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"

January 10, 2022

Author Ture Claußen
Matriculation number: 1531067
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. Vorname Name
Abteilung Informatik, Fakultät IV
Hochschule Hannover
email-Adresse

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, January 10, 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, these knowledge-based approaches are severely limited by our ability to properly model the problem correctly and exhaustively. 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 in computational power (and data) has enabled two subsymbolic methods to find large 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 and gradient descent to train an evaluation function based on a large 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 lookahead 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]

At this point our endeavor begins, as the purpose of this writing 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 goal of the game is to push six of the opponent's marbles off the playing field.

Naturally, the question arises whether this is a worthwhile thing to do. Firstly, academic interest in Abalone has remained steady even though the popularity of the game declined. However, so far the application of learning methods to Abalone is less explored, especially regarding the novel techniques introduced by AlphaZero.

Further investigating the applicability of RL is a relevant topic. Aside from (super-) human performance in games [8, 9, 10], RL continues to find more industry application in areas like improving data center cooling at Google [11] or content recommendation at Spotify [12] and Netflix. [13] A very recent example is the floor planning for Google's latest TPU chip, which was aided by a deep RL algorithm. Floor planning is the process in which the chip designer tries to arrange the functional units of a chip like cores, memory, interfaces etc. in such a way that the wire length, power consumption and total area of the chip are minimized. The floor planning for the previous TPU took several months of work for a human team. In contrast, the RL algorithm only takes one day of compute time to produce a design that meets the same design criteria. [14]

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, computational requirements, etc.

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 the specialized knowledge such as deep reinforcement learning in particular.

2.1 Artificial intelligence

The introduction has already foreshadowed how the field of artificial intelligence has undergone a shift in their methodology. In the 1950s and 1960s figures like Alan Turing and von Neumann laid the foundations for modern computers. This sparked the idea one could create programs for these new machines that have similar abilities like humans and other organisms. Researchers at that time assumed there is 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. [15, p. 8f.]

2.1.1 Rational 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 has to be further qualified by calling the agent rational. This means that it acts so as "to achieve the best outcome or, when there is uncertainty, the best expected outcome". [16, p. 36]

The agent exists in an environment which it perceives through sensors and it 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.

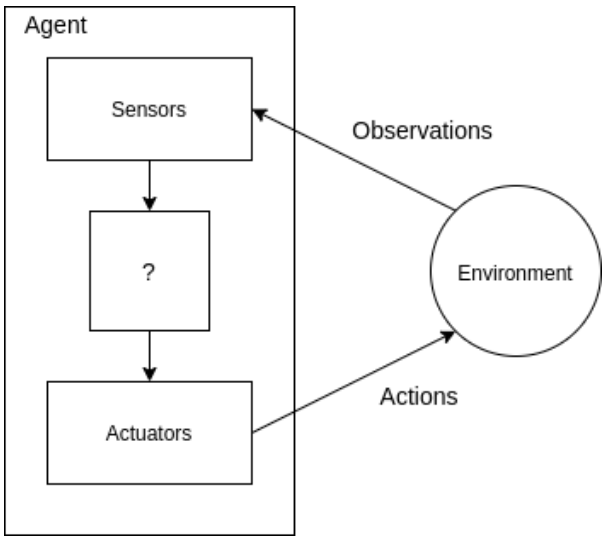


Figure 2.1: The agent-environment interaction loop [16, cf. p. 96]

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 could use a complex algorithm with an additional model.

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. 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 key components. Let us specify the task environment for a machine classifying defective parts in a production line:

1. Performance measure: This might be the percentage of correctly broken parts (true positives) weighted 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).

There are a few categorizations of the properties of task environments that are extremely helpful for narrowing down potential applicability of different classes of algorithms. A key 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 players' cards and the upcoming cards cannot be seen but are highly relevant to the agent's actions. As the current board state of chess fully 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. An autonomous vehicle for instance is dealing with continuous time and also continuous states. The speed of the car 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) permutations of the board and the marbles. 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. There might be other car drivers taking unexpected actions or a comet hitting our car.

Further expanding on the passage of time we have to take into account if actions have consequences for future states. If each combination of percept and action is independent of each other we call it *episodic* and *sequential* otherwise. If we 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, the other vehicles apart from our autonomous vehicle all profit when it avoids a collision and vice versa. Therefore, they cooperate.

2.2 Classic agent

Now that we have an intricate understanding of the environment of agents we have also narrowed down the choice of algorithms we can employ. In general, perfect information games can be solved by adversarial search algorithms. That means in theory one can find the optimal solution by traversing the entire game tree to terminal states (game ending state). The theory behind this type of agent was already laid out as early as 1945 by Konrad Zuse's program generating legal chess moves [17], but was described most comprehensively by Claude Shannon in 1950 in "Programming a Computer for Playing Chess". [18]

2.2.1 Minimax algorithm

Minimax assumes two roles: The minimizer (min) and the maximizer (max). We start the search from the current board state as the role of the maximizer and then alternate between the two. The result of the minimax search gives us the maximum utility for the given state, assuming both players behave optimally. Let us define the functions [16, p. 303f.]

- $\texttt{utility}(s, p)$ 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.
- $\texttt{is-terminal}(s)$ returns whether the given state s is a terminal state or not.
- $\texttt{to-move}(s)$ returns the current player for the state s, either min or max.
- $\texttt{result}(s, a)$ returns the resulting state if in state s and taking action a.
- $\texttt{actions}(s)$ returns all legal moves for the given state s.

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

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 up 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. This value again is passed up to the parent until we reach the parent node which is always a maximizer, yielding us the maximum utility we can achieve given our opponent plays optimally.

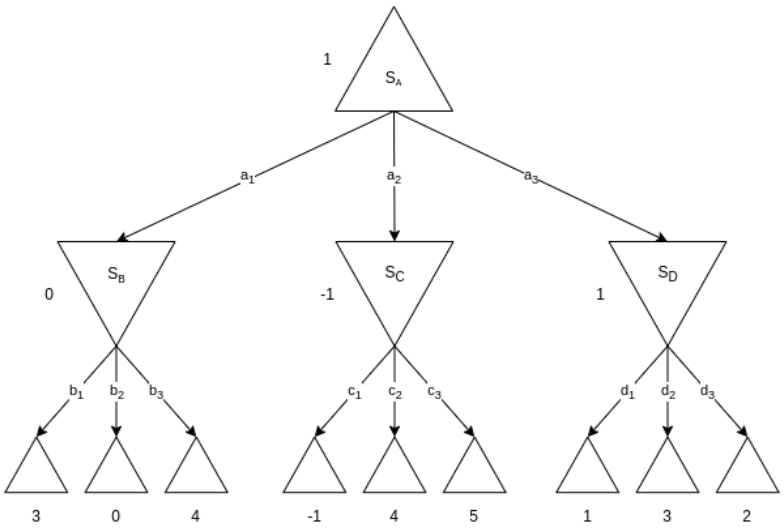


Figure 2.2: Minimax for a small search tree, resulting in an utility value of 1

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. Our 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. A sensible function for chess might be a linear combination in the form of:

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

With functions f_i calculating different values such as

- Material: First, we assign an integer to each piece 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 king's position, count attacking pieces close to the king etc.
- Win and loss: As a more definitive measure we can indicate whether the current state is a terminal state and hence a winning or losing state.
- Etc.

By applying different weights ω_i to the functions f_i we essentially give incentives to the agent to prioritize certain behavior. If the win or loss function returns a value of either -1 or $+1$, we 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 we can find good moves with minimax search even in highly complex state spaces.

However, the problem with heuristic functions is we need expert knowledge and a lot of 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

We can further improve minimax search 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 we visit nodes in minimax is similar to a graph traversal with depth first search, meaning we descend down until we find a leaf node. This gives us information about the utility of that node and, consequently, part of the tree. Going up the tree we keep an alpha value for the minimum value the maximizer will receive and a beta value for the maximum value the minimizer will achieve. For instance this lets us know if the minimizer already can choose a move worse than what we can achieve with another move, we do not descend further ($\alpha > \beta$).

Looking at the example in figure 2.3 will help us illustrate this principle. Our search revealed that choosing move a_1 will yield us 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. We 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 we expand which node matters decides how many nodes we can prune. Had we visited move c_3 and c_2 first, pruning wouldn't have been possible. The best case of

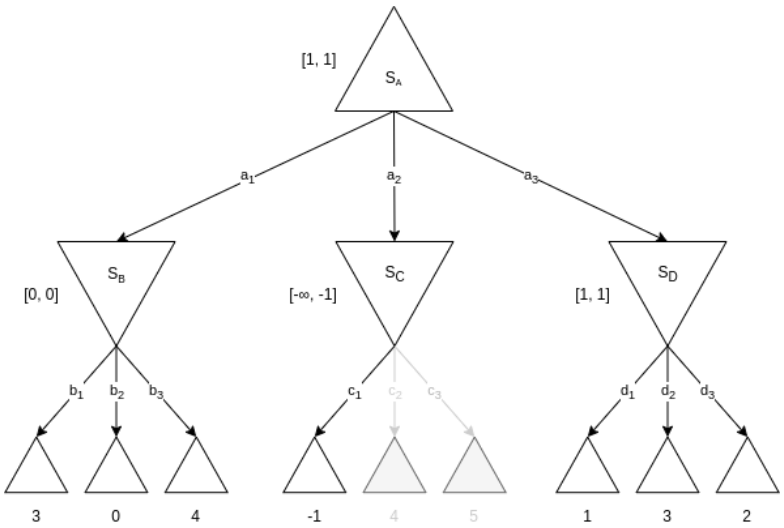


Figure 2.3: Our previous example but with alpha beta pruning applied. The grayed out nodes indicate, that these in fact could be pruned from the tree

$O(\sqrt{b^d})$ is entirely dependent on this ordering. We could find 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 a lot of redundancy, we make up for this more than enough by pruning much more effectively.

Other improvements to the procedure are thinkable as well. Once we perform a search for a certain state, we can store the resulting utility. If we encounter this position again, because of a different permutation of the move sequence (transposition), we can just look up the state utility 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 we cannot use the utility of terminal states, 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). [19, 20]

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 [21] 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, meaning we take a state or node to be investigated and let two random players take turns until a terminal state is reached. Kocsis and Szepesvári [22] showed that it does in fact converge to optimal play. For games with a high branching factor we need a large number of simulations to get any meaningful information from the simulations, so we might use a *rollout policy* instead. This policy guides 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. We start at the root node and select a node until we reach a leaf node. This is the selected Node. We could select the nodes according to some stationary distribution or we could use the information that we gain over time.
- **Expansion** is the step in which we expand the selected node by appending a fresh child node.
- **Simulation** is as described before the step in which we perform a simulation with our rollout policy starting from the state of the newly generated child node.
- **Back-propagation** is the last step. We take the result of the simulation (utility) and write it to the node and parent nodes above until we reach the root node. 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. 2.3.4).

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 [24]. In the

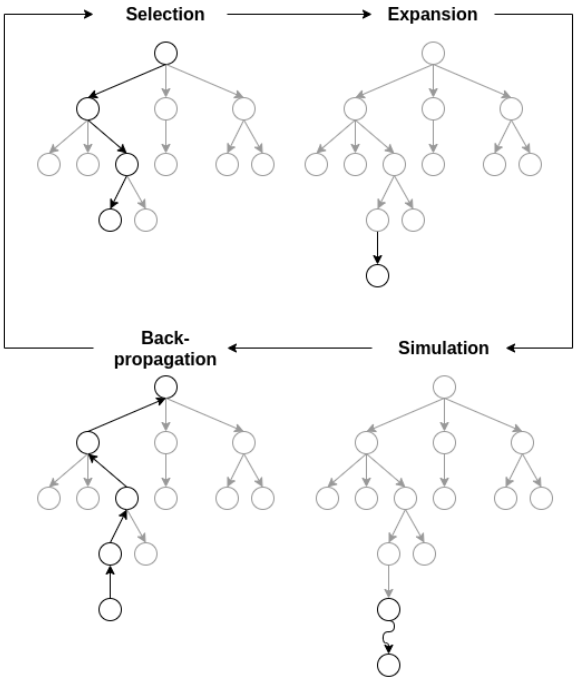


Figure 2.4: Monte Carlo tree search stages [23]

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. [21, p. 4]

Another idea for selection is the upper confidence bound formula (*UCB1*) [25], that weights how often we have visited a node n and how promising it is. Let us define [16, 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.

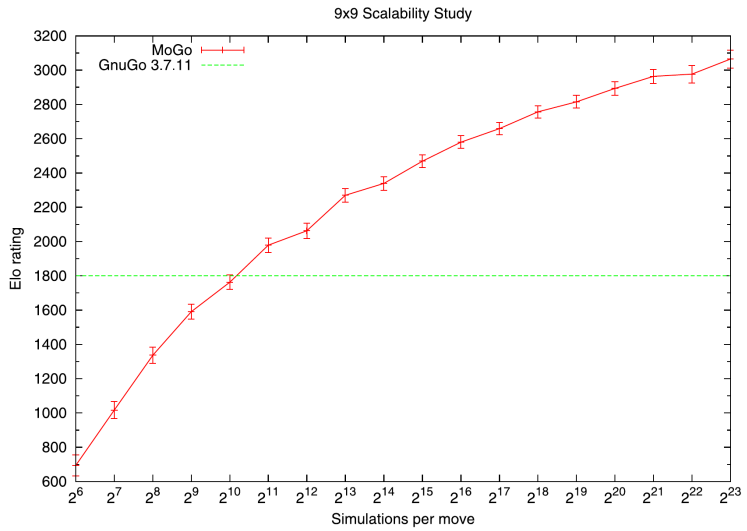


Figure 2.5: Elo rating of MoGo in relation to the computational resources granted to the algorithm [27, p. 1872]

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* 2.3.3 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 [26] for the game of Go. In 2011 they published a comprehensive paper [27] 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. [27, p. 1873]

One of the ideas introduced is Rapid Action Value Estimation (RAVE). It 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.

2.3 Learning agent

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." [16, p. 331]

2.3.1 Reinforcement learning

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". [15, p. 1]

Reinforcement learning is not a specific solution or method, all methods for "goal-directed agents interacting in an uncertain environment" [15, 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.

2.3.2 Markov decision processes

Central to reinforcement learning is a formalism for the environment called Markov decision process (MDP). Originally derived as an extension of Markov Chains by Richard Bellman [28, 29], a (finite) MDP describes sequential decision making where actions have long-term consequences, thus, affecting future rewards. The MDP is an abstraction of "goal directed learning from abstraction" that reduces the problem to three components: Actions, states and rewards that are being exchanged between the agent and the environment. We already introduced the agent-environment interaction loop 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.

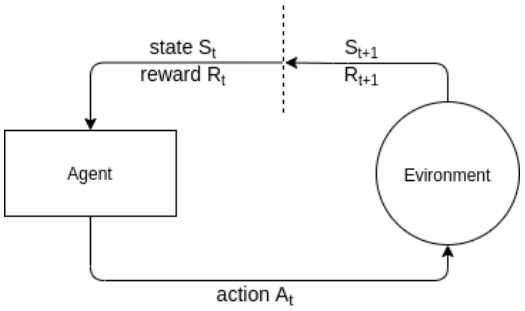


Figure 2.6: The markov decision process as agent-environment interaction loop

We divide the passage of time 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.

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 "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*". [15, 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). [15, 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:

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

  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 *Abstractpi* 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.

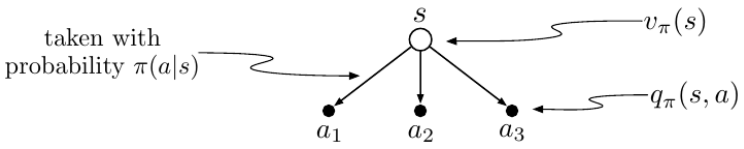


Figure 2.7: A visual explanation of policy, value and action-value [15, p.62]

- **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 $s \in mathcalS$. 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. For this optimal policy we also have an optimal action-value function q_* for all $s \in \mathcal{S}$:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{2.10}$$

2.3.3 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.

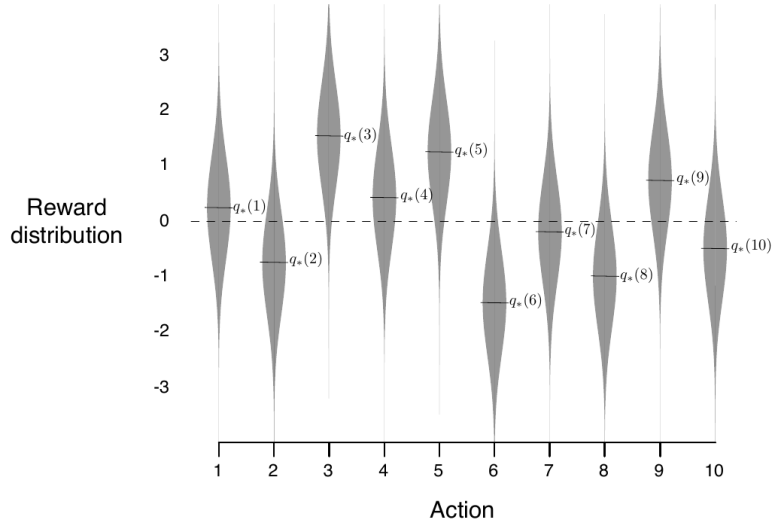


Figure 2.8: The reward distributions of a 10-armed bandit [15, p.28]

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

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.

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

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. 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:

$$A_t = \begin{cases} \operatorname{argmax}_a Q_t(a) & \text{with } 1-\varepsilon \\ \text{random } a & \text{with } \varepsilon \end{cases}$$

If we continue for an infinite number of times Q_t is guaranteed to converge to q_* as we sample each action 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.

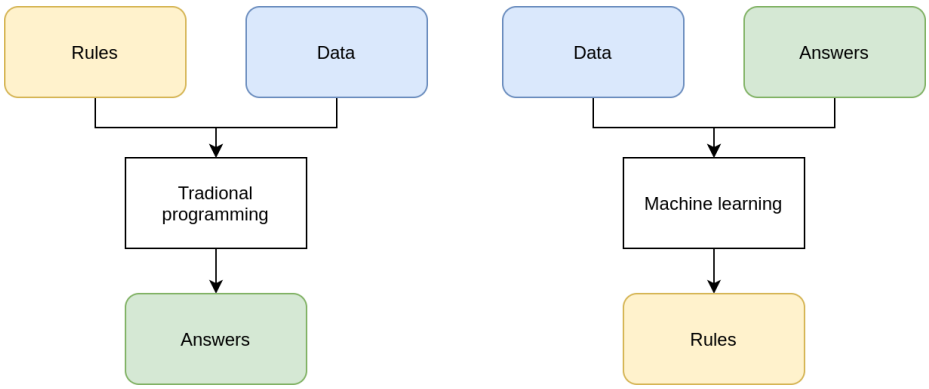


Figure 2.9: A fundamental shift in how we think about programming [30, p. 5f.]

2.3.4 (Deep) Neural networks

As reinforcement learning is a very general framework things like the value function $v(s)$ are just left as an abstract function. As neural networks are a general function approximator, we could use them to approximate the optimal value function $v_*(s)$ to maximize rewards. Hence, we should introduce the basic building blocks of this technique to gauge whether we might utilize them further.

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.

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.

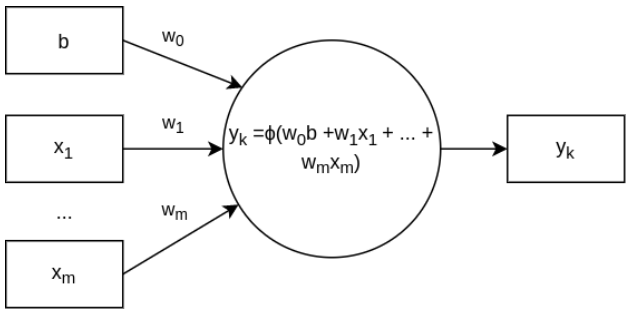


Figure 2.10: The fundamental idea behind neurons

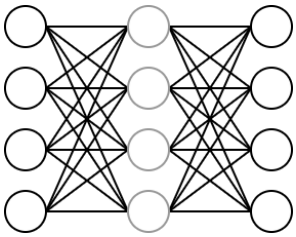


Figure 2.11: A small neural network arranged in layers. An input layer, a hidden layer and an output layer

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 2.10. 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 (2.11). To build such networks we need to generalize the linear function of the neuron for higher dimensional input:

$$y_k = \phi \left(\sum_{j=0}^{m} w_j x_j \right) \tag{2.11}$$

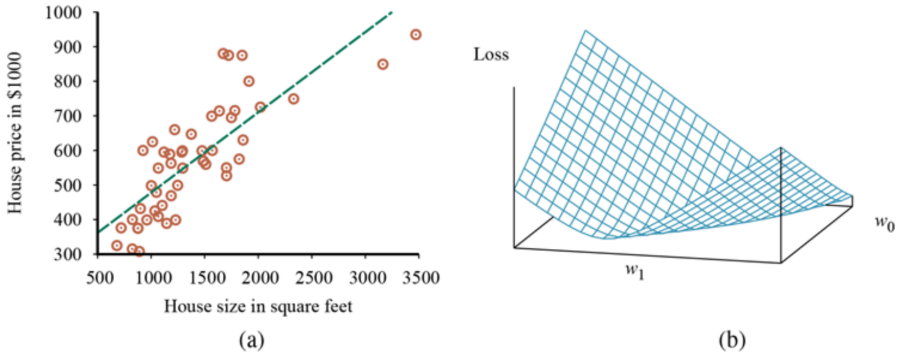


Figure 2.12: (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared-error loss: y = 0.232x + 246. (b) Plot of the loss function for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum. [16, p. 1251]

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:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(X_i - Y_i)^2 \tag{2.12}$$

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 2.12. 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 2.12(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 [16, p. 1253]

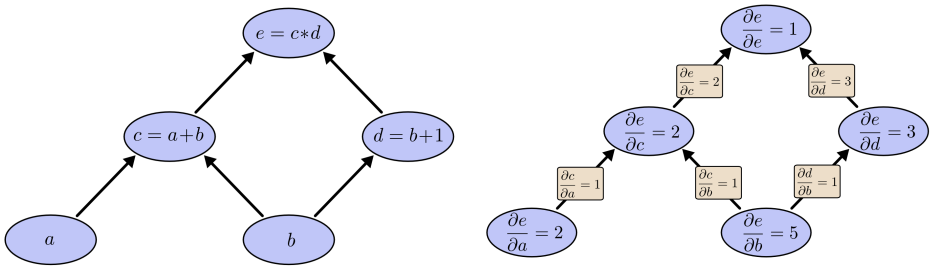
$w \leftarrow$ any point in the parameter space
while not converged **do**
  for each w_i in w **do**
    $w_i \leftarrow w_i - \alpha \frac{\partial f}{\partial w_i}(Loss(w))$
  end for
end while

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, by defining a neural network, we define a weight space. This space is essentially a room of all possible values of the weights. We can think of neural networks as functions or programs. Thus, the architecture of the neural network decides on a



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

Figure 2.13: An example of a computational graph

range of possible programs. Each of these programs is more or less suited for the task given. The feedback given by the loss function indicates the suitability. With gradient descent we search this space of programs for one that minimizes the error. Due to the nature of gradient descent, we are not guaranteed to find the globally optimal program.

2.3.5 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 2.11 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. As a result, permutations of the values of the image wouldn't change the result of the output.

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 256x256 would need 65536 weights for the first layer alone. Ideally we would also like the output of the network to be invariant to translation. This means that a pattern found in the corner should produce the same response as a pattern found in the center of the image.

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 function f." [31]

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

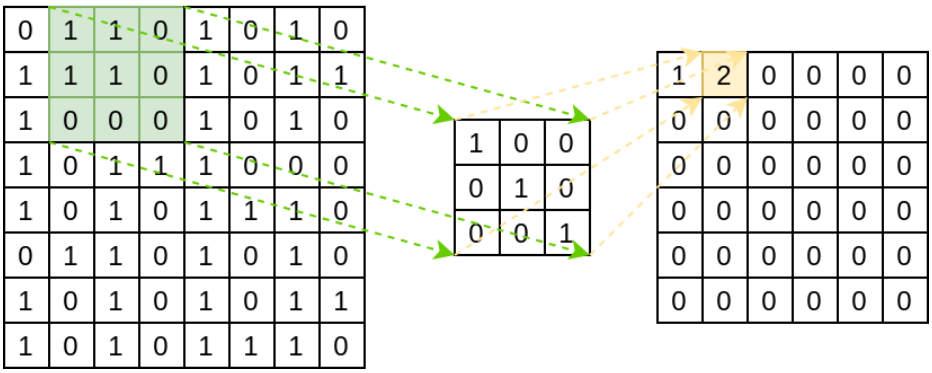


Figure 2.14: Convolution of a 8x8 black and white image with a 3x3 kernel, no padding and a stride of 1. [33]

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:

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

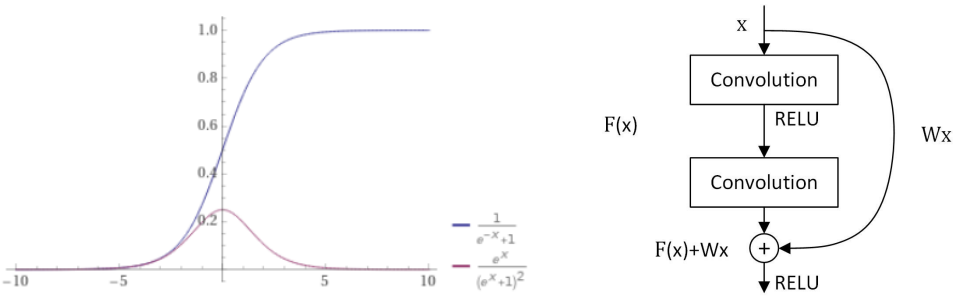
For a given coordinate u, v each value in the Kernel H is multiplied by the corresponding value of the image returned by function I. The figure 2.14 illustrates how a convolution would look on a black and white image. In image processing for example, we might use a Laplacian filter to extract certain 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. By combining multiple layers of convolutions the network can extract higher level abstractions of the image. [32]

The application of this method for image related tasks has become very popular. The first large scale application in 2012 by Krizhevsky et al. [34] 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.

2.3.6 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 the sigmoid function is in it's

limit 0 for $-\infty$ and 1 for ∞ and its derivative approaches 0 for both cases as depicted in figure 2.15 (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 [35]

(b) A residual block [36]

Figure 2.15: Motivation and implementation of residual networks

To mitigate this issue one could either use different activation functions like the ReLu or use residual connections. The value of the input before passing through the neuron is added with some weight to the output as shown in figure 2.15(b).

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 [37] and the combination of neural networks and gradient descent. To dive deeper the book "Artificial Intelligence: A Modern Approach" [16] offers a very good first impression.

2.3.7 AlphaGo

Now we have reached the culmination of our theoretical analysis. All the components we introduced so far constitute the knowledge necessary to understand AlphaGo and AlphaZero. Just like Abalone Go is a perfect information zero-sum board game. It already popped up in the complexity comparison 3.1, which clearly shows that Go is more complex in its state space and its search tree. 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 [38]. The only successful method at the time was MCTS, for which we already saw two influential papers in the context of Go and MCTS by David Silver.

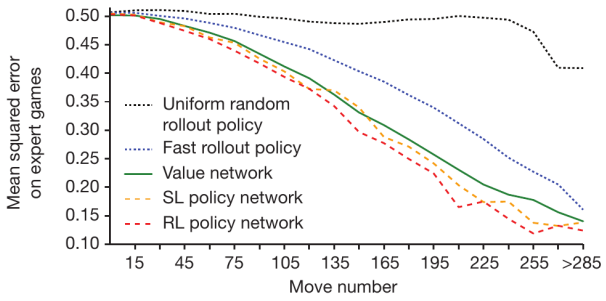


Figure 2.16: Comparison prediction quality of the different components [39]

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. Lookahead 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. The rollout policy network p_π is a relatively small network that is supposed to provide quick rollouts (simulations). As we already encountered for Abalone 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 2.16.

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" [39, 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 of the games are taken and the REINFORCE algorithm 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.

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 ex-

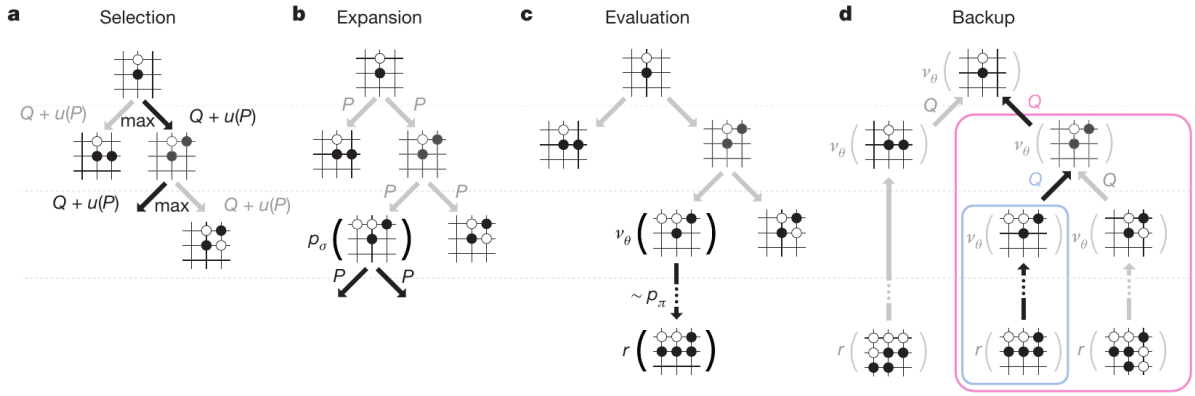


Figure 2.17: Monte Carlo Tree Search in AlphaGo. [39] Note the algorithm structure mentioned in figure 2.4 is maintained, but selection and simulation is more advanced

pected outcome of the game given the policy p_ρ: $v_\theta(s) \approx v_{p_\rho}(s)$. Figure 2.16 shows how v_θ appraches the same prediction accuracy as doing 15,000 rollouts with policy p_ρ.

During live play against an opponent AlphaGo uses MCTS to improve the performance versus just using the pure output of the policy network. The edges, or state s action a pairs, 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 2.17).

- 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 during selection can be described as t yielding:

$$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t,a) + u(s_t,a)) \tag{2.13}$$

  with the exploration term:

$$u(s,a) \propto \frac{P(s,a)}{1+N(s,a)} \tag{2.14}$$

- 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 be 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:

$$V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z_L \tag{2.15}$$

- 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:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) V(s_L^i) \tag{2.16}$$

  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" [39, p. 529].

The computational requirements for AlphaGo are significant. During training the team utilized 50 GPUs, the supervised learning stage ran for 3 weeks anyways. 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 entry to the topic is provided by the nature article "Mastering the game of Go with deep neural networks and tree search". [39]

2.3.8 AlphaZero

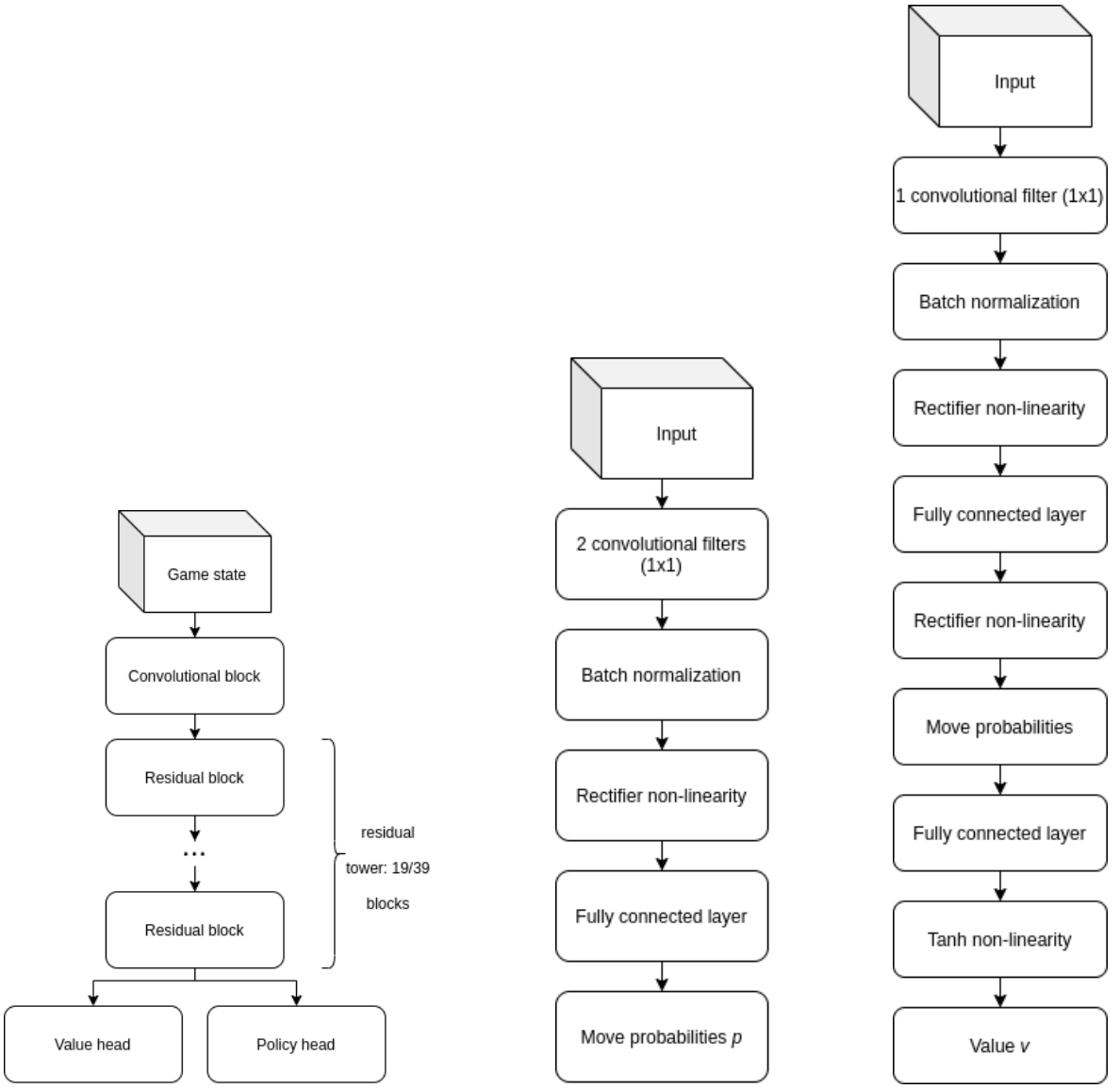
A problem with AlphaGo's architecture is its complexity. There are many hyperparameters for the four different neural networks and many other moving parts. AlphaZero is a simplification that conveniently performs even better and is easier to generalize.

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 architecture of the neural network 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 doesn't 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 for self-play.

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

As shown by figure 2.18 (a) the network is divided into four types of blocks:



(a) The general architecture of the network

(b) The layers of the value head

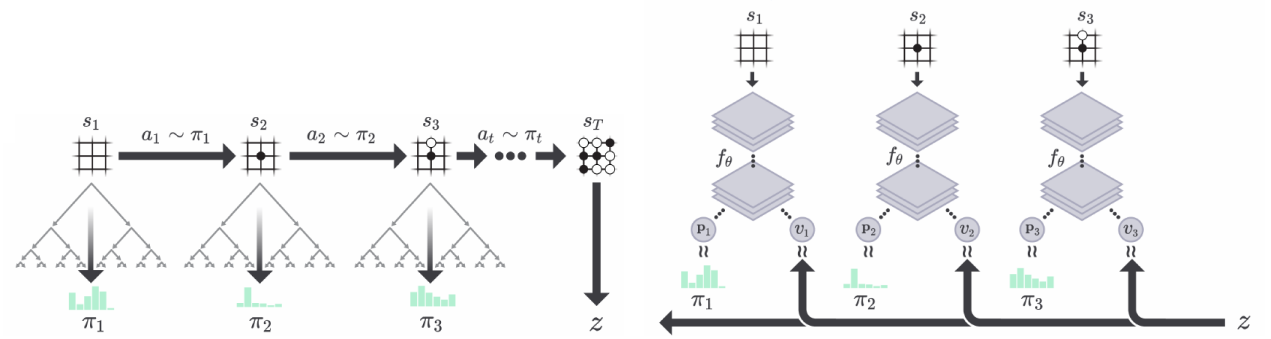
(c) The layers of the value head

Figure 2.18: The architecture of the neural network f_θ [6]

- The convolutional block and residual tower hold their main chunk of trainable parameters.
- 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* [40] architecture works

significantly better for Go than other architectures.



(a) The self-play

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

Figure 2.19: The self-play training pipeline of AlphaZero [6, p. 5]

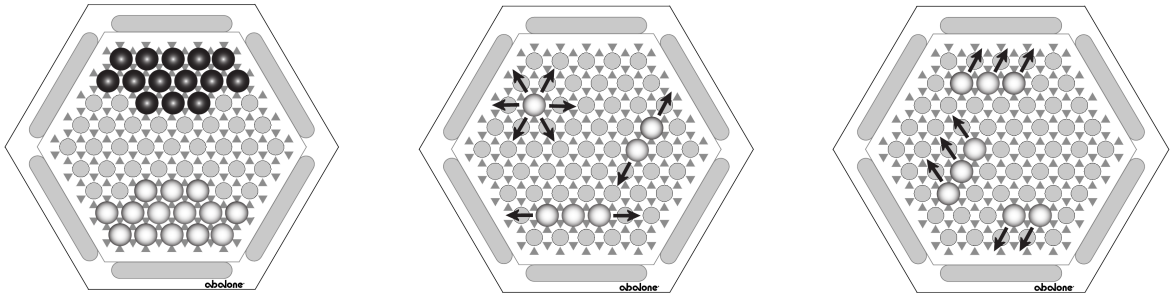
To train the network f_θ, it is initialized with random weights. At each turn i during self-play f_θ guides a MCTS, which produces a probability distribution π_t over all (legal) moves. The move a_i is selected according to the probabilities pi: a_i π_t. So instead of using the network f_θ directly to decide on the next move, MCTS produces an improved policy through repeated application of f_θ. The authors describe MCTS in this context as a *policy improvement* operator. The self-play games are played until a terminal state s_T is reached. The result of the game is $z_T \in \{-1, +1\}$ for either a loss or a win. For each move during the game he result of the game z_t is recorded together with the board state s_t and the search probabilities π_t, where $z_t \mp z_T$ depending on the players perspective: (s_t, π_t, z_t). The self-play is illustrated in figure 2.19 (a).

This tuple is stored in an experience buffer which is used for training the next iteration of the network. The parameters of the networks are fitted to mini-batches from the experience buffer that match the probabilities π_t for the policy head and the game result z_t for the value head. In total, $700,000$ mini batches of size $2,048$ from 4.9 million games were sampled.

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 [41].

3.1 Rules

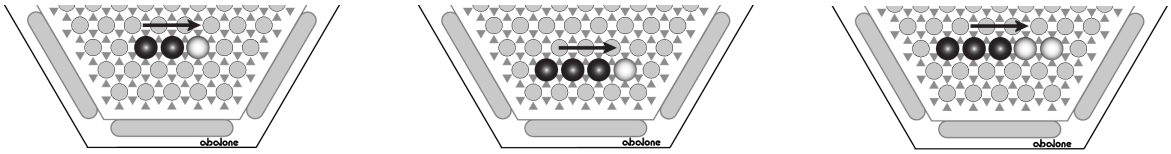


(a) Starting position (b) "In-line" moves (c) "Side-step" moves

Figure 3.1: Basic moves [42]

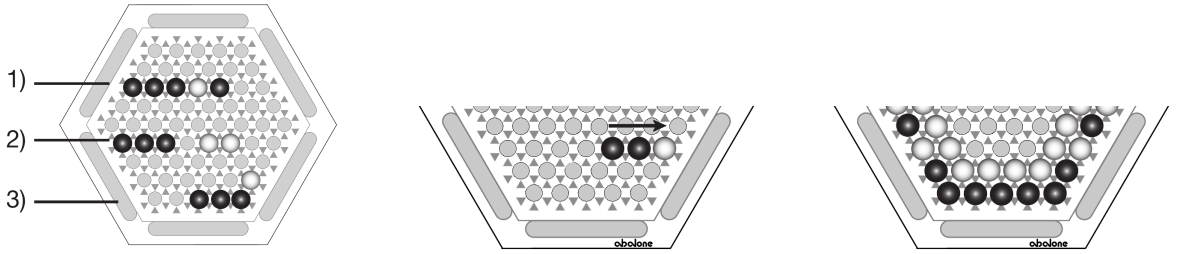
In the classical variant each player places 14 marbles on opposing sides. The game's starting position is depicted in figure 3.1 (a). There are also other starting positions like "German daisy" and "Belgian daisy", as well as four player variants, which will not be considered. One, two, or three adjacent marbles (of the player's own color) may be moved in one of the six possible directions during a player's turn. The marbles have to move in the same direction and can 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 and the opponent's marbles can not be blocked.



(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 [42]



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

Figure 3.3: Additional relevant board positions [42]

The figure 3.3 (a) shows ways in which a sumito might be blocked. 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 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 either limited by time or by 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. [16, 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 Complexity

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

State space complexity The state space is the set of all possible states "the environment can be in".[16, p. 150] For Abalone this means we have to consider all possible board configurations with different numbers of marbles present. Additionally, we would have to remove duplicates that arise from the symmetries of the board. In the case of abalone we have 6 rotations and 6 axes we can mirror the board on. The following formula gives us a good upper bound:

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

As the board has six rotational symmetries and additional six mirror axes we have to divide the result by 12 which results in a total of 6×10^{23} possible board configurations. [43, p. 4]

Game tree complexity The game tree 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. In the case of Abalone the game tree is unbound and has an infinite height as actions might be taken repeatedly forming loops. Therefore, Abalone's complexity can only be approximated by an average search tree not the game 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 3.4. The depth d of the tree depends on the number of turns per game. Looking at the average again a game takes in the region of $d = 87$ turns. To get a measure of the complexity the number of nodes in a tree is given by

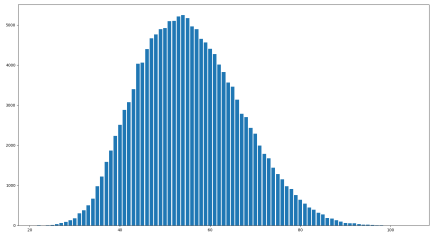


Figure 3.4: Counts of moves available a random player in 5 games

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 [44]

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

resulting in a total of $60^{87} \approx 5 \times 10^{154}$ nodes. [43]

As those numbers in isolation are hard to grasp it is useful to put Abalone's complexity in relation with 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)

3.4 Existing game playing agents

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, ... [45, p. 64]

There are multiple implementations of minimax for Abalone [44, 43, 46, 47], but software is only openly available for ABA-PRO by Tino Werner from 2002 [48]. There are a few mentions of (commercial) Abalone programs like RandomAba (Random Soft), AliAba, AbaloneNet mentioned by Lemmens [43, p. 7] but those have not been attainable through the internet. Even though ABA-PRO was not the strongest algorithm at the time, it's 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. [45] 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 [49]. This reimplementation in Java [50] is also the reference for later benchmarks as it is open source [50] and thus allows programmatic interaction.

The investigations into MCTS in the context of Abalone are quite limited so far. Pascal Chorus has undertaken a comparison of the vanilla implementation against a heuristic agent, showing the dominance of the heuristic agent. [44] 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 very weak signals. Without any more sophisticated rollout-policy this approach does not work very well.

"Abalearn" was the first learning-based approach to playing Abalone. [51] 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") [52] and for Abalone the authors achieved to draw ABA-PRO up to a search depth of five. An interesting feature of their approach is the fact that they exclusively used self-play for training 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. [53]

4 System architecture

Well equipped with all the basic theoretical tools that we need, we can move on to implementing them for the game of Abalone. A 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 RL-agent architecture

4.2 Software

4.2.1 Training framework

As 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 foudation.

4.3 Training pipeline

5 Experiments and results

6 Conclusion

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 [54] ("Huang's Law" [55]). 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 [56] or Jim Keller's Grayskull. [57] The same reason reignited interest in optical computing accelerators to bring drastic changes in power requirements and performance for matrix multiplication. [58, 59]

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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

[25] P. Auer and N. Cesa-Bianchi, "Finite-time Analysis of the Multiarmed Bandit Problem," p. 22.

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

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

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

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

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

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

[32] 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.

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

[34] 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.

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

[36] D. Li, A. Cong, and S. Guo, "Sewer damage detection from imbalanced CCTV inspection data using deep convolutional neural networks with hierarchical classification," *Automation in Construction*, vol. 101, pp. 199–208, May 2019.

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

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

[39] 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.

[40] 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.

[41] "Abalone (board game)," https://en.wikipedia.org/w/index.php?title=Abalone_(board_game)&oldid=9 Dec. 2020.

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

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

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

[45] 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.

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

[47] 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.

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

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

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

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

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

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

[54] 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.

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

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

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

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

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