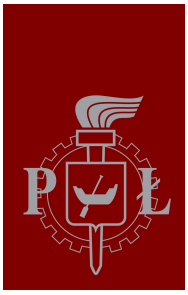


Lodz University of Technology

Faculty of Electrical, Electronic, Computer and Control Engineering

Institute of Applied Computer Science

Bachelor of Science Thesis

Applying Deep Reinforcement Learning Algorithms to the Game of Draughts

Zastosowanie algorytmów uczenia głębokiego ze wzmacnianiem w grze w warcaby

Kateryna Ocheretian

Student's number: 226448

Supervisor:

Piotr Duch, Ph.D Eng.

Lodz, February 2022

Abstract

Games are used as a playground for AI algorithms because they resemble a very simplified model of the real world, and by solving problems in them, we can learn how to solve more complicated problems in reality. In the literature, we can find many examples of the use of AI in board games such as chess, checkers and the game of Go.

This study aims to implement a deep reinforcement learning algorithm for solving the game of draughts. A proposed network consists of two parts. The first network, based on the board state, selects the piece that it wants to move. Based on the board and the selected piece, the second network chooses a move for it. This algorithm was tested against search algorithms (Minimax, Alpha-beta pruning, Monte Carlo tree search) and deep learning algorithms. Even though the network could not learn how to choose the right moves only, it could win most games against other algorithms.

Keywords: deep reinforcement learning, deep Q-learning, machine learning, draughts, neural networks.

Streszczenie

Gry służą jako środowisko do testowania algorytmów AI, ponieważ przypominają bardzo uproszczony model świata rzeczywistego, a rozwiązując występujące w nich problemy, możemy nauczyć się rozwiązywać bardziej skomplikowane problemy w rzeczywistości. W literaturze możemy znaleźć wiele przykładów wykorzystania AI w grach planszowych, takich jak szachy, warcaby czy gra w Go.

Niniejsze badanie ma na celu zaimplementowanie algorytmu uczenia głębokiego ze wzmocnieniem do rozwiązania gry w warcaby. Proponowana sieć składa się z dwóch części. Pierwsza sieć, na podstawie układu planszy, wybiera figurę, którą chce przesunąć. Na podstawie planszy i wybranej figury, druga wybiera dla niego ruch. Algorytm ten został przetestowany względem algorytmów wyszukiwania (Minimax, Alpha-beta, Monte Carlo tree search) oraz algorytmów głębokiego uczenia. Mimo że sieć nie była w stanie nauczyć się wykonywać tylko właściwe ruchy, to była w stanie wygrać większość gier z innymi algorytmami.

Słowa kluczowe: uczenie głębokie ze wzmocnieniem, deep Q-learning, uczenie maszynowe, warcaby, sieci neuronowe.

Contents

Chapter 1

Introduction

The development of machine learning (ML) algorithms began with the advent of artificial intelligence (AI). In the literature, we can find many examples of applying AI algorithms in games. This is so important because the games resemble a very simplified model of the real world, and by solving problems in games, we learn to solve problems in reality [1].

The first program capable of self-learning in the game of checkers was invented by Arthur Samuel in 1952 [2]. The program analyzed the current positions and chose the best options for subsequent moves. Then Samuel gave the first definition of machine learning - "Field of study that gives computers the ability to learn without being explicitly programmed" [3]. In 1957, a neural network (NN) model like modern deep learning (DL) algorithms were proposed. Since then ML systems have been actively developed, including decision tree algorithms (1986), support vector machines (1995) [4], and since 2005 - deep learning.

This thesis aims to develop deep reinforcement learning algorithm for the game of draughts. Developed algorithm is compared with selected algorithms in terms of their speed and performance. In this thesis two approaches for neural network training are presented. One based on data gathered from games played by other algorithms. And second one based on self-play. In order to increase learning outcomes the network will be paralleled train on two types of data: forbidden moves - to learn rules of the game, and correct moves, to learn how to play in an optimal way.

Chapter 2

The game of draughts

Draughts (British English), more commonly known as checkers (American English), is a well-known board game that is mainly played by two people. A single game of checkers can also be played by two groups of two. In the game of checkers pieces moves across the diagonal fields. And the goal is to win the game by capturing all of the opponent's pieces [5].

This game was chosen for the project because it is a deterministic game with perfect information. It means that the course of the game depends only on the players' decisions, and both players can see the full game state at all times (the whole board is visible).

2.1 History

The history of the origin of this game goes far into antiquity and is considered a mystery. To date, there are only myths about the origin of checkers. This wonderful game originated in different nations.

The most reliable version of the origin of checkers says that the game appeared in ancient Egypt (Fig. 2.1). Evidence is the paintings in the tombs of nobles close to the pharaohs. Also, in the tomb of Tutankhamun, a board broken into 30 cells was found.

Until the nineteenth century, checkers was not a popular game. Although back in 1547 in Spain, the first manual on checkers by A. Torquemada was published, which has not survived

Figure 2.1: Ancient Egyptian wall paintings of a game very similar to checkers [Source: gamescasual.ru]

to this day. Except for Spain itself, as well as France and England, checkers was played almost nowhere else in those days. This, despite the fact that the church, which condemned gambling, made no claims against checkers [6].

Modern checkers was invented by the French, around the middle of the nineteenth century, who combined the ancient Greek game Alkerk with a chess. Checkers pieces began to move like the king in chess. In England, the game is still called - drafts - the king's move in chess.

The first World Drafts Championship was held in 1847. The winner was Andrew Anderson from Scotland. Since most of the games at the world championship ended in a draw, in 1890 the two-move rule was introduced - the first two moves were chosen absolutely randomly. In 1934, the rule was transformed into a three-way ban, when already three moves were chosen randomly, but from an allowed list of combinations. Over time, the championships began to gain momentum and are regularly held in almost every country. Nowadays, checkers is considered a sport game. Many people are professionally trained in it.

Over time, various changes were made to the rules of the game. Multiple variants of checkers began to appear, such as [2]:

- Blue and Gray,
- Cheskers,
- Damath,
- Dameo,
- Hexdame,

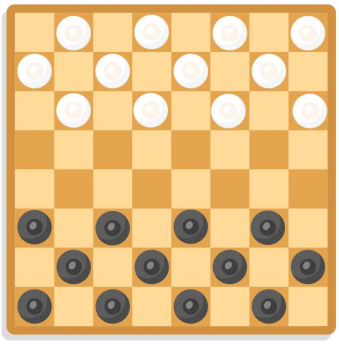


Figure 2.2: Starting position for English draughts on an 8×8 checkerboard

- Lasca,
- Philosophy shogi checkers,
- Suicide checkers,
- Tiers,
- Vigman's draughts.

Now there are many variations of the game of checkers, and even checkers for three have appeared [7].

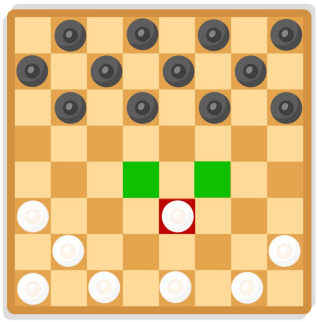
2.2 General rules

This project uses English draughts [5], which differ in rules from international checkers. The rules specific to this type are described below.

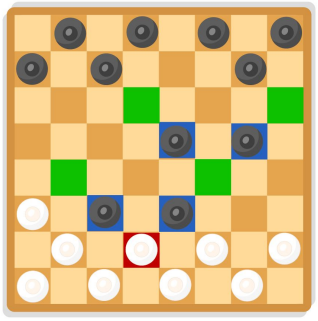
Game Composition

1. Game board consist of 64 (8x8) cells (Fig. 2.2) of two contrasting colors, usually white and dark (gray, brown, red), located diagonally.

2. Checkers pieces of two different colors, and there are 12 of each.



(a) An example of a simple move



(b) An example of a jump over the piece

Figure 2.3: Move and jump rules for uncrowned pieces (men). (Red fields indicate pieces that will be moved in this round, green fields indicate the fields on which selected man is able to move and intermediate fields during the jump, blue fields indicate the fields on which there are enemy pieces that will be captured after a jump)

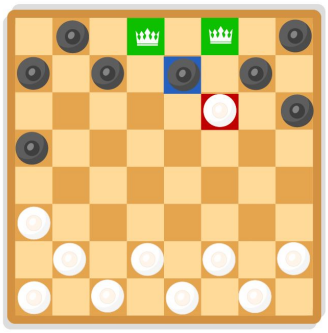
Purpose

Win the game - when the opponent has not a single checker left, the opponent's checkers are blocked, or the opponent recognizes his defeat ahead of time. If no participants in the game can win, the game is considered finished in a draw.

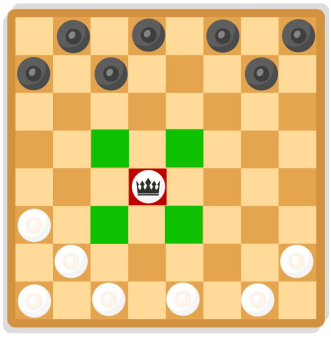
Rules

The game is played by two players. Players are located on opposite sides of the board. The playing field (board) is located in such a way that the corner dark cell is located on the left side of the player. The choice of colour by the players is determined by lot or by agreement. Checkers are placed on three rows closest to the player on dark cells. The right of the first move usually belongs to the player who plays black (dark) checkers. Opponents make moves in turn.

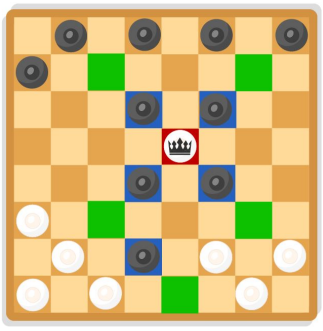
There are two kinds of pieces, simple uncrowned pieces (men) and crowned pieces (kings). At the beginning of the game, all checkers pieces are simple. Uncrowned pieces can be moved forward diagonally to an adjacent free cell (Fig. 2.3a). The capture of an opponent's checker is carried out by jumping one checker over it. It can be done when the opponent checker is located on a diagonal cell adjacent to a man, and there is a free field behind it. Capturing an opponent's checker with a man can only be done forward (Fig. 2.3b). In a single turn, all checkers pieces can capture multiple enemy pieces. If jumps are not in the same line, it calls "zigzag" jump.



(a) An example a jump, after which the man becomes a king



(b) An example of a simple move



(c) An example of a jump over the opponent's pieces

Figure 2.4: Move and jump rules for crownhead pieces (kings)

If a man reaches the last horizontal, it becomes a king and is indicated by a flip (Fig. 2.4a). The king can move one square diagonally forward or backward (Fig. 2.4b). The king, during capture, moves only through one field in any direction and not to any diagonal field, as in Russian or international checkers (Fig. 2.4c). Taking the opponent's checker is obligatory, but if there are several continuations of the "battle", anyone that is most tactically expedient is chosen (the main criterion is the absence of further continuations for captures).

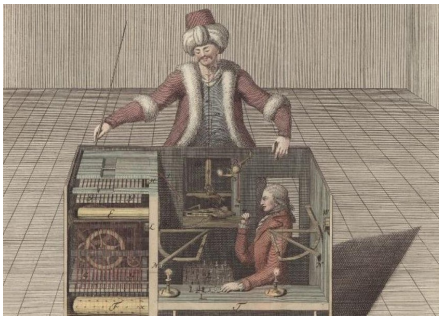
A move is considered made if the player releases his hand after moving the checker. If a player touches a checker, he must make a move with it. If any of the opponents wants to correct the checkers, he is obliged to warn in advance [4].

2.3 First board games engines

When there were no computers yet, people were already thinking about creating a chess machine. There is a famous story about the "Turk", created in the 18th century for Maria Theresa, the Austrian Empress (Fig. 2.5). The device played surprisingly well, but, alas, it turned out to be a fake - a live chess player was sitting inside.

The next step was taken after the Second World War, when one of the best mathematicians of the time, Alan Turing, created an algorithm for teaching a machine to play chess. In 1947 he specified the first chess program.

(a) A reconstruction of the Turk (the original burned down in 1854)



(b) A drawing of how the Turk works - a tiny person sits inside it

Figure 2.5: The Turk - the mechanical chess player [Source: en.wikipedia.org]

Simultaneously with Turing, another mathematician, Claude Shannon, was working on this problem. In 1949-1950, he identified the main problem: the number of continuation options would increase with each move. The researcher singled out two ways to enumerate options:

- first strategy, with an enumeration of all options without exception,
- second strategy, discarding unsuitable options, based on people's chess experience.

Today, there are hundreds of chess and checkers engines, and new ones appear periodically in the list. Tournaments are held regularly to identify the strongest engine. High competition and frequent releases of new versions contribute to the constant growth of the level of the game.

Chapter 3

Background information

An essential stage for artificial intelligence came in 1955 - when the term "artificial intelligence" itself appeared. It was invented by the American scientist John McCarthy, and three years later, he created the Lisp programming language, which became the main one in working with AI.

This section describes important and popular AI programs that not only know how to play games (such as chess, checkers, poker and go), but also beat the best players in the world. This proves that machines and algorithms can better cope with complex tasks than people.

3.1 Review of game of checkers algorithms

To begin with, let's look at several world-famous programs that have been created to solve the game of checkers, from the very first IBM 701 to the most famous Chinook [8].

IBM 701

The first artificial intelligence program launched in the United States was the checkers program written in 1952 by Arthur Samuel for a prototype in the IBM 701 [9]. This is the world's first self-learning computer that plays checkers (Fig. 3.1). Samuel chose checkers because of the elementary rules that require a specific strategy.

Figure 3.1: Arthur Samuel playing checkers with an IBM 701 computer, 1959 [Source kordon.org.ua]

The computer was trained on simple game guides that could be bought at the store. They described hundreds of games with good and bad moves. In 1955, Samuel added features that allowed the program to learn from experience. He incorporated both rote learning and generalization mechanisms, refinements that eventually led to his program winning one game against the former Connecticut Checkers Champion of 1962.

Nemesis

Nemesis [10] is an English checkers program by Murray Cash. Today, Nemesis is no longer commercially available; development stopped many years ago.

Nemesis was the strongest program in 2002 when it won the UK Computer Championship against Wyllie, a 16-game match ending +5=11 in favour of Nemesis. This game was played at the World Computer Drafts Championship in Las Vegas.

KingsRow

KingsRow [11] is another powerful engine for American/English Drafts and International Drafts. It was released by Ed Gilbert in 2000. The engine is available as free software. At the only World Computer Drafts Championship, Nemesis took first place and KingsRow took second. In 2005, Ed Gilbert completed the creation of a 10-element endgame database for use with KingsRow.

The best checkers engines like KingsRow are almost unbeatable today. Due to the large opening books, the engines stay in the book long enough for the search function to see the database in the endgame table.

Chinook

A team led by Jonathan Shea, Rob Lake, Paul Lu, Martin Bryant and Norman Treloar developed Chinook, a computer program that plays checkers [12][13], between 1989 and 2007 at TA University. In 1997 Jonathan Schae wrote a book about the Chinook called "One Jump Ahead: Challenging Human Supremacy in Checkers" [14]. An updated version of the book was published in November 2008. Program performance include four aspects [14]:

- search algorithm,
- evaluation function,
- endgame databases,
- opening book.

Chinook uses a parallel Alpha-beta search algorithm and has average minimum search depth of 19 layers. Some research in chess has shown that there is a linear relationship between search depth and performance. But there comes a time when increasing the search depth leads to a decrease in performance. The data suggest that Chinook has already reached the point where deeper searches are of little avail. The cumulative effect of profound selective search results in the Chinook algorithm does not decrease the occurrence of tactical mistakes.

Good stroke evaluation function is determining how good a position is. The evaluation function consists of two parts. Firstly, the concept of a piece account (material balance) is hard-coded in the program. Secondly, it includes an evolutionary-computational evaluation.

Endgame databases are a set of databases of end games for all positions, complete information about which endgame positions are won, lost and drawn. From 1992 the program has access to a database containing the value (win, lose, draw) for all positions with seven or fewer pieces on the board, as well as a small percentage of databases with eight pieces. The databases contained approximately 40 billion positions compressed into two gigabytes of data.

The opening book is a database of the first lines to start the game. In the beginning, a small collection of opening positions should have enriched Chinook's knowledge. This type of lore is often referred to as an introductory book in reference to a similar resource that a human player would use. If Chinook had encountered one of these positions in the game, it would have made

Figure 3.2: Tinsley (right) and Schaeffer (left) squaring off for the World Checkers Championship, 1990 [Source: 1.bp.blogspot.com]

a move suggested in the opening book, regardless of what the Chinook thought about it. The Chinook team set themselves much higher goals and ended up creating a big opening book and huge endgame databases.

Chinook is the first computer program to win a world title against a human. In 1990, it qualified to play in the human World Championship, behind Marion Tinsley in the USA (Fig. 3.2). At first, the American Checker Federation (ACF)[15], and the English Drafts Association (EDA) were against the participation of a computer in the human championship. When Tinsley relinquished his title in protest, the ACF and EDA [16] created a new title, Man vs. Machine World Championship, and Marion Tinsley won by four wins to two against Chinook. In 1995, the Chinook team defended their Man-Machine title in a 32-game match against Don Lafferty. The final score was 1-0 for Chinook over the Lafferty.

The Chinook website [17] contains materials describing the operation of the program and an article submitted to the journal Science. Jaap van den Herik, the editor of the International Computer Games Journal, called the achievement very significant in the field of artificial intelligence.

Figure 3.3: Match Garry Kasparov and Deep Blue, 1997 [Source: kordon.org.ua]

Figure 3.4: Ke Jie playing with AlphaGo, 2017 [Source: kordon.org.ua]

3.2 Review of board games algorithms

Artificial intelligence algorithms began to be applied not only to checkers but also to other games with more complex rules and decisions. Below are some of the most outstanding programs that have solved games such as chess and Go.

Chess and Deep Blue

In 1985, Carnegie Mellon University began developing the ChipTest, a chess computer. In 1988, IBM joined the project, and the prototype was renamed Deep Thought [18]. A year later, they decided to test the program in action and invited Garry Kasparov, who easily won both games.

Seven years later, IBM introduced Deep Thought II, which was later named Deep Blue, referring to the company's nickname, Big Blue. A year later, the first match between Kasparov and an improved computer took place. The man won again: Kasparov won three times in six games and lost once; two matches ended in a draw.

In May 1997, a greatly improved Deep Blue scored two wins in the return match, lost once and drew three, becoming the first computer to beat a reigning the World Chess Championship (Fig. 3.3). In the early 2000s, computers consistently beat world champions, and checkers and chess were the first games that humans lost to computers.

Go and AlphaGo

The first computers that could really compete with humans appeared only in this decade. In 2014, Google DeepMind introduced the AlphaGo algorithm [19][20][21], which competed with people on an equal footing for two years but won its first significant victory only in October 2015, defeating the European champion.

In 2019 Max Pumperla and Kevin Ferguson published a book called "Deep Learning and the Game of Go" [19]. AlphaGo uses various search algorithms, such as Minimax, Alpha-beta, Monte Carlo tree search (MCTS), as well as neural networks. The result is a convolutional NN with two outputs combined with their MCTS algorithm. One output indicates which moves are essential, and the other output indicates which player is ahead. The AlphaGo Zero tree search algorithm is similar to MCTS with two significant differences. Instead of using random games to estimate position, it relies solely on a neural network. Also, it uses a NN to find new branches.

A year later, on the popular Asian server Tygem, where world champions also play, a user appeared under the nickname Master. In a few days, he played 60 matches and never lost, which caused outrage and suspicion of foul play. On January 4, 2017, Google disclosed that an improved version of AlphaGo had been hiding under the nickname all this time.

In May 2017, AlphaGo - the same one that became famous on the network under the nickname Master - fought Ke Jie, the first Go player in the world ranking, and won three out of three matches (Fig. 3.4). In October, Google DeepMind released a version that was more powerful than Master. AlphaGo Zero was self-learning without human intervention at all, just endlessly playing with itself. After 21 days, it reached the Master level, and after 40 days, it was already better than all previous versions.

Chapter 4

Theory

Back in the late 1920s, John von Neumann established the main problem of game theory, which remains relevant to this day: "Players play a given game. What moves should the players play in order to achieve the best possible outcome?".

Soon after, problems of this kind evolved into a challenge of great importance for developing one of the most popular areas of computer science today - artificial intelligence. Some of the most outstanding achievements in the field of artificial intelligence have been reached in the field of strategy games - the world champions in various strategy games have already been beaten by computers.

Although these programs are very successful, their way of making decisions is very different from the human one. Most of these programs are based on efficient search algorithms, and more recently, machine learning. This section describes the algorithms that were applied in this project.

4.1 Search algorithms

First, let's look at search algorithms and their associated terms.

The best way to describe the rules of many board games is to use a tree graph, with nodes as legal positions and edges as legal moves. The graph is oriented, so we cannot return exactly

where we came from in the previous step. For example, in checkers, a piece can only move forward. This graph is called a game tree. Moving down the game tree means that one of the players makes a move, and the game state changes from one legal position to another.

In this project, I have implemented selected tree search algorithms such as Minimax, Alpha-beta pruning and Monte Carlo tree search.

4.1.1 Random moves

The first algorithm is just choosing the next move randomly from a list of possible actions. This is not the best algorithm, but in this situation, the selected move will always be possible and generated quickly enough. An algorithm like this is usually not considered to be the strongest opponent, but thanks to it, it is possible to find different positions in the game. This is good in cases when one needs rich and varied data.

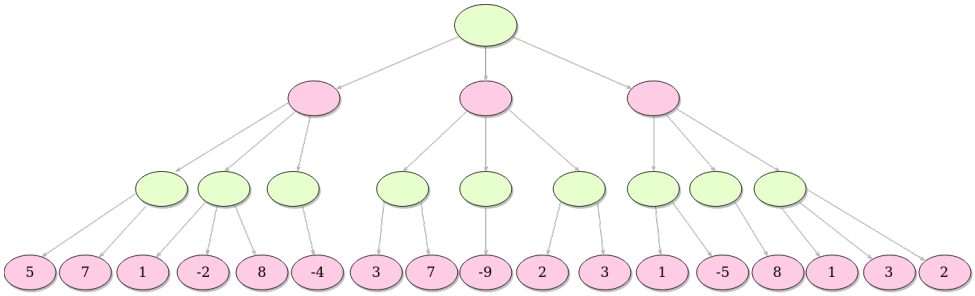
4.1.2 Minimax

The Minimax algorithm [22] relies on systematic search, or more precisely, on brute force and a simple evaluation function. Each time, when deciding the next move, the algorithm searches the entire tree, up to the leaves. In fact, the algorithm considers all possible outcomes and each time determines the best possible move.

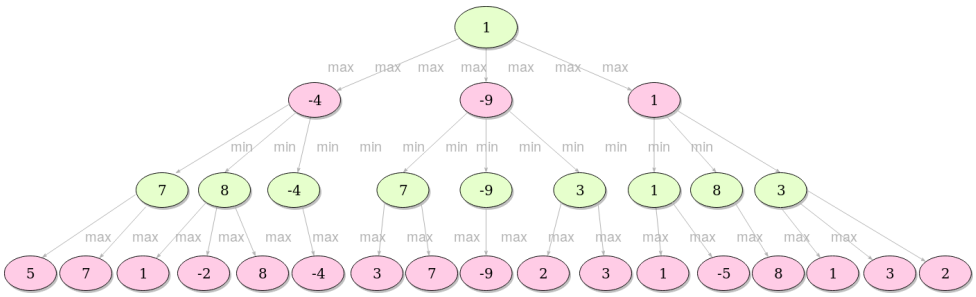
However, this practice is inapplicable for non-trivial games. Since searching to a certain depth sometimes takes an unacceptable amount of time. Therefore, Minimax applies the search to a reasonably low tree depth with appropriate heuristics and a well-designed but simple scoring function.

In order to determine a good (not necessarily the best) move for a certain player, the algorithm must somehow evaluate the nodes (positions) to compare one with another in quality. This requires an evaluation function - a static number, which, following the game's characteristics, is assigned to each node.

Typically, for a Minimax, the leaves are assigned the maximum value of all children nodes when the algorithm chooses the path that brings the most profit for the player, and minimum value, when we choose the path that brings the greatest loss for the opponent.



(a) Leaf evaluation



(b) Selecting the best move for the green player using depth 3

Figure 4.1: Illustration of Minimax algorithm steps. In this case, algorithm is looking for the maximum value for main node [Source: pythobyte.com]

A common practice is to change the leaf scores by subtracting the depth of that exact leaf so that, of all the moves leading to a win, the algorithm can choose the one that does it in the least number of steps (or choose the action that delays losing if it is unavoidable).

In Figure 4.1 there is a simple illustration of the Minimax steps. In this case, the algorithm looks for the maximum value for the main node. The green layer calls the maximizer method that finds maximum value on the child nodes, and the pink layer calls the minimizer method that finds minimum value on the child nodes. The idea is to find the best possible move for a given node, depth, and evaluation function.

In this example, the green player looks for positive values, while the pink player looks for negative values. The algorithm first evaluates only nodes at a given depth, and the rest of the procedure is recursive [23]. The values of the remaining nodes are the maximum values of their respective children if it is the green player's turn, or similarly the minimum value if it is the pink player's turn. The value at each node represents the next best move based on the given information.

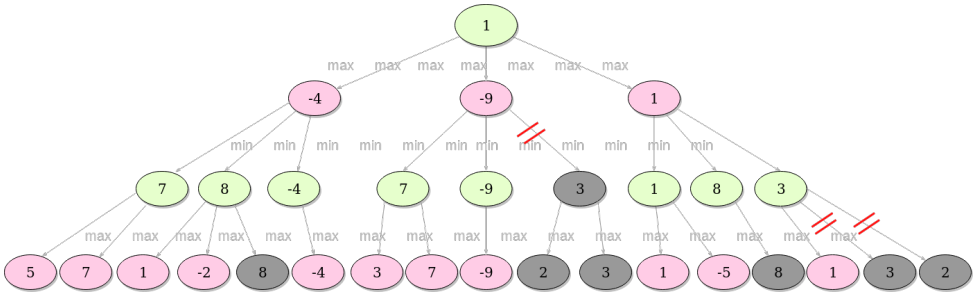


Figure 4.2: Illustration of Alpha-beta algorithm. [Source: pythobyte.com]

When searching in the game tree, we only look at nodes at a fixed (given) depth, not those before or after. This phenomenon is often referred to as the horizon effect [24].

4.1.3 Alpha-beta pruning

The Alpha-beta ($\alpha - \beta$) algorithm [25] was developed independently by several studies in the mid-1900s. Alpha-beta is actually an improved version of Minimax using heuristics. It stops evaluating a move when convinced that it is worse than the previously considered move. Such steps do not need further evaluation.

When added to a simple Minimax algorithm, it gives the same result but cuts off certain branches that cannot affect the final solution - drastically improving performance. The basic concept is to maintain two values throughout the search:

- Alpha (α) - the best option for layer where the algorithm is looking for maximum value,
- Beta (β) - the best option for layer where the algorithm is looking for minimum value.

Initially, alpha is negative infinity, and beta is positive infinity, meaning the algorithm uses the worst possible scores for both players.

Figure 4.2 shows an example of how the previous tree will look if the Alpha-beta method is applied.

When the search reaches the first grey area (8), it will examine the current best (with the minimum value) already explored option along the path for the minimizer, which is currently 7. Since 8 is greater than 7, it is possible to cut off all further children of the node where we are

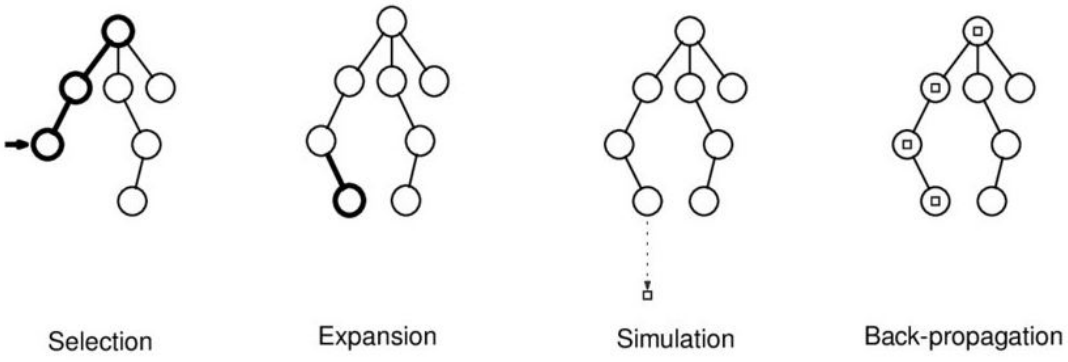


Figure 4.3: Illustration of MCTS algorithm. 1) Selection - Tree traversed using tree policy. 2) Expansion - New node added to the tree (selecting using the tree policy). 3) Simulation - roll-outs are played from new node using default policy. 4) Bach-propagation - Final state value is backpropagated to parent nodes [Source: www.researchgate.net]

(in this case, they are not). If we play this move, the opponent will play a move with a value of 8, which is worse for the player than any possible move.

The best example might be when it comes to the next grey. Pay attention to the nodes with the value -9. At this point, the best (maximum value) explored along the path for the maximizer is -4. Since -9 is less than -4, all other node children can be cut off.

4.1.4 Monte Carlo tree search

The Monte Carlo method [26] is a decision algorithm often used in games as the basis of artificial intelligence. This method uses randomness to solve theoretical problems that are difficult or impossible to solve with other approaches.

Monte Carlo tree search in games is based on many playouts, also called roll-outs. In each playout, the game is played to the very end, choosing moves at random. The final game result of each playout is then used to weigh the nodes in the game tree so that the best nodes are more likely to be selected in future playouts. Each Monte Carlo tree search round consists of four steps (Fig. 4.3):

1. Selection: This process is used to select the tree node with the highest probability of winning. It starts at the root (current state of the game) and selects successive child nodes

until a node with a potential child from which simulation has not yet been initiated is reached. The root is the current state of the game.

2. Expansion: If the game does not end (e.g. win/lose/draw) for any player, create one (or more) child nodes and select a node from one of them. Child nodes are any valid moves from a game position.

3. Simulation: Full one random playout from the node. The replay can be as simple as choosing uniform random moves until the end of the game.

4. Back-propagation: Updating information in nodes on the path to root depending on whether the game was won / lost or ended in a draw.

When all roll-outs have been played, MCTS returns the best move in the root node to be played in the actual game. The best action can be the one with the highest average score or the most visits.

Each roll-out improves the evaluation of a single possible move. But roll-outs are a limited resource. A limited budget allocation is usually used as a standard strategy called the Upper Confidence Bounds for Trees (UCT) method. The formula allows you to select the node that maximizes the estimated reward for each move. This formula creates a balance between two goals:

1. Exploitation spends time looking at the best moves. In this case, the algorithm wants to exploit any advantage that MCTS has discovered so far. The algorithm would spend more roll-outs on the moves with the highest estimated winning percentage.

2. Exploration gives more accurate evaluations for the branches the algorithm has visited the least. By pure chance, MCTS may have a low score for a really good move. After spending a few more roll-outs there, the algorithm can reveal its true quality.

4.2 Machine learning

Machine learning [27] is a form of artificial intelligence that allows a computer to learn without the need for direct programming. Instead of giving specific commands to the computer to complete a task, ML enables the computer to develop its own solution steps using the data for self-learning. The more data a computer has access to, the more efficiently it learns and the "smarter" it becomes, improving its accuracy and performance over time.

With the help of machine learning, AI can analyze data, memorize information, make predictions, reproduce ready-made models and choose the most suitable option from the proposed ones. Machine learning is a division of Data Science dedicated to building intelligent models. Such models can be used to predict the purchase of a product by a user, recommendations in social networks, image recognition, and so on. Machine learning is at the heart of many technologies we use today, including online search, natural language recognition and processing by personal voice assistants (such as Alexa and Siri), marketing personalization, Netflix user recommendations, and many others. ML also enables innovation in self-driving cars, as well as developments in artificial intelligence in robotics.

4.2.1 Deep learning

Deep learning [28] is a set of machine learning algorithms that train neural networks at several levels of abstraction. Deep neural networks (DNN) require huge amounts of data and technical resources. That underlies machine translation, chatbots and voice assistants, music generation and deepfakes, process photos and videos.

Deep learning allows us to improve the process of teaching machines new things. Using rules-based AI and ML technologies, the data scientist determines the rules and features of the data set to be included in the models, and this determines how these models will work. In the case of deep learning, data analysts feed raw data into the algorithm. The system then analyzes this data without any specific pre-programmed rules or functions. After the system makes a prediction, the system's accuracy is checked on a separate data set. The level of accuracy of these predictions then determines the next set of predictions made by the system.

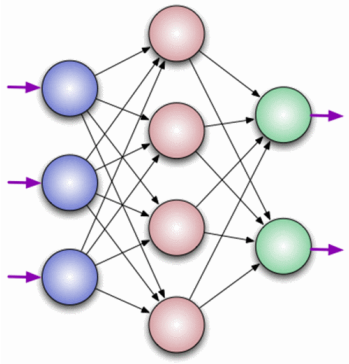


Figure 4.4: Example of a simple neural network with 1 hidden layer of neurons [Source: hsto.org]

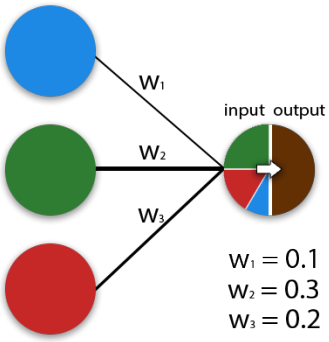


Figure 4.5: Example of transmitting information from three neurons to one [Source: hsto.org]

The word "deep" in the term "deep learning" reflects the number of levels that the neural network gains over time, while the deeper the network becomes, the higher the performance increases. Each layer of the network processes its input in a certain way and then informs the next layer. Therefore, the output from one level becomes the input for the next.

Training deep neural networks is time-consuming and requires large amounts of data to be collected and used in testing as the system gradually improves its model. Neural networks appeared in the 1950s, but it is only very recently that computing power and data storage capacities have reached the point where DL algorithms can be used to create amazing new technologies. For example, deep neural networks allow us to perform computer tasks such as speech recognition, develop bio-informatics and analyze medical images.

Deep neural networks

A neural network [29] is a sequence of neurons connected by synapses. The structure of the neural network came to the world of programming straight from biology. Thanks to this structure, the machine learned to analyse and memorise many information states. Neural networks are also capable of analysing incoming information and reproducing it from their memory. In other words, a neural network is a machine interpretation of the human brain, which contains millions of neurons that transmit information in the form of electrical impulses.

A neuron is a unit that receives information, performs simple calculations on it, and passes it on. They are divided into three main types (Fig. 4.4): input (blue), hidden (red) and output

(green). In the case where the neural network consists of a large number of neurons, the term layer is introduced. Accordingly, there is an input layer that receives information, *n* hidden layers that processes it, and an output layer that displays the result. Usually neurons operate on numbers in the range [0,1] or [-1,1].

A synapse is a connection between two neurons. Synapses have one parameter - weight. Thanks to it, the input information changes when it is transmitted from one neuron to another. Let's say there are 3 neurons that pass information to the next one. Then we have 3 weights corresponding to each of these neurons. For the neuron with the greater weight, that information will be dominant in the next neuron (an example is colour mixing (Fig. 5.9)). During the initialisation of the neural network, the weights are chosen randomly.

Any neural network needs a layer that is deeply connected with its preceding layer which means the neurons of the layer are connected to every neuron of its preceding layer. I use a dense layer [30], which is the most commonly used layer in artificial neural networks.

An activation function is a way to normalise the input. That is, if you have a large number at the input, passing it through the activation function, you will get an output in the range you need. In this project, I'm using the ReLU activation function (Rectified Linear Unit) [31] for hidden layers, the most popular activation function for deep neural networks, and the softmax function [32] in the output layer that turns values into a vector of real values that sum to 1. These values represent the probability distribution. This function allows better training of deep networks compared to the logistic sigmoid [33] and hyperbolic tangent [34] activation functions.

Also an optimiser, loss function and metrics are needed for neural networks to maintain, improve, monitor and maximise performance. I use an optimiser that implements the RMSprop algorithm (root mean square) [35], that is used for full-batch optimisation. It keeps the moving average of the squared gradients for each weight. And then, it divides the gradient by square root the mean square, resulting in faster network learning.

The loss is the mean overseen data of the squared differences between true and predicted values. The loss function measures how good the neural network is with respect to a given training set and expected responses. It may also depend on variables such as weights and biases. In my project I use Mean Squared Error (MSE) loss function [36] and metrics [37].

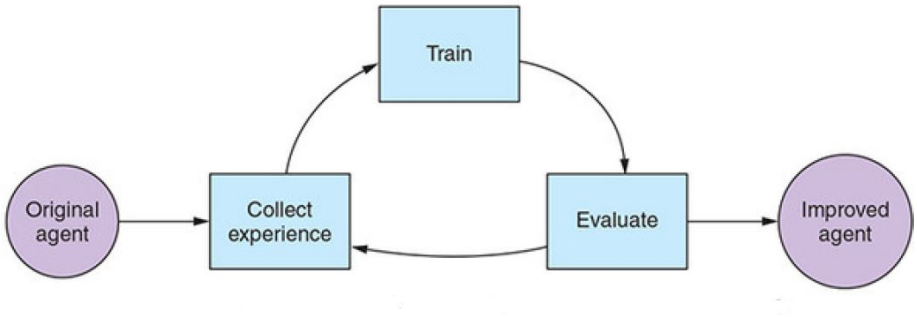


Figure 4.6: General reinforcement-learning cycle [Source: [19]]

MSE is calculated as the average of the squared differences between the predicted and actual values. The result is always positive, regardless of the predicted and actual values sign, and the ideal value is 0.0.

4.2.2 Reinforcement learning

Reinforcement learning (RL) [38] is an independent self-learning system. In order to achieve the best results, the machine learns in a constant practice mode, from which the concept of learning by trial and error follows. Reinforcement learning is similar to deep learning, except for one thing: in the case of reinforcement learning, the machine is trained from its own experience, there is no need to gather database first.

Reinforcement learning uses a method of positive reward for the right action and negative for the wrong one. Thus, the method assigns positive values to desired actions to induce the agent and negative values to undesirable ones. This encourages the agent to seek the long-term and maximum total reward in order to reach the optimal solution.

Reinforcement learning has two main goals. The first goal of the algorithm is to minimize errors. The machine learns to analyze information before each successive move. For example, during training, an unmanned vehicle learns to respond in time to a traffic light, to stop in front of a pedestrian at a crossing, to let a fast moving car or special vehicles pass on the right. To achieve the best result, the car is trained in a virtual city model with random pedestrians and other road users.

The second goal is to get the maximum benefit from the task. In this case, the benefit itself must be programmed in advance: the fastest possible route travel time, the optimal use of enterprise resources, and the service of as many visitors as possible.

Figure 4.6 shows a reinforcement learning cycle in which a computer program improves by repeatedly attempting a task. The cycle consists of 3 phases:

- Collect experience - repeatedly attempt the task in a test environment,
- Train - update the agent's behaviour in response to the experience result,
- Evaluate - compare the agent to a baseline to measure its improvement.

Using deep reinforcement learning (DRL) methods with large neural networks can be less efficient for simple tasks. That's what tabular methods are for. They refer to tasks where the state and action spaces are small enough to be represented as arrays and tables.

4.2.3 Deep reinforcement learning

Some tasks may be too difficult for reinforcement learning. It can be difficult for an algorithm to learn from all states and determine the reward path. Therefore, this is where deep reinforcement learning can help: the "deep" part refers to using a neural network to evaluate states instead of having to map each decision, creating a more manageable decision space in the decision process.

Deep reinforcement learning replaces tabular methods for estimating state values with function approximations. Function approximation not only eliminates the need to store all pairs of states and values in a table, but also allows the agent to generalize state values that it has never seen before, or about which it has partial information, using similar state values. Deep reinforcement learning is used in programs that have beaten some of the best human competitors in games like chess and Go and are also responsible for many advances in robotics.

My project implements the Deep Q-Network (DQN) algorithm, which was developed by DeepMind in 2015. The algorithm was developed by improving the classic RL algorithm called Q-Learning with deep neural networks and a technique called experience replay.

Q-Learning is based on the concept of a Q-function. The Q-function (the state-action value function) measures the expected profit or discounted amount of rewards received from the state by first taking action and then following the policy. The optimal Q-function is defined as the maximum return that can be obtained by starting with observation, taking action, and then following the optimal policy. The optimal Q-function is represented by the following Bellman equation [39]:

$$Q(s, a) = r_t + \gamma * max_a Q(S_{t+1}, a) \tag{4.1}$$

where:

s - current state,

a - chosen action,

$Q(s, a)$ - a Q-value from current state,

r_t - an immediate reward from current state and chosen action,

S_{t+1} - the next state,

$max_a Q(S_{t+1}, a)$ - the maximum possible value from the next state,

γ - a discount factor that controls the contribution of rewards in the future.

But here, it is impossible to deduce the Q-value of new states from already studied states. Because of this, there is a problem of the amount of memory required to save and update data and the amount of time required to study each state. Therefore, the solution is to approximate these Q-values using machine learning models such as a neural network.

In deep Q-learning, the state is given as an input, and the Q-value of all possible actions is generated as an output. The advantage is that all past experiences are stored in the user's memory, the maximum output of the Q-network determines the next action, and the loss function

here is the root mean square error of the predicted quality factor. Going back to the Q-value update equation:

$$L_i(\theta_i) = (y_i - Q(S_t, a_t, \theta_i))^2 \tag{4.2}$$

where:

$$y_i = r_{t+1} + \gamma * max_{a_{t+1}} Q(S_{t+1}, a_{t+1}; \theta_{i-1}) \tag{4.3}$$

$L_i(\theta_i)$ - loss at each step i,

θ - parameters of neural network,

y_i - temporal difference target,

$Q(S_t, a_t, \theta_i)$ - predicted Q-value,

r_{t+1} - the reward from next state,

γ - a discount factor that controls the contribution of rewards in the future,

$max_{a_{t+1}} Q(S_{t+1}, a_{t+1}; \theta_{i-1})$ - the maximum possible value from the next state,

θ_{i-1} - the parameters from the previous iteration.

The algorithm uses a technique called experience replay to make network updates more stable. At each acquisition time step, transitions are added to a circular buffer called the play-back buffer. Then during training, instead of using just the last transition to calculate the loss and its gradient, we estimate them using a mini-batch of transitions fetched from the playback buffer. This has two benefits: improving data efficiency by reusing each transition across many updates and improving stability by having uncorrelated transitions in the batch.

Chapter 5

Practice and experimental results

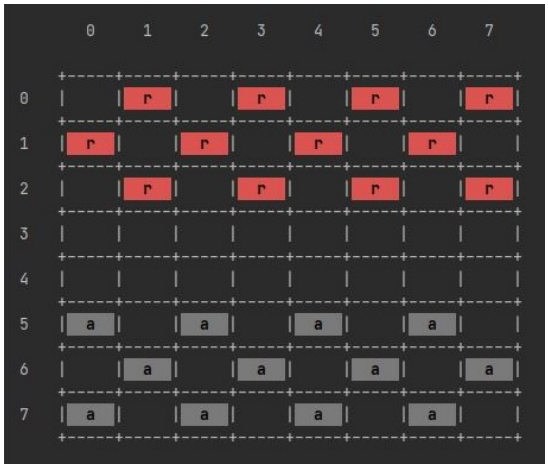
This section describes the algorithms that were developed and implemented in this project. As well as how the data on which the algorithms were learned were created and stored.

5.1 Game state implementation

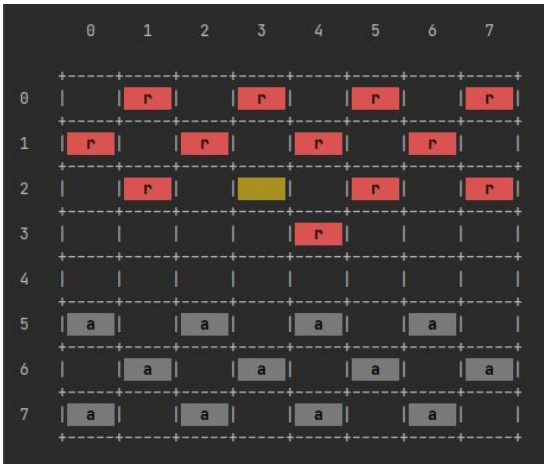
The game state represents the current board state (8x8 array) (Fig. 5.1), the current player who should move (a char letter), and also the counting of how many moves were made without a jump (this is necessary in order to end the game as a draw if at least one of the players has not captured an opponent's piece over the last 15 moves).

5.2 Test data generation

Before training the network, I needed to collect a huge database of various positions and moves. This was accomplished by playing different search algorithms with each other. Basically, these were the Alpha-beta and MCTS algorithms. First, they select the most optimal move from a list of possible actions, which means that I aim to make a database of the best actions for each board state, not just possible moves. Secondly, these two algorithms require less time to sample an action than, for example, the Minimax algorithm. Section 5.5 describes in detail the average move sampling time for each of the implemented algorithms.



(a) Starting board state. Some pieces are red and are denoted by a small letter *r*, others are white noted by a small letter *a*. Accordingly, the kings are capital letters *R* and *A*.



(b) Board position after the piece moved from row 2 and column 3 to the right to diagonal (the yellow square is the position where the piece stood before the move)

Figure 5.1: Screenshots of board state in the console

During the data collection, 2130 games were played: 1000 games the MCTS 600 algorithm, 900 games MCTS 1500 algorithm, and 230 games - Alpha-beta 4 algorithms played with themselves. The number next to the MCTS algorithm indicates the number of roll-outs played before choosing a move, and in the case of the Minimax and Alpha-beta algorithms, the number indicates the depth to which the algorithm descends to select the best action. Each time the algorithm chooses a move, the board and the move are transformed into a network-friendly format and saved to the database. As a result, 127739 checkers boards with different positions were collected, and the same number of moves to each board.

Before data can be saved to files, it must be processed (Fig. 5.2). Firstly take a checkered board, which is an 8x8 array, with player and opponent pieces. All checkers move only diagonally, which means that half of all squares are not used during the game. Therefore, half of the elements in the array can be removed, and only positions where pieces are able to move can be left. The result is a 4x8 array. Then the array is transformed into a 1d array with 1x32 values. Each of these cells is represented in the form of numbers:

– player's king - 4,
– player's man - 3,
– empty cell - 2,

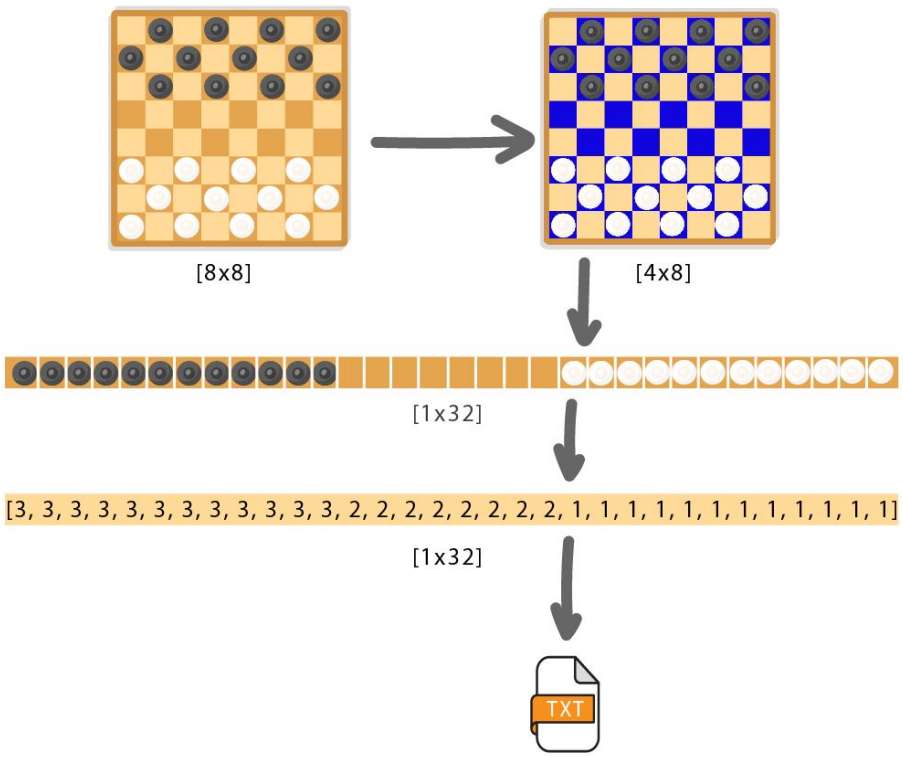


Figure 5.2: Example of transforming the initial board into input vector

– enemy man - 1,
– enemy king - 0.

And after all those transformations, the array is saved to the database.

To train the network, all the recorded data from the files will be read and converted from values from the range [0 - 4] to the range [0 - 1]. This format is more friendly to the network and can be fed as an input.

The moves are stored in the database as two indices: the first is the index of the position from the board array 1x32 on which the piece that will now make a move stands; and the second is the index of the position where this selected piece should move. These indices are from the range [0, 31].

5.3 Deep neural network architectures

In the case of neural networks, it is very important to correctly define the input and output. In the "Deep Reinforcement Learning for Game Playing" project [40], the authors use a neural network for the game of draughts where an input is a vector of the board state and output is the value that represents the evaluation of this state. In turn, in the AlphaZero Chess program [41], the authors describe a network that takes a chess board with figures as an input, and returns 2 arrays. The first array selects a piece for a move, and the second one is responsible for the allowed moves for this piece.

During the project, two different network architectures were tested: one that immediately returns the piece that is going to make the move and its move; and the second one consisting of two separate networks, where one returns the selected piece, and the second one returns the move for that piece.

Single network architecture

The first network has a single network architecture. It takes the board and returns the action. The board state is a 1d array with 32 elements in the range [0, 1]. And the returned action is the indices from the two arrays obtained on the output, where the maximum values lie. These are indices from the board that tell from which to what position the piece should move.

The network structure consists of 5 layers (Fig. 5.3): 1 input layer with 32 neurons; 3 hidden layers (dense layers) with 64, 128, 64 neurons accordingly (these layers have ReLU activation function); and two-headed output layer (Dense layer) with 32 neurons each (with the softmax activation function). This model was compiled with optimizer, loss function and metrics, which are described in section 4.2.1. The model was trained on 50 epochs and with 32 batch size.

Using network in the game

Figure 5.4 illustrates how neural network is incorporated into the game:

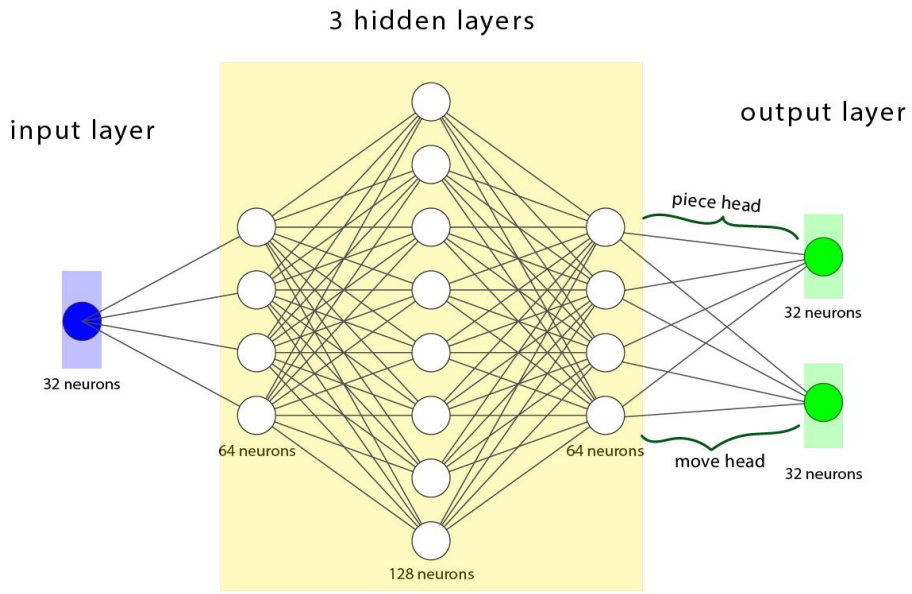


Figure 5.3: Single network architecture diagram

– At the beginning, there is a current board state with 8x8 size. The first step is to encode board data (transform the board into a format that the network can accept). Therefore, the board is transformed to 1x32 array.
– The next step is to feed this array into DNN and predict the next move in the form of 2 arrays. One array is responsible for the piece the network wants to move, and the second array shows which square the piece will need to move to.
– The last step is to take the piece on the board from the position we get from the first output of the network and move it to the position we get from the second output array. These positions are selected based on the largest value in the array. If these positions are not possible for a action, the next one after it is selected.

Results

After playing 50 games against different search algorithms, in the beginning of the game, for the first 15 - 20 moves, the network selects and sends the correct move, but later (when the checkers on the board are in a more chaotic order), the chosen pieces and moves did not correspond at all. The network could choose a piece on one side of the board and a move on the other side. Statistics show that the average ratio of correct to incorrect moves becomes 9.8214%. Therefore, it was decided to change the network structure so that the choice of move would depend on the piece chosen by the network.

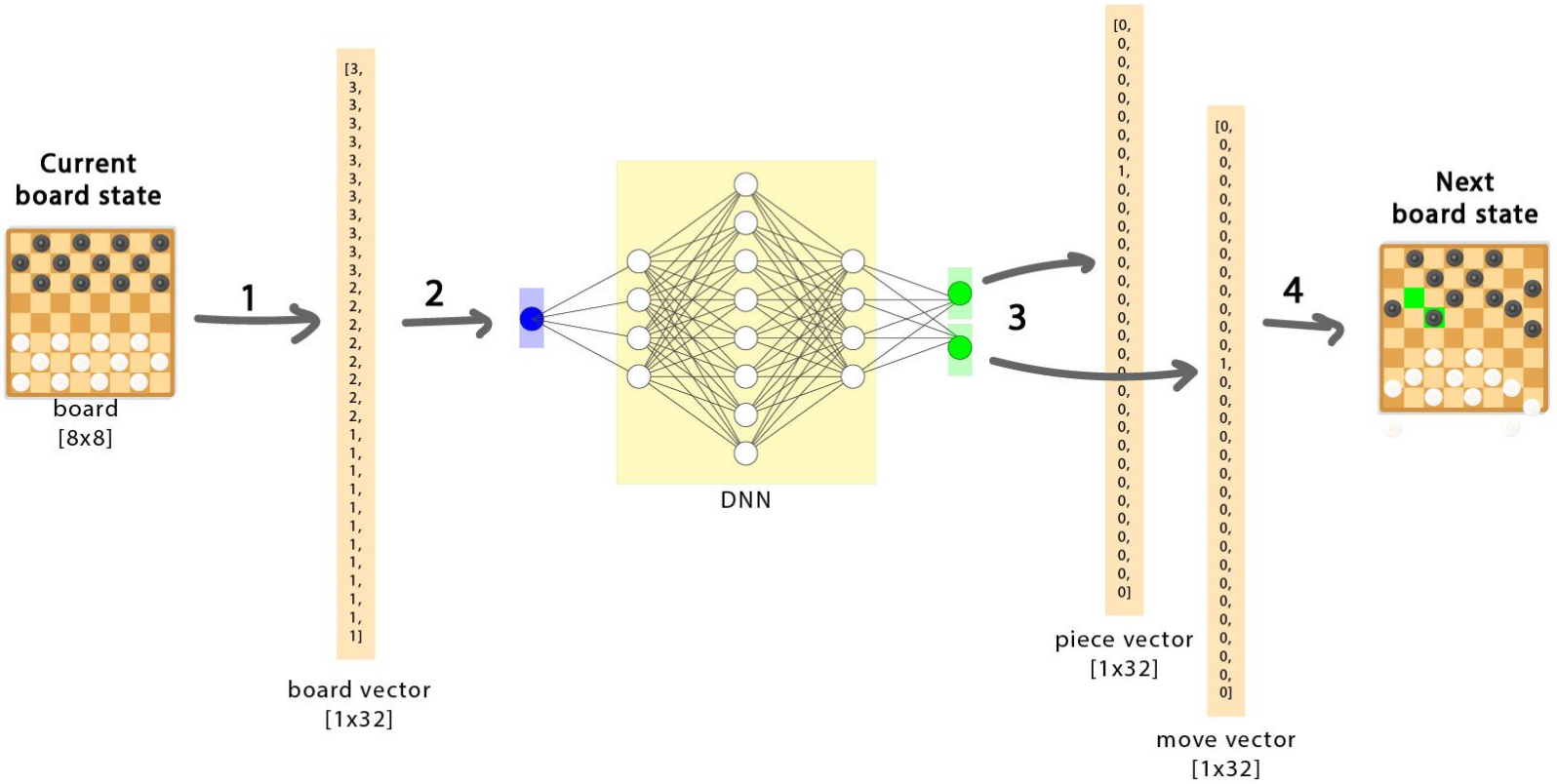


Figure 5.4: Prediction the next move in draughts by using deep learning (1 - encode board data; 2 - feed into network; 3 - predict next piece and its next position; 4 - apply move to board)

Double network architecture

The structure of the second network is similar to the single network. The only difference is that the single network was split into two. The first network, "piece network" (Fig. 5.5a), is a network that has one input and one output layer, with 32 neurons each. The network takes the board and returns the position on which the chosen piece is located. And the second one, "move network", (Fig. 5.5b) has 2 inputs and 1 output layer. The board and previously chosen piece are input, and the position, on which the chosen piece is placed, is an output.

Using network in the game

Figure 5.6 illustrates how the game is connected to a double network and gets the next move from it:

1. The first step is to encode board data to a 1x32 array.

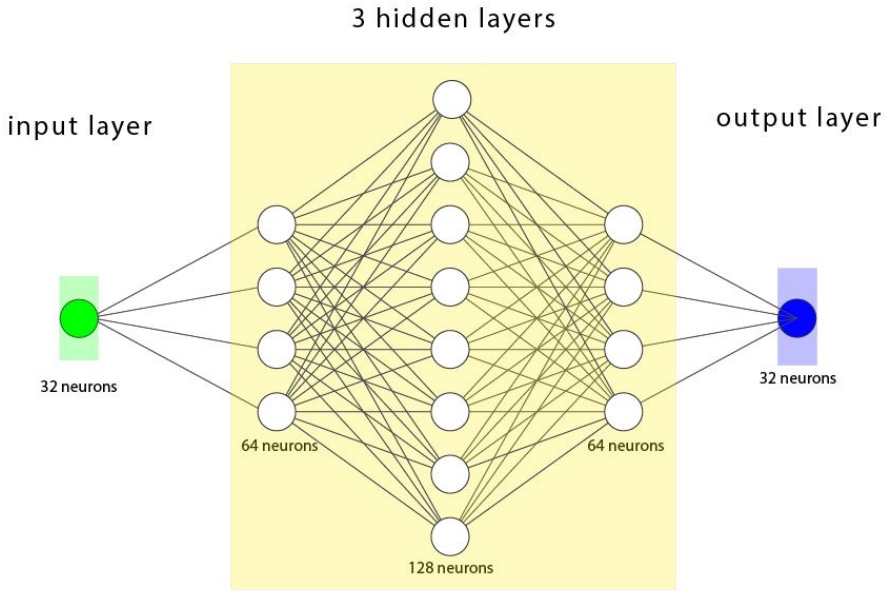
2. The second step is to feed this array into "piece network" and get the prediction of the piece that the network wants to move.

3. In the next step, the board and the chosen piece are passed to "move network". The network returns the position the piece will need to move to.

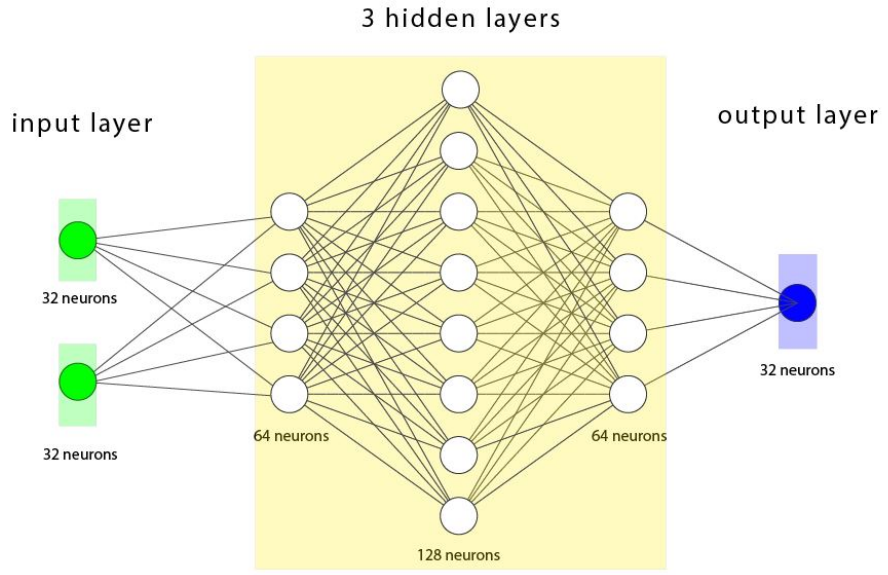
4. Finally, the piece chosen by the "piece network" is moved into the position received from the "move network".

Results

After playing 100 games, the results were more encouraging than the previous one. Each position to which a piece needs to be moved is closely related to it. Yet the networks were not perfect and sometimes made mistakes. Statistics show that the maximum ratio of correct moves to incorrect moves becomes 73.3333%, but the average ratio was 34.4285%. Figure 5.7 shows and describes the types of errors that "piece network" makes when choosing a piece. The most popular mistakes are missed jump opportunities, as well as the choice of a piece which cannot move. Figure 5.8 describes the kinds of mistakes that "move network" makes when selecting a position where chosen piece will be moved. In this case, the most common mistakes were the choice of a move to an occupied cell and the omission of beating the opponent.



(a) "piece network" diagram



(b) "move network" diagram

Figure 5.5: Double network architecture diagram

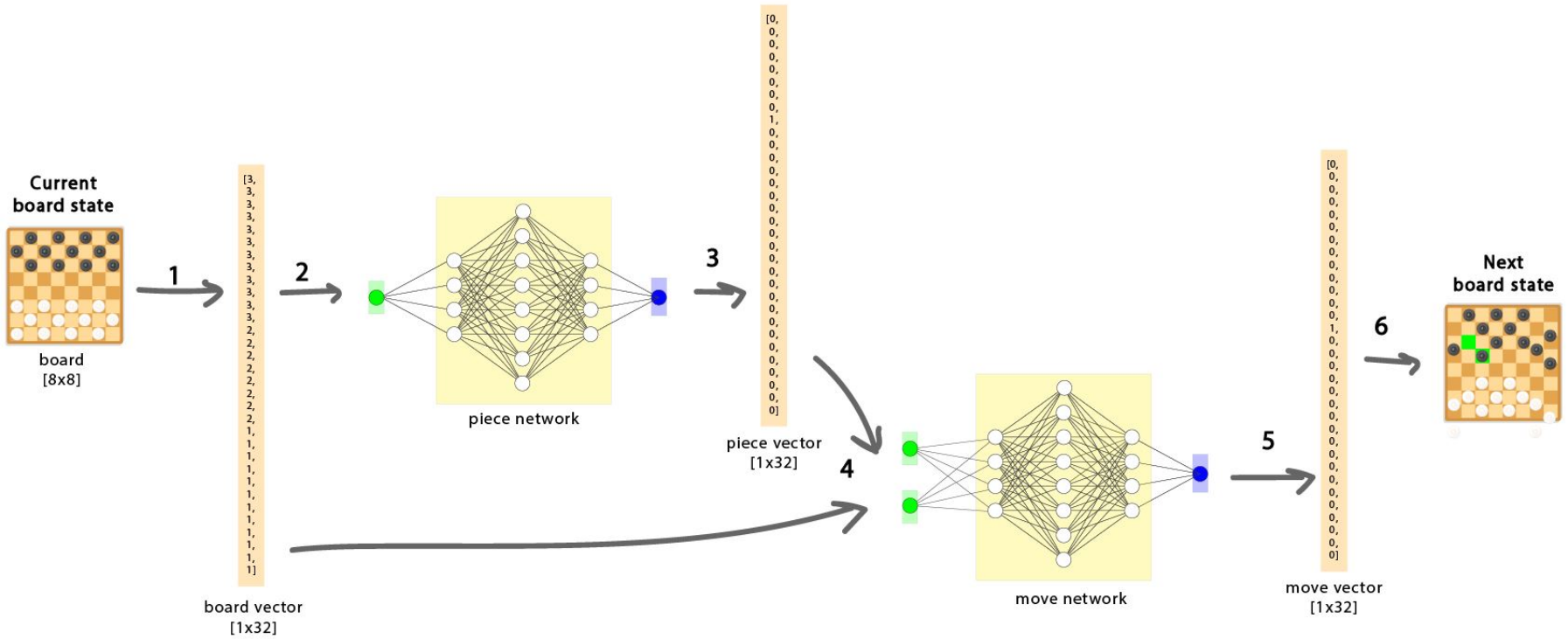
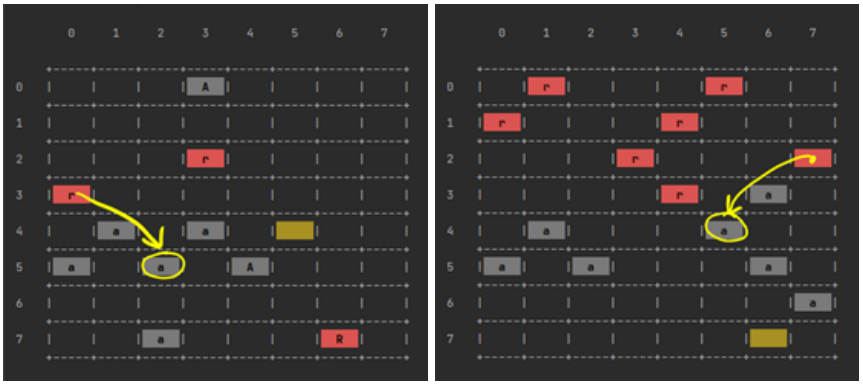
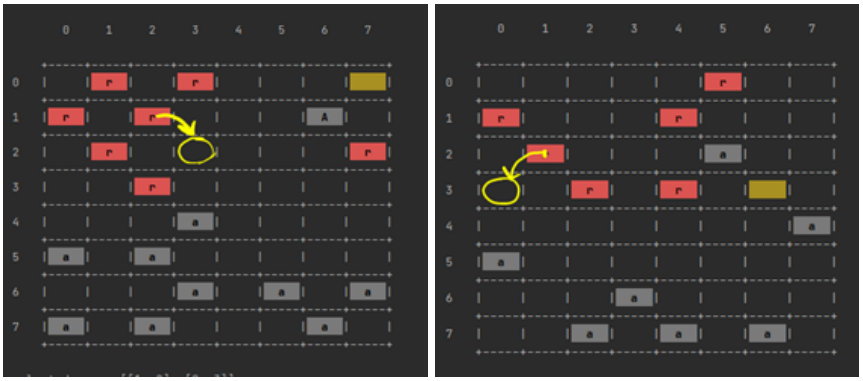


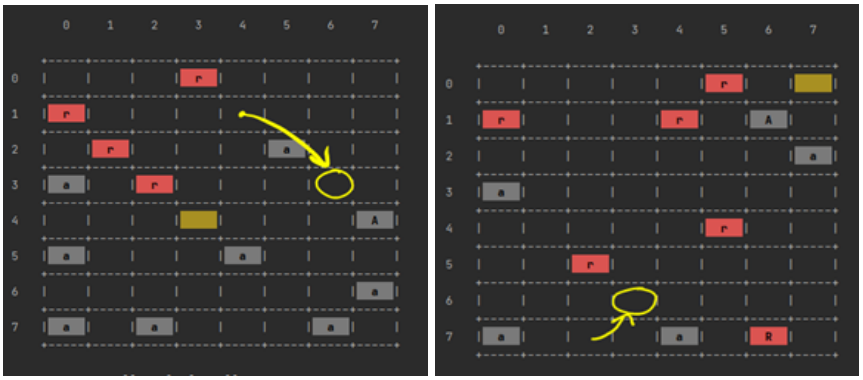
Figure 5.6: Prediction the next move in draughts by using double network (1 - encode board data; 2 - feed into "piece network"; 3 - predict next piece; 4 - feed into "move network"; 5 - predict next position for the piece; 6 - apply action to board)



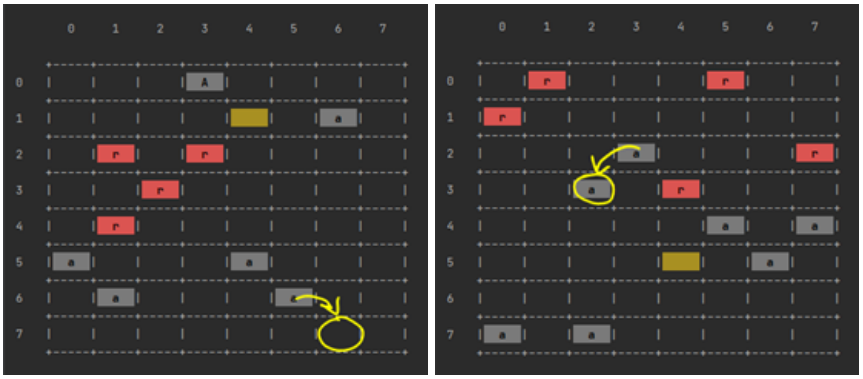
(a) There is no possible move for the selected piece



(b) Jump opportunity not noticed

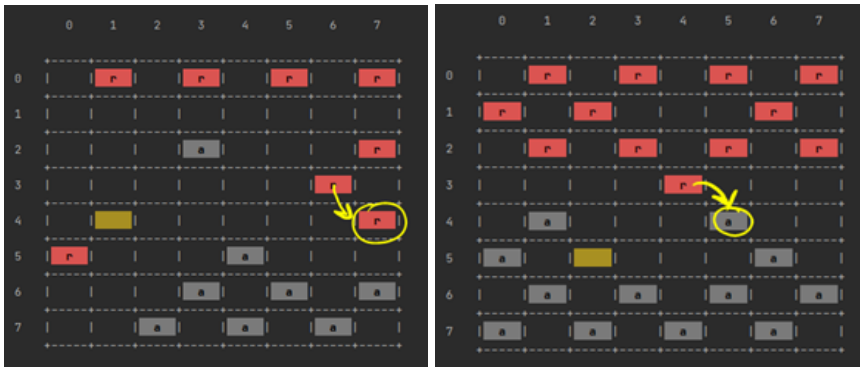


(c) Chosen field does not contain a piece

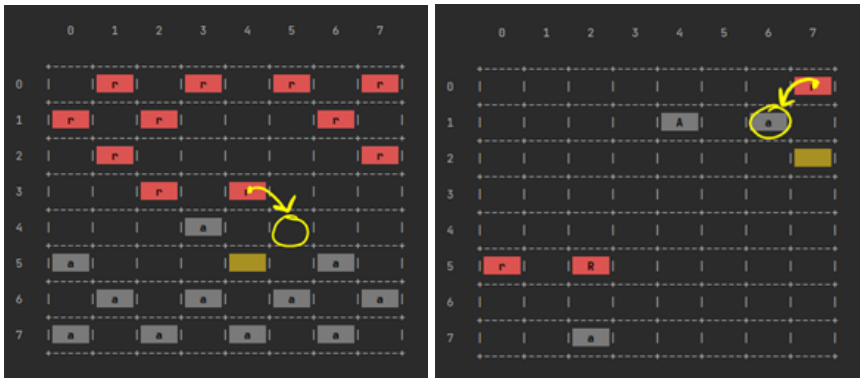


(d) The selected piece is not a current player's piece

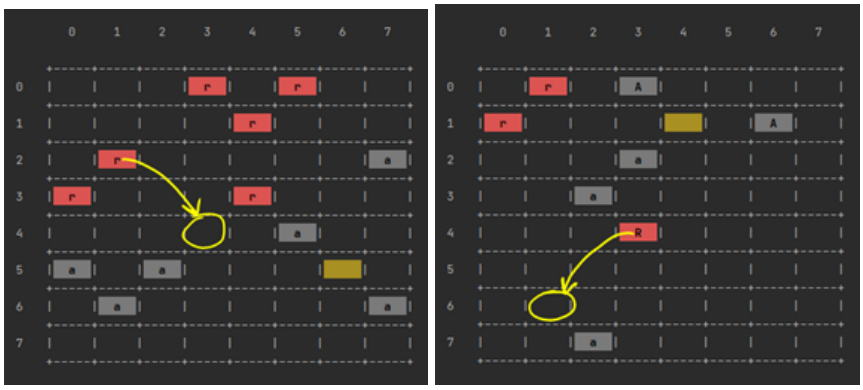
Figure 5.7: Examples of "piece network" errors



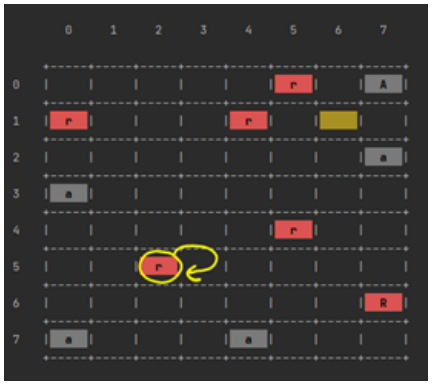
(a) Cannot move to where there is another piece



(b) Jump opportunity not noticed



(c) Jump is possible only through the opponent's piece



(d) Cannot move to the same place where a selected piece was

Figure 5.8: Examples of "move network" errors

5.4 Deep Q-network

The network architecture for this algorithm looks identical to the double network, i.e. two networks are connected together, the network will learn how to play draughts by itself from scratch, exclusively on its good and bad predictions.

The generalized network training scheme is as follows: first, the network plays one set of checkers with itself and, at the same time, records good and bad actions in the database. During the game each board state is sent to the network to get a predicted value array. When predictions are received, the index of the largest value from the array is taken and is checked for the ability to exist as possible move. If this move is not possible, i.e. it is erroneous, the index of the next largest value from the prediction array is taken accordingly. And so on, until the algorithm receives from the network a action that is possible for a given board state. Each such wrong move is recorded in the database of bad actions. After receiving a move that is possible to make, it is also written to the database of possible (good) actions.

Since the network at the beginning does not know how to play draughts at all, in order to diversify the databases, a random action selection is used. In this case, one of the possible moves is selected without the participation of the network using the Epsilon-Greedy method. This method aims to balance exploration and exploitation so that with a poorly trained network, the algorithm will try to discover new variations of the game. But over time, we want to use what the network has already learned to get the best known results. In my algorithm, at the beginning of the game, the ratio between exploration and exploitation is set to 50% and 50% respectively. And after 500,000 cycles, the ratio of exploitation begins to increase to 80%. After the end of this game, the results of the game are recorded in the databases (won, lost, or a draw).

In my project, there are three databases with a certain limit for storing new records. Two of them are databases for illegal moves (separately for "piece network" and "move network"), for which the limit is 10^5 records. And the third is the database of good (possible) actions with a limit of 10^4 records. Once the database is full, the very first records with the oldest data are deleted before the new actions are stored in it.

The second cycle step is to train the network based on the databases (on one epoch and with 32 batch-size). The training consists of 2 parts. The first is learning on the wrong moves:

the algorithm randomly selects 2^{12} records from the database and get results for them from the network. Further, value -1 is entered under the indices of bad actions in the prediction array, and after the changes, this array is fed as a network expected value to train it.

The second part is learning on good steps. The algorithm randomly selects 2^9 records from the database and gets results for them from the network. Further, the algorithm calculates a value that will be entered under the indices of good actions in the prediction array. This value is calculated according to Equation 4.3.

And after all changes have been made, this changed prediction array is passed as network output to train it. The first and second steps are done for both "piece network" and "move network".

The third step in the cycle is to check how well the network has learned and whether it is making progress. I check this by getting the ratio of the number of good to wrong actions from a single game. Over time, this ratio grows, and the network chooses fewer bad moves and learns not just to play checkers, but to play in such a way as to win. This cycle is repeated about 10^6 times.

The algorithm was supposed to learn on 10^6 cycles, but after $300 * 10^3$ cycles, the ratio of correct moves to incorrect moves began to decrease (Fig. 5.9), so training was suspended at $530 * 10^3$ cycle. The average ratio was 9.459%. This may mean that the network structure is too weak to learn all the board states. The solution to this problem may be to increase the number of layers in the network and / or increase the number of neurons in the hidden layers.

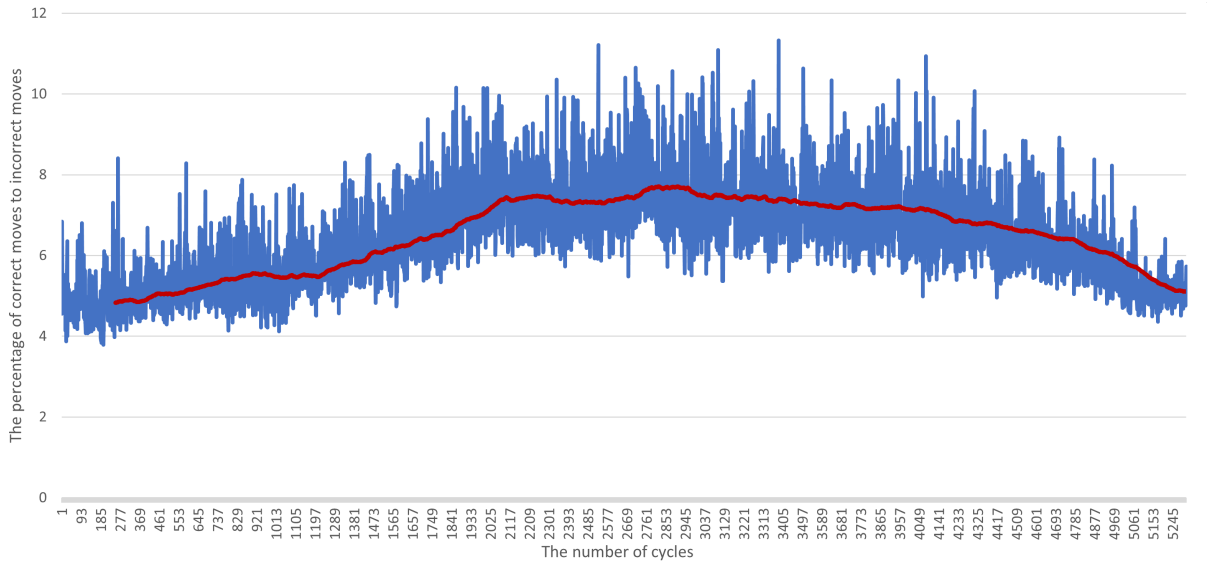


Figure 5.9: The ratio of correct moves to incorrect moves from test games that are played every 10 cycles

5.5 Comparison of algorithms with networks

All algorithms and networks were tested both in terms of their action return speed and in terms of performance, that is, how well they choose a action to win the game.

Table **??** describes the ratio of won, lost and tied games of each algorithm to all others. Vertically are the names of algorithms that played for the player who moves first, and accordingly, horizontally are the algorithms that play as the second player. Here we compare algorithms such as:

- random moves (Rand),
- Minimax (MM) with depth 3 and 6,
- Alpha-beta (AB) with depth 3 and 6,
- Monte Carlo tree search algorithm (MCTS) with 600 and 1800 roll-outs,
- deep neural network with double architecture (NN 1),
- and deep Q-network (NN 2).

The data in the table are presented as three numbers "$w : d : l$": the number of games w - won, d - drawn and l - lost for the algorithms that started the game. All data has been tested based on 100 games for more accurate values.

The fastest but weakest is the algorithm of random choice. The average time for it takes only $7.2 * 10^{-5}$ seconds. But at the same time, it loses to all algorithms. Sometimes it is possible that choosing moves randomly, one on occasion can draw or even win the games, but these are individual cases, the probability of which is less than 1%.

The next are the Minimax and Alpha-beta pruning algorithms. If we consider the average sampling time of choosing a move shown in the graph (Fig. 5.10), we can see that up to depth 5, the average time is quite small (usually less than 1 second). But starting from depth 6 and above, the time grows exponentially for both algorithms. Here the difference and advantage of the Alpha-beta algorithm is clearly visible, which, by cutting off some branches, reduces the step selection time. If we consider these two algorithms from the side of their performance, we see that even at the same depth, the Minimax algorithm is inferior and wins fewer games than Alpha-beta.

The MCTS algorithm showed the best results compared to all the others. Figure 5.11 shows a plot of the average time of action for the Monte Carlo tree search algorithm with roll-outs from 100 to 2000. In this case, the time depends solely on the number of playouts, therefore it grows linearly (usually about 0.35 second for every 100 roll-outs). Average time for MCTS 1800 is two times less then AB 6 and three times less then MM 6. And at the same time, performance is better than all the algorithms together.

The average action selection time of both networks is very similar, and only random move, MM 3 and AB 3 algorithms are faster (Table 5.1). But the networks are definitely better in performance. If we compare a DNN with a DQN (NN 1 and NN 2, respectively in Table **??**), it is clearly seen that both networks have similar results playing one against the other. The first one is able to win against every algorithm except of both versions MCTS. The second one achieved better results against all algorithms except of first neural network.The reason NN 1 is better then NN 2 is the difference in databases. The DNN learned from the best moves. Its database is generated from games, where the players were the MCTS and Alpha-beta algorithms. And the DQN learned from self-playing.

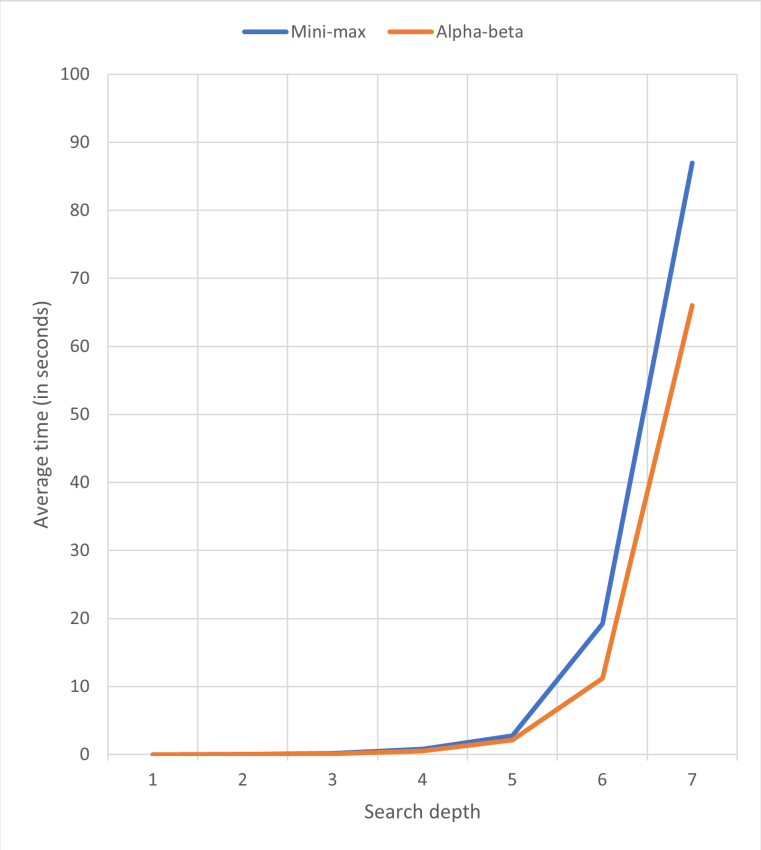


Figure 5.10: Average time of action selection for Minimax and Alpha-beta algorithms

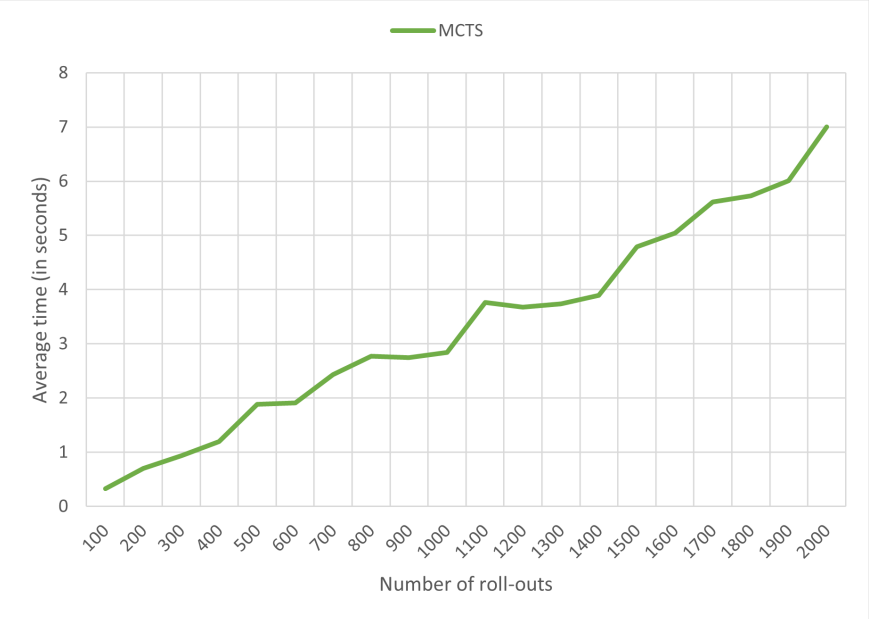


Figure 5.11: Average time of action selection for MCTS algorithm

	Rand	MM 3	MM 6	AB 3	AB 6	MCTS 600	MCTS 1800	Network 1
Network 1	99 : 1 : 0	82 : 14 : 4	60 : 28 : 12	80 : 12 : 8	48 : 30 : 22	46 : 50 : 4	8 : 60 : 32	-
	100 : 0 : 0	80 : 14 : 6	56 : 32 : 12	76 : 16 : 8	42 : 34 : 24	44 : 50 : 6	10 : 56 : 34	
Network 2	98 : 2 : 0	62 : 24 : 14	32 : 40 : 28	64 : 22 : 14	30 : 42 : 28	46 : 36 : 18	32 : 40 : 28	30 : 32 : 38
	99 : 1 : 0	58 : 28 : 14	30 : 40 : 30	60 : 22 : 18	24 : 46 : 30	44 : 40 : 16	32 : 38 : 30	24 : 30 : 44

Table 5.1: Comparison of algorithms of average time of action selection

Algorithm	Rand	MM 3	MM 6	AB 3	AB 6	MCTS 600	MCTS 1800	NN 1	NN 2
Time [s]	$7.2 * 10^{-5}$	0.17	19.24	0.12	11.18	1.91	5.73	0.52	0.63

Chapter 6

Conclusion and further work

The thesis presents an application of deep reinforcement learning algorithm, namely deep Q-Learning, to the game of draughts. In this project, the structure of the network has been proposed, consisting of 2 parts: the first one selects the piece and the second one chooses the move for the piece. The results of the proposed network were compared not only with the deep neural network but also with standard algorithms (random moves, Minimax, Alpha-beta and MCTS algorithms). Even though both networks could not learn how to choose the right moves only, they were able to win against other algorithms.

Further work can be focused on improving the network structure by increasing the number of layers and neurons and again letting it learn for a longer time. In comparison, the network from AlphaGo program was trained on 10^8 cycles. Perhaps this can improve the performance and learn how to choose the best moves for each board state. Also, an additional feature for the project would be a website that would allow users to play a game of draughts with the network.

References

[1] Sebastian Risi and Mike Preuss. From chess and atari to starcraft and beyond: How game ai is driving the world of ai. *KI-Künstliche Intelligenz*, 34(1):7–17, 2020.

[2] Checkers. https://en.wikipedia.org/wiki/Checkers, Jan 2022. Accessed: 2022-02-03.

[3] Parth Bhavsar, Ilya Safro, Nidhal Bouaynaya, Robi Polikar, and Dimah Dera. Machine learning in transportation data analytics. In *Data analytics for intelligent transportation systems*, pages 283–307. Elsevier, 2017.

[4] Li Meng. *Applying Reinforcement Learning with Monte Carlo Tree Search to The Game of Draughts*. PhD thesis, 2019.

[5] English draughts. https://en.wikipedia.org/wiki/English_draughts, Jan 2022. Accessed: 2022-02-03.

[6] History of checkers. https://historygames.ru/nastolnyie-igryi/istoriya-shashek.html. Accessed: 2022-02-03.

[7] Govert Westerveld. *The History of Checkers (Draughts)*. Lulu.com, 2013.

[8] BinaryDistrict Russia. The history of ai and human competitions: who wins. https://vc.ru/flood/39184-istoriya-sorevnovaniy-ii-i-cheloveka-kto-kogo, Jun 2018. Accessed: 2022-02-03.

[9] Ibm 701. https://en.wikipedia.org/wiki/IBM_701, Dec 2021. Accessed: 2022-02-03.

[10] Nemesis (draughts player). https://en.wikipedia.org/wiki/Nemesis_(draughts_player), Jul 2016. Accessed: 2022-02-03.

[11] Kingsrow. https://en.wikipedia.org/wiki/KingsRow, May 2019. Accessed: 2022-02-03.

[12] Chinook (computer program). https://en.wikipedia.org/wiki/Chinook_(computer_program), Dec 2021. Accessed: 2022-02-03.

[13] Martin Bråten. The application of the tsetlin machine in checkers. Master's thesis, University of Agder, 2020.

[14] Jonathan Schaeffer. *One jump ahead: challenging human supremacy in checkers*. Springer Science & Business Media, 2013.

[15] the american checker federation – welcome to the acf website. https://www.usacheckers.com/. Accessed: 2022-02-03.

[16] English draughts association. https://web.archive.org/web/20191025113401/http://www.englishdraughtsassociation.org.uk/. Accessed: 2022-02-03.

[17] Project. https://webdocs.cs.ualberta.ca/~chinook/project/. Accessed: 2022-02-03.

[18] Deep blue (chess computer). https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer), Jan 2022. Accessed: 2022-02-03.

[19] Max Pumperla and Kevin Ferguson. *Deep learning and the game of Go*, volume 231. Manning Publications Company, 2019.

[20] Alphago: The story so far. https://deepmind.com/research/case-studies/alphago-the-story-so-far. Accessed: 2022-02-03.

[21] Alphago. https://en.wikipedia.org/wiki/AlphaGo, Dec 2021. Accessed: 2022-02-03.

[22] Minimax. https://en.wikipedia.org/wiki/Minimax, Jan 2022. Accessed: 2022-02-03.

[23] Recursion (computer science). https://en.wikipedia.org/wiki/Recursion_(computer_science), Jan 2022. Accessed: 2022-02-03.

[24] Horizon effect. https://en.wikipedia.org/wiki/Horizon_effect, Jan 2022. Accessed: 2022-02-03.

[25] Alpha–beta pruning. https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning, Jan 2022. Accessed: 2022-02-03.

[26] Monte carlo tree search. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search, Jan 2022. Accessed: 2022-02-03.

[27] Machine learning. https://en.wikipedia.org/wiki/Machine_learning#:~:text=Machine%20learning%20(ML)%20is%20the,by%20the%20use%20of%20data.&text=Machine%20learning%20algorithms%20build%20a,explicitly%20programmed%20to%20do%20so., Jan 2022. Accessed: 2022-02-03.

[28] Deep learning. https://en.wikipedia.org/wiki/Deep_learning, Jan 2022. Accessed: 2022-02-03.

[29] Neural network. https://en.wikipedia.org/wiki/Neural_network, Jan 2022. Accessed: 2022-02-03.

[30] Keras documentation: Dense layer. https://keras.io/api/layers/core_layers/dense/. Accessed: 2022-02-03.

[31] Rectifier (neural networks). https://en.wikipedia.org/wiki/Rectifier_(neural_networks), Jan 2022. Accessed: 2022-02-03.

[32] Softmax function. https://en.wikipedia.org/wiki/Softmax_function, Jan 2022. Accessed: 2022-02-03.

[33] Logistic function. https://en.wikipedia.org/wiki/Logistic_function, Jan 2022. Accessed: 2022-02-03.

[34] Hyperbolic functions. https://en.wikipedia.org/wiki/Hyperbolic_functions, Jan 2022. Accessed: 2022-02-03.

[35] Keras documentation: Rmsprop. https://keras.io/api/optimizers/rmsprop/. Accessed: 2022-02-03.

[36] Keras documentation: Regression losses. https://keras.io/api/losses/regression_losses/#meansquarederror-class. Accessed: 2022-02-03.

[37] Keras documentation: Regression metrics. https://keras.io/api/metrics/regression_metrics/#meansquarederror-class. Accessed: 2022-02-03.

[38] Reinforcement learning. https://en.wikipedia.org/wiki/Reinforcement_learning#:~:text=Reinforcement%20learning%20(RL)%20is%20an,supervised%20learning%20and%20unsupervised%20learning., Jan 2022. Accessed: 2022-02-03.

[39] Q-learning. https://en.wikipedia.org/wiki/Q-learning, Jan 2022. Accessed: 2022-02-03.

[40] Ai programming (it-3105) project module 4: Deep reinforcement learning for game playing. https://www.idi.ntnu.no/emner/it3105/materials/neural/deep-rl-pegs.pdf, Jan 2022. Accessed: 2022-02-03.

[41] Alphazero. https://en.wikipedia.org/wiki/AlphaZero, Jan 2022. Accessed: 2022-02-03.

List of Figures