TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

BACHELOR

Chess Endgame Analysis
Reinforcement Learning vs. Search Tree Algorithms

Ypma, Jord J.

Award date:
2024

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 21. Mar. 2025

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Bachelor End Project
Postbus 513, 5600 MB Eindhoven
The Netherlands
www.tue.nl

Author
Jord Ypma (1630342)

Responsible Lecturer
Maxence Delorme

Date
June 13, 2024

Chess Endgame Analysis: Reinforcement Learning vs. Search Tree Algorithms

A Comparative Study of Reinforcement Learning and Search Tree Methods in Chess Endgame Solving

Jord Ypma (1630342)
j.j.ypma@student.tue.nl

Where innovation starts

Technische Universiteit
Eindhoven
University of Technology

Table of contents

Title
Chess Endgame Analysis: Reinforcement Learning vs. Search Tree Algorithms

1 Introduction 1

2 Literature overview 2
2.0.1 AlphaGo . . . 2
2.0.2 Starcraft . . . 2
2.0.3 Poker . . . 2
2.1 Tree search algorithms . . . 3
2.1.1 Monte Carlo Tree Search . . . 3
2.1.2 The minimax algorithm . . . 3
2.2 Q-learning . . . 3
2.2.1 Nash Q-learning . . . 3
2.3 Deep Q-learning . . . 4
2.4 Deep Reinforcement learning in chess . . . 4

3 Methodology 5
3.1 MiniMax algorithm . . . 5
3.2 Monte Carlo Tree Search . . . 7
3.3 Q-learning for zero sum games . . . 8
3.4 Deep Q-learning for zero sum games . . . 9
3.5 Stockfish . . . 10

4 The game of chess and chess endgames 11
4.1 Game rules . . . 11
4.2 Stalemate . . . 14
4.3 Insufficient material . . . 14
4.4 50-move rule . . . 14
4.5 Three fold repetition . . . 15

5 Implementation 16
5.1 Simulating endgames . . . 16
5.2 Monte Carlo Tree Search methods . . . 17

Where innovation starts

Technische Universiteit **Eindhoven** University of Technology

Table of contents

Title
Chess Endgame Analysis: Reinforcement Learning vs. Search Tree Algorithms

5.3 MiniMax methods 17
5.4 Chess heuristics 17
5.5 Reward and evaluation functions 18
5.6 Q-learning methods 19
5.6.1 Q-table representation 19
5.6.2 Exploration strategy 19
5.6.3 Q-learning algorithm for a chess endgame 19
5.7 Deep Q-learning methods 22
5.7.1 State representation 22
5.7.2 Action representation 22
5.7.3 Neural network structure 23
5.7.4 DQN algorithm for chess endgame 24

6 Experiments and Findings 25
6.1 Tree algorithms for Different Endgames 25
6.1.1 Time complexity 25
6.1.2 MCTS algorithm for different endgames 25
6.1.3 Minimax algorithm for different endgames 26
6.2 Q-learning on Different Board Sizes 28
6.2.1 Checkmating probabilities 28
6.2.2 Training on different board sizes 28
6.3 Q-learning Agent on full board 30
6.3.1 Training against different opponents 30
6.4 DQN results 32

7 Conclusion and future work 34

1 Introduction

Games offer an entertaining opportunity for socializing and keeping your mind engaged. From traditional board games to modern video games, there has been a constant presence in human culture. Beyond entertainment, games facilitate strategic thinking, problem-solving and social bonding. Moreover, games promote cognitive skills such as planning and adaptability, which are also beneficial in various real-life situations.

For every game that exists, people have always looked for good strategies and tactics to win. Strategies are long-term plans for success, while tactics are short-term ideas to gain an advantage. Games vary greatly in complexity, often linked to the number of possible game states. This number increases rapidly as games become more complex; for example, tic-tac-toe already has 5,478 [1] possible game states.

Chess, an ancient board game originating in India, is well known for its profound strategies, decision-making and intellectual challenging character. It is played as a 2 player turn based game on an 8x8 board with 64 squares with 6 different pieces that have different moving restrictions. Each side has 16 pieces in total. The objective of the game is to checkmate the opponent's king, meaning the king is in a position to be captured and has no more legal moves to escape.

Chess has become globally popular as an intellectual sport. It is played at both amateur and professional levels in tournaments, competitions and online platforms. The game is known for containing strategic thinking, forward planning and pattern recognition. Recent neuroimaging studies have showed that expert chess players have unique brain differences related to playing chess. These findings suggest that the brains of expert chess players are structurally adapted to the demands of the game, highlighting the neural basis of chess expertise. [2]

For computers and artificial intelligence (AI), chess presents a challenging domain due to its immense complexity and vast search space of possible moves and possible counter-moves. Despite these challenges, AI has made significant progress in developing chess programs that can match or even beat the strongest human players. These chess engines use advanced search algorithms and machine learning methods to find strong strategic and tactical ideas.

Most chess games have 3 phases. There is the opening, the middle game and the endgame. In chess endgames, the focus shifts towards achieving specific objectives. The last objective is to checkmate the opponent's king. Unlike the opening and middle game phases, where the board is full with pieces and the strategy involves controlling the center and developing pieces, endgames often have fewer pieces. This allows for clearer strategic goals, such as checkmating the opponent's king. This requires players to use the remaining pieces to efficiently corner the opponents king and restrict its movement.

Two distinct methodologies for chess endgame solving will be explored in this research: reinforcement learning (RL) techniques and traditional search tree algorithms. RL offers flexibility and can learn optimal strategies through interactions with the environment guided by reward structures. This makes it well-suited for dynamic and complex scenarios like chess endgames. In contrast, traditional search tree algorithms rely on predefined heuristics and exhaustive search strategies to evaluate moves. While they could be highly effective, these methods can be computationally expensive and may struggle with the vast search space and the strategic nuances of chess endgames.

This research aims to address the following question: how does the performance of RL techniques compare with traditional search tree algorithms methods in training agents for chess endgames, particularly in terms of decision-making efficacy and adaptability to different reward structures?

2 Literature overview

This literature review explores the methodologies and advancements relevant to the investigation of reinforcement learning (RL) techniques and traditional search tree algorithms and their application in games. Understanding these approaches gives a basis for comparing their performance in solving chess endgames.

Artificial intelligence (AI) has greatly changed the world of games. It has introduced new ways to play and understand games. AI is used in simple board games and complex real-time strategy games to improve gameplay, create smarter opponents and generate new game content. The connection between AI and games has expanded game design and helped advance AI research. Games provide a controlled environment where researchers can test and improve AI algorithms with clear goals and feedback. This section explores different AI algorithms and how they are used in various games. It highlights important developments and contributions in this area.

2.0.1 AlphaGo

AlphaGo, developed by Google DeepMind, made headlines by defeating world champion Go player Lee Sedol in 2016. Go is a complex board game with a vast number of possible positions. The number of possible game states in this game is so great that traditional search tree methods become impractical. AlphaGo combined Monte Carlo Tree Search (MCTS) with deep neural networks to evaluate board positions and select moves. This hybrid approach let AlphaGo handle the immense complexity of Go and make strategic decisions at a superhuman level[3]. This is relevant to this study as it demonstrates the power of using modern tree search methods and deep learning for a game with a lot of game states.

2.0.2 Starcraft

StarCraft II is a real-time strategy game that involves resource management, strategy planning and real-time decision-making. DeepMind developed AlphaStar, which is an AI agent that uses deep reinforcement learning and neural networks to play StarCraft II. AlphaStar was trained using a combination of supervised learning from human games and reinforcement learning through self-play. This allowed AlphaStar to develop advanced strategies and achieve a high level of play in this complex, multi-agent environment [4]. This example highlights the effectiveness of reinforcement learning in a complex environment that has more than one player.

2.0.3 Poker

Poker is a game of incomplete information, where players must make decisions based on probabilistic reasoning, but where players also use bluffing. Libratus, an AI developed by Carnegie Mellon University, showed superhuman performance in the Texas hold'em variant of poker. Libratus used a combination of game theory, reinforcement learning and a decomposition approach to handle the game's complexity. It played multiple strategies simultaneously and adjusted them based on the opponents' actions. This led to its success against professional poker players [5]. This is also relevant for this study as it shows how RL can be adapted to environments with incomplete information, which can be useful in complex chess endgames where the exact outcomes of moves may also not always be clear.

2.1 Tree search algorithms

2.1.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a technique widely used in artificial intelligence for making optimal decisions in various areas. It is especially used for games. This search tree method involves exploring decision options and building a search tree based on sampling methods. It has shown significant progress since its introduction. Researchers have been able to adapt and improve the basic MCTS algorithm for different tasks, leading to its widespread application in different domains. [6]

2.1.2 The minimax algorithm

The minimax algorithm is an algorithm created by John von Neumann in 1928. He researched strategies for zero-sum games. This means that one player's gain is equal to the opponent's loss. Examples of such games are chess and tic-tac-toe. The algorithm works using a decision tree that contains all possible game states. It evaluates these states to find the best move in each state. This ensures a good evaluation of a situation in the game.

Von Neumann wrote a paper about this called "Zur Theorie der Gesellschaftsspiele" in the scientific journal "Mathematische Annalen" [7]. Von Neumann's research concluded that the minimax algorithm provides a mathematically optimal strategy, allowing a player to minimize their maximum possible loss and ensure the best possible outcome against a perfectly rational playing opponent. This means that the Minimax algorithm provides a fundamental approach to decision-making in two player games like chess.

2.2 Q-learning

Q-learning, introduced by Christopher Watkins and Peter Dayan in 1992, is a fundamental reinforcement learning algorithm that can learn an agent to make decisions in an environment through trial and error. The agent learns by interacting with the environment, receiving feedback in the form of rewards or penalties based on its actions. Q-learning maintains a table of values, known as Q-values, which represent the expected cumulative rewards for taking specific actions in specific states. By updating these values using a learning rate and the observed rewards, the agent gradually learns the optimal policy for selecting actions in different states to maximize its long-term reward. This process allows the agent to explore different strategies and learn from its experiences, ultimately converging to an optimal policy for the given environment.[8]

2.2.1 Nash Q-learning

Nash Q-learning is an extension of the Q-learning algorithm to the context of noncooperative multiagent systems. It is particularly used in the field of general-sum stochastic games. In Nash Q-learning, each learning agent maintains Q-functions over joint actions. These represent the expected sum of discounted rewards when all agents follow a specified Nash equilibrium strategy. Unlike traditional single-agent Q-learning, where Q-values are purely based on the agent's own optimal strategy, Nash Q-values take into account the joint strategies of all agents in the system. The goal of Nash Q-learning is to learn these equilibrium Q-values through repeated play. It allows agents to make strategic decisions that lead to stable Nash equilibria. This approach enables agents to adapt and compete in dynamic and competitive environments where the actions of multiple agents influence each other's outcomes.[9]. For the chess endgames in this research one side will have to learn to checkmate while the other side has to give best

resistance. Ideally this would result in the best line that leads to checkmate for both sides. Therefore, the principle of Nash Q-learning will be used in this study.

2.3 Deep Q-learning

In Q-learning an agent learns to make decisions by trying out different actions and analyzing the results. However, Q-learning doesn't work well when there are too many possible states in a game because it requires storing a Q-value for each state-action pair, which becomes impractical if the environment is too complex.

Deep Q-learning offers a solution to this problem. Instead of saving all the Q-values, it uses a neural network to approximate them. This allows the algorithm to handle much larger state spaces. The neural network takes the current state as input and outputs Q-values for all possible actions. This makes it possible to decide the best action without needing to store Q-values for every state.

Deep Q-learning was developed by the Google DeepMind team in 2013. They demonstrated its effectiveness by applying it to play Atari. In these games, the algorithm evaluates game states using only the pixels on the screen. This means the neural network processes the image data from the game and learns to predict which actions will lead to the highest scores. The approach allows the algorithm to learn and generalize from the game experiences. This allows it to play the games at a level comparable to or even better than human players.[10]

2.4 Deep Reinforcement learning in chess

Chess is a good test for seeing how well artificial intelligence algorithms can perform. Over the years, AI researchers have explored a lot of different approaches to improve computer chess engines. The traditional way these engines are constructed are with search tree algorithms that have complex evaluation functions. Stockfish is currently the most famous and strongest traditional engine. More recent advancements also include reinforcement learning (RL). AlphaZero, created by the Google Deepmind team, was the first algorithm trained with RL to challenge stockfish and the other traditional chess engines. AlphaZero adjusts its parameters using pure self play. It rewards positive outcomes and discourages negative ones. Ultimately, it got very strong at the game without any help from people or previous knowledge of chess.[11].

3 Methodology

In this study, two types of algorithms will be used to try to solve different chess endgames: tree algorithms and reinforcement learning techniques. The tree algorithms are Minimax and Monte Carlo Tree Search, while the reinforcement learning methods are Q-learning and Deep Q-learning. These methods were chosen for their potential to solve chess endgames. The reinforcement learning methods in this study will treat the problem of chess endgames as a Markov Decision Process, where decisions are made based only on the current state. On the other hand, the tree algorithms will choose a move based on possible future scenarios and chess heuristics.

3.1 MiniMax algorithm

The minimax algorithm is widely used in game theory for finding the best move in a two-player game. It works by exploring all possible moves in a game-tree and selecting the option that maximizes the player's advantage while minimizing the opponent's. This allows for a comprehensive evaluation of the game state, because it considers both players' perspectives.

The algorithm starts by creating a tree structure of all possible moves from the current position with a given depth. For each possible move, it calculates a value that represents how good that move is for the player. These values are calculated using an evaluation function. This function is designed to give a good evaluation of a position using heuristics of the game. These values are then passed up the tree, with the algorithm choosing the action that leads to the highest value for the current player, while keeping in mind that the opponent will execute the actions that leads to the lowest evaluation. An example of the algorithm applied to a game state of tic tac toe can be seen in figure 3.1. In this example the circle player is the maximizing player, while the cross player is the minimizing player.

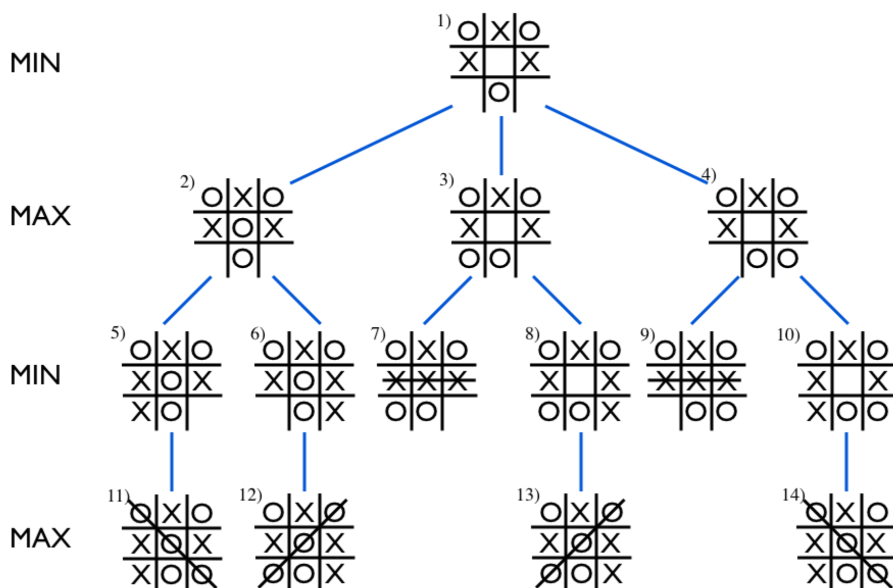


Figure 3.1: Minimax algorithm applied to a tic tac toe game state [12]

However, the algorithm's exhaustive search can be computationally intensive, as it examines every possible move and outcome. To address this, several enhancemants to the algorithm can be made. In this research, alpha-beta pruning will be applied. This technique reduces the search space by discarding

branches of the game tree that are unlikely to lead to a better outcome for the current player taking into account the opponent will perform the best actions. By eliminating these unpromising paths, alpha-beta pruning can improve the efficiency of the minimax algorithm without sacrificing accuracy. The minimax algorithm applied to a 2 player game can be seen in Algorithm 1. [13]

Algorithm 1 Minimax Algorithm with Alpha-Beta Pruning

```
1:  function MINIMAX(node, depth, α, β, maximizingPlayer)
2:
3:      if depth = 0 or node is a terminal node then
4:          return evaluate(node)
5:
6:      if maximizingPlayer then
7:          bestValue ← −∞
8:          for all child in node.children do
9:              value ← MINIMAX(child, depth − 1, alpha, beta, false)
10:             bestValue ← MAX(bestValue, value)
11:             alpha ← MAX(alpha, bestValue)
12:             if β ≤ α then
13:                 break
14:         return bestValue
15:
16:     else
17:         bestValue ← +∞
18:         for all child in node.children do
19:             value ← MINIMAX(child, depth − 1, alpha, beta, true)
20:             bestValue ← MIN(bestValue, value)
21:             beta ← MIN(beta, bestValue)
22:             if β ≤ α then
23:                 break
24:         return bestValue
```

3.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an algorithm that differs from minimax by focusing more on exploring the depth of the search tree rather than its breadth. In MCTS, a selection policy is used to pick nodes for further exploration. Once a node is selected, MCTS conducts a simulation or rollout from that node until a terminal state is reached, often by making random moves. This process helps in estimating the potential value of the node. MCTS prioritizes nodes with higher potential, which are the ones visited more frequently by the algorithm. To create a balance between exploiting known paths and exploring new ones, the Upper Confidence Bound for Trees (UCT) formula is often used as given in 3.1, because it has proven to give a strong exploration-exploitation balance [6]. The MCTS algorithm applied to connect four can be seen in figure 3.2.

$$UCT(s,a) = \frac{Q(s,a)}{N(s,a)} + c \cdot \sqrt{\frac{\ln(N(s))}{N(s,a)}} \tag{3.1}$$

- $UCT(s,a)$: Upper Confidence Bound for Trees (UCT) for state s and action a, indicating confidence in selecting a from s.
- $Q(s,a)$: Estimated value of performing action a in state s, updated during MCTS execution.
- $N(s,a)$: Number of times action a has been performed from state s.
- $N(s)$: Total number of times state s has been visited.
- c: Constant balancing exploration and exploitation.

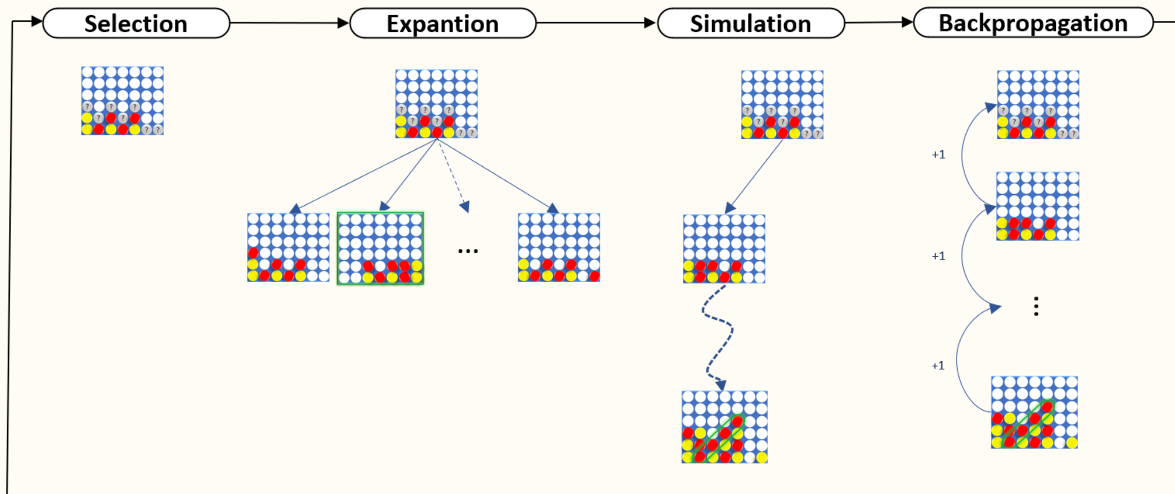


Figure 3.2: Example of MCTS steps for a game of connect four [14]

The pseudocode for the application of MCTS can be seen in Algorithm 2.

Algorithm 2 Monte Carlo Tree Search (MCTS)

```
1:  function (MCTS)root, iterations
2:
3:      for i ← 1 to iterations do
4:          node ← root
5:
6:          while node is not a leaf and node is fully expanded do
7:              node ← selectChild(node)
8:
9:          if node is not a terminal node then
10:             expand(node)
11:             node ← selectChild(node)
12:
13:         reward ← simulate(node)
14:         backpropagate(node, reward)
15:
16:     return argmax_{child of root} visits(child)
```

Algorithm 3 Supporting functions of MCTS

```
1:  function SELECTCHILD(node)
2:      return argmax_{child of node} UCT(child)
3:
4:  function UCT(node, c)
5:      return value(node)/visits(node) + c × sqrt(log visits(parent) / visits(node))
6:
7:  function EXPAND(node)
8:      randomly select untried action from node
9:      add child node for selected action
10:
11: function SIMULATE(node)
12:     randomly simulate from node until terminal state reached
13:     return reward of terminal state
14:
15: function BACKPROPAGATE(node, reward)
16:     while node is not null do
17:         update node visits and value with reward
18:         node ← parent of node
```

Line 5 of Algorithm 3: $\text{return } \frac{\text{value(node)}}{\text{visits(node)}} + c \times \sqrt{\frac{\log \text{visits(parent)}}{\text{visits(node)}}}$

3.3 Q-learning for zero sum games

Q-learning is a reinforcement learning algorithm that is model-free, gradient-free and off-policy. This means that it does not require an explicit model of the environment, such as transition functions or reward functions to learn optimal actions. Instead, Q-learning learns purely from interacting with the environment through trial-and-error. It updates its Q-values iteratively based on received rewards and the maximum expected future rewards. Additionally, Q-learning does not rely on gradients or derivatives of the reward function, this distinguishes it from most other optimization algorithms. Furthermore, being an off-policy algorithm allows Q-learning to learn from experiences obtained using a different policy

than the one that is currently being followed.

In Q-learning, the Q-values are stored in a Q-table. Each entry represents the expected return for taking a specific action in a specific state. This Q-table is initialized at the beginning of the algorithm, typically with small random values or zeros. As the agent interacts with the environment, it explores different state-action pairs and updates the corresponding Q-values based on the observed rewards and future expected rewards.

During exploration, the agent encounters both negative and positive rewards. These rewards are determined by the consequences of the actions that are chosen in the environment. The rewards can reflect the immediate outcomes of actions or the cumulative effects leading to a terminal state. The Bellman equation plays an important role in Q-learning. It states that the value of being in a particular state is the immediate reward from that state plus the best possible value from the next state considering all possible actions. Using the properties of this equation, the algorithm is able to recursively update the Q-values to approximate the expected returns for each state-action pair. This equation ensures that neighboring states to terminal reward states get values that reflect how close they are to those rewards. From The Bellman equation, the following formula can be derived: [15]

$$Q(s,a) = r(s,a) + \gamma \cdot \max_{a'} Q(s',a') \tag{3.2}$$

- $Q(s,a)$ is the value of taking action a in state s.
- r is the immediate reward received after taking action a in state s.
- γ is the discount factor influencing future rewards.
- $\max_{a'} Q(s',a')$ represents the maximum value of all possible actions a' in the next state s'.

To update Q-values during the training of an agent using Q-learning the following formula is used:

$$Q(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s',a')) \tag{3.3}$$

- $Q(s,a)$: This represents the Q-value, which denotes the expected total reward that can be obtained by taking action a in state s.
- $(1-\alpha)$: This is the update factor for the old Q-value, where α is the learning rate. It signifies the portion of the old Q-value that is retained during the update.
- r: This stands for the reward received after taking action a in state s.
- γ: This is the discount factor, which indicates the extent to which future rewards are weighted relative to the current reward.
- $\max_{a'} Q(s',a')$: This represents the maximum Q-value of the next state s', taking the maximum value over all possible actions a' in that state.

3.4 Deep Q-learning for zero sum games

Solving complex zero-sum games such as chess using Q-learning poses a challenge due to the large state space, especially when the board has many pieces. With 64 squares, there are countless possible arrangements, making it computationally expensive to explore every position. Deep Q-learning addresses this issue by using neural networks to approximate the Q-function and estimate the values of different actions. This approach offers a more efficient alternative to the brute-force Q-learning method.

A Deep Q-Network (DQN) uses two networks: a target network and a policy network. Having a separate target network stabilizes training, because it provides fixed targets. This avoids fluctuations that can occur when updating Q-values. This setup also reduces correlations between the target and predicted Q-values, also leading to better training stability. The formula for a DQN combines the elements of the Bellman-equation and the use of two Convolutional neural networks (CNNs) as shown in 3.4.

$$Q(s,a;\theta) \approx Q_{\text{target}}(s,a;\theta) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta')\right] \tag{3.4}$$

- $Q(s,a;\theta)$: The estimated value of action a in state s with parameters θ of the policy network.
- $Q_{\text{target}}(s,a;\theta)$: The target value of action a in state s with the same parameters θ.
- $\mathbb{E}_{s' \sim \mathcal{E}}$: The expectation over the next state s' sampled from the replay memory $\mathcal{E}$.
- r: The immediate reward received after taking action a in state s.
- γ: The discount factor influencing future rewards.
- $\max_{a'} Q(s',a';\theta')$: The maximum estimated value of all actions a' in the next state s', with the parameters θ' of the target network.

3.5 Stockfish

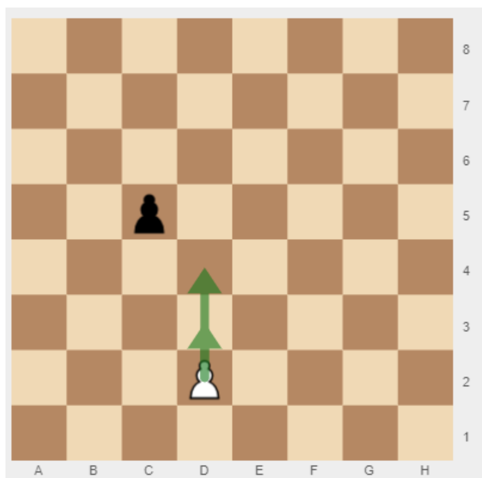
Stockfish is one of the strongest chess engines globally available and has a rich history and impressive capabilities. Stockfish evaluates positions deeply and accurately by analyzing millions of moves per second, making it a dominant player in computer chess competitions. It achieves this through techniques such as alpha-beta pruning with enhancements like aspiration windows and iterative deepening. Additionally, Stockfish uses a sophisticated evaluation function that includes neural networks and that considers things like material balance, piece activity, king safety and pawn structure to decide the strength of a position. In this study, Stockfish will be used as a strong opponent to test the different algorithms and agents.

4 The game of chess and chess endgames

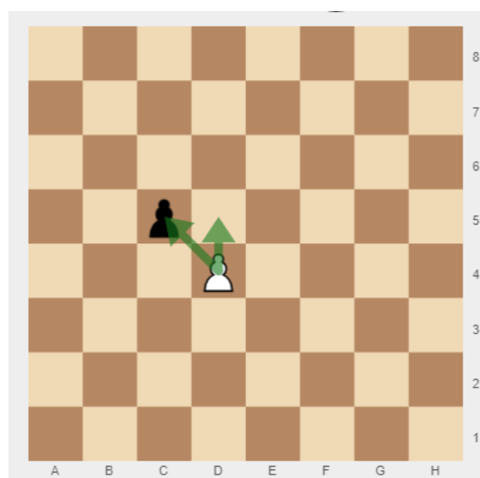
4.1 Game rules

Chess is a classic two-player strategy game played on a square board with 64 squares. Each player has 16 pieces: one king, one queen, two rooks (also known as towers), two knights (or horses), two bishops, and eight pawns. The primary objective of chess is to checkmate your opponent's king, which means putting the king in a position where it is under attack and cannot escape capture. Every piece has different moving restrictions.

Pawn: Pawns move forward one square, but capture diagonally. On their first move, they have the option to move two squares forward.



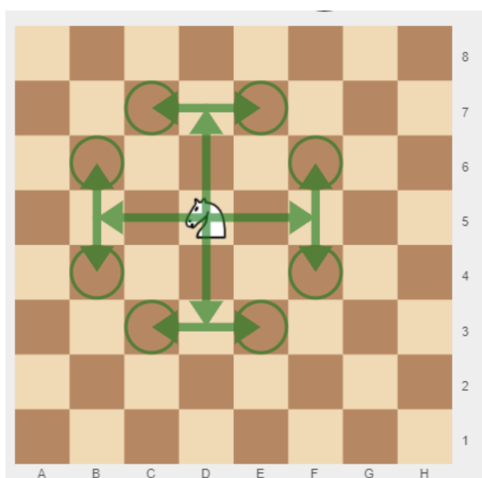
(a) The pawn can choose for 1 or 2 squares in starting position



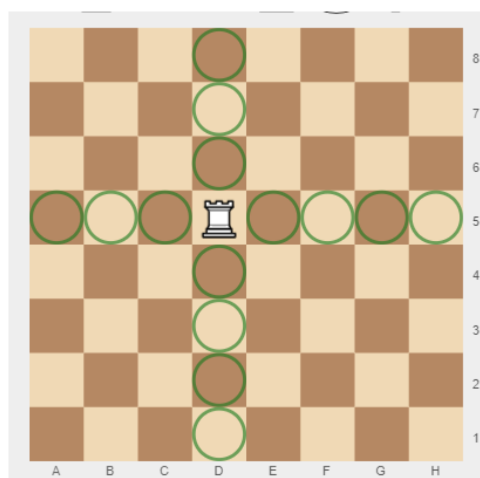
(b) The pawn moves 1 square and only captures diagonally

Knight: The knight moves in an L-shape, two squares in one direction and then one square perpendicular to that. It's the only piece that can jump over other pieces.

Rook: Rooks move horizontally or vertically any number of squares.



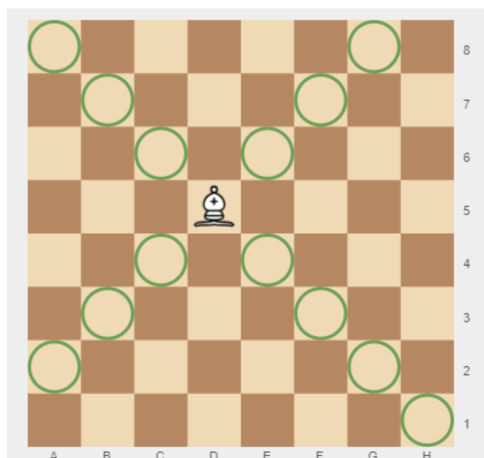
(a) The knight moves in a L-shape



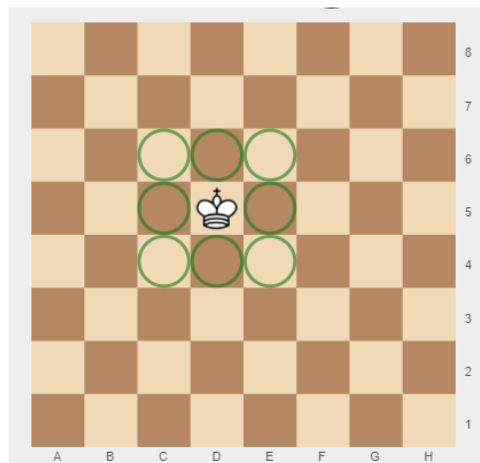
(b) The rook moves horizontally and vertically

Bishop: Bishops move diagonally any number of squares.

King: The king moves one square in any direction. It's the most important piece, and if it's under attack, it must be moved out of check.



(a) The knight moves in a L-shape



(b) The king can move to all adjacent squares

Queen: The queen combines the moves of the rook and bishop, moving horizontally, vertically, or diagonally any number of squares.

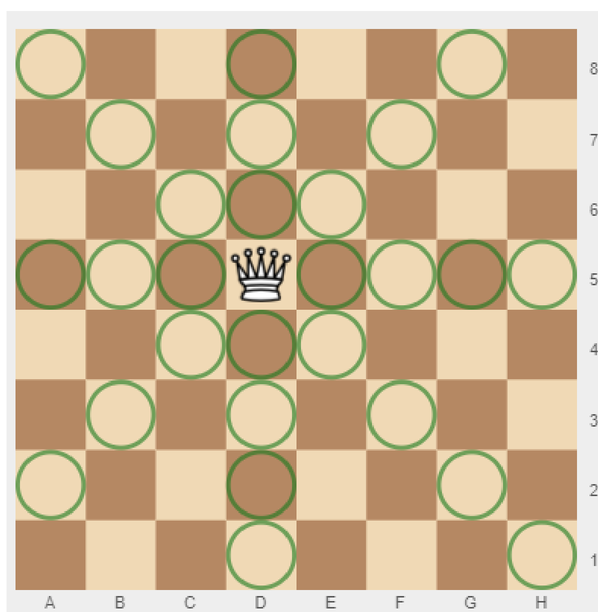


Figure 4.4: The Queen has acces to the most squares and is therefore the most powerful piece

For chess beginners, focusing on the endgame means dealing with clear tasks. One of the first things to learn is how to checkmate the opponent's king in a King + Queen versus king situation. After that a player will learn how to checkmate with King + Rook versus king. Once a chess player mastered this and becomes stronger overall, the player can move on to learn more complex endgame scenarios like King + Bishop + Knight versus King. This study explores how algorithms can help achieve these endgame goals, which are different in difficulty.

The endgame in chess is the stage of the game when few pieces are left on the board. It's a critical phase where players aim to use their remaining pieces efficiently to achieve victory. Often the last task in the endgame is to checkmate the opponents lone king with the help of your own king and the pieces you still have. The most common situations in endgames involve achieving checkmate using specific

combinations of pieces, such as the rook, queen, bishop, and knight. Each piece has its own unique abilities and strategies to contribute to the checkmate.

Checkmate with the Rook: (KRK endgame) Rooks are often used to deliver checkmate by controlling ranks or files, restricting the opponent's king's movement until it is cornered and checkmated.

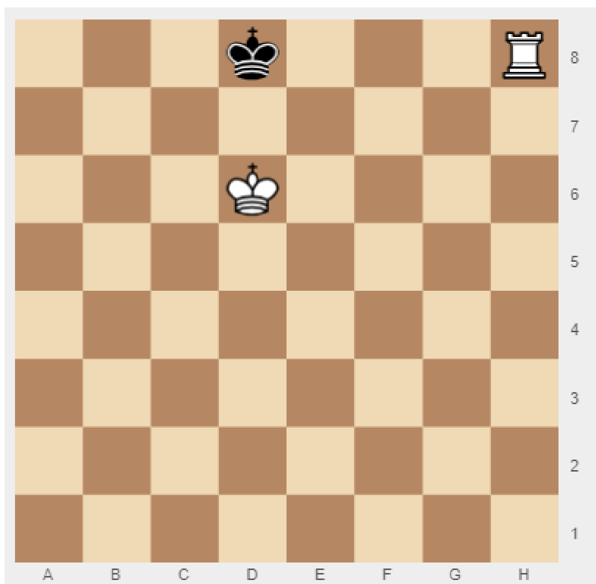


Figure 4.5: The rook checks the king on the back rank while the white king protects all squares on the rank before that

Checkmate with the Queen: (KQK endgame) Queens are powerful pieces that can deliver checkmate along ranks, files, or diagonals, often coordinating with other pieces to create mating threats.

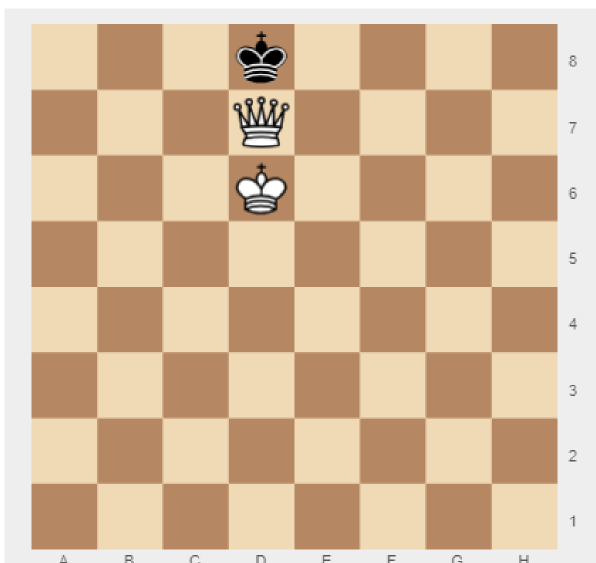


Figure 4.6: The Queen checks the black king and protects all squares while the white king defends the queen

Checkmate with the Bishop and Knight: (KBNK endgame) Checkmating with a bishop and knight is an elegant and challenging technique. By working together, these pieces can create a mating net, gradu-

ally driving the opponent's king towards the corner of the board until it's checkmated. This checkmate can only be forcefully delivered if the king is driven to the corner that has the same colour as the bishop is on.

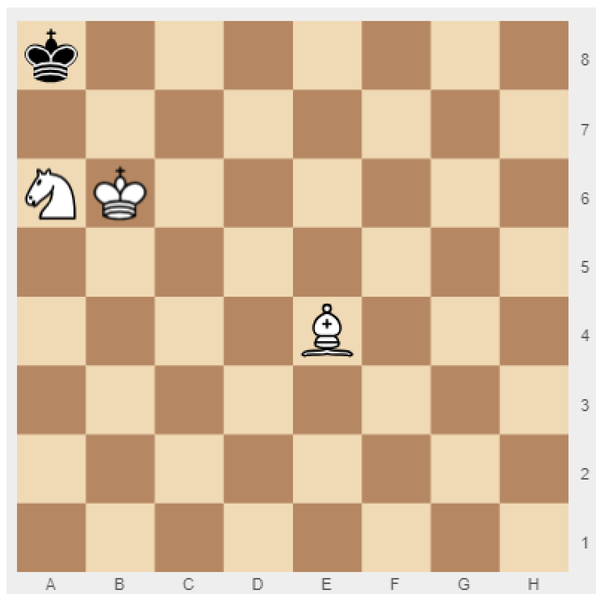


Figure 4.7: The knight and king protect all surrounding squares of the black king while the bishop checks the black king

4.2 Stalemate

In a checkmate situation the black king cannot go to any square and is in check at the same time. However, it can also occur that the black king cannot move to a square while it is not in check. In that case it is stalemate, which means the game is drawn.

4.3 Insufficient material

In the endgames in this study, the white side has just enough pieces to force a checkmate. If it loses a piece because the black king captures it, checkmate can no longer be forced. It will then be a draw.

4.4 50-move rule

In chess, the fifty-move rule means that if no pawn is moved and no piece is captured within fifty consecutive moves by each player, the game ends in a draw.

In simulations in this study, there is set a maximum of fifty moves in each simulation. If the game doesn't end with checkmate within these fifty moves, it's considered a draw. To encourage quicker checkmates within this rule, there's often a small penalty for each move, pushing agents to aim for checkmate within the fifty-move limit.

4.5 Three fold repetition

If a position occurred 3 times in a game, the game is considered a draw. This rule will also be embedded in evaluation functions during this study, preventing that an algorithm will go back and forth in a certain position.

5 Implementation

5.1 Simulating endgames

This study used the Python chess library to establish a chess environment capable of executing moves on a virtual chessboard. The observation space within this environment is the chessboard itself. Adherence to the rules of chess is strictly enforced within this environment, allowing only moves that comply with the rules of the game. In this environment, chess endgame positions can be initialized using the Forsyth-Edwards Notation (FEN). FEN is a compact way to represent the current position of a chess game using alphanumeric characters. It includes information about the placement of pieces, the player to move, castling rights, en passant squares and the number of moves made in the game so far. The ranks (8-1) are seperated by a slash /. Pieces are represented with the first letter of the piece, except for the Knight that is represented with the letter N. Numbers represent the amount of empty squares. An example of a King + Rook versus King (KRK) position with a board FEN notation can be seen in 5.1. In the experiments in this study, a random board FEN will be generated for the endgames in section 4.1 after which games can be simulated.

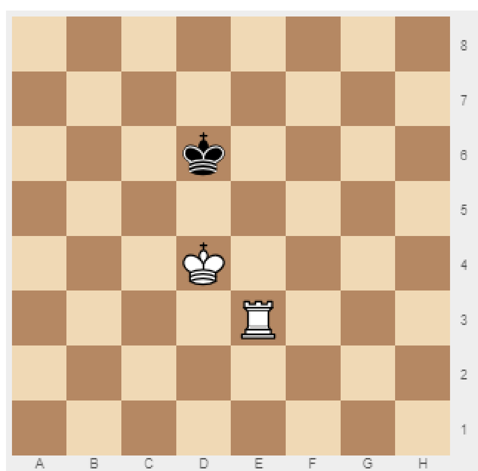
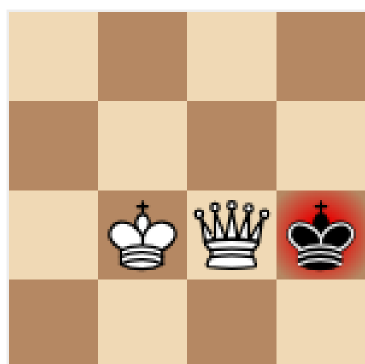


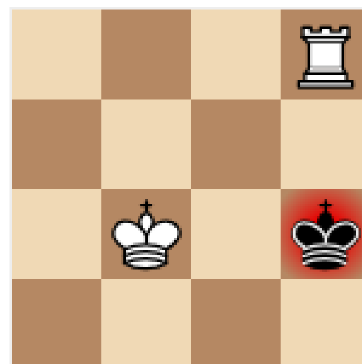
Figure 5.1: Example of KRK endgame position with FEN: 8/8/3k4/8/3K4/4R3/8/8

During this study, also chess games on a smaller board will be generated for experiments. On a smaller chess board the rules and principles for checkmating with the King and Rook or Queen stay the same.

Figure 5.2: Checkmate delivered on a smaller board



(a) Checkmate with the Queen on a 4x4 board



(b) Checkmate with the Rook on a 4x4 board

5.2 Monte Carlo Tree Search methods

MCTS does not need an evaluation function as the MiniMax method does, but will simulate a lot of games to the end with random moves to evaluate the potential of possible moves. The way in which these moves are selected is called the rollout policy. The default way to select these moves is randomly, but this can also be done using heuristics of the game.

In this study depth limiting in is used to restrict the number of moves considered in each playout. This allows for more simulations within the same time frame. This technique is particularly useful, because the optimal number of moves to achieve checkmate is not very high and simulating towards a high number of moves (e.g., 50 moves) does not provide useful information about the potential of the chosen starting move. Overall depth limiting ensures that the simulation resources are used more efficiently.

Because MCTS produces noisy results, it is expected that it does not perform well for chess endgames without an advanced playout policy. Therefore, for this research datasets were created using online syzygy table bases. These tablebases have a lot of endgame positions with a known optimal distance to checkmate. The created dataset has endgame positions with varying levels of difficulty for the three different types of endgames. The difficulty level depends on how quickly checkmate can be achieved if white and black would both play the best sequence of moves.

5.3 MiniMax methods

The main requirement for the application of MiniMax is the development of an evaluation function, which provides a static evaluation of a given chess position. For basic chess endgames, several key heuristics come into play. These include checking for checkmate and draw conditions, evaluating the position of the opponent's king to critical board locations such as the back rank or corners and considering the distance of pieces to the opponent's king. In Section 5.5, these heuristics are translated into an example evaluation function.

In this study all possible moves in a certain position are taken and the minimax algorithm is used to compute the corresponding evaluations. The move that gives the highest evaluation score is then played.

5.4 Chess heuristics

The most simple heuristics in a game of chess are whether a position is checkmate or a draw. However, there are countless other heuristics in chess. These heuristics measure elements of the game like piece activity, piece coordination, pawn structures etc. For endgames, these heuristics and their importance are easier to measure because of their simplicity. For instance, if the white side wants to checkmate the black side, it's important to force the black king to the edge or corner of the board. To measure the progress, vector norms can be used. Two distance measures that are very useful in the game of chess are the Chebyshev distance and Manhattan distance.

The Chebyshev distance measures the maximum distance between two points in a grid, allowing movement in any direction (horizontally, vertically, or diagonally). It has the following formula:

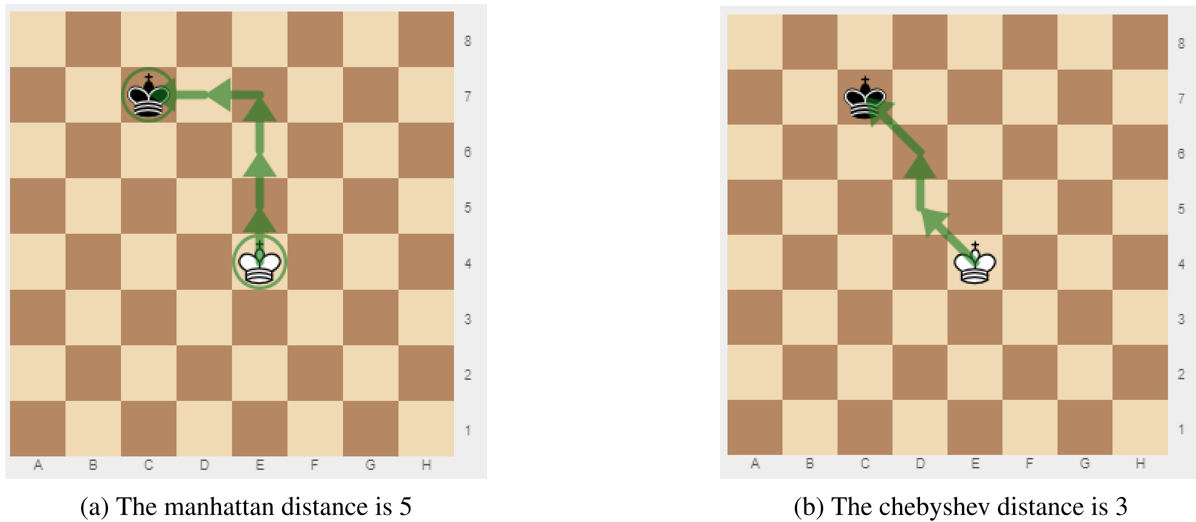
$$\|\vec{v}\|_\infty = \max(|v_x|, |v_y|)$$

The Manhattan distance measures the distance between two points in a grid, considering only movement along the grid lines (horizontally and vertically). The formula is as follows:

$$\|\vec{v}\|_1 = |v_x| + |v_y|$$

Applying this to the game of chess, the distances will be computed as in figure 5.3.

Figure 5.3: Manhattan and Chebyshev distance between the kings



(a) The manhattan distance is 5

(b) The chebyshev distance is 3

5.5 Reward and evaluation functions

Both the minimax algorithm and reinforcement learning algorithms need a predefined evaluation and reward function. Minimax relies entirely on the evaluation function it receives to assess the quality of each chess position. The RL algorithms use the reward system they receive as a guiding mechanism throughout the learning process. An example of a simple reward system for a white RL agent that has a checkmating task can be seen in 5.1. An example of a simple evaluation function for a white agent that has a checkmating task can be seen in 5.2, here d is the distance to the edge of the board. Different evaluation functions will be tested in Section 6.

Table 5.1: Reward System for white agent in chess endgame

Outcome	**Reward**
Win	+50
Move	-1
Draw	-50

Table 5.2: Evaluation function for King+Rook vs King situation

State	**Evaluation**
Checkmate	+50
Draw	-50
Enemy king in a corner	+10
Enemy king close to the edge	$+d$
Enemy king far from the edge	$-d$

5.6 Q-learning methods

5.6.1 Q-table representation

To represent chess endgames using a Q-table, the table is set up with one axis representing all possible positions and the other axis representing all possible moves an agent can make. The number of possible positions depends on the number of pieces on the board. In a standard chessboard with 64 squares, there are 64^n ways to arrange n pieces, although not all positions are legal. Each move in the Q-table is denoted by the piece and the square it moves to. For a KRK endgame this results in $64 * 2$ columns for the white Q-table and 64 columns for the black Q-table. However, in a given position, not all squares may be reachable legally. As a result, a significant portion of the table remains unused, with only legal moves being used for learning. The resulting table can be seen in 5.4

To initialize the Q-tables, all entries in the white Q-table are set to zero, while all entries in the black Q-table are initialized to a value of 50. This sets the stage for learning and adapting strategies as the agent explores different moves and positions in the endgame scenario.

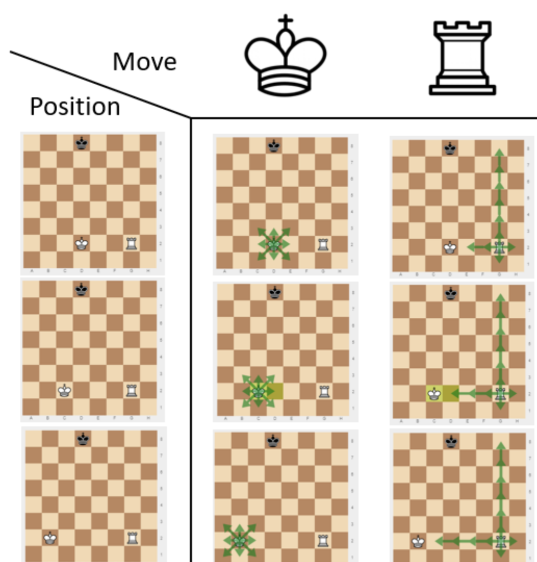


Figure 5.4: Q-table for King + Rook vs King endgame

5.6.2 Exploration strategy

While exploring different methods, a decaying epsilon strategy has been chosen. This strategy involves reducing the value of epsilon gradually over successive episodes. Initially, a higher epsilon allows for increased exploration, enabling the agent to explore a broader range of moves and positions. As training progresses and the agent accumulates more experience, the exploration rate decreases. This strategy maintains a good balance between exploration and exploitation for dynamic environments [16], which is also essential for effective learning in chess endgames.

5.6.3 Q-learning algorithm for a chess endgame

In figure 5.5 the State, Action, Reward, Next State and Max Q-value respectively can be seen for a checkmate in 2 situation. After the action of white is selected, the reward is -1, because there is a -1

reward for making a move for white. The next state observed is after black plays its move. The corresponding Q-value of the next best action in this case should be 50, because white is able to checkmate the king with a rook move.

In this study, two agents will undergo simultaneous training. One agent is dedicated to playing as the white side, with the objective of checkmating the black king. Conversely, the black agent is tasked with implementing the most effective defense strategy, aiming to avoid and delay checkmate for as long as possible.

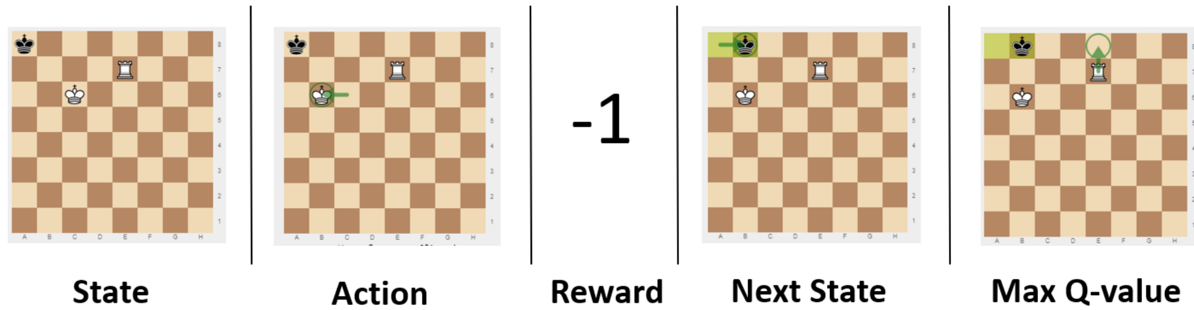


Figure 5.5: (State, Action, Reward, Next state, Max Q-value) for checkmate in 2 position

During each episode, a complete chess game is played out and the Q-learning algorithm updates the values within the Q-table. The exploration strategy will be a decaying epsilon strategy as discussed in section 5.6.2. Additionally, the learning rate and discount factor are represented by α and γ, respectively. The Q-learning algorithm applied to a chess endgame is shown in Algorithm 4.

Algorithm 4 Q-learning chess endgame (episodes, Q-table, ϵ-decay α, γ
)

1: **for** episode **in** episodes **do**
2: initialize starting position
3:
4: **while** game is not over **do**
5:
6: **if** white to move **then**
7: Select action a for white agent based on ϵ-greedy strategy
8: **if** game is over **then**
9: Q-table white(position, move) = reward
10: **else**
11: Create copy of environment
12: find and execute move of opponent with highest Q-value
13: **if** game is over **then**
14: $maxQ^{future} = reward$
15: $Q^{new} = Q^{old} + \alpha * (reward + \gamma * maxQ^{future} - Q^{old})$
16: **else**
17: $maxQ^{future} =$ Q value of strongest move
18: $Q^{new} = Q^{old} + \alpha * (reward + \gamma * maxQ^{future} - Q^{old})$
19:
20: **if** black to move **then**
21: Select action a for white agent based on ϵ-greedy strategy
22: **if** game is over **then**
23: Q-table black(position, move) = reward
24: **else**
25: Create copy of environment
26: find and execute move of opponent with highest Q-value
27: **if** game is over **then**
28: $maxQ^{future} = reward$
29: $Q^{new} = Q^{old} + \alpha * (reward + \gamma * maxQ^{future} - Q^{old})$
30: **else**
31: $maxQ^{future} =$ Q value of strongest move
32: $Q^{new} = Q^{old} + \alpha(*reward + \gamma * maxQ^{future} - Q^{old})$
33:
34: **if** Epsilon is between begin decaying and end decaying **then**
35: $\epsilon = \epsilon - \epsilon$-decay

5.7 Deep Q-learning methods

5.7.1 State representation

In this study, chess positions are represented as a 3x8x8 matrix for training the Deep Q-Network (DQN). In this matrix, the second two dimensions (8x8) correspond to the chessboard grid, while the first dimension (3) represents the three types of chess pieces: King, Rook and King.

Each square on the chessboard is encoded using a one-hot encoding scheme, where all elements are zeros except for the position where a piece is located, which is marked as one. This encoding method allows the convolutional neural network (CNN) to interpret the chess positions as if they were images, with each channel corresponding to a different type of chess piece. This way of representing the position of pieces can be seen in figure 5.6.

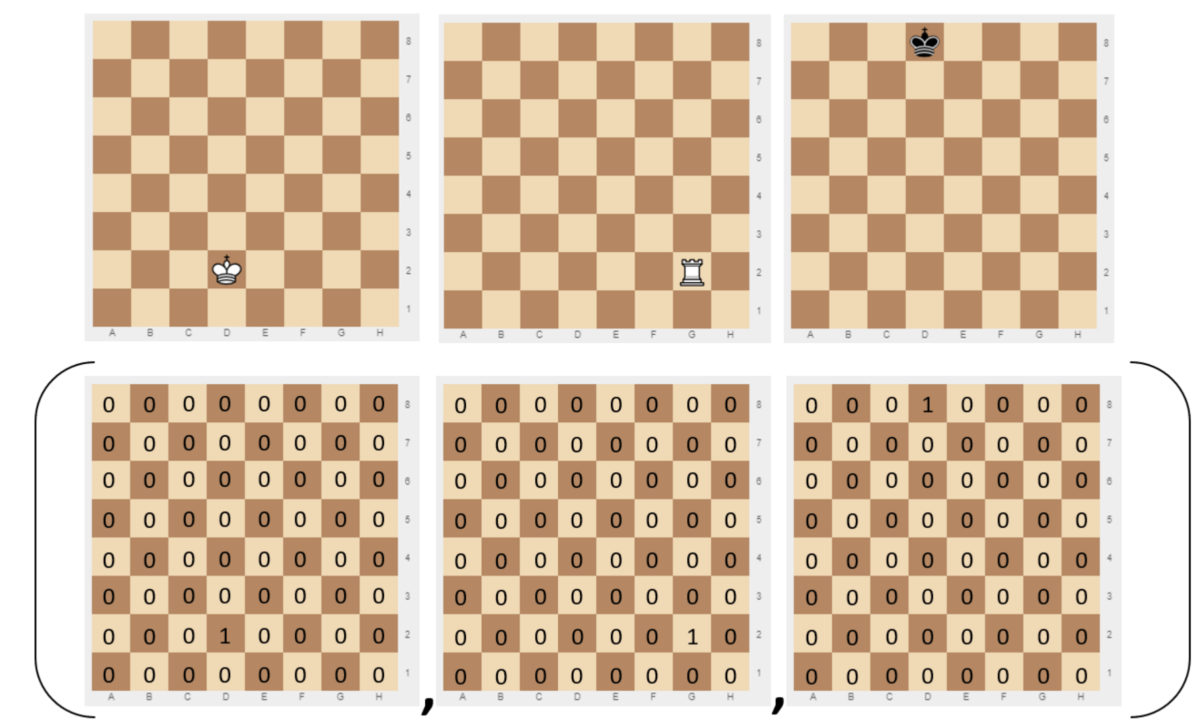


Figure 5.6: 3 channels representing the position of 3 different chess pieces

5.7.2 Action representation

The neural network's output layer represents all possible actions, each associated with a predicted Q-value. In chess, the actions consist of moving pieces to different squares on the board. However, considering every piece and the potential square it can move to results in a large action space. To address this, the output layer of the network is structured to include each piece and its permissible moves based on standard chess rules. For example, a king can move one square horizontally, vertically or diagonally, resulting in eight possible moves. Similarly, a rook can move horizontally or vertically in four directions, with a maximum of seven squares per direction. Consequently, the white agent in a KRK endgame has an output layer size of 36, calculated as 8 (for the king) plus 7 times 4 (for the rook). This is reconstructed in figure 5.7. The red circles mean the move is not legal. The numbers 15, 19-22, 24-29 and 34-36 are not on the board and would therefore be considered illegal moves as well.

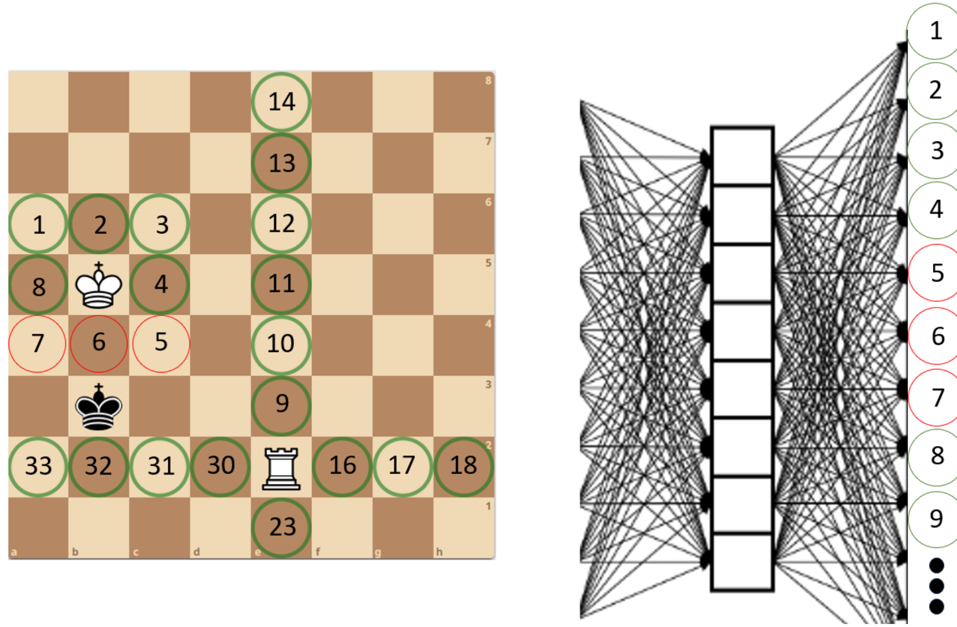


Figure 5.7: The moves represented as actions in the neural networks output layer

5.7.3 Neural network structure

The CNN for both agents will have 1 input layer, 2 hidden layers and 1 output layer. The input size will always be nx8x8, where n is the amount of pieces. After each layer, there will be an activation layer with the ReLu function given in 5.1.

$$f(x) = \max(0, x) \tag{5.1}$$

The loss of the model is calculated with the mean squared error (MSE) formula. The loss will be the difference between the predicted Q-values from the policy network and the target value that is calculated using the target network as seen in 5.2. After this, the model is optimized using the update function in 5.3 using the Adam optimizer with a learning rate of 0.001.

$$Y = R + \gamma \max_{a'} Q(S', a'; \theta') \tag{5.2}$$

- Y: The target value for the Q-learning update.
- R: The immediate reward received after taking action a in state S.
- γ: The discount factor influencing future rewards.
- $\max_{a'} Q(S', a'; \theta')$: The maximum estimated value of all actions a' in the next state S', with the parameters θ' of the target network.

$$\theta \leftarrow \theta - \alpha \nabla_\theta (Y - Q(S, a; \theta))^2 \tag{5.3}$$

- θ: The parameters of the neural network, which are updated during training.
- α: The learning rate, determining the size of the step taken during parameter updates.

- ∇_θ: The gradient operator with respect to the parameters θ.
- Y: The target value for the Q-learning update.
- $Q(S, a; \theta)$: The estimated Q-value for taking action a in state S with parameters θ.

5.7.4 DQN algorithm for chess endgame

The DQN algorithm is similar to the Q-learning algorithm for a chess endgame. The difference is that DQN uses a target network to provide Q-values and that it trains the policy network using batches of data from a replay memory through backpropagation with an optimizer algorithm. The full training process for a chess endgame can be seen in Algorithm 5. If the network returns an illegal move as the move with the highest Q-value, an experience with a penalty will be added to the replay memory. In that case a random move will be executed to keep the game going.

Algorithm 5 Training DQN for Chess Endgames

1: Initialize policy network weights θ randomly
2: Clone policy network to create target network with weights θ'
3:
4: **for** each episode **do**
5:     Initialize environment with a random configuration of King, Rook, and King position
6:
7:     **while** playing a game **do**
8:
9:         **if** white to move **then**
10:             Select action a for white agent based on ϵ-greedy strategy
11:             Execute action a for white agent
12:             Execute optimal action for black agent based on current policy
13:             Observe reward R and next state S'
14:             Store experience tuple $(S, a, R, S', \text{done})$ in replay memory D_W
15:             Sample random mini-batch from replay memory D_W
16:             Calculate target Y for white agent using target network: $Y = R + \gamma \max_{a'} Q(S', a'; \theta')$
17:             Update policy network θ for white agent using ADAM: $\theta \leftarrow \theta - \alpha \nabla_\theta (Y - Q(S, a; \theta))^2$
18:
19:         **else**
20:             Select action a for black agent based on ϵ-greedy strategy
21:             Execute action a for black agent
22:             Execute optimal action for white agent based on current policy
23:             Observe reward R and next state S'
24:             Store experience tuple $(S, a, R, S', \text{done})$ in replay memory D_B
25:             Sample random mini-batch from replay memory D_B
26:             Calculate target Y for black agent using target network: $Y = R + \gamma \max_{a'} Q(S', a'; \theta')$
27:             Update policy network θ for black agent using ADAM $\theta \leftarrow \theta - \alpha \nabla_\theta (Y - Q(S, a; \theta))^2$
28:
29:         Update target network weights θ' to weights θ every C steps

6 Experiments and Findings

6.1 Tree algorithms for Different Endgames

6.1.1 Time complexity

Both Monte Carlo Tree Search and minimax use information from potential scenarios to decide on the best move. The depth of this lookahead correlates exponentially with the number of possible positions. Table 6.1 illustrates the development of this for the three different endgames.

Table 6.1: Average total number of chess positions in the complete game tree for 20 random games at different depths for the 3 different endgames

Depth/Endgame	KQK	KRK	KBNK
0	1	1	1
1	2.72×10^1	1.00×10^1	1.86×10^1
2	1.14×10^2	9.63×10^1	9.30×10^1
3	3.14×10^3	1.97×10^3	1.70×10^3
4	1.23×10^4	9.12×10^3	7.74×10^3

The number of game states changes differently for each endgame. The states for rook and knight plus bishop are quite similar. The queen can make the most moves per turn, even more than the bishop and knight together. This makes the number of game states grow the fastest. On the other hand, forcing a checkmate with a bishop and knight is much more complicated and has fewer possibilities. For all endgames it can be seen that analyzing positions gets significantly more complex from a depth of 4.

6.1.2 MCTS algorithm for different endgames

As mentioned in section 5.2, it is challenging for MCTS to win chess endgames with a random playout policy due to the large number of possible moves. To analyze MCTS, it was given different endgame positions from the 3 endgames, with increasingly difficult positions to solve. 20 positions of each endgame were generated due to the running time of the algorithm. The optimal amount of moves to checkmate in the positions is known. However, MCTS gets 30 moves for both sides to try to win this endgame. If it is playing more moves, it will be assumed that it is not making any more progress. The results of this analysis are presented in Table 6.2.

Table 6.2: Checkmating rate of MCTS for 20 positions against stockfish using 1,000 simulations per move for different endgames with different difficulty

Checkmate/Endgame	KQK	KRK	KBNK
Mate in 1	100%	100%	100%
Mate in 3	100%	100%	100%
Mate in 7	100%	100%	10%
Mate in 11	50%	75%	10%
Mate in 15	40%	55%	0%
Random position	45%	15%	0%

Based on the results presented in Table 6.2, it can be concluded that Monte Carlo Tree Search performs

effectively in simpler endgames, but that it struggles significantly as the complexity of the endgame increases. This is especially the case for positions that have less possible ways to achieve checkmate. For example, while MCTS maintains a relatively high success rate in KRK and KQK scenarios, its performance drops considerably in more challenging situations for the KBNK endgame and randomly generated positions. This is not the case for the King + Queen endgame, but randomly generated positions for KQK have an overall lower distance to checkmate than 15.

6.1.3 Minimax algorithm for different endgames

The goal for chess players in a King + Rook and King + Queen vs. King situation is to push the opponent's king to the edge of the board. To checkmate, the white king needs to stay close to the black king. Therefore, using a combination of both strategies and looking a few moves ahead provides a good approach to force checkmate. Function 6.1 offers a possible method for this. In this context, 'r' and 'f' stand for rank and file, while 'W', 'B', and 'c' represent the white king, the black king, and the center squares.

$$f(x) = \begin{cases} 100 & \text{if checkmate} \\ 0 & \text{if stalemate} \\ 0 & \text{if three fold} \\ 0 & \text{if insufficient material} \\ \beta_1 \times (14 - \|(r_{\mathrm{W}}, f_{\mathrm{W}}) - (r_{\mathrm{B}}, f_{\mathrm{B}})\|_1) + \beta_2 \times \min_{\mathrm{c}} \|(r_B, f_B) - (r_c, f_c)\|_1 & \text{otherwise} \end{cases} \tag{6.1}$$

This is a linear combination of the Manhattan distance between the White (W) and Black (B) kings and the minimum Manhattan distance between the black king and the four center squares. As this value increases, the black king is closer to the edge of the board. The maximum Manhattan distance between the kings when they are both in opposite corners of the board is $7 + 7 = 14$. The Manhattan distance between the kings is subtracted from 14, so that the value gets higher when the kings are closer.

50 endgames are generated in advance. These endgame are played by the minimax algorithm with different depths against Stockfish. The win rates for these different depths are shown in table 6.3 and table 6.4. The average moves to checkmate for the games that ended in checkmate is also included.

Table 6.3: Win rate and average moves to checkmate of minimax algorithm against Stockfish within 50-move rule (100 moves for black and white in total) at different depths for King + Queen vs King endgame with $\beta_1 = 2$ and $\beta_2 = 5$.

Depth	Win rate	Avg moves to mate
1	64%	61.7
2	100%	21.6
3	100%	14.7
4	100%	13.3

Table 6.4: Win rate and average moves to checkmate of minimax algorithm against Stockfish within 50-move rule at different depths for King + Rook vs King endgame with $\beta_1 = 2$ and $\beta_2 = 5$.

Depth	Win rate	Avg moves to mate
1	16%	32.6
2	100%	23.8
3	92%	28.5
4	88%	29.0

The algorithm is able to gradually solve the King + Queen vs King endgame and the the King + Rook vs King endgame already at a depth of 2. It's noticeable that as the depth of minimax gets higher, the average win rate decreases. This might happen because the algorithm focuses too much on short-term advantages and misses opportunities to checkmate the opponent. Also, alpha-beta pruning might discard branches that seem less favorable at first, but could actually lead to a checkmate.

If the algorithm is applied to the King + Bishop + Knight endgame, it does not work. The bishop and knight endgame also is more complex as the king can only be forcefully checkmated in the corner square that has the same colour as the bishop. Also, only 1 move that lets the black king escape during the process of forcing it to the corner, already makes it such that the forcing sequence to checkmate cannot be completed within the 50-move rule. To be able to solve the bishop + knight endgame, the evaluation function has to be more advanced or a higher depth is needed to force a checkmate.

This depth issue is also evident in the King + Rook vs. King endgame, where the evaluation function is always optimized for the given depth. At a depth of 2, the endgame is always solved. However, it will first optimize the evaluation function at this depth, usually leading to a position like the one shown in figure 6.1. Here, the black king is as far from the center squares as possible. The white king is also close to the black king but cannot get closer without causing stalemate. This results in the white rook moving along the rank repeatedly until a threefold repetition becomes inevitable. Only then will it choose a different move, finding the deeper checkmate.

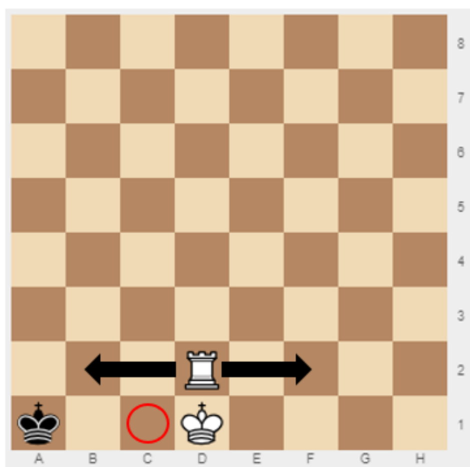


Figure 6.1: Both heuristics are optimized as the king getting closer leads to stalemate and the depth is not high enough to find the checkmate, leading to repeating moves

6.2 Q-learning on Different Board Sizes

6.2.1 Checkmating probabilities

The principles of checkmating the black king in the Rook and Queen endgame are not overly complex. Essentially, the king needs to be confined by the Rook or Queen in an increasingly smaller area, driving the black king towards the edge where it can be checkmated. In this study, we start with a smaller board than the standard 8 by 8 board, as the principles to achieve checkmate remain the same. For the King + Bishop and Knight this is not the case, as there must be a very specific coordination between the pieces. Therefore this endgame is not researched on a smaller board.

As the board size increases, the probability of achieving checkmate with random moves decreases, simply because it becomes more difficult to confine the black king with a larger board. In table 6.5, you can see the checkmating probabilities using Monte Carlo simulations with random moves for King + Queen and King + Rook versus king endgame for various board sizes.

Table 6.5: Stochastic checkmate probabilities for 10,000 random King + Rook and King + Queen endgames on different board sizes

Board Size/Endgame	**KQK**	**KRK**
4x4	2.84%	1.80%
5x5	2.02%	1.39%
6x6	1.12%	0.68%
7x7	0.88%	0.36%
8x8	0.95%	0.44%

6.2.2 Training on different board sizes

We simulate randomly initialized King Rook vs King and King + Queen vs King endgames on different boards with different sizes with the hyperparameters in 5.2. Optimistic initial values are used at the initialization of the Q-table for the black agent. The reason for this is to encourage the agent to explore more at the beginning. By starting with higher values, the agent is motivated to try different actions and learn more about the environment. This helps in better learning outcomes by finding a balance between trying new things and using what's already known. [17] The Q table of the white agent starts with zeros, but has a penalization of -1 for each move.

Table 6.6: Hyperparameters and Rewards for King Rook vs King endgame

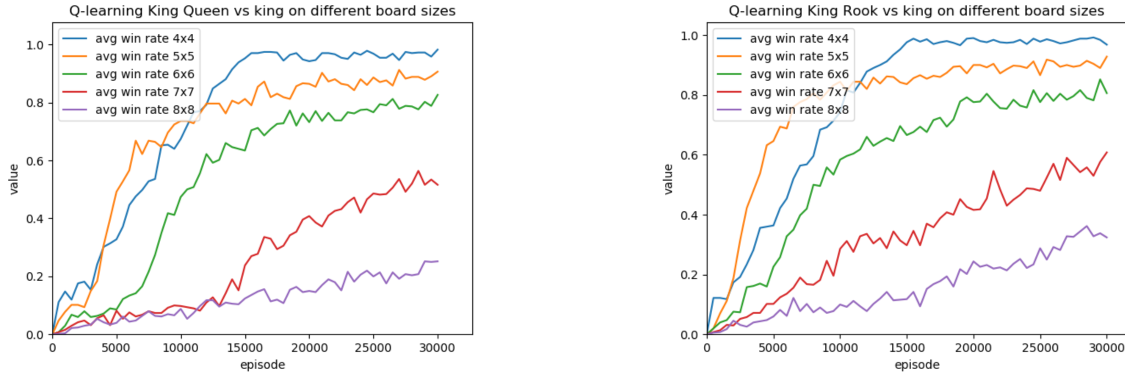
Table 6.7: Hyperparameters

Hyperparameter	**Value**
Episodes	30,000
Learning Rate	0.01
Discount Factor (Gamma)	0.99
Epsilon (Linear Decay)	0.9 to 0.05

Table 6.8: Rewards

Type of Q-value	**Value**
Initial Q-values for white agent	0
Checkmate for white agent	+50
draw for white agent	-50
move for white agent	-1
Initial Q-values for black agent	50
Checkmate for black agent	-50
draw for black agent	+50
move for black agent	+1

Figure 6.2: Training Q-learning agents to checkmate with king and queen on different board sizes



(a) Training Q-learning agents to checkmate with king and queen on different board sizes

(b) Training Q-learning agents to checkmate with king and rook on different board sizes

It is clear that training on a larger board takes longer to converge. To test the agents, they play against 3 different agents. First, a Random agent to see if the agent knows what to do in random situations. Second, a Black agent, which is trained in the same parallel way. Existing chess agents are trained to play on a full board. Therefore, the agents are tested against a heuristic agent that will minimize the black kings distance to the center of the board and will take every opportunity to draw the game. This means that the black king will stay away from the edge of the board as much as possible and that it will capture the white rook if it is able to, leading to stronger counter moves than the random agent.

Table 6.9: Win rate of different agents against different opponents over $N = 1{,}000$ games for King + Queen vs King endgame

Agent/Opponent	**Random**	**Black Agent**	**Heuristic**	**States visited**	**Average visit**
4x4 Agent	96.4%	100%	98%	2.34×10^3	98.6
5x5 Agent	72.6%	88.6%	83.8%	1.05×10^4	44.1
6x6 Agent	39.9%	80.2%	59.8%	3.03×10^4	19.1
7x7 Agent	26.7%	72.7%	55.4%	6.34×10^4	11.7
8x8 Agent	13.4%	40.9%	37.6%	1.03×10^5	7.51

Table 6.10: Win rate of different agents against different opponents over $N = 1{,}000$ games for King + Rook vs King endgame

Agent/Opponent	**Random**	**Black Agent**	**Heuristic**	**States visited**	**Average visit**
4x4 Agent	93.9%	99%	90.2%	2.32×10^3	152
5x5 Agent	60.9%	88.4%	75.1%	1.04×10^4	59.0
6x6 Agent	26.0%	76.6%	76.7%	3.01×10^4	29.6
7x7 Agent	13.6%	51.7%	51.2%	6.15×10^4	13.4
8x8 Agent	6.5%	14.7%	30.5%	9.66×10^4	9.03

It can be clearly observed that increasing the win rate becomes increasingly challenging as the board size grows. To ensure that the agent masters the endgames on the entire board, fine-tuning the hyperparameters is necessary. This is tested in section 6.3.

As the board size increases, the number of states visited grows exponentially, while the average number

of visits per state decreases significantly. Overall, these findings suggests that larger boards lead to a broader exploration of the state space but less intensive learning about each individual state.

Another thing that can be observed is the difference in win rates against the different opponents. The win rates are significantly higher against the self trained black agent than against the random and heuristic agent. This is most probably the result of the co-evolution of the 2 agents' strategies during training. They use each others values to predict the opponents next move and the next state. The heuristic agent scores in between the random and black agent. The heuristic agent will take the opponents piece if possible and will try to keep its king away from the edges of the board. The black agent also learns about this during training. This can therefore be an explanation why the white against scores better against the heuristic agent than against the random agent.

To further look into why the agents do not always win, the game outcomes that are not checkmate can be analyzed. The average rates for all agents on all board sizes can be seen in table 6.11.

Table 6.11: Reason for a draw in test games for KQK endgames

Agent/Draw reason	Stalemate	50-move rule	Lost piece	Repetition
Random Agent	8.60%	28.8%	53.2%	9.40%
Black Agent	7.90%	1.20%	42.3%	48.6%
Heuristic Agent	10.9%	0.30%	34.9%	53.9%

It can be observed that a significant portion of non-wins during the training of Q-learning agents arises due to the 50- move rule and move repetitions. This suggests that the current reward structure may not sufficiently penalize repetitions or incentivize progress towards checkmate. The Agents also do not have information about past moves. Consequently, agents may adopt strategies that involve repeated moves, avoiding immediate draws, but also failing to secure a win.

In Q-learning, overestimating action values can cause an agent to make suboptimal decisions, thinking certain actions are better than they are. This can lead to short-sighted behavior, especially if there's a penalty for each move, causing agents to favor less penalized but suboptimal actions.[18]

On the full board, the state space for the bishop and knight endgame becomes very large. This is because there are 4 pieces, including the white and black kings, that can be on 64 squares. Additionally, checkmating is much more complex and the chance of doing it randomly is much lower. Deep Q-learning provides a way to handle this larger state space.

6.3 Q-learning Agent on full board

6.3.1 Training against different opponents

In this section, different methods will be explored to successfully train an agent for the King + Queen vs King endgame for a complete chessboard. 100,000 episodes will be used to train the agents so they can explore more state spaces. It will also be tested how different opponents for training affects the training process. A different opponent has impact on the training process because it influences the next state. This allows Q^{future} to be estimated differently. The heuristic agent for example makes fairly good moves instantly instead of first having to learn. The results for training with the black agent, random agent and heuristic agent can be seen in table 6.14. The Black agent that is used as opponent is the agent that is trained parallel with the white agent that is trained in 100,000 episodes as well.

Table 6.12: Win rate for KQK endgame of different agents against different opponents over $N = 1{,}000$ games

Agent/Opponent	Random Agent	Black Agent	Heuristic Agent	Stockfish
Random Agent	2.00%	1.10%	2.50%	0.10%
Agent trained vs Random	10.2%	5.40%	13.6%	3.20%
Agent trained vs Heuristic	9.60%	5.30%	81.0%	2.00%
Agent trained vs Agent	33.6%	47.3%	50.3%	8.50%

The table shows that the agent trained against a random agent did not learn much compared to the other agents. The agent trained against the heuristic agent performs very strongly in test games against this opponent. This can be explained by the fact that the heuristic agent always follows the same policy in every situation, making it very predictable compared to the random agent and the black agent, which also learns and changes its policy over time. The results against Stockfish are the best benchmark for measuring the performance of the agents. Stockfish is a very strong opponent and is independent of any of the training processes. The performances are not very strong, indicating that there are still many states where learning has not occurred effectively. However, the results do show that training against a black agent, which also learns over time, results in the highest win rate.

6.4 DQN results

To train the DQN, there must be a penalty for illegal moves. Therefore, the agent will receive a reward of -50 for any illegal move. The reward system will be as described in Table 6.13. Initially, the agent will be trained against a random opponent due to less hours of training time. It can be expected that this might not lead to good training results, similar to the results of the Q-learning agent trained against a random opponent for a KQK endgame, as shown in Table 6.14. However, even when playing against a random opponent, the agent must learn to choose legal moves over illegal ones.

Table 6.13: Reward System for white DQN agent in chess endgame

Outcome	Reward
Win	+50
Move	-1
Draw	-50
Illegal Move	-50

The frequency of legal move selection when the select action function is called and a move is selected greedily will be tracked. This provides good insight into how effectively the DQN network chooses legal moves over illegal ones over the course of the training time. The results for this while training the DQN agent with 40,000 episodes are shown in figure 6.3.

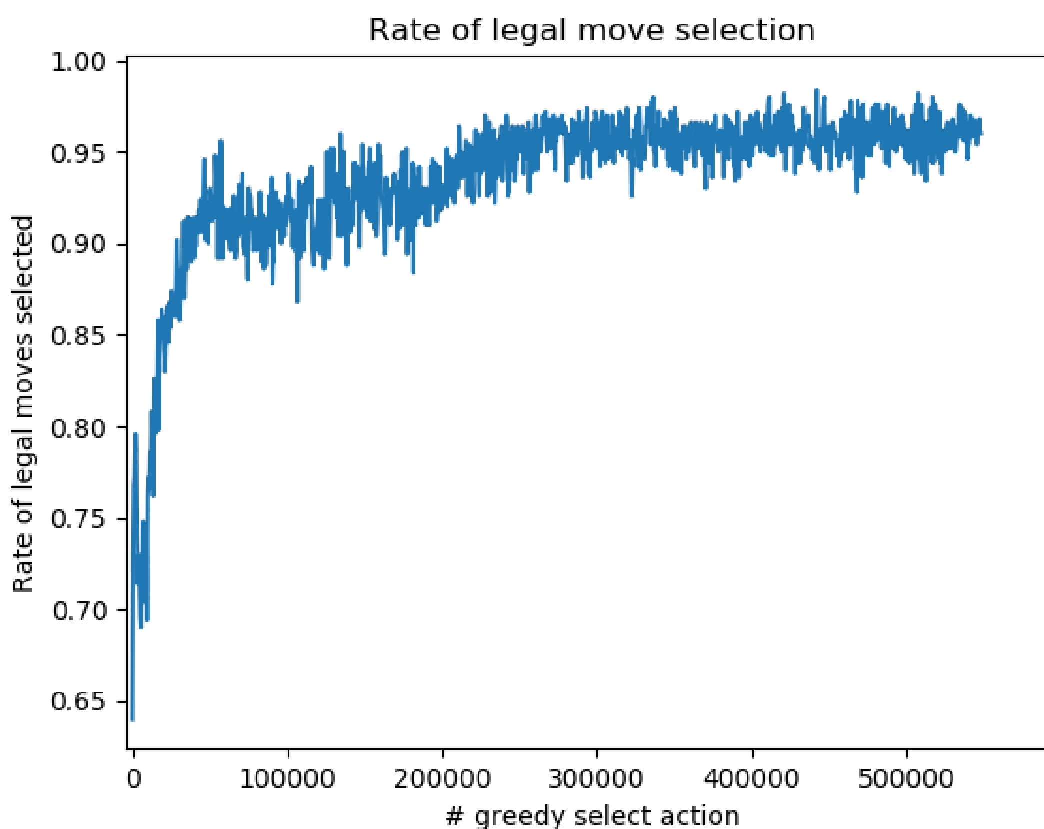


Figure 6.3: Rate of selecting a legal move in King + Rook endgame

The probabililty of a randomly selected action being legal is between 0.65 and 0.7. It can be observed that the Agent genuinely learns to choose legal moves over illegal ones. To test whether the Agens has generalized well, all starting positions from the training process were saved. The performance of the DQN agent on these training games will be compared to that on newly generated endgames.

Table 6.14: Results of DQN agent on 5,000 random starting positions from training and 5,000 new random starting positions

Games/Measure	Legal move	Win rate
Training games vs random	96.8%	36.7%
Training games vs stockfish	95.8%	0.70%
Test games vs random	96.5%	35.1%
Test games vs stockfish	94.4%	0.60%

The DQN agent shows that it learns to become better at playing against a random opponent. This is clear when you compare its win rate against a random player against a random opponent for a KRK endgame on a full board in section 6.2.1. However, its win rate against Stockfish is still very low. This means that while the algorithm has learned to handle randomness better, it doesn't know how to play against a strong opponent. This is not surprising since the agent was also trained against a random opponent. Due to time constraints, it was not possible to extend the research and conduct experiments involving Deep Q-learning for two agents. Looking at the results of Q-learning against stockfish with two agents this certainly has potential to become stronger at playing stockfish.

7 Conclusion and future work

This thesis explored the effectiveness of different algorithms, specifically Monte Carlo Tree Search (MCTS), minimax, Q-learning and deep Q-learning for solving basic chess endgames. The study compared traditional search tree methods with reinforcement learning approaches to determine their strengths and limitations in the context of chess endgames. The results indicated that while MCTS provides a comprehensive search of possible moves, its performance is hindered by the vast search space. This leads to significant computational time.

The minimax algorithm, even with a simple evaluation function, showed reliable performance, but faced challenges with the number of nodes growing exponentially with deeper searches. Higher depths led to a significantly larger number of nodes, while lower depths caused the algorithm to drive the king to the corner and repeat moves until just avoiding threefold repetition, only opting for deeper checkmate variants just before that point.

The application of Q-learning in mastering chess endgames presents both successes and challenges. The method led to effectively trained agents to achieve checkmate against different opponents on various board sizes, which shows its adaptability and potential in strategic decision-making. However, the need for a more extensive amount of training episodes, particularly on larger boards, underscores the complexity of the learning process. Furthermore, issues such as non-wins due to the 50-move rule and move repetitions highlight areas for possible improvement in reward structures and action value estimation. Overall, while Q-learning shows potential in learning endgame strategies, it would need more refinement to perform better in more complex decision-making situations.

Deep Q-learning showed potential in learning optimal strategies through experience and was able to recognize legal moves. However, it struggled with learning strong counter moves, due to the limitation of only training against a random opponent for now. When faced with new endgame scenarios against stockfish, its performance dropped significantly.

Future research could address the limitations of this research by exploring more efficient variants of MCTS, such as parallel MCTS or incorporating different heuristic evaluations to prune the search space more effectively. For minimax, implementing more enhancements can significantly reduce the number of nodes evaluated, improving its efficiency for playing chess endgames or other games.

Training with two agents as described in the methodology has more potential than training against a random opponent. Further research could explore this approach in more depth. It would also allow for experimentation with different strategies. For instance, one could pause the learning process of the agents alternately to stabilize the training process. Similar methods and other exploration-exploitation strategies could bring the DQN agent to a higher level.

To support the training of DQN agents, several extra techniques could be employed. Firstly, data augmentation techniques could be used to diversify the training data, thereby introducing some variations in endgame scenarios. This could be techniques such as rotating the board, mirroring positions or subtly adjusting piece placements. In this way data about states overall, but most of all terminal states could be generated faster. Additionally, regularization methods like L2 regularization (weight decay) could be implemented to penalize large weights in the neural network.

Early stopping could also be used to monitor the model's performance on a validation set while training and halt training when performance begins to stagnate. This would prevent the model better from learning noise in the training data. Ensemble learning techniques, such as bagging or boosting, could be used to combine predictions from multiple models. This could enhance generalization and lower the risk of overfitting.

Finally, transfer learning approaches can be used by pre-training the model on more chess games or a related task before fine-tuning it on specific endgame scenarios. This could enable the model to use prior knowledge and improve its generalization capabilities.

The King + Queen and King + Rook endgames have been proven useful during this research for quickly understanding the progress and limitations of the algorithms due to their simplicity. The King + Bishop + Knight vs King endgame is useful for testing whether an algorithm can recognize more complex patterns. In this study, none of the methodologies solved this endgame, but all methodologies showed potential to do so.

In conclusion, this research highlights the distinct strengths and limitations of MCTS, minimax, Q-learning and deep Q-learning when applied to different chess endgames. Each algorithm demonstrates unique advantages, such as MCTS's comprehensive search, minimax's reliability, Q-learning's adaptability and deep Q-learning's potential for learning patterns from experience. However, they also face significant challenges like computational intensity, scalability issues and extensive training requirements. Addressing these limitations through more advanced techniques can enhance their performance. Future research could focus on refining these algorithms and exploring their application to other strategic games or aspects of games, but also to complex real-world decision-making scenarios, thereby broadening their usefulness and effectiveness.

Bibliography

[1] Mathematics Stack Exchange. combinatorics - tictactoe state space choose calculation, 2020. Retrieved 2020-04-08.

[2] Jürgen Hänggi, Karin Brütsch, Adrian M Siegel, and Lutz Jäncke. The architecture of the chess player's brain. *Neuropsychologia*, 62:152–162, 2014.

[3] David Silver, Aja Huang, Chris J Maddison, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[4] Kai Arulkumaran, Antoine Cully, and Julian Togelius. Alphastar: An evolutionary computation perspective. *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2019(1):314–315, July 2019.

[5] Noam Brown and Tuomas Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018.

[6] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Daniel Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[7] John von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[8] Christopher J.C. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

[9] Junling Hu and Michael P Wellman. Nash q-learning for general-sum stochastic games. *Journal of machine learning research*, 4(Nov):1039–1069, 2003.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[11] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2018.

[12] Mohammed AbuRass. The future of technology, 2023.

[13] S. H. Fuller, J. G. Gaschnig, and J. J. Gillogly. Analysis of the alpha-beta pruning algorithm. page 0055, 1973.

[14] Fractalytics. Application of MCTS within the Connect4 game, april 20 2021.

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, second edition, 2018.

[16] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[17] Eyal Even-Dar and Yishay Mansour. Convergence of optimistic and incremental q-learning. *Advances in neural information processing systems*, 14, 2001.

[18] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2016.