HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Mastering the Game of Abalone using Deep Reinforcement learning and Self-play

Ture Claußen

Bachelor thesis in "Applied computer science"

February 1, 2022

Author Ture Claußen
Matriculation number: 1531067
Hochschule Hannover
tu.cl@pm.me

First examiner: Prof. Dr. Adrian Pigors
Abteilung Informatik, Fakultät IV
Hochschule Hannover
adrian.pigors@hs-hannover.de

Second examiner: Prof. Dr. Ralf Bruns
Abteilung Informatik, Fakultät IV
Hochschule Hannover
ralf.bruns@hs-hannover.de

Declaration of authorship

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Hannover, February 1, 2022 Signature

Contents

Abstract

Explanation

1 Introduction

1.1 Background and Motivation

Board games are and have been a popular environment to test the capabilities of state-of-the-art artificial intelligence against human opponents. Many board games are widely known, making them a tangible measure of performance. The most prominent examples are the games of Chess and Go. For both, machines defeating the current best players has been representative of fundamental progress in computing.

IBM's "Deep Blue" defeated Gary Kasparov in 1996 [1] by utilizing search to look ahead into the game tree and deliberate on the next move. This approach is a prime example for symbolic AI approaches, "good-old-fashioned-AI" ("GOFAI") [2, p. 112f], which rely on logic and search on symbolic representations.

However, our ability to model the problem correctly and exhaustively severely limits these knowledge-based approaches. For example, in the case of Deep Blue, it requires us to encode expert knowledge about chess in a heuristic function to evaluate the board. Only then we can search for actions that maximize this function. Problems with large complexity would require tremendous efforts, which just become infeasible at a certain point.

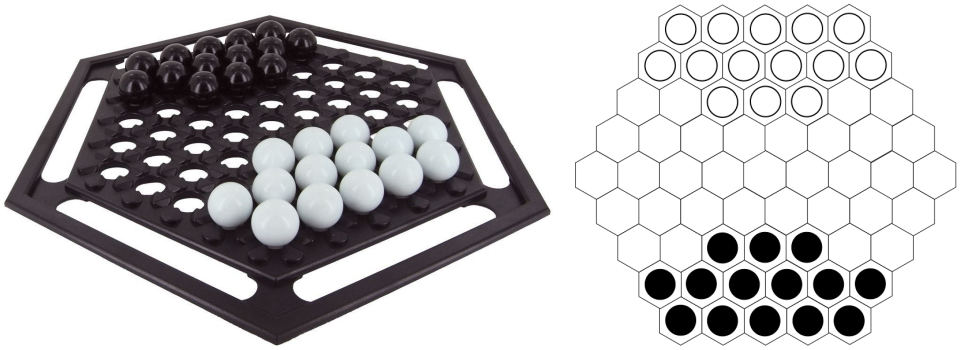
A different approach would be devising (general) methods to learn the necessary domain knowledge from scratch, *tabula rasa*. As Alan Turing put it:

> Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain. Presumably the child-brain is something like a note-book as one buys it from the stationers. Rather little mechanism, and lots of blank sheets. [...] Our hope is that there is so little mechanism in the child-brain that something like it can be easily programmed. [3]

The recent success of "AlphaGo" in 2016 against the long-time world champion Lee Sedol [4] in the game Go is a milestone that perfectly demonstrates this shift towards "bottom-up" or subsymbolic methods [5]. The increasing availability of computational power (and data) has enabled two subsymbolic methods to find considerable success in unclaimed

territory, such as computer vision or natural language processing. Namely, those are neural networks and (stochastic) gradient descent with backpropagation. Combined, they provide a general function approximator that can be trained in a process akin to the learning described by Turing.

In the case of Go, designing a powerful heuristic function was deemed impossible for humans. The team from DeepMind created AlphaGo using (deep) neural networks to learn an evaluation function based on an extensive database of expert moves. With the help of *reinforcement learning* (RL), they improved this network even further by letting it play against itself. They used this trained function to then perform a look-ahead search on the game tree more effectively [6]. Building on this success, DeepMind further improved the architecture. "AlphaGo Zero" and the generalization "AlphaZero" learn without the help of the database of expert moves. AlphaZero's learning process exclusively relies on deep reinforcement learning and *self-play*. Nevertheless, it surpassed the performance of AlphaGo significantly. Since then the architecture has been applied to Chess, Shogi and Atari games. "MuZero" went even further by removing the last piece of human knowledge in the system: the rules of the game [7].



(a) The original physical board of the game [8]

(b) Note the hexagonal shape of the board and the fields. A marble can be moved in upto six directions.

Figure 1.1: The board of Abalone

At this point, our endeavor begins. The goal of this thesis is to apply the methods of AlphaZero to the game of Abalone. Abalone is a relatively young board game from 1987. The main variant is played by two players on a hexagonal board with 61 fields and 14 marbles for black and white, respectively. The game's goal is to push six of the opponent's marbles off the playing field.

Naturally, the question arises whether this goal is a worthwhile thing to do. Firstly, academic interest in Abalone has remained steady even though the game's popularity

declined. However, the application of learning methods to Abalone is less explored, especially regarding the novel techniques introduced by AlphaZero.

Secondly, further investigating the applicability of RL is a relevant topic. Along with supervised and unsupervised learning, reinforcement learning is one of three basic machine learning paradigms. Aside from (super-) human performance in games [9, 10, 11], RL continues to find more industry application in areas like improving data center cooling at Google [12] or content recommendation at Spotify [13] and Netflix [14]. A very recent example is the floor planning for Google's latest TPU chip, which was aided by a deep RL algorithm [15].

Additionally, RL finds application in robotic control. Through continuous trial and error learning robots learned to grasp [16, 17], poke [18] or to avoid obstacles [19]. The learning was either performed in a simulated digital environment or through physical interaction.

1.2 Research Goals

First, let us establish the main research questions that will guide us throughout this thesis.

The first goal is to apply the general framework of self-play learning outlined in "Mastering the game of Go without human knowledge" to the board game of Abalone. [6] The original paper gives clear instructions on the theoretical groundwork for the system but omits clear instructions for the implementation. There is no open-source code provided.

The second goal is to compare classical search-based methods to AlphaZero's deep reinforcement learning based on several criteria such as win/loss ratio and computational requirements.

1.3 Structure

To provide the theoretical knowledge for understanding AlphaZero, the chapter 2 describes fundamentals in artificial intelligence, game-playing algorithms, (deep) reinforcement learning. The structure also mirrors the historical development of the methods.

The chapter **??** uses some of the previously introduced knowledge to analyze Abalone and also provides insight into existing game-playing programs to gauge the state of the art.

The chapter 4 is about the concrete implementation AlphaZero's methods for Abalone. It encompasses considerations about third-party software, Abalone specific adaptations, architecture, and algorithms.

Based on this software, chapter 5 shows the experimental setup and the results for the implementation.

Lastly, in chapter 6 an evaluation of the results given and an outlook for the continuation of the work is given.

2 Background Theory

Before we move to the nuts and bolts of AlphaZero and the concrete implementation for Abalone, we should establish a general understanding of the problem. That includes building the necessary theoretical background in artificial intelligence in general, as well as insight into specialized knowledge such as deep reinforcement learning in particular.

2.1 Artificial Intelligence

The introduction has already foreshadowed how artificial intelligence has undergone a shift in its methodology. In the 1950s and 1960s, figures like Alan Turing and von Neumann laid the foundations for modern computers. These new machines sparked the idea that one could create programs with similar abilities to humans and other organisms. Researchers at that time assumed there was no "universal principle" behind intelligence and focused on reason and symbol manipulation. Therefore these methods were considered "strong techniques," methods that relied on general principles like learning were labeled "weak techniques ." Nowadays, the consensus in the field has reversed. [20, cf. p. 8f.]

2.1.1 Rational Agent

Any setting that involves artificial intelligence has an agent. Stemming from the Latin word *agere* meaning "to act," an agent is something that acts. As one expects an agent to take sensible or intelligent actions, the definition must be further qualified by calling the agent rational. A rational agent acts "to achieve the best outcome or, when there is uncertainty, the best expected outcome." [21, p. 36]

The agent exists in an environment that it perceives through sensors and takes actions through its actuators. The content of the sensor's output for one observation is referred to as *percept*: A cat uses eyes, ears, and other organs to perceive the world and its legs, claws, and so on to interact with the world. An autonomous car might use radar and cameras for acquiring information and steering and motors for navigation.

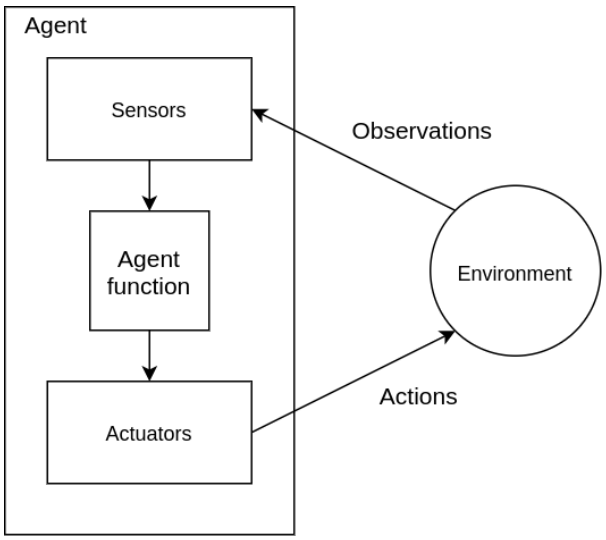


Figure 2.1: The agent-environment interaction loop [21, cf. p. 96]

Internally the agent might have some built-in knowledge about the world, such as rules on how the environment works. The *agent function* takes the entire history of percepts observed and this built-in knowledge and maps it to an action. A concrete implementation of this abstract function is called *agent program*. The agent program might just be a simple tabular mapping from percepts to actions or a complex algorithm with an additional model.

This abstract definition of an agent encompasses a simple program that plays Tic-Tac-Toe and very complex scenarios like a humanoid robot tasked to help in the household.

2.1.2 (Task) Environment

As we are trying to build an agent that tries to achieve some specified goal, we can consider its environment as a problem or *task* the agent tries to solve. The Tic-Tac-Toe program lives in a game world with simple rules while the humanoid robot interacts with physical reality. The environment might change either due to other influences or by the agent's actions. Putting together both agent and the environment, we see a loop of observing, deliberating, and finally taking an action as depicted in figure 2.1.

To specify the task environment, there are four core components illustrated for a machine classifying defective parts in a production line:

1. Performance measure: This might be the percentage of correctly broken parts (true positives) weighed against the number of incorrectly identified parts (false positives).

2. Environment: The conveyor belt and the parts
3. Actuators: An arm to push the parts to a different conveyor belt
4. Sensors: Possibly a camera, infrared sensors, etc.

The initial letters form the acronym PEAS (framework). Aside from the specification of the task environment, there are also a few categorizations of the properties of task environments that are extremely helpful for narrowing down the potential applicability of different classes of algorithms.

A fundamental property is the observability of the environment. If the environment is *fully observable*, the sensors detect all the information that is in any way relevant for taking an action. Conversely, if not all information can be observed, we call it *partial observability*. For example, in poker, the other player's cards and the upcoming cards cannot be seen but are highly relevant to the agent's actions. As the current board state of chess entirely comprises all information necessary to make a move, we can classify it as fully observable.

The values of the state of the environment and time can be categorized into discrete and continuous. For instance, an autonomous vehicle deals with continuous time and continuous states. The car's speed takes a smooth range of real values, and time can be meaningfully split into increasingly small intervals. Board games are entirely discrete. The set of all states is a finite collection of all (legal) configurations of the board and the gaming pieces. Time progresses on the basis of turns.

An agent's actions might also be *non-deterministic*. When dealing with systems of high complexity, the next state might not only depend on the previous state and the action taken. Other car drivers might take unexpected actions, or a comet hit the car.

It has to be taken into account if actions have consequences for future states. If each combination of percept and action is independent of each other, it is called *episodic* and *sequential* otherwise. If one had to classify a production line of circuit boards as either defective or functional, it would be an episodic environment. The classification of an individual board does not matter for the next one.

Another aspect of time is whether the environment changes while the agent takes time to deliberate on the next move. In a *dynamic* environment like the autonomous vehicle operates in, the environment changes continuously. By the time the car decides whether to go right to avoid collision with a wall, this decision might have already become obsolete. A turn-based game like chess is static as the board only changes when a move is made.

Lastly, an additional dimension to consider is the number of agents involved. The classification of circuit boards only involves one agent, whereas chess is a *multi-agent* environment. We also have to distinguish whether those multiple agents compete for

the performance measure. In most board games, one player's win is the other player's loss. In contrast, apart from an autonomous vehicle, the other vehicles all profit when it avoids a collision and vice versa. Therefore, they cooperate.

2.2 Adversarial Search

With an intricate understanding of agents' environment, the choice of algorithms one can employ can be narrowed down. In general, perfect information games can be solved by adversarial search algorithms. That means, in theory, one can find the optimal solution by considering all legal moves and the resulting game states up until terminal states (game-ending states). This is the *game tree*. It contains all possible transitions between board positions (nodes) through moves (edges). The root of the tree is the default start position. The *search tree* is potentially a subset of the game tree if not all paths are visited or the search is not started at the starting position. The theory behind this type of algorithm was already laid out as early as 1945 by Konrad Zuse's program generating legal chess moves [22], but was described most comprehensively by Claude Shannon in 1950 in "Programming a Computer for Playing Chess." [23]

2.2.1 Minimax Algorithm

Minimax assumes two roles: The minimizer (min) and the maximizer (max). The search starts from the current board state as the role of the maximizer and then alternates between the two. The result of the minimax search gives the maximum utility for the given state, assuming both players behave optimally. Let us define the functions [21, p. 303f.]

- `utility`(s, p) returns the utility or the payoff for the terminal state s seen from the perspective of player p. In the case of chess this might be $-1, 0$ and 1 for a loss, draw and a win.
- `is-terminal`(s) returns whether the given state s is a terminal state or not.
- `to-move`(s) returns the current player for the state s, either min or max.
- `result`(s, a) returns the resulting state if in state s and taking action a.
- `actions`(s) returns all legal moves for the given state s.

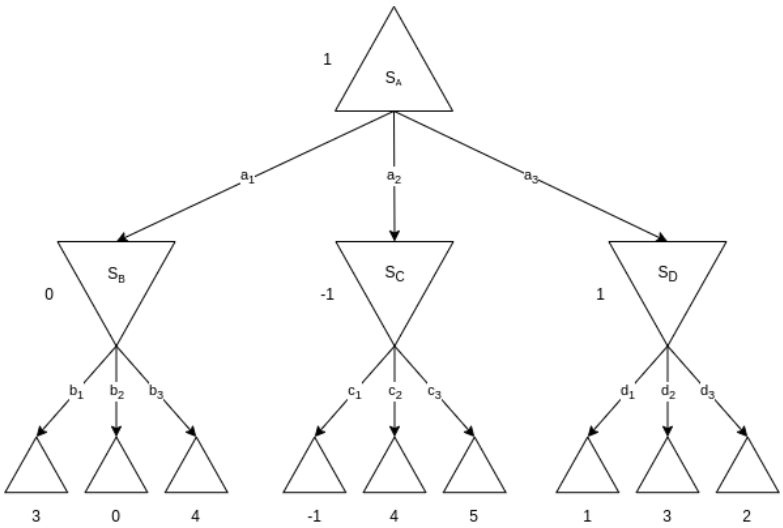


Figure 2.2: Minimax for a small search tree, resulting in an utility value of 1 [21, cf. p. 303]

$$\texttt{minimax}(s) = \begin{cases} \texttt{utility}(s, max) & \texttt{is-terminal}(s) \\ \texttt{max}_{a \in \texttt{actions}(s)} \texttt{minimax}(\texttt{result}(s, a)) & \texttt{to-move}(s) == max \\ \texttt{min}_{a \in \texttt{actions}(s)} \texttt{minimax}(\texttt{result}(s, a)) & \texttt{to-move}(s) == min \end{cases} \tag{2.1}$$

Putting this together, we can see in figure 2.2 a graphical representation of the search tree for an abstract example. The algorithm traverses down to a leaf node, evaluates its utility, and passes the value back to the parent node. Depending on whether it is a minimizer or a maximizer, it chooses the smallest or the largest value passed up by its children. Again, this value is passed up to the parent until the algorithm reaches the parent node, which is always a maximizer, yielding the maximum utility the maximizer can achieve given the opponent plays optimally.

2.2.2 Heuristic Functions

As the number of nodes of the game tree gets very large, the search on the tree usually does not reach terminal leaves that indicate a clear loss or win. The computational resources will get exhausted first. For example minimax has already visited $361 * 360 * 359 * 358 = 16,702,719,120$ nodes at a depth of $d = 4$ in the case of an average Go game.

Therefore, one has to limit the search to a computationally feasible depth and evaluate the intermediary result of a given transposition based on a so-called *heuristic function*.

This function replaces our previous $utility(s)$ for terminal states and is based on human knowledge. The function should give precise feedback on the quality of a state from the perspective of the given player. Most commonly, the heuristic function combines multiple state evaluations into a total numerical value [21, cf. p. 316]. For instance, in chess, the evaluations f_i could be be functions calculating different values such as:

- Material: First, each piece is assigned an integer that represents the piece's relative value (pawn = 1, bishop/knight = 3, rook = 5, and queen = 9). Then sum the values of the pieces left on the board for each player.
- Space: Count the squares controlled by each player.
- King safety: Check weaknesses in the king's position or count attacking pieces close to the king.
- Win and loss: As a more definitive measure to indicate whether the current state is a terminal state and hence a winning or losing state.

These evaluation functions can be combined into a linear combination of the form of:

$$h(s) = \omega_0 f_0(s) + ... + \omega_n f_n(s) \tag{2.2}$$

By applying different weights, ω_i to the functions f_i, the agent is given an incentive to prioritize particular behavior. If the win or loss function returns a value of either -1 or $+1$, one might combine it with a weight of 10,000 to make sure we choose winning states and avoid losing states above all. Armed with this heuristic function, one can find good moves with minimax search even in highly complex state spaces.

However, the problem with heuristic functions is that they require expert knowledge and much empirical testing to find a suitable heuristic. In some cases like Go, such a heuristic function might not be competitive with even moderate human players. In other cases, such as chess, this strategy is very powerful. As mentioned in the introduction, IBM's Deep Blue could beat the world's best player Gary Kasparov with heuristic-based adversarial search.

2.2.3 Alpha-beta Pruning

The minimax search can be improved markedly by using Alpha-beta-pruning. This method tries to eliminate unnecessary traversals down the search tree. In the best case, this leads to a reduction of nodes from $O(b^d)$ to $O(\sqrt{b^d})$.

The order in which the nodes are visited in minimax is similar to a graph traversal with depth-first search, meaning the algorithm descends until it finds a leaf node. This gives

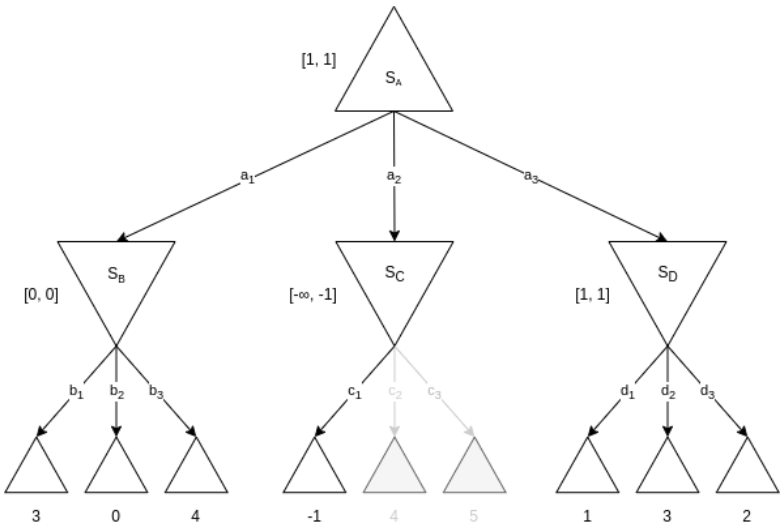


Figure 2.3: The previous example but with alpha beta pruning applied. The grayed out nodes indicate, that these in fact could be pruned from the tree [21, cf. p. 308]

information about the utility of that node and, consequently, part of the tree. An alpha value is kept for the minimum value the maximizer will receive and a beta value for the maximum value the minimizer will achieve. For instance, this lets one know if the minimizer already can choose a move worse than what can be achieved with another move. The algorithm does not descend further ($\alpha > \beta$).

Looking at the example in figure 2.3 will help illustrate this principle. The search revealed that choosing move a_1 will yield a utility of at least 0. Traversing down move a_2 the first leaf has a utility of -1. Hence, the minimizer will choose a move that is at most -1, which is already worse than the utility of 0. One need not look further at this part of the tree.

The example also shows us an important prerequisite for this method to work. The order in which each node is expanded matters as it decides how many nodes we can prune. Had the algorithm visited move c_3 and c_2 first, pruning would not have been possible. The best case of $O(\sqrt{b^d})$ is entirely dependent on this ordering. There are different ways of ranking the moves:

- *Killer move heuristic* prioritizes moves that are usually undoubtedly good, like taking a piece in chess.
- *Iterative deepening* Performs a minimax search only to a depth of one and uses the resulting values to rank the moves. Then searching one level deeper, we use this ranking for ordering the moves. Even though there is redundancy, it is made up for more than enough by pruning much more effectively.

Other improvements to the procedure are thinkable as well. Once one performs a search for a specific state, the resulting utility can be stored. If this position is encountered again, because of a different move sequence (transposition), the state's utility can be looked up in the *transposition table*.

Combined with alpha beta pruning, the minimax algorithm is a very efficient way of finding the optimal utility in an adversarial search situation. However, as mentioned before, in most games the utility of the terminal states cannot be used, because the search tree grows too quickly. By optimizing for a heuristic function the quality of play solely depends on this function. For games such as chess minimax has been very successful, because humans could devise meaningful heuristic functions. The chess engine stockfish has been the most successful computer player for a long time and is based on this algorithm (and many optimizations). [24, 25]

2.2.4 Monte Carlo Tree Search

For games like Go it was deemed impossible to find powerful heuristic functions, which makes the previous approach of minimax not a viable option. In addition, the initial position of a $19 \cdot 19$ Go board has a branching factor of 361 decreasing only by one for each stone placed. A method proposed in 2006 by Coulom [26] called Monte Carlo Tree search was more successful for Go. The main idea is to use simulations or *rollouts (or playouts)* to gain information on the quality of a state. To manage the complexity of the search tree more effectively the algorithm is *selective* in which parts of the tree are *expanded*. This ensures that resources are not wasted on unpromising moves.

In its purest form the simulations are performed randomly.This means for a state or node to be investigated, two random players take turns until a terminal state is reached. Kocsis and Szepesvári [27] showed that it does in fact converge to optimal play. For games with a high branching factor a large number of simulations is needed to get any meaningful information from the simulations. Hence a non-random *rollout policy* could be used instead. The policy may guide the moves taken in the simulation towards better moves. This might be as simple as favoring capturing moves or as we will see later neural networks.

The algorithm iteratively expands the search tree. For each iteration it runs through four steps:

- **Selection** is the process of deciding which node to consider next. Starting at the root node a node is selected until a leaf node is reached. This is the selected node. The selection of the nodes could be based on some probability distribution or use the knowledge gained over time.

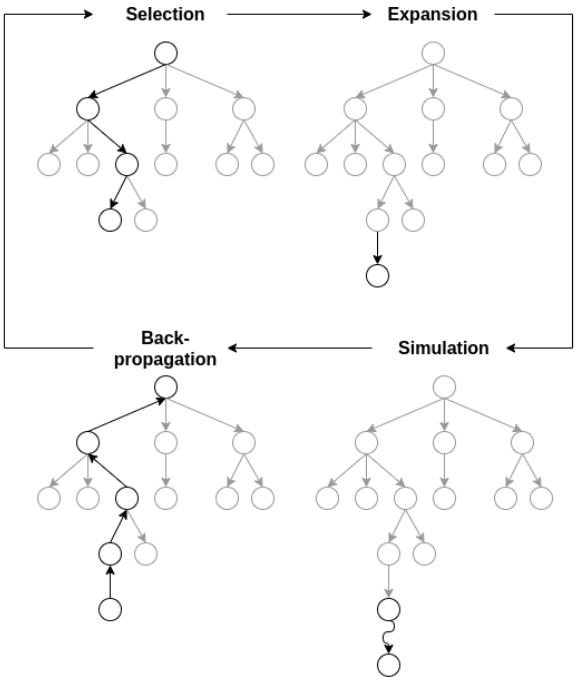


Figure 2.4: Monte Carlo tree search stages, cf. [28]

- **Expansion** is the step in which the selected node is expanded by appending a fresh child node.
- **Simulation** is as described before the step in which a a simulation is performed with a rollout policy starting from the state of the newly generated child node.We take t
- **Back-propagation** is the last step. The result of the simulation (utility) is taken and written to the node and parent nodes above until the root node is reached. Each node updates its cumulative utility $U(n)$ and the number of times it was visited $N(n)$. It is important to note, that this to be differentiated from backpropagation in the context of neural networks (c.f. 2.4.1).

The more this cycle is repeated, the more certainty is gained about the best move to take.

The development of MCTS led to significant improvements in performance of game-playing agents in the game of Go. The algorithm "Crazy Stone" from Coulom won the 10th KGS computer-Go tournament against competitors such as Indigo [29]. In the selection phase Crazy Stone estimates the probability of that move being better than the current best move and selects them according to that probability. The probability distribution over the moves is similar to the Gaussian distribution and the Boltzman equations. [26, p. 4]

Another idea for selection is the upper confidence bound formula (*UCB1*) [30], that weighs a node n is visited and how promising it is. Let us define [21, cf. p. 328]:

1. $\texttt{Parent}(n)$ returns the parent node of node n.
2. $N(n)$ returns the number of playouts performed on node n and its children.
3. $U(n)$ returns the cumulative utility of node n. For instance, this might be the number of wins for node n and its children.

$$UCB1(n) = \frac{U(n)}{N(n)} + C \cdot \sqrt{\frac{\log N(\texttt{Parent}(n))}{N(n)}} \tag{2.3}$$

The cumulative utility $U(n)$ is normalized by the number of times the node was visited $N(u)$. This helps favor moves that are either relatively unexplored and promising or have proven to be good over a larger set of nodes. It is also called the exploitation term. The additional term is called the exploration term. The more often a node is visited, the smaller this term gets, converging to 0 for large $N(n)$. The constant factor C is subject to some debate which value might be optimal, some choose $\sqrt{2}$. In general, this hints at another point of investigation: The problem of *exploration vs. exploitation* 2.3.2 that we will inspect more closely later.

The leading researcher behind AlphaGo and AlphaZero, David Silver, started his research on Go with MCTS. As early as 2006 he, and Sylvian Gelly, investigated optimizations to MCTS [31] for the game of Go. In 2011 they published a comprehensive paper [32] proposing the algorithm MoGo and evaluating different strategies to improve the effectiveness of MCTS in Go. Seeing that

> [...] professional Go players often play moves according to intuitive feelings that are hard to express or quantify. Precisely encoding their knowledge into machine-understandable rules has proven to be a dead-end: a classic example of the knowledge acquisition bottleneck. [32, p. 1873]

One of the ideas introduced is Rapid Action Value Estimation (RAVE). It was already mentioned how one could reuse information gathered for minimax through a transposition table. In a search tree one will encounter transpositions for that searches were already performed. RAVE allows to reuse experience gathered from simulations for related positions. A key property observed by Silver was that MoGo scales proportional to the amount of compute or rather number of simulations it can perform per turn as depicted in figure 2.5.

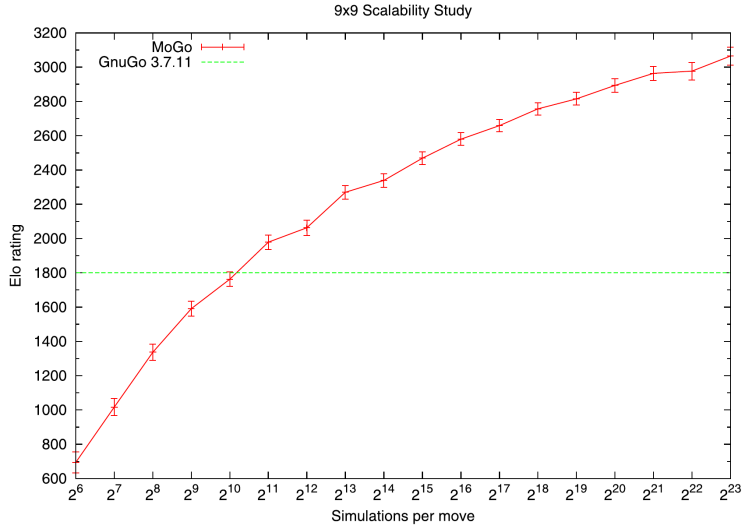


Figure 2.5: Elo rating of MoGo in relation to the computational resources granted to the algorithm [32, p. 1872]

2.3 Reinforcement Learning

The methods described until this point can be described as ”Good old fashioned AI”. They rely on search and human knowledge to perform adequately. Now we shift our focus to methods that use learning mechanisms to improve their play. With MCTS we’ve actually seen a kind of intermediary form of algorithm as it is ”simulating moves into the future, observing the outcome, and using the outcome to determine which moves are good ones is one kind of reinforcement learning.” [21, p. 331]

By devising a heuristic function, we basically told our agent what to do by indicating how a good position looks. The agent optimized its actions to be in a good position as described by the function. In reinforcement learning the agent learns what action to take through interaction. We don’t predefine which actions are to be taken. The agent tries to discover which actions yield the best *reward*. The numerical reward signal might come immediately, but e.g. in the case of chess the reward for actions taken comes much later by winning the game (or losing it). According to Sutton and Barto those are the key components of reinforcement learning: ”trial-and-error search and delayed reward”. [20, p. 1]

Reinforcement learning is not a specific solution or method, all methods for ”goal-directed agents interacting in an uncertain environment” [20, p. 3] are types of reinforcement learning. It is a general formalism that will help us reframe the problem of playing a board game (well) in a new light.

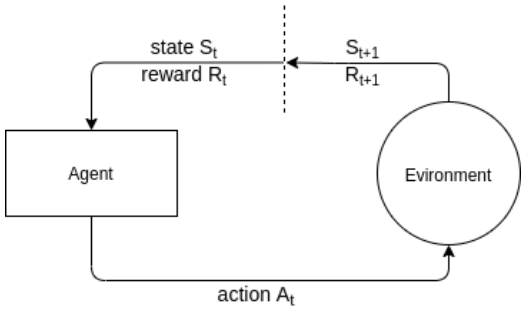


Figure 2.6: The markov decision process as agent-environment interaction loop [20, cf. p. 48]

2.3.1 Markov Decision Processes

Central to reinforcement learning is a formalism for the environment called Markov decision process (MDP). The MDP introduces some constraints and specifications that are useful for making precise statements and building further theory on it. If the constraints and properties of a MDP apply to an environment, one can resort to the algorithms and proofs already built on the basis of MDPs. At the same time the constraints are so loose, that many learning problems can be formulated as MDP.

Basis for the MDP is the Markov Chain. Markov Chains are used to describe sequential decision making. Each decision results in a state. The transition between states is stochastic, which means the following state depends on a probability. The transition probability does not depend on the history of previous states, it is independent.

Richard bellman extended Markov Chains by actions and rewards to derive MDPs [33, 34]. Actions cause transitions between states. They have long-term consequences, thus, effect future reward. The passage of time is divided into discrete time steps t at which the agent senses the state $S_t \in \mathcal{S}$ and then selects some action $A_t \in \mathcal{A}(S_t)$. Resulting in that action is some reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ that is received in the next timestep. This is very similar to the interaction loop previously discussed in figure 2.1. We can reframe this image with the new terminology of the MDP framework in figure 2.6.

For a *finite* MDP the sets of all actions $\mathcal{A}$ (*action space*), states $\mathcal{S}$ (*state space*) and rewards $\mathcal{R}$ have a finite amount of elements. The transition probabilities between the state s and the next state s' and its reward $r \in \mathcal{R}$ is given by the function p which essentially defines the decision process as a whole. A game of chess is deterministic, all transitions have a probability of 1 and these probabilities are defined by the rules of the game. However, the MDP is formulated to incorporate stochastic environments as well:

$$p(s', r|s, a) = P\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.4}$$

In a MDP, just as in a Markov Chain, "the state must include information about all aspects of the past agent environment interaction that make a difference for the future. If it does, then the state is said to have the *Markov property*". [20, p. 48] This means the transition probability p is independent of the history of preceding states.

From these properties of the MDP we can define the four central components of Reinforcement learning:

- **The reward signal** is the description of the agent's goal. A controller of a cooling system for a server farm might have the goal to minimize the energy spent for cooling while keeping the servers below a certain threshold. The reward signal then encompasses both of these subgoals. The controller has to maximize this reward. The *reward hypothesis* states that:

  > That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward). [20, p. 52]

  Therefore, we introduce the reward R_t and the goal G_t which is in the simplest case the sum of future rewards until the final time step T.

  $$G_t = R_{t+1} + R_{t+2} + ... + R_T \tag{2.5}$$

  We might also discount the future rewards by some factor $\gamma \in [0, 1]$ to account for the decrease in certainty we have about future rewards:

  $$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.6}$$

- **The policy** is a mapping between the perceived states to probabilities of selecting each possible action. For TicTacToe we might imagine a table that lists for all possible states to the agent's action. Due to the size of the state space 3.1 in more complex games this would not be feasible, therefore we used search processes as a policy so far.

  We denote the policy as the function π. It defines a probability distribution over all actions $a \in \mathcal{A}(s)$ for each $s \in \mathcal{S}$. $\pi(a|s)$ is probability at a given timestep t for the action to be $a = A_t$ under the condition that $s = S_t$.

- **The value function** is an estimation of the reward for the agent to be in a given state s. In other words, it is the expected value $\mathbb{E}$ for the goal G_t given the state s. As the reward for a state depends on what actions we take in the future, the value depends on the policy π defined above. Given that we are at timestep t and in state $s = S_t$, the *state-value function* v_π:

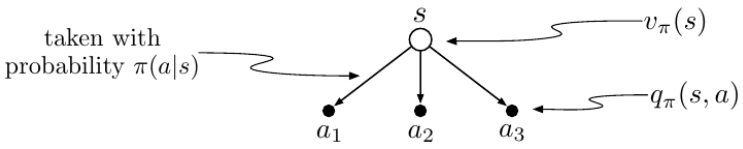


Figure 2.7: A visual explanation of policy, value and action-value [20, p. 62]

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right] \tag{2.7}$$

The heuristic function encountered previously was essentially a value function being used to guide the game-tree search.

Furthermore, we define the value of action a while being in state s under the policy π as the *action-value function* q:

$$q_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right] \tag{2.8}$$

The figure 2.7 shows how the three elements of policy, state-value and action-value relate to each other.

- **The model** helps to make predictions about how the environment might behave. For a given state and action it might return to the next state which aides in planning ahead. In the case of Abalone the model is the rules of the game, used by the function that returns all legal moves or the resulting board from a given board and a move.

The goal of reinforcement learning is to find (or approximate) a policy that maximizes the future reward for each action and state. For MDPs we define the *optimal policy* π_* as having higher or equal return than all other policies. Thus, the optimal policy maximizes the value function, resulting in the *optimal state-value function*:

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.9}$$

for all $s \in \mathcal{S}$. For instance, in chess the goal is to find a policy that approximates the utility returned by exhaustive search. Given such a policy one could evaluate the best possible outcome for any state s with the optimal state-value function.

An example will help illustrate these abstract specifications. We might view a pick-and-place robot arm through the lens of reinforcement learning. It has the task to place items from a pick-up location into a target area, cf. [35].

- The state space $\mathcal{S}$ is all possible combinations of the joint angles and velocities.
- The action space $\mathcal{A}$ is the amount of voltage that can be applied to each of the motors.
- The rewards $\mathcal{R}$ are $+100$ for each sucessfully placed item and a -1 reward is occurred each time a unit of energy is spent.
- Lastly, the state transitions $p(s', r|s, a)$ might be stochastic. The motors might have some imprecisions. An applied voltage might not always change the angle of a joint as expected.

In this scenario, the optimal policy π_* is to place the objects successfully with minimal energy expenditure. As mentioned in the introduction 1, researchers have sucessfully used RL methods to let robot arms learn, through interaction, how to pick and place items. The fundamentals described in this chapter are taken from [20, p. 47ff.], which goes more into detail about the details of MDPs and provides many additional examples.

2.3.2 Exploration vs. Exploitation

As the agent builds its knowledge while it is engaged in the environment it has to weigh *exploiting* the gathered knowledge for ensuring a safe short term reward or sacrificing it for *exploring* other actions that might turn out to bring higher rewards in the future. To illustrate this fundamental tradeoff in reinforcement learning, let us imagine a gambling machine with 10 levers, a 10-armed bandit. At each timestep t we have to decide which lever to pull and then we receive some reward R_t. Each lever has some unchanging (*stationary*) distribution over the rewards that is hidden from us. The distribution looks like the one given in figure 2.8.

To estimate the action-value of each lever, we sum the rewards received for that lever `total-reward`(a) and divide it by the number of times we've chosen the lever $N_t(a)$ at the timestep t.

$$Q_t(a) = \frac{\texttt{total-reward}(a)}{N_t(a)}$$

This is called the *sample-average* method. The simplest policy would be to just always choose the action with the largest sample-average Q_t a *greedy* policy.

$$A_t = \underset{a}{\operatorname{argmax}}\, Q_t(a)$$

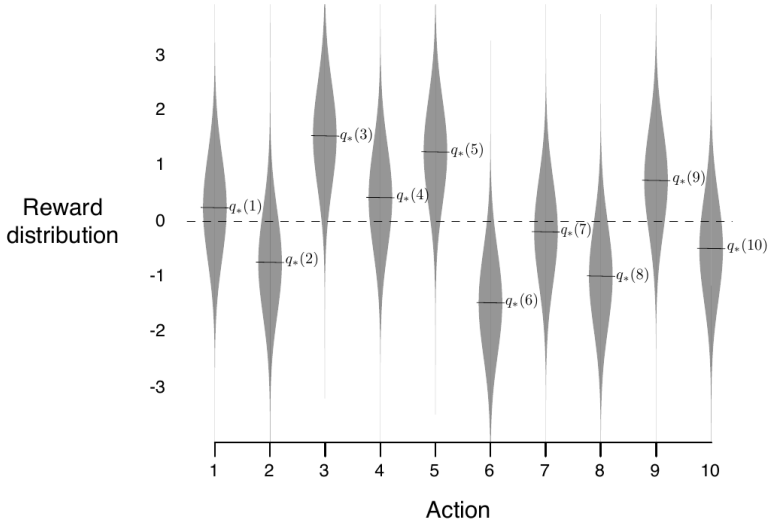


Figure 2.8: The reward distributions of a 10-armed bandit [20, p. 28]

First all action-values $Q_t(a)$ are initialized with the value 0.Initially, we might choose a random lever and that yields a reward of 0. As all Q_t are still 0 we choose another random lever giving us a reward of 1. Then, according to the greedy policy, we just repeatedly pull this lever indefinitely. The lever has the maximum action-value. We just get stuck on exploiting the small knowledge we have gained. To allow for some exploration of other actions. we introduce a small probability ε of choosing a different (random) action: With a probability of $1 - \varepsilon$ the agent chooses the greedy action, with a probability ε a different action.

If we continue for an infinite number of times the sample average for each action is guaranteed to converge to q_*. Each action is sampled enough to estimate its stationary distribution. We might also let the ϵ decay over time to ensure we exploit the optimal lever eventually. This shows how we have to carefully consider what knowledge we have and how we plan to expand it further. This also indicates the difficulty that is introduced when the problem is not stationary, which creates the necessity of continuous exploration.

2.4 Deep Reinforcement Learning

As reinforcement learning is a very general framework things like the value function $v(s)$ are just left as an abstract function. There are many different methods that build on the foundation of MDPs and fill these abstract functions with concrete instructions. One such method is deep RL, where the RL is combined with deep neural networks. Deep neural networks are *general function approximators*. Hence, they could be used to approximate the optimal value function $v_*(s)$ to maximize rewards.

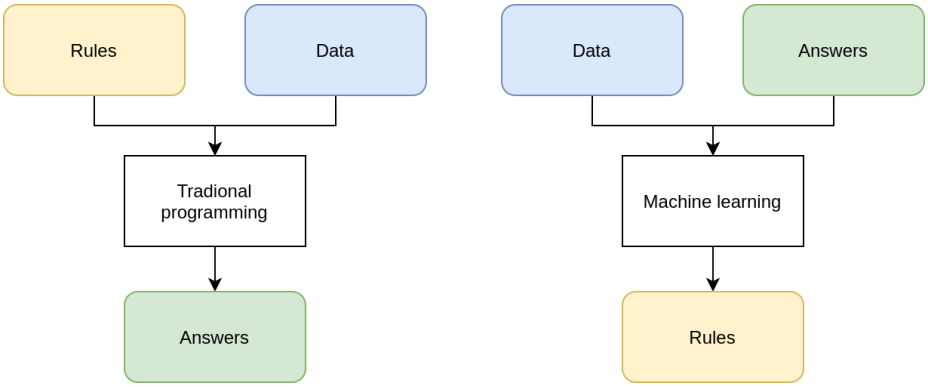


Figure 2.9: A fundamental shift in how we think about programming [36, cf. p. 5f.]

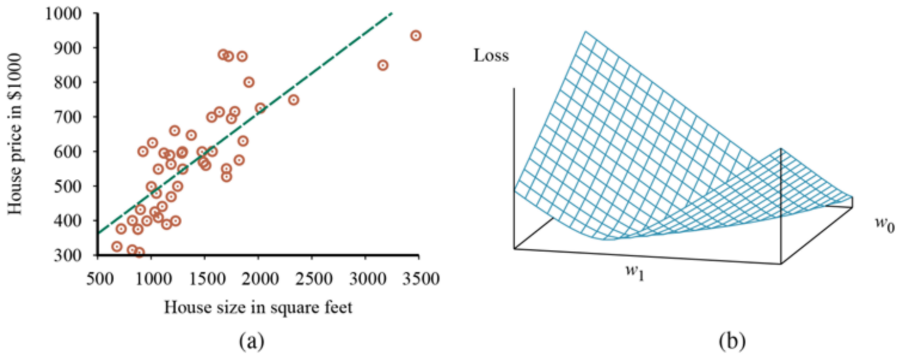


Figure 2.10: (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared-error loss: y = 0.232x + 246. (b) Plot of the loss function for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum. [21, p. 1251]

2.4.1 (Deep) Neural Networks

Neural networks are one specific machine learning method that has had large success in the recent past. Machine learning has been a paradigm shift in the way we think about building programs. In the classical development one uses the data and predefined rules as input for the development. This means we have to have intricate knowledge about the problem domain to produce the answers we want. For tasks like image classification this is a difficult process as it is very hard to think of patterns and conditions an image has to have, to find cats in them.

In machine learning we use the answers and the data as input to the development process and the rules are the output we produce. Figure 2.9 contrasts this change in programming.

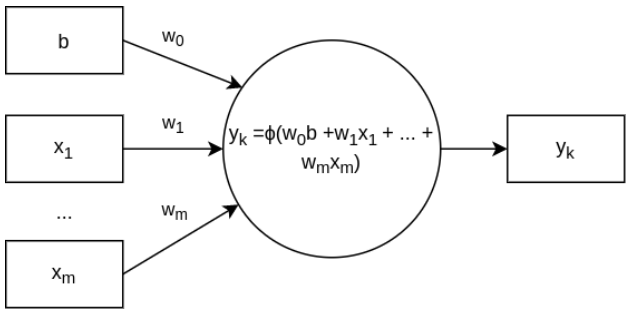


Figure 2.11: The fundamental idea behind neurons

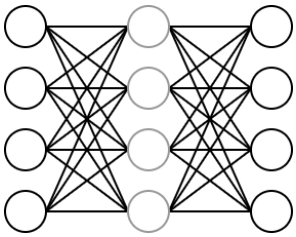


Figure 2.12: A small neural network arranged in layers. An input layer, a hidden layer and an output layer

Consider any linear function of the form:

$$f(x) = w_1 x + w_0,$$

with $x, w_1, w_0 \in \mathbb{R}$. We could use this function to make predictions about one dimensional input data. For example we might have a (training) dataset of living space of houses in m^2 X and their corresponding price Y. Each row X_i corresponds to the row Y_i. We could choose the bias w_0 and the *weight* w_1 such that the squared error for all rows i is minimal in a process we know as linear regression. When given new data our linear model can make predictions of the potential prices of the houses as described in figure 2.10.

Activation function An important component is not only the linear function but also the activation function: Before we pass the value of the function f(x) on, we apply some function ϕ. An activation function applied to a linear function is referred to as *neuron*. The activation function allows us to introduce a non-linearity, which in turn makes smaller networks of neurons able to perform much better on more complex tasks than just a simple linear activation. Common functions used are:

- ReLU: $\phi : max(0, f(x))$
- Sigmoid: $\phi : \frac{1}{1+e^{f(x)}}$

- Binary step: $\phi : \begin{cases} 0 & f(x) < 0 \\ 1 & f(x) \geq 0 \end{cases}$

This resulting basic neuron is depicted in figure 2.11. If we want to make more complex inferences than just the prices of houses, we can arrange multiple neurons into larger structures like chains or layers (2.12). To build such networks we need to generalize the linear function of the neuron for higher dimensional input:

$$y_k = \phi \left(\sum_{j=0}^{m} w_j x_j \right) \tag{2.10}$$

But this poses a problem. We can find globally optimal solutions for linear neurons with linear regression. However, this does not work for other activation functions. Moreover, as the size of the network grows, the computational cost of these methods grows too large. Moreover A different approach is necessary.

Loss functions In the context of linear regression the term *mean squared error* was already mentioned. It is the average squared difference between the predictions and the desired output:

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (X_i - Y_i)^2 \tag{2.11}$$

This is one type of *loss function*, which in general is a measure to describe the error we seek to *minimize* in an optimization process.

(Stochastic) gradient descent and backpropagation By utilizing the loss function we can feed an input into the network and measure for any permutation of the weights w_i how big the error of the network is. By measuring the loss of the training set X we can find out how well the current configuration of the network fits the data.

Let's go back to the problem of house price prediction. If we plot the MSE for the weights w_0 and w_1 of our single neuron the result is a parabola in figure 2.10. Ideally, we want to walk down into the valley where the error is minimal. So for any given combination of the weights we have to find out in which direction the downward slope maximal. The direction of the greatest change of a scalar function is called gradient, which is formalized as follows:

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \frac{\partial f}{\partial x_2}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$$

, where the point p is a point in the *parameter space*, e.g. as depicted in figure 2.10(b) for w_0 and w_1. For a single neuron we can use the methods learned in calculus to find the respective *partial derivatives* $\frac{\partial f}{\partial x_i}(p)$. By repeatedly calculating the gradient and updating the weights to move further in the direction of the downward slope, we describe the algorithm of gradient descent:

Algorithm 1 Gradient descent outline [21, p. 1253]

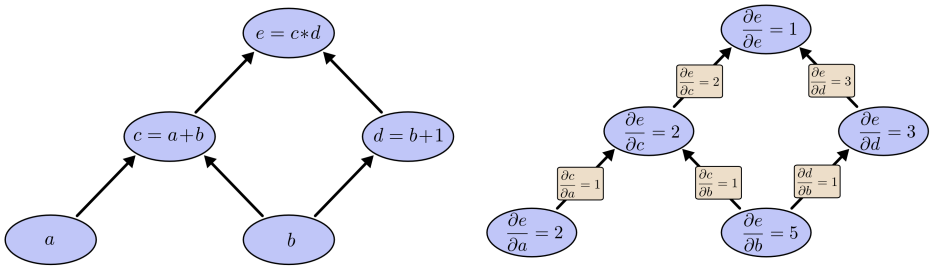
$w \leftarrow$ any point in the parameter space
while not converged **do**
  for each w_i in w **do**
    $w_i \leftarrow w_i - \alpha \frac{\partial f}{\partial w_i}(Loss(w))$
  end for
end while

The variable α for gradient descent describes the *learning rate*, so by how much we update the weights. As we have to differentiate not only a single neuron but a network of neurons, some constraints have to be defined to make derivation possible:

- All functions utilized by the neuron and the network (activation, linear combination of weights) have to be differentiable
- The network has to be a directed acyclic graph, thus, no loops etc.

This makes the neural networks one type of *computational graph*, which have the convenient property letting us find the gradients of the weights in respect to the loss by *backpropagation* by essentially applying the chain rule. Figure 2.13 shows an example of a computational graph and its derivation.

In summary, the neural network architecture describes a space of possible functions (or programs). The training data describes the desired output of the function. The loss function measures how much the output of the current configuration of the neural network differs from the desired output. Gradient descent adjusts the weights of the network in such a way that the ouput of the network fits the training data better: Gradient descent searches this space of functions for one that minimizes the error. Due



(a) Functions are nodes (b) Differentiate the graph by node e

Figure 2.13: An example of a computational graph [37]

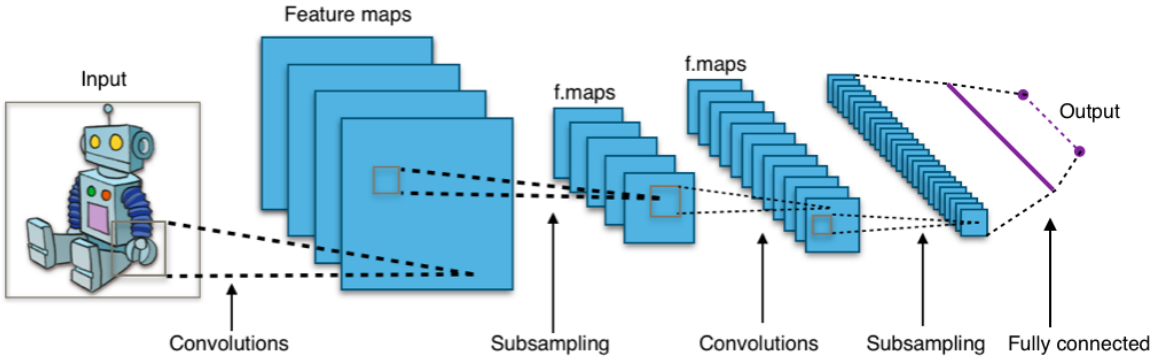


Figure 2.14: A typical CNN architecture [38]

to the nature of gradient descent, we are not guaranteed to find the globally optimal program. A different architecture might define a function space that contains more suitable functions than others.

2.4.2 Convolutional Neural Network

As already stated, the neural network or computational graph is not limited to one specific type of function as long as the function can be differentiated. As shown in 2.12 in its basic form, neural networks have a n-dimensional vector as input. If we have an input like an image, we would have to break down the information about the adjacency to fit the form of a vector.

Furthermore, the computational requirements for such an approach would grow rapidly. A fully connected first layer would need n^2 weights for an image of the size n. A small image of the size $256 \cdot 256$ would need $65,536$ weights for the first layer alone.

To achieve this we can utilize convolution. In general terms a "convolution is an integral that expresses the amount of overlap of one function g as it is shifted over another

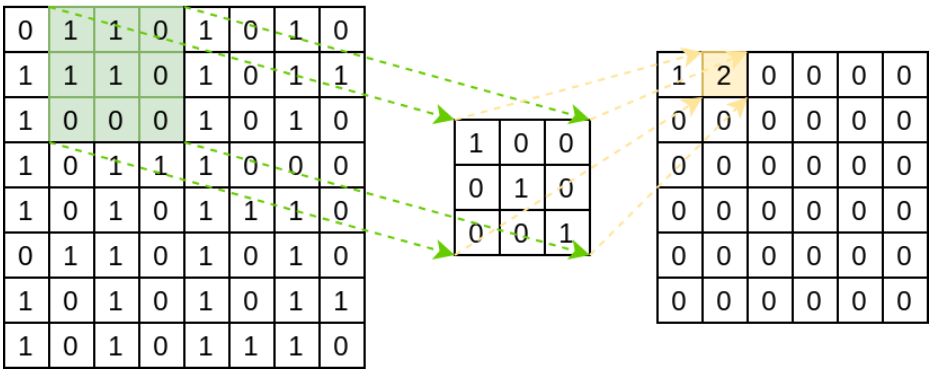


Figure 2.15: Convolution of a 8x8 black and white image with a 3x3 kernel, no padding and a stride of 1. [40, cf. p. 13]

function f." [39]

$$(f * g)(x) = \int_{-\infty}^{\infty} f(t)g(x-t)dt$$

This might seem very abstract but firstly, only the discrete case is relevant to us and secondly, only the finite case. This boils down the convolution to filtering F, which also comes up in image processing (technically this is the cross-correlation):

$$F(u,v) = \sum_x \sum_y I(u+x, v+y)H(x,y)$$

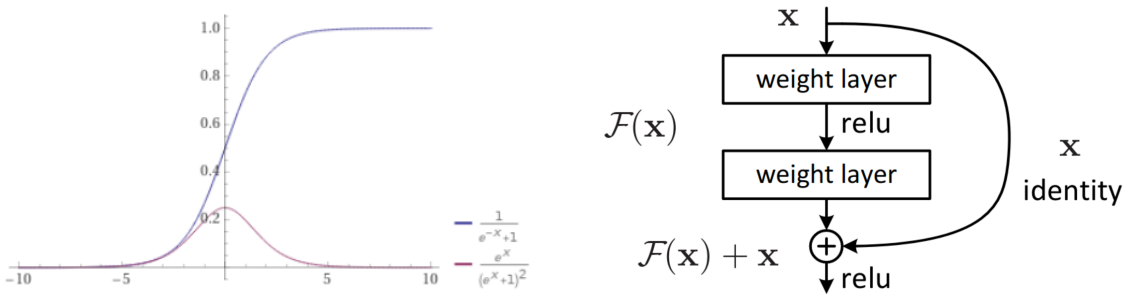
For a given coordinate u, v each value in the matrix H (also called *kernel*) is multiplied by the corresponding value of the image returned by function I. The figure 2.15 illustrates how a convolution would look on a black and white image. In image processing for example, we might use a Laplacian filter to extract certain features like edges. By using convolutions in a neural network we move away from these hand-crafted features by letting the parameters of the kernels be trained to fit the desired outcome. The result is a *convolutional neural network* (CNN) as depicted in figure 2.14. The properties (and advantages) of CNNs are:

- By combining multiple layers of convolutions the network can extract higher level abstractions of the image [41]. This also introduces sparsity between the layers, as a neuron is only connected to a part of the previous layer.
- The output of the network is equivariant to translation. This means that a pattern found in the corner produces the same magnitude of a response as a pattern found in the center of the image. However, the response is in a different location. [42].

The application of this method for image related tasks has become very popular. The first large scale application in 2012 by Krizhevsky et al. [43] vastly outperformed any previous methods. This approach is very relevant to the board game setting in Abalone as each board position is essentially an image. Patterns of marbles are relevant in different positions of the board just as different shapes and objects in different places in an image.

2.4.3 Residual Networks

By stacking many layers a network can fit more complex domains. Their size has earned them the name *deep neural networks*. But a central issue with increasing the size of the network is the problem of *vanishing gradients*. For example, as the arguments approach $-\infty$ and ∞, the sigmoid function is in it's limit 0 and 1 respectively. The derivative approaches 0 for both cases as depicted in figure 2.16 (a). That means in backpropagation, larger input values produce very small output values when passed through such a neuron. As the derivatives are multiplied during the backpropagation (chain rule) they become even smaller, thus vanishing as they further approach 0 with each added layer.



(a) Sigmoid function and its derivative [44]

(b) A residual block [45]

Figure 2.16: Motivation and implementation of residual networks

To mitigate this issue one could use different activation functions like the ReLU or use *residual connections*. In a residual (or skip) connection the value of the input before passing through the neuron is added with some weight to the output as shown in figure 2.16(b). A *residual block* is comprised of a group of layers and a residual connection as shown in figure 2.16. These units can be stacked together very "deeply" without issues.

This section on neural networks has only touched the raw fundamentals of neural networks very swiftly. Nevertheless it shows that the core ideas are quite simple (in hindsight of course), especially compared to the very comprehensive methods of classical AI. The theoretic foundations to these methods are actually quite old. With the unlocking of

more and more computational power and data researchers discovered the "unreasonable effectiveness of data" [46] and the combination of neural networks and gradient descent. To dive deeper the book "Artificial Intelligence: A Modern Approach" [21] offers a very good first impression.

2.4.4 AlphaGo

All the components we introduced so far constitute the knowledge necessary to understand AlphaGo and AlphaZero. Go was invented 2,500 years ago in China making it likely "the oldest continuously played board game" [47]. The complexity of Go's state space and search space surpasses that of chess significantly (cf. table 3.1). At the time when there was a lot of success in other games with minimax, many people saw Go as the most challenging board game [48]. The only successful method at the time was MCTS, for which we already saw two influential papers in the context of Go by David Silver.

By combining the reinforcement learning framework with MCTS and neural networks, David Silver et al. achieved the milestone of beating Lee Seedol. The program AlphaGo consists of three key components:

1. A rollout policy network p_π and a policy network p_σ trained by supervised learning
2. A policy network p_ρ trained with reinforcement learning and a value network v_θ derived from p_ρ
3. Look-ahead search using (asynchronous) Monte Carlo Tree Search guided by p_σ and v_θ. The rollouts for the search are performed by the rollout network p_π.

The first step is the supervised training based on the KGS Go Server dataset of 30 million positions [49, p. 485]. The rollout policy network p_π is a relatively small network that is supposed to provide quick rollouts (simulations). This is necessary because random rollouts for such complex games can produce very weak guidance for low numbers and also very long matches. This is supported by comparisons made between the prediction quality of 100 rollouts with a uniformly random policy versus with the network p_π in figure 2.17.

The policy network p_σ is trained in the same fashion except it is larger in size and thus is computationally more expensive. The policy network p_σ is used for the next step. The goal is to adjust "the policy towards the correct goal of winning games, rather than maximizing predictive accuracy" [49, p.484] by using policy gradient RL and self-play. Initially, the network p_ρ is initialized with structure and weights equivalent to p_σ. Then games with the current version of the policy network p_p are played against random previous versions of the network. At timestep t, for a mini-batch of n games, the results

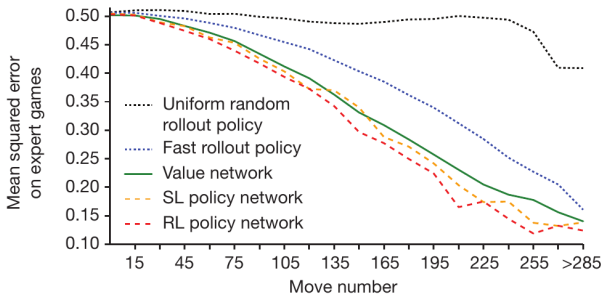


Figure 2.17: Comparison prediction quality of the different components [49]

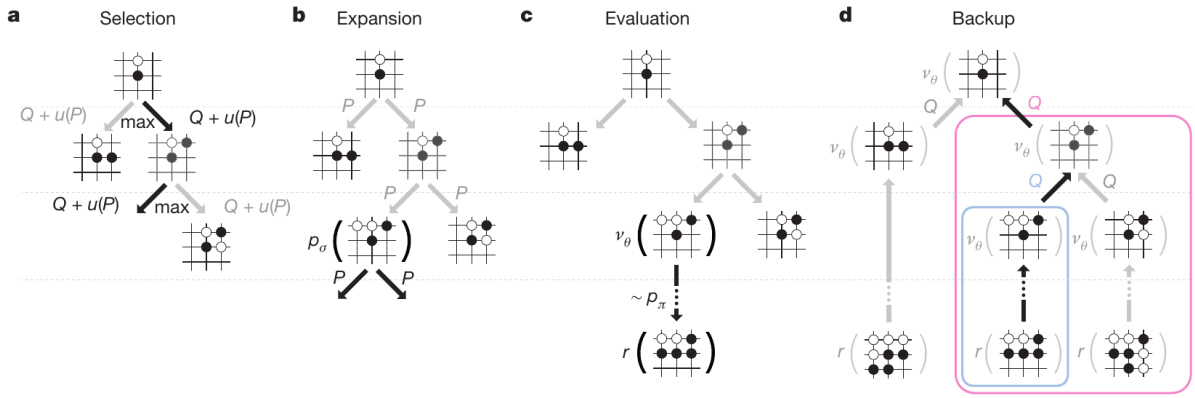


Figure 2.18: Monte Carlo Tree Search in AlphaGo. [49] Note the algorithm structure mentioned in figure 2.4 is maintained, but selection and simulation is more advanced

of the games are taken and the REINFORCE algorithm [50] is used to fit the weights of the networks to the outcome of the game z_t. The newly trained policy network p_ρ wins 80% of the games against p_σ [49, p. 485].

Lastly, p_ρ is used to train a neural network with weights θ for the evaluation function $v_\theta(s)$. This evaluation function approaches the state-value function $v_{p_\rho}(s)$, so the expected outcome of the game given the policy p_ρ: $v_\theta(s) \approx v_{p_\rho}(s)$. Figure 2.17 shows how v_θ approaches the same prediction accuracy as doing 15,000 rollouts with policy p_ρ.

During live play against an opponent, AlphaGo uses MCTS search. The search has performance versus just using the pure output of the policy network. The edges, or the pairs (s, a) of state s and action a, of the search tree store an action value $Q(s, a)$, a visit count $N(s, a)$ and a prior probability $P(s, a)$ (cf. figure 2.18).

- During the **selection** phase the next child is selected by taking the action with maximum action value. To encourage exploration in the beginning an additional term $u(s, a)$ is added to the action value Q. Each step t during selection an action

is selected by:

$$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t, a) + u(s_t, a)) \tag{2.12}$$

with $u(s_t, a)$ being a variant of the PUCT algorithm [51]. Just like the UCB1 2.3 it encourages exploration in the beginning and then increasingly prefers moves with high action value.

- When a leaf node is reached at timestep T, the state s_T is **expanded** by the policy network p_σ (Note that p_σ is being used instead of p_ρ, empirical analysis by the team showed this performs better). The resulting probability distribution over the actions is stored as prior probabilities: $P(s_T, a) = p_(a|s_T)$.
- The **evaluation** of the leaf node s_L is done by combining the evaluation function v_θ and the results of the rollouts z_L played with the rollout network p_π. Both elements are weighted by a mixing parameter λ:

$$V(s_L) = (1 - \lambda)v_\theta(s_L) + \lambda z_L \tag{2.13}$$

- Lastly, after each simulation the results are **backpropagated** through all visited edges of the root node. The count $N(s, a)$ is incremented and $Q(s, a)$ is updated by:

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{n} 1(s, a, i)V(s_L^i) \tag{2.14}$$

where n is the number of simulations "s_L^i is the leaf node from the ith simulation, and $1(s, a, i)$ indicates whether the edge was traversed during the ith simulation" [49, p. 529]. The action values stored on the edges of the search tree are the average of all evaluations of the child nodes.

The computational requirements for AlphaGo are significant. During training, the team utilized 50 GPUs. The supervised learning stage ran for three weeks regardless. During live play, they utilized a machine with 48 CPUs and 8 GPUs. A distributed variant with 1,202 CPUs and 176 GPUs was tested as well.

A more detailed introduction to the topic is provided by the nature article "Mastering the game of Go with deep neural networks and tree search". [49]

2.4.5 AlphaZero

A problem with AlphaGo's architecture is its complexity. There are many hyperparameters for the four different neural networks and many other moving parts. In a first step DeepMind simplified AlphaGo to AlphaGo Zero, which conveniently performs even better. In a next step the architecture was adapted to fit other games like chess and

shogi as well, called AlphaZero [52]. We will stick with Go to illustrate the concept, but refer to the algorithm as AlphaZero, even though it is a bit imprecise.

AlphaZero learns "tabula rasa" from a blank slate. Essentially, AlphaZero only uses a modified version of the self-play RL mechanism from AlphaGo. All other parts from the training pipeline are scrapped. The two main components of AlphaZero are:

1. A single neural network f_θ. Compared to AlphaGo, the neural network's architecture has been simplified by combining the value network and the policy network into one network with two heads. The network is only trained through self-play.
2. A simplified version of AlphaGo's MCTS, that does not use rollouts and only relies on the network f_θ to guide the search. The MCTS variant is not only used for live play but also self-play.

The network f_θ takes the current board state and the last seven boards as input. The position of the player's marbles is split into separate planes, 1 for a placed stone and 0 if there is none. Additionally, one feature plane indicates the player in turn, which is set to 1 for black's turn and 0 for white's turn. A total of 17 planes forms an input stack of the size $19 \cdot 19 \cdot 17$.

As shown by figure 2.19 (a), the network is divided into four types of blocks:

- The residual blocks arranged in the residual tower hold the main chunk of trainable parameters. Two sizes for the residual tower were tested, 19 and 39 blocks. The variant with 19 blocks performed better.
- The single convolutional block is composed of the same layers as the residual block, just without the skip connection.
- The value head outputs a scalar representing the value function.
- The policy head returns a probability distribution over all possible moves.

The team showed empirically [6, p. 9] that this *dueling-network* [53] architecture works significantly better for Go than other architectures. A dueling-network, as depicted in figure 2.21, has two "heads" for the state-value function and the actions or policy. Both heads share the underlying model layers. This improved performance for Atari-playing algorithms [53, p. 7] and also performed better than separate networks in AlphaGo Zero [6, p. 9].

At the start of training, the network is initialized with random weights. Each turn i during self-play, f_θ guides a MCTS, which produces a probability distribution π_t over all (legal) moves. The move a_i is selected according to the probabilities π. So instead of using the network f_θ directly to decide on the next move, MCTS produces an improved policy through repeated application of f_θ. The authors describe MCTS in this context

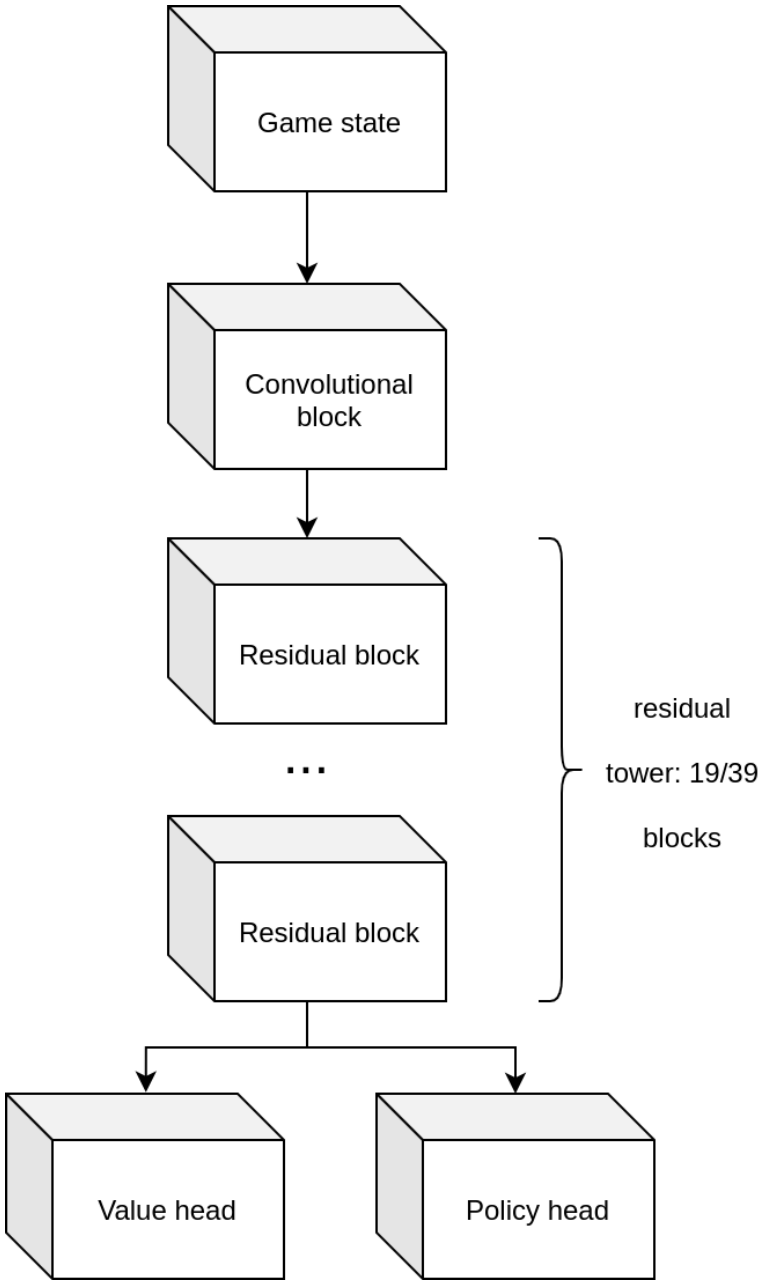
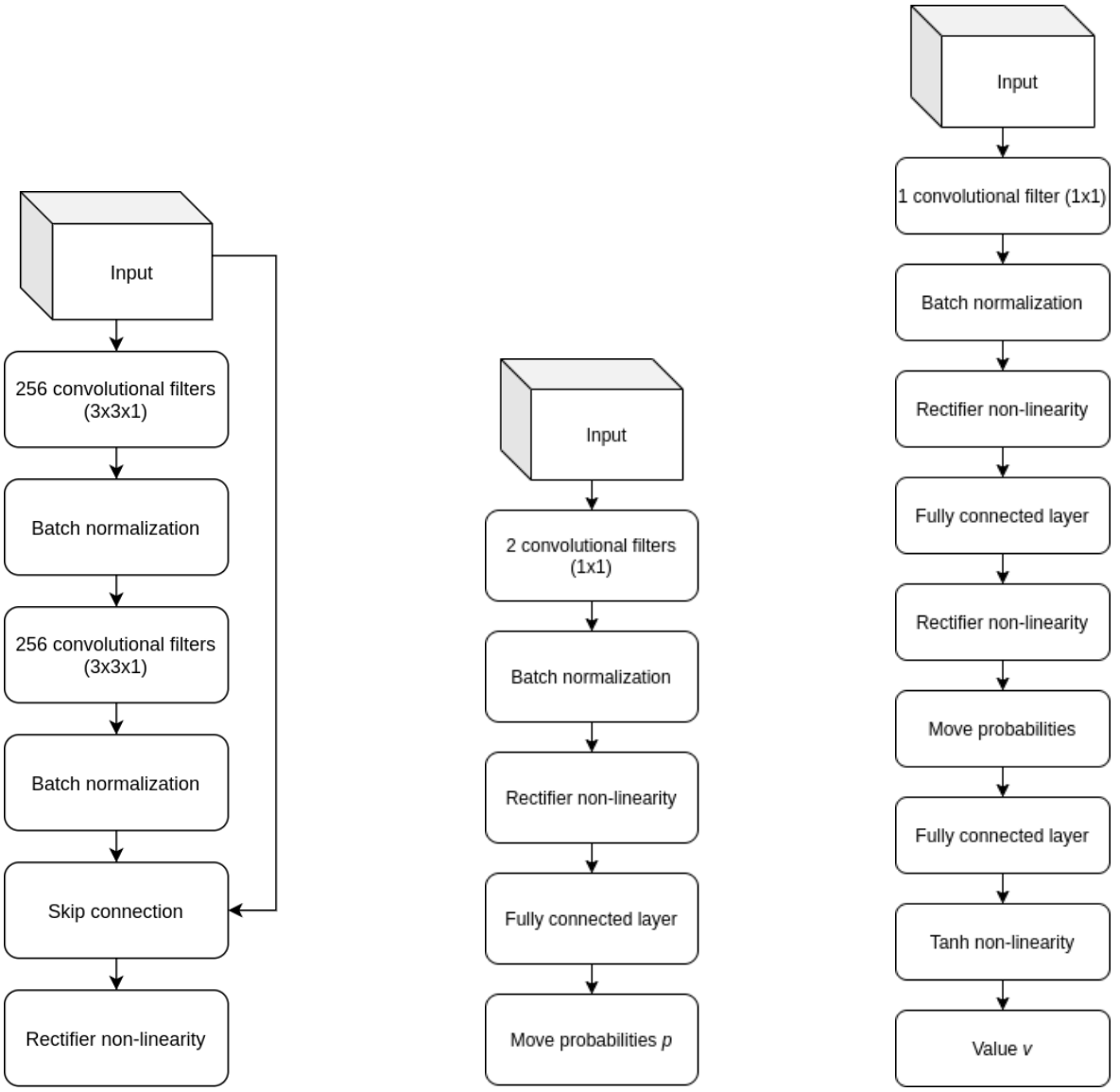


Figure 2.19: The general architecture of the network f_θ [6, cf. p. 27ff.]



(a) The layers of a residual block

(b) The layers of the value head

(c) The layers of the value head

Figure 2.20: The blocks of the neural network in detail, [6, cf. p. 27ff.]

as a *policy improvement* operator. The self-play games are played until a terminal state s_T is reached. The result of the game $z_T \in \{-1, +1\}$ is recorded for either a loss or a win. For each move during the game, the game's result z_t is recorded. Together with the board state s_t and the search probabilities π_t a tuple for one piece of experience is formed: (s_t, π_t, z_t). $z_t \mp z_T$ depends on the player's perspective who is in turn. The self-play is illustrated in figure 2.22 (a).

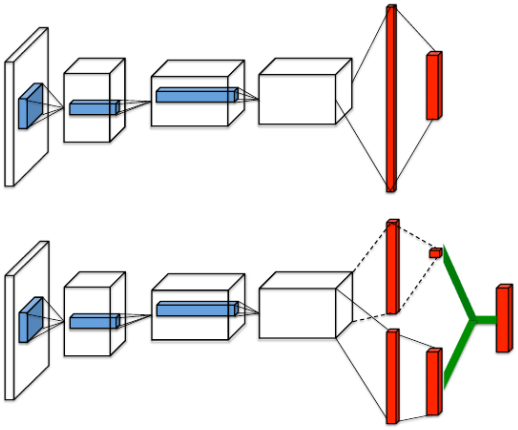
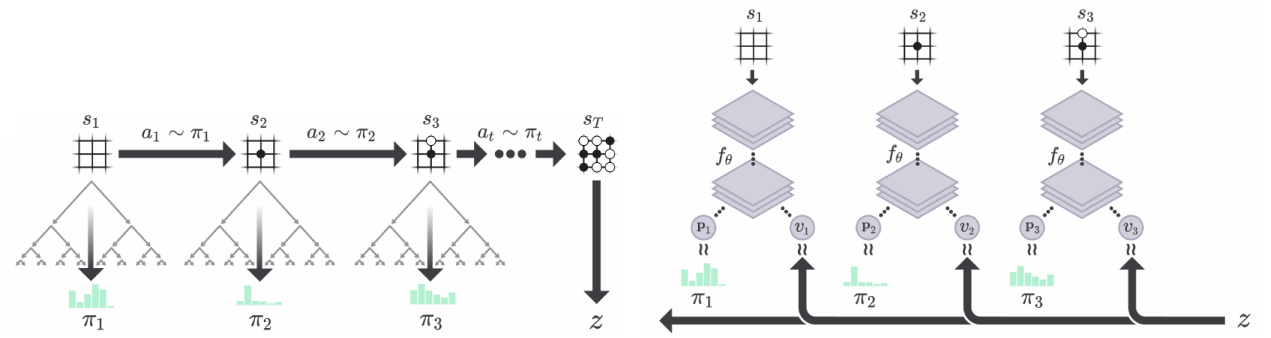


Figure 2.21: A comparison of a "single-stream" network (top) and a dueling network (bottom) [53]



(a) The self-play

(b) The training step with experience (s_i, π_i, z) from self-play

Figure 2.22: The self-play training pipeline of AlphaZero [6, p. 5]

This tuple is stored in an experience buffer used for training the next iteration of the network. In each iteration, mini-batches are sampled from the experience buffer. The network is fitted to match the probabilities π_t with the policy head and the game result z_t with the value head. The newly trained network is compared to the current best network in 400 games. If the new network wins more than a threshold of 55% of games, it is accepted. Self-play is performed with the new network from that point on. In total, $700,000$ mini-batches of size $2,048$ from 4.9 million games were sampled [6, p. 6].

To summarize, the self-play of AlphaZero uses the much stronger search policy generated by MCTS and improves the neural network by fitting the policy head to this search policy. The authors describe this as a projection of the search policy "back into the function space of the neural network" [6, p. 19]. Using the games' results, the search

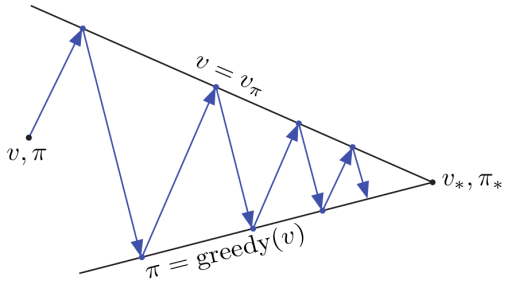


Figure 2.23: "Policy iteration consists of two simultaneous, interacting processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement)" [20, 86]

policy is evaluated, and the value head is fitted to match that evaluation. This procedure is an approximate variant of the dynamic programming method *policy iteration* which is depicted in figure 2.23.

The MCTS is very similar to the previous version in AlphaGo:

- During the selection phase, an action a_t is selected that has maximum action value and CPUCT until a leaf node s_L is reached.
- The network f_θ is evaluated, and the probabilities $P(s_L, a)$ are stored.
- Only the evaluation v is backed up. No rollouts are performed. The action values of the parent nodes are updated to the mean value of all v for that node.

The visit counts $N(s_0, a)$ in relation to the sum of all visits to other nodes is used to select the next move. The more a move was selected from the root node s_0, the better the network deemed the move and the following positions. A temperature parameter τ is considered to further incentivize exploration during the first moves of the self-play games:

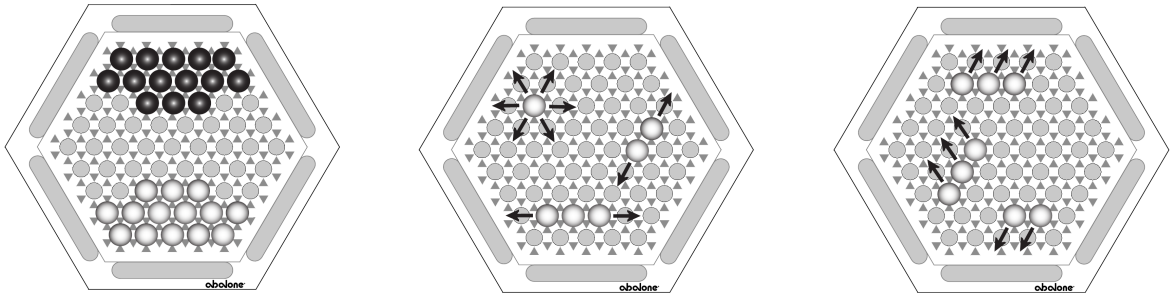
$$\pi(a|s_0) = \frac{N(s_0, a)^{\frac{1}{\tau}}}{\sum_b N(s_0, b)^{\frac{1}{\tau}}} \tag{2.15}$$

As τ approaches 0, the move with the maximum number is selected deterministically, and if $\tau = 1$, other moves are selected proportional to their probability. For self-play, τ was set to 1 for the first 30 moves and $\tau \to 0$ for the rest of a game and during the matches of the new and the current best network.

3 Abalone

Abalone was devised by Michel Lalet and Laurent Lévi. Even though it was created fairly recently, more than four million global sales have established Abalone as a classic game [54].

3.1 Rules



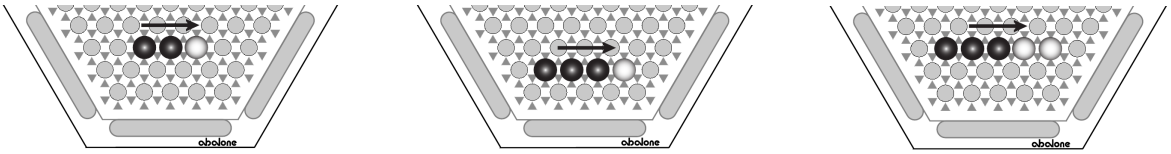
(a) Starting position (b) ”In-line” moves (c) ”Side-step” moves

Figure 3.1: Basic moves [55]

In the classical variant, each player places 14 marbles on opposing sides. Figure 3.1 (a) depicts the game's default starting position. Other starting positions like ”German daisy” and ”Belgian daisy” and four-player variants will not be considered. The player may move one, two, or three adjacent marbles in one of the six possible directions. The marbles have to move in the same direction and only move to a neighboring field. We differentiate between broadside or ”side-step” moves and ”in-line” moves, depending on how the chain of marbles moves relative to its direction. The difference is shown in figure 3.1 (b) and (c).

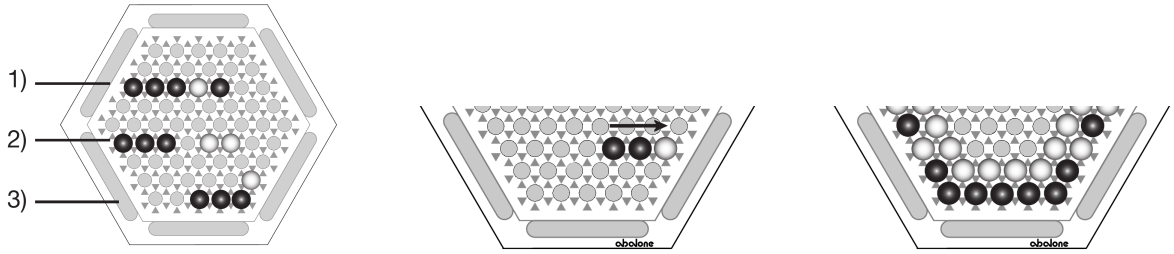
A move pushing the opponent's marbles is called ”sumito” and comes in three variations, as shown by figure 3.2. Essentially, the player has to push with superior numbers.

A sumito might be blocked by other marbles, as shown in figure 3.3 (a). In 1) the sumito by black is blocked by the black marble, in 2) there is a free space between the marbles, and 3) shows how a side-step cannot push a marble. Sumito moves are the only moves



(a) "2-push-1" sumito (b) "3-push-1" sumito (c) "3-push-2" sumito

Figure 3.2: Sumito positions allow pushing the opponent's marbles [55]



(a) Different blocking situations (b) Attacking position (c) No legal move available to black

Figure 3.3: Additional relevant board positions [55]

that allow for pushing the enemy's marbles. Therefore, they are the only attacking moves. Figure 3.3 (b) shows a situation in which we can push an enemy marble from the board. The player that pushes six of the opponent's marbles from the board has won. The basic ruleset does not account for a draw, but there are, in theory, positions like a stalemate in chess, where no move is possible for one player. In figure 3.3 (c), the black player is locked to the brink of the board and has no move available. Moreover, to force a more eventful game, games are often limited by time or the number of moves. Thus, a draw might occur when the number of marbles left on the board is equal for each player.

3.2 Task environment

Based on the PEAS framework, we can specify Abalone as a task environment and show the key components for the implementation of our agent. [21, p.107]

Performance measure Win/loss, number of moves, time to deliberate

Environment Digital playing board and rules of the game

Actuators Move marbles

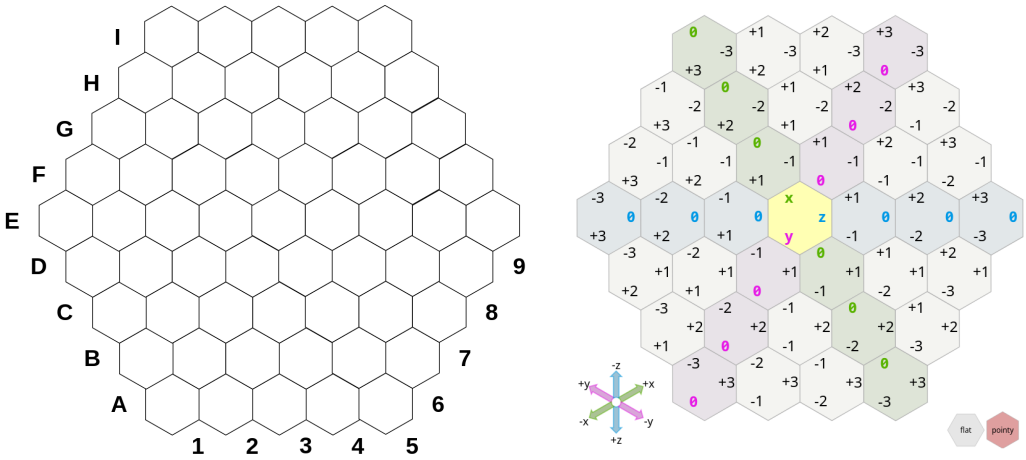
Sensors Position of marbles

Using the environment properties learned in 2.1.2 we can classify Abalone as **a fully observable, deterministic, two-agent, competitive, sequential, static, and discrete environment**. Another popular term for this type of environment is a *deterministic two-player turn-based perfect information zero-sum game.*

3.3 Board Representations

There are multiple possible coordinate systems for the hexagonal boards to address the marble positions universally. In Abalone, the most common way is to label the rows alphabetically from A-I starting at the bottom row. The ”columns” are labeled numerically from 1-9 as depicted in figure 3.4 (a).

Cube coordinates, as proposed by Patel [56], are a convenient way to represent hexagonal boards. The idea is to imagine a cube with a cartesian coordinate system originating from its center. At $x+y+z=0$ a diagonal plane is sliced out resulting in the coordinate system represented in figure 3.4 (b). It allows for the simpler application of a wide variety of formulas and algorithms like Manhattan distance, accessing neighbors, finding paths, etc., to the hexagonal board. Later, the cube coordinates will be utilized to calculate heuristics and to create symmetrical boards.



(a) The most common coordinate system for Abalone

(b) The cube coordinate system [56]

Figure 3.4: Different possibilities for addressing the fields of Abalone

A different representation is advantageous to store the board state in a program or feed it to a neural network: Transforming the board into a two-dimensional array of integers. The players’ marbles are represented as 1 for black, -1 for white, and 0 for a space. As suggested by ”towzeur” [57], shifting the upper part of the board to the right

creates an orthogonal basis. Therefore, the adjacency of the original hexagonal board is maintained.

```
   0 1 2 3 4 5 6 7 8
0        -1-1-1-1-1
1      -1-1-1-1-1-1
2    0 0-1-1-1 0 0
3  0 0 0 0 0 0 0 0 0
4  0 0 0 0 0 0 0 0 0
5  0 0 0 0 0 0 0 0
6  0 0 1 1 1 0 0
7  1 1 1 1 1 1
8  1 1 1 1 1
```

Figure 3.5: The matrix representation (the 0 values for the corners of the matrix are ignored, for better visibility)

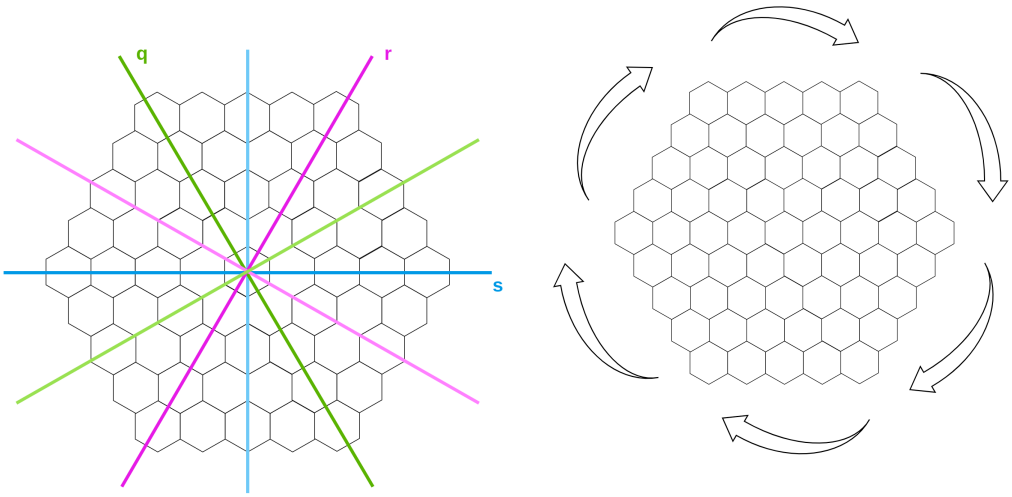
3.4 Move notation

There is no officially standardized notation for the moves of Abalone. The notation described by Aichholzer [58] has wide adoption in the papers investigated. To contribute to the proliferation of standards [59] we used a different notation that stems from the Abalone engine Abalone-BoAI [60].

Inline moves have the structure of "{MarbleCoordinate}{Direction}" and broadside moves the structure of "{MarbleCoordinate}{MarbleCoordinate}{Direction}". Additionally, the marble coordinates are ordered for broadside moves. The marble coordinates have the form outlined in figure 3.4 (a). The directions are always seen from the black player's starting position, north pointing straight in the direction of the white player (default position). This is the main difference to the notation described by Aichholzer, who uses a destination coordinate instead of a direction. The direction improves the readability and understandability of moves. The six directions are arranged just as in a compass, N for north, NW for north-west and so on. As follows, the regex for the notation:

```
([A-I][1-9]){1}([A-I][1-9]){0,1}((NE)|(E)|(SE)|(SE)|(SW)|(W)|(NW)){1}
```

For example, an inline move with the marble at A1 as trailing marble in the direction north-east would be denoted as A1NE. A broadside move of a row of marbles from C3 until C5 in the direction of north-west would be denoted as C3C5NW.



(a) Mirror axes (b) Rotations

Figure 3.6: The symmetries of the Abalone board

3.5 Symmetries

The board of Abalone has six rotational symmetries and additional six mirror axes. There are three main axes q, r, s that represent the coordinate basis shown in figure 3.4 (b). Each of these has an additional orthogonal axis resulting in six distinct axes.

The rotations depicted in figure 3.6 each describe a rotation by 60° clockwise.

3.6 Complexity

As Abalone has a finite amount of discrete states, we can make precise statements about its complexity, which one can describe in two relevant dimensions.

State space complexity The state space is the set of all possible states "the environment can be in." [21, p. 150] For Abalone, this means we have to consider all possible board configurations with different numbers of marbles present. Additionally, duplicates that arise from the symmetries of the board have to be removed. The following formula gives us a good upper bound:

$$\sum_{k=8}^{14} \sum_{m=9}^{14} \frac{61!}{k!(61-k)!} \cdot \frac{(61-k)!}{m!((61-k)-m)!} \tag{3.1}$$

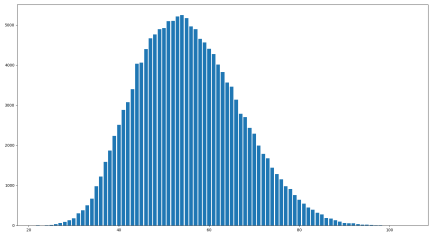


Figure 3.7: Counts of moves available a random player in 5 games

Game	state-space complexity (log)	game-tree complexity (log)
Tic-tac-toe	3	5
Reversi	28	58
Chess	46	123
Abalone	24	154
Go	172	360

Table 3.1: Abalone in comparison with other games [62]

Due to the board's symmetries,the results have to be divided by 12 which results in a total of $6 \cdot 10^{23}$ possible board configurations. [61, p. 4]

Game tree complexity Abalone's game tree is unbound and has an infinite height as actions might be taken repeatedly, forming loops. Therefore, Abalone's complexity is not determined by the game tree but is approximated by an average search tree. First, we consider the branching factor b, or the number of possible moves for any given state. This number greatly varies between different states. On average Abalone has $b = 60$ possible moves per state as measured in figure 3.7. The depth d of the tree depends on the number of turns per game. The average game takes approximately $d = 87$ turns. The number of nodes in a tree gives a measure of the complexity:

$$b^d \tag{3.2}$$

resulting in a total of $60^{87} \approx 5 \cdot 10^{154}$ nodes. [61]

As those numbers in isolation are hard to grasp, it is helpful to put Abalone's complexity in relation to other popular games. Its state-space complexity is on the same level as Reversi, whilst its game tree surpasses chess in complexity (c.f. table 3.1)

3.7 Existing game playing agents

3.7.1 Minimax

For all the previously discussed methods, game-playing agents based on minimax have been the most successful so far. Heuristic functions are quite similar to those mentioned for chess. Some of the more significant game situations optimized for are:

- Adjacency: As a majority of marbles is required to push the opponent's marbles and conversely an equal amount of marbles is needed to avoid being pushed, it can be assumed that keeping one's marbles grouped together is a good move.
- Distance to center: Marbles that are close to the brink of the board put them into danger of being attacked, wherefore it is generally good to place all of the marbles into the center of the board. For each player's marbles, we measure their distance from the center of the board as the smallest amount of moves it would take to reach the center (Manhattan distance).
- Number of marbles, formation break, single and double marble capturing danger, ... [63, p. 64]

There are multiple implementations of minimax for Abalone [62, 61, 64, 65], but software is only openly available for ABA-PRO by Tino Werner from 2002 [66]. There are a few mentions of (commercial) Abalone programs like RandomAba (Random Soft), AliAba, AbaloneNet mentioned by Lemmens [61, p. 7], but those have not been attainable through the internet. Even though ABA-PRO was not the strongest algorithm at the time, its availability has made it a frequent benchmark for other papers. Aside from previously mentioned optimizations for minimax like alpha-beta pruning, these programs use more advanced techniques like quiescence search, aspiration windows, and combined move ordering. A more recent publication from 2012 by Papadopoulos et al. claimed to have devised a more successful player. [63] Those claims could not be confirmed entirely in a recent reimplementation thesis by Michiel Verloop as Papadopoulos' does not describe the weights for the heuristic [67]. This reimplementation in Java [68] is also the reference for later benchmarks as it is open source [68] and thus allows programmatic interaction.

3.7.2 MCTS

The investigations into MCTS in the context of Abalone are pretty limited so far. Pascal Chorus has undertaken a comparison of the vanilla implementation against a heuristic agent, showing the dominance of the heuristic agent. [62] While in games like Go, we don't have loops in Abalone, random players can get stuck in very long games making

the results of simulations fragile signals. This approach does not work very well without a more sophisticated rollout policy.

3.7.3 Reinforcement Learning

"Abalearn" was the first learning-based approach to playing Abalone. [69] It was created in 2003 based on temporal difference learning (TD-learning), which is a reinforcement learning method. In the years before, TD-learning had been very successful for backgammon ("TD-Gammon") [70] and for Abalone, the authors achieved to draw ABA-PRO up to a search depth of five. An interesting feature of their approach is that they exclusively used self-play to train the algorithm. Moreover, they introduced a tunable mechanism for making the risk sensitivity of the algorithm dealing with the problem of the agent playing very passively or getting stuck in loops. Modern reinforcement learning methods like Q-learning have only been considered in a smaller project that achieved better than random performance. [71]

4 System Architecture

Well equipped with all the essential theoretical tools that we need, we can move on to implementing them for the game of Abalone. The first logical step would be to look at the existing software landscape to decide if we can utilize existing tools to speed up development.

4.1 Software

4.1.1 Deep Learning Library

Deep learning projects share many components. Most commonly, that is the declaration of the computational graph and the training of the graph. The libraries provide those components and bring significant optimizations and specialized code for hardware acceleration. Therefore, it is imperative to decide on a suitable library to speed up development by several orders of magnitude.

The most relevant libraries are Facebook's PyTorch and Google's TensorFlow. Aside from all differences between both libraries, the choice was guided by two practical reasons. Initially, we selected TensorFlow due to the included support for TPUs as Google granted this project free access to their Research Cloud [72]. At a later stage, it became clear that Google was unwilling to increase the CPU quota for the account, limiting the server to 8 cores which posed a significant problem for parallel execution.

In table 4.2 there is also a significant performance difference in the evaluation with the neural network (inference) during the MCTS. There are two ways in TensorFlow to perform inference, either through the `predict` function or by calling the `__call__` method of the *model* itself. In the implementation, the the states fed to the neural network are not batched. Inference is done for individual board states. The usage of the latter option is faster [73]. Nevertheless, when using the GPU, PyTorch is about five times faster as shown in table 4.2. As discussed later, this is the reason to pivot to PyTorch as a library.

HW	Framework	Neural net size	`predict`(s)	`__call__`(s)
CPU	tf	small	0.027s	0.011s
GPU	tf	small	0.024s	0.005s
CPU	tf	large	0.027s	0.015s
GPU	tf	large	0.025s	0.011s

Table 4.1: The average time ($n = 3,000$) taken to perform the feed-forward through the network for state s with either (`predict`(s)) or (`__call__`(s)) in tensorflow

HW	Framework	Neural net size	`predict`(s)	`search`(s)
CPU	tf	small	0.01s	0.016s
GPU	tf	small	0.005s	0.011s
CPU	tf	large	0.014s	0.02s
GPU	tf	large	0.011s	0.017s
CPU	PyTorch	small	0.005s	0.011s
GPU	PyTorch	small	0.001s	0.007s
CPU	PyTorch	large	0.005s	0.011s
GPU	PyTorch	large	0.002s	0.008s

Table 4.2: The average time ($n = 3,000$) taken to perform the feed-forward through the network for state s (`predict`(s)) and one iteration of MCTS (`search`(s))

Criterion	AlphaZero General	Deep MCTS
Parallel training	0	1
Parallel search	0	0
ML library agnostic	1	0
Game library agnostic	1	1
Verified performance	1	0
Simplicity	1	0
Sum	4	2

Table 4.3: A comparison of existing AlphaZero frameworks

4.1.2 Training Framework

There are existing frameworks that have implemented the system described in the AlphaZero paper in a more general and adaptable fashion. It has to be considered building on their foundation:

- AlphaZero General is a framework developed as a university project at Stanford originally for the game of Hex and Othello. [74, 75].
- Deep MCTS is a framework developed in the context of a Master's thesis [40, 76].

We considered a catalog of criteria to compare both options. Parallel training and parallel search measure whether the software implements the parallel training pipeline and MCTS-APV proposed by the AlphaZero paper. Moreover, we checked whether the frameworks are agnostic to the game and ML library. Lastly, it is relevant to investigate whether other users could verify the performance.

As shown by table 4.3, the winner by those criteria is AlphaZero General. The core advantages of the library are its simplicity and popularity. Deep MCTS makes use of a lot of abstractions and generics, which makes the code more professional and reusable. Nevertheless, this also poses significant difficulties in understanding. The application of AlphaZero General to multiple other games like Gobang, Santorini, Connect4, and more has shown its performance. The major drawback is the lack of a parallelized training pipeline.

4.1.3 Game Engine

There are multiple relevant implementations of game engines for Abalone. As the interfacing language for PyTorch is Python, it makes sense to restrict the engines to Python:

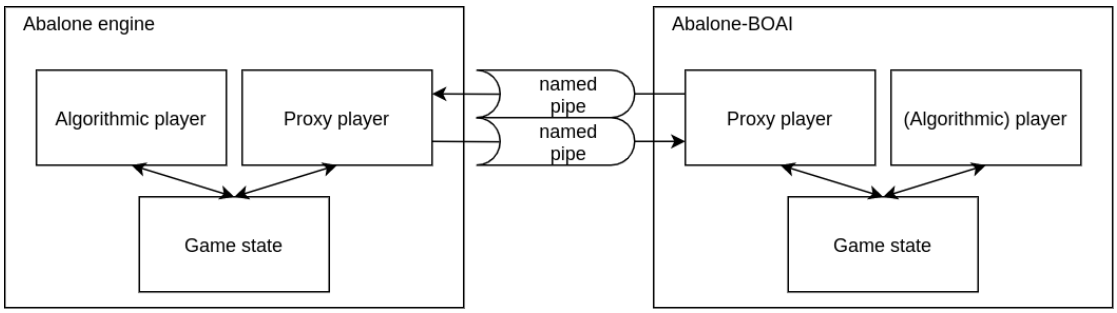


Figure 4.1: The inter process communication between the python game engine and the Java implementation of ABA-PRO

- Abalone-BoAI is a game engine specifically designed for interfacing with computational agents. It is straightforward and has been used for the precursor project of this thesis [60].
- Gym-abalone is tightly integrated with OpenAI's gym [77], potentially allowing agents to play other games as well. The API is suited for the reinforcement learning setting [57].

Two arguments lead to the decision of choosing Abalone-BOAI:

1. We already optimized it in a previous project [78], like the generation of legal moves.
2. AlphaZero General expects the game engine to be designed in a functional way. The API of the engine needs to be adapted to fit the interface of the training framework. Both engines are stateful, meaning that they rely on the class instance context to perform moves, check for legal moves, and other operations. For instance, the operations are supposed to be applied to the input matrix, and the modified matrix is then returned. Therefore, intricate knowledge of the engine is of advantage.

Due to those necessary modifications, we forked the original game engine to allow for more accessible packaging [79].

As mentioned in the section 3.7, Verloop's reimplementation [68] offers the simplest solution for interfacing with (one of) the strongest Abalone algorithms. In a small tournament, personal implementations of minimax in Python [78] were inferior. As the Java implementation is tightly coupled with the internal game engine, we decided to implement a proxy player for each game engine. Each engine is a separate process, and the proxy players relay the moves between the engines. The proxy players communicate through a named pipe as depicted in figure 4.1. The downside of this solution is the need to synchronize two separate game states and translate between the different move

notations. We used the notation introduced in section 3.4 to serialize the moves and added a translation function, to convert the intermediary notation to the internal format.

4.2 Neural Network

4.2.1 Dimensions

In order to apply the neural network architecture proposed by AlphaZero (cf. figure 2.19), two dimensions need to be changed:

1. The input tensor of size $19 \cdot 19 \cdot 17$ needs to be adjusted to fit the dimensions of the hexagonal board. The hexagonal board is transformed into an orthogonal base as shown in 3.5 such that it can be represented as a $9 \cdot 9$ matrix. The third dimension of size 17 can be reduced to 1 by removing the move history and only passing the current state. As mentioned in section 3, the official rules don't prohibit repitions. Adding logic to checking move sequences against the history would have introduced too much complexity and we would have had to change the function signatures of the framework.

   AlphaGo Zero represented the position of the players' pieces on separate planes. By representing the state as proposed in 3.3, there are no separate planes necessary. Additionally, the board is always passed in its canonical form to the network. The board states are always seen from black's perspective. If it is the black player's turn, the board remains unchanged. If it is white's turn, the board's colors are switched. From the perspective of the neural network, white (-1) is always the opponent. The in-turn player is represented as 1 and -1 for black and white. The matrix values can be multiplied with the in-turn player value to create the canonical form.

2. AlphaZero's output vector π of size 361 represents all possible positions a piece can be placed on the board. As Abalone has a much more complex move system, the size of this vector needs to match the number of all possible moves. We found the number of all possible moves by generating all moves programmatically. The generated moves were stored in a bijective map. Each move is assigned to exactly one index in π and each index is assigned exactly one move. The move represented as a string in the proposed notation above. Therefore, using the bijective map, a move's index can be found by its string representation and vice versa.

   As depicted in figure 4.2, the inline moves are generated by iterating over all fields of the board. For each field, all directions are tested for being a valid move. If that is the case, we check if the move is present in the bijective map. If not, the move is assigned the next index in *pi* and added to the map.

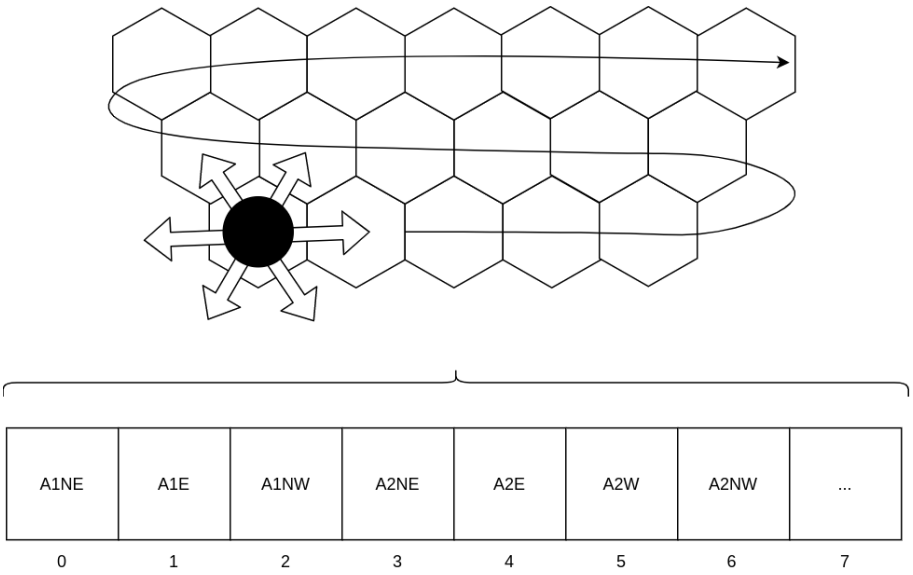


Figure 4.2: Generation of inline moves, starting at position A1 and direction NE

Figure 4.3 shows the generation of the broadside moves. The algorithm iterates each field. Marble lines of length two and three are generated in all (possible) directions for each field. For each line broadside in all directions, moves are checked for validity as the move might be out of bounds. The move direction of the marble line direction is ignored.

The resulting vector π has a length of 1452.

4.2.2 Architecture

4.3 Training Pipeline

4.3.1 Components

The training pipeline of AlphaZero General has five main components:

1. **Coach**: The main module that orchestrates the training process.
2. **Game**: Provides an abstract interface that generalizes to many types of board games. Functions like the generation of legal moves, creating a unique string representation of the board, and other functions need to be implemented.

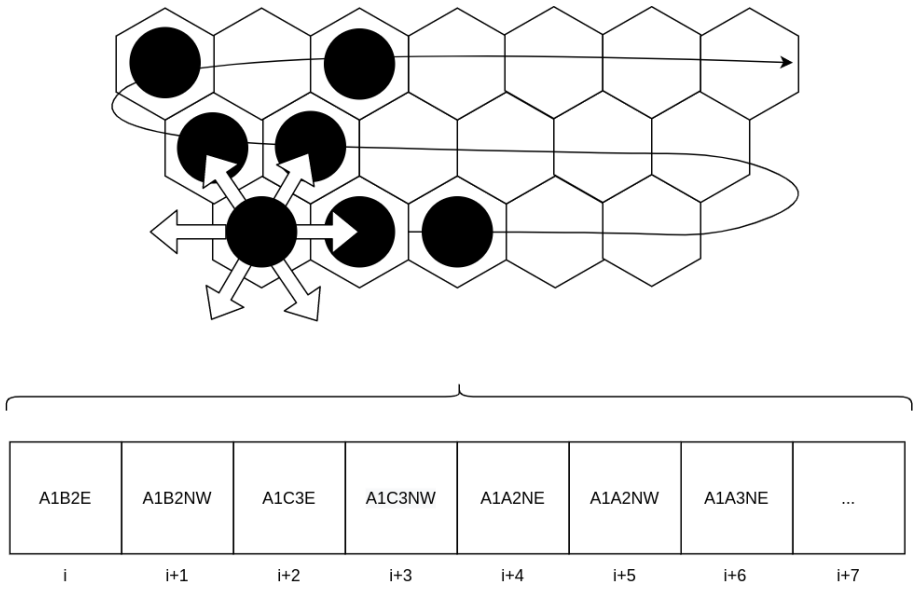


Figure 4.3: Generation of broadside moves starting at position A1, line direction NE, line length of two and at move direction NE

3. **Neural Net**: Is a wrapper for any neural network. It needs to be implemented for the specific framework used.
4. **MCTS**: Encapsulates the logic to perform Monte Carlo Tree Search on the game tree with the help of the neural network and the Game module.
5. **Arena**: Has the task to perform the matches between different agents.

We not only parallelized the framework, as explained in section 4.3.3, but also made multiple other modifications:

- The Arena was modified to use the Abalone engine [78] and game-playing agents implemented for that engine. This way, the neural nets can be faced off against other algorithms like Verloop's minimax.
- Moreover, the MCTS implementation could not deal with games with board states appearing multiple times in the search tree. The implementation uses hash maps to retrieve the action-values and counts for a board state. If the board state exists multiple times in the search tree, the values are skewed. We extended the hash of a state by a hash of the parent node's hash and the current depth of the node to solve the problem.
- Lastly, the Arena was parallelized because the matces with a random agent or a previous version requires a more significant number of games to get a statistically

relevant result. As discussed in section 4.3.3, the significant length of a game would slow down the training process too much.

Additionally, we added a CLI entry point to pass arguments to the training process. All relevant hyperparameters (cf. section 5.2) of a training run are persisted in a JSON file. Performance data (such as time per iteration) and the Arena results are logged as CSV tables.

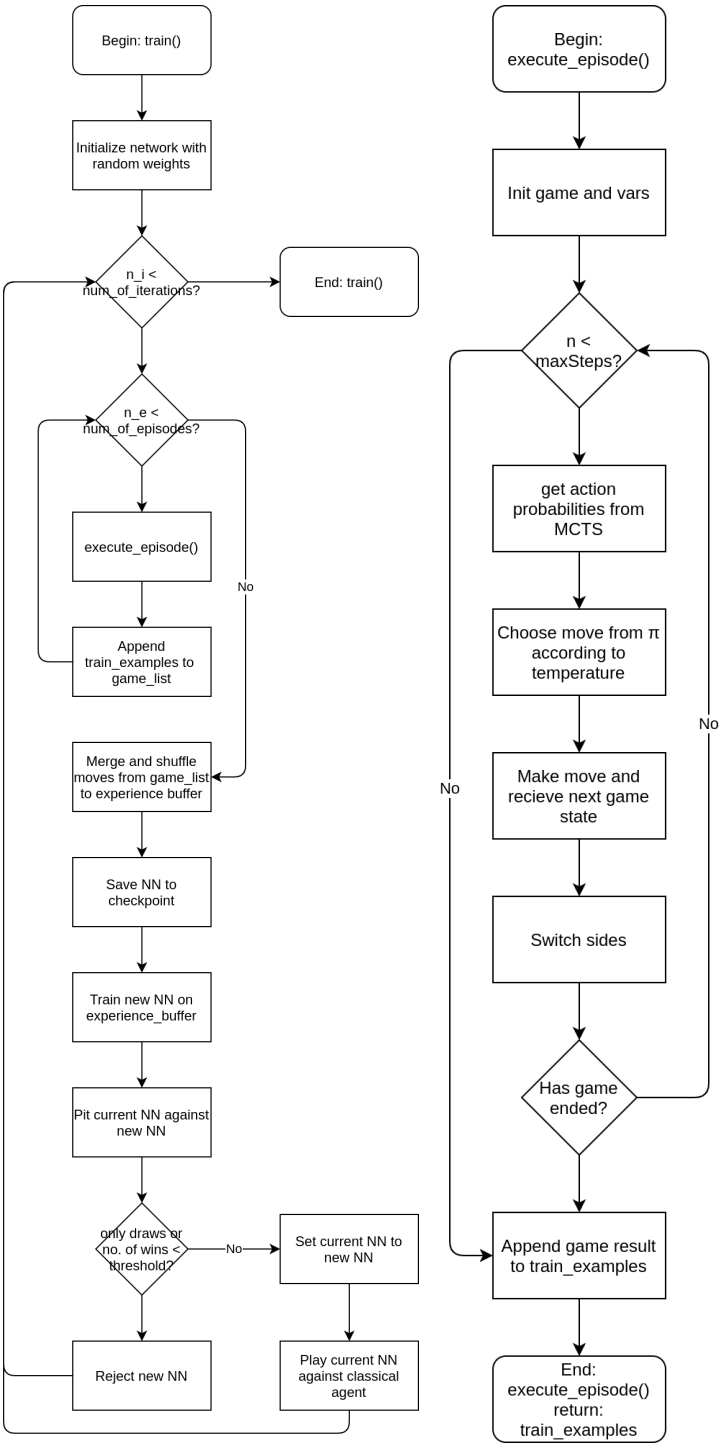
4.3.2 Training Algorithm

Even though we will alter the main training routine later to be executed in parallel, the logic remains quite similar to the original algorithm devised in AlphaZero General. Therefore, it makes sense to describe the coarse structure that is also outlined in figure 4.4.

The outer loop defines how many iterations of self-play and subsequent neural network training should be performed. At the beginning of each iteration, a predefined number of games (episodes) is supposed to be played.

In each episode, an MCTS is performed with the defined number of simulations. The temperature parameter τ determines the exploration. Either a move is selected with a probability proportional to the visit count, or the move with the highest count is selected. The move is applied to the board state. After switching the sides, it is checked whether a terminal state was reached. If not, the game is continued. Otherwise, the reward r is calculated. With r, the list of experiences (s, π, z) is generated, with z being the reward r from the perspective of the player in turn.

After the desired number of games has been played, the new experience is appended to the experience buffer. Moreover, the buffer is stored as a checkpoint. If the buffer exceeds a defined length, the oldest experience is removed. Then a new version of the network f_θ is trained and played against the previous version in the Arena. If the new network performs better than the previous version, it is stored on disk for checkpointing. Otherwise, the network is discarded. The training is then continued until the desired number of iterations is reached, or the training is aborted.



(a) The main loop of the training algorithm (b) The episode loop

Figure 4.4: The self-play training pipeline

4.3.3 Parallelization

AlphaZero General's training pipeline does not offer parallel training. By comparing the time taken to execute one episode for the game of Othello and Abalone, it showed that a game of Abalone takes on average 40 times (tested for $n = 100$) longer to finish: 1s for Othello and 40s for Abalone. Investigating potential improvements, it became clear that two major factors determine the runtime of an episode:

1. Length of the game. A game of Abalone takes significantly more turns, caused by rules not forbidding loops such that games can be infinite. Moreover, defensive play styles can draw out games significantly. In order to alleviate the issue, we limited the number of turns m per game. In this case, the length was limited to $m = 200$. The newly created terminal state is scored with a partial score: The scored marbles are subtracted by the opponent's score for each player. We scored win or loss as either 1 or -1. Hence, the partial score is normalized by the number of marbles needed to win. The partial score reflects which player had the upper hand at the artificial terminal state. For instance, the black player pushed three marbles off the board, the white player only one. From black's perspective the partial score is $\frac{3}{6} - \frac{1}{6} = \frac{1}{3}$, for white it is $\frac{1}{6} - \frac{3}{6} = -\frac{1}{3}$.

2. The MCTS being computationally expensive: A closer inspection of the execution time, holding all variables constant, demonstrates the culprit. The simulation step of MCTS uses the neural network. The time taken by this function is one order of magnitude larger than the second most expensive operation, which is the generation of legal moves. For example, when running 15 MCTS iterations with the original TensorFlow implementation, one turn takes

- Total time: 0.44s
- One MCTS iteration: 0.03s
- Neural network : 0.025s
- Valid move generation: 0.05s

Looking at the main variables that control the length of the nested loops, it becomes clear that reducing the time an MCTS iteration takes is essential. Assuming function f is the operation to perform one Monte Carlo simulation:

$$k := \text{number of games}$$

$$m := \text{number of turns per game}$$

$$n := \text{number of MCTS simulations}$$

$$f \in O(kmn)$$

Abalone requires a larger number of MCTS simulations. The other projects [40, 74] used only 30 iterations for Othello and Hex, but Abalone has a much more complex game tree. AlphaZero performs $1,600$ iterations [6, p. 11] for each move. The MCTS needs to recommend a better move than the pure network to improve. It is the policy improvement operator. Those properties were the reason to decide to use PyTorch rather than TensorFlow. The library brings a 5-times improvement for that step as shown in 4.2. Nevertheless, this only brings down execution time for one episode by a factor of 5, still being 8-times slower than the reference implementation for Othello. AlphaZero uses two ways to alleviate this problem:

- Parallelization of the self-play training
- Parallelization of the MCTS (APV-MCTS)

As Python's global interpreter lock [80] does not allow for true multithreading, parallelizing the MCTS poses significant complexity and difficulty. However, allowing for simultaneous self-play and neural network training is feasible. As already mentioned in section 4.1.2, the implementation by Bruåsdal [40] provides this feature. For that reason, its architecture is used as a blueprint for migrating AlphaZero General into a parallel architecture. The figure 4.5 depicts how the components mentioned interact in a parallel fashion.

The self-play workers use the current best neural network f_θ to generate experience. The experience is put into a queue to allow for asynchronous communication between the worker processes and the Coach. The queue is emptied and loaded into the replay buffer for each neural network training iteration. If the experience buffer exceeds the maximum size, the oldest experience is dropped. The number of training batches is defined by the batch size and buffer size (buffer size divided by batch size). The training batches are created by randomly sampling tuples (s, π, z) from the buffer. No tuple is used twice so that the entire buffer is utilized.

After training the neural network with the batches, the newly created network is pitted against the old version in the Arena. As both variants have to be pitted against each other multiple times, we parallelized the Arena. If the new network is stronger than the previous version, it is saved on disk. Then the version counter is incremented. The version counter is a shared variable between the workers and the Coach. After each completed self-play game, the worker checks if the version counter is larger than its stored version. If that is the case, the new network is loaded from disk.

An important factor is to balance the number of workers against the time it takes to train the neural network. If the buffer grows large, training takes a long time. In the meantime, much new experience is potentially generated, overriding all previous experience. As a result, all experience is only generated from one network. The training process might become less robust.

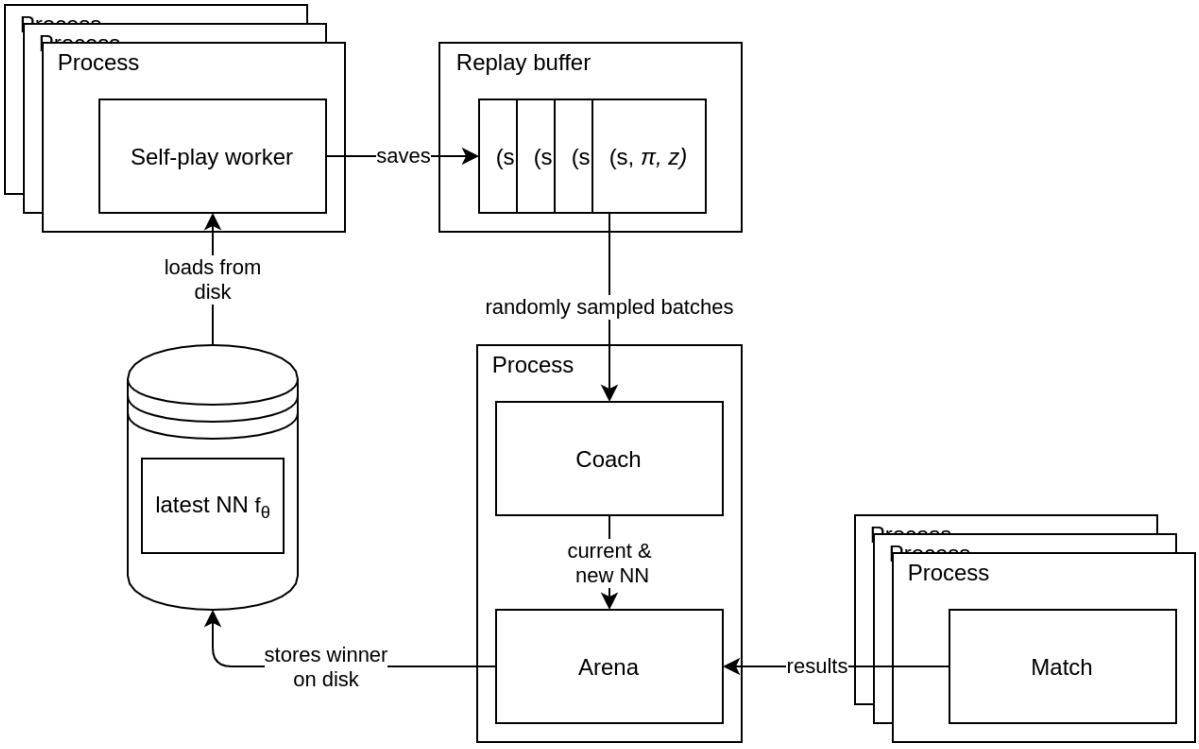


Figure 4.5: The different processes during the parallel training [40, cf. p. 45]

4.3.4 Distribution

The table 4.2 shows the usage of a GPU accelerates the inference speed significantly, particularly for PyTorch. Ergo, accelerating the self-play is highly desireable. When multiple processes utilize a GPU, each process requires a portion of the cards memory. PyTorch has the feature to share a model between multiple processes, wherefore a model does not need to be fully loaded for each process [81]. However, in PyTorch "every-time a process holds any pytorch object that is allocated on the GPU, then it allocates an individual copy of all the kernels (cuda functions) that pytorch uses, which is about 1GB" [82]. We observed that an unshared model uses 1.2GB of VRAM per process and decided to not to use the memory-sharing feature. The additional overhead to implement it did not justify the saved memory. Each process loads its own instance of the model.

Either way at some point the worker's VRAM needs outgrows the GPU's resources. We implemented the parallelization in such a way, that the user can assign GPU identifiers to each component (arena, training loop and workers). The processes then distribute the processes equally to the given hardware.

4.3.5 Symmetrical Board Generation

Abalone's board has six rotational symmetries and six mirror axes, as mentioned in section 3.6. By generating the symmetrical board states for each tuple (s, π, z), the data can be augmented by a factor of twelve. That means π and v are equivalent for up to 12 symmetrical board states. Some board configurations have less symmetrical boards, as some are identical. For example, rotating the default starting position by 120° produces the same board as mirroring it by the s-axis (cf. figure 3.5). In general, the factor of 12 does hold for almost all board states.

The first step in generating the symmetrical boards is to turn the marble positions into cube coordinates. There is a simple way for cube coordinates to mirror and rotate coordinates. A mapping function transforms marble coordinates from their matrix coordinates of the form (x, y) to (q, r, s). The origin of the cube coordinate system is laid into the center of the Abalone board, such that e.g. $E5 = (0, 0, 0)$ or $A1 = (-4, 4, 0)$.

The transformed coordinates can be rotated around the origin by 60° clockwise by shifting the values to the right and negating the values:

```
( q,  r,  s )
to  (-r, -s, -q )
to     (  s,  q,  r )
```

To mirror a cube coordinate along one of the three main coordinate axes q, r or s the two coordinates that don't belong to the axis are swapped:

```
function reflect_q(h) { return Cube(h.q, h.s, h.r); }
function reflect_r(h) { return Cube(h.s, h.r, h.q); }
function reflect_s(h) { return Cube(h.r, h.q, h.s); }
```

The coordinates need to be negated first for mirroring along the orthogonal axis to the main axis. Besides all marbles, all moves in π, whose probability is not 0, have to be transformed as well. First, the move corresponding to the index in π is looked up. As a move consists of one or two marble coordinates, those are transformed by the same scheme. The directions can be transformed into cube coordinates as well:

```
(+1, -1, 0): NORTH_EAST
(+1, 0, -1): EAST
(0, +1, -1): SOUTH_EAST
(-1, +1, 0): SOUTH_WEST
(-1, 0, +1): WEST
(0, -1, +1): NORTH_WEST
```

This way, the same functions can be applied to the directions. After all operations are applied, the cube coordinates are converted back to marble coordinates and moves. In

the case of the moves, the probabilities from the original π are written to the new index position.

4.3.6 Warm-Up

AlphaZero starts learning from scratch. Thus, there is no information about what constitutes a good move or strategy. The result is a cold-start problem. The self-play learning might not improve performance or even harm performance for an extended period of time. Equipped with ample computational resources like DeepMind, this poses no problem and is even desirable: The network is not nudged into a specific direction and can produce more novel insights.

Investigations into the application of AlphaZero's training principle showed possible ways to warm-start the training of the network [83]. Wang proposes an adaptive rollout-based warm-start. The method reintroduces rollouts into the MCTS that are performed at the beginning of the training process and are slowly phased out over the course of training.

Another possibility would be to reintroduce components of AlphaGo. Either by using a database of moves or letting the agent train against a heuristic agent. For Abalone, we introduce a similar method. By letting a heuristic agent play against a random agent, an initial experience buffer is created. Thus, the first iteration of the training of f_θ produces a function that approximates the heuristic player. We hypothesized that the self-play further improves this warmed-up agent like it did in the RL step in AlphaGo.

5 Experiments and Results

Each experiment has a hypothesis, a setup and a result.

5.1 Hardware

For the purpose of this thesis we had access to two machines. The smaller machine is a personal machine and a large machine rented at the provider Exoscale [84]. They will be referred to as *Balthazar* and *Melchior* respectively. Their specifications are described in table 5.1.

Component	Balthazar	Melchior
CPU	6 Core AMD Ryzen 3600	24 Core Intel Broadwell
RAM	32GB	120GB
GPU	NVIDIA GTX 1660 Super	$3 \cdot$ NVIDIA P100
VRAM	6GB	$3 \cdot 16$GB
Storage	500GB SSD	400GB SSD

Table 5.1: Hardware specifications of the utilized machines

5.2 Parameters

The behavior of the system is controlled by a multitude of parameters. The most relevant parameters are listed in table 5.2. If one of the parameters differs from the default values in an experiment, it will be mentioned. For each experiment, all parameters are pesisted by the system.

Name	Default	Explanation
temp_treshhold	60	The number of moves for which the next move is sampled (cf. equation 2.15)
update_treshold	0.6	The percentage of matches, that has to be won for a new network to be accepted
num_MCTS_sims	120	The number of times the search tree is expanded during MCTS
num_self_play_workers	9	The number of workers used for parallel self-play
num_arena_workers	8	The number of workers used for parallel Arena matches
load_model	false	Indicates whether to load an existing model
maxlen_experience_buffer	1,000,000	The maximum number of tuples (s, π, z) in the buffer
nnet_size	mini	The neural net used as introduced in 4.2.2
lr	0.001	The learning rate during training of the neural network
epochs	10	The number of of epochs during training of the neural network
batch_size	64	The size of the batches during training of the neural network

Table 5.2: The parameters of the training pipeline

5.3 Validation

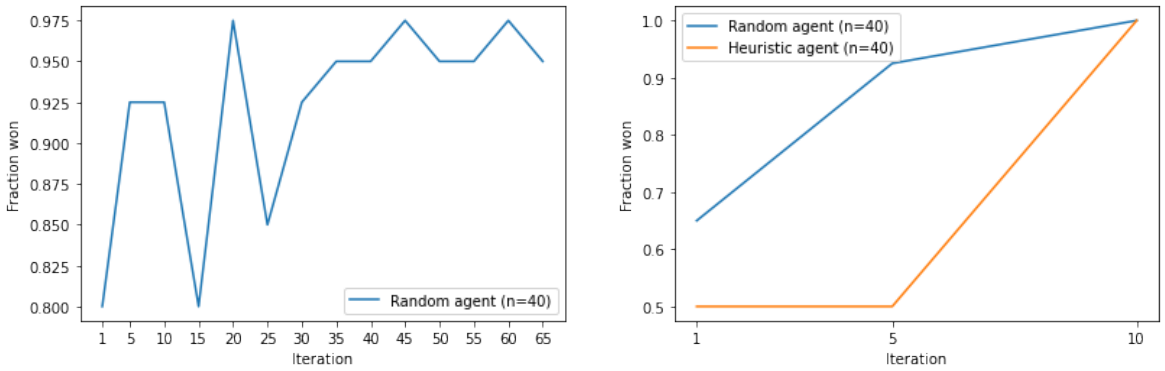
Hypothesis The modified framework still converges to optimal play for TicTacToe and Othello.

Setup Run pipeline with modified implemenation of TicTacToe and Othello from [75] on Balthazar.

Name	Value
temp_treshhold	15
num_MCTS_sims	30
num_self_play_workers	2
num_arena_workers	2
maxlen_experience_buffer	960,000

Table 5.3: The parameters for the naive run

Result Convergence to optimal play for TicTacToe (cf. figure 5.1) and Othello very likely.



(a) TicTacToe $3 \cdot 3$ field

(b) Othello $6 \cdot 6$ field

Figure 5.1: Win-ratio against random baseline in TicTacToe and Othello. Win-ratio is $\frac{\text{gamesWon}}{\text{allGames}}$

5.4 Application

5.4.1 Naive Run

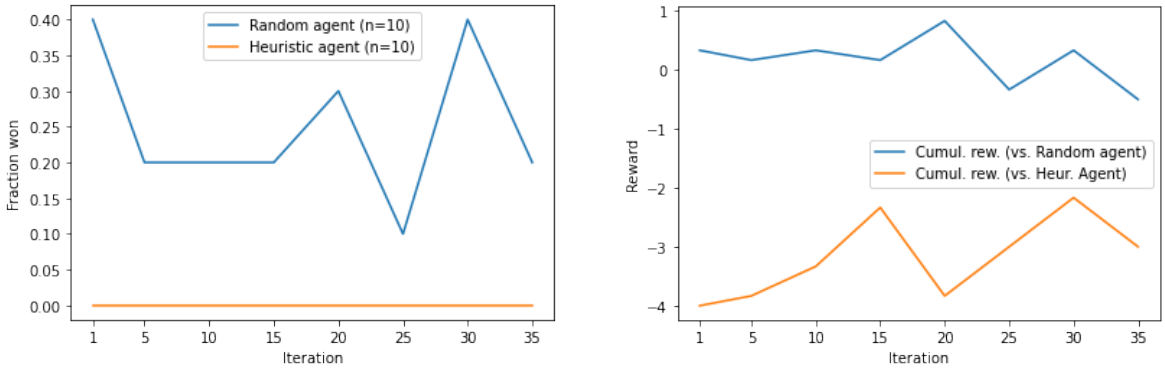
Hypothesis The naive implementation without any Abalone specific modifications converges to optimal play.

Setup Run pipeline on Balthazar.

Name	Value
num_MCTS_sims	60
num_self_play_workers	2
num_arena_workers	2
maxlen_experience_buffer	960,000

Table 5.4: The parameters for the naive run on Balthazar

Result No improvement in playing performance, cf. figure 5.2



(a) The win-ratio of the naive implementation

(b) The cumulative reward recieved by the naive implementation

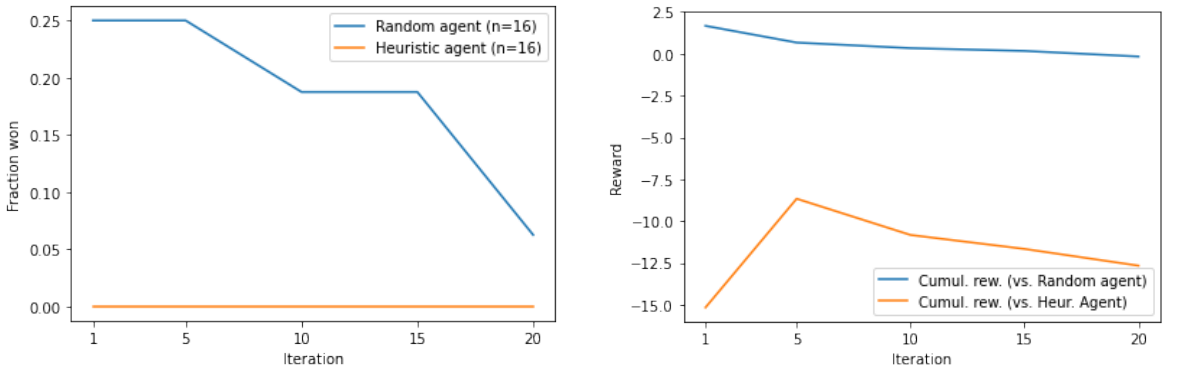
Figure 5.2

5.4.2 Scaled naive Run

Hypothesis The naive implementation without any Abalone specific modifications converges to optimal play on a larger machine with a bigger buffer and more workers.

Setup Run pipeline on Melchior.

Result No improvement in playing performance, divergence, cf. figure 5.3



(a) The win-ratio of the naive implementation

(b) The cumulative reward recieved by the naive implementation

Figure 5.3

5.4.3 Reward Distribution

Hypothesis The distribution of game results z is uneven.

Setup Plot histogram of z in experience buffer of previous run.

Result Most of the games end up as drawn or with minor advantage for one player, cf. figure 5.4).

5.4.4 Scaled warmed-up Run

Hypothesis Pretraining the network on experience generated by Verloop's minimax against a random player nudges the network towards more aggressive play. The resulting experience buffer has stronger signals z, which improves performance.

Setup Run pipeline on Melchior with pretrained network.

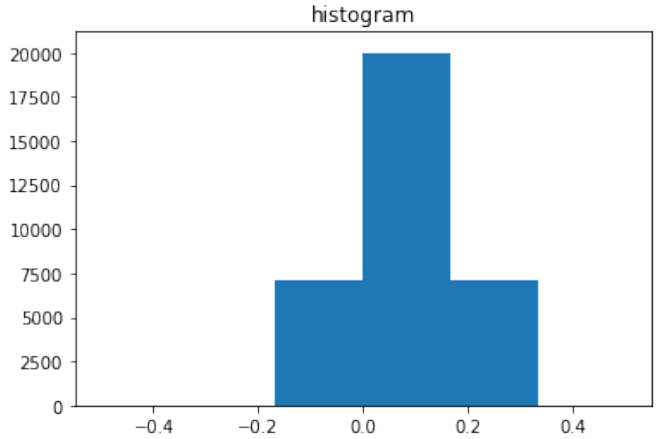
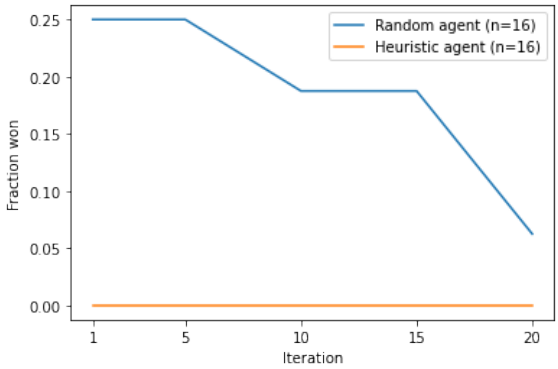


Figure 5.4: Distribution of z in experience buffer

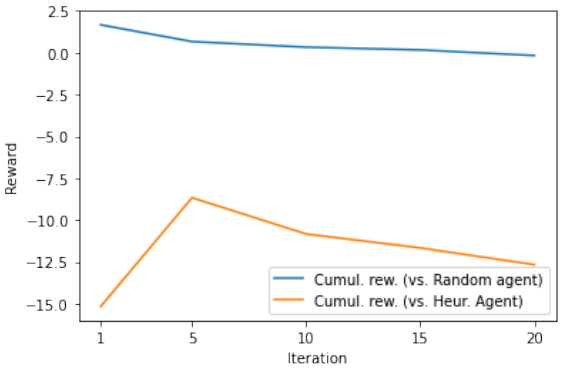
Name	Value
load_model	true

Table 5.5: The parameters for the validation runs

Result The experience buffer during different iterations of the training run looks promising. The distribution shows much higher rewards, many games even ending before the cutoff of 200 moves, cf. figure 5.6.

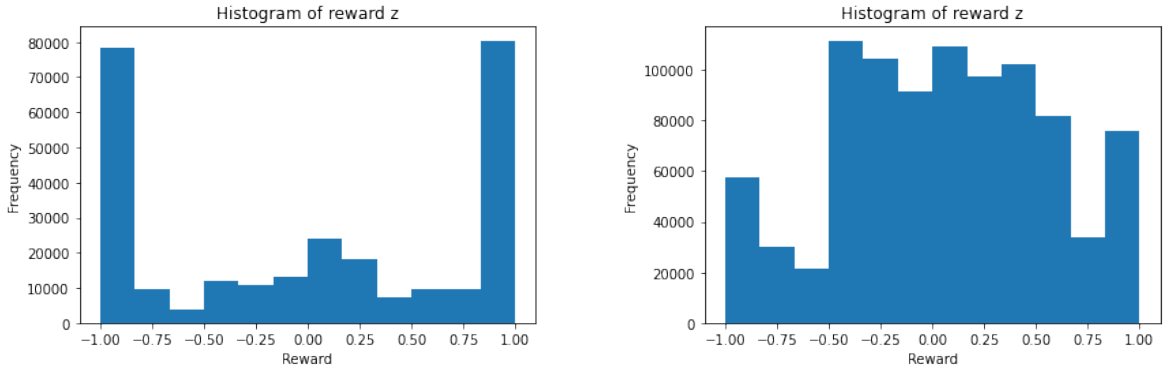


(a) The win-ratio of the naive implementation



(b) The cumulative reward recieved by the naive implementation

Figure 5.5



(a) The experience buffer after the first iteration

(b) The experience buffer after the tenth iteration

Figure 5.6: Initially the buffer is still filled with the experience from the prewarming phase, where all games ended with eiter -1 or 1 reward.

```
BLACK 14 - WHITE 14
    I · · @ @ ·
   H · @ @ @ @ ·
  G · · @ @ @ @ ·
 F · · @ @ @ @ · ·
E · · O O O O · · ·
 D · · O O O O · · 9
  C · · O O O O · 8
   B · · · O O · 7
    A · · · · · 6
      1 2 3 4 5
```

(a) The agent shows strong play, drawing against the heuristic agent

```
BLACK 9 - WHITE 14
    I · · · · · ·
   H · · @ @ · ·
  G · · @ @ @ · ·
 F · · @ @ @ @ · ·
E · · O @ @ @ · · ·
 D · · · @ @ · O · 9
  C · · O · · O · 8
   B · · · · O O 7
    A · O O O · 6
      1 2 3 4 5
```

(b) The agent shows weak play, loosing formation and getting pressed against the brink of the board

Figure 5.7: The final board positions against heuristic player, during training iteration 10 with the warmed up NN

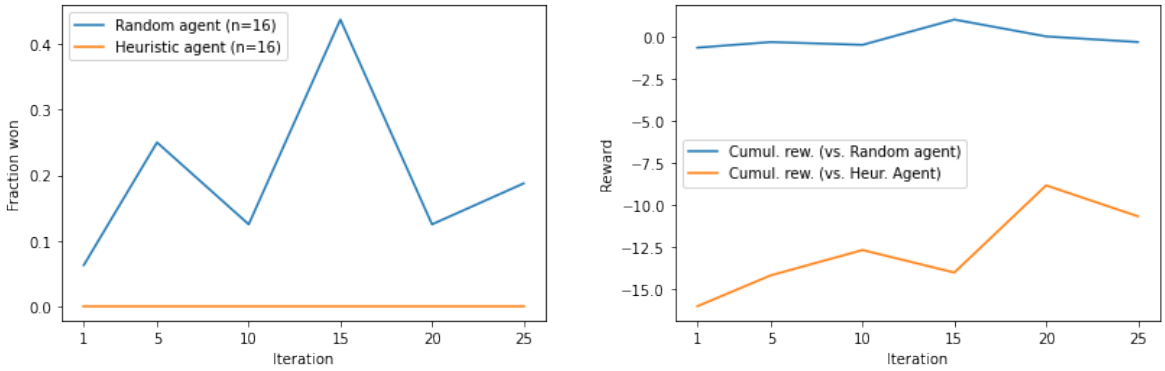
5.4.5 Scaled Run with adjusted Reward z

Hypothesis The reward function provides no penalty for long matches, that result in a draw.

Setup Run pipeline on Melchior with adjusted reward function. Each turn has a reward of -0.001, for a maximum game length of 200 that is a maximum of -0.2. The other

rewards remain unchanged.

Result Inconclusive for the amount of iterations performed. A tendency of improvement is present.



(a) The win-ratio of the naive implementation

(b) The cumulative reward recieved by the naive implementation

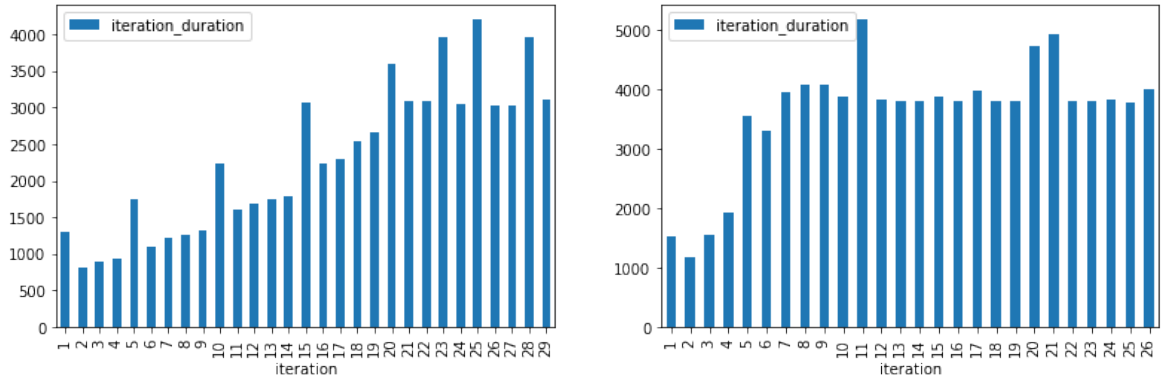
Figure 5.8

5.4.6 Runtime of Experiments

Hypothesis The training of the neural network has become a bottlenet.

Setup Compare lengths of training iterations on Balthazar and Melchior.

Result Due to the parallel nature, self-play games are generated continuously. The longer one training iteration takes, the bigger the backlog of new games becomes. One iteration took on average 1h. The training of the network has become the limiting factor not, the generation of self-play games, cf. figure 5.9.



(a) Training run on Balthazar

(b) Training run on Melchior

Figure 5.9: Duration of one training iteration in seconds. The training times stabilize once the experience buffer has reached its maximum size.

6 Conclusion

6.1 Goal Evaluation

The application of AlphaZero to Abalone has proven to be difficult. Without the availability of an training framework based on AlphaZero, the time-frame for this thesis would have been too small. The framework gave some guarantees about correctness and saved engineering efforts. The hexagonal board and the more complex move system introduced difficulties in applying the original architecture. A more significant problem is the state space and game tree complexity of Abalone, which require much more compute than Hex or Othello. The high demands for compute limited the amount of experiments, that could be run. Machine learning is highly empirical, therefore less iterations means less hypotheses and hyperparameter configurations could be tested. The originally stated goal was to apply the methods of AlphaZero to Abalone. As we created a training pipeline and game-playing agent capable of running, this goal is at least partially achieved. However, no trends of convergence towards an optimal policy was observed. The most successful variant was the network, that was warmed-up with experience from the heuristic agent. It was immediately able to beat the random agent in 100% of the games.

The achieved results are weak when compared with the performance of minimax based methods. The compute resources required overshadow the modest single threaded implementation by Verloop. As Abalone has a lower strategic complexity than e.g. chess, the heuristics designed by humans are quite powerful. A big advantage of the self-play learning is the ability to learn new strategies, previously not known to humans, as observed in AlphaGo. We could not achieve this augmentation of human knowledge for Abalone. The win-ratio against heuristic players was too low.

6.2 Future Work

The method proposed by AlphaZero is extremely powerful and has proven to be promising for Abalone as well. Due to limitations in compute for the experiments it has not been possible to replicate the ground breaking success achieved in Go for Abalone. This

also points at the major downside of the method as it requires significantly more compute than any classical knowledge based methods. Gradient descent and even simple feedfowards for neural networks are very expensive operations even with the proliferation of ever more powerful hardware accelerators. The hardware used by DeepMind for AlphaGo is only accessible to top researchers in the field due to the high cost. There are two main potential avenues through which we could reap the benefits of this method with lower capital requirements:

- Further theoretic improvements bringing significant speedups. This could be something along the lines of the incremental improvement between AlphaGo and AlphaZero or full paradigm shifts in the methodology.
- The cost for training neural networks coming down by an order of magnitude from current levels. This could be due simply to the passage of time as in the past accelerators like GPUs still followed an exponential improvementrate as observed in Moore's Law for CPUs [85] ("Huang's Law" [86]). Another factor would be archtitectural changes that improve scalability and performance for deep learning specifically. Additionally, the rising economic significance of machine learning has provided an incentive for more specialized hardware like TPUs [72] or Jim Keller's Grayskull. [87] The same reason reignited interest in optical computing accelerators to bring drastic changes in power requirements and performance for matrix multiplication. [88, 89]

In addition to gaining more compute resources, there are also potential theoretic improvements to make. Abalearn describes a "risk-sensitive" approach as "the problems we encountered was that self-play was not effective because the agent repeatedly kept playing the same kind of moves, never ending a game" [69, p. 8]. The function described could be transferred to this work in order to encourage more aggressive game-play.

Adjusting the reward to nudge the behaviour of the agent into a certain direction remains promising as well, even though the attempt was not successful.

Bibliography

[1] C. Higgins, "A Brief History of Deep Blue, IBM's Chess Computer — Mental Floss," https://web.archive.org/web/20170803130439/https://www.mentalfloss.com/article/503178/history-deep-blue-ibms-chess-computer, Jul. 2017.

[2] J. Haugeland, *Artificial Intelligence: The Very Idea*. Cambridge, Mass: MIT Press, 1985.

[3] A. M. TURING, "I.—COMPUTING MACHINERY AND INTELLIGENCE," *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950.

[4] DeepMind, "Match 1 - Google DeepMind Challenge Match: Lee Sedol vs AlphaGo," https://www.youtube.com/watch?v=vFr3K2DORc8&t=7020s.

[5] N. J. Nilsson, *Artificial Intelligence: A New Synthesis*. Elsevier, Apr. 1998.

[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017.

[7] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, "Mastering Atari, Go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, Dec. 2020.

[8] D. Bradley, "[CLOSED] Win! Win! Win! There are boxes of tabletop game Abalone on offer with our hot new newsletter," https://www.pocketgamer.com/abalone/closed-win-win-win-there-are-boxes-of-tabletop-game-abalone-on-offer-with-our-ho/, 2018.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[10] C. Berner, G. Brockman, B. Chan, V. Cheung, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with Large Scale Deep Reinforcement Learning," p. 66, Dec. 2019.

[11] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, Nov. 2019.

[12] C. Gamble and J. Gao, "Safety-first AI for autonomous data centre cooling and industrial control," https://deepmind.com/blog/article/safety-first-ai-autonomous-data-centre-cooling-and-industrial-control, Aug. 2018.

[13] T. Jebara, "For Your Ears Only: Personalizing Spotify Home with Machine Learning," https://engineering.atspotify.com/2020/01/16/for-your-ears-only-personalizing-spotify-home-with-machine-learning/, Jan. 2020.

[14] F. Siddiqi, "ML Platform Meetup: Infra for Contextual Bandits and Reinforcement Learning," https://netflixtechblog.com/ml-platform-meetup-infra-for-contextual-bandits-and-reinforcement-learning-4a90305948ef, Oct. 2019.

[15] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, Q. V. Le, J. Laudon, R. Ho, R. Carpenter, and J. Dean, "A graph placement methodology for fast chip design," *Nature*, vol. 594, no. 7862, pp. 207–212, Jun. 2021.

[16] L. Pinto and A. Gupta, "Supersizing self-supervision: Learning to grasp from 50K tries and 700 robot hours," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*. Stockholm, Sweden: IEEE, May 2016, pp. 3406–3413.

[17] A. Zeng, S. Song, S. Welker, J. Lee, A. Rodriguez, and T. Funkhouser, "Learning Synergies Between Pushing and Grasping with Self-Supervised Deep Reinforcement Learning," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Madrid: IEEE, Oct. 2018, pp. 4238–4245.

[18] P. Agrawal, A. V. Nair, P. Abbeel, J. Malik, and S. Levine, "Learning to Poke by Poking: Experiential Learning of Intuitive Physics."

[19] G. Kahn, A. Villaflor, V. Pong, P. Abbeel, and S. Levine, "Uncertainty-Aware Reinforcement Learning for Collision Avoidance," *arXiv:1702.01182 [cs]*, Feb. 2017.

[20] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, second edition ed., ser. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press, 2018.

[21] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson Education, Inc, 2021.

[22] D. E. Knuth and L. T. Pardo, "The Early Development of Programming Languages," in *A History of Computing in the Twentieth Century*, N. Metropolis, J. Howlett, and G.-C. Rota, Eds. San Diego: Academic Press, Jan. 1980, pp. 197–273.

[23] C. E. Shannon, "XXII. Programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, Mar. 1950.

[24] "Stockfish (chess)," *Wikipedia*, Dec. 2021.

[25] "Stockfish - Open Source Chess Engine," https://stockfishchess.org/.

[26] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search," in *Computers and Games*, ser. Lecture Notes in Computer Science, H. J. van den Herik, P. Ciancarini, and H. H. L. M. J. Donkers, Eds. Berlin, Heidelberg: Springer, 2007, pp. 72–83.

[27] L. Kocsis and C. Szepesvári, "Bandit Based Monte-Carlo Planning," in *Machine Learning: ECML 2006*, ser. Lecture Notes in Computer Science, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Berlin, Heidelberg: Springer, 2006, pp. 282–293.

[28] "Fig. 1. Outline of a Monte-Carlo Tree Search." https://www.researchgate.net/figure/Outline-of-a-Monte-Carlo-Tree-Search_fig1_23751563.

[29] B. Bouzy, "Associating Shallow and Selective Global Tree Search with Monte Carlo for 9 × 9 Go," in *Computers and Games*, ser. Lecture Notes in Computer Science, H. J. van den Herik, Y. Björnsson, and N. S. Netanyahu, Eds. Berlin, Heidelberg: Springer, 2006, pp. 67–80.

[30] P. Auer and N. Cesa-Bianchi, "Finite-time Analysis of the Multiarmed Bandit Problem."

[31] S. Gelly and D. Silver, "Achieving Master Level Play in 9 × 9 Computer Go," p. 4.

[32] ——, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, Jul. 2011.

[33] X. Yang, "Markov Chain and Its Applications," Social Science Research Network, Rochester, NY, SSRN Scholarly Paper ID 3562746, Mar. 2019.

[34] R. Bellman, "A Markovian Decision Process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957.

[35] "Examples of MDPs - Markov Decision Processes," https://www.coursera.org/lecture/fundamentals-of-reinforcement-learning/examples-of-mdps-ACRYv.

[36] L. Moroney, *AI and Machine Learning for Coders: A Programmer's Guide to Artificial Intelligence*. O'Reilly, 2020.

[37] C. Colah, "Calculus on Computational Graphs: Backpropagation – colah's blog," https://colah.github.io/posts/2015-08-Backprop/.

[38] "Convolutional neural network," *Wikipedia*, Jan. 2022.

[39] E. W. Weisstein, "Convolution," https://mathworld.wolfram.com/Convolution.html.

[40] H. Bruåsdal, "Deep reinforcement Learning Using Monte-Carlo Tree Search for Hex and Othello," 2020.

[41] R. Ilin, T. Watson, and R. Kozma, "Abstraction hierarchy in deep learning neural networks," in *2017 International Joint Conference on Neural Networks (IJCNN)*. Anchorage, AK, USA: IEEE, May 2017, pp. 768–774.

[42] M. K. (https://stats.stackexchange.com/users/7250/matt-krause), "What is translation invariance in computer vision and convolutional neural network?" Cross Validated.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2017.

[44] "Wolfram—Alpha Widgets: "Plot two functions" - Free Mathematics Widget," https://www.wolframalpha.com/widgets/view.jsp?id=59752a7f2c9aa5d375de1f1d13a3f5c4.

[45] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778.

[46] A. Halevy, P. Norvig, and F. Pereira, "The Unreasonable Effectiveness of Data," *IEEE Intelligent Systems*, vol. 24, no. 2, pp. 8–12, Mar. 2009.

[47] "Go (game)," *Wikipedia*, Jan. 2022.

[48] M. Müller, "Computer Go," *Artificial Intelligence*, vol. 134, no. 1, pp. 145–179, Jan. 2002.

[49] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 2016.

[50] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," 1992.

[51] C. D. Rosin, "Multi-armed bandits with episode context," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 3, pp. 203–230, Mar. 2011.

[52] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," Dec. 2017.

[53] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling Network Architectures for Deep Reinforcement Learning," p. 9, 2016.

[54] "Abalone (board game)," https://en.wikipedia.org/wiki/Abalone_(board_game), Dec. 2020.

[55] A. S.A., "Abalone rulebook," https://cdn.1j1ju.com/medias/c2/b0/3a-abalone-rulebook.pdf.

[56] "Red Blob Games: Hexagonal Grids," https://www.redblobgames.com/grids/hexagons/.

[57] towzeur, "Towzeur/gym-abalone," Jan. 2021.

[58] O. Aichholzer, "Abalone games," http://www.ist.tugraz.at/staff/aichholzer/research/rp/abal
2006.

[59] XKCD, "Standards," https://xkcd.com/927/.

[60] Scriptim, "Scriptim/Abalone-BoAI," Apr. 2021.

[61] N. Lemmens, "Constructing an abalone game-playing agent," in *Bachelor Conference Knowledge Engineering, Universiteit Maastricht*. Citeseer, 2005.

[62] P. Chorus, "Implementing a computer player for abalone using alpha-beta and monte-carlo search," Master's thesis, Citeseer, 2009.

[63] A. Papadopoulos, K. Toumpas, A. Chrysopoulos, and P. A. Mitkas, "Exploring Optimization Strategies in Board Game Abalone for Alpha-Beta Search," *IEEE Conference on Computational Intelligence and Games*, p. 8, 2012.

[64] E. Ozcan and B. Hulagu, "A Simple Intelligent Agent for Playing Abalone Game: ABLA," p. 10, 2004.

[65] O. Aichholzer, F. Aurenhammer, and T. Werner, "Algorithmic fun-abalone," *Special Issue on Foundations of Information Processing of TELEMATIK*, vol. 1, pp. 4–6, 2002.

[66] O. Aichholzer, "Oswin Aichholzer's homepage," http://www.ist.tugraz.at/staff/aichholzer/research/rp/abalone/, 2006.

[67] M. Verloop, "A Critical Review: Exploring Optimization Strategies in Board Game Abalone for Alpha-Beta Search," p. 49.

[68] ——, "AbaloneAI," https://github.com/MichielVerloop/AbaloneAI.

[69] P. Campos and T. Langlois, "Abalearn: Ecient Self-Play Learning of the game Abalone," 2003.

[70] G. Tesauro, "TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play," *Neural Computation*, vol. 6, no. 2, pp. 215–219, Mar. 1994.

[71] R. Mizrachi, G. Golran, O. Jacobi, and R. Zats, "Introduction to artificial intelligence Final Project," The Hebrew University of Jerusalem, Tech. Rep., 2017.

[72] "TPU Research Cloud," https://sites.research.google/trc/.

[73] "Tf.keras.Model — TensorFlow Core v2.7.0," https://www.tensorflow.org/api_docs/python/tf/keras/M

[74] S. Thakoor, S. Nair, and M. Jhunjhunwala, "Learning to Play Othello Without Human Knowledge," Stanford University, Final Project Report.

[75] ——, "Suragnair/alpha-zero-general: A clean implementation based on AlphaZero for any game in any framework + tutorial + Othello/Gobang/TicTacToe/Connect4 and more," https://github.com/suragnair/alpha-zero-general.

[76] henribru, "Deep MCTS," Oct. 2021.

[77] OpenAI, "Gym: A toolkit for developing and comparing reinforcement learning algorithms," https://gym.openai.com.

[78] T. Claussen, "Abalone," 2021.

[79] ——, "Campfireman/Abalone-BoAI," Jun. 2021.

[80] "GlobalInterpreterLock - Python Wiki," https://wiki.python.org/moin/GlobalInterpreterLock.

[81] "Module — PyTorch 1.10.1 documentation," https://pytorch.org/docs/stable/generated/torch.nn.Mod

[82] radim_shark, "Sharing model between processes automatically allocates new memory," https://discuss.pytorch.org/t/sharing-model-between-processes-automatically-allocates-new-memory/96724/3, 2020.

[83] H. Wang, M. Preuss, and A. Plaat, "Adaptive Warm-Start MCTS in AlphaZero-like Deep Reinforcement Learning," *arXiv:2105.06136 [cs]*, May 2021.

[84] "Exoscale," https://www.exoscale.com/.

[85] G. E. Moore, "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, Sep. 2006.

[86] "Huang's law," *Wikipedia*, Oct. 2021.

[87] "Grayskull," https://tenstorrent.com/grayskull/.

[88] "Lightmatter," https://lightmatter.co/.

[89] "Lightelligence," https://www.lightelligence.ai/.