MASTERARBEIT | MASTER'S THESIS

Titel | Title

Esports Trainer

verfasst von | submitted by

David Cömert BEd

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Education (MEd)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 199 514 520 02
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Lehramt Sek (AB) Unterrichtsfach Informatik Unterrichtsfach Mathematik
Betreut von | Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Acknowledgements

First of all, a big thank you to my supervisor, Univ.-Prof. Dr. Helmut Hlavacs, who has been really patient with me throughout the process of writing this thesis and has supported me every time I needed advice. Thanks to Pia and Thomas, who are experts in the field of machine learning and mathematics. They helped me with some complications, proofread my work and gave me some great feedback. I would also like to thank my mother for reading this thesis without any technical background and giving me some valuable feedback. Last but not least, I would like to thank to my girlfriend Bettina, who was patient during the holidays while I was checking the logs of the training sessions. Not only that, but the mental support and cat care during the trip was very helpful.

Abstract

The aim of this master's thesis is to find a way in which artificial intelligence (AI) can help to improve human game performance. The AI itself has to understand the context of the game. This is done by learning in the specific game with various methods from machine learning. This AI must significantly exceed the capabilities of the human player in order to provide valuable support and feedback. Reinforcement learning is at the heart of all game AIs and a sub category of machine learning. The principle is similar to human learning and is based on a trial-and-error concept. This is further enhanced by neural networks.

One of the best AIs currently available in the field of computer games is MuZero, a direct successor to the well-known AlphaZero algorithm. By playing against itself, it is possible to improve to such an extent that it far surpasses the abilities of humans. MuZero was used as a foundation for analyzing the human games in this thesis.

The first step was to create the AI by providing a working AI. Compromises have to be made compared to the state-of-the-art. The biggest difference from other related work is the hardware limitation. This results in much smaller models. This AI performs well in the given use cases, but not perfect. Different methods were used to extract information from AI. The first method, *Perfect Game*, compares the actions of the player and the AI. The greater the deviation, the worse the player's action is rated. This method is particularly suitable for imitating the AI's moves. However, it is only recommended for players who have already reached a high skill level. This method is often characterized by opaque reasons and is difficult to implement for humans, because there are some times counterintuitive, which is why other methods should also be used. Another way is to use similar good actions and evaluate them positively. *Value only Game*, describes the method of evaluating individual positions in the game by only taking the value of the current and future observations. With this evaluation method, actions can be differentiated granularly, and therefore good actions are also used and not just the best ones. This method is less suitable for very good players, as the AI is weaker without the learned action patterns. The last method, *Easy Game*, focuses on finding the simplest possible moves that still lead to a good result. These methods were examined and compared using theoretical and practical examples. Each of them has other use cases and different parameters to fine tune the results.

The results show that these methods are applicable and that the players can analyze their games at a high level and thus work on their skills, at least with a few caveats. One of them is, that the AI has to outperform the human consistently. Another one is, that it is necessary to train the AI beforehand, which also includes the need of an environment and the hardware.

Kurzfassung

Ziel dieser Masterarbeit ist es, eine Möglichkeit zu finden, wie ein Spieler oder eine Spielerin durch künstliche Intelligenz (KI) in einem Computerspiel sich verbessern kann. In einem ersten Schritt muss diese KI die Fähigkeiten des Menschen deutlich übertreffen, um eine wertvolle Unterstützung bieten zu können. Reinforcement Learning ist das Herzstück jeder Spiel-KI. Das Prinzip ähnelt dem menschlichen Lernen und basiert auf einem Versuch-und-Irrtum-Konzept. Es wird durch neuronale Netze weiter verbessert.

Eine der derzeit besten KI im Bereich der Computerspiele ist MuZero, ein direkter Nachfolger des bekannten AlphaZero-Algorithmus. Durch Spiele gegen sich selbst ist es möglich, dass sich die KI so weit zu verbessert, dass es die Fähigkeiten des Menschen bei weitem übertrifft. MuZero ist die grundlegende AI in dieser Arbeit, um die Spiele von Menschen zu analysieren.

Der erste Schritt war die Erstellung der KI. Diese KI unterscheidet sich etwas vom Stand der Forschung, da die Bedingungen nicht die gleichen sind. Die KI-Modelle sind aufgrund von Hardwarebeschränkungen deutlich kleiner. Dadurch erreicht die KI nur gute und keine perfekten Ergebnisse. Es wurden verschiedene Methoden verwendet, um Informationen aus der KI zu extrahieren. Die erste Methode *Perfect Game* vergleicht die Aktionen des Menschen und der KI, je größer die Abweichung, desto schlechter wird die Aktion des Menschen bewertet. Diese Methode eignet sich besonders, um die Spielzüge der KI nachzuahmen. Sie ist jedoch nur für Spielerinnen und Spieler zu empfehlen, die bereits ein hohes Fähigkeitsniveau erreicht haben. Da diese Methode oft intransparent und schwer umsetzbar ist, wurde nach Methoden gesucht, die auch ähnlich gute Aktionen vorschlagen und positiv bewerten. *Value only Game* beschreibt die Methode, einzelne Positionen im Spiel zu bewerten, ohne auf erlernte Handlungsmuster zurückzugreifen. Mit dieser Bewertungsmethode können Aktionen granular unterschieden werden und somit werden auch gute und nicht nur die besten Aktionen verwendet. Für sehr gute Spielerinnen und Spieler ist diese Methode eher weniger geeignet, da die KI ohne die gelernten Handlungsmuster schwächer ist. Die letzte Methode *Easy Game* konzentriert sich darauf, möglichst einfache Spielzüge zu finden, die trotzdem zu einem guten Ergebnis führen. Diese Methoden wurden anhand von Praxisbeispielen untersucht und verglichen. Jede dieser Methoden hat bestimmte Anwendungsmöglichkeiten und verschiedene Parameter, um die Ergebnisse zu optimieren.

Die Ergebnisse zeigen, dass diese Methoden anwendbar sind und dass Spielerinnen und Spieler ihre Spiele auf hohem Niveau analysieren und dadurch ihre Fähigkeiten verbessern können. Wichtig ist, dass die KI deutlich besser spielt als der Mensch und dass die KI vorher trainiert werden muss. Dieses Training setzt auch das Vorhandensein einer Trainingsumgebung und entsprechender Hardware voraus.

Contents

Acknowledgements i

Abstract iii

Kurzfassung v

List of Tables ix

List of Figures xi

List of Algorithms xiii

1 Introduction 1

2 Related Work 3
- 2.1 Reinforcement Learning (RL) . . . 3
  - 2.1.1 Model-based vs. model-free learning . . . 8
  - 2.1.2 Monte Carlo Planning . . . 9
- 2.2 Machine Learning (ML) . . . 12
  - 2.2.1 Neural Network . . . 12
  - 2.2.2 Fully Connected Neural Network (FCN) . . . 13
  - 2.2.3 Convolutional Neural Networks (CNN) . . . 16
  - 2.2.4 Residual Networks (ResNet) . . . 21
- 2.3 Combining ML with RL . . . 22
- 2.4 AlphaZero . . . 23
- 2.5 MuZero . . . 28
- 2.6 Chess Position Evaluation . . . 30

3 Esports Trainer 33
- 3.1 Analyzing the player with the AI . . . 33
- 3.2 Approach . . . 36
- 3.3 Software Architecture . . . 37
- 3.4 Data Engineering . . . 39
  - 3.4.1 Data Storage . . . 41
  - 3.4.2 Data Preparation . . . 41
  - 3.4.3 Neural Network . . . 42

4 Evaluation **45**
4.1 Experiments 45
4.1.1 Breakout 45
4.1.2 Car Racing 54
4.2 Discussion 59
4.2.1 Perfect Game 61
4.2.2 Value only Game 62
4.2.3 Easy Game 62
4.2.4 Game Observation 63
4.2.5 Miscellaneous 63

5 Conclusion **65**

Bibliography **67**

List of Tables

3.1 Action space of the Atari games 41

4.1 Analyzing the different c_{init} 52
4.2 Car Racing reward comparison 57

List of Figures

2.1 Markov Decision Process 5
2.2 MCTS Simulation 10
2.3 Asymmetric MCTS 11
2.4 Regression Model as Neural Network 13
2.5 The Layer network 14
2.6 Rectified Linear Activation Function 15
2.7 Convolution 18
2.8 Padding 18
2.9 Stride 19
2.10 VGG-16 20
2.11 Normalization 21
2.12 Off-line Training 24
2.13 MCTS from AlphaZero 26
2.14 WDL head of Leela 31

3.1 MCTS Example 34
3.2 Easy Game MCTS 36
3.3 Project build 38

4.1 Image of Breakout 46
4.2 Comparison between time and played games 47
4.3 c_{init} in Breakout 48
4.4 AVG value Breakout 48
4.5 Game length Breakout 49
4.6 Policy Loss Breakout 49
4.7 Reward loss Breakout 50
4.8 Value loss Breakout 50
4.9 Breakout sample actions 51
4.10 Breakout Perfect Game 51
4.11 Breakout Value only Game 53
4.12 Breakout point of no return 53
4.13 Easy Game comparison with and without policy 54
4.14 Breakout action suggestion comparison 55
4.15 Image comparison Car Racing 56
4.16 Comparison between human and AI in Car Racing 58
4.17 Car Racing Value only Game comparison 60
4.18 Car Racing action suggestion 61

4.19 Perfect balanced MCTS 64

List of Algorithms

1 Stochastic gradient descent (SGD) with momentum [GBC16]. 16
2 Adam: Adaptive moment estimation [Dro22]. 16

1 Introduction

Esports is defined as a competitive form of computer gaming. Over the past few years, Esports has grown to the point where some events are watched by millions of people simultaneously and is therefore monetized [unk24, Jin21, Hig20]. The most popular forms are some kind of multiplayer games where teams play against other teams or against AIs (artificial intelligence) [Hig20]. On a high level of popularity and competition, they play for high prize money. Over time, the field has become more professional. Nowadays, teams hire data analysts and coaches to improve their game [NKJS23, Hon23, Pip18]. With this demand, the goal is to study this field in a scientific way, using artificial intelligence to support the teams, coaches and data analysts. The audience are not only experts, but all players with different experience.

Most of the research so far has focused on improving the performance of AIs. Every few years there are new AIs for other games or stronger ones [AG21, SPC23]. There is a lack of research on how to use these AIs to improve human play. There are projects to use them for board games like chess [Lya, MC23]. The problem is that they are limited to one type of game, especially board games.

This thesis tries to close this gap by developing an AI powered Esports Trainer. By using reinforcement learning and advanced neural networks, different valuable information are extracted from a state of the art AI. With this in mind, the following questions arise:

- How can a machine learning based Esports Trainer give feedback to human players?
- How can an Esports Trainer evaluate scores during a match?
- How valuable is feedback from the Esports Trainer?

This thesis attempts to answer these questions and will present some approaches. One of the strongest competitive AI at the moment is MuZero [SAH+20], an evolution of AlphaZero [SHS+17]. AlphaZero is famous for playing at a superhuman level in the game of Go, where humans were previously unbeatable. This was a milestone in the history of AI. Another breakthrough was to beat other chess computers. MuZero has also outperformed other current engines in various games [SAH+20]. This was all done by only playing against itself. The idea for the Esports Trainer is to use this algorithm as a basis to help human players improve. MuZero is a reinforcement learning (RL) algorithm that learns from experience. To understand MuZero, it is necessary to understand the basic concepts of RL. RL alone cannot achieve these results, only in combination with neural networks complex information can be processed.

With this theoretical foundation, ways of extracting more information from MuZero to help players improve will be explored. The methodology will involve the following steps:

1. **AI Training:** A powerful AI agent based on MuZero will be trained in different games.
2. **Data Collection:** Data from a human and the AI are collected, including game state, actions and results.
3. **Feature Engineering:** Relevant information will be extracted, such as decision patterns and state predictions.
4. **Evaluation:** Methods for the AI are developed to evaluate the human gameplay based on their game knowledge.

The main hypothesis is that from a well-trained AI, information can be extracted. This information can be used to analyze and evaluate human actions in the game. In addition, it should be possible to give concrete suggestions for improvement without always using the perfect action. There are several ways to achieve this. Some methods have been explored and tested experimentally. These experiments were carried out with different games to provide variation. Finally, the results will be analyzed and discussed.

2 Related Work

The main idea to create the Esports Trainer is to use the best or one of the best AIs. These AIs must be able to play any game and play it well. The field of reinforcement learning focuses on games and how AI can master them. It is combined with machine learning for improvement and pattern recognition. All this is needed to examine the construction of AlphaZero, one of the best chess, go and shogi AIs, and its evolution MuZero, which is capable of playing not only complex board games, it is capable of playing modern games at super human level.

2.1 Reinforcement Learning (RL)

Reinforcement learning (RL) is a framework in which an agent observes the environment, chooses an action according to previously learned situations. The environment rewards the actions if something has been achieved in the game. The goal is to maximize the reward in the environment [SB18].

One of the fundamental challenges of reinforcement learning is the balance between exploration and exploitation. This makes it very different from other learning algorithms - the reward is maximized by the learning agent using good past actions in similar situations. To gather more information about the environment, the agent must explore new territory to find new good actions. It is important that the agent finds new better actions while discovering new strategies. This can lead to actions that seem bad at first, but later turn out to be better. This dilemma, by either prioritizing exploration over exploitation, results in poor performance. Only by balancing these parameters the AI will perform well. In the context of stochastic tasks, it is necessary to repeat each action to predict the correct outcome, but in this work only non-stochastic problems are studied. Another important fact about reinforcement learning is the strong focus on goal-directed agent behavior in an unpredictable environment. The basis of reinforcement learning is an agent that tries to interact with the environment in a fully functional and goal-oriented way. This means that the agent knows all possible actions and tries to get the maximum reward. The goal is described as clear measurable targets, usually reward points. The agent also perceives the entire environment as an observation and has the ability to select actions that affect the environment. Reinforcement learning is the closest type of learning algorithm to the way humans and animals learn. Many of the core reinforcement learning algorithms were originally inspired by biological learning systems, similar to behaviorism in psychology. The most prominent example is Pavlov's dog, where a dog is conditioned to salivate at the sound of a bell.

In addition to the agent and the environment, there are four basic key components in an RL system: A policy, a reward signal, a value function, and optionally an environment model.

A *policy* describes the way how to act in a given situation. It can be seen as the probability of taking the actions in the current state. It is like a blueprint for the next actions of the agent.

A *reward* signal guides a reinforcement learning agent towards its goal. It is a numeric feedback after each action. It can also be zero. This encourages actions that result in more reward.

For long-term planning, instead of just evaluating the last action, a *value* function defines the long-term value or reward. It is the total reward collected in the future from a given state. So the difference between reward and value is that reward is imitative and value functions consider future potential. The main objective is to maximize the rewards over the duration, which implies further state with a maximum value function. Without the reward, the value function could only depend on win or loss, which makes the reward crucial for the learning process and its performance. The problem is that value estimates are difficult to achieve, as it requires analyzing sequences of observations over the entire lifetime of an agent.

The last element is an environment model. This is the only property of reinforcement learning which is not necessary. It is a model of the environment. It allows the agent to simulate the next actions to look into the future and predict the next states and rewards. That can be used to improve the agent's performance. Reinforcement learning algorithms that use an *environment model* are called model-based RL algorithms. These algorithms are often more efficient than model-free RL algorithms, which do not use an *environment model*, because they can make more informed decisions about what actions to taken [SB18].
Reinforcement Learning (RL) uses the Markov Decision Process (MDP) framework to formalize the interactions between a learning agent and its environment. This framework divides the decision process into three distinct components: states, actions, and rewards.

A mathematical view of RL is to solve Markov decision processes (MDPs). MDPs are a well-established framework for modeling sequential decision problems, where the consequences of an action extend beyond the immediate reward and influence future states, which in turn affect subsequent rewards or their value function [SB18]. It formalizes the interaction between the learning agent and the environment. MDPs include immediate and delayed rewards. It requires a balance between short-term and long-term gains of rewards. It simplifies the problem of learning from interactions to achieve a goal. The agent learns and decides while the environment surrounds it. Agent and environment interact continuously, with the agent choosing actions and the environment responding with new situations and rewards, including actions with no response. This process can be seen in Figure 2.1. The following is a (slightly) more formal view of the MDP. It is by no means complete and only highlights the important facts, collected, filtered and summarized from [SB18]. Discrete time steps $t \in \mathbb{N}$ are used for each interaction between the environment and the agent. At each step the agent gets a state $S_t \in \mathcal{S}$ from the

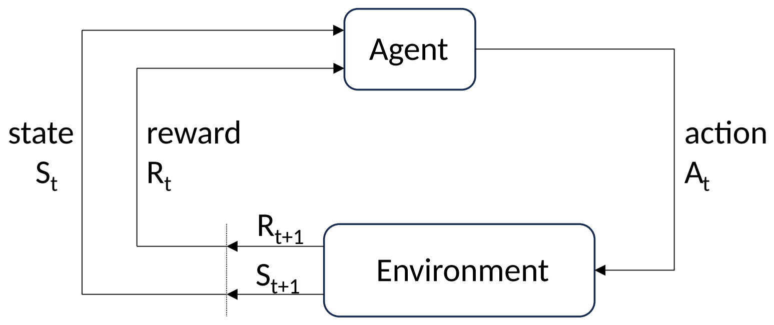


Figure 2.1: The agent–environment interaction in a Markov decision process [SB18].

environment. The agent reacts to this state with an action $A_t \in \mathcal{A}(s)$, this action is rewarded with $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ at the next time step. Thus, the MDP and the agent together generate the sequence or trajectory like this:

$$S_0, A_0, R_1, S_1, A_1, R_2, ... \tag{2.1}$$

The sets of a finite MDP ($\mathcal{S}, \mathcal{A}$ and $\mathcal{R}$) have a finite number of elements. Thus, the variables R_t and S_t have a well-defined discrete probability distribution. State and action are the only variables that affect the reward and the next state. Let $s' \in \mathcal{S}$ and $r' \in \mathcal{R}$, then there is a probability of those values where they are at time t:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.2}$$

for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(\int)$. It defines the dynamics of the MDP. The dynamics function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \times \to [0, 1]$ is a deterministic function with four arguments. For all possible choices the result is

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s). \tag{2.3}$$

If the state contains all past information which are relevant to future decisions, then the function p has the *Markov property* [SB18]. For further discussion, the Markov property is assumed, but it may be violated in some corner cases in the practical problem-solving process. A more mathematical view can be found in [DS05].

At each time step, the environment returns a feedback (reward) to the agent. The agent's goal is defined by this feedback. It tries to maximize the reward over time, rather than focusing on the immediate reward. It is important to note that the reward defines the goal, not the method of collecting it. The goal is the cumulative reward from a sequence of actions. This reward, denoted G_t, is calculated as a function of the reward sequence. In the simplest case, the cumulative reward is the sum of all rewards received,

$$G_t = R_{t+1} + R_{2+t} + R_{3+t} + ... + R_T, \tag{2.4}$$

where T is the time of termination. This equation fails to prioritize short-term rewards over long-term gains. To address this problem, there is a concept called discounting.

Discounting reduces the value of later rewards and diminishes their impact on G_t The main purpose is to balance the immediate and future rewards. It can be expressed as

$$G_t = R_{t+1} + \gamma R_{2+t} + \gamma^2 R_{3+t} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma G_{t+1}, \tag{2.5}$$

where γ is the discount rate, with $0 \leq \gamma \leq 1$ [Mah96, Pit19, SB18] .

The value function uses these reward values and represents them as a numerical number of the outcome of state-action pairs. It is the cumulative reward that the agent collects in the future [Bar03]. These value functions are defined in the context of the way the actions are chosen, also called the policy. The policy maps states to probabilities of selecting each possible action. The policy is denoted as π at time t, so $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Reinforcement learning algorithms determine how to update the agent's behavior or policy based on its interactions with the environment [NNXS17, Sze10]. Let $E_\pi(x)$ be the expected value of a random variable, if the agent follows the policy π then it holds for every policy π and every state s, that the state-value function $v_\pi(s)$ is:

$$\begin{aligned} v_\pi(s) &\doteq E_\pi(G_t|S_t = s) \\ &= E_\pi(R_{t+1} + \gamma G_{t+1}|S_t = s) \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)(r + \gamma E_\pi(G_{t+1}|S_{t+1} = s')) \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')), \text{ for all } s \in \mathcal{S} \end{aligned} \tag{2.6}$$

with $s' \in \mathcal{S}$, $a \in \mathcal{A}(s)$, and $r \in \mathcal{R}$. The last part of the Equation (2.6) is called the Bellman equation [SB18, Bel57]. There are simpler and shorter formulations, for example

$$V(s) = \max_a (R(s, a) + \gamma V(s')) \tag{2.7}$$

where s is the current state, a is the next action, R is the reward function. γ is the discount factor that decreases the reward over time. S' is the future state from taking an action [LBE10]. The discount factor is typically between 0.9 and 0.99. A lower value encourages short-term thinking and a higher value emphasizes long-term rewards. At each step, the exact value of the future state must be known in order to calculate the current state. This is a recursive function as long as there is no end state. The end state is defined by not having a next state, a terminal state. The Bellman equation averages over all possibilities and assigns weights to each based on probability. It states that the value of the starting state is equal to the expected discounted value of the next state plus the immediate reward. In the second Bellman Equation (2.7) it is simply the maximum of the reward and the assumed value. The goal of solving a reinforcement task also means finding the best policy to achieve the most reward over the run by selecting the best actions for the highest value. The optimal state-value function, denoted v_*, is defined as

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.8}$$

for all s $\in \mathcal{S}$. The same optimal action-value function $q_\pi(s, a)$ is shared by the optimal policies, denoted by q_* and defined as

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{2.9}$$

for all s $\in \mathcal{S}$ and a $\in \mathcal{A}$ [SB18]. It can be rewritten to

$$q_*(s, a) = E[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]. \tag{2.10}$$

The Bellman Equation (2.7) is a system of n equations, reflecting the n states, with n unknowns for each equation due to its recursive property, and thus also for each value and policy function. For environments with many possible states, it would take a long time to solve the Bellman equation. It does not matter if it is about finding the value function v_* or q_* due to its redundant behavior. Many methods in RL are ways of approximating a solution to the Bellman equation. For example, heuristic search techniques expand the Bellman equation multiple times to form a tree of possibilities, using a heuristic function to estimate values at the leaf nodes.

One such heuristic can be for instance Monte Carlo methods, which are used to estimate value functions and find optimal policies. It is a sampling technique that does not require complete knowledge of the environment. It generates data by simulating interactions with the environment [KW08]. This allows the agent to make good decisions without understanding the underlying dynamics of the environment. Learning by only interacting with the environment without knowledge of it, removes a lot of complexity, but more importance in many cases the underlying mechanisms of the environment are unknown. With this capability, the agent learns only from experience, without any additional rules or models. With this tool it is possible to solve the Bellman equation by sampling the returns of the environment. For each state-action pair, a reward-state pair is returned. With this information, the value can be estimated by the cumulative discounted reward of the samples. The agent explores the environment and generates more samples to get more information, resulting in a convergence to the true expected value. This is the basis of all Monte Carlo methods [Sil15].

Random sampling in a game would be a bad idea due to the large number of possibilities and the complex structure of a game. This brings up the tension between exploration and exploitation. On the one hand, exploration is necessary because new samples must be taken to increase the accuracy of the action value estimates. At first glance, the best actions would be the greedy and well-explored ones, but other actions may perform better and lead to unknown states. On the other hand, unknown territory needs to be explored to find the optimal solutions. The balance between the two is crucial for good performance. This can be done by considering the closeness of estimates and the uncertainties in those estimates. A good way to do this is

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{ln(t)}{N_T(a)}} \right], \tag{2.11}$$

which is called the upper confidence bound (UCB) [Aue00, KS06a]. This equation balances exploration and exploitation by taking into account the natural logarithm of time ($ln(t)$)

and the number of times the action a has been selected ($N_T(a)$). The constant $c > 0$ controls the overall balance between these two aspects. If $N_T(a)$ equals zero, it means that action a was never selected, it is considered a maximizing action, and the policy assigns it a probability of 1. $Q_t(a)$ as the values (average reward) for given actions [SB18].

2.1.1 Model-based vs. model-free learning

Given a state and an action, the *model* returns or produces the next state and the next reward. This is possible for all possible states. To solve the Bellman Equation (2.6), an environmental model can be used to find an optimal policy. With this model, the agent can look at the expected reward for each action and this for each next state, starting from the current one. With this knowledge, the best action can be chosen. For many applications, such a model, which eliminates the possibility of simulating further states and rewards, does not exist. The agent does not have access to the same information as it would have with the model. Without a model, a state can only be obtained by the sequence of actions from the beginning. An example of providing a model is in the game of chess, where it is possible to start from any state without knowing the previous one. It is like a simulator where all the rules are known. Most games are not accessible in this way, they can be played from the beginning, but not started from any state. This fundamental difference requires a different approach. Model-free and model-based reinforcement learning are two different approaches to learning in RL.

Model-free RL algorithms learn from observed payoffs and resulting states by interacting with the environment. They learn the policy and value function from experience by using the environment. They are much easier to implement than model-based ones, but the downside is that they learn much slower because they have to learn the environment's behavior first. They start exploring by using a trial-and-error approach. The most well-known model-free algorithms are Q-learning [WD92] including Deep Q-Network DQN [MKS$^+$] and Proximal Policy Optimization PPO [SWD$^+$17].

Model-based RL algorithms, on the other hand, learn by using a model of the environment. To distinguish it from model-free RL, «Predictive models can be used to ask "what if?" questions to guide future decisions.» [KLM96]. This model is used to simulate the environment and predict the rewards and new states. The mode can ask the "what if" question, with this information the next steps can be planned. There is no need for trial and error because it knows what will happen in the next states. The result is known from the simulation. Prominent examples are AlphaZero [SHS$^+$17], which uses a Monte Carlo Tree Search (MCTS) [Cou06] as a model-based algorithm.

There are differences as well as similarities between these methods. The main functionality of both methods is the calculation of value functions. They are used to look ahead, calculate a value, and use it to approximate the value function. What does it mean to look ahead or plan the next moves? It is a search through the state space for an optimal policy or path to a goal. It is also called state space planning. The idea is to compute value functions to improve the policy. These value functions are updated by simulated experience. This follows: Model $\rightarrow$ Simulated Experience $\rightarrow$ Update $\rightarrow$ Values $\rightarrow$ Policy
In model-free models, simulated experience is modified by real experience provided by the

environment. Planning is used to improve a policy or value function with the experience gained from the model. A simple approach to planning is to select a single action A_t at a given state S_t, then check the value of the next state S_{t+1} and compare it to other states discovered by other actions. Searching deeper improves the selection of actions. For a perfect model and an imperfect action-value function, a deeper search will usually produce better policies. This deep search costs a lot of computation time. Some games have a large state space where it is impossible to compute every possible state guided by actions. A classical method to plan ahead is heuristic search. The states and value functions are represented in a tree. A leaf node holds the information (state-action) and is then traced back to the root (current state). The best path (best value) is chosen to get to the new state.

2.1.2 Monte Carlo Planning

Monte Carlo methods are previewed before, now they can be used to plan the next steps. To look into the future, the next states must be rolled out. This type of algorithm is called a rollout. The goal is to start with a policy and get a better policy [Ber]. Rollout algorithms are planning algorithms based on Monte Carlo methods. Given a policy, they estimate action values by averaging the payoffs of simulated trajectories. A trajectory is the path through the state space taken by the agent to the end. It is a sequence of state actions and associated value functions. One of the most promising rollout algorithms is the Monte Carlo Tree Search (MCTS) [SB18].

The MCTS is applicable to the Markov Decision Process (MDP) ($\mathcal{S}, \mathcal{A}$ and R), so the same variables and sets can be used to understand the MCTS. $\mathcal{S}$ as the set of states, with $s_0 \in \mathcal{S}$ as the initial state. $\mathcal{A}$ denotes the set of possible actions. Sometimes the possible actions vary, such as in a game of chess, where the possible moves depend on the position of the pieces, resulting in the action set $\mathcal{A}(s)$. The immediate reward $r \in \mathcal{R}$ given by a new state s by taking an action a. The MCTS can be stopped at any state and return the current best action by using the Equation (2.8). The tree consists of connected state nodes (leaves). Each node is an action value function. Each edge is another action. The probability of choosing an action is determined by the policy, in the sense of Monte Carlo it is sampled and not fully rolled out. Each iteration of the search consists of four steps [SB18], see Figure 2.2.

1. Selection - Starting from the root node, at each level the next node is selected according to the (tree) policy. Only nodes that have been visited before are selected. So each of them is already in memory. This is done until a node does not have a next node.

2. Expansion - After selection, a new node is added to the last selected node. It is reached by performing an action step from the state of the last node. If the node is the final state, e.g. the last node. Then the next step is executed (the simulation is skipped).

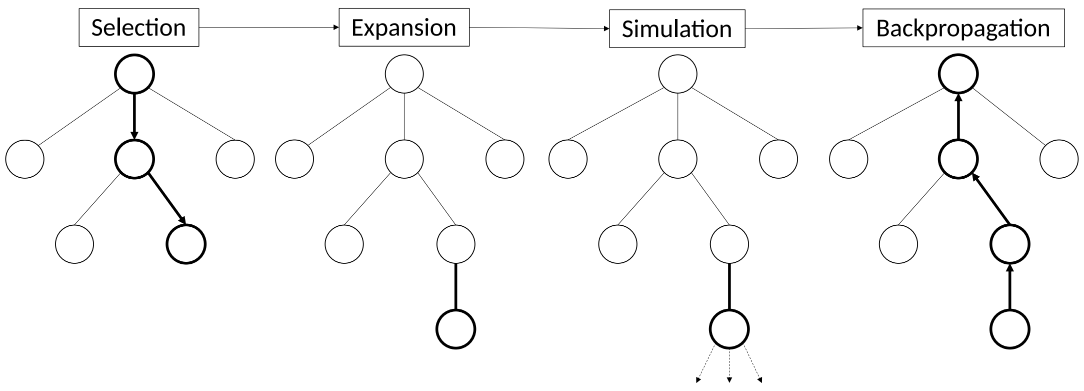


Figure 2.2: A complete simulation through a Monte Carlo tree. The first step is to **Select** the most promising leaf node, by some heuristic (UCB or UCT). This path is then **Expanded** by a new leaf node chosen by a rollout policy. The next edges are discovered by the **Simulation**. The last step is to update all visited nodes with the new information by **Backpropagation**.

3. Simulation - Run through the problem or game to the end using complete random steps. Collect all rewards and state-action pairs. This is the Monte Carlo step.
4. Backpropagation - All scores are delivered from bottom to top to the assigned and visited nodes in the tree. Each node now adjusts its value function based on the outcome of the game.

As discussed earlier, one of the main problems in reinforcement learning is the balance between exploration and exploitation. The use of the UCB Equation (2.11) helps to find this balance. Within a tree it is sometimes called UCT - Upper Confidence Bounds applied for Trees. The UCT is used to select the next node in the selection process. The children with the highest score are taken. The UCT leads to an asymmetric expansion of the tree, because more promising nodes are selected more often, as can be seen in the Figure 2.3.

Most (modern) games have a large action space, with many different possible actions in each state. This results in a large tree. For example, chess and go have a large number of possible actions to check each move. This is a hindrance to planning. Deep trees are more conducive to planing because they allow a look far ahead. There are three approaches to avoid the huge branching factor:

- Reduction of possible actions.
- The use of UCT Alternatives.
- An early termination of the game.

The goal of action reduction methods is to reduce this effect by removing some of the actions. The number of possible actions may be too large. In this case, the tree

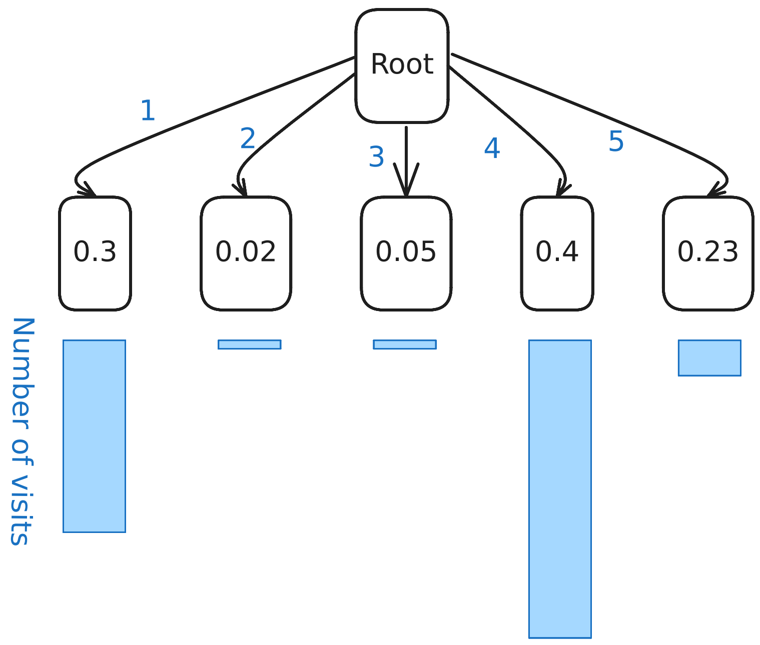


Figure 2.3: A section of an MCTS showing only the first layer. The edges are labeled with different actions. The nodes themselves contain the policy as a probability. The visits represent the number of simulations along the path. Thus, through action *4* the most child nodes where visited, and action *2* and *3* leads to the fewest, resulting in an unbalanced asymmetric tree.

grows sideways. Games like Go have actions that are not strictly dependent on the state in which they are played. The all-moves-as-first (AMAF) heuristic was developed for this purpose [Brü93]. It evaluates each move a by the total reward divided by the number of simulations in which a has ever been played. A case study [BPW$^+$12] shows that incorporating domain knowledge into a classical position in a chess or go game can dramatically increase MCTS performance. Strong playouts reduce the risk of convergence to a wrong move. In its classical form, MCTS evaluates game states by simulating random playouts, i.e. by simulating complete playthroughs to the end states. In early termination, a node is sampled only to an arbitrary depth. This saves simulation time and in some cases it is the only way to use the MCTS. In chess there are too many possible nodes to compute the whole tree from the beginning of the game. At some point in an endgame it is possible to build the whole tree. The depth has to be limited somehow, and it is a big problem to find the optimal point. The more nodes are explored, the more certain a good move is. There are several modifications of the UCB formula to solve this problem. Instead of using different UCBs, neural networks can approximate the state and value functions of the nodes

Before neural networks where widely used and accessible there where big search trees to find the values. Every possible action has to be mapped inside. Deep Blue defeated former world champion Garry Kasparov with such a large search tree in 1997. These brute force methods quickly get out of hand. More complex tasks require bigger search trees and at some point it is not possible to map all possibilities or even enough to prove a good result. This is where (deep) neural networks come in. Instead of brute force, the networks learn the pattern to evaluate the value of the state.

Multiplayer and singleplayer games can be distinguished in reinforcement learning, especially in the MCTS. A single-player game can be easily mapped into the MCTS and used to plan the next steps. Each node represents the value function driven by the player's actions. The environment provides the action state tuple derived from the game rules. In two-player games, such as chess or go, the next move always depends on the opponent's move. Both players use the same environment state and both have perfect information about the game and the next possible moves. This is only true for games like chess and go, where the algorithms are usually model-based. Here, each opponent's move is simply the next node row. Every other node row is the same player's move. Games with imperfect information, such as card games where the cards in the opponent's hand are hidden, have problems with MCTS. They increase the MCTS branching factor enormously, since each hidden piece of information results in a distribution of probabilistic game states. Each piece of information can be thought of as a node in the tree.

The main problem is that the computing power required to compute the state value functions is so enormous that approximation methods are needed. As hinted before, to improve the performance of RL beyond the human level, the use of machine learning is mandatory. The core ideas of machine learning are redefined and the techniques used to make it applicable to RL are described where appropriate.

2.2 Machine Learning (ML)

The used methods and algorithms use supervised learning. The learning network is provided by a set of training data consisting of multiple input-output pairs. This network is then trained to predict the output from a given input. The training is done iteratively, where at each iteration the training examples are taken and compared with the output of the network. Learning is done by adjusting and minimizing the error between the prediction and the output of the original training data. A well-trained network can then make predictions for new inputs that were not seen in the training data. One of the most promising methods for predicting such a complex outcome, as required for games, is the neural network.

2.2.1 Neural Network

As a starting point the basic knowledge of single layer neural networks, where it can solve only linear problems is assumed. These are regression models where the outputs are linear functions. A layer contains a set of weights and biases. One weight and one bias for each neuron in that layer. It can be expressed as $y(x, w) = w_0 + \sum_{j=1}^{M-1} w_j x_j$.

Where x is the input, M is the number of parameters in this model, w_0 is called the bias and will be referred to as b. w are the weights of each neuron. This can be rewritten to $y(x, w) = b + w^T x$, which allows us to take the inner product of w and x. The graphical representation can be seen in the Figure 2.4 [BB23].

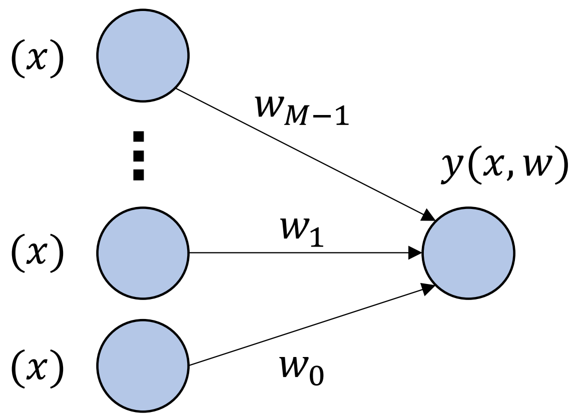


Figure 2.4: Linear regression model, shown as a neural network diagram. (x) are the inputs, w are the weights, and y is the output [BB23].

2.2.2 Fully Connected Neural Network (FCN)

An FCN is successfully used for many tasks such as image classification. It contains not only input and output neurons, but also layers of neurons in between. The name fully connected suggests that each neuron in a layer is connected to each neuron in the next layer. Let F^l represent layers $l = 0, ..., L$, where each function F^l has a linear part W (matrix multiplication) and a nonlinear part f. Then the fully connected network is a concatenation of functions

$$F^L(\ldots F^1(F^0(x))) \tag{2.12}$$

[Dro22]. The layers of a neural network can be thought of as a Markov chain. Let $x = a^0$ be the layer $l = 0$, every other layer has a pre-activation vector $z^l = W^{l^T a^{l-1}}$ and an activation vector $a^l = f(z^l)$. Only the previous layer is relevant for the next one. The output is $y = a^L$, which in the case of Figure 2.5 is $L = 3$. Matrices store the weights W^l of the layers. The biases are values in the neurons to influence the output of individual neurons [Dro22].

FCNs are trained by supervised learning. The network receives a set of training data, which are input-output pairs, where each input is coherent to a desired output.

In general, an FCN is a neural network in which each layer of neurons influences the outcome of the next layer. Each neuron performs a weighted sum of its input and then applies an activation function. This output is passed to the next layer of neurons. For classification problems, for example, these outputs must be in the range (0, 1), which can be thought of as probabilities. These FCNS are universal approximators, which are able

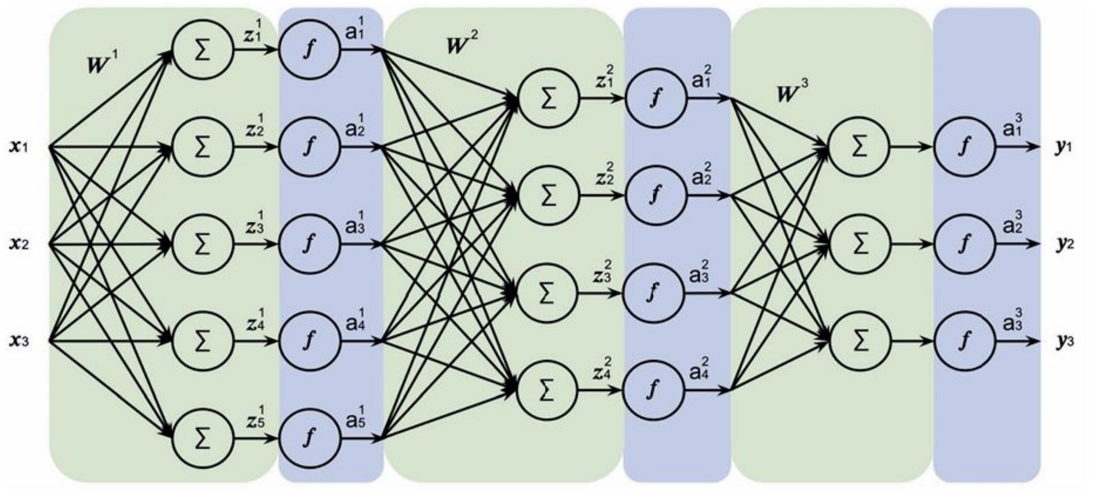


Figure 2.5: A three-layer neural network. The green area marks the linear part, the blue the nonlinear part, the activation function. The input x is called the input layer. y denotes the output. Image taken from [Dro22].

to approximate any continuous function arbitrarily well. Of course, this depends on the size of the network, training, sample data, and other limitations.

Let f be the activation function with $f \to \mathbb{R} \to \mathbb{R}$. These activation functions f convert the linear function into a nonlinear one, because f is typically nonlinear. The most common ones, and the ones relevant to this paper, are **rectified linear unit (ReLU)** and softmax [HSS15, NH10, Dro22, BB23]. The ReLU is defined as

$$g(z) = max(0, z). \tag{2.13}$$

The graph can be seen in Figure 2.6. The ReLU maps negative values to zero, at $z = 0$ there is the only point where it loses linearity. Another important activation function is Softmax. Softmax is used for multiclass classification and maps an input vector z of K real numbers to $[0; 1]$ which sums to 1. It can be seen as a probability distribution of z. So even negative values will have a corresponding value between 0 and 1. The function is given by

$$f(z)_i = \frac{e^{z_i}}{\sum_{c=1}^{k} e^{z_c}} \tag{2.14}$$

where $z \in \mathbb{R}^k$ is a vector, $i = 1, ..., k$, k is the number of classes, $f(z) \in [0, 1]^k$ which sums up to 1: $\sum_{c=1}^{k} f(z) = 1$ and c are the classes $c = 1, ..., k$ [Agg18].

In supervised learning, the input is classified. The simplest version is when the label is binary, for example, there are points on a line and the goal is to separate them into two groups (two labels). The learning system can now learn to predict the separation into classes. The mismatch (error) between the predicted and observed values is penalized with a **loss function** [Agg18]. The goal is to minimize the loss function, which indicates

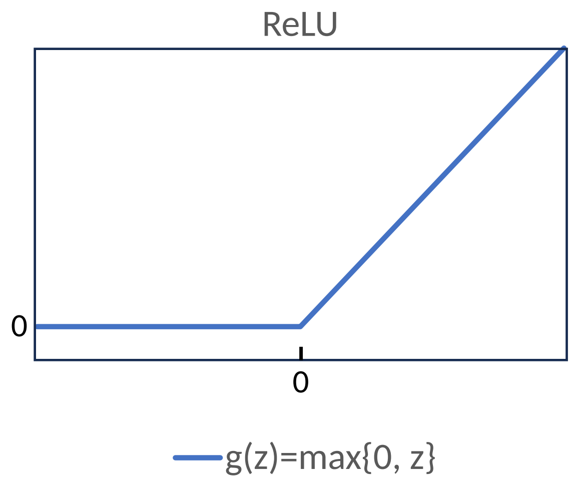


Figure 2.6: The rectified linear activation function (ReLU) is used to transform linear to nonlinear outputs by preserving many linear properties [GBC16].

a minimum error between the prediction and the true value. The global minimum for all inputs is the best result, but it is hard to find the global minimum in complex functions. Often, good enough functions are assumed to provide a good, but not the best, solution. One of the most practical methods to find solutions, also called **optimizer**, is the **gradient descent**. It iteratively finds a new local minimum by going in the direction of the steepest descent [Dro22]. It follows:

$$x' = x - \epsilon \nabla_x f(x), \tag{2.15}$$

where $\nabla_x f(x)$ is the gradient of f, ϵ is the learning rate, which is the step size in one direction, and x is a point [GBC16]. The problem with this is that on a large data set it takes a very long time to compute all the gradients for all the points. More important is the stochastic gradient descent (SGD) algorithm [RM51]. It is an extension of the gradient descent, it approximates the gradient from only a single data point or a mini pile. This results in a less accurate solution, but good enough.

One way to improve the gradient descent algorithm is to add momentum to it to make it converge even faster. The idea is to use a fraction of the gradient of the previous step to update the actual steps. This fraction can be larger or smaller, depending on the problem. This mostly affects flat regions where SDG would take a long time to converge [Dro22, BB23].

The complete algorithm can be seen in Algorithm 1. There are two mandatory (hyper) parameters. The learning rate from Equation (2.15) and the momentum α. The momentum here is similar to the behavior in physics. Momentum tracks a weighted history of past gradients, favoring more recent ones, and uses this history to guide future updates, making it more efficient. It is like a rolling ball, where momentum considers the direction it came from (past gradients) and keeps moving in that direction. Older gradients have less influence than new ones.

Finding the perfect learning rate seems to be a difficult task. Momentum tries to

Algorithm 1: Stochastic gradient descent (SGD) with momentum [GBC16].

```
given Learning rate $\epsilon$, momentum $\alpha$;
initial parameter $\theta$, initial velocity $v$ ;
while stopping criterion not met do
    Sample a minibatch of $m$ examples from the training set $\{x^{(1)}, ..., x^{(m)}\}$ with
      corresponding targets $y^{(i)}$;
    Compute gradient estimate: $g = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$;
    Compute velocity update: $v = \alpha v - \epsilon g$;
    Apply update: $\theta = \theta + v$ ;
end
```

mitigate this problem, but it is not optimal. Adaptive learning rates solve this problem by automatically adjusting the learning rate during the learning process. **Adam** [KB17] is a type of this algorithm. The name comes from adaptive moments. It uses a variable momentum and a variable learning rate. The algorithm can be seen in Algorithm 2.

Algorithm 2: Adam: Adaptive moment estimation [Dro22].

```
given $x_1 \in dom f$;
given learning rates $\{a_t\}_{t=1}^{T}$;
given decay rates $\beta_1, \beta_2 \in [0, 1)$ close to 1;
given small $\epsilon > 0$ close to 1;
initial $m_0 = 0, v_0 = 0$;
while not converged do
    $g_t = \nabla f(x_t)$ gradient;
    $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$ first momentum $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ second
      momentum $x_{t+1} = x_t - a_t \frac{m_t}{\sqrt{v_t} + \epsilon}$ update
end
```

With this foundation it is possible to approximate the bellman Equation (2.6). Fully connected networks work well on unstructured and small inputs, but games consist of many images that are structured. There are better networks for analyzing images.

2.2.3 Convolutional Neural Networks (CNN)

One advantage of using images is that they are human readable and interpretable. Games are usually displayed as images (frames), which makes it easy to access the data. A digital image is a rectangular array of pixels, where each pixel is usually a triplet of red, green, and blue channels. Each color has its own level of intensity. Depending on the color space (8bit, 32bit...) it is possible to display more or less combinations. For simplicity and to reduce the amount of data, sometimes grayscale images are used. These images contain only one channel. Neural networks are used to learn patterns in the data. With an FCN, the size of the weights and biases would increase enormously, for each pixel there would be one parameter for each layer. A color image of 1000×1000 pixels combined with a layer of 1000 hidden units would already have 3×10^9 weights for the first layer alone. There is also a structured problem that affects the significance of the structured data in images. The best way to find and learn patterns in images, but not only in images are convolutional neural networks (CNN) and was developed to recognize handwritten digits [Fuk80]. The basis of convolution is to add each element of the image to its local neighbors, weighted by the kernel. The two-dimensional discrete convolution of a filter f with a signal g is defined as

$$(f \star g)(i,j) = \sum_{u=-s}^{s} \sum_{v=-s}^{s} g(u,v) f(i-u, j-v), \tag{2.16}$$

where g can be an image with pixel values at the image position i and$j \in \mathbb{N}$ and the filter f as a weight matrix. $s \in \mathbb{N}$ determines the size of the image. [Dro22] The name convolutional neural network is a bit misleading because most ML libraries use cross-correlation during a normal forward pass, which is very similar to convolution and differs only in the sign of the result. [Med] The two-dimensional discrete cross-correlation of a filter f with a signal g is defined as

$$(f \odot g)(i,j) = \sum_{u=-s}^{s} \sum_{v=-s}^{s} g(u,v) f(i+u, j+v). \tag{2.17}$$

Convolution is a linear operator and can be represented as a matrix multiplication. With this approach, the primary goal in image recognition is to learn feature detection functions that can accurately identify the presence or absence of local visual features in an image [Kel19]. In the learning sense, there is no difference because the weights are only applied to the formula used. To avoid confusion, the term convolution will be used throughout. In the Figure 2.7 it can be observed that the resulting matrix loses size after the convolution. This depends on the size of the kernel. If a kernel is a $M \times M$ (typically squared) and the image is $J \times K$, then the dimension of the feature map is $(J - M + 1) \times (K - M + 1)$ [Dro22]. This property can be used to reduce the size of the output, but sometimes it need to be the same size as the input. A simple solution is to add zeros around the input image, which is called *padding*, see Figure 2.8. To get the same size, the number of symmetric padding can be obtained as follows. For Padding $P \in \mathbb{N}$ then the size of the output is

$$(J + 2P - M + 1)(K + 2P - M + 1) => P = (M - 1)/2 \tag{2.18}$$

. Padding can be seen as an way to make the image larger. It is also possible to do it the other way around and down-sample with the convolution. Instead of stepping over each pixel with the filter, the step size S can be increased. This is called the stride. In most cases, it is the same size horizontally and vertically. The size of the feature map is then calculated by

$$\left(\frac{J + 2P - M}{S} - 1\right)\left(\frac{K + 2P - M}{S} - 1)\right). \tag{2.19}$$

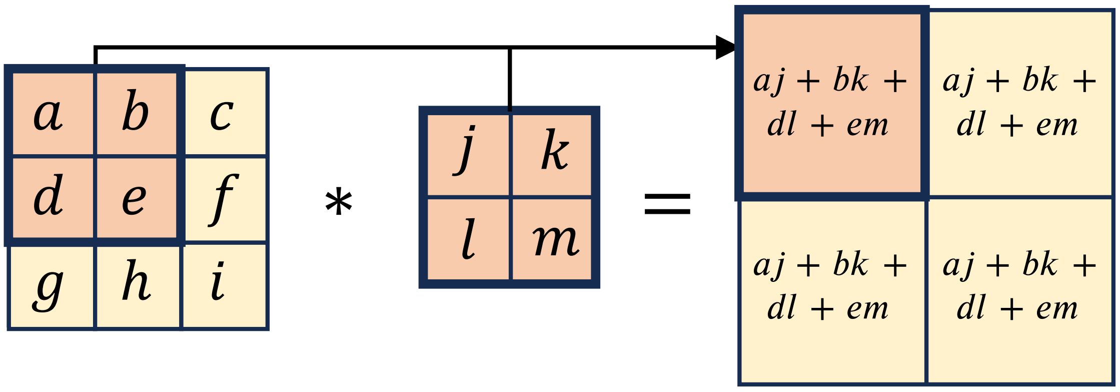


Figure 2.7: A visualized 2D convolution. The first step is shaded gray. The first square is the image and the second is the filter. The output size is reduced by one on each side [BB23].

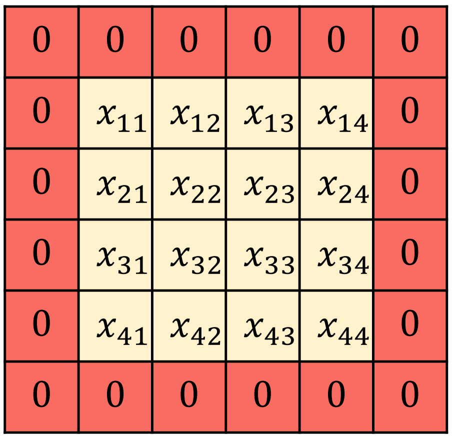

0 0 0 0 0 0
0 x_{11} x_{12} x_{13} x_{14} 0
0 x_{21} x_{22} x_{23} x_{24} 0
0 x_{31} x_{32} x_{33} x_{34} 0
0 x_{41} x_{42} x_{43} x_{44} 0
0 0 0 0 0 0

Figure 2.8: Example of a matrix with padding, where the zeros are placed around. The dimension changes from 4×4 to 6×6 [BB23].

Sometimes it is desirable to have smaller feature maps to reduce complexity by reducing the parameters used. Another way to reduce complexity is to add a pooling layer after convolution, which is often used in CNNs. Instead of making the output invariant to translations, sometimes small changes in the location of the input will not affect the reliability of the network. This can be done by using pooling after convolution. The pooling operation divides the feature map into a grid $P_q \times P_q$ of non-overlapping regions, see Figure 2.9. By using max pooling, the maximum values of the small region are taken. This reduces the output size while preserving depth. By taking the maximum values, the most important features of each region are selected, regardless of their position. This reduces overfitting by reducing the amount of information the model has to learn. In addition, performance is improved by reducing the size. The new size is then calculated

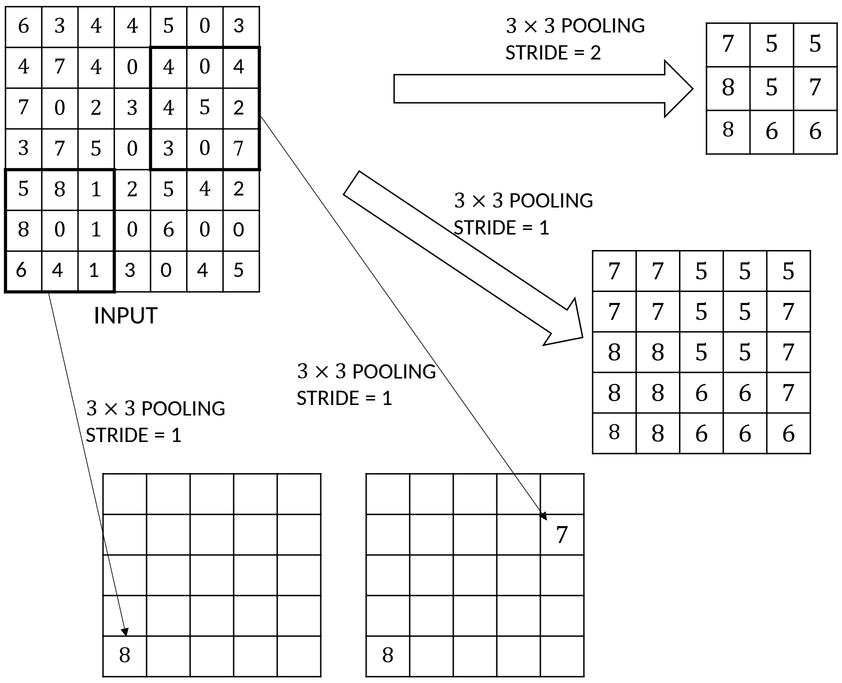


Figure 2.9: An example of the max pooling and stride operation can be seen. A 3×3 pooling is used to take the highest value of a range. If the stride is set to 2, the new matrix is reduced even more in size [Agg18].

by

$$(J - P_q + 1) \times (K - P_q + 1). \tag{2.20}$$

Average pooling is another pooling method to reduce the spatial size of the feature map without changing the number of feature maps. Instead of taking the maximum values, it takes the average values of the grid. Instead of taking the important values, it smooths and blues the values in the grid. The pooling functions also use striding and padding to reduce the duplication of values and further reduce the size. The pooling operation also improves the translation invariance of the model, so that small shifts in the input can be ignored. The primary goal of CNNs is to create a network that extracts local visual features in the early layers of the network and combines these features in later layers to form higher-order features. They share the weights for different parts of an image at each layer. The function for detecting features that all neurons in the convolution implement is defined by the kernel. When convolving over an image, the kernel acts as a local visual feature detector, recording all locations in the image where the visual feature was present. The result is a map of all locations in the image where the corresponding visual feature occurred. This process is commonly referred to as a *feature map.* A convolution represents up to one feature of an image. CNNs can have a lot feature maps to represent different features. Each map has a different kernel weight. In the learning process, the optimizer tries to find the optimal weights for all the different kernels. The kernel weights are

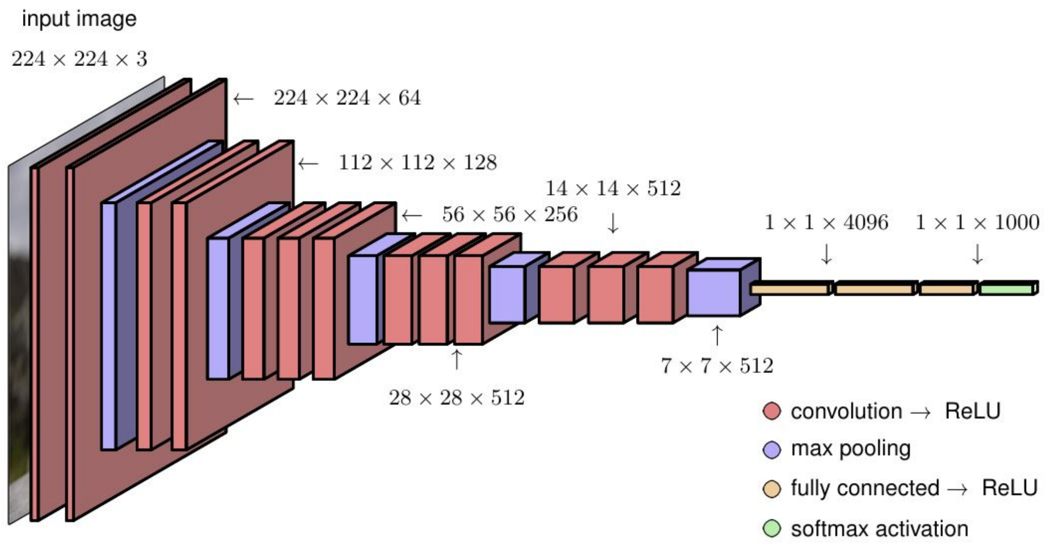


Figure 2.10: A typical deep convolutional network. It is called VGG-16 [SZ15, BB23].

usually 3×3 or 5×5 matrices. The learning is similar to the FCN, by finding the filter weights of the convolutional kernel to minimize the loss function, the CNN learns. To produce a prediction, the last convolutional layer is connected to a linear layer as in the FCN. The linear layer contains a weight matrix and the bias with y=Wx+b. Finally, this layer can be fed into the softmax activation function to obtain values between 0 and 1. a A typical CNN, VG-16 can be seen in Figure 2.10.The input is a 224×224 rgb image. The first layer contains 64 different kernels, but keeps the same dimensions. With the pooling layer, only the important features are used for the next step, which reduces the size. Each step increases the number of filters. After the 13th convolutional layer, followed by the ReLU, normal linear neural layers are used. The last station is the softmax function for a classification value output. VGG-16 [SZ15] was a top performer in the 2014 ILSVRC. This challenge evaluates algorithms for object detection and image classification at large scale [RDS+15]. It uses between 11 and 16 layers. It uses relatively small filters with increased depth (more layers). Small filters can usually only detect small regions or features, unless the network is deep. A deeper network extracts more non-linear features because of more ReLU layers. It also requires fewer parameters than larger kernels. VGG kernels are 3×3 and a pooling layer is 2×2. In the convolutional layer, a stride of 1 and a padding of 1 were used, resulting in the same dimensions as the input. Pooling was done with a stride of 2, which reduced the size by a factor of 2 after the pooling layer. The first convolution layer used 64 different filters. This number is doubled after each pooling. This idea was later adopted by ResNet. The increased depth led to instability and was avoided with pre-trained layers. After the last pooling layer, the first fully connected layer with 4096 neurons was connected to the $7 \times 7 \times 512$ pooling layer. Interestingly, this connection requires $7 \times 7 \times 512 \times 4096 = 102\ 760\ 448$ parameters, which is about 75 % of the total 138 000 000 parameters required for the whole network, which is a lot for just one layer connection.

Batch normalization is a relatively new concept [IS15] and therefore not yet implemented in the Vgg model. It further reduces vanishing and exploding gradients, i.e. the optimizer like SGD or Adam gets stuck in a local minimum or the opposite. In addition, in deep networks a covariate shift can occur during training, by changing the parameters in the first layers, the later layers change a lot, so they are unstable, resulting in slower convergence in the learning process. Santurkar suggests [STIM19] the reduction of the shift comes from the batch normalization, but the improvements of the results are from the smoothness of the error function after the batch normalization. Batch normalization is performed after each convolutional layer. This allows the model to converge much faster in training, and therefore allows for higher learning rates. Batch normalization normalizes each feature within a batch of samples by scaling the data by $\frac{1}{\sqrt{n}}$, where n is the batch size [Dro22]. For non-convolutional networks, it turns out that layer normalization is a better fit [BKH16]. Instead of normalizing each feature in a batch, all features of a sample are normalized. The difference can be seen in the Figure 2.11.

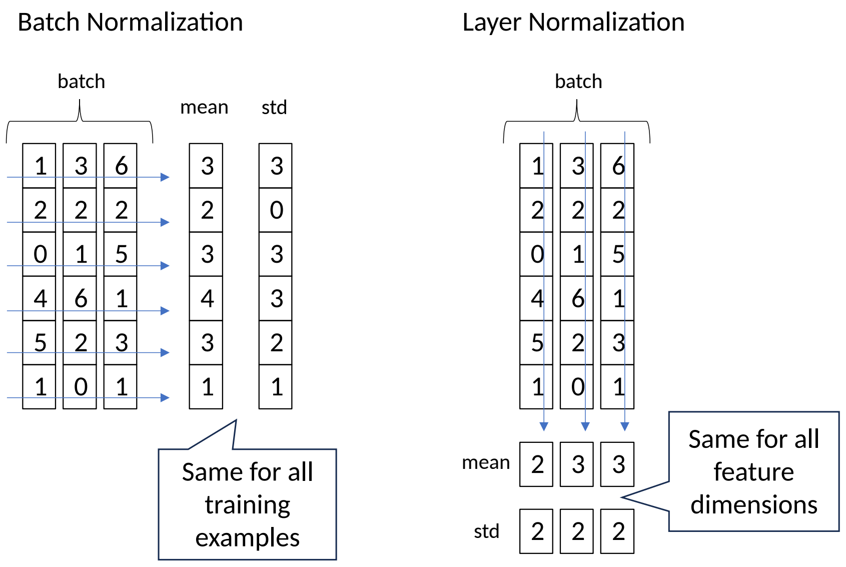


Figure 2.11: Comparison of layer normalization and batch normalization. Both attempt to achieve non-standard data by normalizing it to a desired format. They are two different approaches and both have their advantages. For computer vision, batch normalization is advantageous [Alg23].

2.2.4 Residual Networks (ResNet)

Another important neural network architecture that uses batch normalization and a truly deep convolutional network architecture without the drawbacks of being too deep is the Residual Network (ResNet) [HZRS15]. Each layer added to a neural network adds complexity. More weights must be found for each layer. The vanishing gradient is avoided by using skip connections between successive layers. The notation follows [Dro22]. Let be the output of a convolutional layer:

$$a^{l+1} = f(W^l, a^l) \tag{2.21}$$

where f is the activation function, W^l are the weights of layer l, and a^l is the input. Skip connections add the input directly to the function, resulting in

$$a^{l+1} = a^l + f(W^l, a^l) \tag{2.22}$$

.

It is also possible to skip more levels after adding the input. Learning the difference between a function and the identity function is easier than learning the function itself. Adding this difference, also known as the residual function, to the input produces the desired function [SB18].

A ResNet can have up to 150 layers, with increased performance for some tasks. Zagoruyko [ZK17] suggests that in most cases 50 layers are sufficient, because the complex structures captured by the later layers aren not used anyway.

2.3 Combining ML with RL

The combination of both achieve gives satisfying results in video games. With the function estimator collected by neural networks, it is possible to use it in reinforcement learning to avoid calculating each possible state or value function. It is possible to predict the value and policy functions by a neural network.

The first important step for an AI combined with neural networks to play at a super-human level was TD-Gammon, which used a combination of a simple neural network and self-play. As the name suggests, it was playing the game of backgammon. It is a two player game with two dice. It has about 10^{20} possible states, in comparison chess has about 10^{46}. The number of states is so large that a lookup table is not possible. The branching factor (number of children per node) is 21 (dice combinations) $\times$ 20 (legal moves per dice combination) $\approx$ 420. Chess has on average only about 30-40 possible moves. TD-Gammon required little knowledge of the game to play extremely well back in 1995. It uses a TD(λ) algorithm combined with a fully connected neural network. This network used 198 input neurons for each possible position, in the latest version it used 80 hidden units (only one hidden layer), the output was the value of the current position. As an estimator, this neural network tried to guess the value function directly from the current samples, and was good at it. To train the network, it played up to 1.5×10^6 games against itself and used the collected data to predict the value [Tes95].

18 years and a lot more hardware power later, the next big step forward was the Deep Q Network (DQN). Unlike TD-Gammon's online approach, DQN stores the agent's experience at given time steps and saves it over many episodes in a replay buffer. The network learns by selecting random samples from the buffer. The agent then chooses the

action by a ϵ-greedy policy. Another difference from TD-Gammon is that DQN works completely without knowledge of the game. To demonstrate the algorithm, it competed against humans and other RL algorithms on the Atari 2600 computer games, using only the pixel values as input [MKS$^+$]. The Atari 2600 is a home video game console from 1977, with famous games like Breakout, Space Invaders, Pong ... and so on. These games are used today to compare the level of performance in RL. These games contain complex policies to master, which DQN where able to master up to a certain level. In most games it was at human level or even beyond [MKS$^+$15].

The MCTS unfolds its true power by combining it with machine learning. Since the first appearance of MCTS in 2006 [KS06b]. It has been shown by AlphaGo [Sil15] the first AI to beat human professionals in the game of Go. Until then, AI had only been able to win against amateurs. It won 4-1 against Lee Sedol, one of the strongest players of all time [SHS$^+$17]. It was a milestone for AI in recent years. AlphaZero was an evolution of AlphaGo because it was able to play any board game without any knowledge. The foundation of the most RL algorithms where introduced in the last chapter. With this knowledge it is possible to analyze the MuZero and it predecessor AlphaZero. Finally chess position-evaluation methods are described, because in chess AI board evaluation is heavily used since AI overcomes human skill in chess, not only by AlphaZero, even before since 1997 [Pan97].

2.4 AlphaZero

AlphaZero, a model-based reinforcement learning program that evolved from AlphaGo, developed by DeepMind Inc, the same company that developed the DQN, has made a major step forward in the field of AI. As described in the paper [SHS$^+$17], it has surpassed its predecessor AlphaGo in terms of performance and versatility, demonstrating its ability to be superior in various games, including chess and go. For many games, AI has dominated these types of games for a long time. Go was a kind of last bastion that fell in 2017. AlphaZero's reputation by today, is one of the best AI to play games. The zero in the name, derives from zero knowledge of the game and relies solely on data generated by self-play. It took only a few hours to achieve superhuman playing skills. This was only possible due to large hardware behind the scenes, about 4500 TPU where used for these breakthroughs.

AlphaZero can be divided into two main RL phases during the execution of the algorithm:

(a) Offline training: This part of the algorithm evaluates positions using policy and value functions. This is done by a neural network. This network is trained with previously played on-line games or collected data from human play.

(b) On-line play: It takes the policy and value function from off-line training and uses it to make optimal moves in real time.

This is very similar to the TD-Gammon algorithm [Tes95], in fact they use the same principle. This seems the reason, why alphaZero performs so well, as Bertsekas say: «An important empirical fact is that AlphaZero's online player plays much better than its extensively trained offline player. This supports the conceptual idea that ... the performance of an off-line trained policy can be greatly improved by on-line play.» [Ber22].

AlphaZero's off-line training algorithm is done by playing games only against itself. This is what most distinguishes it from AlphaGo, which used human moves and games to train. This self-play generates a sequence of policies and position evaluators based on deep neural networks, see Figure 2.12. These policies assign probabilities to each move, the position evaluator is a value function and assigns the expected future reward (value) to positions or states. Each position is re-evaluated, which results in refining the strategies and becoming more effective in playing the game. The training algorithm is based on policy iteration. It works by updating the policy and value function until the best version of all players is found. Offline training is divided into two phases: policy evaluation and policy improvement.

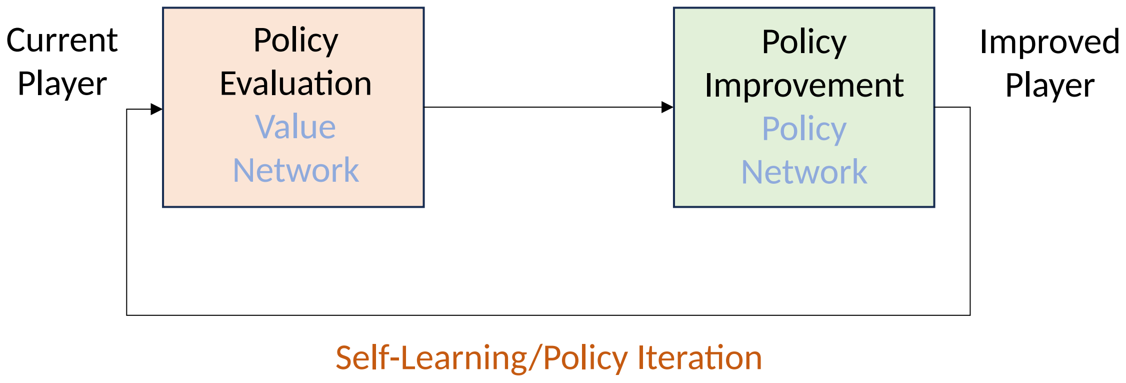


Figure 2.12: Picture of the offline training of AlphaZero. It consists of two networks, both taking as input the chess position and outputting the value or policy estimate [Ber22].

In the *Policy Evaluation* phase, AlphaZero evaluates the current position using the collected data from recent online games. This data consists of actions used, rewards earned, and whether the game was won. Different positions in all games are used to have good samples to train the value network. This network is then trained with the data. The output of the network is the value function estimator, which is then used as the value function to predict the value in a given state. These values can be seen as the quality of each position, which can be used to improve the next moves. In other games, there are rewards involved, but in this type of board game, such as chess, there is only win-draw-loss, so the value estimator only evaluates a state in terms of these three outcomes.

The next stage is policy improvement, which uses the information from the *Policy Evaluation* stage. A second network is trained, but now instead of evaluating the current position, it improves the action probabilities by using move sequences from played games. These sequences are obtained using the *Monte Carlo Tree Search* (MCTS). The MCTS takes a *multistep lookahead* to gather promising probabilities of future moves. By using it, it allows a randomized approach to evaluating the next moves. It prunes unimportant moves using the policy network. This pruning reduces the number of potentially bad moves while minimizing the computational cost. It evaluates them against the observed moves that led to the best outcome. This process is iterative, and each iteration tries to improve the network. The new network weights are then used in the online game to generate better decisions, resulting in better patterns for the offline player to improve.

A common belief is that the MCTS are the core of AlphaZero's success, but with the deep mathematical analysis and references to dynamic programming by Bertsekas in his book "Lessons from Alpha Zero" [Ber22] he provides a fundamental and mathematical description of the idea behind AlphaZero and explains why it works so well. Without doubt, MCTS combined with deep convolutional networks provide significant performance, but they are not fundamental.

While deep neural networks provide superior approximation capabilities, normal neural networks can provide similar performance but with reduced sampling efficiency. Similarly, while MCTS is more computationally efficient than exhaustive search, it cannot surpass the accuracy of exhaustive search in terms of lookahead. In other words, AlphaZero plays much better online than the best player it has produced with offline training. AlphaZero's online player performs significantly better than its offline counterpart due to Newton's method [Ber22, Dro22]. This feature allows AlphaZero to make rapid and significant improvements in its decision making based on the initial guidance provided by off-line training.

The online approximation involves a single step in Newton's method for solving Bellman's Equation (2.6). The starting point of the Newton step is based on the results of the offline training. This step can be further improved by using longer lookahead minimization and rollout. AlphaZero takes advantage of this by using neural networks and the MCTS to increase efficiency.

A general explanation of how the AlphaZero algorithm works:

1. Creates the root node of the MCTS from the current state of the game.
2. Selects the next node. Or, if it is a leaf, it expands it to a boundary, guided by the neural networks and UCT.
3. The best action provided by the MCTS see Figure 2.13, is now used to make the next move.
4. This is repeated until the game is over.
5. This game with all states, actions and values is stored in the replay buffer.
6. The neural net learns the strategy and value function from all the games in the replay buffer.

A more detailed version follows, but the basic idea is that during self-play, the MCTS uses the neural networks to predict next states and uses the policy as a guide through the tree. The network learns from past games and improves its predictions.

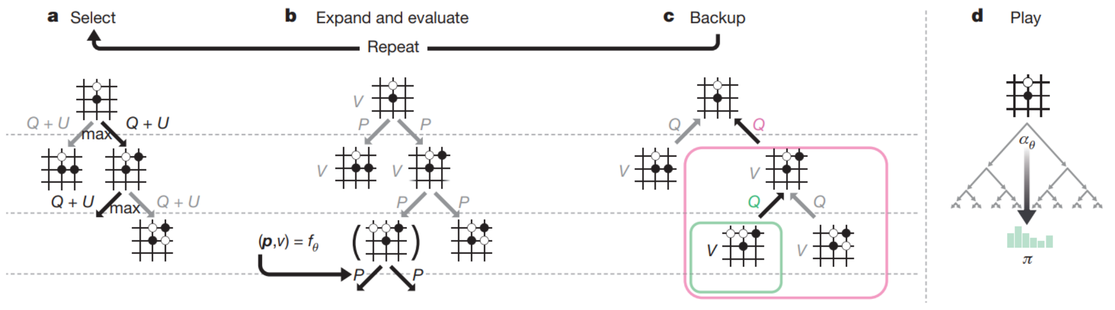


Figure 2.13: Used MCTS from AlphaZero, with a schematic Go board [SSS+17]. The functionality is the same as discussed in the last chapter. In this case, the MCTS is controlled by neural networks, which are used instead of a full rollout. Sampling is also based on the output of the value and policy network.

1. It starts by building a tree of possible game states. The tree starts with the current game state as the root node of the MCTS. Then it uses the network to create the next row of all possible leaves determined by the corresponding action. These leaves store the prior probability from the policy network and the value predicted by the value network.

2. Then it selects the next node in the tree, guided by the neural network. The most promising ones are calculated by the prediction of the network (already stored information from (1)) in combination with the UCB U (Upper Confidence Bound) from Equation (2.11) score. Let s be the state and a the action of the parent state or node. This can be expressed with

$$a_t = \arg\max_a (Q(s_t, a) + U(s_t, a)), \tag{2.23}$$

where a_t is the next action and $Q(s,a)$ is the value function. The neural network outputs a score for the position, which is an estimate of the expected future rewards from that position, e.g., the value, and the probability of further moves, e.g., the policy. The UCB is slightly different from the one described and commonly used in RL. The authors never mentioned why this formula was chosen over the usual one, or why the hyperparameters were set this way. AlphaZero's UCB U is defined as

$$U(s,a) = C(s)P(s,a)\frac{\sqrt{N(s)}}{1 + N(s,a)}, \tag{2.24}$$

where the number of visits in a state s and the subsequent action a is $N(s,a)$, $P(s,a)$ as the prior probability of choosing the action a in the state s. This is gathered by the neural network, as the policy estimation. The exploration rate, which increases over time C, is defined as

$$C(s) = \log\left(\frac{1 + N(s) + c_{base}}{c_{base}}\right) + c_{init} \tag{2.25}$$

with the hyperparameters $c_{init} = 1.25$ and $c_{base} = 19652$. As discussed at the beginning of the introduction to RL, there needs to be a balance between exploitation and exploration in the game, which can be adjusted by these parameters.

3. Next, the MCTS selects the child node with the most promising action of the current node and expands the tree. All previous leaves are updated accordingly with the value of other nodes.

4. Steps 2 and 3 are repeated until it reaches a leaf node in the tree. A leaf node is a node where there are no more possible moves or the event horizon is so far away that the search stops after a predefined number of visited leaves. AlphaZero used 800 simulations, which is interchangeable with the number of leaf nodes visited in the MCTS.

5. AlphaZero then chooses the move by adding an additional exploration factor to the best move chosen by the MCTS, the best move being the one with the most visits. The actual selected move is chosen with the prior probability P, defined as

$$P(s,a) = (1-\epsilon)p_a + \epsilon \times \mathrm{Dir}(\alpha), \tag{2.26}$$

with the hyperparameter $\epsilon = 0.25$ and for chess, for example, $\alpha = 0.3$. Dir refers to the Dirichlet function. With the move probability suggested by the MCTS

$$p_a = \frac{[N_0, N_1, ..., N_n]}{\sum_n N_n}, \tag{2.27}$$

where N is the number of visits each child leaf has. is the suggested action distribution from the MCTS. This additional noise is only useful in training games, not in competition.

6. AlphaZero repeats this process until the end of the game.

7. The value net now learns from the outcome of the game, which can be a loss, a draw, or a win. The policy network does the same, as the policy values compare the visits for the next possible actions. This learning process can be done in parallel to the games played. Each time a new game is played, it is saved and can be used for the next batch of training.

AlphaZero was able to learn how to beat even other AIs by a wide margin and easily outperform humans. This learning process involved playing games against itself. The starting point is a random policy that improves over time by using a MCTS to explore possible game states. Eventually it is able to play at a superhuman level. AlphaZero was groundbreaking for a number of reasons:

- The AI achieved superhuman performances against humans, but not only that, it also defeated other AIs.

- It can play several different games and perform at each of them.

- It was only while playing itself that it learned these games and discovered interesting new strategies. Humans play differently now because they learned from AlphaZero.
- It wins from seemingly disadvantageous positions and surprises all experts. It plays very aggressively and sacrifices pieces for positional advantages.
- It was a milestone in AI history, overcoming the last human-dominated bastion of Go.

This success of AlphaZero had a great impact on the field of AI. It showed that AI was capable of solving complex tasks without human intervention. In recent years, many algorithms have been developed. One example is MuZero.

2.5 MuZero

MuZero is an algorithm based on AlphaZero, but extended in important areas. MuZero is a model-free algorithm, which means that it learns the model of the game and uses it to plan the next actions, not like AlphaZero which is model-based. There is a need for a simulator that knows all upcoming states of the game. In the case of chess, AlphaZero used a simulator to know exactly the next possible states. All the information of the next possible states can be accurately gathered from the environment. Most modern games have some hidden information like randomness or the rules are too complex to simulate directly, which does not allow such a simulation.

In contrast, in model-free AI's, for example in the game Breakout, there is no open knowledge about how and where the ball will fall. So the AI has to learn the behavior of the game from scratch. The result is that MuZero can learn to play games without knowing the rules, because it learns the rules by observation. MuZero also outperforms AlphaZero in Chess, Shogi and Go. MuZero learns the rules by using past game states (observations) and feeding them into a neural network. The network tries to predict the resulting reward of the action taken in a current state and also in the next game state. With the predicted game state, the value and policy network is able to predict the value and policy. With the new game state, the next game state and reward can be predicted. This results in a prediction of multiple next states and rewards. So instead of knowing the next game state as AlphaZero does by simulating the games with known rules, the neural network predicts the outcome. Another big difference is that AlphaZero was only designed to play board games with no in-between rewards, there were only lose, draw and win as possible outcomes. Especially the Atari 2600 games and other modern ones have scores after reaching a goal and some did not even have a possible win. To learn the environment, to predict the value and the policy there are three different neural networks. A representation network, a prediction network and a dynamic network. Each of them fit together and built from each other.

Representation Network: The main goal is to encode the true state or observation of the game. In the case of a game state represented as an image, this would be the currently displayed image. As suggested in [SAH$^+$20] MuZero applied a 96×96 resolution with 3 color channels to the Atari games used as the observable image. In addition, they used the last 32 frames (observation). A ResNet was used to reduce the spatial resolution from 96x96 to an output resolution of 6x6, called hidden state. This was done by using

- 1 convolution with stride 2 and 128 feature maps, output resolution 48×48,
- 2 residual blocks with 128 feature maps,
- 1 Convolution with stride 2 and 256 feature maps, output resolution 24×24,
- 3 residue blocks with 256 feature maps,
- Average pooling with stride 2, output resolution 12×12,
- 3 remaining blocks with 256 feature maps,
- average pooling with stride 2, output resolution 6×6 and
- 16 remaining blocks with 256 feature maps.

A 3×3 kernel was used for all operations. Each residual block contains two convolutional and batch normalized layers with an additional ReLU after batch normalization. This adds up to 48 convolutional layers. This 6×6 output image is no longer human readable, it is just a representation of the image before. The network learns to capture all the important features of the original image and encodes them into the 6×6 image as the hidden state.

Prediction Network: This network takes a 6x6 matrix as input and the result is the value and the policy. This value is a real number and the policy is an array with the length of the possible actions that add up to one. As the prior probability for each action. It contains two heads, one is responsible for the value while the other is responsible for the policy. The base consists of

- 16 remaining blocks with 256 feature maps.

A 3×3 kernel is used for all operations. Each residue block contains two convolutional and batch norm layers. The **value head** then uses two fully connected layers with

- $6 \times 6 \times 256 = 9216$ neurons and
- 512 neurons

with a single output value - the value. The **policyhead** is similar with

- $6 \times 6 \times 256 = 9216$ neurons and
- 512 neurons,

but the output length depends on the possible actions in this game. The output then uses the softmax function to represent it as a probability. For both outputs the ReLU is applied last. **Dynamics mesh:** The dynamics network is used for immediate reward and hidden state prediction. It takes as input the 6×6 hidden state from the representation network, or the hidden state produced by itself. The network is built like the previous ones, using

- 16 remaining blocks with 256 feature maps,
- $6 \times 6 \times 256 = 9216$ neurons and
- 512 neurons.

The new hidden state is taken immediately after the remaining blocks, while the reward is a single value collected from the last layer of neurons. For both outputs, the ReLU is applied last.

Training the networks For the board games, they used 16 Google Cloud TPUs for learning from the replay buffer and 1000 TPUs for selfplay. For the Atari games, they used much less due to the small action space, 8 TPUs for learning and 32 for selfplay. To achieve the superhuman performance, it trained for 8 hours. This is a big chunk of hardware and hard to rebuild, but it can be seen from the size of the network that it must be a big one. Since AlphaZero and MuZero is out, there are many ideas out there that take the principles and improve them. These superhuman AIs are used to compete against humans, to compete against other AIs, to show the power of the algorithm, or to improve human gameplay. The focus of this work is to improve human knowledge and gameplay.

2.6 Chess Position Evaluation

For many years chess AIs have outplayed humans by a mile. It is a standard procedure to use AI to analyze games and evaluate the current position in a match. Many chess platforms have integrated an AI into the system, which allows an easy to use analysis of the played game. Before the AIs where based on neural nets, there were complex algorithms to evaluate the positions. Most of them were based on the method of counting pieces and their value. A pawn was worth one point, a rook five points and so on. With this system it was easy to see supposed advantages.

This system has many limitations, although it can be useful for evaluating trades and can be a guideline for a player. For example, trading three pawns for a knight because of the same point value. On the other hand, sometimes it is better to sacrifice a piece for a positional advantage or perhaps a checkmate [SOL22, Lin23]. An interesting method of calculating the win percentage is based on empirical data and is represented in a logistic model. Let $w \in [0, 1]$ be the win percentage and $p \in \mathcal{N}$ be the pawn advantage, then

$$w = \frac{1}{1 + 10^{-\frac{p}{4}}} \tag{2.28}$$

[Ise18]. It turned out that different phases of the game have different values for the pieces. The rook is worth less at the beginning than at the end. This and other adjustments increase the complexity.

Because of the long tradition in chess, this system was kept and improved by the AIs. It is called average centipawn loss (ACPL). One centipawn is equal to 0.01 pawns. This system allows to evaluate the positions more granularly, without losing the practical use of human readability. The best AI does not lose a single centipawn. The AI evaluates the position and predicts the outcome of the game, and with this information the ACPL value is generated. The number of pieces where used only implicitly and not as a base, like the time before AIs. The ACPL of a player can be seen as the quality of the player and the moves [Lic24]. AlphaZero's approach is different and rates the position between -1 (loss) and 1 (win), where 0 is a draw. This is easy for outsiders to understand, but for people in chess the ACPL is more intuitive, due to the practical use in analyzing games without AI. There are other formulas than in Equation (2.28) outside. Leela, a strong chess AI based on AlphaZero and improved in chess, uses

$$w = 111.714640912 \times \tan(1.5620688421 \times Q) \tag{2.29}$$

, where $Q \in [-1, 1]$ is the value from the MCTS. The authors of Leela suggest that it would be better to use the win percentage for each outcome (win, draw, loss). The ACPL is too limited and prone to errors. An example of such a misjudgment from the ACPL can be seen in Figure 2.14. The problem is that the difference from 0 to 1 points is a draw, from 1 to 2 is mostly a lost game, and from 11 to 12 it makes no difference [Lya].

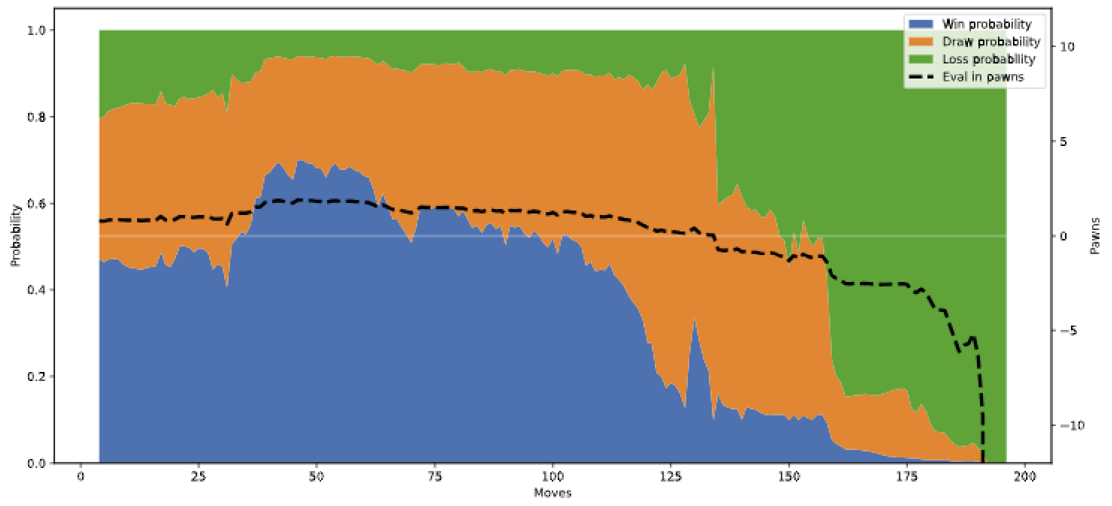


Figure 2.14: Analyzing a chess game with the Leela chess engine. The ACPL does not represent the actual loss event with the same significance as the percentage does [Lya].

With this last chapter, all the tools needed to build an AI that can help the players in any game to get better will be described.

3 Esports Trainer

After visiting the theory of AI and what technology is behind it, it is clear that human cannot beat AI's anymore if there is enough effort behind the success. There are many games that are unsolved by the AI, but mostly there is a lack of motivation or access to solve these games. Now, watching an AI play games may seem interesting for a short time, but games are made for humans. Humans play them for relaxation, for competition or other reasons, as explored earlier in chess, professionals use the concepts of games played by AIs to improve their own game. This led to the following questions

How can an esports trainer based on machine learning, give feedback to human players?

How can an esports trainer evaluate scores during a match?

How valuable is the feedback from an esports trainer?

With the help of MuZero, one of the strongest AIs, methods will be shown how to analyze the gameplay of the AI and how to extract new strategies. To demonstrate this, some games will be analyzed in detail.

3.1 Analyzing the player with the AI

For the following it is assumed that the AI is superior to human play in any game and at a superhuman level. It is possible to analyze each state of a game in depth to improve the human players knowledge of their gameplay. The MCTS provide that information and insights. This is the foundation to improve the human player in this thesis. The MCTS is based on the value and the policy, as described before. An example MCTS can be seen in Figure 3.1. The value is the expected long term reward, so each state has a value assigned by the neural network, of how good this actual state is.

With this information at any given time point, it is clear what outcome is predicted by the neural network. By traversing the search tree by any number of steps e.g., a far look ahead, the confidence to trust the MCTS rises, because many search paths are evaluated. When going deeper into the tree and exploring more nodes, the algorithm adapts the value due to the value, reward and policy predictions of the network and the visit count based on the UCB score from Equation (2.24). This value can be normalized between 0 and 1. This normalization used the maximum value from all simulations of all games or if the maximum possible reward of an game is known that this can also be used. In board

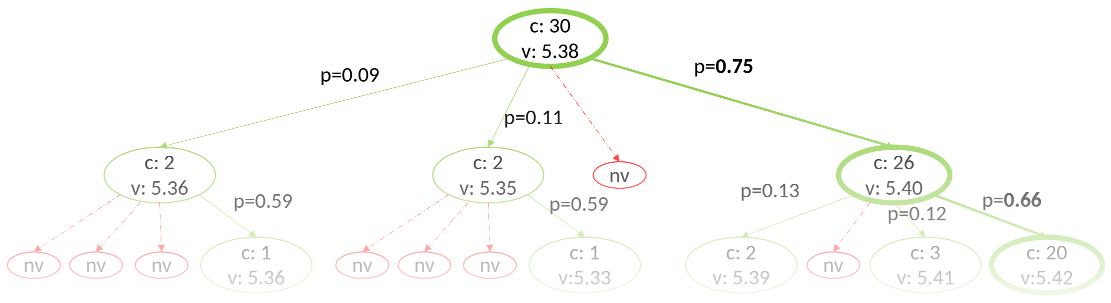


Figure 3.1: Example MCTS of the game Breakout. It is the first action to pick. Each node is a discovered state. c indicates the visits, v the value and p the priors of the policy. nv stands for not visited. The thick green nodes where the highest ranked path through the tree, according to the UCB seen in Equation (2.11).

games or games without reward the maximum value is 1 for a win. The values are based on the Equation (2.23) which includes rewards. After the normalization of the value, it can be interpreted as a percentage of how good the position is compared to the known maximum value. So there are few ways to rate a human player move in contrast to the AI.

1. **Perfect Game**: It takes the direct result of the MCTS p_a from Equation (2.27), which assigns a probability value to each possible action. This is compared with the action taken by the human. Taking the highest rated action results in 100 % accuracy and perfect play. A hypothesis to test is, do similar good moves have a comparable high probability?

   Let $P_s = \{p_1, p_2, ..., p_n\}$ and p_a be the number of visits per action a in a given state s, then the accuracy of a *Perfect Game* $A_{PG} : (a, s) \rightarrow [0, 1]$ of the chosen action a is defined as

   $$A_{PG}(a, s) = \frac{p_a}{\max(P_s)}. \tag{3.1}$$

   Now each possible action is compared to the best one of the AI. Similar good moves have a high value near 1 and bad ones should be near 0. This can be seen as an accuracy. This can be applied to every action of the player over a whole game. The important statistic is the average of the accuracy, where an accuracy of 1 is a flawless game. The gradient is another important factor, it tells about big vs. small mistakes on decreasing accuracy or medium vs. good play on increasing accuracy.

2. **Value only game:** Another way to evaluate human moves is to use the value inside the node as a reference instead of the output of the MCTS. So instead of using the

number of visits per node, only the value (which includes the predicted reward) of the node is used. First, this value must be normalized between 0 and 1. Normally the value is the average future reward and this is game depended. The highest value among all values explored in the nodes is used as the best move. Some trees are very deep, which limits exploration. With this approach, the MCTS must be modified to discover more states vertically. One way to do this is by making the exploration rate higher than the exploitation rate by using a lower c_{init} of the Equation (2.25). Additionally, the size of the simulations can be increased to explore even more, since deeper trees are more computationally expensive. If this is not enough another way is to remove the policy completely out of the UCB formula (2.24). This results in

$$U(s,a) = C(s)\frac{\sqrt{N(s)}}{1+N(s,a)}. \tag{3.2}$$

This ensures a completely exploration and a completely balanced MCTS.

The accuracy of the *Value only Game* $A_{VoG} : (a,s) \rightarrow [0,1]$ is defined as

$$A_{VoG}(a,s) = Q_n(a,s), \tag{3.3}$$

where Q_n is the normalized value function. This value is gathered by the MCTS. This approach should help to better distinguish between really bad, bad, good and very good moves. One of the problems might be that the performance of the AI is lacking without the policy. One thing to look at, is the impact during the evaluation phase. One important detail is that the training includes the policy.

3. **Easy Game:** Sometimes the optimal solution is hard to play and similar moves lead to the same or similar result. This value approach can be extended by not always taking the best solution, but instead taking the easiest or simplest solutions that are most error prone and lead to a high value. To do this, the value of each possible nodes for each action path is summed and divided by the number of nodes. This value is then compared to the other paths. The path with the highest value has the highest average value instead of the highest path value. This can be seen in the Figure 3.2. This *Easy Game* approach can be extended to let the AI suggest good enough moves, providing alternatives to the perfect action, which can lead to complicated states. The average value of different actions that are larger than a threshold can be used. This threshold has to be set by experience, but the higher it is, the better the actions are. For beginners it can be set lower.

4. The next approach is to watch the AI gameplay to see hidden strategies or new ways to play the game by a human expert. Some AIs are so creative that they even find and exploit bugs in the game [BKM+20]. Some insights can be gained not only from the latest trained version of the AI, but during the training process the outliers can be used to improve the understanding of the environment.

Its important to differentiate the results of the network and the MCTS. Both returning a value from a given state. The network recreates learned values from a given state. The

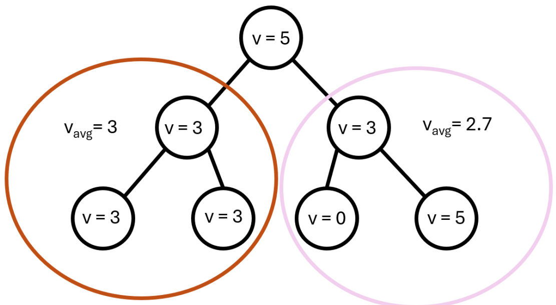


Figure 3.2: Example of an *Easy Game* calculation. The left path is taken instead of the right, which results in a higher average value of all possible nodes for this action, instead of a high value path.

MCTS uses this prediction only for guidance to explore more. The MCTS factor next moves in by using the prediction of the network of future actions.

Expected Errors All information generated by the AI must be used with caution, because the AI may not be perfect and may miss some opportunities or have some bugs.The big problem is that it only applies to the best known moves of the AI, which makes an evaluation difficult. For example in chess the standard evaluation in chess tournaments is the chess AI Stockfish. Every move made by professionals or even hobbyists is compared to what Stockfish would do and how it predicts the chances of winning. We saw in 2017 [SHS+17] that Stockfish was beaten by AlphaZero, which reduces the validity of Stockfish's evaluation because there are flaws in it. On the other hand, Stockfish is so much stronger than humans that for us humans the moves and rating are sufficient. At the top player level, the Stockfish rating is established and used every time.

3.2 Approach

The approach was to build a simple version of AlphaZero from scratch to get some implementation insights to understand the principle of the algorithm and how and where to grab the data. Only small games like Connect Four or Tic Tac Toe can be used on consumer hardware. Chess, as seen in the previous chapter, requires an enormous amount of hardware power to play at a superhuman level. This makes it impossible with today's consumer hardware standards. To show the analytical possibilities, it is necessary to use other games with a smaller action space than board games. The standard for comparing AI and human performance are the Atari 2600 games. All of them are model free, without any simulator. This is the reason for using MuZero instead. There is no official MuZero release with published source code, or even MuZero as closed software. There are many attempts to rebuild the AI based on the paper, the detailed description, and the pseudocode provided. Most of them only work on small games like Tic Tac Toe or Cart Pole where the goal is to balance the pendulum upright, more advanced versions like the popular muzero-general by werner-duvaud [WD19] generally work well, but when it comes to the Atari games like Breakout or Space Invaders, it did not achieve good performance [yz22]. Based on the pseudocode provided by [SAH+20], the use of [WD19, Fö23a, Fö23b] and a lot from [Ohm21] is used to create a sort of rebuild of MuZero. To fit into consumer hardware, many compromises were made in network size, VRAM, and RAM space. Most important is the time it takes to get good results, which indicates a good playing pattern. They used 40 TPUs in MuZero for the Atari games and over 1000! TPUs for board games like chess, here the simulations and experiments are done with the following hardware:

- NVIDIA GeForce RTX 3060 8 GB (which uses tensor cores to improve the machine learning performance. The performance is about 1/4 compared to their TPU which has 32 GB VRAM [Bal22, Red19]),
- 64 GB RAM and
- Intel i5-12600K with 10 cores.

The main software used to generate the experiments and analyze the games was

- the Operating System Windows 10 [BS19] and
- Python 3.10 [VRD09] with the following libraries:
  - *Pytorch*: «PyTorch is a tensor library optimized for deep learning on GPUs and CPUs.» [PGC+17], which accesses the cuda cores in the GPU,
  - *Ray*: «Ray is an open source, unified framework for scaling AI and Python applications. It provides the compute layer for parallel processing so you don't need to be a distributed systems expert.» [MNW+18],
  - *TensorBoard*: «TensorBoard provides the visualization and tools needed to experiment with machine learning.» [AAB+15] and
  - *Gymnasium*: «Gymnasium is a project that provides an API for all single agent reinforcement learning environments and includes implementations of common environments: cartpole, pendulum, mountain-car, mujoco, atari, and more.» [TTK+23].

This technology stack was used, because of the access hardware, broad support and state-of-the-art algorithms.

3.3 Software Architecture

The software project is used to test and showcase the results of the research. The base architecture is used from MuZeros pseudocode [SAH+20] and some replicas of it [Ohm21]. This base was extended by new games, new libraries, new networks, new configurations, bug fixes, the analyzer and many more things. The base structure can be seen in Figure 3.3.

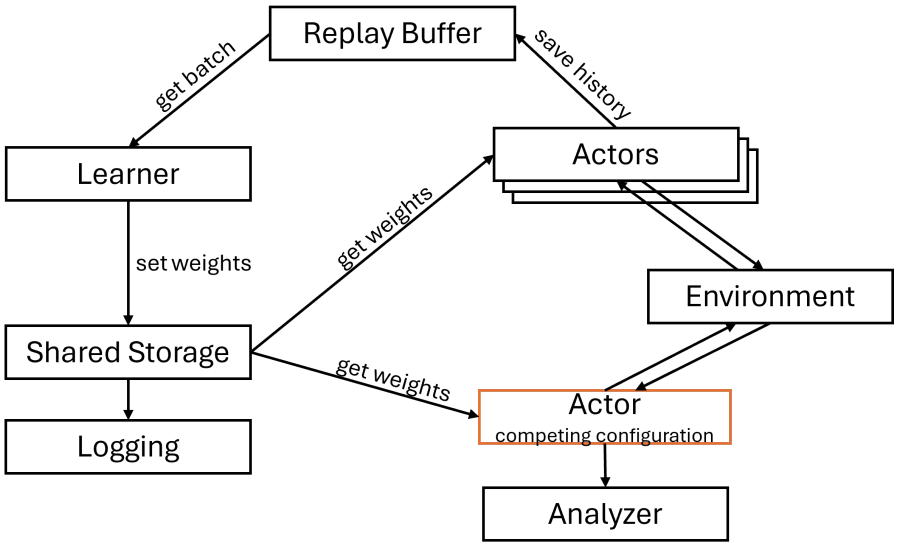


Figure 3.3: The code structure of the Esport Trainer. The actors that play the game, after each game the history is stored in the replay buffer. The learner, where the neural networks are trained, fetches the data from the buffer. The weights from the network are stored in shared memory, where they're used by the actors with the improved version.

Actor

The self-play to generate data was done by many actors using some kind of environment. The idea was to run the maximum number of actors on the CPU to generate a lot of fast data. This was different from MuZero and AlphaZero where they only used APUs for the games. During development it turned out that running the games on the CPU instead of the GPU was 10 times faster. These actors use a temperature during training, which adds a noise to the MCTS output to explore even more and reduce overfitting. Most of the time 8 to 10 different actors with different temperature were used. The actor uses the latest weights from the network in the MCTS. Each time a game is finished, a new game is started and the history (actions, states, and rewards) is stored in the replay buffer, ordered by error rate. This error rate is the difference between the actual outcome of a game and the outcome predicted by the network. This type of ordering is called prioritized replay and is used to improve performance, but it can also be a random order for a more uniform sampling process.

Learner

The learner fetches data blocks from the replay buffer. These batches consist of 1024 samples of five consecutive states, actions, rewards, and values from previous games. The batch size can be variable, but 512 to 1024 worked well. A forward and backward propagation is then applied to the data. The weights of the net are updated. Instead of learning not only to look one single step ahead, the network uses the five other elements to learn to predict better. The reward, value and policy loss is getting minimized by these operations.

Logging

The logging is done by TensorBoard. The loss values, the number of steps, the games played and other interesting data are logged. This data can be viewed in real time to monitor the training. The output is graphs that are highly observable and intuitive. Each new start of the AI is stored separately and can be compared with each other.

Analyzer

The analyzer is the core of this work and provides insight to the AI. The AI takes the history, mostly the player's observations, and uses the competitive actor without randomness or noise as a comparison. The analyzer then calculates the *Perfect Game*, the *Value only Game*, and the *Easy Game* with the action suggestions. The output is the value for each frame played. In addition, for better analysis, each frame is captured to reference the situations.

3.4 Data Engineering

Data engineering is more complex than retrieving some data from a collected set or database. MuZero is an online learning AI, which means that it generates data as it plays. The better it gets at playing the game, the more advanced game data it will have available. For each game played, the following statistics are generated and stored for each move, e.g., each frame, in a data block. The first three (observation, reward and done) are generated by the environment. The others are generated during training.

- Observations: Observations are recognized data collected directly from the game. It is usually the most important information. It is by far the largest amount of data to handle. This can be the whole frame as an rgb image with the given resolution. The Atari games are baseline at 210×160 pixel resolution, but all necessary information is also down-sampled to 96×96 as [SAH+20] used for MuZero. In some cases, to reduce complexity even further, instead of taking the rgb images, they were converted to grayscale images with only one color channel instead of three. This is especially beneficial for consumer hardware at the time. Instead of saving $210 \times 160 \times 3$ 8-bit integers (100 kB) per observation, there are only $96 \times 96 \times 1$ 8-bit integers (9 kB). It is also possible, at least with the Atari games, to access the RAM data and collect more and hidden information, such as life or time, which is not shown in the image. Not only that, the RAM state is only 128 bytes, which reduces the needed space by a factor of approximately 90. Not many games support this kind of API and more advanced and complex games use much more RAM. It mostly depends on the game and the AI support which one is preferable. In this thesis it is all about the comparison between a human and the machine and to give both the same information and observations, the frame observations are more in the focus.

- Rewards: Rewards are like points for completing a task. This task can be breaking a wall with a ball, shooting enemies, and many more. Some games like chess don not even have rewards. They are not mandatory. Rewards can speed up training by achieving early to win the game. By recognizing good moves or behaviors early in training and jumping from reward to reward, it is easier to master the game. Not all games work this way, some early rewards can be false positive and misleading, leading to longer training. Important for the training and understanding of the AI is that only the completed task gives rewards, i.e. if a bullet is fired 30 frames ago and hits the target now, the reward is collected now and the game did not tell the AI why this happened and what previous actions led to this result.

- Done: Each end of the game is characterized by a done flag. This makes it possible to determine a winner and the end of the game.

The following data is generated by the self-play during training, but is not gathered from the game environment.

- Actions: An action $a \in A$, where $A \in \{0, 1, 2, 3, ...\}$ is the action space, the size of which is defined by the game being played. Actions are game inputs. This can be a keystroke from a keyboard or even a controller input. The action space (all possible actions) are the keys supported by a game. At Table 3.1 it can be seen the action space for Atari games. There are 18 possible keys to press, but not all games need that much, e.g., Breakout only uses the first four actions that result in a behavior. The others do the same as NOOP (No Operation). In contrast to the MUZero paper, where they use all 18 keys for each game, this implementation uses only the supported ones. Each action used is followed by the next frame. The game waits for the next action to be pressed until it stops and nothing happens. This means that the game or the API must be time independent.

- Child visits: Reference to the MCTS where the search is performed by visiting the most promising states, e.g., child nodes. Each of these nodes is stored to trace back the MCTS in a human-readable form. For example, the Figure 3.1 is recreated with this information. It is not mandatory for the AI to do this, but to analyze the AI. Not all nodes will be visited. Some unpromising actions or nodes will not be visited at all.

- Root Values: The root value $q_r \in \mathbb{R}$ is the raw value of the actual state after the MCTS has run. It takes into account each simulation. In most cases this will be 50 different simulations of further possible actions. It is the actual score of the current state. For each new game state this value is stored. It is not normalized, i.e. it grows with training, because there are usually better states. The normalization is only done during the calculation in MCTS. The maximum value is taken from all

Value	Meaning	Value	Meaning	Value	Meaning
0	NOOP	1	FIRE	2	UP
3	RIGHT	4	LEFT	5	DOWN
6	UPRIGHT	7	UPLEFT	8	DOWRIGHT
9	DOWNLEFT	10	UPFIRE	11	RIGHTFIRE
12	LEFTFIRE	13	DOWNFIRE	14	UPRIGHFIRE
15	UPLEFTFIRE	16	DOWNRIGHTFIRE	17	DOWNLEFTFIRE

Table 3.1: Action space of the Atari games [TTK+23].

saved values over the whole training period. This value varies between different games and training time. It is also possible to have negative values, which indicates a training state where the whole game is predicted to be lost and it is hopeless for the AI to win. For statistical reasons the average value is also stored. This shows how confident the AI was in predicting the next moves.

- Errors: Errors $p_i \in \mathbb{R}$ are the difference between the root value and the predicted root value directly from the neural network. It is used for priority learning, where the block of data with the highest errors is used most during training of the neural network. This can be expressed as $p_i = |q_i - z_i|$ where $z_i \in \mathbb{R}$ is the predicted value from the neural network.

3.4.1 Data Storage

All data generated and observed during training is stored in RAM only. To improve performance, games and simulations are run in parallel. Each actor plays the game until its end, then it starts another game. The generated statistics and observations of each actor are stored in the replay buffer. The data storage and access mechanism is done via ray, a unified computational framework for AIs [MNW+18]. The replay buffer does all the preprocessing and preparation of the data for the neural network. Inside the replay buffer, the entire history of each game played is stored. The maximum number of saved games depends on the configuration, but is only limited by the size of the RAM. Most of the data comes from the observation, especially if it is an RGB observation. The weights and biases of the neural network are stored in a shared memory to share the latest updates of the neural network.

3.4.2 Data Preparation

The data must then be prepared to fit into the learning process and the neural networks. Each game as a whole is saved after a time constraint or the *done* flag appears. Because of the desire to look more than one step ahead, the next steps must be *unrolled* and the values discounted, so that the first step is the most important. It can be expressed as $q_{new} = q_r \times \gamma^k$, for each observation, where $k \in \{0, 1, 2, 3...K\}$, here $K = 10$, but it is an adjustable hyperparameter. In the last state of the game, it is set to zero and the policy is set to $\pi^A = (0, 0, 0, ..., 0)$, where A is the action space.

According to MuZero [SAH+20], there are two ways to sample data from the stored data. Uniform Sampling or Priority Sampling. They used uniform sampling on board games and priority sampling [SQAS16] on Atari games. The priority is defined by the error p_i.

3.4.3 Neural Network

To work around the hardware limitations and optimize performance, three different neural nets were created. One for non-pixel input data, where the game returns its memory instead of the observable frame. This is done with the FCN with only two layers in each of the different types (Representation, Prediction and Dynamics Network), which was suggested by [Ber22] that small function estimators are mostly sufficient. Next is the network rebuilt from MuZero [SAH+20], but to fit into the VRAM some changes had to be made. The last one is a really small version of MuZero, named TinyNetwork.

The only difference between the original MuZero network and the rebuilt one is that instead of 256 feature maps, only 128 are used in each convolutional layer. The FCN used uses a similar basic architecture to that proposed by MuZero. It uses five networks, a representation network, and a dynamics network, similar to MuZero, but uses additional networks for reward, value, and policy prediction. The size and construction are based on experiments and inspired by [Ohm21]. For all networks, the ReLU activation function was used between each convolutional and fully connected layer.

Representation Network: The main goal is to encode the true state or observation of the game. Similar to the original MuZero network, this is used to encode the input, but with only neurons. The input can be an image or the RAM state of the game. The input of any size is encoded into 50 output values and followed by 1 fully connected layer of 512 neurons.

This output vector of 50 numbers is no longer human readable, it is just a representation of the previous input. The network learns to capture all the important features of the original input and encodes them in the 50 numbers as *hidden state.* **Value, Reward, and Policy Prediction Networks:** The three networks are almost identical, the only difference is the output. While the output of the value and reward networks is a single real number, the policy network is based on softmax and the possible actions, which is an array of probabilities that sum to 1. This network takes as input the hidden state, a vector of 50 numbers. The basis consists of 1 fully connected layer with 512 neurons.

Dynamic Network: The dynamics network is used for hidden state prediction. It takes as input the hidden state (50 values) from the representation network or the hidden state generated by itself. The network is built like the previous networks by using 1 fully connected layer of 512 neurons.

In order to handle more complex observations such as images, but without being as heavy as MuZero, a **Tiny Network** was created. This tiny network is a small version of the ResNet with far fewer layers. It does, however, use skip connections. **Representation**

Network: The main goal is to encode the true state or observation of the game. In the case of a game state represented as an image, this would be the currently displayed image. This network reduces the spatial resolution from 96×96 to an output resolution of 6×6, also called the hidden state. This was done by using

- 1 convolution layer with stride 2, padding 1 and 32 feature maps, output resolution 48×48,
- 1 max pooling layer with stride 2 and padding 1, output 24×24,
- 1 convolution layer with stride 2, padding 1, and 64 feature maps, output resolution 12×12,
- 1 max pooling layer with stride 2 and padding 1 produces an output of 6×6 and
- 3 residual convolution layers with padding 1 to hold the size and 64 feature maps.

All operations use a 3×3 kernel. This adds up to 5 convolution layers. This 6×6 output image is no longer human readable, it is just a representation of the previous input. The network learns to capture all the important features of the original image and encodes them into the 6×6 image as the hidden state.

Value, Reward, and Policy Prediction Networks: The three networks are almost identical, the only difference is the output. While the output of the value and reward networks is a single real number, the policy network is based on softmax and the possible actions, which is an array of probabilities that sum to 1. This network takes as input the hidden state, a vector of 50 numbers. The basis consists of

- 2 residual convolution layers with 64 feature maps,
- 1 fully connected layer with $6 \times 6 \times 64 = 2304$ neurons and
- 1 fully connected hidden layer with 512 neurons.

Dynamic Network: The dynamics network is used for hidden state prediction. It takes as input the hidden state (50 values) from the representation network or the hidden state generated by itself. The network is built like the previous networks by using 2 residual convolution layers with 64 feature maps.

Training Process The training process starts after 5000 games have been played and saved. This is mandatory to avoid a small variation that can lead to getting stuck in a local minimum. Only one leaner is used to learn the weights. Each time the learning and playing process starts, even with a saved weight, new games had to be played because no history is saved on a hard disk. The main goal is to learn the value, policy, reward, and hidden state (dynamics and representation) function for the prediction. As mentioned before, each sample contains a current observation with the associated reward, value, and policy, and 10 subsequent hidden states, reward, value, and policy for the 10 future steps taken.

1. The display network is used with the observation as input. The output is the *hidden state*.
2. The value, reward, and policy are collected by the predictive network fed with the hidden state.
3. The MSE loss is computed for the value and the cross entropy loss for the policy.
4. The next hidden state is then predicted by the dynamical network.
5. This new hidden state is then used to predict the next value, policy, and reward.
6. The loss is calculated from the sampled and predicted value, reward and policy. Each loss is summed.
7. (5) and (6) are repeated 10 times.
8. The optimizer, either Adam or SGD, tries to minimize the loss using back propagation.

To take advantage of the parallelization power of the GPU, up to 512 samples per batch were taken. Each iteration with a batch is called a *step*.

Playing Games with the MCTS Actors play games in parallel until they finish a game or reach a time limit. They collect all the data needed for the learning process during self-play and store it in the replay buffer. Each actor works independently. Action selection is described in the AlphaZero chapter, basically it returns a probability vector of how good each move would be in the current situation. The network used by the MCTS updates the weights from every 1000th game played, so there is a balance between updating and working. Each actor runs on the GPU using CUDA cores. The number of actors used depends on the size of the network and the type of observation used. The only limit is the VRAM of the GPU.

4 Evaluation

To evaluate the AI and the human player, different games with different characteristics are selected. Some games have a very large observation and action space, these kind of games cannot be used to train the AI due to hardware limitations. In addition, games without a suitable interface for the AI cannot be used, which reduces the range of games enormously. Another limitation is that the action space must be discrete. Only one action can be pressed at a time. For this reason, most games are taken from the gymnasium library of the forked Open AI standard (reference) environments, such as the Atari 2600 games or other simple games like Pendulum or Cart Pole. Only the game Breakout from the Atari games is fully analyzed during each step. From the beginning of training to the analyzed human games with the fully trained AI. For reference and more in-depth analysis, Breakout is used because of the obvious good and bad actions to choose. The game Car Racing is used to show how the Esports Trainer can be used for more complex games. To evaluate human gameplay, only really simple actions were used. Most of the time, repeated actions are used to lose the game in the first few frames. This makes it much easier to draw graphs and show some highlights. Otherwise the games can have a few thousand frames. On Car Racing there is a little more advanced human player to compare the AI.

4.1 Experiments

4.1.1 Breakout

Breakout is one of Atari Inc.'s most famous games and was shipped with the Atari 2600 game console in 1978 [Acc01]. Breakout starts with 8 rows of bricks, as can be seen in Figures 4.1 and 4.12. The goal is to catch the ball with the paddle and hit the bricks. The paddle is only movable in the x-axis, the paddle uses some acceleration and not a linear movement pattern. The speed of the ball will eventually increase. After the first time there are no bricks left, a second set of bricks will be generated. Each set is worth 432 points, for a total of 864 points. This game is a good benchmark against other AIs [BNVB12]. It is not trivial to learn and many AIs are not able to finish the game [MKS+13]. This environment is simulated by the Stella emulator and distributed by The Arcade Learning Environment framework ALE [BNVB13]. There are two ways to play the game. One is to take only the memory image as observation data (Breakout RAM) and the other approach is to take the RGB pixels from the game. There is no time limit and the game is the same every time.

Configuration The network used was a fully connected network with 512 hidden nodes. There were 30 000 players due to self-play. The most important parameter is that

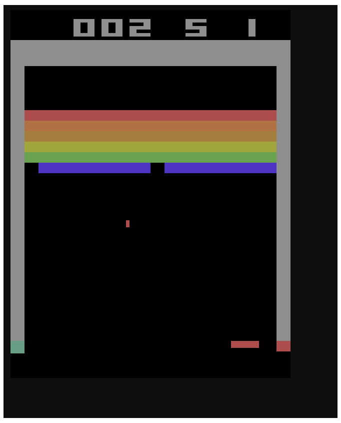
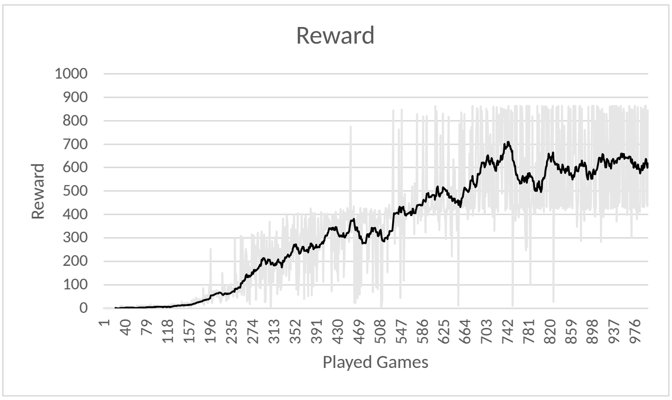


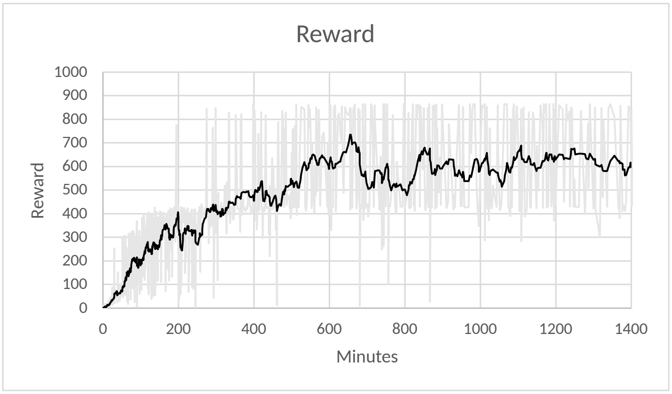
Figure 4.1: One Frame of the game Breakout.

50 MCTS simulations are used during self-play. Training the AI works well. After a few (about 5 hours) hours it has achieved the maximum number of points (864) for the first time. This was achieved after 4500 played games and 2.5 million training steps (batches in a forward and backward pass).

During training it is difficult to compare different learning curves. A learning curve example can be seen in Figure 4.2. Two different approaches can be seen to visualize the training speed and the trainings history. In the first Figure 4.2a the raw number of played games in combination with the reward are taken. This representations is misleading, due to the fact that the game durations are not equal. The AI is at the start weak and the games are short. A more intuitive scaling can be seen in Figure 4.2b, where the time is taken. It depends strongly on the hardware and the number of workers used, each game is played in a separate thread. So the time comparison is less useful when different configurations where used. In addition, the learning thread depends on the configuration, which makes it hard to reduce the training time to a single variable, such as epochs or training steps who are common parameters to compare the learning speed. Different values of c_{init} can be seen in Figures 4.3 and 4.4. The difference between a normal and a high c_{init} is marginal, but in most cases MuZero's default parameter is slightly better. The most interesting part is that $c_{init} = 1.25$ is more consistent and more powerful in the early stages, but the meaningfulness is limited, because of the small sample size of one training sequence. The average value during the training was also examined with different c_{init}. The high fluctuation in the value correlates with the game length as it can be seen in Figure 4.5. The more frames a game has, the less average values are achieved. For the best model different c_{init} values where compared against each other in Table 4.1. Another example is the convolutional network takes longer to play the games, but learns the game faster. This can only be observed by looking at the duration (see Figure 4.5). In most cases there were 8 games running in parallel with one learner. The same hardware was used for all experiments, and for the best intuitive overview most graphs are scaled by the training time. Different configurations were used to find differences in performance, especially the exploitation vs. exploration was examined. The experiments highlight that the parameter used by MuZero where the best performing one. Because of the focus on value, the average value of each game played was also examined. Short games seem to have a higher average value, but it is not clear what hidden information is still to be seen. The three network loss values are very unspectacular as it can be seen in Figures 4.6, 4.7 and 4.8, they are very low from the beginning to the end and the network seems to learn very fast. In the beginning there are only a few games to learn and because these networks are immediately used to produce new games, the prediction does not differ much from the previous games.



(a) Learning curve of Breakout, scaled by played games.



(b) Learning curve of Breakout. Learning curve of Breakout, scaled by time.

Figure 4.2: These are both graphs of the same learning process of the game Breakout. The same parameters were used as in the MuZero paper. The difference is in the scaling of the two graphs (Time and played games).

After training, the AI is competitive and strong, which can achieve the maximum number of points in Breakout. The comparison with a human is done by importing the action sequence and the frames to compare it with the AI. To show a basic analysis, the human actions are very simple and repeated at some point as it can be seen in Figure 4.9. Four different methods where presented in the last chapter. Each of them is now used to

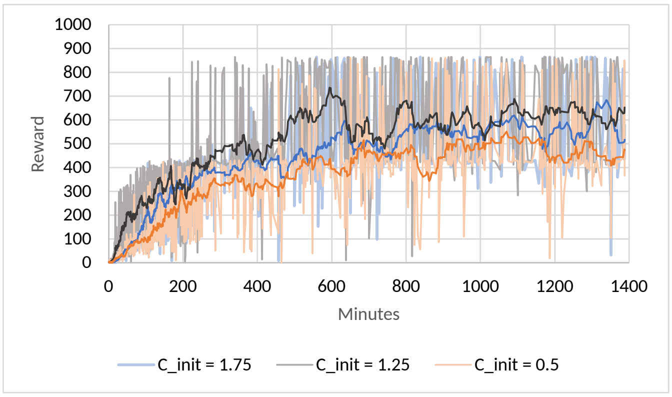


Figure 4.3: Reward learning curve of the game Breakout, with different c_{init} to adjust exploration versus exploitation.

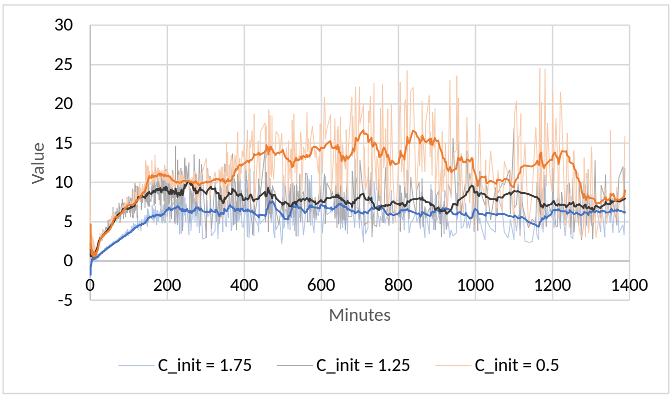


Figure 4.4: The average values (expected future reward) of the played Breakout games during training, with different c_{init} to see the difference between exploration and exploitation.

analyze the game of the human:

1. Perfect Game
2. Value only Game
3. Easy Game

Perfect Game

First, the human actions are compared to the perfect AI game using the MCTS result. The human actions are compared to the MCTS result. The AI uses the player's frames as the current state. With this information, the AI gives the best possible action according to the AI. For example, action *2* (move right) was taken by the human and the MCTS

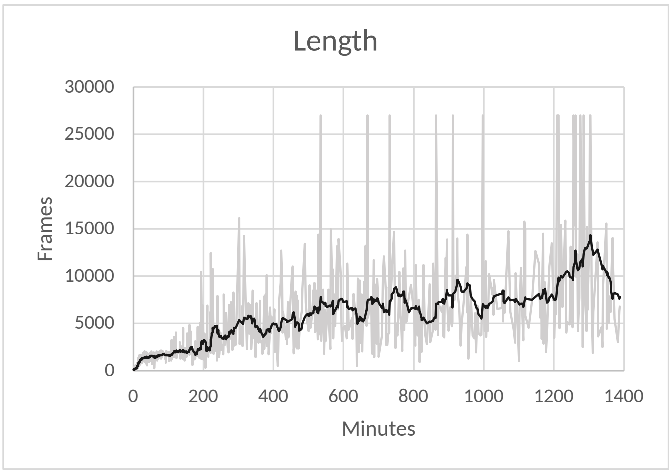


Figure 4.5: This trend shows the length of games during training. The maximum duration is limited to 27,000 frames to avoid infinite game loops.

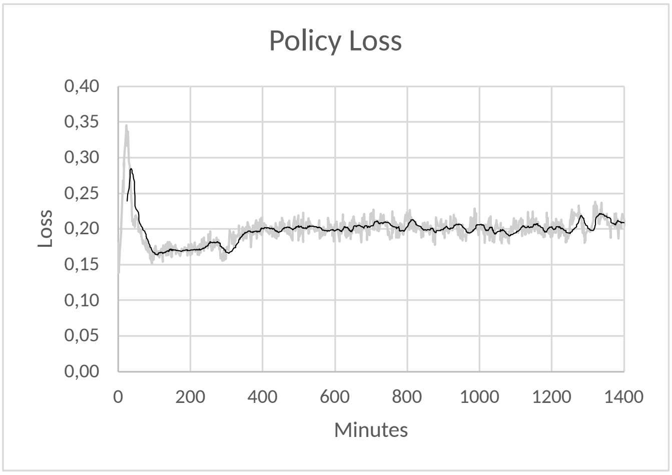


Figure 4.6: Policy Loss during the training of Breakout. It is calculated by the Cross Entropy Loss function.

returns an array of action probability distributions: [0.1, 0.1, 0.3, 0.5] (which always add up to 1). Usually the AI chooses the action with the highest probability, in this case the last action (3). The accuracy of choosing action *2* (move right) is according to Equation (3.1): $A_{PG}(3) = \frac{0.3}{0.5} = 0.6$, which is an accuracy of 60 %. As described this method is only a comparison between the optimal moves and the human move. The AI does not care about similar good moves.

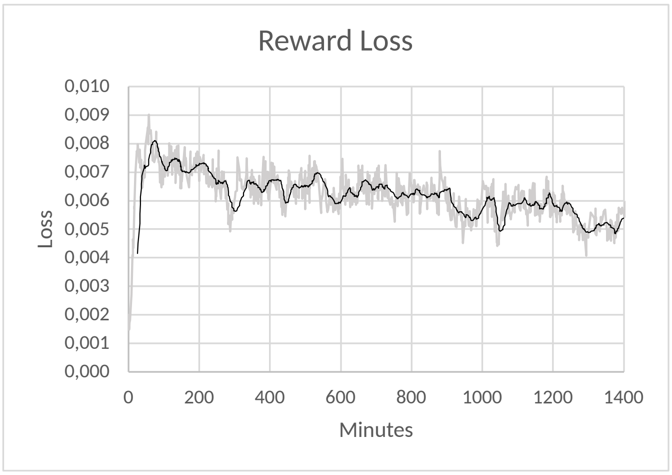


Figure 4.7: Reward loss during the training of Breakout. The MSE Loss function was used.

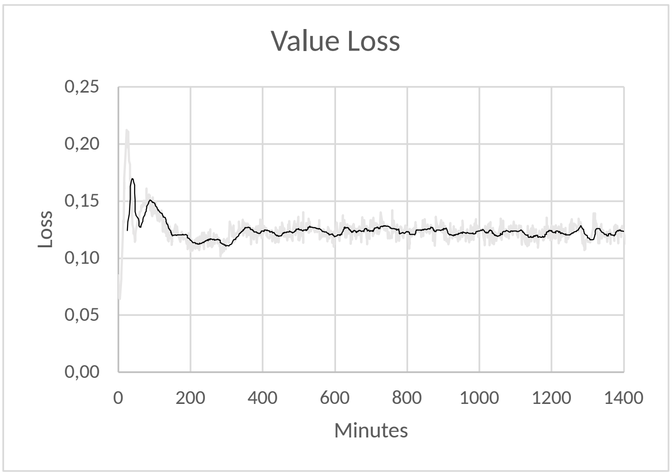


Figure 4.8: Value Loss during the training of Breakout. The MSE Loss function was used.

Value only Game

The AI does not care about bad search paths in the MCTS, which sometimes result in a very deep path, but the human player sometimes makes bad or not perfect moves. To force the AI to get a better picture of the current state and the alternative actions, the policy is turned off for the following analysis. This was tried to be done by setting the c_{init} accordingly. Due to a low c_{init} the value of each node is more important than the policy. This results in more similar move suggestions for each action. This c_{init} can be set to 0, to factor only values in, without the policy, this should results in a maximum exploration rate. This was not helpful at all. During testing it turned out, that the impact is still there. The MCTS did not go in the width, instead it was deep as before.

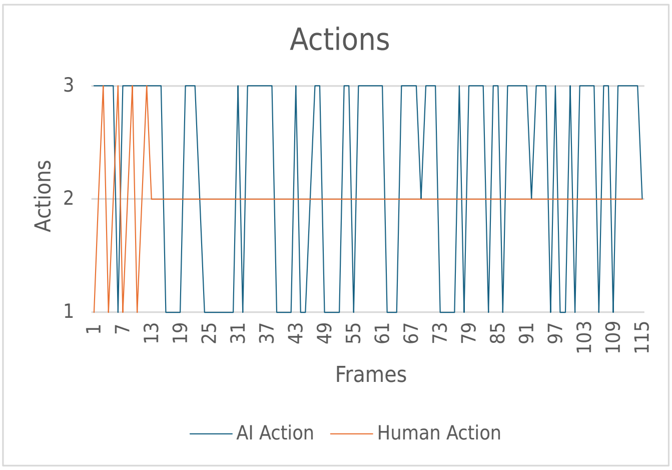


Figure 4.9: A Comparison between human and AI actions in Breakout. The human actions are used to play the game. The actions are encoded, so that 0: No Operation, 1: Shoot, 2: Move Right and 3: Move Left. The following human action sequence was used: 123123123123 followed by 100 times action 2.

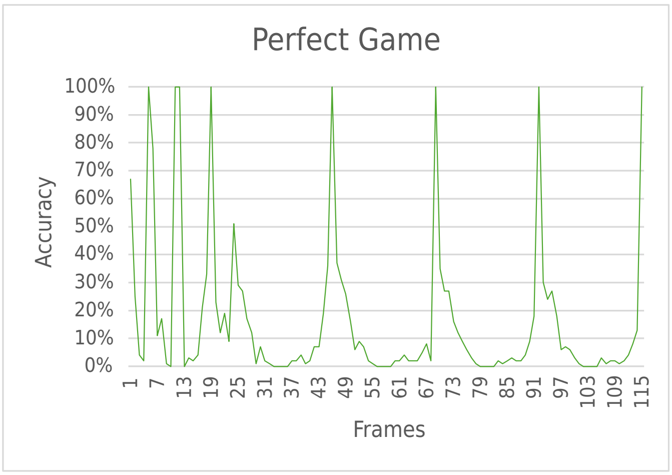


Figure 4.10: This graph shows the accuracy compared to a *Perfect Game*, where the human actions are directly compared to the action distribution from the MCTS.

The idea is to use the value only, or to have a minimal impact from the policy to guide the MCTS with the value. The solution was to remove the priors complete from the UCB Equation (2.24). This resulted in the desired evaluation without the policy. So only the value is considered. This results in a perfectly balanced tree where every simulated node can be observed, as can be seen in Figure 4.19.

The Value only Game uses this balanced tree to compute the action probability distribution. The accuracy itself is calculated in the same way as Equation (3.1), but

MCTS	c_{init}	Reward
10	1.75	411
15	1.75	408
20	1.75	415
30	1.75	438
50	1.75	409
10	1.25	428
15	1.25	426
20	1.25	414
30	1.25	412
50	1.25	401
10	0.5	400
15	0.5	430
20	0.5	398
30	0.5	393
50	0.5	360
10	0.1	426
15	0.1	394
20	0.1	402
30	0.1	403
50	0.1	406

Table 4.1: Analyzing the different c_{init} in Breakout. These tests where done with 100 game of different parameters and shows the average reward for this games.

with a different distribution, since the policy is not included. This means that no policy is involved in the MCTS exploration and node selection process. In comparison to Figure 4.10 similar good moves are not rated as bad, because the value of the nodes is similar. In the Figure 4.11 it can be seen the trend for the accuracy. Bad moves have 0 % accuracy and result in bad states. At this point, no action can be rewarded. These valleys indicate the loss of life during a game. Any large percentage drop can be considered a big mistake. Due to really bad moves by the human, the accuracy is very low most of the time. The average accuracy is 16 %. It can be observed that there are 5 low points where the accuracy is around zero. These five periods are the life losses in the game. The first one is not easy to detect because there are good moves in between bad ones. These big valleys are at frames 11, 36, 56, 81 and 101 as it can be seen in Figure 4.12. In Breakout the player has five health points. Because of the bad moves, the value of the states is zero. In Figure 4.12 the frames of these moments can be seen and it highlights the problem. In each of these frames, the ball is on the far left, the bar is on the far right, and it appears

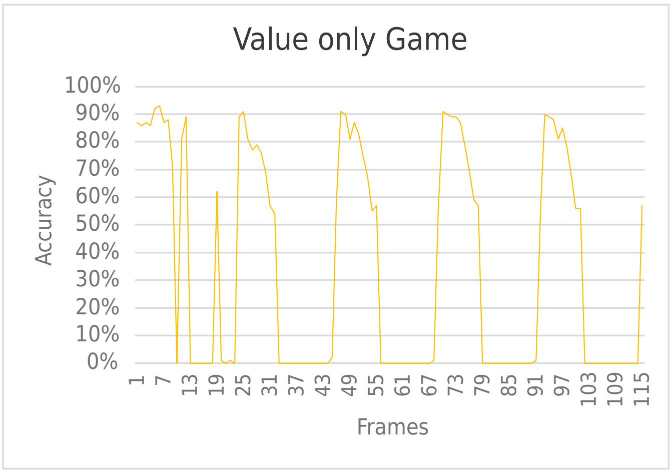


Figure 4.11: Breakout *Value Only* Game, a comparison between the human player and the AI by using only values without the policy.

that the ball is unreachable at this point.

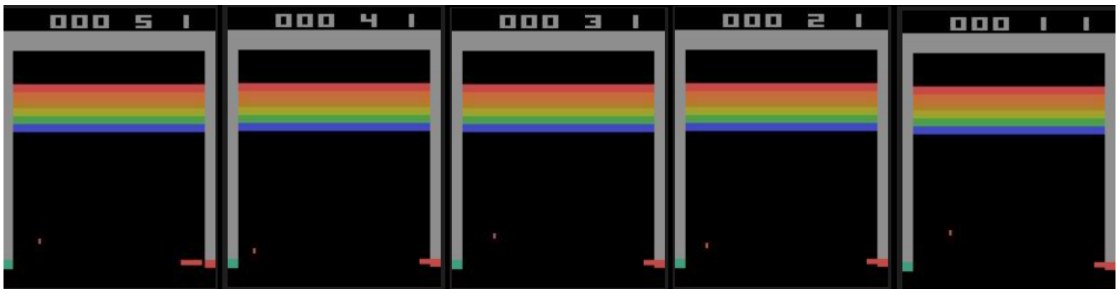


Figure 4.12: Breakout Screenshots at frame 11, 36, 56, 81 and 101. This highlights the situations of no return.

Easy Game

The *Easy Game* does not use the distribution for accuracy, it measures the difference of the value of all explored nodes. All explored search paths are considered and evaluated. The amplitude of *Easy Game* values strongly depends on whether the policy is used or not. It uses the average values of all visited nodes in the search path, a good value should indicate a simple and good action.

The comparison on using the policy and not using it can be seen in Figure 4.13. Without policy the the rating is very smooth and more foreseeable, but the this small drops would suggest that only minor mistakes where made, but that is not the case. It is each time a life loss and at the end, the game loss. In an *Easy Game* one might ask after looking at Figure 4.13 how the actions and conditions without policy can be rated so high, even though lives were lost. There are two guesses, one is the percentage drop is due to using

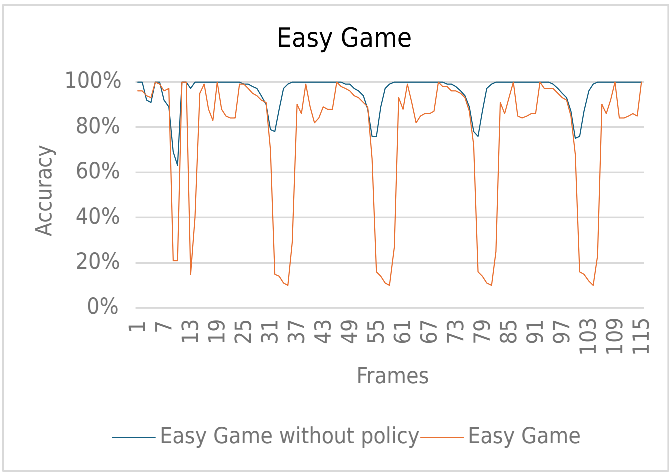


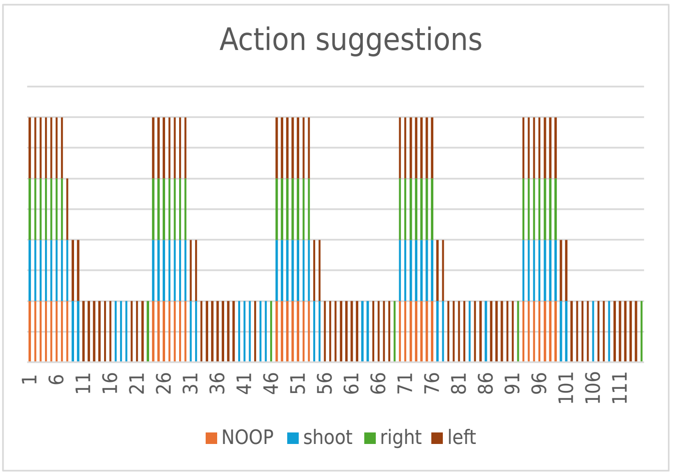
Figure 4.13: Breakout *Easy Game* comparison with and without policy.

a bad action in the given situation. At the moment the situation is hopeless and it does not matter which action is taken, the percentage goes up. The second guess is, that the MCTS without the policy is not deep enough to see the problem. Once the life loss is near, it knows that it can get new rewards shortly after the soft reset. Another picture can be found in the action recommendations in Figure 4.14, where all actions who have at least a 80 % accuracy are visualized. With and without policy approaches are similar by their suggestions. The main difference is that without policy there are no good actions after the point of no return. Sometimes actions who are not perfect are totally fine for the players. Sometime the suggested top action result in a more difficult game.

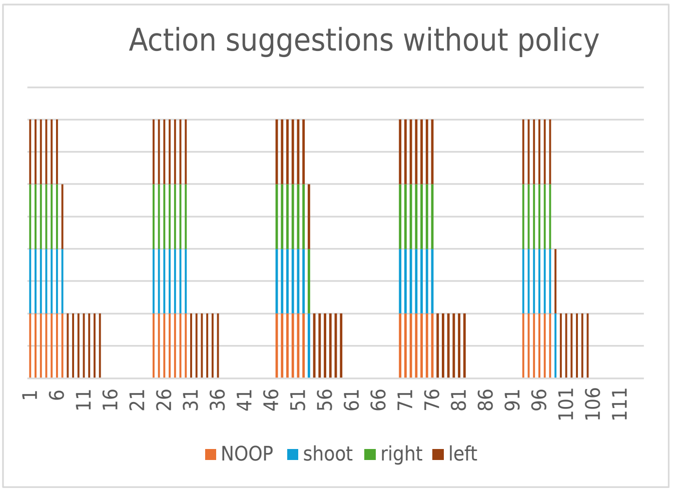
In Figure 4.14 it can be seen that at some point, only the action *left* promises to be rewarded, no other action results in a well-rated state. It is like a forced checkmate for every possible action, an inevitable event. By taking the policy into account, there is always a move, it is not clear why. Maybe as stated before the depth of the tree has influence. This one suggested action is not necessary as a good one, it is more like taking the best of the bad actions. In the game Breakout the analysis shows that it is possible to extract the key positions of the game by using all the different methods. Some options give a clearer picture of the game. Especially the options without considering the rules will help to detect mistakes in human play. The action suggestion shows which actions lead to a good game, without focusing too much on the *Perfect Game*. The problem is that without a policy the AI is different. Stronger in some chases, weaker in others.

4.1.2 Car Racing

Car Racing is a simple racing game simulator as it can be seen in Figure 4.15. The goal is to drive a lap as fast as possible. This car is powerful and uses rear wheels to drive, which means it is not a good idea to turn and accelerate at the same time. There are ABS sensor values, speed, wheel position and a gyroscope visible. The game is considered



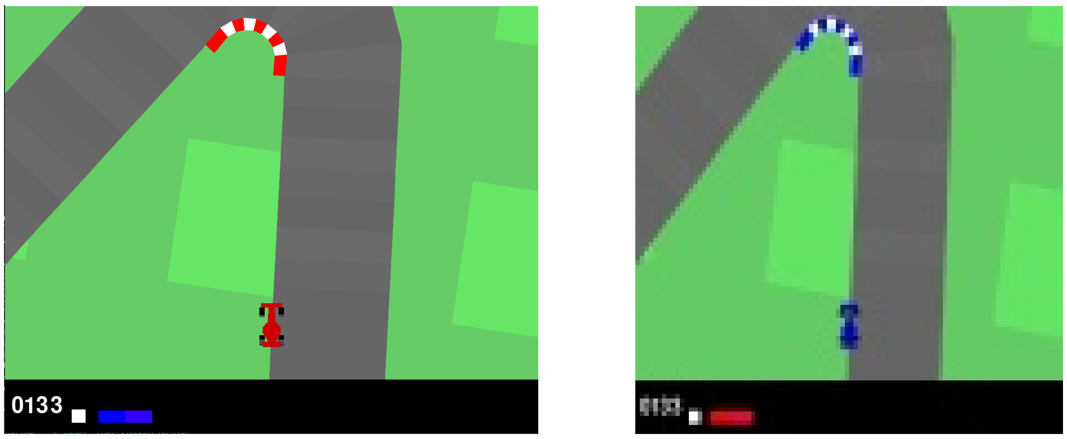
(a) A Breakout *Easy Game* action suggestions.



(b) A Breakout *Easy Game* action suggestion without policy

Figure 4.14: Breakout action suggestion with and without policy.

solved when the AI consistently scores over 900 points. The reward is collected by visiting road tiles. This game is much more complex than the previous ones and the driving can be optimized much more. First the AI is compared with a very weak human player, this highlights the theoretical insights, similar to Breakout. The AI is then compared to a human with experience. In some cases, the human beats the AI by a few points. The human has not practiced the game or the track much, so it is by no means an expert. The AI did not train this specific course at all before the comparison. It is not the purpose of this work to determine the better player, nor is it a competition. Two different AIs where trained, one was trained by using the fully connected network, which means that the image of the RAM memory of the game are used as the observation as described in the last chapter. To then find the best hyperparameter, 100 random tracks where used as comparison. In addition the Tiny Network which includes convolutions and uses the screen of the game as observation was the second major base of the AI. Table 4.2 showcase the results. A different number of simulations were used for the MCTS, these are the visited nodes in one whole search. The depth is the resulting depth of the MCTS after the simulations. These parameters are used as an example and cannot be seen as a general guideline. For the competing AI, which was used for the analysis, the FC was set to 250 for the *Value only Game* and FC 50 for the *Perfect Game*. The differences where marginal between the FC and the Tiny Network with the same time used during training.



(a) A screenshot of the game Car Racing, played by a human.

(b) Down sampled image (including changing the aspect ratio).

Figure 4.15: Comparison between the original and the down-sampled image at frame number 45.

Perfect Game

The *Perfect Game* in Car Racing is similar to the one in Breakout. Both show the exact moment when the human action differs from the AI and is categorized as a bad action e.g. a low accuracy. Figure 4.16 shows an example of such behavior. During the evaluation, the AI was very clear about which actions were preferable due to a high visit count on the corresponding nodes in the MCTS, so all others were rated very low. On the other hand, when comparing the AI with a strong human, the reason for the rating of the human actions is unclear. There are many drops in accuracy and actions rated with 0 %. There are a few reasons why this happens. First of all, the game lasts longer, this game of Car Racing lasts about 200 frames. The AI and the human, both achieve about 920 points on the same track. The meaningfulness of accuracy drops in a *Perfect Game* is greatly reduced due to a lot of them. The AI judges 25 % of the human actions with zero percent accuracy. After replaying the situations, it is unclear why the AI rates these actions so low. There are a few guesses here, one important one is that the AI is not perfect and

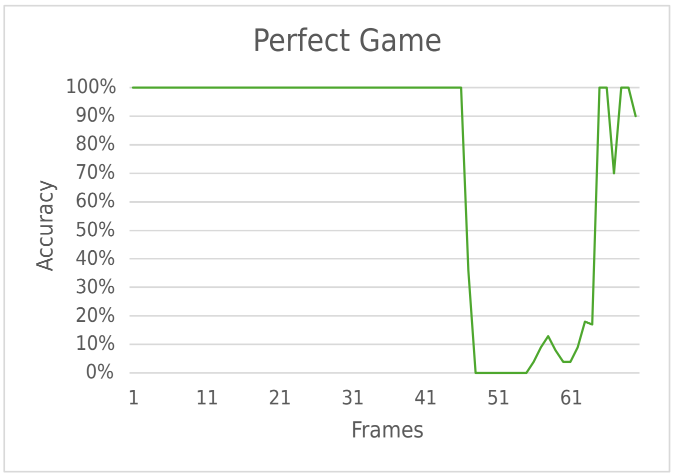
Network	MCTS simulations	Policy	Reward	Depth
FC Network	500	False	879	4.2
FC Network	**250**	**False**	**920**	**3.8**
FC Network	125	False	875	3.3
FC Network	50	False	875	2.9
FC Network	30	False	790	2.7
FC Network	15	False	790	2.5
FC Network	500	True	879	25.4
FC Network	250	True	905	25.8
FC Network	125	True	875	21.7
FC Network	50	True	914	26.5
FC Network	30	True	914	16.5
FC Network	15	True	914	9.0
Tiny Network	500	False	774	4.7
Tiny Network	250	False	668	4.2
Tiny Network	125	False	643	3.8
Tiny Network	50	False	562	3.3
Tiny Network	30	False	457	2.9
Tiny Network	15	False	38	2.7
Tiny Network	500	True	906	28.8
Tiny Network	250	True	906	27.6
Tiny Network	125	True	907	28.3
Tiny Network	50	True	869	23.8
Tiny Network	30	True	909	16.5
Tiny Network	15	True	909	9.0

Table 4.2: A table showing the different results with different parameters in the game Car Racing.

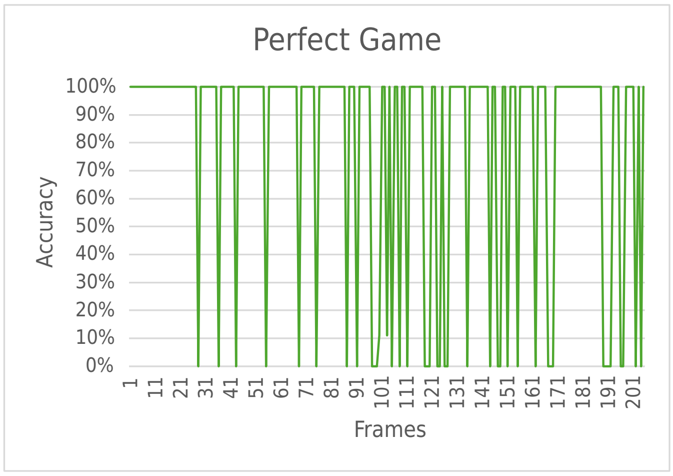
apparently not even better than the human player. Another factor is that the AI usually only rates its preferred actions as best, even though others are also viable. This is in line with the thesis of this work and is the reason to introduce more and different methods to analyze the game, especially the experiments without the policy. Most of the accuracy drops are caused by starting a turn. Sometimes the AI starts to turn earlier, sometimes vice versa.

Value Only Game

In the game Car Racing the *Value only Game* is very similar to the *Perfect Game*. It gives a clear picture of the current state in every frame as it can be seen in Figure 4.17. This means that no policy is involved in the MCTS exploration and node selection process. Very low valleys indicate the unavoidable loss. Any large percentage drop can be considered a big mistake. In comparison to Figure 4.14 similar good moves are not



(a) This graph shows the accuracy in a *Perfect Game*, where the human uses the gas action in the game Car Racing the whole time.



(b) A Breakout *Perfect Game* played by a well-playing human who scored similar points to the AI in the game.

Figure 4.16: This figure shows a different *Perfect Game* analysis between bad and good human actions.

rated as bad, because the value of each node is similar. This means, that each action result in a similar outcome and each bad action can be compensated by the next. So the rating is not as low as in the *Perfect Game*. The game is not lost after a blunder, such as driving in the grass, but i wont be a new high score. Due to the fast pace of the game, the moment of an such an event comes very abruptly. Most of the time, accelerating is the most promising action, at least until a corner will come and it needs to brake and turn left or right. The AI is optimized to drive the track as fast as possible. There are only a few frames between winning and losing. This was different in the game Breakout, where there were many frames left to correct the mistakes. This cannot be seen in the *Perfect Game*, only in the *Value only Game*. It is very interesting that after the car has gone far off the road and is in a terrible state where it cannot get any reward, the accuracy goes up. This behavior occurs because the AI does not know what to do in this situation and considers any action as good, where no action would lead to any reward. Maybe turning around and going back to the road would do that, but the AI has only learned to drive on the road and does that very well.

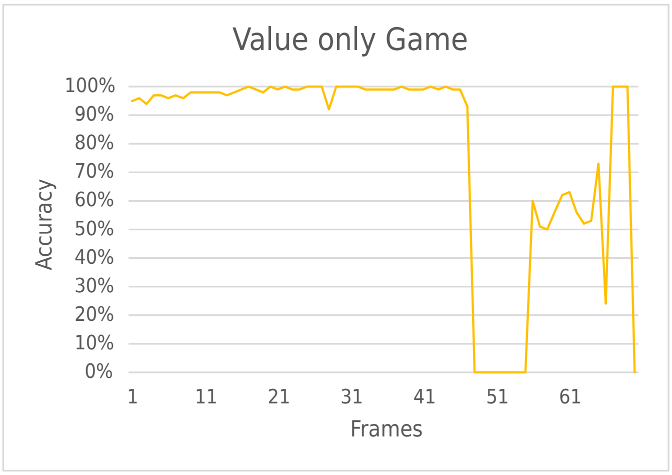
To compare it with the (expert) human, one thing has to be mentioned. Consistency is a big problem of the currently trained AI, depending on different parameters the AI performs differently. Changing the MCTS simulations changes the performance of a single track and the reward varies between 0 and 900 points. Most of the time it performs well with a reward of about 900 points. The best configuration is used for the evaluation phase. The configurations are also only representative for the track played by the human. The comparison between a good human player and the AI in a value-only game shows that the AI mostly rated the actions with an accuracy above 70 %. Small percentage drops happen all the time, mostly on corners, because there are the most difficult things in a driving game. The assumption is that the AI would simply start turning a little later than the human. It is hard to say which way is better. Both have a similar result.

Easy Game

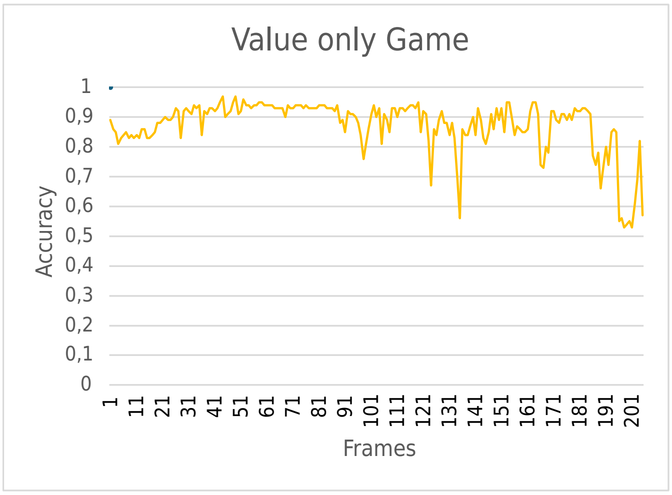
The *Easy Game* does not use the distribution for accuracy, it measures the difference of the value of all explored nodes. All explored search paths are considered and evaluated. The amplitude of *Easy Game* values strongly depends on whether the policy is used or not. It uses the average values of all visited nodes in the search path, a good value should indicate a simple and good action. The action suggestion seen in Figure 4.18, which highlights the difference of the policy. Without the policy it looks like that only the most important thing is, not to leave the track, but it lacks in efficiency. This is shown by the fact that braking is recommended most of the time.

4.2 Discussion

All three different approaches to learning from AI are examined here. It is very difficult to produce meaningful results because of the sheer amount of training time, different parameters, and different games. The results are using the observed data and interpreting it. The trained AI is on a high level of skill, but far from being perfect. On Breakout it achieves in around 70 % games rewards of 864 and some times it got stuck at 432 points. The first obstacle is at 432, there are only a few bricks left and the AI cannot aim, so it played for a long time at this stage. After the 432 all bricks appear again, the problem now is that at this stage the speed of the ball is very fast, but the single frame observation is the same as at the beginning of the game. This is indistinguishable and makes it hard to train well. In Car Racing the track is different each time, to perfect the skill it would



(a) A Car Racing *Value only Game* analysis. Only simple actions where taken.

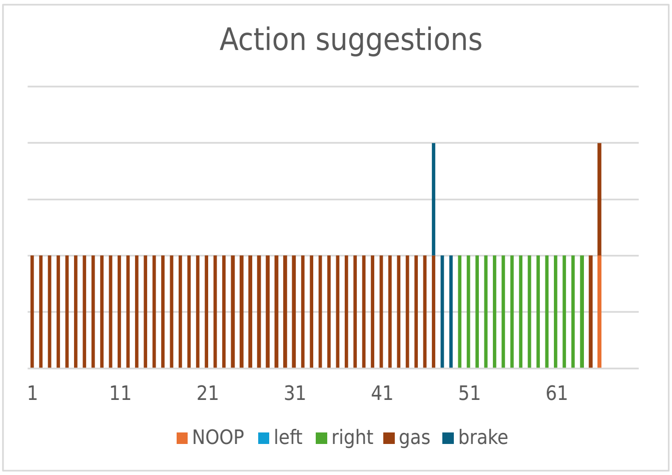


(b) A Car Racing *Value only Game* analysis. A good human driver was analyzed.

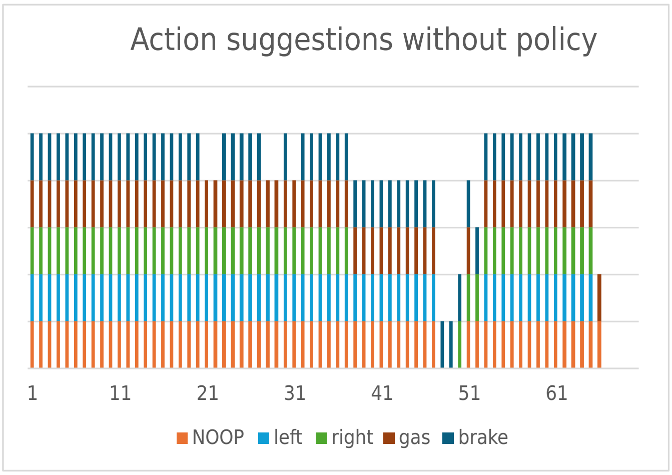
Figure 4.17: Comparison between the human player and the AI by using only values without the policy.

be take a lot more training's time.

One of the most important parameters is c_{init} from the Equation (2.25), which indicates exploration vs. exploitation. A low c_{init} favors the value over the policy, which leads to more exploration. During training, this parameter can have a positive effect on the learning curve, as can be seen in Figure 4.3, where Breakout was played, but it does not make that big a difference at all. During the evaluation phase, the impact was not there ether. For the *Value only Game* it did not change anything and to achieve the desired outcome of taking the value only, other things had to be done. Overall the specific UCB



(a) A Car Racing *Easy Game* actions suggestion.



(b) A Car Racing *Easy Game* action suggestion without policy.

Figure 4.18: Car Racing action suggestion.

formula from AlphaZero from Equation (2.24) had not a high impact at all, at least the fixed values. Four different approaches to improve the human players performance where examined. The first approach was to take the output of the MCTS directly as a guide and compare moves for the human.

4.2.1 Perfect Game

The *Perfect Game* show the perfect moves according to the AI. All other actions were rated with low accuracy. This kind of analysis only helps players who are already playing at a high level and only need minor improvements in certain situations. It is also imperative that the AI achieves consistent superhuman performance. This was a big problem during the evaluation. The AI was very inconsistent, sometimes performing and sometimes not. It was rarely really bad, but for example in Breakout it fluctuates a lot between 400 and 900 points, even after long training. It is not clear if more training would lead to more consistency or if the implementation has a problem. From the MuZero [SAH+20] it is not clear if the results where that consistent, but they also did a lot more training.

One of the problems that needed to be solved next was that only the best actions were suggested and all other even similarly good ones were rated 0 %. This problem was seen in Breakout, where most actions were equally good, because only the actions before the ball drop were important. This lead to another problem, big mistakes were not distinguishable from other good but not perfect actions. There is a method needed to explore the other nodes in the MCTS. This approach is called *Value only Game*.

4.2.2 Value only Game

The first attempt was to rebalance the UCB score using c_{init}, but this changes the result only slightly. The $P(s, a)$ from the Equation (2.24), which is the policy predicted by the network, was set to 1 to use only that value. This resulted in a balanced tree. All nodes were explored equally and the result was a *Value only Game*. Especially in Car Racing there were very nuanced graphs of action accuracy. Big mistakes were shown as 0 % and good but not perfect actions were shown as 90 %. This seems to be a good approach to improve human performance and give feedback on actual moves.

One concern was that the AI is weaker without the guidelines. MuZero is an algorithm that combines both value and policy approaches, which is one of its strengths. During training it was clear that without the policy it did not achieve anything and was far from learning effectively, but as a fully trained AI it learned the values guided by the policy very well. This led to some experiments with and without the policy. During these tests the inconsistency of the AI was another factor, but it seems that a well trained AI is strong even without the policy. Not as consistently strong as with the policy, but usable. It was not clear at the beginning how well the value-only game characterized the gameplay, which led to another way of evaluating the actions.

4.2.3 Easy Game

The big difference from the *Easy Game* and the others is that it does not use the number of visits to choose the action. It uses the average value of all the nodes explored. This results in not choosing the best path in the MCTS, but the path with the highest average value. To explore the result, games with and without the policy were compared. The difference is similar to the *Value only Game* vs. *Perfect Game*. The *Easy Game* with policy has one advantage over the *Perfect Game*, namely that the evaluation of non-perfect actions is more granular and not just between 0 and 100 %. During the experiments it can be seen that the accuracy during big mistakes is very low, around 10-20 %. This is a bit misleading, because the game is lost at this point. Maybe there should be a threshold where the game is lost. On the other hand, by factoring out the policy, the AI seems to be fine with all actions. At the moment of the big mistakes the accuracy dropped to

70-80 %. This is also a bit counterintuitive. The graph is much smoother and also shows bad actions. Both methods work and show bad actions and rate passable actions high, which is the goal of the Easy Game. All three methods can be used for different use cases and detailed analysis of games. A big problem with all these graphs and data is that it is hard to make an intuitive connection to the game. For this, state observations e.g. via video analysis is very useful.

4.2.4 Game Observation

The exact frame can show where the mistake of a player happen and which action should be the best or similarly good. In all analyses here implicit images of the game were used as orientation. Each value drop was compared with the frame at the same time to clarify the result. The AI was observed during the self-play to validate the result. It can be used not only to compare the other methods. It can also be a way to gather new strategies to solve a game. In the examples such a new way was not found. One reason may be that the games are too simple or well researched, another reason may be that the AI was not trained enough. The training games are not analyzed, it could be that hidden strategies where found, but where not that promising. Game observation is very useful and is used in many games to improve performance. Even in soccer games, video analysis is used to improve the game.

4.2.5 Miscellaneous

The AI itself had many parameters to train and compete with. There is a big room for improvement on both sides. The inconsistency was a big problem during the evaluation phase, but with better hardware this could be avoided. With better hardware comes the ability to play other and modern games. The AI is not limited to 2D games or a small action space. It scales very well to more hardware. The evaluation itself can be used on a consumer CPU. It is also possible to run the algorithms in real time, during a game.

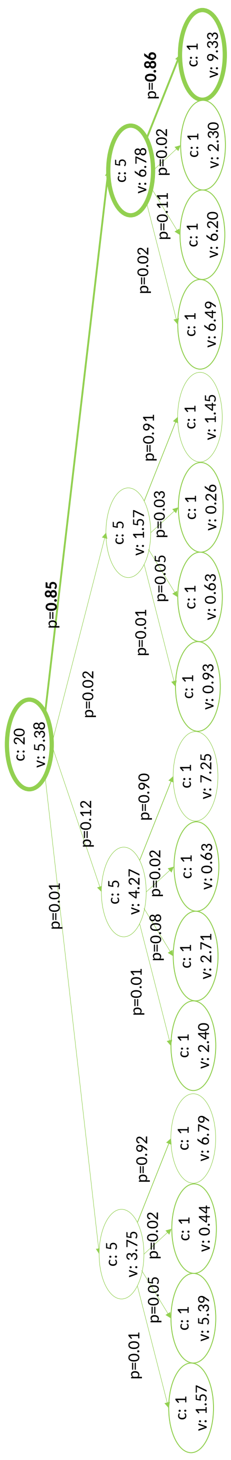


Figure 4.19: The first level of a perfectly balanced MCTS of the game Breakout. The balance comes from the left out policy, only the value is taken into account. The current state is frame 11. This is one of the last possible moments where it is possible to catch the ball, at least by not taking action *2* (move right). This can be read from the nodes. The variable v is the value and c is the number of visits during the simulation.

5 Conclusion

The aim of this work was to improve the knowledge and skill of a human player with AI. Game AI is based on reinforcement learning, where the AI learns by receiving feedback. RL's value and policy function is key. The Bellman equation describes this behavior. This equation has too many steps for one solution. Neural networks achieve the value and policy for further states. One of the best AIs, AlphaZero, and later MuZero, learns to play without knowledge of the game, just by playing itself. It uses Monte Carlo Tree Search with a neural network to achieve superhuman performance. This AI is then used to analyze human play. Four feedback methods were presented and analyzed.

First, the *Perfect Game* was described, where the performance of the AI was compared to the human's actions by using the evaluation from the MCTS for each possible next action. The AI rate only the exact actions it would take in this moment as good ones. These action led to the perfect game according to the AI. Other actions, even though they would lead to the same state, were rated as bad. This method can be used mainly for highly skilled players who want to improve and only want the perfect move, but they may also be interested in equally good actions. The *Value only Game* showed some insight into how to improve all types of players. It is based on the value from the MCTS and does not have the bias from politics during the evaluation. It showed mistakes and weaker actions very accurately. The trade-off is that it may not be the most competitive configuration of the AI. In contrast to these two approaches, the *Easy Game* does not use the UCB as a final calculation of the exploration of the nodes, instead it takes the sum of the values from each explored node path. It scores all actions that did not lose the game as good. This can be used to help beginners in a game or to improve intermediate players. It had some flaws too because it is more optimized to not lose a game instead of winning. In addition, *Value only Game* and *Easy Game* can give good action suggestions for each situation. These suggestions are very helpful to show which actions are possible and lead to good results at that moment. At least during the evaluation it was proven that watching the AI replays improves the understanding and knowledge of the game. Each situation can be watched in detail, with the corresponding evaluation of the AI. The insights are great and can reveal new and interesting strategies. This can be done during the training and evaluation phase.

There will certainly be many different approaches and many different new AI systems in the future to help people improve their game performance. Different games can be approached from different angles. There is a lot of research to be done to improve the insight of an AI.

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[Acc01] Joe J Accardi. The Ultimate History of Video Games: From Pong to Pokemon. *Library Journal*, 126(16):134–134, 2001.

[AG21] Fadi AlMahamid and Katarina Grolinger. Reinforcement Learning Algorithms: An Overview and Classification. In *2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, volume 2021-, pages 1–7. IEEE, 2021.

[Agg18] Charu C. Aggarwal. *Neural Networks and Deep Learning: A Textbook*. Springer, Cham, 2018.

[Alg23] Florian Algo. BatchNorm and LayerNorm. `https://medium.com/@florian_algo/batchnorm-and-layernorm-2637f46a998b`, 2023. last accessed on 13/09/23.

[Aue00] P. Auer. Using upper confidence bounds for online learning. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 270–279, 2000.

[Bal22] Stephen Balaban. RTX 2080 TI deep learning benchmarks with tensorflow. `https://lambdalabs.com/blog/2080-ti-deep-learning-benchmarks`, Aug 2022. last accessed on 22/03/23.

[Bar03] Peter L. Bartlett. *An Introduction to Reinforcement Learning Theory: Value Function Methods*, pages 184–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[BB23] Christopher Michael Bishop and Hugh Bishop. *Deep Learning - Foundations and Concepts*. Cham, Springer, 2023.

[Bel57] Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, USA, 1957.

[Ber] Dimitri Bertsekas. Lecture 4, 2021: Approximation in value and policy space; rollout. ASU. `https://www.youtube.com/watch?v=k-r2el7rW7I`. last accessed on 04/11/23.

[Ber22] Dimitri Bertsekas. *Lessons from AlphaZero for Optimal, Model Predictive, and Adaptive Control.* Athena Scientific, 2022.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization, 2016.

[BKM$^+$20] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula, 2020.

[BNVB12] Marc G Bellemare, Yuri Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2012.

[BNVB13] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

[BPW$^+$12] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[Brü93] Bernd Brügmann. Monte carlo go. Technical report, Technical report, Physics Department, Syracuse University Syracuse, NY, 1993.

[BS19] Ed Bott and Craig Stinson. *Windows 10 inside out.* Microsoft Press, 2019.

[Cou06] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[Dro22] Iddo Drori. *The science of deep learning.* Cambridge University Press, 2022.

[DS05] Morris H. DeGroot and Mark J. Schervish. *Probability: Theory and Examples.* Brooks/Cole, Belmont, CA, 4 edition, 2005.

[Fuk80] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, April 1980.

[Fö23a] Robert Förster. AlphaZeroFromScratch. `https://github.com/foersterrobert/AlphaZeroFromScratch`, 2023. last accessed on 05/05/23.

[Fö23b] Robert Förster. MuZero. `https://github.com/foersterrobert/MuZero`, 2023. last accessed on 12/05/23.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.

[Hig20] M.G Higgins. *Esports.* White Lightning Nonfiction. Saddleback Educational Publishing, 2020.

[Hon23] Hee Jung Hong. eSports: the need for a structured support system for players. *European sport management quarterly*, 23(5):1430–1453, 2023.

[HSS15] Kazuyuki Hara, Daisuke Saito, and Hayaru Shouno. Analysis of function of rectified linear unit used in deep learning. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, 2015.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition, 2015.

[IS15] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015.

[Ise18] Gerd Isenberg. Pawn Advantage, Win Percentage, and Elo. `www.chessprogramming.org`, 2018. last accessed on 11/02/24.

[Jin21] Dal Yong Jin. *Global Esports: Transformation of Cultural Perceptions of Competitive Gaming.* Bloomsbury Academic, 2021.

[KB17] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2017.

[Kel19] John D. Kelleher. *Deep learning.* The MIT Press essential knowledge series. The MIT Press, Cambridge, Mass. London, 2019.

[KLM96] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[KS06a] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[KS06b] Levente Kocsis and Csaba Szepesvári. Bandit Based Monte-Carlo Planning, 09 2006.

[KW08] Malvin H. Kalos and Paula A. Whitlock. *Monte Carlo methods*. WILEY-VCH, Weinheim, 2008.

[LBE10] Bart De Schutter Lucian Busoniu, Robert Bebuska and Damien Ernst. Reinforcement learning and dynamic programming using function approximators. *SciTech Book News*, 34(3), 2010.

[Lic24] Lichess.org. Lichess.org. https://lichess.org/, 2024. last accessed on 10/11/23.

[Lin23] Ray Linville. Understanding Average Centipawn Loss In Chess. `https://www.chess.com/blog/raync910/average-centipawn-loss-chess-acpl`, 2023. last accessed on 11/02/24.

[Lya] Alexander Lyashuk. Win-Draw-Loss evaluation. `https://lczero.org/blog/2020/04/wdl-head/`. last accessed on 11/02/24.

[Mah96] Sridhar Mahadevan. Average Reward Reinforcement Learning: Foundations, Algorithms, and Empirical Results. In *Recent Advances in Reinforcement Learning*, pages 159–195. Springer US, Boston, MA, 1996.

[MC23] Andrea Manzo and Paolo Ciancarini. Enhancing Stockfish: A Chess Engine Tailored for Training Human Players. In Paolo Ciancarini, Angelo Di Iorio, Helmut Hlavacs, and Francesco Poggi, editors, *Entertainment Computing – ICEC 2023*, pages 275–289, Singapore, 2023. Springer Nature Singapore.

[Med] Glassbox Medicine. Convolution vs. cross-correlation.

[MKS$^+$] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning.

[MKS$^+$13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Breakout on Atari 2600: A Benchmark for Reinforcement Learning. *Proceedings of the 25th International Conference on Machine Learning*, 2013.

[MKS$^+$15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning, author=Mnih, Volodymyr and Kavukcuoglu, Koray and Silver, David and Rusu, Andrei and Veness, Joel and Bellemare, Marc and Graves, Alex and Riedmiller, Martin and Fidjeland, Andreas and Ostrovski, Georg and others. *Nature*, 518(7540):529–533, 2015.

[MNW$^+$18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications, 2018.

[NH10] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[NKJS23] Justyna Nyćkowiak, Tomasz Kołodziej, Michał Jasny, and Piotr Siuda. Toward Successful Esports Team: How Does National Diversity Affect Multiplayer Online Battle Arena Video Games. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, volume 2023-, pages 3902–3911, 2023.

[NNXS17] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the Gap Between Value and Policy Based Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 30, pages 2775–2785, 2017.

[Ohm21] Jim Ohman. model-based-rl. `https://github.com/JimOhman/model-based-rl`, 2021. last accessed on 20/07/23.

[Pan97] Bruce Pandolfini. *Kasparov vs. Deep Blue : the historic chess match between man and machine.* Fireside book. Simon & Schuster, New York, NY [u.a.], 1997.

[PGC$^+$17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

[Pip18] Julia Piper. To Stay in the Game, Colleges Recruit Esports Coaches.(HIRING TRENDS). *The Chronicle of higher education*, 65(12):A39, 2018.

[Pit19] Silviu Pitis. Rethinking the Discount Factor in Reinforcement Learning: A Decision Theoretic Approach. *arXiv preprint arXiv:1902.02893*, 2019.

[RDS$^+$15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[Red19] Nikhil Reddy. A Survey on Specialised Hardware for Machine Learning, 07 2019.

[RM51] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400 – 407, 1951.

[SAH$^+$20] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, December 2020.

[SB18] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[SHS$^+$17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, 2017.

[Sil15] David Silver. Lectures on Reinforcement Learning. URL: `https://www.davidsilver.uk/teaching/`, 2015. last accessed on 15/08/23.

[SOL22] NATE SOLON. Centipawns Suck. `https://zwischenzug.substack.com/p/centipawns-suck`, 2022. last accessed on 11/02/24.

[SPC23] Ashish Kumar Shakya, Gopinatha Pillai, and Sohom Chakrabarty. Reinforcement learning algorithms: A brief survey. *Expert systems with applications*, 231:120495, 2023.

[SQAS16] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay, 2016.

[SSS$^+$17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature (London)*, 550(7676):354–359, 2017.

[STIM19] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How Does Batch Normalization Help Optimization?, 2019.

[SWD$^+$17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms, 2017.

[SZ15] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, 2015.

[Sze10] Csaba Szepesvári. *Algorithms for Reinforcement Learning.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2010.

[Tes95] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Commun. ACM*, 38(3):58–68, mar 1995.

[TTK+23] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, March 2023.

[unk24] unknown. Esports Earnings. `https://www.esportsearnings.com/`, 2024. last accessed on 19/07/24.

[VRD09] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.

[WD19] Aurèle Hainaut Werner Duvaud. MuZero General: Open Reimplementation of MuZero. `https://github.com/werner-duvaud/muzero-general`, 2019. last accessed on 02/08/23.

[yz22] dillonmsandhu yungangwu, JohnPPP and zsn2021. The model does not converge for breakout. `https://github.com/werner-duvaud/muzero-general/issues/211`, 2022. last accessed on 02/08/23.

[ZK17] Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks, 2017.