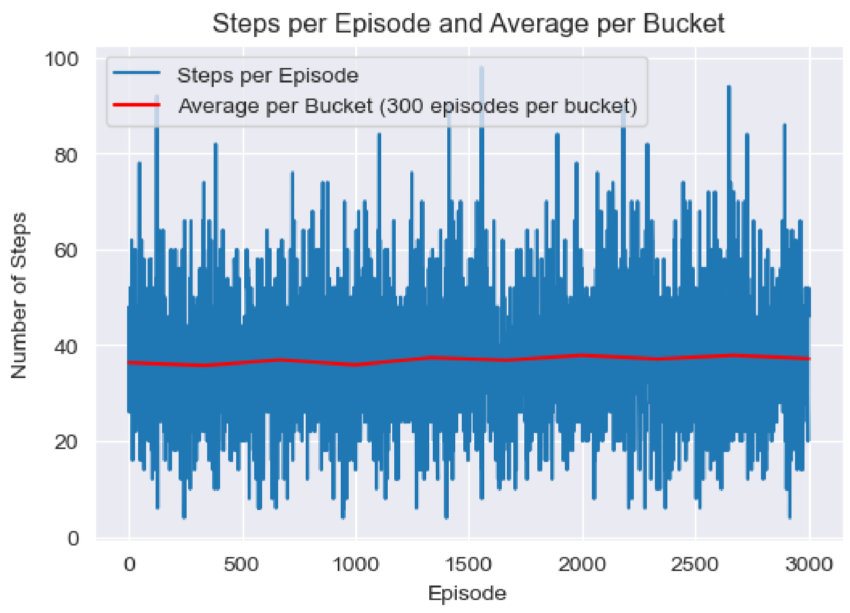


Figure 13: Steps per Episode and average per bucket on a total of 3000 for the Deep Q-Network playing against Stockfish algorithm and 300 episodes per bucket.

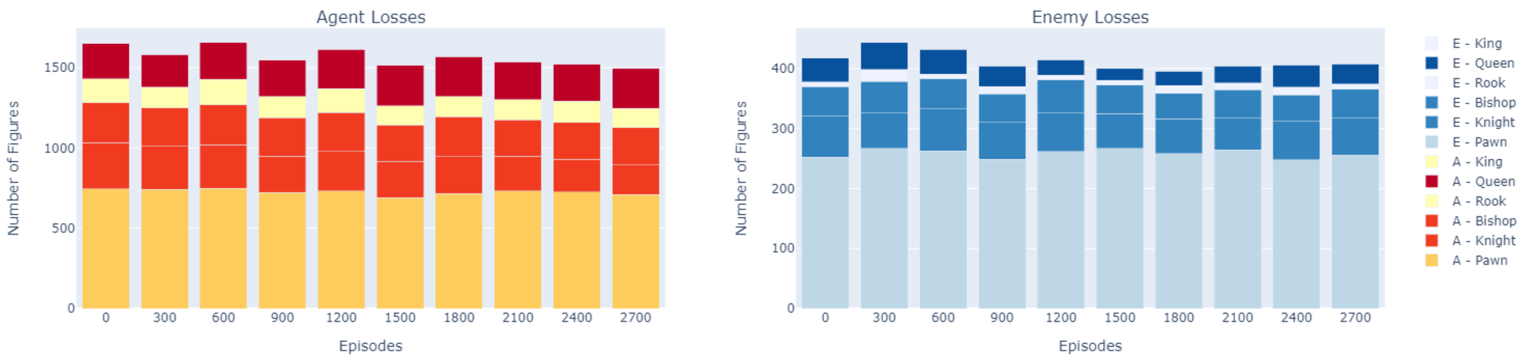


Figure 14: Number of figures lost per Agent and Enemy on Deep Q-Network against Stockfish on a total of a 3000 episodes batched in 300.

Deep Q-Network Against Random:

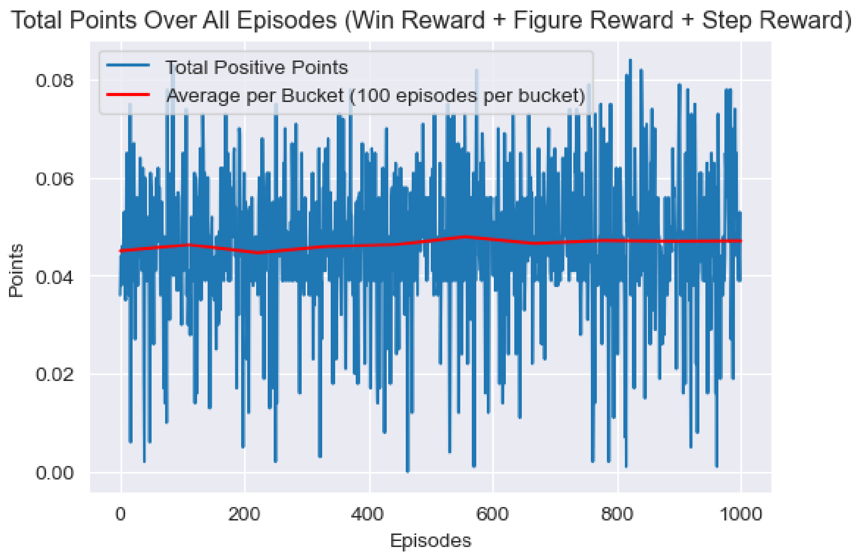


Figure 15: Total points over all episodes on a total of 1000 for the Deep Q-Network algorithm playing against Random.

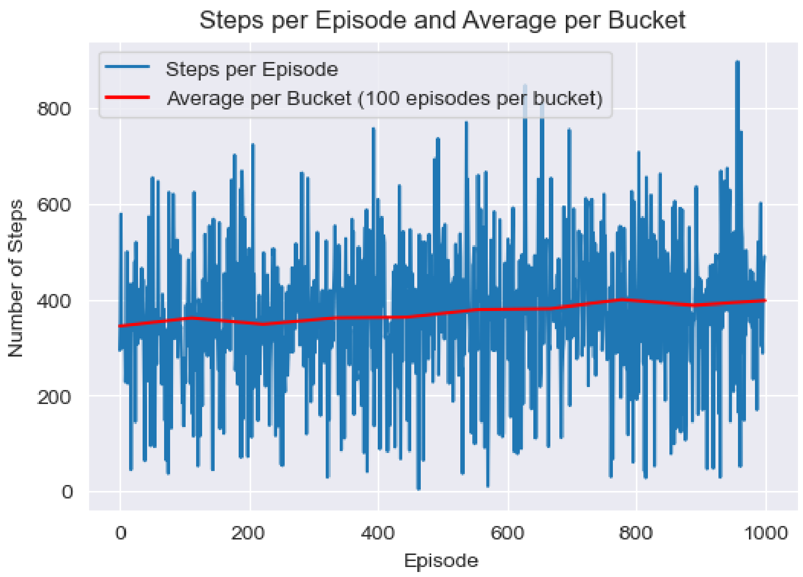


Figure 16: Steps per Episode and average per bucket on a total of 1000 for the Deep Q-Network playing against Random algorithm and 100 episodes per bucket.

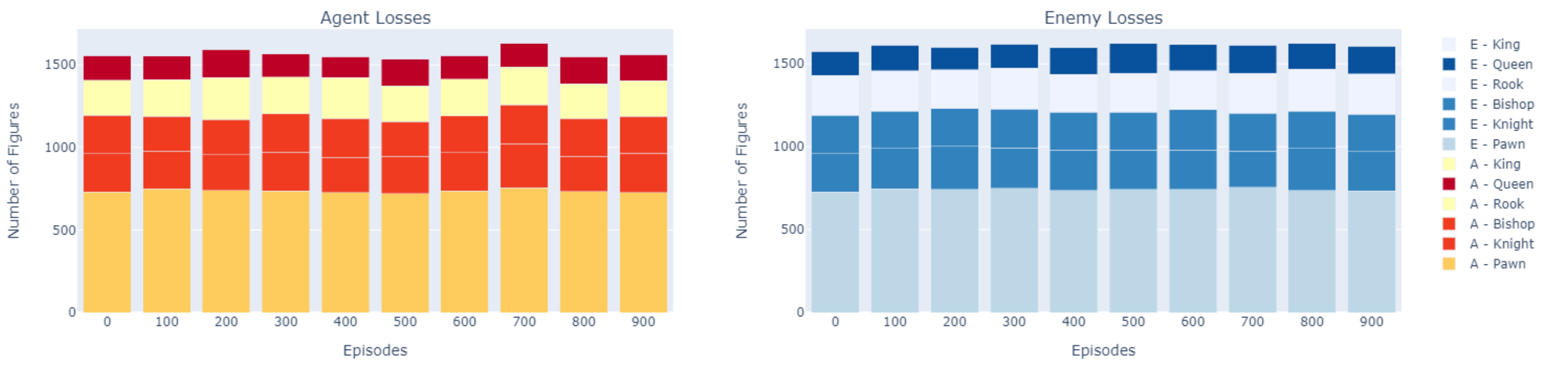


Figure 17: Number of figures lost per Agent and Enemy on Deep Q-Network against Random on a total of a 1000 episodes batched in 100.