Solving Chess Endgames Using Q Learning

Bachelor's Project Thesis

Antonio-Ionut Boar, s3656217, a.boar@student.rug.nl,
Supervisor: Dr M. Sabatelli

Abstract: Chess is one of the oldest and consistently popular games in human history. The same is reflected in computing history, with attempts getting better and better at developing machines that can play the game. However, most of the academic literature centers around more complex solutions than simple Reinforcement Learning. Therefore, this research reduces the scope and complexity, aiming to explore the ability of Reinforcement Learning algorithms, Q Learning specifically, to learn how to checkmate in a winning endgame scenario. This paper shows that a Q Learning agent can yield results that are worse, but comparable to more advanced chess engines in the well-known endgames of King and Queen, King and Rook and King and Two Bishops.

1 Introduction

Chess is a strategy board game played by two players, typically using white and black pieces respectively, with a relatively simple set of rules that combine to form complex strategies and an immense number of possible states. It is considered an abstract strategy game with no hidden information, since, as the name suggests, it is a game of strategy involving little to no narrative theme, taking place in a system which is perfectly known at every step by both players involved ("Chess", 2022). While it is considered an ancient board game, the most commonly known form only appeared in the 15th century, with organized competition starting in the 19th century. It is played on a board with 64 squares in an 8 by 8 grid, with each side having 16 pieces: 8 pawns, 2 knights, 2 bishops, 2 rooks, a king and a queen. Players take turns, starting with the player with the white pieces, until either one player wins or a draw is reached.

There are many types of chess engines out there, such as Stockfish, AlphaZero, Leela Chess Zero (e.g. "Engines", 2022; "Chess Engine", 2021), all of which have reached such a level of proficiency that it has become impossible for humans, even the best in the world, to defeat them unless skill level limitations are applied to the engines. The first time a computer beat the top chess player at the time was in 1997 when Deep Blue managed to defeat GM Garry Kasparov, after first losing to him in 1996 ("Chess Engine", 2021). Chess engines have since become much better and more varied, but one that has been popular, due to its open source nature, and very performant, with how many engine competitions it managed to win ("Tournament Results", 2022), is Stockfish (Romstad, 2011). Therefore, Stockfish was chosen as the engine against which the training was performed. This is to ensure that the agent being trained is able to perform properly even against an opponent who plays optimally, requiring it to also play optimally as a consequence in order to win.

Chess Endgames are a subset of possible positions in chess, in which the outcome can be determined through optimal play. Over time, optimal play has been expanded upon, new strategies developed, and so the positions that can be considered endgames have also expanded accordingly (e.g. Müller and Lamprecht, 2001; Dvoretsky, 2020). However, for the purpose of this research, only three of the simple, well-known endgames were chosen: King and Queen, King and Rook, and King and Two Bishops. As the names suggest, these endgames only have a few pieces left on the board: the ones in the name for one player, and the king for the other. These endgames have a few advantages, which is why they were chosen. First, the winning side has a clear advantage without the need for pawn promotion or complex sacrifices. Second, when people are taught how to play solve these endgames, the simplest approach is using some rules of thumb for each, which, even if not the most efficient in some situations, always lead to a forced win.

Reinforcement Learning is a category of algorithms that is widely popular as a way to create artificial intelligence capable of, among other purposes, learning to play games, with many books written on the subject (e.g. Busoniu, 2010; Sutton and Barto, 2018; Plaat, 2020). In short, the aim of these algorithms is to create an agent able to map actions to numerical reward values, in order to maximize said rewards when interacting with its environment (Sutton and Barto, 2018). Through trial and error, the agent learns to approximate what reward it is going to get, not just at the time of choosing an action, but also what rewards are possible as future consequences of the current decision. For this research, a slightly modified version of the Q Learning algorithm was used, mostly due to its popularity and ease of applicability. Using this algorithm, the agent learns to directly approximate the reward it is able to get by taking action a (a chess move) at state s (the placement of the pieces on the board), usually referred to as $Q(s, a)$.

With all of that in mind, this research will try to answer the following questions: Can a simple approach to Q Learning be used for solving chess endgames? If so, which reward functions work best and is that performance consistent across all of the chosen endgames?

As this paper will show, the Q Learning agent can learn to solve different endgames in a relatively short amount of training time (5000 matches), but the results deteriorate as the agent's advantage is smaller and as the number of the moves required to win increases. In some rare cases, the Q Learning agent actually reaches solutions in a lower number of moves than Stockfish at default settings, and for one position it managed to learn how to solve it even when the Stockfish agent reached a draw by repetition.

2 System Description

The system was designed in the python programming language with a modular approach in mind. It features both a Graphical User Interface (GUI) and a command-line version (code: Boar, 2022). The former was used in the initial development of the reward functions and in the validation of results by observing the agent(s) playing, while the latter was used for data collection. The core loop of the program is to ask, at each time step, the agent whose turn it is, to choose a move to make, change the state of the board, and repeat until a desirable end state is found (Algorithm 2.1). To facilitate this process, the python-chess (Fiekas, 2022) library was used. For the Stockfish evaluation and finding of the moves for the Stockfish agent, the stockfish library was used in connection with the Ubuntu version of the Stockfish engine (Romstad, 2011) using default settings.

Algorithm 2.1 Main loop

```
Require: agent_white, agent_black, board initialized
  while not DesiredEnd do
    active_agent ⇐ agent_white
    if board.side_to_move = black then
      active_agent ⇐ agent_black
    end if
    move ⇐ active_agent.findNextMove
    board.push(move)
  end while
```

2.1 Reinforcement Learning

Reinforcement Learning consists of agents learning to perform a task through interaction with their environment. Typically, Markov decision processes (MDPs) are used as a formal way to model Reinforcement Learning, since the problems typically solved by RL have the Markov property: "the next state depends only on the current state and the actions available in it (no memory of previous states or other information is necessary)" (Plaat, 2020). MDPs are a 5-tuple in the form of (S, A, P, R, γ). In order, these are: the possible states, the possible actions, the probability of a specific action being taken in a given state and at a given time step, the reward obtained after the state transition caused by taking an action, and the discount factor for future rewards. While RL generally involves policy functions and value functions, in this research the relevant one is specifically the action-value function used in Q Learning. Typically noted

$Q(s, a)$, but, more precisely, $Q^{\pi}(s, a)$, it refers to the value of taking action a in state s while under policy π. However, Q Learning is considered to be off-policy learning, since it learns from backups of the best action that can be taken after the current action is taken instead of the actual action that will be taken, so the notation can again be reduced to $Q(s, a)$. The following subsections will introduce both how this value is calculated and the ϵ-greedy approach to the best action after the current one.

2.2 Q Learning

Q Learning was first introduced by Watkins (1989) and is a model-free off-policy RL algorithm. It is model-free because it learns its value function through direct observations without needing an intermediate model of transition probabilities (Plaat, 2020). As already mentioned, it is also considered off-policy since it uses a greedy approach to the future move it uses when updating its value instead of having a specific policy to decide it. At any time step t, the agent is in state s_t and the action it chooses is a_t. Therefore, the action-value function is represented as $Q(s_t, a_t)$. This value is updated every time the agent interacts with the environment, using equation 2.1.

$$Q^{new}(s_t, a_t) = \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} * \overbrace{\underbrace{(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} * \underbrace{\max_a Q(s_{t+1}, a)}_{\text{est. future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}})}^{\text{temporal difference}} \tag{2.1}$$

A learning rate α is used to limit the influence of a single move on the learning process of the agent, and a discount factor γ is used to determine how influential estimations of future rewards are. Considering that in the case of chess endgames the sequence of moves to mate is more important than individual moves, small values were chosen for α (close to 0) and high for γ (close to 1). Finally, the last element is r_t, the reward that the agent receives by making move a_t in state s_t. In this way, Q Learning collects all available information from the moves the agent already made, and evaluates it as if a greedy policy was used, until the evaluation

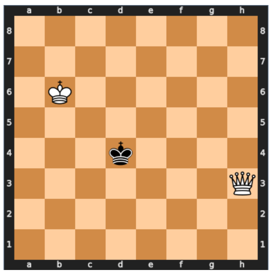


Figure 2.1: The position description part of the FEN of this position is 8/8/1K6/8/3k4/7Q/8/8, and the key of the Q table for the best move is True8/8/1K6/8/3k4/7Q/8/8h3f3, for the white queen moving to f3

stabilizes around the value of the reward obtained over time by choosing any specific action.

The Q Learning agents are split into different versions, to ease data collection and separation, however they share most of their functionality. The differences are the reward functions and the exploration rate, both of which will be discussed in further detail in their own subsection. The agent's learning and decision-making is based on a table of estimates of state-action pairs which updates every time the agent makes a move, as described above. This state-action pair evaluation will henceforth be referred to either as a Q estimate or the Q table, whichever appropriate for the context. An example of the representation of a state and a possible state-action pair can be found in Figure 2.1.

2.3 Least Best Move

An element essential to the agent truly learning the best sequence of moves is the part of the Q Learning equation which learns the best move at the next time step, after the current move is executed. However, in chess, no agent acts multiple times in a row. That has two important implications: firstly, any position reached after a move is made in the current position is impossible as a next state, since the opponent also needs to make a move. Secondly, the agent needs to assume that the opponent will make the best move available, whether or not it actually will, otherwise it cannot be stated that it is actually learning to play correctly. Therefore, the estimate of future value in the Q Learning equation changes like so (2.2). It finds the best move it knows it can do after each of the opponent's moves, and then choosing the minimal Q estimation among them, signifying the opponent playing optimally.

$$\underbrace{\min_b(\overbrace{\max_a(Q(\underbrace{s_{t+2}}_{\text{(a)}}, \underbrace{a_b}_{\text{(b)}}))}^{\text{best move after opp. move}})}_{\text{least best move after opponent's possible moves}}$$

(a) - State after both agent and opponent move

(b) - Agent's moves after opponent moves

(2.2)

2.4 Reward Functions

There are two main reward functions that were explored in order to try to find both the best approach and the minimum complexity needed in order to still achieve a desirable result. While the reward function itself changes between approaches, two elements remain constant. Firstly, all rewards are relative to the previously given reward. This is because the goal is to incentivize the agent to always improve its position. This is necessary since, through the setup of the experiment, the agent is always at an advantage, which it needs to increase until it eventually wins. Secondly, the win reward is divided by the number of moves it took the agent to reach the winning position. While this deviates from the approach of letting each move, and, therefore, series of moves to count for itself regardless of the starting position, it is also a way to prevent the agent from learning a completely inefficient line, either at the start and then repeating it, or after it already has learned a better solution, to be rewarded highly for worsening it.

The first reward function that was attempted is based on the Stockfish evaluation of the position. The version of the Stockfish agent this system was run on returned either a value up to around 6000 for how good the position is for white (or the negative of the value if it is in black's favor) or the number of moves in which a forced mate can be achieved. The advantage value was normalized heavily to range between 0 and 0.3 because not having a forced mate available in the types of endgames that were explored is considered a very bad situation, so the reward needs to reflect it. The number of moves to mate was normalized inversely such that 20 moves to mate would yield a reward of 0, while 0 would yield a value of 2. This value is then squared and a 1 added to it, to represent how even the worst situation of the sort is still better than not having a forced mate. Finally, regardless of the initial processing of the evaluation, the reward is then multiplied by 10 and squared, such that the differences of evaluation differ on an exponential scale. The mathematical form can be found in equation 2.3. Therefore, even after subtracting the previous reward, going from a mate in 3 to a mate in 2 will still be considered better than going from mate in 8 to mate in 7, since the agent is closer to the goal of delivering check mate.

$$\text{rew.} = \begin{cases} \text{rew., if stalemate, repetition} \\ \dfrac{\text{WIN}}{\text{nrMoves}}, \text{ if white checkmate} \\ (\dfrac{\text{eval}}{20000} * 10)^2 * \text{sign}, \text{ if not mate in x} \\ (1 + (\dfrac{20 - |\text{eval}|}{10})^2 * 10)^2 * \text{sign} \end{cases}$$

rew. - Reward

LOSE - Lose reward, set to -10000

WIN - Win reward, set to 10000

nrMoves - Number of moves required to win

eval - Value of the Stockfish evaluation

sign - Positive for white, negative for black

(2.3)

The second reward function is based on a heuristic similar to how people are usually taught to solve these kind of endgames in real life. It is comprised of two elements: how restricted the area around the enemy king is and how far apart the kings are. By minimizing both elements of the position, the agent eventually reaches a state in which one more move is enough to deliver check mate. The available area for the enemy king is calculated using a simple maze filling algorithm in which all of the neighbors of the king are visited, and the neighbors of the visited squares and so on, as long as they are not attacked or occupied by enemy pieces. This is necessary, instead of only counting the neighbors and/or positioning of the king, because there are certain positions in which the agent needs to let the opposing king move back and forth, while bringing its own king forward. In those cases, the value of the enemy king restriction remains the same, contributing the same amount to the reward, while still considering moves that bring the king closer as better. This is simpler and more efficient than the alternative of needing to balance which of these two factors is more important, something that changes in terms of the current position regardless. The amount of available space around the enemy king and the distance between kings is subtracted from an exaggerated worst case scenario of 63 area and 7 distance. Then, that value is squared, for the same reason as for the Stockfish reward function, and then divided by 10 in order to limit the possible reward to under 500. The mathematical notation can be found in equation 2.4.

$$\text{rew.} = \begin{cases} \text{LOSE}, \text{ if stalemate, repetition} \\ \dfrac{\text{WIN}}{\text{nrMoves}}, \text{ if white checkmate} \\ \dfrac{((7+63)-(\text{distance}+\text{section}))^2}{10} \end{cases}$$
$$\begin{aligned} \text{rew.} &\text{ - Reward} \\ \text{LOSE} &\text{ - Lose reward, set to -10000} \\ \text{WIN} &\text{ - Win reward, set to 10000} \\ \text{distance} &\text{ - King moves distance between kings} \\ \text{section} &\text{ - Squares in opposing king's section} \end{aligned} \tag{2.4}$$

In both cases, winning yielded a reward of 10000 divided by the amount of full moves (both agents have moved) to get the results, while drawing, whether by stalemate or repetition, yielded a reward of -10000.

2.5 Exploration

In order to allow the agent to have a chance to improve on suboptimal solutions that are still considered good, it also has a chance to explore instead of always just choosing the least best move. This turns the approach from a greedy one to ϵ-greedy, where ϵ is the chance for the agent to explore. This is more relevant with the heuristic reward function than the Stockfish one, since, in theory, one cannot improve a mate in 4 position to a mate in 2 position without the opponent making a mistake. Even so, the winning reward and, in general, the least best move learning factor could, even in the case of the Stockfish reward function, be improved by exploration, at least in a small number of situations.

Two exploration strategies were explored. The first, and simplest, is a static exploration rate (ϵ) every time the agent chooses a move to do. However, the cost of this simplicity is that the expected win percentage over time plateau is absolute, and based on the amount of moves needed to win. The more moves needed to win, the more times the exploration can occur per match, leading to a statistical limit to the win percentage of the agent, even if it were ran for an infinite amount of time with the perfect solution.

Knowing the unfavorable expectation of the static exploration rate, the second strategy that was studied is that of a decaying exploration rate. The intent is to start from a static exploration rate, which eventually decays after the agent starts winning. While this, on its own, would seem to reduce the agent's ability to improve on suboptimal solutions after a certain amount of wins, in practice this is not much of a concern. The reason for it is that, even with a small learning rate, if the Q estimation keeps increasing every time, it will eventually reach a point when there is no possible reward that would be enough to make the new move have a higher estimation, therefore having it be chosen over the previous move. This would require the agent to explore in the exact same state and randomly choose the exact same move many times over, which, unless running for an infinite amount of time, is simply not practical. With that said, the decay rate should still be relatively low in the beginning, when exploration can still make a reasonable difference, and only truly approach and reach 0 after many wins. The solution proposed for this is dividing the static exploration rate by an exponential function based on the number of wins the agent has accrued during the run. Two such functions were studied, one being the powers of 1.1 and the other the powers of 1.01, as it was considered enough to see if this effect actually helped and is worthwhile to explore further. In both cases, the exploration is not expected to change much within the first dozen wins, but to eventually approach 0, allowing the win percentage over time to approach 100%.

2.6 Data Gathering And Processing

In order to expand on the number of positions tested, 10 starting positions were generated for each of the 3 endgames: King and Queen, King and Rook and King and Bishops (also referred to as King and Bishops for simplicity). The only restrictions on the generation of these positions was that pieces cannot be neighbors in any direction and that the position must be valid. Two main metrics were analyzed: win percentage over time and the number of moves necessary in order to win, relative to Stockfish. The first metric measures how quickly the agent learns and how well it manages to retain that knowledge, while the second measures the actual quality of the solution found. However, the system itself collects a lot more data, both for validation and demonstration purposes. For each experimental setup, meaning agent version, hyperparameters and endgame combination that was studied, the agent was trained over batches of 10 runs of 5000 matches each. The exception is the initial hyperparameter comparison which was over 10000 matches. This, as will be shown later, proved to be needlessly long, so it was reduced to 5000. The results of the 10 runs are compiled together, and, in the cases where more setups were combined, the same process was applied between the compilations of the setups. A compilation in this context refers to averaging results such as the win percentage, and concatenating results such as the moves to win. All of the experiments were run with Stockfish as the opponent, as this provided an informed opponent against which to properly be able to tell if the Q Learning agent was actually learning to play optimally.

For each run, three files are generated: a .csv of the results, a .json of the Q table of the agent, and a .txt file of the logs generated by the agent. The results csv contains, for each match: the total games won, the percentage of games won, the moves to win (or -1 if the agent did not) and the total moves of the match regardless of winning. Lines 1 and 4 were only used for validation of lines 2 and 3. The json of the agent simply contains the json parsing of the dictionary representing the Q table, which can be used with the GUI in order to visually validate the solution learned by the agent and for demonstration purposes of the trained agent. In theory, it can also be used to compile the acquired knowledge of many agents, but it is not something that was explored in this study. Lastly, the log file contains the hyperparameters, starting FEN, the moves required by Stockfish to win from the position, and then information on each match. The number of moves required by Stockfish to win is generated at the start of the run by making both the white and black agent play the moves that Stockfish recommends and retreiving the fullmove number from the python-chess board. For each match, two lines are generated: one containing the match number, number of moves to end the match and the current win percentage, and the other listing the reason for ending the match. Again, this information was used for validation of the results.

For each batch, an additional log is generated and the results are compiled. The log contains the fen file that was used for the batch, the agents that played each side, the hyperparameters for the Q Learning agent, the number of matches and the number of runs. This is relatively redundant information since all batches and runs contain this information in their respective file name, but, again, this was used for validation, since the pipeline for running the system and compiling the results contained several scripts run in sequence. Since there were multiple points of failure, it was important that all of this information remain consistent. For the results csv, all of the initial lines are averaged over the individual runs, with an additional line inserted after the moves to win which replaces any -1 results with the last known number of moves to win. Moreover, all of the non -1 results of the moves to win line were concatenated in an extra line at the end, which, in turn, is further processed into a final line containing the moves to win minus the number of moves it took Stockfish to win in that position. This means that, in order to generate the graphs in the results section, only lines 2 and 7 and the number of moves it took Stockfish to win were needed.

Finally, one last script took the information outlined previously, further compiled together batches that all fell into one of the categories being compared in the graphs, and plotted, using matplotlib (Hunter, 2007), a line graph of the move to win percentage over the matches played, and a histogram of the moves to win relative to Stockfish, using density bins. This results in some of the results averaging over 10 runs, and some over 100 runs, depending on the categories being compared, which is indicated in the following section for each experiment.

2.7 Validation

In earlier subsections, it was mentioned that the GUI is used to validate results. This is only one of the methods used, and is especially important, since it eliminates possible counting errors caused by errors in scripts. However, turning this process into a manual one takes considerably more time, so this method was only used for some of the less expected results, indicated in the next section. The Q table of the agent after training for the 5000 matches of the run it was part of is loaded into a skeleton of the Q Learning agent, without the exploration or the value updating. Then, the agent is observed playing against Stockfish to essentially retrace the exact steps it learned are the best and see how well it actually learned.

For the rest of the results, a script was set up to simulate the above process, but only to count again how many moves it would take Stockfish to win the position and how many it would take each version of the agent after each of the runs. This was also used to generate an average of results of only the runs where the agent reached a solution at all, to study whether it influences the win percentage over time. It was also used to generate a table with the percentage of wins throughout the runs of each combination of endgame and reward function.

3 Results

As described in the previous section, the primary results are presented in terms of win percentage over time and the number of moves required to win, relative to how many it would take the Stockfish agent to win. Moreover, some of the results also split the win percentage over time into two separate plots, one averaging all of the results, the other only averaging the results of the runs in which the agent was able to win. Tables are also provided for both an overall success rate and two edge cases which contain interesting results.

3.1 Parameter Sets

Before all of the other results could be gathered, the most important comparison to make was between different sets of hyperparameters, in order to figure out which set would yield the best results (see Figure 3.1). These hyperparameters are: learning rate (α), discount factor (γ) and exploration rate (ϵ). These hyperparameters are noted in the figures below as ALPHA, GAMMA and R respectively. As mentioned in the previous section, it was expected that a low learning rate, a high discount factor and a low exploration rate would be optimal, and that is also what we see in the graphs. While both sets [0.05, 0.9, 0.003] and [0.05, 0.95, 0.003] have almost identical win percentages over time, the latter has a better distribution of number of wins relative to Stockfish. In other words, the agent learns just as fast under both hyperparameter sets, but the solutions it reaches are much better. Therefore, for all following results, the [0.05, 0.95, 0.003] was used. Moreover, the results were observed to stabilize by the 5000 matches mark, so, both for this reason and to reduce training time to be able to gather more results within the scope of this research, the amount of matches was reduced to 5000 per run.

It should be noted that these results were gathered using the Stockfish reward function over only one starting position. This means that there is a chance these are not the optimal hyperparameters for every situation further investigated, however it was decided, for simplicity, to be used as a ceiling of performance.

3.2 Exploration Decay Strategies

The next important detail to investigate was which decaying strategy would work best to reduce the exploration rate of the agent as it starts winning. For this purpose, three versions of the agent were investigated, all based on the Stockfish reward function: the normal, static version, one where the exploration rate is divided by the powers of 1.1 and another by the powers of 1.01. Moreover, the results were split by the type of endgame studied: King and Queen, King and Rook, and King and Bishops (Figure 3.3). The expected results were that the first endgame would not improve by much, while the other two would see great improvements. Overall, as mentioned in the previous section, the win

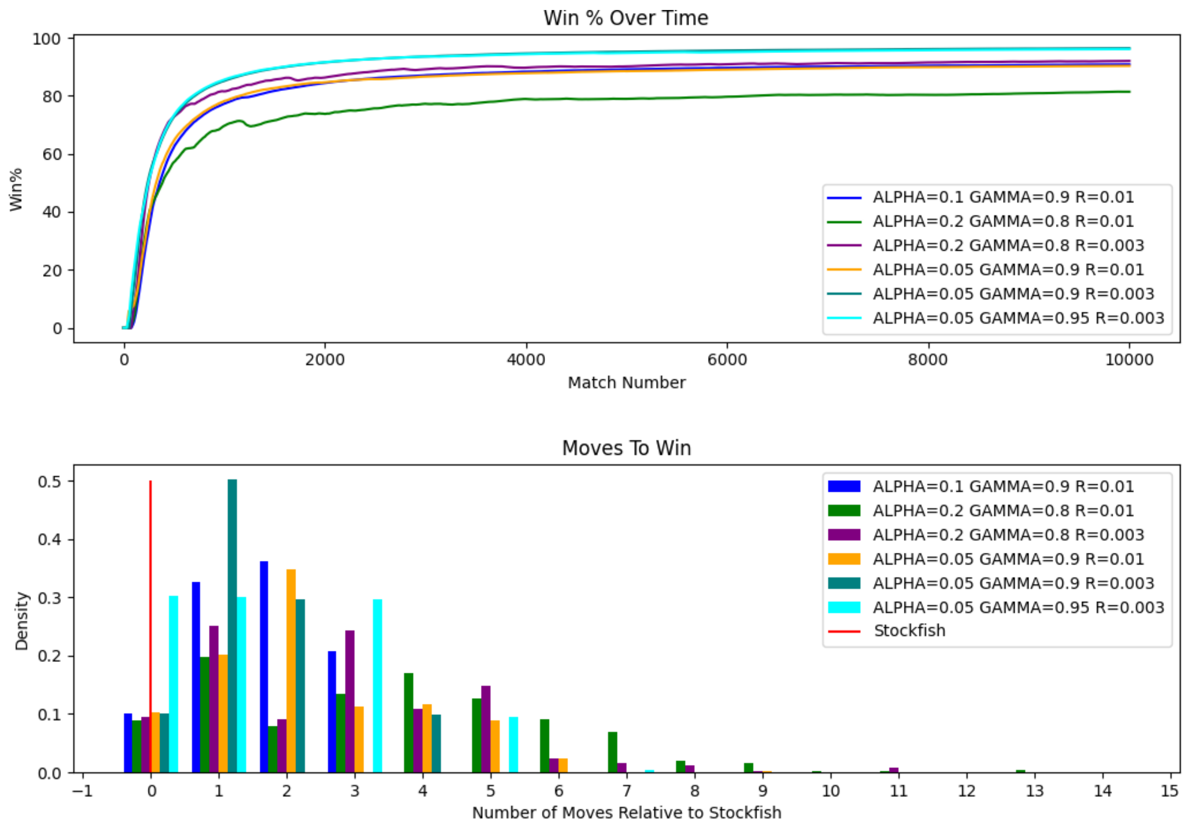


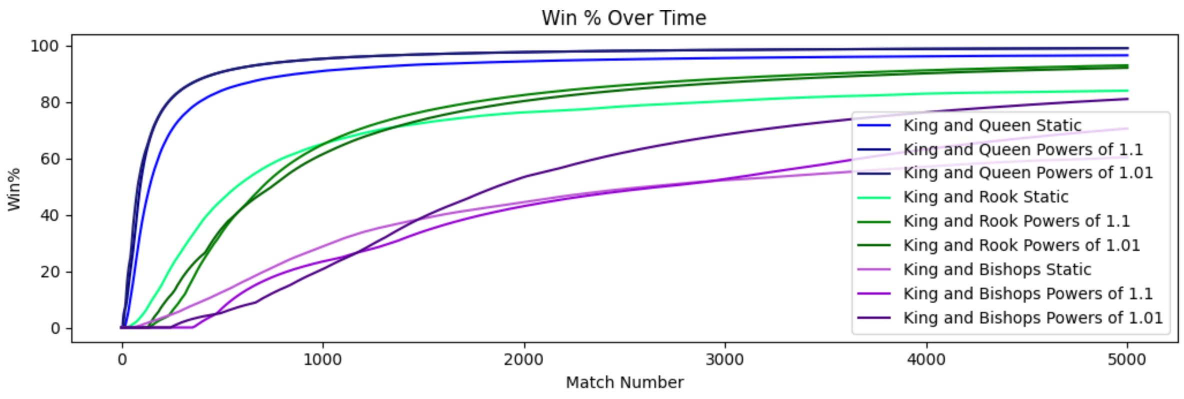
Figure 3.1: Comparison of results of the Q Learning agent with the Stockfish reward function, static exploration rate, 10 runs over 1 King and Queen position

percentage for all cases where exploration decay is implemented was expected to eventually approach 100%. It was also expected that the moves to win relative to Stockfish might worsen due to less exploration possibly leading to less improvement of found solutions.

These predictions were not entirely accurate (see Figure 3.2). Firstly, the win percentage improved for all three endgames, not just two of them. However, this change only makes the case of King and Queen reach a solution faster, while for King and Rook and King and Bishops the improvement comes in the elimination of the early plateau present without decay. The former case almost reaches the performance of King and Queen within 5000 matches. Secondly, the moves to win results actually improved instead of worsening when adding exploration decay. While the difference is minor, it was decided that, while it leads to a worse win percentage specifically for the King and Bishops scenario, the distribution of the moves to win results were overall the best for the powers of 1.1 version. Therefore, for the following experiments, the powers of 1.1 exploration decay strategy was used. It should be noted that, due to the clutter of having these many results to compare, the win percentage plot was not split up to also show only solved runs and the move to win results were cut off below 0 since the bars are slim enough as they are.

3.3 Endgame Performance Comparison

With all of the previous parameters established, the first comparison performed was between the performance of the three studied endgames. It was expected that the performance would decrease between King and Queen, King and Rook, and King and Bishops, in this order, due to the increasing



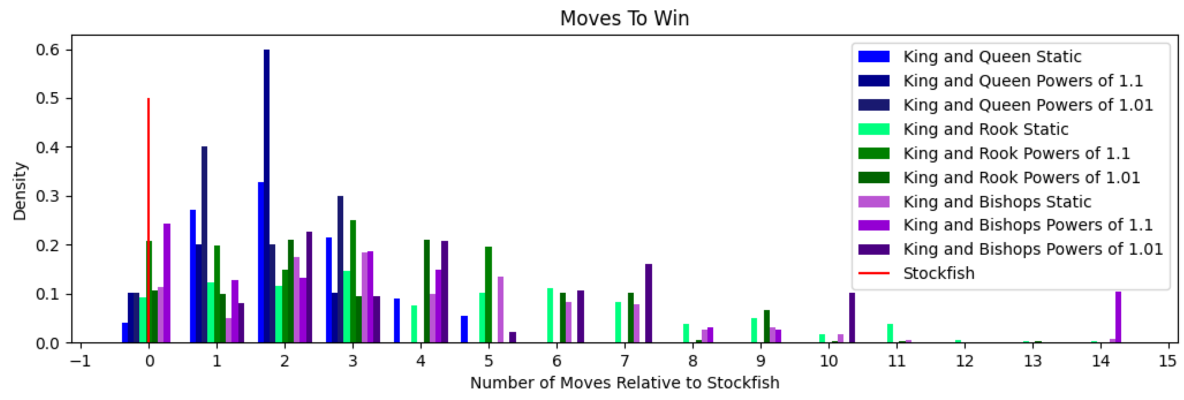


Figure 3.2: Comparison of results based on the strategy for decaying the exploration rate, split over the combinations of endgame and strategy when applied to the Stockfish reward function. Note: the Moves To Win histogram was reduced to the 0-15 range to increase clarity, due to the low percentage of <0 results and the high number of bars to display

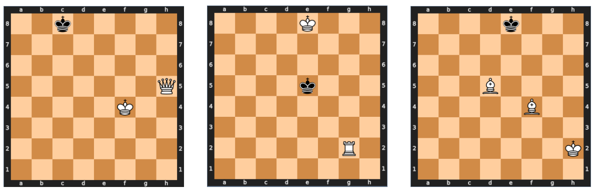


Figure 3.3: Examples in order: King and Queen, King and Rook, King and Bishops

number of moves required to win in each of these positions. Even so, it was expected that the the win percentages of the latter two endgames would increase when only taking into account the runs which reached a solution, and that the moves to win would be mostly distributed around plus 0-5 relative to Stockfish.

Once again, the results prove somewhat surprising (see Figure 3.4). Firstly, there is very little noticeable difference between the two win percentage graphs, meaning that the relatively low results of the latter two endgames is caused by the agent being slow to learn how to win in those positions, and not it being dragged down by runs that fail completely. Even in the case of King and Bishops, where there is an increase in the solved only graph, it is still worse off than King and Rook, let alone King and Queen. Once again, mathematically, the win percentage of all scenarios will eventually approach 100% if they find any solution at all, but that is not visible withing the first 5000 matches for the latter two endgames.

With that said, the most surprising part of the results is that there are runs in which the solution is actually better than that of the Stockfish agent. Even more interestingly, the best endgame

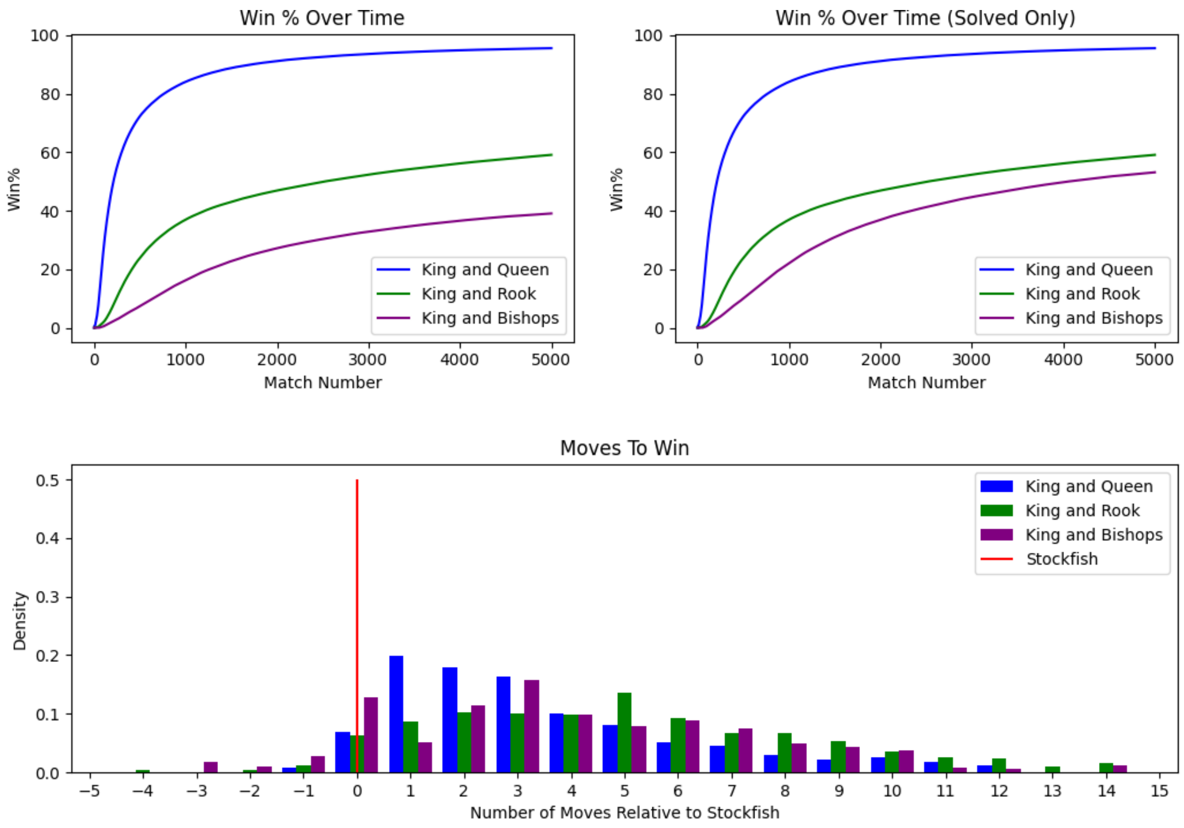


Figure 3.4: Comparison of results based on the endgame studied, aggregating over the 4 versions of the reward functions with 10 runs for each of the 10 positions in each endgame

in terms of achieving better results than Stockfish is actually King and Bishops, which usually scores the worst. This had led to investigation and validation of the results and the agent's movecounts each run, since Stockfish was expected to be the most optimal solution possible. Two King and Bishops positions were removed from the results due to having such extreme results that they were incomparable with the other runs of any endgame (see Subsection 3.5). Even so, the other better than Stockfish results could still be validated both manually and through a script.

3.4 Reward Function Comparison

Finally, the last comparison performed was between the two different reward functions, further split up between static and decaying exploration rates. It was expected that the Stockfish reward function would perform better, but that the heuristic should yield similar results, especially when only taking into account the runs that reached a solution. The decaying exploration versions of the reward functions was also expected to perform better, and avoid reaching an early plateau within the 5000 matches.

This time, the results were mostly as expected (see Figure 3.5). However, the heuristic performed, on average, half as well as the Stockfish reward function, even when only taking into account runs that reached a solution. This is most likely since the heuristic versions did not have time within 5000 matches to account for the loss in win % incurred by reaching a solution slower than the Stockfish reward versions. This is also backed up by the difference in performance between the static and decaying versions of the reward function is being much more visible for the Stockfish reward versions, the static starting to reach its plateau, while the static version of the heuristic reward does not.

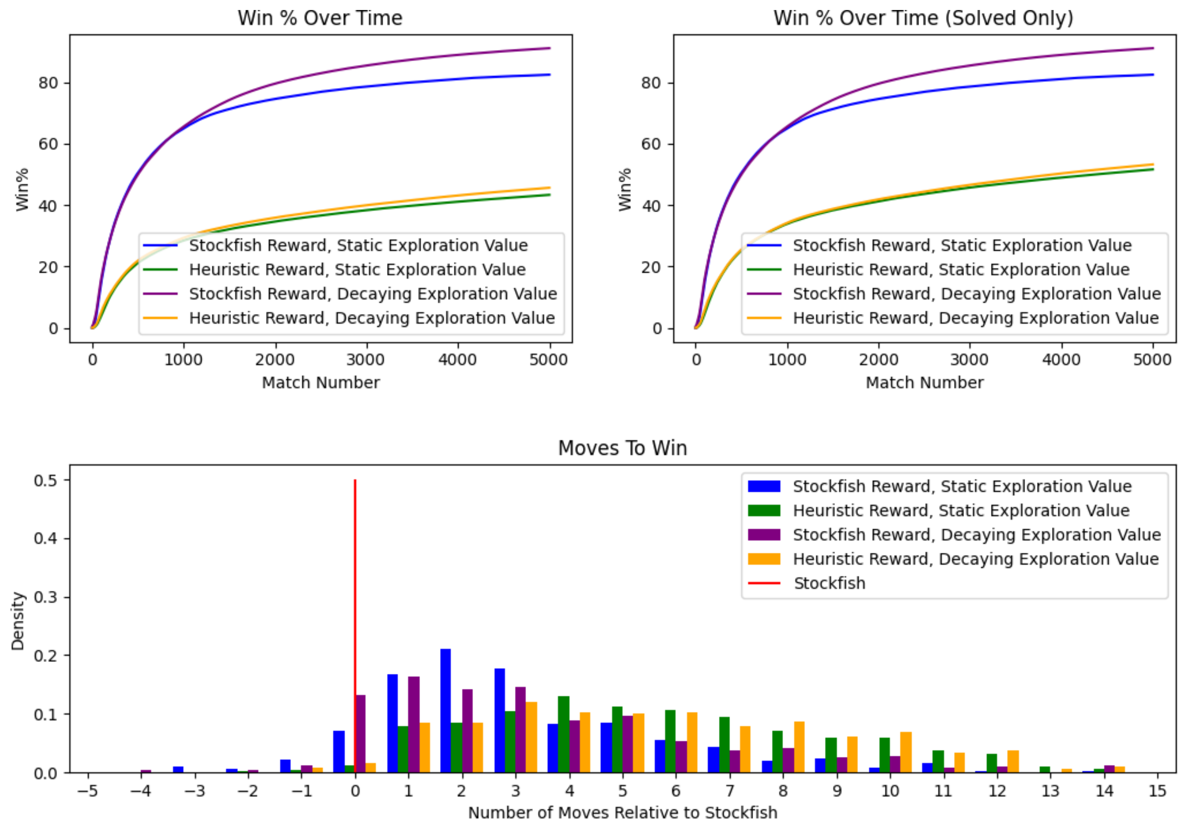


Figure 3.5: Comparison of results based on the reward function, further split up by static and decaying exploration rates, aggregated over all endgames.

Table 3.1: Special cases A and B, 2 of the generated King and Bishops positions, runs 1-10.

	Stockfish	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Stockfish Static A	-	21	-	26	24	19	20	-	22	20	27
Stockfish Decaying A	-	25	27	27	23	15	23	23	21	20	22
Heuristic Static A	-	-	-	-	-	-	-	-	-	-	-
Heuristic Decaying A	-	-	-	-	-	-	-	-	-	-	-
Stockfish Static B	49	22	20	22	19	-	18	20	27	19	21
Stockfish Decaying B	49	20	22	21	24	13	19	16	23	24	20
Heuristic Static B	49	-	-	-	-	-	-	-	-	-	-
Heuristic Decaying B	49	-	-	-	-	-	18	-	-	28	-

Table 3.2: Percentage of runs reaching a solution.

	King and Queen	**King and Rook**	**King and Bishops**
Stockfish Static	100.0	93.0	82.0
Stockfish Decaying	100.0	100.0	100.0
Heuristic Static	99.0	47.5	5.0
Heuristic Decaying	100.0	58.0	7.0

3.5 Special Cases And Overall Performance

During validation, two positions of the King and Bishops endgame stood out as having results so unexpected that they are incomparable. In Table 3.1 cases A and B are presented, with the number of moves required for Stockfish to solve the position, and the number of moves each of the runs of the agent took to solve the position during validation. Case A was interesting because Stockfish was not able to solve the position at all (stuck in 3 move repetition), while the Stockfish reward function versions of the agent were able to solve it. There does not seem to be a significant difference between the performance of the static and decaying exploration rate versions, but the latter seems slightly better. Case B was interesting because Stockfish required a large amount of moves to win, while both Stockfish reward function versions almost consistently solved the position in half the number of moves. In both cases, the heuristic reward functions versions did not manage to solve the positions at all, except two of the runs of the decaying version for case B. It is unclear from the results if this is a problem of not having had enough matches in order to solve it, or if it would also have strange results like the Stockfish agent when ran for longer.

Finally, after the amount of unexpected effects in the graphical results, it was important to also check numerically how many of the agents in each of the explored scenarios actually managed to reach a solution at all. As is shown in Table 3.2, the same trends appear as with the previous results: the decaying versions of the reward functions perform better than the static ones, and the performance drops for the three endgames in the order: King and Queen, King and Rook, King and Bishops. The only version that managed to solve every position studied is the Stockfish reward function with a decaying exploration rate, while the worst performing were both versions of the heuristic reward function for the King and Bishops endgame.

4 Conclusions

Overall, the agent performed better than expected, with some exceptions. In the majority of cases, it managed to find a winning solution to the endgame position it was presented with, within a reasonable amount of inefficiency when compared to Stockfish. The biggest surprise was the Q Learning agent performing better than the Stockfish agent in some scenarios, especially in the two special cases. After validation with outside sources (chess.com), it became apparent that the most likely explanation for this is the default settings that the Stockfish engine was run with. As most modern chess engines, Stockfish allows for the user to set the rating of the engine (how good it is), the depth (how many moves ahead it will evaluate), a timeout, etc. By changing the Stockfish settings to the highest level of the engine (25, default 15) and a lower depth (25, default 15), special cases A and B get solved in 14, respectively 18 moves. While these results seem much better than the Q Learning results in these cases, it should be noted that, since the opponent played differently, it is impossible to actually compare these results with the ones from training. The Q Learning agent would need to be re-trained against the better version of Stockfish to properly compare. However, this would mean much longer training times, unless an efficient caching mechanism was also implemented, since each call to the Stockfish engine would take much longer. It is unclear whether this would also improve the performance of the Q Learning agent using the Stockfish reward function versions of the agent, but it would have no positive, and even possibly negative, effect on the heuristic reward function versions.

The place where the Q Learning agent performs much more poorly than expected is for the heuristic function reward versions playing the King and Bishops positions. Some performance degradation was of course expected due to the limiting mechanics on the maximum amount of moves considered appropriate for the agent to solve positions in, but reaching 5% and 7% solution rates is much lower than expected. The obvious fix, then, would be to modify the reward function such that it allows for more moves to be made before just improving the position is considered better than winning in the currently trained sequence of moves. However, this would need to be done very carefully since removing the mechanism completely would most likely lead to an overall increase in the number of moves necessary to win compared to Stockfish. A better solution would be to come up with a better heuristic, such that the agent learns to solve the position in fewer moves and with a better strategy. This is, of course, a much more difficult option.

4.1 Limitations and Future Work

Apart from the improvements suggested above, there are other areas which could use improvement. Firstly, as mentioned, some of the results have suffered extreme performance degradation due to the move limiting mechanism of the reward function. In theory, the best balance should be to limit the total number of moves using this mechanism to 50 full moves, since, after that, the game would have to end due to having passed 50 moves without capturing any pawn. However, this is not applicable to all types of endgames, specifically those including pawns. Even then, the hasty solution of resetting the counter every time a pawn is captured should also not be implemented without careful consideration, since the agent might end up developing strange, inefficient patterns by stumbling upon cases where capturing a pawn right before winning gives it a much higher reward. The mechanism itself could be removed entirely, of course, but the likeliest outcome, from studying the system during the development and iteration process, is that, the more the agent is run, the more the number of moves to win increases. In that case, a limitation on the number of matches in which the agent actually trains is recommended as a replacement.

Another limitation is the number of matches per run being adjusted to the best case scenarios, not the ones which would have needed more matches to reach a palteau of performance. However, increasing the number of matches for the 100 runs of every category that was compared would have potentially doubled or tripled the training time, which would have been outside the scope of this research, but would definitely be important to study in future works on the subject. The question whether the agents which have not reached a solution have done so due to problems with the reward function or simply a lack of training time is of particular importance.

Finally, the Q Learning algorithm was designed to only train for white to win, even if the overall system was planned to be able to handle any combination of agents, even those training against one another. Moreover, the heuristic reward function only works for clearly winning endgames requiring no pawn promotions, and the results were underwhelming, so a better one entirely would help with the training of the agent. It would be interesting to expand the algorithm such that white can also learn to play for a draw if it is at a disadvantage, and for black to be able to train as well. These cases were out of the scope of this research, as it is an implementation difference rather than a conceptual one to study and report on, but they would allow further research on the performance of Q Learning agents training against one another. The most interesting goal would be to see if an adversarial mechanism can lead to agents performing close to those with the Stockfish reward function. Another obvious avenue of research is, of course, also expanding to other model-free Reinforcement Learning algorithms, but they would each probably require the same amount of effort to explore the peak performance as for Q Learning, such that the comparison between the algorithms is fair.

References

Boar, A. I. (2022). Q learning chess endgames repository. `https://github.com/ToniSnakes/QLearningChessEndgames`. Accessed: 2022-24-07.

Busoniu, L. (2010). *Reinforcement learning and dynamic programming using function approximators.* CRC Press.

"Chess" (2022). Chess. `https://en.wikipedia.org/wiki/Chess`. Accessed: 2022-24-07.

"Chess Engine" (2021). Chess engine. `https://www.chess.com/terms/chess-engine`. Accessed: 2022-24-07.

Dvoretsky, M. I. (2020). *Dvoretsky's Endgame Manual.* Russell Enterprises, Inc., 5th edition.

"Engines" (2022). Engines. `https://www.chessprogramming.org/Engines`. Accessed: 2022-24-07.

Fiekas, N. (2022). *python-chess*, v1.9.2 edition.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95.

Müller, K. and Lamprecht, F. (2001). *Fundamental Chess Endings.* Gambit Publications.

Plaat, A. (2020). *Learning to play : reinforcement learning and games.* Springer.

Romstad, T. (2011). Stockfish-open source chess engine. *https://stockfishchess.org/.*

Sutton, S. S. and Barto, A. G. (2018). *Reinforcement learning : an introduction.* The MIT Press, second edition.

"Tournament Results" (2022). Top chess engine championship tournament results. `https://en.wikipedia.org/wiki/Top_Chess_Engine_Championship#Tournament_results`. Accessed: 2022-24-07.

Watkins, C. J. C. H. (1989). Learning from delayed rewards.