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"

December 21, 2021

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, December 21, 2021 Signature

Contents

Abstract

Explanation

1 Introduction

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], 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 our 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 unfeasable at a certain point. A different approach would be devising (general) methods to learn the necessary domain knowledge from scratch, *tablula 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 milestones 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 copmuter vision or natural language processing. Namely those are neural networks and (stochastic) gradient descent. 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 not possible for humans. AlphaGo used (deep) neural networks and gradient descent to train a evaluation function based on a large database of expert moves. With the help of Monte Carlo Tree Search they used this function to play against itself and improve further. [6]

Building on this success DeepMind, the company behind AlphaGo, further improved the architecture. "AlphaGo Zero" and the generalization "AlphaZero" learn, without the help of the database of expert moves and surpassed the performance of AlphaGo significantly. Since then the architecture has been applied to Chess, Shogi and Atari games by removing the last piece of human knowledge in the system: The rules of the game. [7]

At this point our endeauvor begins, as the purpose of this writing is to apply the methods of AlphaZero to the game of Abalone.

1.1 Research goals

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

The 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.

Sub-goal 1 is to compare classical search based methods to this AlphaZero's deep reinforcement learning based on several criteria such as win/loss ratio, computational requirements, etc.

2 Analysis

Before we move to the nuts and bolts of AlphaZero and our concrete implementation, 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

Since the late 1960s, many artificial intelligence researchers have assumed that there are no universal principles to be uncovered, and that intelligence is instead based on a large variety of specialized techniques, processes, and heuristics. It was once stated that if we could merely feed a machine enough relevant data, say one million or one billion, it would become intelligent. "Weak techniques" were defined as those based on generic principles, such as search or learning, and "strong methods" were defined as those based on specialized knowledge. Today, this point of view is unusual. It was premature, in our opinion, because too little effort had been put into the search for universal principles to come to the conclusion that there were none. Much of today's artificial intelligence research is focused on discovering broad principles of learning, search, and decision-making. It's unclear how far back the pendulum will swing, but reinforcement learning research is undoubtedly contributing to a return to artificial intelligence's simpler and fewer general principles. [8]

2.1.1 Rational agent

Stemming from the latin word *agere* meaning "to act", an agent is something that acts. As we expect our agent to take sensible or intelligent actions we further qualify this definition by calling it rational. This means that it acts so as "to achieve the best outcome or, when there is uncertainty, the best expected outcome". [9, p. 36]

The agent exists in an environment which it perceives through sensors and it takes actions through its actuators. We refer to the content of the sensor's output for one observation as *percept*. The cat uses eyes, ears and other organs to perceive the world

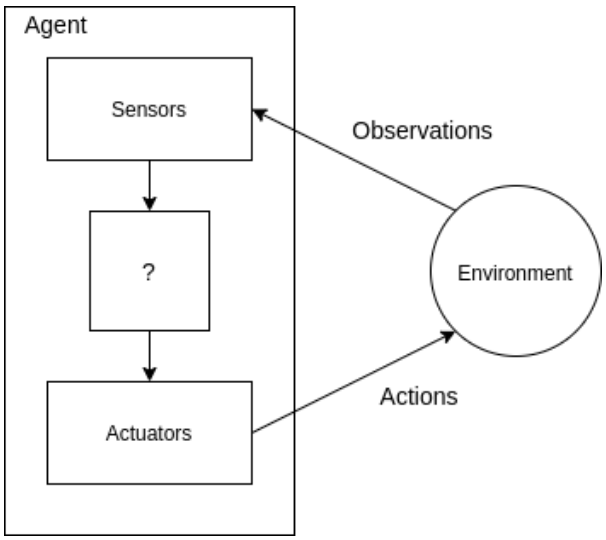


Figure 2.1: The agent-environment interaction loop

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.

Internally our 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 our environment as a problem or *task* our 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.

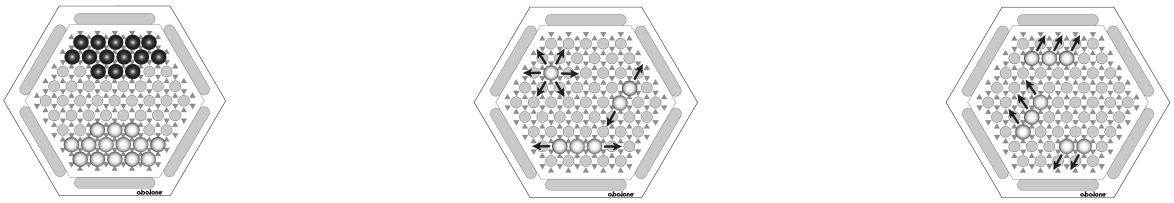
2.2 Environment

Now that we have a general understanding of agents and environments, we can use this knowledge to have a closer look at Abalone. It is a fairly new game that was devised in 1987 by Michel Lalet and Laurent Lévi. Nevertheless, with more than four million global sales it has established itself as a classic game [10]. Abalone is a two-player

game consisting of a hexagonal board with 61 fields and 14 marbles for black and white respectively.

2.2.1 Abalone rules

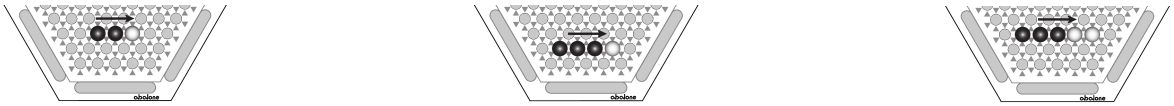
The goal of the game is to push six of the opponent's marbles off the playing field. The game's starting position is depicted in figure 2.2 (a). One, two, or three adjacent marbles (of the player's own color) may be moved in any of the six possible directions during a player's turn. We differentiate between broadside or "side-step" moves and "in-line" moves, depending on how the chain of marbles moves relative to its direction, which is shown in figure 2.2 (b) and (c).



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

Figure 2.2: Basic moves [11]

A move pushing the opponent's marbles is called "sumito" and comes in three variations, as shown by figure 2.3. Essentially, the player has to push with superior numbers and the opponent's marbles can not be blocked. This is the game mechanic that allows for pushing the marbles out of the game and winning.



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

Figure 2.3: Sumito positions allow pushing the opponent's marbles [11]

2.2.2 PEAS and task properties

Based on the PEAS framework we can specify Abalone as a task environment and show the key components for the implementation of our agent. [9, p.107]

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

Environment Digital playing board

Actuators Move marbles, display text to CLI

Sensors Position of marbles

There are a few categorizations 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 Abalone 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. The autonomous vehicle for instance is dealing with continuous time and also continuous states. The speeds of the car take a smooth range of real values and time can be meaningfully split into increasingly small intervals. However, Abalone is 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.

The actions that the agents take 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. In Abalone none of these issues arise as it is deterministic.

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. In the case of Abalone, moves taken have long drawn out consequences for later stages of the game.

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. As any turn-based game, Abalone is a *static* environment, as the board only changes after 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 Abalone is a *multi-agent* environment. We also have to distinguish whether those multiple agents compete for

the performance measure. In Abalone, 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.

Summing this up **Abalone is a fully observable, deterministic, two-agent, competitive, sequential, static and discrete environment**. Another popular term for this type of environment is a *perfect information zero-sum game*.

2.2.3 Abalone 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.[9, 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:

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

Game tree complexity The game tree defines all transitions between board positions (nodes) through moves (edges). 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. To get a measure of the complexity the number of nodes in a tree is given by:

$$b^d$$

First we consider the branching factor b, or the number of possible moves for any given state. We can only approximate this, as this number greatly varies between different states. On average abalone has $b = 60$ possible moves per state as measured in figure 2.4. 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, giving us a total of 60^{87} nodes. To be precise this is the complexity of an average search tree not the game tree, as mentioned above. [12]

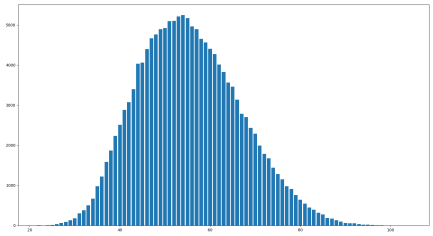


Figure 2.4: Counts of moves available for random for random player in 5 games

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 2.1: Abalone in comparison with other games [13]

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 2.1)

2.3 Classic agent

Now that we have an intricate understanding of the environment for our agent 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 we 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 [14], but was described most comprehensively by Claude Shannon in 1950 in "programming a Computer for Playing Chess". [15]

2.3.1 Minimax algorithm

Minimax assumes two roles: The minimizer and the maximizer. We start the search from the current board state as the role of the maximizer and then alternate between

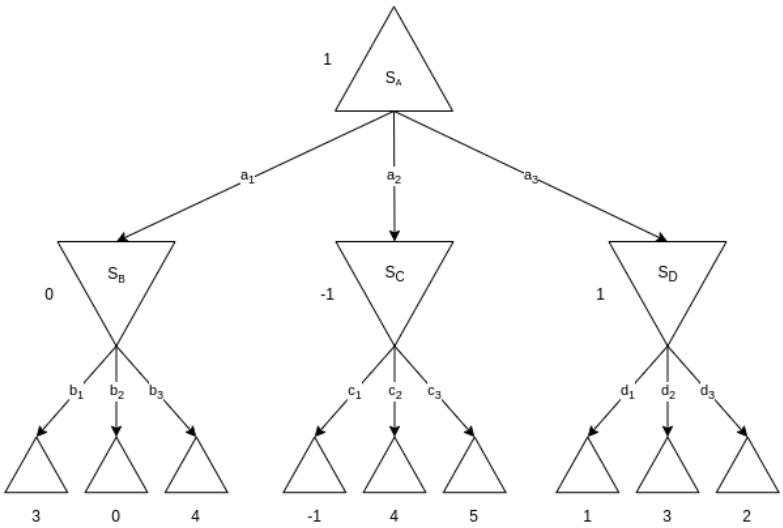


Figure 2.5: Minimax for a small search tree, resulting in an utility value of 1

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

- 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 abalone 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
- result(s, a) returns the resulting state if in state s and taking action a

such that we can define minimax recursively as:

$$\text{minimax}(s) = \begin{cases} \text{utility}(s, max) & \text{is-terminal}(s) \\ \max_{a \in \text{actions}(s)} \text{minimax}(\text{result}(s, a)) & \text{to-move}(s) == max \\ \min_{a \in \text{actions}(s)} \text{minimax}(\text{result}(s, a)) & \text{to-move}(s) == min \end{cases}$$

Putting this together we can see in figure 2.5 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.

2.3.2 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 $60^4 = 1.2960.000$ nodes at a depth of $d = 4$ in the case of an average Abalone 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 Abalone might be a linear combination in the form of:

$h(s) = \omega_0 f_0(s) + ... + \omega_n f_n(s)$

With functions f_i calculating different values such as

- 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).
- 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 is we need expert knowledge and a lot of empirical testing to find a suitable heuristic. In some cases like with 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 this heuristic based adversarial search.

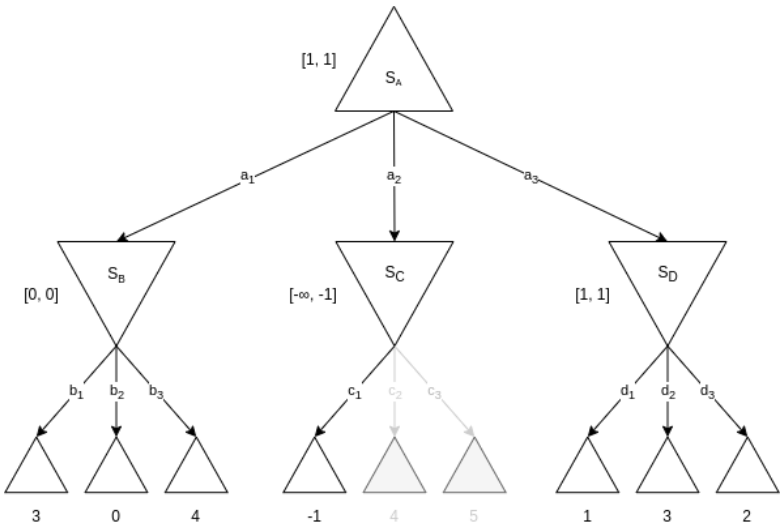


Figure 2.6: Our previous example but with alpha beta pruning applied. The grayed out nodes indicate, that these in fact could be pruned from the tree

2.3.3 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 2.6 will help us illustrate this principle. Our search revealed that choosing move a_1 will yield us an 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 $O(\sqrt{b^d})$ is entirely dependent on this ordering. We could find different ways of ranking the moves:

- *Killer move heuristic* prioritizes that are usually undoubtedly good like taking a marble in Abalone.
- *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 or Abalone minimax has been very successful, because humans could devise meaningful heuristic functions. For chess the engine stockfish has been the most successful computer player for a long time and is based on this algorithm (and many optimizations). [16, 17] In Abalone there are multiple implementations, the most successful has been ABA-PRO by Tino Werner from 2002 [18] which has been the reference for other algorithms such as Abalearn [19]. A more recent publication by Papadopoulos et. al. claimed to have devised a more successful player than Werner. [20] This was confirmed in a very recent reimplementation thesis by Michiel Verloop [21]. This reimplementation in Java [22] is also the reference for later benchmarks.

2.3.4 Monte Carlo Tree Search

For games like Go we cannot find powerful heuristic functions, which makes the previous approach of minimax not a viable option. In addition, the initial position of a 19x19 Go board is 361 decreasing only by one for each stone placed. A method proposed in 2006 by Coulom [23] called Monte Carlo Tree search. The main idea is to use simulations or *rollouts* 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 [24] showed that it does in fact converge to optimal play. For games like Abalone 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 can be structured into four stages:

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 $N(n)$.

The more we repeated this cycle, the more certainty we gained about the optimal move to take.

The inception 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 [26]. To select the next node he estimated the probability of that move being better than the current best move. We order the moves by their current estimated value μ_i and calculate the corresponding variance σ_i^2 getting the ordering of $\mu_0 > \mu_1 > ... > \mu_N$. Selection of a move is proportional to:

$$u_i = exp\left(-2.4\frac{\mu_0 - \mu_i}{\sqrt{2(\sigma_0^2 + \sigma_i^2)}}\right) + \epsilon_i$$

The Term ϵ_i ensures that the probability of selection never goes to zero as defined by $\epsilon_i = \frac{0.1+2^{-i}+a_i}{N}$. The resulting distribution is similar to the Gaussian distribution and the Boltzman equations [23]

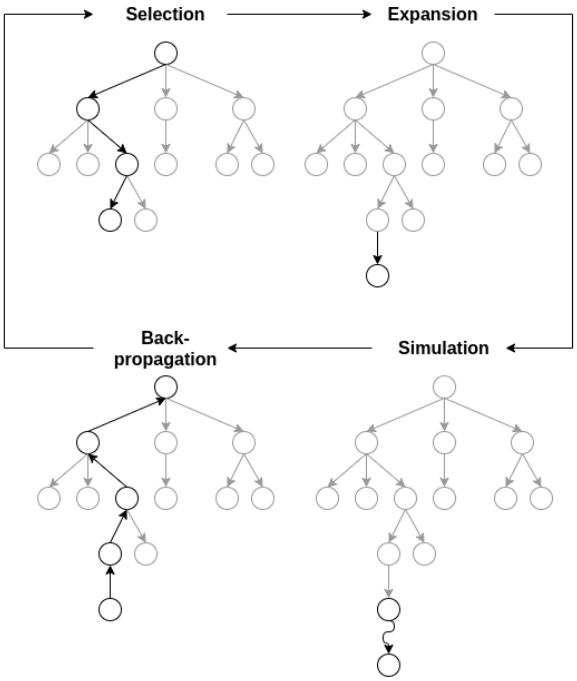


Figure 2.7: Monte Carlo tree search stages [25]

Another idea for selection is the UCB1 formula [27], that weights how often we have visited a node and how promising it is.

$$\text{UCB1}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(Parent(n))}{N(n)}}$$

The cumulative utility $U(n)$ is normalized by the number of times we have visited the node $N(u)$. 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 exploitation term. The more often we visit a node, 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* 2.4.3 that we will inspect more closely later.

Here we can already see David Silver's handwriting on the wall. As early as 2006 he, and Sylvian Gelly, investigated optimization to MCTS [28] for the game of Go. In 2011 they published a comprehensive paper [29] proposing the algorithm MoGo and evaluating different strategies to improve the effectiveness of MCTS in Go. Seeing that

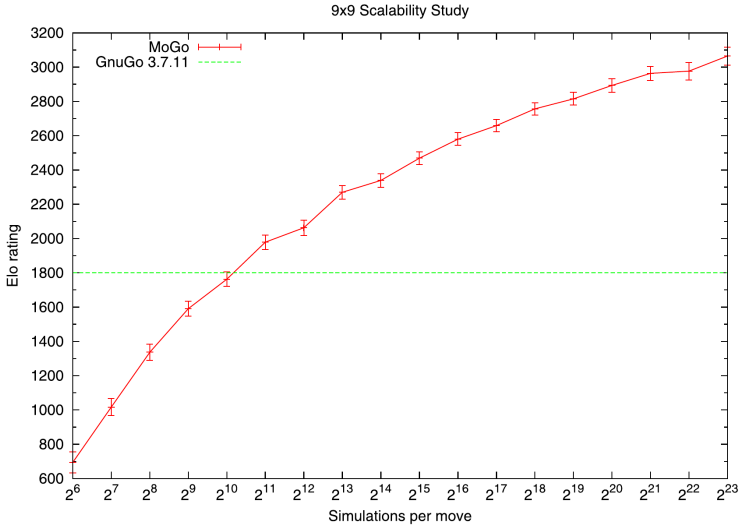


Figure 2.8: Elo rating of MoGo in relation to the computational resources granted to the algorithm [29]

> [...] 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.

One of the ideas introduced is Rapid Action Value Estimation (RAVE). We already saw how we could reuse information gathered for minimax through a transposition table. In our search tree we will encounter transpositions for that we already performed searches. RAVE allows us 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 2.8.

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. [13] 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.

2.4 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. We shift our focus now 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." [9]

2.4.1 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 Abalone 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". [8]

Reinforcement learning is not a specific solution or method for this problem domain, all methods for "goal-directed agents interacting in an uncertain environment" are types of reinforcement learning. It is a general formalism that will help us reframe the problem of playing Abalone (well) in a new light.

2.4.2 Markov decision process

For reinforcement learning we also need to find a formalism for the environment. Originally derived as an extension of Markov Chains by Richard Bellman [30, 31], the (finite) Markov Decision Process (MDP) describes sequential decision making where actions have long-term consequences, thus, effect future rewards. The MDP is an abstraction of "goal directed learning from abstraction" that reduces the problem to three signals of 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 2.9.

We devide the passage of time into discrete timesteps t at which the agent senses the state $S_t \in \mathcal{S}$ and then selects some action $A_t \in \mathcal{A}(s)$. Resulting in that action is some

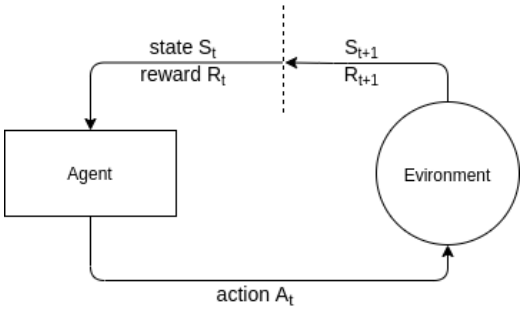


Figure 2.9: The markov decision process as agent-environment interaction loop

reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ that is recieved in the next timestep. For a *finite* MDP the sets $\mathcal{A}, \mathcal{S}$ and $\mathcal{R}$ have a finite amount of elements.

The transitions between the state s and the next state s' is given by the function p which essentially defines the decision process as a whole:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|s_{t-1} = s, A_{t-1} = a\}$$

As the random variables R_t and S_t only depend on the previous state and action, "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*". [8, p. 48]

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).

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

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

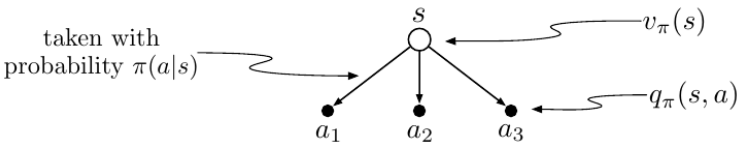


Figure 2.10: A visual explanation of policy, value and action-value [8, p.62]

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:

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

The policy is a mapping between the perceived states to probabilities for 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 2.1 in Abalone this would not be feasible, therefore we used search processes as a policy so far.

We denote the policy as the function π 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. 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:

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

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

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

The figure 2.10 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 our 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 maximzes the future reward for each action and state. For MDPs we define the *optimal policy* for all $s \in mathcalS$ by:

$$v_*(s) \doteq \max_\pi v_\pi(s)$$

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

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

2.4.3 Exploration vs. Exploitation

As the agent builds its knowledge while it is engaged in the 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 recieve some reward R_t. Each lever has some stationary distribution over the rewards, that is hidden from us. The distribution looks the the one given in figure 2.11.

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

$$Q_t(a) \doteq \frac{R_t(a)}{N_t a}$$

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

$$A_t \doteq \operatorname*{argmmax}_a Q_t(a)$$

Initially, we might choose a random lever and that yields a reward of 0. As all Q_t are still equivalent we choose another random lever giving us a reward of 1. Then ,according

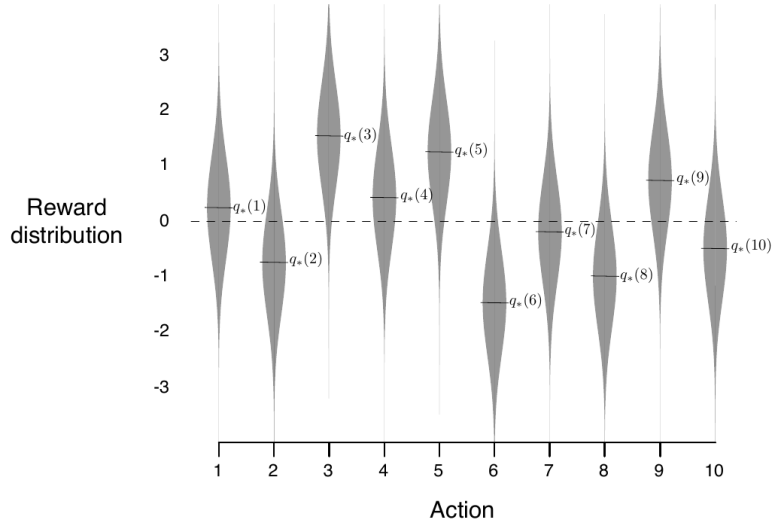


Figure 2.11: The reward distributions of a 10-armed bandit [8, p.28]

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:

$$A_t = \begin{cases} \text{greedy } a & p = 1 - \epsilon \\ \text{other } a & p = \epsilon \end{cases}$$

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.

2.4.4 (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 used the data and predefined rules

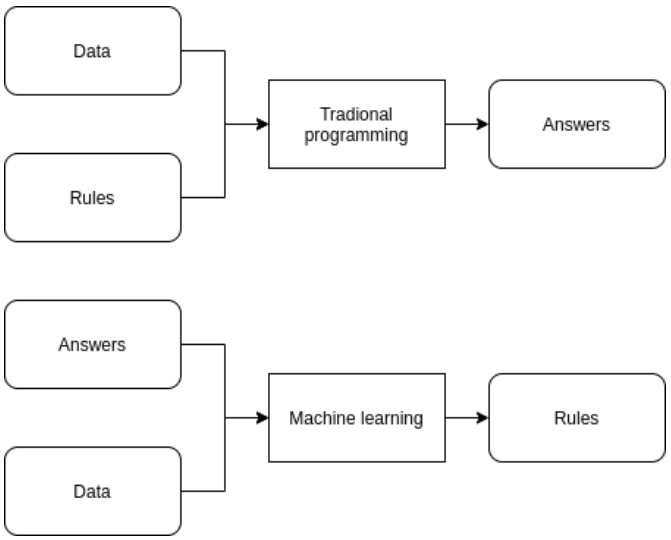


Figure 2.12: A fundatmental shift in how we think about programming

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 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 2.12 contrasts this change in programming. [32]

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 predicions about one dimensional input data. For example we might have living space of houses in m^2 X_i and their corresponding price Y_i. We could choose the bias w_0 and the *weight* w_1 such that the squared error for the given data 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.

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

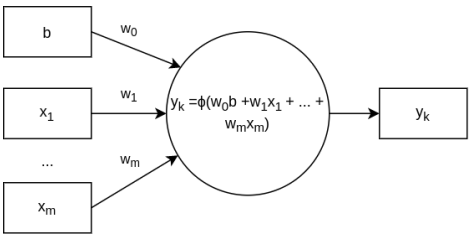


Figure 2.13: The fundamental idea behind neurons

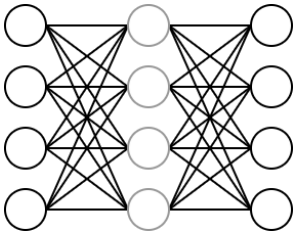


Figure 2.14: A small neural network arranged in layers. An input layer, a hidden layer and an output layer

- 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 2.13. 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 (2.14). To build such networks we need to generalize the linear function of the neuron for higher dimensional input:

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

But this poses a problem. We can find globally optimal solutions with linear regression and the like, but as the size of the network grows, the computational cost of these methods grows too large. 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 perdictions and the desired output:

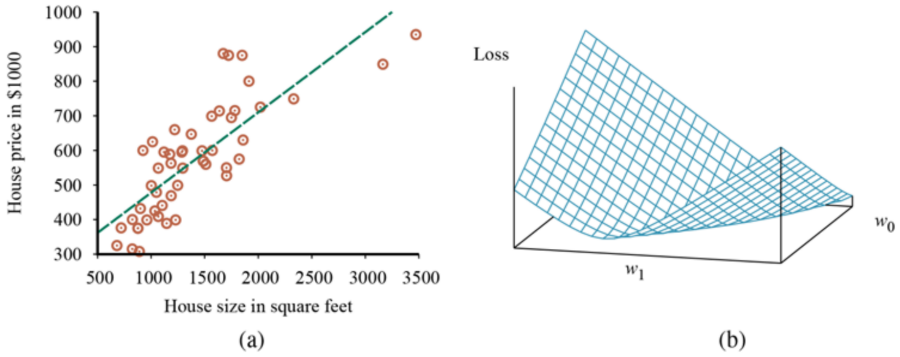


Figure 2.15: (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared-error loss: y = 0.232x + 246. (b) Plot of the loss function for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum. [9, p. 1251]

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

This is one type of *loss function*, which in general are 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_i we can find out how well the current configuration of the network fits the data.

Lets 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 parabula in figure 2.15. Ideally, we want to walk down into the valley where the error in 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 2.15(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 [9, p. 1253]

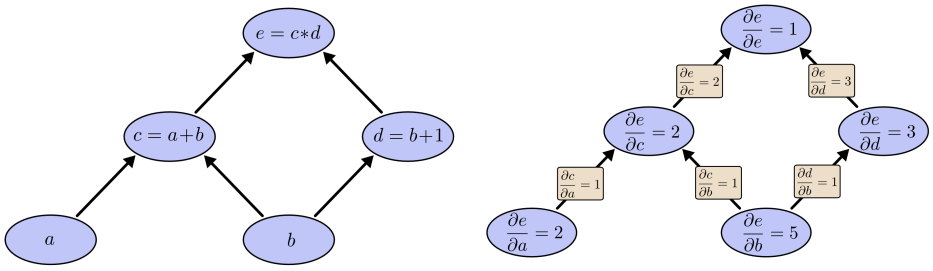
$w \leftarrow$ any point in the parameter space
while not converged **do**
  for each w_i in w **do**
    $w_i \leftarrow w_i - \alpha \frac{\partial f}{\partial w_i}(Loss(w))$
  end for
end while

The variable α for gradient descent describes the *learning rate*, so by how much we update the weights. The problem is, as soon as we start combining multiple neurons the intuition gained in calculus on how to calculate the gradient breaks down. Luckily our tools still suffice, we just need to inspect the network more closely. Lets constrain our range of possible neural networks by two conditions:

- 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 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 2.16 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 permutations of the weights and the network as a whole. We can think of neural think of neural networks as functions of programs. Thus, the archtecture of the neural network decides on a range of possible programs. Each



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

Figure 2.16: An example of a computational graph

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.

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 couse), 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” and the combination of neural networks and gradient descent. To dive deeper the book ”Artificial Intelligence: A Modern Approach” [9] offers a very good first impression.

2.4.5 Convolutional Neural Network

2.4.6 AlphaGo and AlphaZero

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.

3 System architecture

3.1 RL-agent architecture

3.2 Software

3.2.1 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.

3.3 Training pipeline

4 Experiments and results

5 Conclusion

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] 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.

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

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

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

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

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

[14] 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.

[15] 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.

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

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

[18] 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.

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

[20] 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.

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

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

[23] 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.

[24] 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.

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

[26] 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.

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

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

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

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

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

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