DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Playing the Game of Risk with an AlphaZero Agent

ERIK BLOMQVIST

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Playing the Game of Risk with an AlphaZero Agent

ERIK BLOMQVIST

Master in Computer Science, specializing in Systems, Control and Robotics
Date: November 18, 2020
Supervisor: Mika Cohen
Examiner: Mads Dam
School of Electrical Engineering and Computer Science
Host company: Swedish Defence Research Agency (Totalförsvarets forskningsinstitut), FOI
Swedish title: Spela Risk med en AlphaZero agent

Acknowledgements

I would like to thank my supervisors Mika Cohen at KTH and Farzad Kamrani at FOI for their valuable support and discussions throughout the project. There have been many problems along the way and the project wouldn’t have been possible without their guidance.

Abstract

This thesis is done in collaboration with the Swedish Defense Research Agency and investigates the learning capabilities of zero learning algorithms from a war gaming perspective by applying them to the classic strategy board game Risk. Agents are designed using the Monte Carlo Tree Search algorithm for online decision making and is aided by a neural network that learns offline action policies and a state evaluation function. The zero learning process is based on the Expert Iteration algorithm, an alternative to the famous AlphaZero algorithm, learning the game from self-play. To suit Risk, the neural network used a flat state input representation and had five output policies, one for each decision included in the game. Results show that zero learning could be used to train the neural network such that its policy output improved performance of the Monte Carlo Tree Search algorithm. Agent performance did however not improve with further iterations of network training and the network failed to learn a good scalar state evaluation function.

Sammanfattning

Denna avhandling är genomförd i samarbete med Totalförsvarets forskningsinstitut, FOI, och undersöker inlärningsmöjligheterna hos zero learning algoritmer utifrån ett krigsspelsperspektiv genom att applicera dem på det klassiska strategibrädspelet Risk. Spelagenterna är designade att använda Monte Carlo Tree Search algoritmen för löpande beslutstagande och stöds av ett neuronnätverk vilket lär sig en fix beslutspolicy och att utvädera spelets tillstånd. Zero learning-processen är baserad på Expert Iteration algorithmen, vilken är ett alternativ till den välkända AlphaZero algorithmen, och lär sig spelet genom att spela mot sig själv. För att fungera för Risk så använde neuronnätverket en platt representation av speltillståndet som indata och fem beslutspolicies som utdata, en för varje typ av beslut i spelet. Resultaten visar att zero learning kunde användas för att träna ett neuronnätverk på så sätt att dess beslutspolicies förbättrade spelstyrkan för Monte Carlo Tree Search algoritmen. Agentens spelstyrka förbättrades dock inte med ytterligare iterationer av nätverksträning och nätverket misslyckades med att lära sig en bra funktion för att utvärdera speltillståndet.

Contents

Chapter 1

Introduction

Artificial intelligence research can be said to have started when John McCarthy coined the term in his invitations to a summer conference at Dartmouth College in 1956. At that time it was about defining and developing concepts for "thinking machines", which were becoming part of a small set of different fields in parallel with the ongoing development of the computer [1]. Since then, artificial intelligence (AI) and its methods have branched far out and improved drastically. Today, artificial intelligence systems have a daily impact on our lives and its potential for improvement is seen as one of the most important technology developments of the future.

A philosophical idea that has been driving the work on AI is the creation of a super-intelligence that is far greater than the capabilities of us humans. It is known that this goal comes with a large number of ethical problems apart from the pure technological challenge, but in practice, we are far from this goal. However, games is an area that has long been used as a platform for testing AI algorithms and have step by step reached into the superhuman level. A reason why games are well suited for AI development is that they are artificial environments, which can be changed arbitrarily to include different characteristics and challenges. Some games, like Tic-Tac-Toe, are trivially solved by searching forward through all possible final outcomes, whereas games like Checkers, Chess or Go are harder and require some sense of strategy.

One of the first published systems that could play games on an average human level was created by Arthur Lee at IBM, who also participated at the 1956 conference. His work was done on the game Checkers during the 50s and 60s and used search-trees and scoring functions as basis for the decisions. The system was an early implementation of what is known as the alpha-beta algorithm and was able to learn both through data from professional games and by playing the game against itself. On a high level perspective, this type of learning is very similar to how games are learned with AI today. Following his success, the focus shifted towards the grand game of Chess. This was a hard challenge and it would take all the way until 1997 for IBM and their system DeepBlue to win a six game match against the chess world champion Garry Kasparov [2], thus taking AI to the superhuman level.

1.1 Deepmind's leap

In 2016 the team at Deepmind and their algorithm AlphaGo managed to win with 4-1 against one of the worlds best Go players, Lee Sedol. This was the next major breakthrough for artificial intelligence in games. Go had long been considered a pinnacle challenge and its solution was, at the time, estimated to a decade further into the future. The AlphaGo system was an efficient combination of tree search methods, neural networks and reinforcement learning. To reach their results, the algorithm was trained with data from 160,000 games of professional human play and supplied with a set of handcrafted Go-features, the rest of the learning was then done by self-play [3]. During the year following this publication, the algorithm was improved to the level that it could learn entirely from playing against itself, *tabula rasa*. It only needed to know the rules of the game. Deepmind named the new algorithm AlphaGoZero and when played against AlphaGo, it won with 100-0 [4]. The remarkable thing about this achievement was that they had found a general algorithm to beat Go. With some further updates they created the fully generic general-purpose reinforcement learning algorithm AlphaZero. When tested on Chess it outplayed the worlds best Chess program Stockfish after just four hours of training [5].

The development step from AlphaGo to AlphaZero was also done, independently of Deepmind, by Thomas Anthony et al. [6] in the same time frame for a different game. They proposed the Expert Iteration algorithm (EXIT), which on a conceptual level is the same as AlphaZero, and tested it on the game Hex where it surpassed the best program MoHex. With these new discoveries, AI in games became both general and far more superhuman.

1.2 Thesis background

This thesis is done in collaboration with the Swedish Defense Research Agency (FOI). FOI is an agency that provides a wide range of expertise regarding techniques to handle threats and vulnerabilities for the Swedish Armed Forces and the civil society. With the recent improvements of game AI, there is an interest of investigating how AI algorithms can potentially be used as a decision support system within war-gaming scenarios. Unlike board games, war games are a field of games aimed to simulate real scenarios in order to find a good strategy for the scenario at hand. These are games that have long been used as analytic and educational tools within defence forces around the world. One step of the investigation is to apply the new AI algorithms to games with war-gaming characteristics.

1.2.1 Artificial intelligence in war-gaming

The application of artificial intelligence in war-gaming has had some publications in the recent years. J. Goodman et al. provide a thorough investigation of war-game features and the progress of AI techniques along with recommendations of how these two subjects can be connected in future work [7]. Their key finding is that the biggest challenge is to create an AI the understands the core of war-games such that it does not require large development efforts for each application scenario. Goodman also see significant potential benefits of developing these methods, mainly regarding training of military staff and better decision making.

Development of artificial intelligence methods in war applications is however an ethically complex subject. The core issue is that the potential of AI has been described as the third revolution in warfare, for which a large amount of inhumane offensive actions follow [8]. These actions are related to the use of lethal autonomous weapons that can acquire and engage targets without human intervention. It raises the question if machines or humans should be making the decision of who to kill and what can happen when machines fail to follow their intended objectives. Using these weapons also reduces the cost of going to war by removing the need for military training and importantly the risk of own casualties which would be very beneficial for terrorist organisations. For war-gaming, these ethical questions are less crucial since the AI system is used as decision-support for the humans in charge but it is still necessary to maintain an ethical approach here.

To guard society from a devastating global AI arms race, the Future of Life Institute has suggested a ban on offensive autonomous weapons, like the international ban on nuclear weapons, and gathered over 4000 AI researchers that support this view [8].

1.3 Thesis purpose

The purpose of this thesis is to investigate the learning capabilities of zero learning algorithms (EXIT and AlphaZero) when applied to one game with war-gaming characteristics. The chosen environment is the classic strategy board game Risk, which is a fictional military game about conquering the world. Although not a war game, Risk provides relevant war-gaming features such as

- battles between armies are decided with dices, introducing non-deterministic outcomes where sometimes the army at a disadvantage wins,
- conquering of more areas gives the player access to additional army resources at the next turn, a reflection of how military forces make advancements towards key positions.

From an AI perspective, Risk is interesting since each turn is a series of decisions spread over three sequential phases rather than just one decision as for Chess or Go. This difference, along with the random battle outcomes, yields that the state can change far more between turns which can be a challenge for search-based algorithms.

The goal is to use zero learning algorithms to design and train an agent that reaches superhuman game-play level. The agent design is intended to be done as close to *tabula rasa* as possible, although some trivial modifications will be done to the game outside the perspective of the algorithm. Design is not restricted to follow either EXIT or AlphaZero exactly but should modify them to suit Risk. It is of further interest to achieve this goal as there currently are no known advanced agent designs published for the game. There are also limited possibilities of comparing the agent to other systems so estimates will be made to see how the different sub components of the algorithm compare against each other. The purpose of the report can be summarised in the following research questions.

- What is the learning performance of zero learning algorithms when applied to the non-deterministic and large branching environment of the classic board game Risk?
- What are the performance differences between playing Risk with a neural network agent, a tree search agent or a zero learning agent that combines both?

1.3.1 Scope

The project was limited to only assessing the performance of the algorithms in a 1 vs. 1 scenario. This setting is how zero learning have been applied for games before, and since the focus of the report is the Risk environment rather than its additional multiplayer dynamics then these dynamics have been removed. The prior judgement, based on how the game has a different structure and results from *An Intelligent Artificial Player for the Game of Risk* by M. Wolf [9], is that Risk is complex enough to try and learn even in the 1 vs. 1 setting.

Due to time limitations and the amount of computational resources available at FOI, the project was also limited to only the playing phase of Risk. The initial setup with territory drafting and reinforcing, further explained in subsection 2.1.1, will therefore be randomized at the start of each game. Some additional adjustments are made to the game in terms of which settings are used, these are specified in section 4.1.

1.4 Outline

The remainder of this report is structured as follows. Chapter 2 presents general knowledge of the game Risk and related work done in the field, both for the game in specific and how comparable games have been approached. Chapter 3 describes the relevant theory for understanding the content of the project. In chapter 4 the methodology is covered, motivating choices of game modifications, agent design, the learning process, and experiments performed. Chapter 5 presents the results which are followed by a discussion in chapter 6. The report is summarised with a conclusion and short comments regarding future work in chapter 7.

Chapter 2

Background

This chapter is intended to provide a general background of the game Risk and give an insight in the prior attempts of creating a good AI for the game. It also covers research done on other comparable games.

2.1 Game of Risk

Risk is a classic strategy board game for two to six players that was invented in 1957 by Albert Lamorisse under the name *La Conquete du Monde*. It was then modified for the American market and published by Parker Brothers in 1959 as *Risk Continental Game*. Risk is a fictional war-game, played with pieces representing armies that are used to attack and defend. The game board is a map of the world, divided into 42 different territories across six continents as shown by Figure 2.1. During the game, each territory is ruled by the player who has armies in that territory and the objective is to conquer the world by occupying all territories, eliminating the other opponents in the process. The game also includes a set of 42 Risk cards, each representing a territory and an army symbol, plus two wild cards that are used in various occasions of the game.

The following section gives a brief overview of how the game is played, further information and the official rules are provided by Hasbro[1].

[1]Official rules for the game Risk `https://www.hasbro.com/common/instruct/risk.pdf`

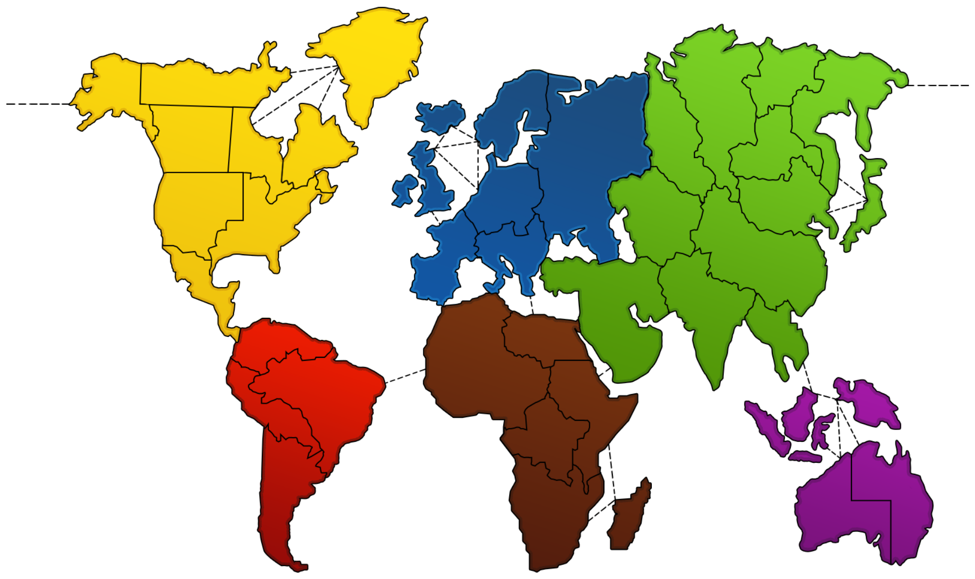


Figure 2.1: Illustration of the standard Risk game map.

2.1.1 Game setup

At the start of the game, players receive 20 to 36 armies (troops), depending on how many are playing. The territories are then assigned to the players by taking turns of placing one army in any unoccupied territory. An alternative is to use the Risk cards to randomize the assignment. The setup is completed after the remaining armies have been placed in the occupied territories, which is done in turns, placing one at a time, like the initial placements.

2.1.2 Taking a turn

Risk is a turn-based game and each turn a player goes through three different phases. These are receive and place troops, attacking, and fortification. The attack and fortification phases are optional and the attack phase continues until the player decides to stop, or no more attacks are possible.

Receive and place troops

A turn is started by calculating the number of new armies to receive. There are three different sources that adds to this bonus.

- The number of own territories.
- The value of continents occupied by the player.

- The value of Risk cards, if traded in.

The territories yield one army for every three territories occupied, or three armies if less than nine territories are occupied. If a player occupies all territories of a continent at the start of his turn then additional troops are received according to the value of the continent, these continent values are specified on the game board. The last army bonus is through trading in Risk cards that have been gathered earlier. Each Risk card has an army symbol of either infantry, cavalry or artillery. The cards are traded in as sets of three according to any of the combinations in Table 2.1. The pile of cards also have two wild cards which can be substituted for any of the symbols to form a valid combination. The game can be played with fixed or progressive card bonuses. The fixed bonuses are shown in Table 2.1 and the progressive rule is that the first set gives four troops and increases with two for every next set until five sets have been used, regardless of which player did the trading. The sixth set is worth 15 troops and every set after that is worth five more. The received troops are then freely placed in any of the own territories.

Card Combination	**Value**
Three Infantry	4
Three Cavalry	6
Three Artillery	8
One of each	10

Table 2.1: Different combinations of trading in Risk cards and their value for fixed card bonus play.

Attacking

A player can choose to attack neighbouring enemy territories from own territories that have two or more armies. The attacks are done in sequential battles of 3 vs. 2 where the attacker rolls three dices and the defender two, fewer dices are used if there are less armies on either side. The highest outcome of each side is compared and the attacker takes out one defending army if the value is higher, defenders win equal outcomes. The second highest outcomes are then compared the same way. After each battle, the player can choose to continue the attack on the same territory by doing another battle, attack other enemy territories with battles there or end the attack phase. If all defenders are taken out in a battle the attacker chooses how many troops to move into the new territory, always leaving at least one behind. For digital play, it is normal to remove the repeated decision of battling again and let the computer simulate until one side has won to speed up the game, this is called blitz. At the end of the attack phase, the player draws one Risk card if at least one new territory was occupied.

Fortification

The last part of the turn is fortification which means that the player can chose to move troops from one territory to another as long as there is a path of own territories between them. The player always has to leave at least one army in the territory where they are moved from.

2.1.3 Two-player Risk

The setup for two-player Risk is slightly different than for the normal multiplayer version. The game is here initialized with a third neutral player. At the start, the two players and the neutral are randomly given 14 territories to place one troop in. After this follows the initial reinforcement phase where players take turns placing two troops in own territories and one in one of the neutral territories until the three players have 36 troops in total across the board. It is common that also this phase is randomized when playing on the computer or in an app. From here on, the game continues as the multiplayer version except that the neutral player is completely passive, it doesn't receive any troops to reinforce with, attack, nor fortify. This adjustment to the game lowers the amount of territories occupied at the start from 21 to 14 which makes the games a little longer and the strategic depth increases as it is up to the two players to decide if they want to claim the territories of the neutrals or attack the active opponent.

2.2 Related work

Zero learning algorithms like AlphaZero and EXIT are the center of this work and together they have proven to produce superhuman results in challenging zero-sum, combinatorial game domains (Chess, Go, Hex and Shogi among others). Risk is however a substantially different board game where the strategic challenges are also amplified by random initialization of the game and the attack phase outcomes. At the current point in time, there have been no publications of applying zero learing methods to Risk, and on a further note, the amount of scientific research on creating a clever AI for the game is limited.

2.2.1 Prior work on Risk

In 2003, Jason Osborne published a Markov chain analysis for the battle system in Risk [10]. The goal was to understand the probabilities of conquering a territory and the expected loss of armies as a function of the number of attackers and defenders. His results show that the attacker has an advantage even when the number of attacking and defending armies are equal (conditioned on there being at least five each) and that this advantage is increasing for larger amounts of equal armies in the battle. Graphs for the probabilities of winning were also presented.

State and decision complexity

Following the findings of Osborne, attempts were made to translate the knowledge of Risk into an AI system. One important paper that addresses this challenge was written by Michael Wolf as his thesis in 2005. A key contribution of this paper is the analysis of the game's state and decision complexity. Risk may have a game board and discrete state-space like Chess and Go but the way the game is played causes the state-space to evolve very differently.

In a turn of Risk, the player receives new troops to place but there is no rule that forces the player to attack. So, if all players continuously doesn't attack, then no armies will be lost in battle while new are added. The result of this train of thought is that the state-space of Risk is theoretically infinite [9]. This is not how the game is played in practice and Wolf show an example calculation that yield a state-space of 10^{47} for a four player game with 200 total armies across the board.

The decision complexity of Risk can be measured with the average branching factor, which represents the average number of states reachable from one decision step at any point of the game. Wolf computes two approximations of this factor, although defined as the average branching between start and end of one players turn since it is a series of decisions, yielding 10^{33} and 10^{85}. These are notably large numbers and could be reduced down to 10^{6} after some modifications to the decision structure of the place troops phase. As a comparison, the estimates for Chess and Go are 31 and 250 respectively. The massive number can be reasoned about just like theoretic lack of limit to the state-space. Both the placement of troops and fortification phase requires the player to decide the number of troops to place or move. For the case of receiving new armies, each of these can be freely placed in any of the owned territories. This yields that the number of valid actions (all different distributions of armies) for just that phase is very large. The attack phase is less branching as there might only be a handful of territories possible to attack but here the random battle outcomes still causes the set of states after finishing the phase to be somewhat large. What is of matter is not the exact branching value but that the rules allow for substantially larger branching, specifically for placing troops.

Artificial intelligence systems

Wolf also contributes with two AI agents, one basic player and one enhanced player. The basic player chooses among its possible decisions by simulating each, rating the outcome and greedily choosing the best option. The simulations include no further look ahead and the rating is done with a linear evaluation function. This functions takes the game state and current player as input and computes a set of handcrafted feature-values which are returned as a weighted sum. Wolf states that the performance of this simple approach was poor and one big problem was that it didn't show any sense of coordination between decisions within a turn. To solve this, the enhanced player received additional features that were computed first to assess if a high-level plan would be activated. Examples of plans were conquering a specific continent or eliminating a weak opponent. If a plan was activated, then actions according to the plan were rated higher in the ordinary feature evaluation. Changes were also made to the placing of troops phase such that it would evaluate the value of placing multiple troops at once according to some simple distribution and compare this value to the values of just placing one as the basic player. The last and most significant change was that the weights for combining all features were parameterized and learned with the reinforcement learning technique Temporal-Difference learning. The performance results of these changes was that the addition of high-level plans and tools to work towards them improved the basic player by a factor of 60 according to his rating system. The learned feature weights improved the performance 20 % further which made it capable of winning regularly versus human beginners, and occasional wins versus more experienced humans. The main conclusion to take from this work is that simple and handcrafted evaluation methods paired with weight learning can be sufficient for a computer system to play Risk.

In 2013, a bachelor thesis was published by Manuela Lütolf at University of Basel which also used features and TD-learning for Risk. His results were similar to those of Wolf and he states that there are situations where the agent is at a disadvantage due to it not looking further than one step ahead [11].

AlphaZero and EXIT are both based on the Monte Carlo Tree Search algorithm. This algorithm was applied to Risk in 2009 but limited to only assessing the performance of the method for drafting territories during the initial setup [12]. A similarity their work has to the later zero learning methods is that they aided the tree search algorithm with an automated evaluation function to score options during the tree simulation. This function was just a simple linear regression trained by supervised learning rather than a full neural network. Different from zero learning, they generated their data for supervision only once and without the tree search algorithm. The results of this method was a bot that outperformed all of the strong bots available among Lux Delux agents, a framework for playing Risk. Their bot was however only different from the agents of the framework during the initial draft and used one of the agents unmodified during the playing phase. The conclusion is thus that advantages of game winning significance can be achieved already during the setup phase of Risk and that this type of methods work well, indicating promising potential for zero learning the whole game.

2.2.2 Settlers of Catan

Settlers of Catan is a similar multiplayer strategy board game where the strategic focus is about resource management to expand the own territory while under the influence of random dice rolls. For this game there have been multiple publications of creating a sophisticated AI agent. One of these publications, written by Pieter Spronck et al. in 2009, investigates the applicability of the Monte Carlo Tree Search algorithm [13]. Their work simplified the game such that the algorithm didn't deal with any hidden information and the agent was restricted from trading with other players. The intention of these changes was to reduce the complexity of integrating the game with the algorithm while only causing minor changes to the overall strategic game play. This type of idea can also be applied to Risk. To measure the performance of the MCTS agent, it was compared to the JSettlers framework which is widely used and provides a heuristics based agent that at the time, was considered as one of the best for the game. The results show that the MCTS algorithm performs well and frequently beats the JSettlers agent in a multiplayer setting. Some tests were also done against humans where the results were unreliable but concluded as the agent showing reasonably strong play. On a whole, the algorithm was marked as "a viable approach for other multi-player games with complex rules", thus further establishing the potential of the method for Risk .

2.2.3 War-gaming with AlphaZero

In connection to the interest of war-gaming, there was an article published in November 2019 by G. Moy and S. Shekh where they applied AlphaZero to a simple war-game, Coral Sea [14]. Coral Sea is a two-player asymmetric game with multi-step turns played on a hexagonal grid. The structure of this game make it very different from Chess and Go, and with the multi-step turns, it has similarities to Risk. Their work focuses on exploring how the differences in game characteristics, such as, problem representation, goal asymmetry, and limited strategic depth as well as lack of significant computational resources affect the performance of AlphaZero. To overcome the investigated challenges, they propose that AlphaZero is bootstraped with supervised learning, incorporating heuristic knowledge in the action selection during self-play. Results show that AlphaZero was able to learn the game to the level that it outperformed the heuristics which it was aided by. This work show that AlphaZero, with some modifications, is capable of learning to play games with significant structural differences compared to previous applications. It does however not include technical details of the system design so it can only be considered as an indication that AlphaZero will also work for Risk rather than as a methodological blueprint.

Chapter 3

Theory

The purpose of this chapter is to cover the theoretic subjects required to understand and analyze zero learning algorithms. Its first section describes the reinforcement learning field with according general concepts. The next two sections goes more into detail with the Monte Carlo Tree Search (MCTS) algorithm and the neural networks that are the two parts of zero learning decision making. The last section then describes zero learning with a focus on the EXIT algorithm.

3.1 Reinforcement Learning

Reinforcement learning (RL) is a field of machine learning that covers the task of learning what action to take in order to maximize some measure of reward. It is an iterative process where the learning agent is not told what to do but has to find which decisions are the best by trying them. From a general learning perspective, the field can be labeled as a computational approach to learning from trial-and-error interaction. This type of learning is applicable to a wide range of tasks and has been of most interest for sequential decision making problems. The extra challenge of sequential decision making is that it might not be sufficient to take good decisions for the current situation as some actions can have indirect consequences on actions and rewards available at later stages. The systems thus needs to take actions such that their sequence achieves the long-term goal.

Reinforcement learning problems might also take place within intelligent environments, take for example a game where an RL agent plays against humans. If the agent acts predictably then humans are able to notice and change their play-style accordingly, thus dynamically changing the environment's response to the agents actions. These are challenges that also need to be considered in the decision process.

The iterative learning of an RL agent is stored in policy and value functions which are used to dictate the agent's behaviour, further explained in section 3.1.2. Since these are trained using self generated experience, the system needs to make sure that this set covers enough states and action decisions to find optimal solutions, this is known as the exploration versus exploitation tradeoff. A reinforcement learning algorithm is therefore any method that specify both a structure for generating experience and how the experience should be used to incrementally change the policy and value functions such that their estimates give better behaviour.

Further insights into the reinforcement learning field can be found in the book *Reinforcement Learning: An Introduction* by Sutton and Barto which this and the following section are based on [15].

3.1.1 Markov decision processes

The Markov decision process (MDP) is a mathematical formulation for modeling sequential decision making problems. MDP is based around dividing the agent's interactions with the environment as a sequence of discrete time steps, $t = 0, 1, 2, 3...$. For every time step, the agent receives a representation of the environments state, $s_t \in S$, a measure of reward, $r_t \in R$, and is to decide which action, $a_t \in A$, to take. This description of the interactions yields a state, reward and action trajectory

$$s_0, a_0, s_1, r_1, a_1, s_2, r_2, a_2, s_3, ...$$

which can either be episodic and finite or infinite, depending on the task. Games played from start until there is a winner are an example of finite and episodic tasks, one episode corresponds to one game.

The transition dynamics of an MDP is modelled as a probability distribution over states and rewards reachable from the current state and chosen action according to the following equation

$$p(s', r|s, a) = \mathcal{P}(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a). \tag{3.1}$$

This equation assumes the Markov property which is a restriction on the state to fulfill $p(s_{t+1}|s_t, a_t, ..., s_0, a_0) = p(s_{t+1}|s_t, a_t)$. The Markov property means that the current state contains everything necessary for understanding the next transition. Equation 3.1 is used to model all information of the environment and can be used to compute the expected reward function r for state-action pairs (s, a). This function is a property of the environment in combination with how the reward signal is specified and is defined as

$$r(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in S} p(s', r|s, a). \tag{3.2}$$

It is a summation of the reward signal r over states s' reachable from (s, a). The goal of the agent is to maximize the accumulated reward from this function. In the context of games, $r(s, a)$ normally only yields non-zero rewards for the state-action pairs that causes a transition to terminal states, indicating that one player has won.

3.1.2 Policy and value functions

To achieve the goals of the MDP problem formulation, the agent needs to act such that its trajectory yields the highest long-term reward. A learning agent's behaviour is specified by a policy function which represents the mapping from states to the actions that it should take. Formally, this mapping is defined as a probability distribution, π, over possible actions for each state of the environment according to the following equation

$$\pi(a|s) = \mathcal{P}(A_t = a|S_t = s). \tag{3.3}$$

During learning of the policy function, the quantification of which actions yield most long-term reward can be done with the use of a value function. This is a function that for a given state estimates how much reward an agent can expect to accumulate from that state and onward. The estimate does however, depend on the policy function since it is the policy that controls which states and rewards are encountered later. Therefore, the value function, v, is defined together with π as

$$\mathrm{v}_\pi(s) = \mathbb{E}_\pi[R_{t+1}|S_t = s], \tag{3.4}$$

where $\mathbb{E}_\pi$ is the expectation over rewards gathered when following policy π.

Policy and value functions can either be represented in tabular form or with a function approximator. The tabular form is simply a table of trainable values for each state and chosen action combination the system can be in, memorizing experience seen at each slot. It is therefore only suited for tasks with small and discrete state-spaces since the table needs to be of size S x $\mathcal{A}$ to even represent a policy. A function approximator is instead a method that takes the variables of the state representation and applies a mapping, using a set of parameters θ, from the input to an output policy or value. By using a function approximator, the set of trainable values reduces to just the parameters of the approximator. An even better reason for using a function approximator during training is that this method yields the property of generalizing from states that have been visited to those that haven't been seen, thus requiring even less experience. The downside of the method is that the approximator needs to be appropriately complex such that it can reflect the unknown true function [1].

3.2 Monte Carlo Tree Search

Monte Carlo Tree Search is an online decision making algorithm that uses Monte Carlo random simulations to bias a tree search towards promising actions. The algorithm was introduced in 2006 in a paper by Coulom where he applied it to Go and achieved major improvements compared to previous versions of Monte Carlo search methods [16]. The algorithm was additionally updated in the same year by Kocsis and Szepesvári with their introduction of the Upper Confidence Bounds applied to Trees (UCT) algorithm that guides the selection of actions during search [17]. Since then, the algorithm has been an active field of research and translated into state-of-the-art programs for a variety of games. Much of its later variations and enhancements can be found in Browne's survey of the algorithm along with references to 250 other MCTS publications [18]. Two favourable properties of the algorithm is that it is not dependent on domain-specific knowledge and the tree search is asymmetric. The algorithm only requires that the domain where it is applied is able to generate the set of possible actions to push the state of the tree search forward. The asymmetric property is that the tree is free to expand deeper asymmetrically rather than layer by layer as in a breath-first search. The MCTS algorithm can thus focus on areas of the tree that are believed to be most favourable.

3.2.1 Algorithm description

The MCTS algorithm is an algorithm which for each decision builds a game tree and uses the search statistics to compute a policy over currently available actions, it does not depend on any previously learnt offline knowledge. Building of the game tree is an iterative process starting from the current state of the game, where each child node is a future state reached by the action which labels the edge between parent and child. The process of building this tree is divided in four steps, selection, expansion, rollout and back-propagation. Each of these steps are done once per iteration and will hereby be called an MCTS simulation. When the requested amount of simulations are done, the algorithm returns the action below the root that was taken the highest number of times during search. Following is a short summary of the algorithm steps, further details and variations can be found in [18].

Selection

During selection, the algorithm starts from the root and successively selects children until a leaf node is reached. Nodes are selected with the UCT in-tree policy function that uses node reward and visit statistics to trade-off exploration with exploitation. This is the most important step of the algorithm as it decides how the tree grows forward into the state-space, choosing which actions to evaluate more than others. The UCT function is, when using notation from [6], defined by

$$UCT(s,a) = \frac{r(s,a)}{n(s,a)} + c_b\sqrt{\frac{\log n(s)}{n(s,a)}}. \tag{3.5}$$

where $r(s,a)$ is the total reward gathered by simulations passing through the node, $n(s,a)$ is the node's visit count and $n(s)$ is the total visit count of the node's parent. c_b is a constant which controls the amount of exploration allowed by the second term. The dynamics behind this term is that it decreases for the action selected, due to the logarithm, and at the same time increases for all other actions. This yields that actions with a low visit count, and likely high influence of variance in the first term, are eventually selected over actions that have a large amount of visits. In each node, a child that has the highest UCT-value is selected.

Expansion

The leaf node reached by the selection phase can either be visited once or un-visited. If the reached node is visited once then the tree is expanded by creating child nodes for each available action from the state of the leaf node. One of these child nodes are then randomly selected. Any selected and un-visited nodes are not expanded at this time and instead move onto the next phases which will mark them as visited.

Rollout

The new node reached after selection and expansion is evaluated by playing the game from the node state until termination. This is the Monte Carlo random simulation, here referred to as a rollout, and is done according to a default policy, commonly just selecting actions at random. The rollout can also be interrupted early, which is know as a cut-off. This requires an accurate evaluation function so that the non-terminal state can be scored.

Back-propagation

The outcome of the rollout phase yields a score that is back-propagated up the tree, along the path taken during selection, to the root. Incrementing each node's reward statistic with the result from the rollout, accounting for which player had the turn at the node state, and increasing their respective visit statistics by one.

3.2.2 Chance nodes

Chance nodes are a method for modelling stochastic state transitions during search, they are a special kind of node that appear as an extra sub-level of the tree. When the search expands leaf nodes with actions that have non-deterministic outcomes, chance nodes are created. The difference between their state and the parent is that the corresponding action has been chosen but not executed. As the chance node is then expanded, this generates all possible state outcomes as new nodes and selection is done by sampling from the probability distribution over the outcomes [19]. Figure 3.1 shows a visualization of how chance nodes are integrated in the tree.

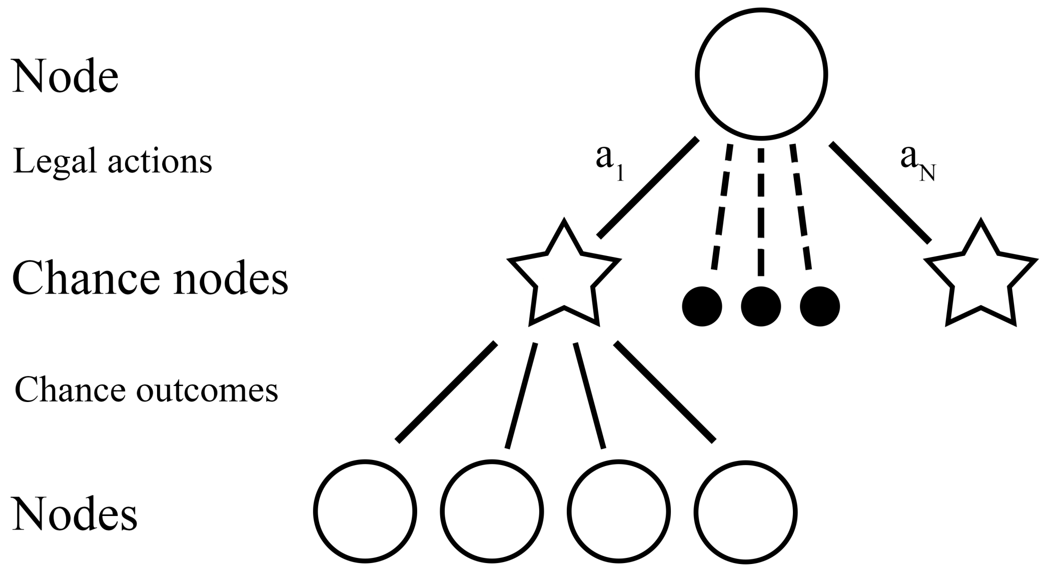


Figure 3.1: Modelling of stochastic events in tree search using chance nodes.

3.3 Neural Networks

Neural networks (NN) are a mathematical construction based around how the neurons in our brains are believed to work. During the last decade, they have proven to yield very successful results within multiple fields of machine learning problems.

Neural networks are used to approximate arbitrary functions f^*. Formally, they define a mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$ which is trained such that the parameters $\boldsymbol{\theta}$ yield the best function approximation. The training is done by pushing $f(\mathbf{x})$ to match $f^*(\mathbf{x})$ using a data set of points $\mathbf{x}_1, ..., \mathbf{x}_n$ where each point has a target output $\mathbf{y} \approx f^*(\mathbf{x})$ [20].

Within reinforcement learning, neural networks are incorporated as very strong approximators for both policy and value functions.

3.3.1 Structure and training

A neural network can be simply described as a directed graph of artificial neurons. Each neuron processes input values **x** received from previous neurons as a linear combination of trainable weight parameters **w** and returns a scalar value describing the neurons response to the input which is passed forward in the network. This computation is described by

$$\sigma(\sum_i w_i x_i + b). \tag{3.6}$$

where b is an additional trainable parameter and $\sigma()$ is a non-linear function known as the activation function. Figure 3.2 visualizes how the directed graph is structured as layers of neurons. The first layer receives a fixed representation of data that the network processes in the hidden layers. A prediction is computed in the output layer where the activation function is changed to suit the task. There are also other configurations of neural networks, for example Convolutional Neural Networks (CNNs) that uses convolutional filters instead of Equation 3.6, these will not however be explained further here.

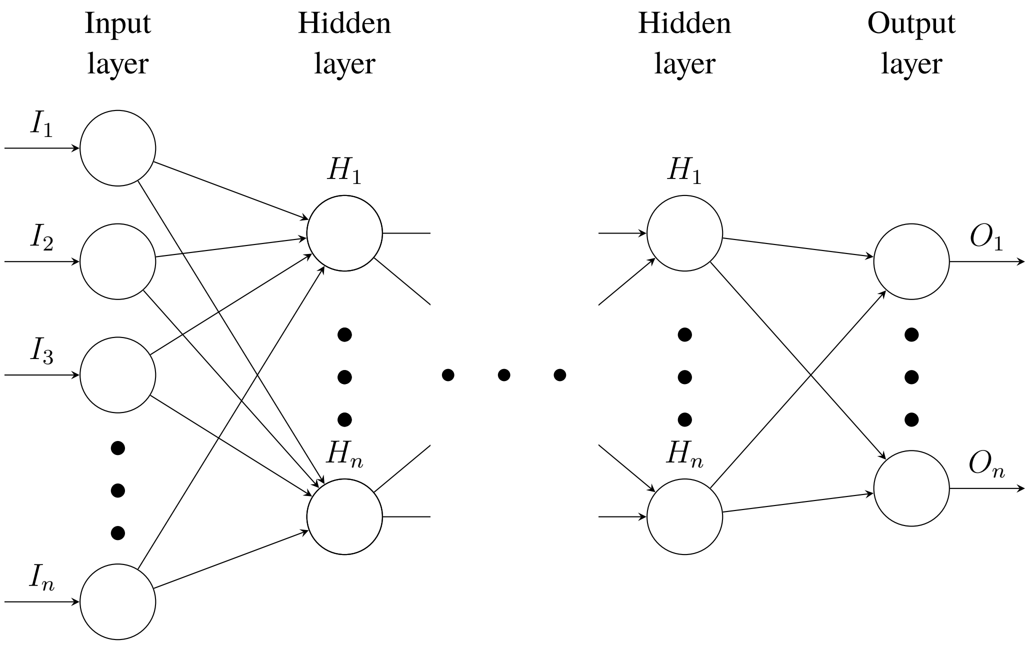


Figure 3.2: Structure of fully connected neural network with multiple hidden layers.

Training

Neural networks are trained using a loss function, l, and an optimization algorithm. The loss function compares the output prediction of the network, $\hat{\mathbf{y}}$, with the corresponding target data, $\mathbf{y}$, and returns a scalar measure of their differences. The optimization algorithm takes the results from the loss function and computes how the weights should be changed such that the model better fits the data. To prevent overfitting, the data is split into training and validation data where the validation data is separate and used to estimate the loss performance on untouched data. The training is done in epochs, for each epoch, all training data is used to update the parameters. Training continues for a fixed number of epochs or until the validation loss is constant within some margin. These details are among the basics of supervised machine learning and are further described in any machine learning book.

3.4 Zero Learning

Zero learning is known as the family name for algorithms based on the design concept of Deepmind's AlphaZero algorithm. These are methods that combine the Monte Carlo Tree Search algorithm with neural networks to learn strategic boards games entirely from self-play. The EXIT algorithm conceptualises the idea of zero learning as having an expert player and an apprentice which together perform iterations of the learning process. It is done by playing the game, letting the expert (MCTS) encounter multiple situations for which the search decision statistics are stored, thus generating a data set of the experts knowledge. The apprentice (NN) then takes the data and trains to match it, yielding a better network. The key with this method is that the online policy building process of the MCTS expert can be improved by incorporating the offline policy stored within the apprentice network. Rather than initializing each new level of the tree without any knowledge it is now initialized with a prior bias from the policy of the network. The MCTS expert thus uses the network as a tool to choose even better moves during the next data set creation, which allows the apprentice to further mimic the expert. They reinforce each other to iteratively learn the game from zero.

3.4.1 Network design

Although EXIT and AlphaZero are general algorithms for learning board games, the network design is unique for each game domain. The common factor between games is that all networks take a representation of the game state, s, and output actions probabilities, **p**, and a scalar value v according to $(\mathbf{p}, \mathrm{v}) = f_\theta(s)$. It is one network which branches into two output heads after a set of hidden layers. The actions probabilities are known as the policy head and the value output is thus the value head.

For the games where AlphaZero and EXIT has been applied, the state was represented as a 2D image with multiple layers where the shape of each layer matches the game board. The networks could then be designed using hidden convolutional layers which are able to utilise the spatial representation for a better understanding of the state.

The output of the policy head is adapted to the specific game domain by changing its size such that it matches the total amount of actions in the game. At each input state, only some of the actions are possible so the network needs to filter illegal actions for the output probabilities to be correct.

The value head represents the network's estimate of the current player's probability to win the game and is not game dependent. The estimate is in the range (-1,1) where 0 corresponds to 50 % chance of winning.

3.4.2 The Expert Iteration algorithm

This section is focused on describing the EXIT algorithm since the majority of this work is based on this approach to zero learning, relevant differences of AlphaZero are included.

The formal definition of the EXIT algorithm is shown by Algorithm 1, taken from T. Anthony et al. [6]. The first two lines represent random initialization of the neural network and the network being integrated to the tree search agent. The for-loop is then the core of the algorithm where each step hides further design details.

Algorithm 1 Expert Iteration

```
1: $\hat{\pi}_0$ = initial_policy()
2: $\pi^*_0$ = build_expert($\hat{\pi}_0$)
3: for i = 1; i $\leq$ max_iterations; i++ do
4:     $S_i$ = sample_self_play($\hat{\pi}_{i-1}$)
5:     $D_i$ = {(s, imitation_learning_target($\pi^*_{i-1}$(s))) | s $\in$ $S_i$}
6:     $\hat{\pi}_i$ = train_policy($D_i$)
7:     $\pi^*_i$ = build_expert($\hat{\pi}_i$)
8: end for
```

Self-play data sampling

To generate self-play data points, the EXIT algorithm repetitively plays the game using an exploration policy. For each game, one state is selected where the expert is asked to do 10000 MCTS simulations for the action decision. The game is then played until termination and the outcome (± 1 for win/loss) is stored along with the selected state and statistics of the action decision. Only storing one decision per game removes any correlations between data points which would otherwise reduce the learning. The exploration policy is defined by the policy of the most recent apprentice network, which yields much faster play than the tree search. However, for the first iteration of data sampling, the network hasn't been trained so it is replaced by MCTS with 1000 simulations.

Data set management

For each EXIT iteration, the sampling of self-play data results in a new data set D_i which is used in the training step and then thrown away. An online version of EXIT, also presented in the same paper, improves the data set management by aggregating new data sets to that of the previous iteration, thus reducing time required for creating new and sufficiently large data sets. This version then either passes all data or a large buffer from recent iterations to the training. Both of the online version's alternatives showed better performance than the formal data set method.

Training

Both EXIT and AlphaZero train the policy head using cross-entropy loss

$$l_p = -\sum_a \frac{n(s,a)}{n(s)} \log[\hat{\pi}(a|s)], \tag{3.7}$$

where $n(s,a)/n(s)$ is the target frequency of action a being chosen by the expert during tree search, $\hat{\pi}(a|s)$ is the predicted policy distribution from the network using input s. This function is only zero for actions where the target is zero and therefore won't return zero loss if the prediction and target is equal but instead reach a minimum.

For the value head, EXIT uses Kullback-Leibler loss

$$l_{\mathrm{v}} = -\hat{\mathrm{v}}\log[\mathrm{v}] - (1-\hat{\mathrm{v}})\log[1-\mathrm{v}], \tag{3.8}$$

and AlphaZero uses mean-squared-error loss

$$l_{\mathrm{v}} = \frac{1}{N}\sum_{i=1}^{N}(\hat{\mathrm{v}}_i - \mathrm{v}_i)^2, \tag{3.9}$$

where $\hat{\mathrm{v}}$ is the predicted value and v is the target. Both of these functions have the property of non-linearly scaling the loss with an absolute difference, returning more loss where the difference is larger.

Integrating network with tree search

The most recently trained network is incorporated into the tree search algorithm as a further bias towards promising actions. For EXIT, the integration is done by adding both a policy term and a value term to the UCT formula of Equation 3.5 as follows,

$$UCT(s,a) = \frac{r(s,a)}{n(s,a)} + c_b\sqrt{\frac{\log n(s)}{n(s,a)}} + w_a\frac{\pi(a|s)}{n(s,a)+1} + w_v\hat{Q}(s,a). \tag{3.10}$$

Both new terms have their respective hyper-parameters w_a and w_v and $\hat{Q}(s,a)$ represents a backed up average of network value estimates through the edge

s, a. An additional detail of EXIT is that the value term is only included in the UCT formula after a few training iterations.

AlphaZero doesn't add new terms to the UCT formula but instead modifies the formula to

$$UCT(s,a) = \frac{r(s,a)}{n(s,a)} + c_b\sqrt{n(s)}\frac{\pi(a|s)}{n(s,a)+1}. \tag{3.11}$$

The second term is similar to EXIT's policy term except that w_a is replaced by $\sqrt{n(s)}$ and the exploration constant c_b is kept. The value head is integrated as a replacement for random rollouts, the factor $r(s,a)$ is thus the total backed up value estimates for simulations through s, a. AlphaZero also modifies how the UCT value is computed for unvisited nodes. The first term is in that case skipped so the UCT expression is

$$UCT(s,a) = c_b\sqrt{n(s)}\,\pi(a|s) \tag{3.12}$$

for nodes where $n(s,a) = 0$.

Chapter 4

Methodology

This chapter covers the approach taken to apply zero learning methods to learn Risk. The first section specifies the game settings that were used . The following section describes the agent design, motivating choices for the network and state space representation as well as details of the MCTS structure. The learning process is then described, detailing how self-play data was generated and used. The chapter concludes with a description of the performed experiments.

4.1 Game settings

As specified by the scope of the project in subsection 1.3.1, Risk games are played in a 1 vs. 1 scenario with an additional neutral player according to the official rules of two-player Risk. Games are initialized such that each of the three players start with 14 random territories and a total of 36 troops spread across these. Selection of player to start between the two active agents is done by rolling of a die.

With these initial settings specified, the rest were chosen under the intention of limiting the amount of decision types and decisions per turn while maintaining most of the core strategic challenges. The reason for this intention is that design of a zero learning agent for Risk faces problems when actions are chosen through the use of either MCTS or a neural network policy. MCTS is a computationally demanding algorithm which can make playing games time-consuming, this effect is particularly noticeable for Risk due to the amount of unique decisions per turn. For the network, each different type of decision requires its own policy head and fraction of the training data. The following settings were chosen:

- Cards are automatically traded in whenever possible.

- Hidden information is removed from the game by playing with open cards.
- Cards yield a fixed amount of troops according to Table 2.1.
- Attacks are made in blitz-mode (i.e. the battle is automatically played until there is a winner) and all armies except one are moved from the attacking territory to the defending territory.

The strategic disadvantage of making the card decision automatic is that players are not able to wait each other out with trading in their cards. For two-player Risk, this is however only relevant if progressive card bonuses are chosen and is thus believed to be outweighed by the benefit of removing this decision type and one decision per turn. With automatically traded cards, it is natural to tone down their effect on the game by playing with open cards and have fixed bonuses. Progressive bonuses would otherwise have the potential of out-scaling the amount of armies received from conquering territories and continents properly.

For the blitz-mode attack phase, agents are now restricted from the ability to abort an ongoing battle if the outcome is poor. This option is essentially only in the game because it is a board game played manually with only 3 + 2 dices. With a computer system, the outcome can be directly computed for any amount of attackers and defenders. This way of playing is also common when playing Risk in the official app *RISK: Global Domination* or through web-based implementations. It is therefore considered as only a marginal playing disadvantage when compared to how many extra decisions would need to be made with the MCTS. Another benefit is that it makes the game more like how humans think about attacking, that it is done under the intention of conquering a territory.

The attack phase also contains the decision of how many to attack with, how many to defend with, and how many to move into the new territory after a victory. With battles done in blitz-mode, the trivially correct decision is to attack or defend with the amount of armies that yield the highest chance of winning. This amount has been proven to always be all armies [10], which is why these two decisions have been removed from the agents of this system. The decision of how many armies to move forward is set to all because it removes a decision type, it could however be argued about how many occasions this decision opens possibilities for stronger play.

4.2 Agent design

The multiple phase turns and variable phase lengths of the Risk environment introduce new challenges for zero learning methods. With their base in learning *tabula rasa* and suitability for board games they still leave game specific details unspecified. The design of a zero learning agent for Risk is therefore mostly a process of adapting the decision structure to the strengths and weaknesses of Monte Carlo Tree Search and neural networks. Agents need tools to effectively pick out strong and strategy coherent actions among a large amount of options.

4.2.1 Agent action space limitations

To play Risk, agents need to take decisions for each of the game's three phases. The decision complexity of the place troops and fortification decision is however so large that, without any modifications, it will cause problems when generating a policy over all unique actions.

For the place troops phase, the issue is a function of how many troops are considered in the decision. If it is modeled as a one-step decision of distributing all troops onto the own territories then this decision has

$$\frac{T^N}{N!} \tag{4.1}$$

unique distributions, where T is the number of own territories and N is the amount of troops to place. By instead seeing the phase as N decisions of placing one, then each of these decisions can be taken with a policy spanning over only $T < 42$ possible options. This method does however add $N - 1$ decisions which is an apparent drawback for MCTS-based agents. Taking inspiration from the approach of Wolf, the decision structure was designed according to the latter method where agents also have the option of placing either all or half (rounded up) of the available troops per decision. This allows the phase to be finished in one or more decisions, without theoretically limiting the agent, while only adding two more legal options per own territory to the policy.

Another problem of this phase is that the number of legal actions is still unnecessarily large, especially at later stages of the game where most of a player's territories are grouped. With the official rules, agents can place troops in any of the own territories, but for the following attack phase, agents can only attack from territories with two or more troops that are also bordering an enemy territory. Troops placed in territories without bordering enemies can thus not be used in the rest of the turn, which is why agents were limited to only placing troops where there are enemy neighbours. The modification stops poor agents from taking particularly bad decisions and reduces the branching of the tree search, but it is also an incorporation of human knowledge which slightly moves the system away from being *tabula rasa*.

For the fortification phase, the decision is to choose which two territories to move troops between and how many troops that are moved from the source. Since there is no limit on the number of troops in the source territory, the representation of how many to move needs to be a discretization over fractions of the total. In terms of total possible fortifications in the game, disregarding the discretization, there are 42 territories that can fortify armies to any of the other 41, which yields a total of $42 * 41 = 1722$ combinations for fortification source and targets. When including the discretization, it will therefore be 1722 unique actions for each discretization option. As the size of the neural network's fortification policy has to include all possible actions of the phase, the discretization over fraction of troops was limited to two options, either all or half of the troops available to move. There is thus a total of $1722*2+1 = 3445$ actions for moving all or half, where the additional one represents the action of not fortifying.

During play, the amount of legal fortifications was also limited to only target territories with neighbouring enemies. The limitation follows the same reasoning as for the placement phase and arguably has an even larger effect of reducing the branching here. The purpose of fortification is to strengthen territories which are potentially attacked by enemies, so it is natural to limit the action space to only these. It does however, occur situations in Risk where it is better to fortify internal territories but these are rare compared to the extra branching they would cause.

4.2.2 Monte Carlo Tree Search

The Monte Carlo Tree Search algorithm was implemented according to its theoretic description in subsection 3.2.1 and follows the rules and limitations of previous sections.

For the attack and fortification phases, the tree was modified to put more emphasis on the high-level decision of taking an action compared to skipping and ending the phase. It was structured as a two level hierarchical decision where the search first expands two nodes representing whether it wants to take action or skip. All legal active actions are then expanded under the node for taking action. Upon finishing search, the action with most visits is chosen, if this is to take action then the algorithm automatically chooses the most visited attack or fortification action below, thus maintaining the same amount of searches.

The random battle outcomes of attack phase actions was modelled with chance nodes. Since the explicit probability distribution over outcomes as a function of attackers and defenders is unknown, although possible to compute, they were implemented with an alternative method. Each time search passes through a chance node, one sample is drawn from the outcome distribution by use of the game simulator. Followingly, the sample is compared to outcomes of previous transitions and if not seen before, then a new node is expanded with state corresponding to that of the outcome.

The default policy of simulation rollouts was set to choosing actions at random and attacks and fortifications are done with a probability of 75 % and 90 % respectively. These parameters were set by manually tuning random agents to a "strong" behaviour.

To reduce computational time, rollouts were only played for a few turns and then scored using an evaluation function. For MCTS, this is known as rollouts with cutoff. The evaluation function was manually designed as a comparison of the active players resources. To keep the function simple, limiting poor design choices, no additional feature design was done. A state is evaluated by first computing the potential resources of both players, that is, the total amount of own troops plus current territory and continent bonuses. These two values are then normalized by their total and mapped from the span (0, 1) to between (-1, 1). Rollout length was set to eight turns to maintain some of the random characteristics without playing for too long.

4.2.3 Network design

The network was designed as a multi-input, multi-output model with one policy for each game decision and a value head estimating the current player's chance to win. Five policy heads were used, three for the different phases and two additional for deciding to attack (or fortify) or not, matching the hierarchical structure of the tree search. The outputs were defined as follows:

- place policy, size 1x126,
- attack policy, size 1x165,
- fortify policy, size 1x3444,

- decide attack policy, size 1x2,
- decide fortify policy, size 1x2,
- value head, size 1x1.

The attack policy represents all combinations of attack source and targets that follow from the neighbouring relations of the map.

Input representation

The state representation at the input was split into four variables, one for the state distribution of troops and territory rulers, one for the cards of each player, and then two phase specific variables. The input also included five additional variables, each representing a one-hot encoded mask over the currently legal actions at the policy outputs.

The previous networks of AlphaZero and EXIT game applications have all represented the state as a two-dimensional image of multiple layers, mapping a position on the board to a spatially corresponding position in the image. This approach is however not suited for Risk as the board is a graph of territories and their neighbour connections rather than a grid. Since the graph (map) is static throughout the game and to simplify design, the state was modeled as a flat representation of size 1x126 where index

- 0 - 41 is the distribution of troops for the opponent player,
- 42-83 is the distribution of troops for the current player,
- 84-125 is the distribution of troops for the neutral player.

All troop counts where normalized by the total amount of troops on the board. This representation is alternating to the view of the current player, alleviating the need for an additional variable or any post processing of value head output.

The card input variable was specified by a 1x8 representation where the first four indexes are the card counts of the opponent player and the last four are cards of the current player. Card counts were normalized by three as this is the maximum amount that can be held for any one type before being automatically used at the start of the next own turn.

The two phase specific variables were one for the current amount of available troops to place, normalized by total troops on the board, and one which specifies if a territory has been conquered previously in the attack phase, indicating if a card will be drawn when it ends.

The state input is thus fully defined by the nine variables

- state distribution, size 1x126,
- cards, size 1x8,
- available troops, size 1x1,
- attack successful, size 1x1,
- place policy mask, size 1x126,
- attack policy mask, size 1x165,
- fortify policy mask, size 1x3444,
- decide attack policy mask, size 1x2,
- decide fortify policy mask, size 1x2.

Layer configuration

The network was designed as a feed forward network with three fully connected hidden layers, taking the state distribution as only input. Each hidden layer is followed by batch normalization and a dropout layer with parameter set to 0.5. After the hidden layers, the card input is concatenated and then this information is passed to the different output heads.

For the place policy head, the available troops input is first added by concatenation, then follows a fully connected layer of 126 nodes, matching the specified policy size. Before applying softmax activation to compute the policy, a lambda layer is used to filter illegal actions according to the place policy mask included at input.

The attack and fortify policy heads have the same structure except that the attack head instead adds the attack successful input and the fortify head doesn't add any inputs more than the cards.

The decide attack and decide fortify heads both take the same input as their respective action policy heads and have two fully connected layers, where the first is followed by batch normalization and dropout.

The value head concatenates both of the phase specific variables for a full representation of the state information and passes this through a fully connected layer, batch normalization, dropout and then a final fully connected layer with one node and $tanh$ activation function to get the correct output interval.

All of the fully connected layers at the output use L2 regularization set to 0.01.

4.3 Learning process

The zero learning training process was based on the EXIT algorithm presented in Algorithm 1. This structure was chosen over AlphaZero because it is easier to work with and requires less computational resources. Modifications were made, taking some inspiration from the method of AlphaZero, and are detailed below.

4.3.1 Self-play data generation

Each algorithm loop iteration started with generating a new data set of expert self-play decisions along with the game outcomes. Games were played in parallel on a computational cluster using an apprentice exploration policy for fast play. Due to the amount of resources available, data set sizes were limited to between 12 and 14 thousand games per iteration.

The first iteration was played using an MCTS agent with 300 simulations as the fast apprentice policy and 10 000 simulations at the expert decision. Later iterations were then played by choosing actions with highest probability (greedy choice) from the policy of the latest network, the expert decision was computed using 10 000 MCTS simulations with the network integrated in the tree search selection formula. The network's decision of choosing to attack or fortify was done by sampling from the two option distribution rather than greedy choice. It was observed that the greedy policy would often attack until no further attacks were available, thus leaving the own territories with one troop which prohibits fortification.

The three different phases of Risk reduces the effective size of the training data set. As the expert is given a game state reached by the apprentice exploration policy, the expert will only compute target statistics for the policy of the current phase. Each data set is therefore a distribution over three types of target data points with their corresponding states at the input. This distribution was chosen as uniform such that all policies would receive equal amounts of training data. A self-play game is thus done by first randomizing which decision type to save and how far into the game it should be saved. The target statistics for the two additional policies of choosing to attack or fortify can be gathered for any attack of fortify phase state as they are done in the same tree search.

4.3.2 Training details

The networks were trained using the new data set of the self-play step as in the standard version of EXIT, no data aggregation or buffers were used, resulting in a new network with updated weights. Data sets were split into 90 % training and 10 % validation data. At training start, weights were set to those of the most recent network, the first training iteration used randomly initialized weights. The incremental progression of EXIT iterations could then be verified by monitoring the decrease of validation loss from training start to finish.

Loss functions were set to cross entropy loss for the policy heads (Equation 3.7) and mean-square-error loss (Equation 3.9) for the value head. For the optimizer, the Adam algorithm was chosen [21], using a learning rate of 0.001. Training was finished using early stopping with a patience of 10.

Training improvements were also explicitly monitored by playing games between networks. To estimate the policy performance of each new network, 200 games were played against all previous networks using greedy action selection across all five policies. The value head output behaviour was observed by playing a handful of games between networks. At the start of each turn, the value head estimate for the current player was computed, generating a plot of its estimates throughout each game. The manual cutoff function of the MCTS rollouts was used as reference plot.

4.3.3 Integration of network in tree search

The integration of the network in the tree search selection phase was done as a combination of how EXIT and AlphaZero work. The problem of the EXIT UCT version is that nodes are initialized with a value of infinity, forcing the algorithm to test all children at least once. For the large branching of Risk, this would put an upper limit on how deep the tree can search regardless of the network's quality. On the other hand, AlphaZero removes the exploration term of the original UCT formula which takes away the possibility to weigh the policy term against the exploration term when the network improves, as done by EXIT.

The resulting choice for Risk was to retain the original exploration term, adding the policy term of AlphaZero with the benefit of keeping weight parameters in the same range and integrating the value head output as a replacement to random rollouts when activated. This yields the UCT formula as

$$UCT(s,a) = \frac{r(s,a)}{n(s,a)} + c_b\sqrt{\frac{\log n(s)}{n(s,a)}} + c_1\sqrt{n(s)}\frac{\pi(a|s)}{n(s,a)+1} \tag{4.2}$$

following the notation of EXIT, also defined in subsection 3.2.1. Nodes were initialized without the first two terms, reducing the expression to $UCT(s,a) = c_1\sqrt{n(s)}\ \pi(a|s)$ as in AlphaZero. c_b and c_1 are constant weight parameters that were manually tuned to give a good balance between exploration and exploitation across the different decision types. The first data set generation (without network integration) used $c_b = 0.3$ and later iterations used $c_b = 0.5$ and $c_1 = 1.5$. The value head was activated, replacing random rollouts, after six training iterations.

4.4 Agent performance experiments

The zero learning process has three types of agents, network agents, a tree search agent, and zero learning agents which can be compared against each other.

Playing full games with zero learning agents were for the implementation time-consuming so one strong version was chosen. Selection was done by playing a tree search integrated with the first network (a zero learning agent) against tree searches with the other networks that were generated before the value head output was included in data generating experts during training. For each agent comparison, 480 matches were played and the tree search used 1000 simulations, the value head output was not included.

Performance of adding the value head to training was observed by playing the winner of the previous selection against zero learning agents using all networks of the value head training iterations. These new agents were played for 240 matches with the value head replacing rollouts and 480 matches without it activated as a reference, tree search simulations were kept at 1000.

The agent types were compared by playing a round-robin tournament between networks, tree search and the selected zero learning agent. Three networks were used, the first, the last of policy training, and the last of the training process with value head included. Action selection was set to greedy for all policies. The tree search and zero learning agents used 1000 simulations and UCT weight parameters c_b and c_1 were the same as during training. Each comparison was played for 240 games.

Chapter 5

Results

Results of this thesis were limited by the computational resources and time frame of the project.

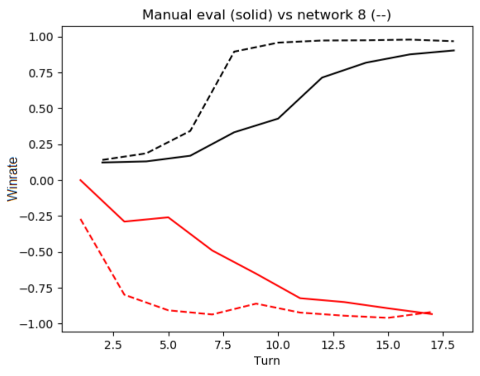
5.1 Network training

The overall network training process showed that networks had problems learning from the target data. One limiting factor was that the process could only be carried out for six iterations with the policy integrated into the tree search before validation losses were observed as flat during training. At this point, only the place policy had its third decimal decrease. The value head was then integrated as replacement to the random rollouts and training continued for two more iterations. Table 5.1 shows the results for playing the eight versions of network weights against each other. Firstly, the table shows how network 2 lost all games versus network 1 and itself, indicating that something was off with how it attacked. Secondly, the column for network 6 shows good results versus networks 1 through 4 compared to prior columns, which does not align with the indication of not learning due to validation losses being flat. The columns for network 7 & 8 then show an increase in win rate for row networks when compared to the columns of network 5 & 6, indicating that training on data influenced by the value head reduces policy performance.

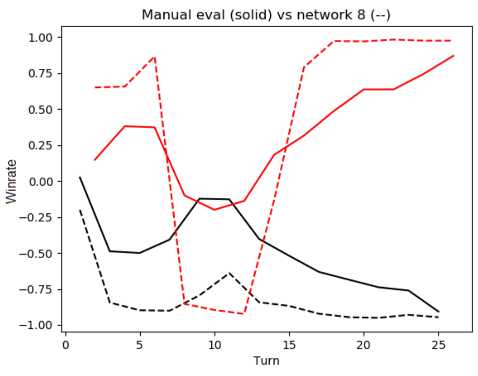
Table 5.1: Network match data, showing win-rates for row network. Main diagonal is the network playing itself and is a reference of variance from 0.5

Network	Net 1	Net 2	Net 3	Net 4	Net 5	Net 6	Net 7	Net 8
Net 1	*0.51*	1.0	0.53	0.57	0.44	0.36	0.48	0.49
Net 2	.	*1.0*	0.55	0.46	0.45	0.32	0.49	0.42
Net 3	.	.	*0.51*	0.42	0.41	0.35	0.42	0.48
Net 4	.	.	.	*0.52*	0.34	0.34	0.47	0.42
Net 5	.	.	.	.	*0.49*	0.55	0.53	0.58
Net 6	.	.	.	.	.	*0.44*	0.52	0.56
Net 7	.	.	.	.	.	.	*0.56*	0.46
Net 8	.	.	.	.	.	.	.	*0.48*

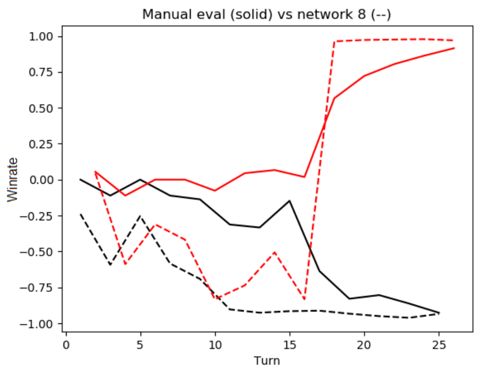
The value head behaviour tests of Figure 5.1 show further indications of poor performance from the value head. Estimates tend to be above or below the reference manual evaluation function and once games have been played to a state with characteristics recognisable by the network, the estimates are very close to either 1 or -1. Neither of these state estimation functions are the true estimation function but through the comparison, it can be stated that the network reach 1 or -1 too early and stay there or return inconsistent predictions between turns. Take for example Figure 5.1b where the game starts well for red, then evens out to around equal and after some turns, red wins. The network overestimates the lead, drops down to a certain loss once the player resources evens out, and then goes back to a certain win. The black player stays at estimating a loss but at least has a small increase where the state is equal.



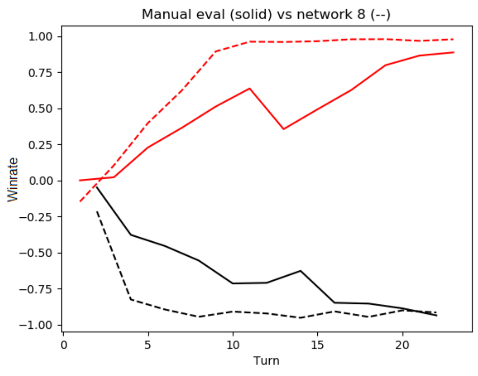
(a)



(b)



(c)



(d)

Figure 5.1: Comparison between value head estimates of network 8 and manual state evaluation function throughout games. Solid lines represent the manual function and dashed lines are the network, red and black are for the respective players.

5.2 Agent performance

The first agent performance test was to find the network with best weights for a zero learning agent. Due to the odd results of network 2 in Table 5.1 it was excluded from testing. Results of Table 5.2 show that using network 1 in the tree search marginally wins over the first three and then plays equal versus network 6. With these results, the first network was chosen as the best zero learning agent for its performance relative to training amount.

Table 5.2: Win rates for playing an EXIT agent using network 1 versus later networks of training steps without influence of value head.

EXIT vs. EXIT	EXIT1000 Net 3	EXIT1000 Net 4	EXIT1000 Net 5	EXIT1000 Net 6
EXIT1000 Net 1	0.569	0.566	0.542	0.495

Table 5.3 shows the results of playing EXIT1000 Net 1 against agents using the two networks that were influenced by the value head during training. Integration of the value head clearly decreases agent performance, which is why further training was stopped. It is also noteworthy that the agent marginally lost versus network 7 & 8 when only their policy was used, which is an improvement over Table 5.2.

Table 5.3: Win rates for playing an EXIT agent using network 1 versus agents integrated with the two networks that were influenced by the value head.

EXIT vs. EXIT + Value Head (VH)	EXIT1000 Net 7	EXIT1000 Net 8	EXIT1000 Net 7 + VH	EXIT1000 Net 8 + VH
EXIT1000 Net 1	0.434	0.450	0.762	0.892

The results for playing EXIT and MCTS agents versus networks are shown in Table 5.4. Both tree search methods outperform all networks and the differences between networks is only minor, with network 6 yielding the best result in both match-ups. For the comparison between EXIT1000 Net 1 and MCTS1000, the EXIT version won with a win rate of 72.5 %. The overall result is thus that a tree search guided by a policy network improves playing performance, subsequent networks had problems improving the performance any further, and all networks could be beaten by the original tree search when played on their own.

Table 5.4: Win rates for EXIT and MCTS agents versus different networks.

Tree vs. Network	Net 1	Net 6	Net 8
EXIT1000 Net 1	0.872	0.814	0.879
MCTS1000	0.755	0.679	0.719

Chapter 6

Discussion

Applying zero learning methods to Risk is a task of learning multiple policies and a state evaluation function by using a tree search to search through a heavily branching tree of multi-step turns. While the game structure presents a challenging task, it was expected that agent performance would reach human levels or better as prior examples of zero learning have done. Results does however show issues limiting continuation of the learning process. As these results are based on only a portion of the typical amount of data for zero learning, it is only possible to see indications of what is working and what is not rather than definitive conclusions about the method's applicability for Risk.

For the network policies, the results are a bit inconclusive. Both the test of playing networks against each other in Table 5.1 and the test with networks against tree searches in Table 5.4 suggest that network 6 is best. The first test does however have a lot of varying results within the table so it is uncertain to draw conclusions, even if 200 games were played in each match-up. Contrary to the suggestion, the comparison between different EXIT agents in Table 5.2, which is the intended use of networks for the project, show that network 1 is better than later networks up until network 6 where it is equal. The training steps might therefore have made the policies better but not enough to improve performance of zero learning agents further. What is clear, is that the network policies have learned to play the game. When played on their own against tree search methods they win up to at most about a third of matches and integration in tree search yields a 72 % win rate over standard tree search.

For the state estimation learning of the network's value head, the results are easier to interpret but also worse. The network fails to learn a function that accurately predicts the win rate for the state of the current player. The network behaves more like a classifier for which player will win rather than returning continuous estimates of how far each state is from winning or losing. While these "classifications" might look accurate, it is not the behaviour that is wanted. Following these observations, performance of zero learning agents was significantly reduced when the network replaced random rollouts (with cutoff) for search state evaluation.

There are multiple possible explanations for why only these results could be achieved.

Starting off with the network, it was designed using a flat state input representation and had only a few fully connected hidden layers for understanding the state before branching to the policies. With the low amount of observed learning, it is believed that the state input representation was not sufficient. It lacks the spatial representation of how territories are positioned on the board and their neighbouring relations. Although static throughout all games, these relations need to be implicitly learned. The alternative is to use a two-dimensional grid representation, like prior zero learning publications, but that would require the grid to be manually designed, translating the map layout to approximately according grid positions which would also include empty positions where there are no territories. As for the network size, more and or wider hidden layers could potentially have resulted in better learning but was set to a low amount such that the generatable amount of training data would suffice. Another design drawback was that only one EXIT learning process could be carried out, so the network layer configuration and hyper parameters had to be specified at the start from learning performance on the data set generated by the MCTS apprentice of the first iteration. The project was therefore limited from network redesign and most hyper parameter tuning.

The poor behaviour of the value head could likely have been improved by aggregating data sets before each training step. It is trained using targets of either 1 or -1 but is intended to predict the win rate between these values. With larger data sets, there are more input states, where some have similarities but their target values can be different. To reduce training loss during network optimization, the network would in these cases need to predict somewhere in the span between targets, thus pushing the learning in the right direction.

For the MCTS, the main subject of discussion is the design and potential performance issues of the UCT formula in the selection phase. With the current design, there are two hyper parameters, c_b and c_1, which are used to control the level of exploration and influence of the network policies. During the self-play data generation process, the MCTS should, using these parameters, distribute its search selections such that the resulting distribution corresponds to how good each action is believed to be. The problem with Risk is that there are multiple policies, where the unknown true distribution of each policy can have very different characteristics. Some decisions can have many legal actions but only a few that are good whereas some have few but similarly good actions. Using the same two parameters for all policies could therefore have limited the algorithm's flexibility to generate appropriate training data target distributions.

The decision of integrating the value head into the UCT formula as a replacement to random rollouts appeared as the most reasonable design for zero learning agents. A network has the theoretic ability of learning a strong state estimation function which should outperform information gained from random play outcomes. Using both, as in EXIT, defeats the purpose of achieving increased algorithm speed through only network state evaluation. In this case, the network did however not have good conditions for learning the evaluation function. Also, the comparison of performance between rollouts and value head of this project is not entirely fair. The rollouts were aided by a manually designed cutoff function, which, judging from Figure 5.1, seemed to yield good evaluations.

To finish the discussion, it is valuable to include a subjective performance analysis of agents, relating them to human performance, which is something that has not been in focus for formal quantification during the project. It was observed that the tree search method had clear signs of understanding the game, managing to place troops and attack along a coherent strategy. A portion of the decisions could be judged as sub-optimal, which under fair initialization allowed humans to regularly beat the agents. Agents usually placed all troops in one territory at the start of their turn and the fortification decision was mostly poor. The major challenge of agents however seemed to be the decision of stopping the attack phase at an appropriate time. They tended to be either passive, only doing some attacks, or very aggressive, attacking wherever possible. Getting this decision right had a large influence on playing performance.

Chapter 7

Conclusion

The conclusion of this thesis project is that zero learning methods, slightly aided by human knowledge and adaptations, could be used to train a neural network such that the network action policies improved playing performance of the Monte Carlo Tree Search decision making algorithm. Agent performance did however not improve with further iterations of network training and the network failed to learn a good scalar state evaluation function. In a comparison between policy networks and tree search agents, all iterations of networks were beaten in over two thirds of matches played.

7.1 Future work

Following the conclusion, zero learning can be applied for Risk but requires more research before superhuman performance is achieved. The network needs an input representation and layer configuration that is better suited for the layout and action space of the game. The learning process should be done using larger amounts of data or designed based on the process of AlphaZero, playing games with the MCTS rather than an apprentice and saving all decisions, in addition to improvement suggestions of the discussion chapter. Apart from these ideas about future research approaches, it is also relevant to investigate the behaviour of the MCTS and see how it is affected by the branching and multi-step turns deeper into the tree.

Bibliography

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall, 2010, pp. 17–18, 845–846.

[2] Murray Campbell, A. J. Hoane, and Feng-hsiung Hsu. “Deep Blue”. In: *Artificial Intelligence* 134 (2002), pp. 57–83. DOI: 10.1016/S0004-3702(01)00129-1.

[3] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[4] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (Oct. 2017), pp. 354–359. DOI: 10.1038/nature24270.

[5] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: (2017). arXiv: 1712.01815 [cs.AI].

[6] Zheng Tian Thomas Anthony and David Barber. “Thinking Fast and Slow with Deep Learning and Tree Search”. In: (2017). arXiv: 1705.08439 [cs.AI].

[7] James Goodman, Sebastian Risi, and Simon Lucas. “AI and Wargaming”. In: (2020). arXiv: 2009.08922 [cs.AI].

[8] Max Tegmark. *Open Letter on Autonomous Weapons*. July 2015. URL: https://futureoflife.org/open-letter-autonomous-weapons/.

[9] Michael Wolf. “An Intelligent Artificial Player for the Game of Risk”. Master’s Thesis, Darmstadt University of Technology, 2005.

[10] Jason A. Osborne. “Markov Chains for the RISK Board Game Revisited”. In: *Mathematics Magazine* 76 (2003), pp. 129–135.

[11] Manuela Lütolf. “A Learning AI for the game Risk usingthe TD()-Algorithm”. Bachelor’s Thesis, University of Basel, 2013.

[12] Richard Gibson, Neesha Desai, and Richard Zhao. “An Automated Technique for Drafting Territories in the Board Game Risk.” In: *Proceedings of the 6th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010* (Jan. 2010).

[13] Istvan Szita, Guillaume Chaslot, and Pieter Spronck. “Monte-Carlo Tree Search in Settlers of Catan”. In: *Ethical Theory and Moral Practice* 6048 (May 2009), pp. 21–32. DOI: 10.1007/978-3-642-12993-3_3.

[14] Glennn Moy and Slava Shekh. “The Application of AlphaZero to Wargaming”. In: *AI 2019: Advances in Artificial Intelligence* 11919 (Nov. 2019), pp. 3–14. DOI: 10.1007/978-3-030-35288-2_1.

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[16] Rémi Coulom. “Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search”. In: *Proceedings of the 5th international conference on Computers and games* 4630 (May 2006). DOI: 10.1007/978-3-540-75538-8_7.

[17] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning”. In: *Machine Learning: ECML* 2006 (Sept. 2006), pp. 282–293. DOI: 10.1007/11871842_29.

[18] Cameron Browne et al. “A Survey of Monte Carlo Tree Search Methods”. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4:1 (Mar. 2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[19] Mark H. M. Winands. “Monte-Carlo Tree Search in Board Games”. In: *Handbook of Digital Games and Entertainment Technologies*. Ed. by Ryohei Nakatsu, Matthias Rauterberg, and Paolo Ciancarini. United States: Springer, 2017, pp. 47–76. ISBN: 978-981-4560-49-8. DOI: 10.1007/978-981-4560-50-4_27.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[21] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: `http://arxiv.org/abs/1412.6980`.

TRITA -EECS-EX-2020:842