DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

STUDIENARBEIT

des Studiengangs Informationstechnik
der Dualen Hochschule Baden-Württemberg Mannheim

THEMA

Development of a chess playing agent using reinforcement learning

---

Timo Salisch

18. April 2023

Bearbeitungszeitraum:	18.10.2022 – 18.04.2023
Matrikelnummer, Kurs:	4785113, TINF20IT1
Betreuer:	Prof. Dr. Sudermann-Merx
Unterschrift Betreuer	

Erklärung

Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema

Development of a chess playing agent using reinforcement learning

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Gilching, den 18. April 2023

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
FIDE	International Chess Federation
GAE	Generalized Advantage Estimation
GUI	Graphical User Interface
KL	Kullback-Leibler
PPO	Proximal Policy Optimization
TCEC	Top Chess Engine Championship
TPU	Tensor Processing Unit
TRPO	Trust Region Policy Optimization
WCCC	World Computer Chess Championship

Abstract

The goal of the project is to see if it is possible for an average person using common hardware (Nvidia RTX 2060) to achieve a strong result using reinforcement learning. To do so, a reinforcement learning agent is trained to play the game of chess using PPO. The best result achieved on the given hardware within 1000 iterations of training is better than an agent choosing an action randomly, it wins 13.5 % more games than it loses. But in comparison to Stockfish or an average human player, it is still weaker. This leads to the conclusion that to achieve usable results for a complex task using reinforcement learning, a huge amount of computation power is needed. Reinforcement learning has therefore become a topic and approach only useful to big organizations that have access to the needed computation power, at least for more complex tasks.

1. Introduction

Reinforcement learning is a topic that receives a lot of attention and is applied to many fields and projects. The interest in this method was evoked by many successful applications of reinforcement learning to solve problems better than seemed possible before. One project that became very widely known is AlphaZero[1]. A team of DeepMind developed a program using reinforcement learning that was able to defeat humans and the best programs at that time in the games of chess, go and shogi. The same algorithm was able to play all these games on a very strong level instead of having one program per game. Another big impact was created by OpenAI. They developed a program (called OpenAI Five) that can play the computer game Dota2 and control all five players of one team at the same time[2]. This program became the first AI to defeat the world champions in an esports game[3]. But reinforcement learning is not just restricted to games, it can be applied to almost every field imaginable. Google created a robot that taught itself how to walk without the help of humans in just a few hours[4]. And even the new and very famous chatbot ChatGPT uses reinforcement learning methods for its training[5]. It has been shown multiple times that reinforcement learning methods can achieve great results unmatched by other programs or even humans. But what all these projects and programs have in common is that they have been trained by big organizations with access to enormous computation power. To create all the results mentioned above, very strong hardware has been used to train the programs and achieve these results. This evokes the question if similar results can be achieved with hardware available to an average person, such as a normal computer or laptop with a graphics card. Or if it is necessary to have such a big amount of computation power to achieve usable results. To test this, a chess program is developed using reinforcement learning on a standard hardware. Chess is chosen as the task because it is one of the most researched board games in regard to playing it with a computer. There are furthermore programs such as AlphaZero or Leela Chess Zero[6] that have already shown that it is possible to create a chess program with reinforcement learning and train it to become one of the strongest programs created. These have also been trained using very much computation power, so they can be used as a baseline for

comparison regarding the question if the hardware used for training plays a key role in its results. To create a meaningful comparison, multiple agents have to be trained to fine-tune the parameters of the agent and training and see which work best. To make these comparable, they have to be trained with the same training parameters, such as the same hardware and being trained for the same amount of iterations. These results can furthermore be used to make recommendations which (hyper-)parameters work best for the chosen environment and algorithm, and might be useful especially when training with little computation power. To test the trained agents, it should be possible to play against them using a Graphical User Interface (GUI).

2. Background

2.1. Chess

2.1.1. Gameplay and chessboard

Chess is a two-player game played on a chessboard using 32 game pieces. All presented chess rules are based on the rule set of the International Chess Federation (FIDE)[7]. A chess game has three possible outcomes: victory, defeat and draw. It can be won by checkmating the opponent's king. The king is in checkmate when it is attacked by an opponent's piece and cannot, in the next turn of the player, be moved out of the attack. The players take turns alternately, and the player with the white pieces starts the game. The game is played on a chessboard which consists of 32 bright (often white) and 32 dark (often black or brown) fields. An image of a chessboard is shown in figure 2.1. The board is squared, and the fields are captioned from A to H horizontally from left to right from the white player's perspective and vertically 1 to 8 beginning at the white player's site going to the black player's site. The name of one field is then its letter appended by its number, such as A3, F8 or C5. In figure 2.2 all fields are labeled with its names. Because of the 64 fields and 32 game pieces, chess has a very high complexity. After 40 moves, there are 10^{120} possible ways a game could go (including legal and illegal moves)[8].

2.1.2. Pieces

The game of chess uses 32 pieces, 16 for every player. There are six different kinds of pieces:

- King (one per player)
- Queen (one per player)

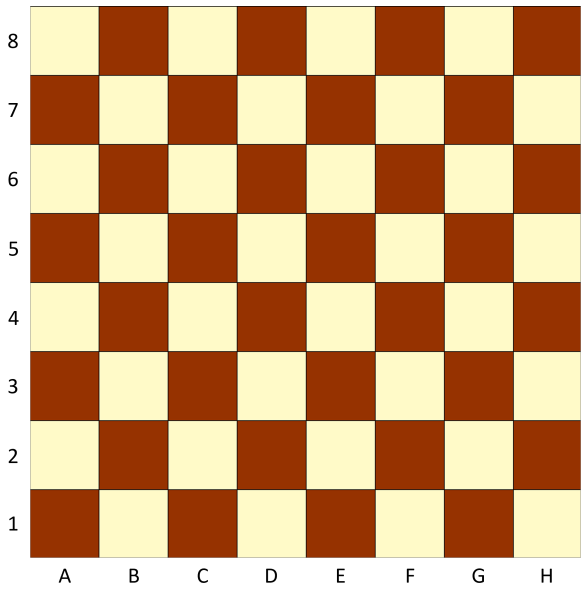


Figure 2.1: Empty chessboard

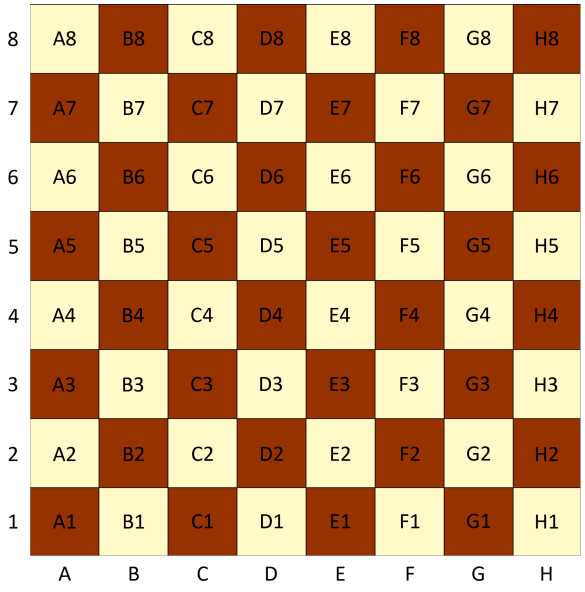


Figure 2.2: Empty chessboard labeled with all field's names

- Rook (two per player)

- Bishop (two per player)
- Knight (two per player)
- Pawn (eight per player)

At the start of the game, these are positioned as shown in figure 2.3.

Figure 2.3: Start formation of chess pieces

All pieces can move differently and except knights, no pieces can move through another one, knights on the other hand can jump over another piece to get to the wanted position. All pieces can only move to an unoccupied field or take the field of an opposing piece by capturing it. It is furthermore not allowed to move one's pieces in a way which would result in one's king to be attacked, thus one has to always try and protect the king. The moving patterns of all pieces are described in the following sections.

King

Because the objective of chess is to attack the opponent's king, it is important to protect one's own king. The king can hereby move one field in any direction per turn.

Figure 2.4: King

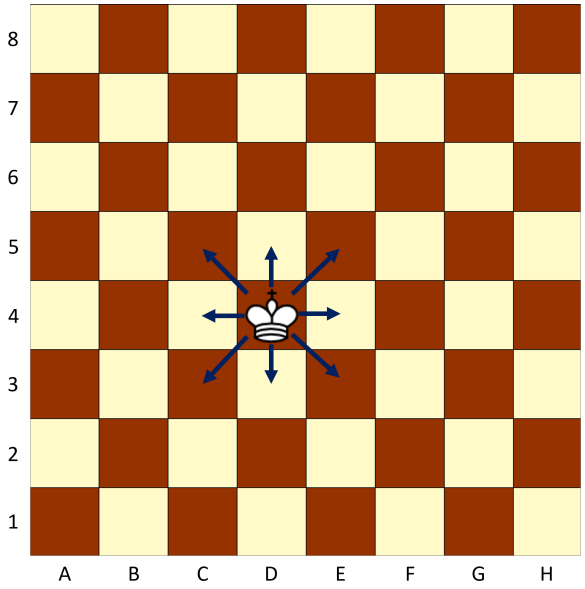
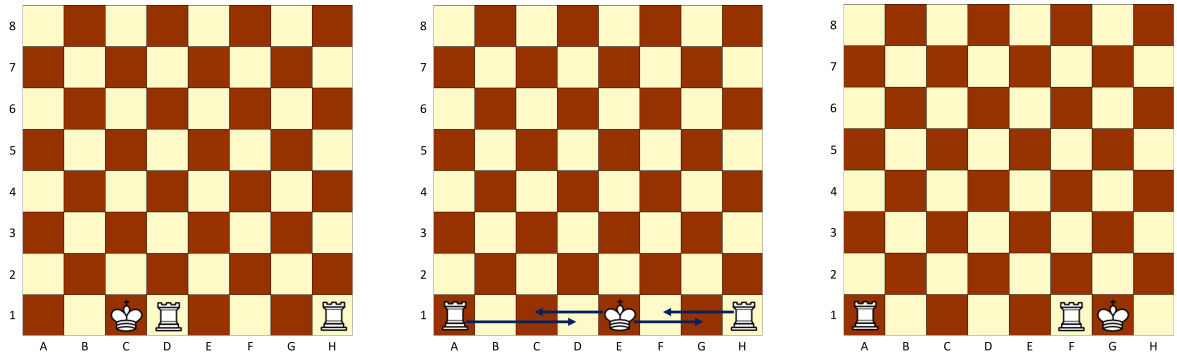


Figure 2.5: Possible king moves

This results in the possible moves demonstrated in figure 2.5. The king can only move to a field that is not under attack, so that it is not moving into check. This movement pattern is the base for every king move. There is only one special move a king can performed, castling (figure 2.6). Castling not only moves the king, but at the same time one rook. This move can only be performed once per game and player, and only if the king and involved rook have not already been moved this game. Furthermore, there cannot be any pieces between the king and rook and the king cannot be in check or move through or into a field that is attacked. Because every player has two rooks, there are two different ways of castling. Castling to the side where three empty fields are between rook and king is called "queenside castling" and to the other side "kingside castling". Independent of the side the king is castling to, the king always moves two fields in this direction and the rook moves over the king next to it. Thus, the rook is to the right of the king after queenside castling and to the left after kingside castling.



(a) Queenside (b) Initial position (c) Kingside

Figure 2.6: Castling

Queen

Figure 2.7: Queen

The queen is the piece with the widest variety of possible moves. She can move in the same directions as the king, but not just one field per turn, but as many as she wants. This makes her the strongest piece (excluding the king) of a player's pieces. The queen's moves are visualized in figure 2.9.

Rook

Figure 2.8: Rook

A rook is also allowed to move as many fields per turn as it wants. But contrary to the queen, the rook can only move horizontally and vertically. Its allowed moves are therefore a subset of the queen's possible moves (figure 2.10).

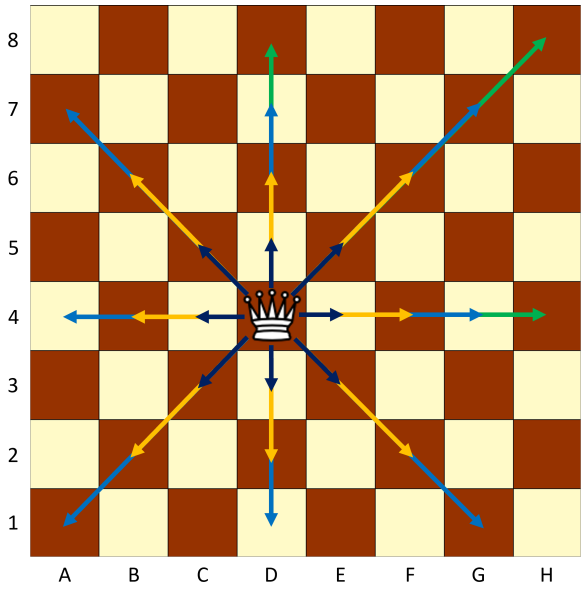


Figure 2.9: Possible queen moves

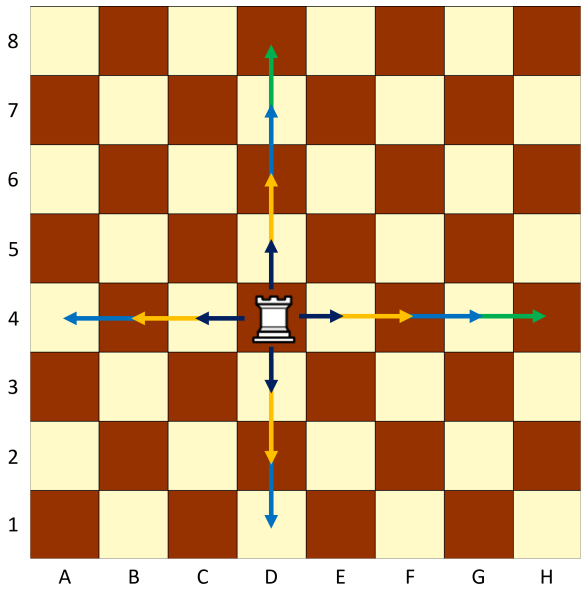
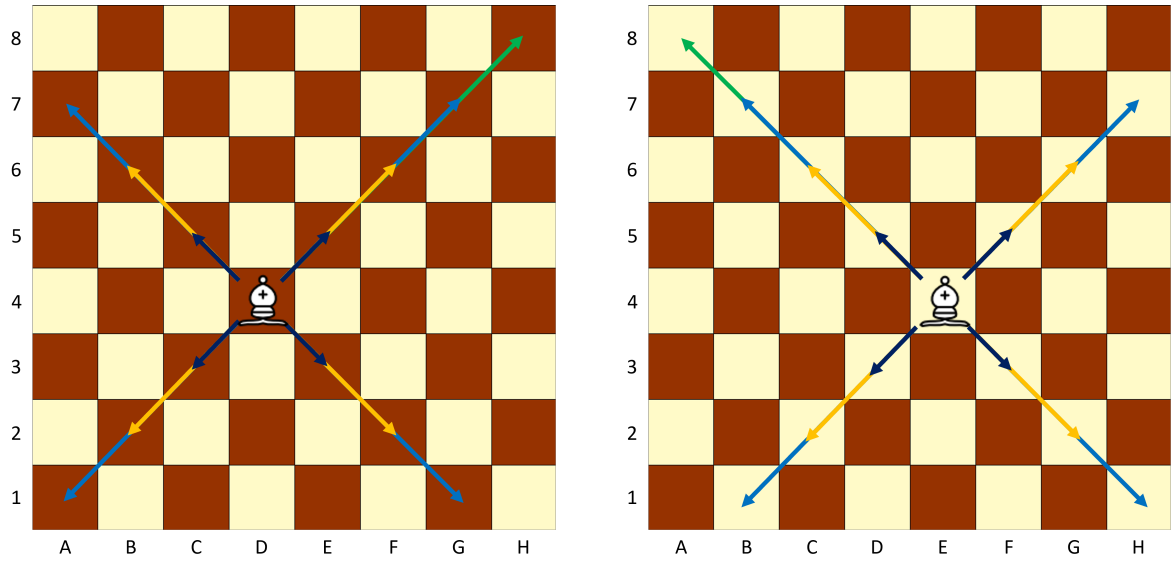


Figure 2.10: Possible rook moves

Bishop

Figure 2.11: Bishop

The bishops are allowed to move only diagonally across the chessboard. They can go as many fields in one move as wanted, but only in one of the four diagonals starting from their position. As can be seen in figure 2.12, this means that a bishop cannot change the color of fields it is walking on. A bishop that starts on a black field is only able to reach black fields throughout the whole game, and a bishop starting on a white field can only go to white fields. From the start formation of chess (figure 2.3) can be seen that both players have one bishop that covers black fields and one that moves along the white fields.



(a) Bishop on a black field

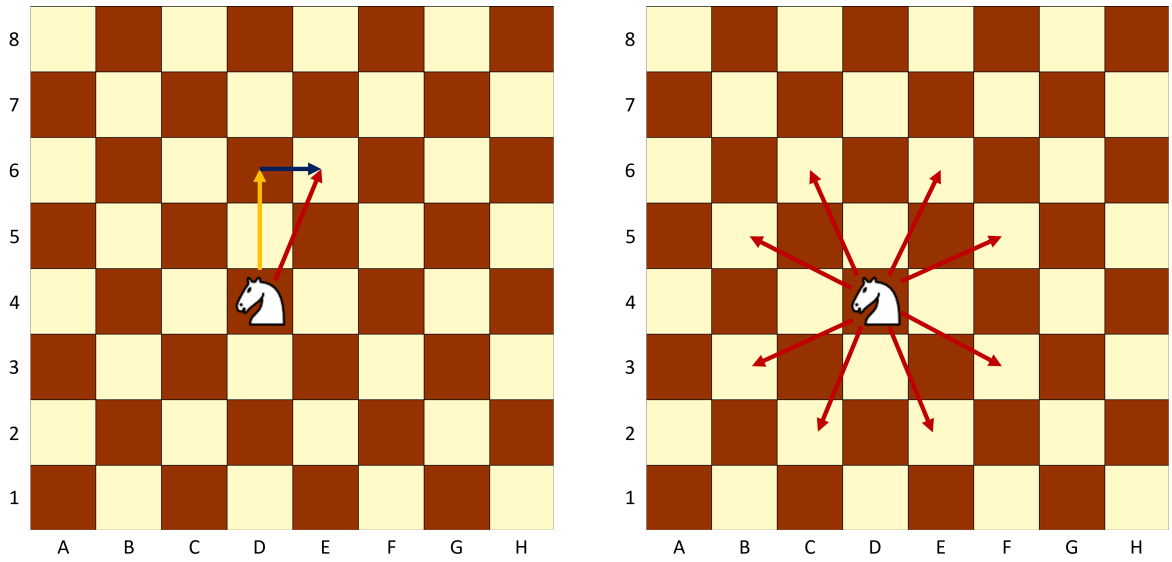
(b) Bishop on a white field

Figure 2.12: Possible bishop moves

Knight

Figure 2.13: Knight

Opposite to the other pieces, knights do not move in a straight line along one axis. Their moves consist of two orthogonal components, two fields along either a horizontal or vertical line and then one field perpendicular to that (figure 2.14a). It does not matter if any piece is standing on one of the intermediate fields, because a knight can jump over them and go straight to the intended field. Looking at all possible combinations of a knight's move, there are eight possible fields a knight could move to (assuming it is not standing at the edge of the chessboard). This is shown in figure 2.14b.



(a) Structure of a knight's move

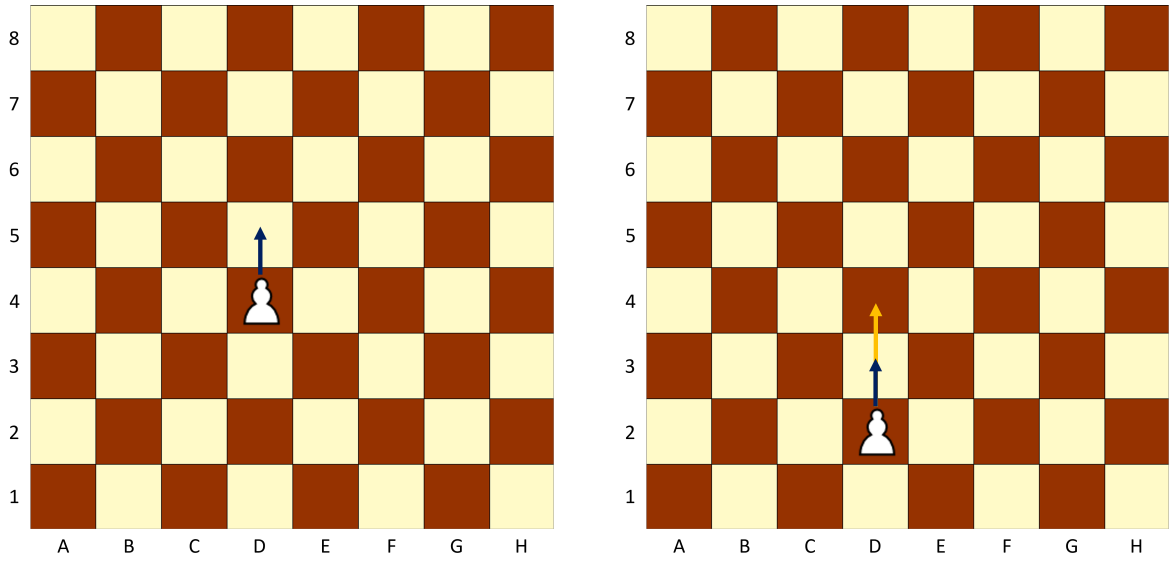
(b) Resulting possible moves

Figure 2.14: Possible knight moves

Pawn

Figure 2.15: Pawn

Pawns are the pieces with the simplest normal movement pattern, but they have a few different exceptions to this standard move. According to their standard movement pattern, they can only move one field vertically towards the side they did not start on (figure 2.16a). But this move is extended if they have not already moved this game, then they can either move one or two fields towards the other side (figure 2.16b).

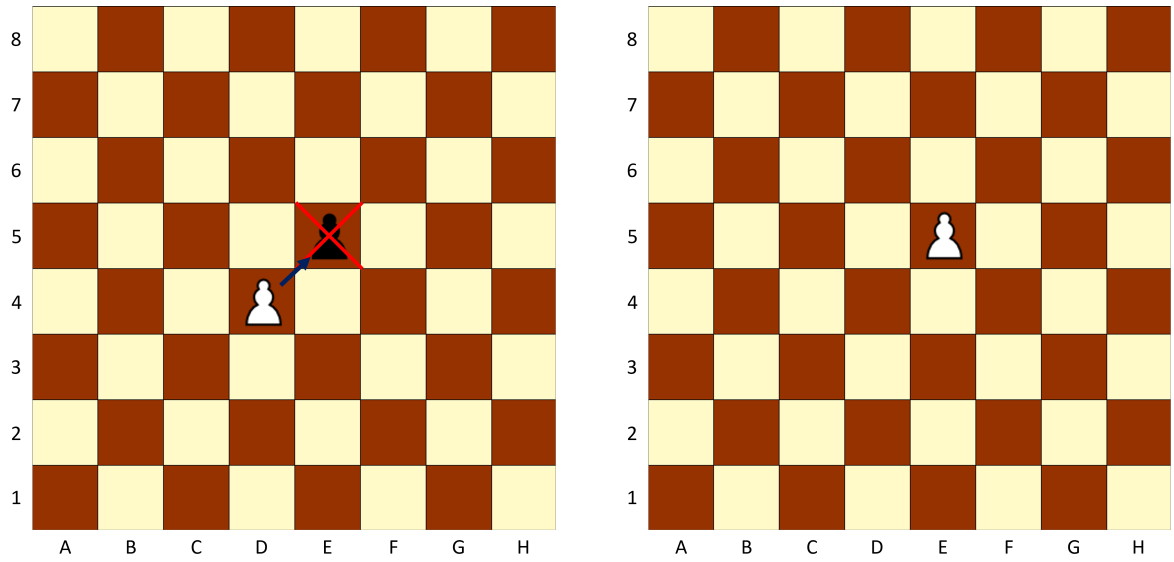


(a) Normal pawn move

(b) Starting pawn moves

Figure 2.16: Possible pawn moves

These moves are only possible, if there are no other pieces standing in front of them. Pawns cannot capture an opponent's piece by moving vertically. To be able to capture an opponent's game piece, this has to be positioned diagonally one field away of the pawn in the direction of its movement. One of the possible positions in which a pawn can capture another one is shown in figure 2.17a. The pawn then moves diagonally onto the field the other figure was standing on, and by doing so captures the opponent's

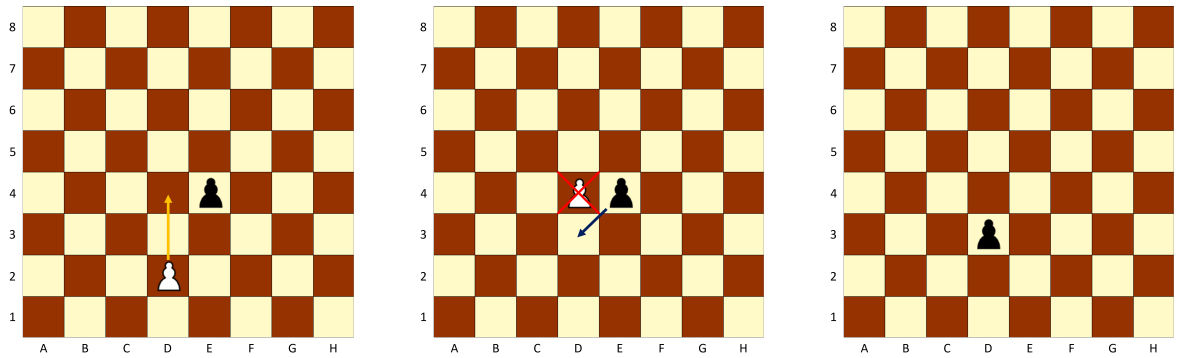


(a) Initial position

(b) Capturing move

Figure 2.17: Pawn capturing move

piece (figure 2.17b). This capturing move cannot be done with the two field move from the starting position. But there is an exception rule regarding capturing and the two field move called "En Passant". When a pawn is in a position where, if an opponent's pawn would do a standard move and go one field forward, the pawn could capture the opponent's pawn, the requirement for an "En Passant" is given. If the opponent's pawn moves one field, the pawn can capture it. But if the opponent's pawn has not moved yet and decides to move two fields forward, the pawn would not be able to capture the opponent's pawn. In this situation, the "En Passant" makes it possible for the pawn to capture the opponent's pawn (only) in the next move, although it is standing next to it. The pawn makes a normal capturing move it would do if the pawn had only moved one field, and by doing so the opponent's pawn gets captured even though it is not standing on this field. A capture executed according to the "En Passant" rule can be seen in figure 2.18.
Pawns are considered as the chess pieces with the lowest value[9] and thus being the worst of all five pieces (excluding the king). But it is possible for pawns to promote to better pieces in a game. To do so, a pawn has to reach the opposite side of the chessboard. When reaching the other side, they are promoted to another piece (except a pawn or king). Because the queen is the strongest piece in chess[9] and can do all moves other pieces can do (except for knights), a pawn is usually promoted to a queen.



(a) White pawn moves two fields

(b) Black pawn pretends white moved only one field

(c) Black moves normally and captures white pawn

Figure 2.18: En Passant

If a player decides to instead promote a pawn to a rook, bishop or knight, this is called an underpromotion.

2.1.3. Draw

A game of chess can have three different outcomes: win, draw, loss. The condition for a win/loss are described in section 2.1.1. But there are different ways a draw can be achieved. If the player whose turn it is cannot do any move that is legal, and he is not in check, the game ends in a draw. This is called "stalemate" (article 5.1 a). Another way a game can end in a draw is the so called "dead position" (article 9.6). This occurs when no player can achieve a checkmate in any number of moves. The simplest form a game is drawn is if both players agree upon a draw, which can be done at all times during the game (article 9.1). Furthermore, a player can claim a draw if a (identical) position has occurred at least three times (article 9.2). Additionally, if both players have made at least 50 moves in a row during which no pawn has been moved or piece captured, a player also can claim a draw (article 9.3).

2.1.4. Opening book and endgame table

Chess opening books or rather databases (not to be confused with books about chess openings) are a collection of chess games. These chess games can be used to derive helpful information about especially opening moves and the earlier moves of a chess game. Because there are so many possible chessboard positions[8], the chances are low

that a given position is contained enough times in such a database to be able to use it for making better moves. But because every game starts from the same board position (figure 2.3), the earlier a position occurs in a game, the higher the chances many other people have had the same position before. Using this idea, these chess databases can be used to statistically evaluate a position and the possible moves in this position. As described, this is only useful for game positions occurring early in the game such as the opening moves, because the sample number is not high enough otherwise. By doing so, it can be seen what positions usually lead to a victory of what side and which moves have a higher win chance than others. To make the derived statistics more meaningful, games of better ranked players are mostly used for the databases. Such opening books therefore give insight into what opening moves are better than others and how to increase the win chances of a given (early) position. To save storage, the actual opening books contain only information about the (opening) positions such as best next move or win percentage for both players instead of saving all games used to derive these statistics. Storing these in a binary format (such as PolyGlot) can further improve the storage efficiency.
Endgame tables on the other hand are created differently. Instead of collecting played endgames, a position can be fully analyzed if there are not too many pieces left. It is possible, to calculate all possible moves and positions from a current position if there are seven or fewer pieces (including both kings) left in the game[10]. For positions with eight pieces left, only a small subset of positions has been computed and analyzed yet[11]. To not having to recompute a position every time, the results (including winner and how many moves are left to victory assuming perfect game-play) are saved in the endgame tables. If a chess game gets to a point where only seven pieces are left, these endgame tables can be used to finish the game in the best possible way. It has to be noted though, that the more pieces an endgame table supports the bigger the endgame table gets. For three to five pieces, the Syzygy endgame table has a size of 939 Megabyte[12]. A six piece Syzygy endgame table is 149.2 Gigabyte big and for seven pieces the Syzygy endgame table has a size of 16.7 Terabyte.

2.2. Mathematics

2.2.1. Kullback-Leibler (KL) divergence

The Kullback-Leibler (KL) divergence is a measurement of divergence between two probability distributions/populations. It can be seen as a measurement of how hard it is to discriminate between them[13]. Meaning, it gives information about how likely it is that one assumes that it was sampled from one distribution, but it was actually sampled from the other one. The KL divergence can be viewed as a distance between the two distributions. Mathematically speaking this is not correct, it has all properties of a distance except it is not symmetric (the same two populations in the reverse order do not have the same KL divergence) and it does not satisfy the triangle inequality property of a distance[14]. But for the understanding and use-case in this project it can be assumed as the difference or distance between two probability distributions. The KL divergence is defined as follows (using the notation of [14]):
Given two probability distributions F_1 and F_2 (with the same sample space), the KL divergence between them is

$$I(F_1 : F_2) = \int_{-\infty}^{\infty} f_1(x) * \log(\frac{f_1(x)}{f_2(x)}) d\lambda(x)$$

If F_1 and F_2 are discrete, this equation can be simplified to

$$I(F_1 : F_2) = \sum_x f_1(x) * \log(\frac{f_1(x)}{f_2(x)})$$

2.2.2. Entropy

The entropy, sometimes called Shannon entropy, is a measurement of the average information stored in a probability distribution and is always positive[15]. If the distribution has a probability of 100 % for one event and 0 % for the other events, the distribution does not contain much information and the entropy is therefore 0. The entropy can thus be understood as the amount of uncertainty or surprise present in a probability distribution. The more events a distribution has and the more equally likely they are, the higher the entropy of the distribution. The entropy can be used to calculate the number of bits required to represent such a distribution. The Shannon entropy is only defined for discrete distribution:

Given a discrete probability distribution P, the entropy of that distribution is

$$H = -\sum_{x} p(x) * \log(p(x)) = \sum_{x} p(x) * \log(\frac{1}{p(x)})$$

For continuous distributions, this formula cannot simply be changed to an integral instead of the sum, as with the KL divergence[16]. Instead, an invariant measure function $m(x)$ is needed to define the entropy for a continuous distribution:
Given a continuous probability distribution P, the entropy of that distribution is

$$H = -\int_{-\infty}^{\infty} p(x) * \log(\frac{p(x)}{m(x)})dx = \int_{-\infty}^{\infty} p(x) * \log(\frac{m(x)}{p(x)})dx$$

2.3. Reinforcement learning

2.3.1. Development

Reinforcement learning is a subset of Machine Learning[17]. It is one of the three main training approaches used in Machine Learning: supervised, unsupervised and reinforcement learning. In contrast to supervised and unsupervised learning, reinforcement learning is not trained on a set of data. Instead, the setup of reinforcement learning is as shown in figure 2.19.

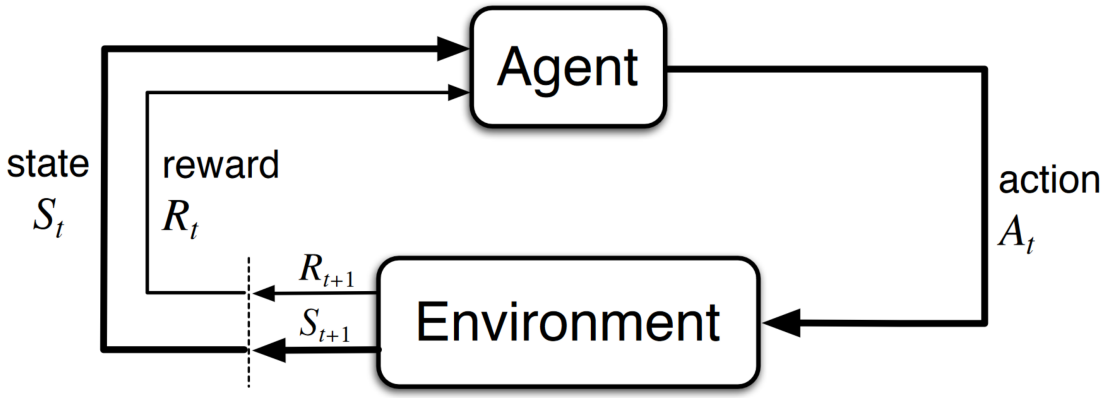


Figure 2.19: Structure of an reinforcement learning task[17]

The model, in reinforcement learning usually called agent, is being trained in an environment to perform a specific task or achieve a predefined goal. This environment might be the real world, a simulation or a specific game for instance. The agent can observe the environment (by getting the current state of the environment) and is able to perform an action. The environment reacts to this action with a new state and gives the agent a reward depending on if the performed action got the agent closer to its goal or has been counterproductive. The agent tries to maximize the return (the sum of future rewards) because this indicates it is achieving its goal. To remember what actions are good and lead to a bigger reward, it keeps a policy. The training is therefore the process of trial and error to figure out which actions are best in a given state of the environment. For this, the agent has to balance exploration and exploitation. It has to explore to find the actions that work best, but also exploit the actions it already learned to be good. There are multiple ways to represent the policy, a tabular approach and function approximation. The older and more simple approach is to use a table to

represent the policy. The table has an entry for every state and action possible in this state. An example is shown in table 2.1.

	action 1	action 2	action 3	action 4	action 5	action 6	action 7
state 1	6.1	-7	-1.1	0	2.8	-4.9	-0.4
state 2	-4.1	2.1	2.3	20.5	-17	23.7	12.7
state 3	-1.3	7.6	30.8	-12	0.5	-5	-8.2

Table 2.1: Example policy with a tabular approach

In every entry, the expected return for taking this action in the state is kept. This way, when choosing an action that is best, the agent can simply look up which action has the highest expected return in the given state. To train and learn, the agent simply performs many actions in the environment and then updates these values in the table to get an as accurate representation as possible. This has the problem that it only works for actions that are discrete and the more actions and state there are the bigger the table becomes. It also means that every state-action combination has to be visited at least once to have any information about it. This approach therefore does not allow for generalized learning where similar states might have a similar behavior. To address these disadvantages, the policy can instead be represented as a function or function approximation. With this approach, a function approximation is used to represent the expected returns in a state for an action. The function takes the state and action and outputs the expected return. The training with that is similar to the tabular approach, the agent performs as many actions in the environment as possible and the function approximation is updated to get as close to the observed returns as possible.

2.3.2. Deep reinforcement learning

As described in the last section, reinforcement learning methods can use functions to approximate the expected return for a given state and action. A simple function can only abstract a simple model, for more complicated environments a more complex function is needed. To simulate these complex functions, machine learning methods such as neural networks can be used. This is called deep reinforcement learning (often times, the used neural nets are deep neural networks)[18]. Most modern projects using reinforcement learning are using deep reinforcement learning methods. This is because these are better fitted for more complicated environments and tasks. These deep reinforcement learning methods are more powerful than the traditional reinforcement learning approaches and can solve problems that were not solvable by a machine before[19]. One family of deep reinforcement learning methods are policy gradient

methods. These do not use the neural network to predict the return for an action, but directly predict the probabilities for the actions for a given state. These probabilities give information about which action is the best (or presumed to be the best by the policy) for this state. The algorithm then directly updates the policy (neural network) according to the returns.

2.3.3. Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a widely used algorithm of the policy gradient family[20]. It has been shown to perform better than alternative approaches for some applications[2]. It is an online policy method, meaning it samples the episodes for training using its current policy and does not have a second policy just for sampling. A benefit of PPO over comparable methods is its simplicity regarding the implementation, it is more general and has a better sample complexity. PPO uses methods of the Trust Region Policy Optimization (TRPO) and improves these to achieve the mentioned benefits. TRPO is an online policy method and reuses samples generated with an old policy. Because the policy has been changed since the sampling of these old samples, the informative value of these is lower than new samples. But assuming the policy change has not been too big, the old samples still hold information that the agent can learn from. It therefore uses a trust region which defines how much the information gain from an old sample can be trusted, and thus to make sure that the changes are smaller the more different the policy which was used for sampling the old sample is from the current policy. It achieves this by solving this equation:

$$\max_{\theta} \mathbb{E}[\frac{\pi_{\theta}(a|s)}{\pi_{\theta_{old}}(a|s)} * A]$$

And to make sure it is kept in the trust region, the equation is maximized subject to the constraint:

$$\mathbb{E}[I(\pi_{\theta_{old}}(*|s) : \pi_{\theta}(*|s))] \leq \delta$$

The used symbols are:

π Policy

a Action

s State

θ Weights of the neural network defining the policy

θ_{old} Old/not yet updated weights of the neural network defining the policy

A Advantage function

I(x:y) KL divergence between x and y as described in section 2.2.1

***** For all actions

δ Parameter

The first equation is maximized by changing the weights of the neural network defining the policy, but only as much that the KL divergence of the new and old policy is less or equal than a parameter δ. This constraint ensures that the update of the weights is not too big (the actual size is controlled by the size of δ). Instead of using the constraint, PPO proposes two alternative methods. The first adds the KL divergence onto the function (that should be maximized) as an error. This changes the maximization function to:

$$\max_{\theta} \mathbb{E}[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} * A - \beta * I(\pi_{\theta_{old}}(*|s) : \pi_\theta(*|s))]$$

The parameter β can either be fixed or adapted automatically throughout the training. The second proposed approach is clipping the function. This leads to the following function to be maximized:

$$\max_{\theta} \mathbb{E}[min(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} * A, clip(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, 1-\epsilon, 1+\epsilon) * A)]$$

The parameter ϵ is the clipping parameter, defining the size of the trust region and therefore the size of the policy update. It is also possible to combine both approaches and maximize the combined function. By maximizing one of these functions, the values of the policy (neural network) are changed to become closer to the observed epochs. Instead of using only one epoch for the policy update, PPO uses a batch of epochs to reduce the variance in the epochs and thus the policy updates.

2.3.4. Invalid action masking

In complex environments, different states can have a varying set of valid actions. To build an agent, the action space has to be defined. A common approach is, to define the action space as the sum of all possible actions over all possible states[21]. If the agent now samples an action for a given state, it cannot be guaranteed that this action is valid for that state. Especially the more complex a game becomes, the more of the actions are invalid for a state and therefore the chances to pick an invalid action

become bigger. It would be possible to give a high negative reward for invalid actions so that the agent learns to avoid these actions, but that can have negative effects on these actions for states where they are valid. It will also prolong the training process, because the agent has to learn more information about its task before it can focus on mastering it. Depending on the environment, it might not be possible to do the invalid action or might even cause harm to the environment or agent. For policy gradient methods, it has become the standard approach to use invalid action masking. It has been shown to be effective and improve the training process in environments with different sets of valid actions for different states[21]. Invalid action masking uses a mask that indicates which actions are valid for a given state (usually by setting the index of a valid action to 1 and 0 for invalid actions). This mask is then used to set the output of the policy to a very high negative value for invalid actions, so that after converting the outputs to the probabilities for the actions, the probability of an invalid action is 0. And when the probability of an invalid action is 0, it will never be picked by the agent and thus solving the problem. This does not interfere with the training and updating the policy.

2.3.5. Multi-agent and self-play

Some environments require multiple agents, such as games with multiple players. The agent that is being trained in this environment needs the other participants in order for it to learn the wanted behavior. In the example of a board game, if there is no other player, the game cannot be played and thus the agent not trained. These environments with multiple actors require a multi-agent setup. This means that there is more than one agent to act. These agents can, like with a round-based board game, act alternately or, like in a boxing game, simultaneously. These other agents can be controlled by the environment or have their own set of rules or policy according to which they act. This makes it possible, to choose the agent being trained to also act as its opponent or the other actors. Especially in competitive environments where the goal is to win, letting the agent train against itself (the so-called self-play) can have benefits. The agent wants to win and tries to use weaknesses of its opponent to defeat it. When it finds a weakness, it will use this to win more often. But because the opponent is itself, its opponent will use the same strategy against it and uses its weakness to win. The agent is therefore forced to eliminate this weakness. This process of finding weaknesses and then having to eliminate those, improves the agent's performance drastically. By playing against itself, the opponent has always the perfect strength for the agent to optimally improve itself[22]. In a scenario where the agent plays against a fixed actor, once the agent becomes better than its opponent it cannot learn much from it anymore and its strength is thus bound by the strength of its opponent. This does not occur

with self-play, because when the agent improves, the opponent also becomes stronger. The strength the agent can achieve is therefore (theoretically) unbound. Using self-play can also have negative effects on the training, especially in the beginning, when the agent still acts close to randomly. In this case, the opponent acts also randomly and therefore the agent cannot learn that much from its opponent in comparison to using an actor that understands the environment and knows how to achieve the goal at hand. In addition, it might occur that the agent gets stuck in a local minimum and needs a lot of training and time to overcome this local minimum.

2.4. Chess programs

2.4.1. Traditional chess programs

The way traditional chess programs work can be split into two parts. First, given the current board position, they find all possible moves that can be executed in this position. They then evaluate all these moves by following the tree, created by the moves, downward to a certain depth and evaluate the positions that these moves will lead to. To efficiently compute the positions a move will lead to, they use alpha-beta search algorithms to reduce the number of paths in the tree by removing impossible or unrealistic paths. The evaluation of the new positions is done by handcrafted rules created by or in corporation with chess Grand-masters. Improvements in these programs are often achieved by either making it more efficient and therefore being able to follow the tree to a bigger depth, or by improving the used evaluation rules. To improve the evaluation rules, a high understanding of chess is necessary and the maximum level of these evaluation rules is bound by the developer's knowledge of chess. This will make progress harder the better these rules become. Two of the strongest traditional chess programs are Stockfish[23] and Komodo Dragon[24]. Stockfish is an open-source project and is publicly available to download for free. Komodo Dragon on the other hand has to be bought, and only older versions are available for free. Stockfish is considered as the strongest chess program, it has won the last consecutive 6 seasons of the Top Chess Engine Championship (TCEC)[25]. In newer versions, even the traditional chess programs make use of machine learning. Stockfish for example takes advantage of neural networks for the evaluation of the positions instead of relying completely on handcrafted rules anymore.

2.4.2. Reinforcement learning in chess

In contrast to traditional chess programs, reinforcement-based programs do not rely on human chess knowledge. The developers only give the program the set of rules of the game (chess) and then use self-play to let the program learn the best strategies on its own. This way, the maximum strength is not limited by the developer's knowledge about chess, but (if not restricted by the algorithm architecture) unbounded. The first program that made people aware that reinforcement learning can be used for creating a very strong chess program is AlphaZero[1]. It defeated the (at this time) current version of Stockfish without losing once, indicating that it was better than Stockfish (there is criticism about the evaluation games though, especially the settings used for Stockfish were far from optimal). AlphaZero has been trained for 700000 steps on

chess using 5000 Tensor Processing Units (TPUs) of the first generation and 64 second generation TPUs. This amounts to an enormous computation power, far beyond what a single person or "normal" organizations can provide. AlphaZero has not been trained further and is therefore not a competition for the current chess programs. But after the publication of AlphaZero, Leela Chess Zero was started using the architecture of AlphaZero as a base[6]. It also uses reinforcement learning with self-play to train the program, but because it is still under development and being improved, it combines the findings from AlphaZero with newer approaches such as attention policies and transformer-like models. In the TCEC League Season 23, Leela Chess Zero entered the Superfinal and lost against Stockfish[25]. In the World Computer Chess Championship (WCCC) 2022, Leela Chess Zero also entered the finals to lose against Komodo Dragon. During the finals, both the programs could only get draws, so an Armageddon game was held where, in case of a draw, black wins. Komodo Dragon won the coin-toss and chose black, and after the game ended in a draw was declared the winner of the tournament. Hence, Komodo Dragon and Leela Chess Zero have been equally strong in this tournament. Stockfish does not participate in the WCCC. It has therefore been shown, that reinforcement learning based chess programs can match the strength of traditional chess programs.

3. Setup

3.1. Environment

To be able to train any reinforcement agent, it needs an environment to be trained in. To optimize the training process, the parameters of the environment should be manually controllable, and it should be possible to make all needed adjustments for the training process. Thus, an own chess environment has been implemented. To not having to implement all chess rules, the library "python-chess"[26] has been used for the game mechanics. The by "python-chess" provided Application Programming Interface (API) is not compatible with most if not all Python reinforcement learning libraries, and especially not with the chosen library "RLlib"[27]. Hence, a wrapper needs to be built around the chess API provided by "python-chess" so that this environment can be used with the algorithms provided by "RLlib". To do so, the environment uses the Gym API.

3.1.1. Gym interface

The Gym library[28] provides a number of reinforcement learning environments that are supported by numerous reinforcement libraries (which provide the algorithms). This caused the Gym API to become the standard for environments, because then an environment could easily be used with most reinforcement libraries. Since 2022, the development of the Gym API has been moved to the Gymnasium library of the Farama Foundation, to be organized by a nonprofit organization[29]. To make the environment usable with as many reinforcement libraries as possible, it should have the Gym API. Figure 3.1 shows the key interfaces the environment has to have to comply with the Gym API. The environment has to offer two main functions, "step" and "reset". "reset" resets the environment and returns the current state of the environment. This is used during the training to sample a new training sample from the same environment. The second method "step" performs an action in the environment and returns the

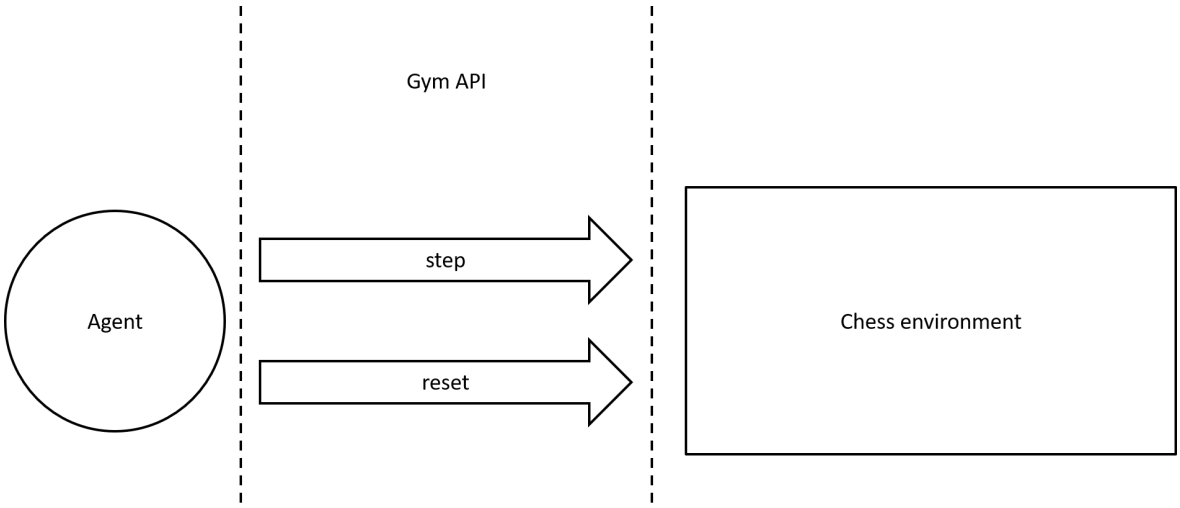


Figure 3.1: Gym API

results of that step. Because the environment is a multi-agent environment, it takes up to one action per agent in the environment. After executing the action(s), it returns the following information:

- New state of the environment
- Rewards received per agent
- Information which agent is done or if the game is over
- Additional information if needed

This information is used for training the agent and computing the next actions. Additional interfaces implemented in the chess environment, but not required by the Gym API, are a "run" and "evaluate" method. The "run" method can be used to run a complete chess game using the environment. "evaluate" evaluates a trained agent in the environment using the method described in section 3.2.3.

3.1.2. Graphical User Interface (GUI)

The implemented wrapper around "python-chess" can be used to train agents using most reinforcement libraries, and also to play a chess game against an agent using the terminal. This is not ideal, to better test the agents and enhance the usability of the environment, it needs a Graphical User Interface (GUI). Therefore, a GUI is added on

top of the environment using the "pygame" library[30]. This GUI should be able to visualize the chess board with its pieces and this way show which moves are being made and how the position looks. But it should furthermore enable the user to play against an agent using this GUI, thus it needs to support user input such as selection of a piece and move. The created GUI can be seen in figure 3.2. A move can be performed by selecting a piece on the board and then selecting the cell to move to. As assistance, the fields a selected piece can move to are highlighted (yellow in the image). This way, an easy and intuitive interface is created to play against a (trained) agent or follow a game played by two agents against each other.

3.1.3. Observation space

As described in section 2.3.1, an environment has an observation space, defining all possible states that can be assumed by the environment. The chosen observation space is defined as an array of length 69. The first 64 entries represent the 64 fields of the chess board. The next entry saves the number of moves played since the beginning of the game, and the one after that the number of half-moves since the last pawn move or capture. The following two keep information about the castling rights of the two players. And the last gives information about if the player whose turn is next can claim a draw by repetition (three times the same board position). For every of the first 64 entries, a value is stored that indicates if a piece is on that field and if so, which one. The values representing the pieces depend on which player's turn is next and can be seen in table 3.1. The first 64 fields give the agent all the information a human would also get by looking at the board. Because the state does not contain the history of the already played moves in this game, the last five entries are necessary. The environment is configured to automatically declare a game a draw if 400 moves are played. Thus, because of the 65th entry, the agent knows how many moves it has left till that happens. The next entry (the half-moves since the last pawn move or capture) gives the agent the information it needs to know if it can claim a draw by the 50-move rule or how many moves it has left till the opponent can claim it (to maybe avoid it). The next two entries give the castling rights. In the first entry for the player whose turn is next and then for the opponent. From the board position cannot necessarily be decided if castling is possible. Because for example the king might have been moved one field and then went back later, but then it is not clear if castling is still allowed from only the board position (in this case it would not be). The castling information is represented as shown in table 3.2. With this information, the agent can decide if castling is possible for it or its opponent. The last entry indicates, if the rule of draw by repetition can be claimed, this can also not be seen purely from the board position. It is represented as a 0 for not possible

Figure 3.2: Image of the GUI

and 1 if it is possible to claim the draw. That leads to in theory $13^{64} * 203 * 77 * 5 * 5 * 2 =$ 153225594504461316861577135759038019354942383487216693790644817790909675469550 possible states, but not all of them can actually occur in a game of chess. This observation space was used for the first version of the agent. For the second version, a

Value	Corresponding piece white's turn	Corresponding piece black's turn
0	Empty/No piece	Empty/No piece
1	White pawn	Black pawn
2	White knight	Black knight
3	White bishop	Black bishop
4	White rook	Black rook
5	White queen	Black queen
6	White king	Black king
7	Black pawn	White pawn
8	Black knight	White knight
9	Black bishop	White bishop
10	Black rook	White rook
11	Black queen	White queen
12	Black king	White king

Table 3.1: Piece value meaning in observation space

Value	Meaning
0	No castling allowed
1	Only kingside castling is allowed
2	Only queenside castling is allowed
3	Castling in both directions is allowed

Table 3.2: Castling rights representation

draw is automatically claimed by the environment whenever possible. Because of that, the observation space could be reduced by removing unnecessary parameters. The new observation space only includes the first 64 entries for the pieces and the two entries for the castling rights. This reduces the number of possible states to $13^{64} * 5 * 5 = 490133691076902683326649401058915038561008200010289468973977409605622402 5$.
Again, not all states can actually occur. But this reduces the observation space with the factor 31262 which might lead to a better and faster training performance.
Because the same agent should be able to play both sides, the state has to be normalized according to the player whose turn it is. This is done by turning the board such that the current player is always at the bottom, and then starting with the top left corner for the fields in the state. By also changing the values of the pieces towards the current player (as shown in figure 3.1) it is achieved, that with the same position, the state looks the same disregarding which player's turn it is. Thus, the agent does not

have to learn every position for both players each.

3.1.4. Action space

The action space, the number of possible actions, is adopted from the AlphaZero paper[1]. The action is split into two parts, first choosing a field and then choosing the move to perform by the piece on that field. There are 64 fields and in total 73 possible moves (56 queen moves, 8 knight moves and 9 underpromotions), which makes it 4672 possible actions in total. To that, one more action is added, which is for claiming a draw (if possible). This representation of actions includes field and move combinations that cannot occur, but because of the usage of invalid action masking, the agent will never choose any not allowed action. To simplify the usage of an invalid action mask, all actions are one-hot-encoded internally. This means, that an action is an array of length 4673 that has exactly one "1" at the index of the action that should be represented and "0" at all other indexes. This is a less compact representation for actions than just the number/index of the action, but by doing so it is easier to add the invalid action mask to the interim results of the used neural net. The standard PPO implementation of RLlib does not support invalid action masks. To use PPO with an invalid action mask, a wrapper for the used model/neural net of the PPO agent is needed. This wrapper needs to keep all provided methods and interfaces of the standard model, but has to accept the invalid action mask as well and use it on the interim results. For every call, this wrapper accepts the state to compute the action for and the invalid action mask and then applies the invalid action mask to the calculated results for all actions, returning the new values for every action. The used model to calculate the values for each action is a fully-connected neural net. Its architecture after the changes of version 2 is presented in figure 3.3. The input to that neural network (the input layer) is the state. To the output of the left side (the policy) the invalid action masking is applied and then from these results the probability distribution for the actions is calculated. The right side (the value) calculates the value how good it is to be in that state by calculating the expected return of that state. In some cases, the two first dense layers are shared by both output layers, but are here two individual layers of just the same size. This neural net has 1366849 trainable parameters in total.
For version 2 of the agent, when also the observation space gets changed, the action space has to be slightly adopted as well. The additional action for claiming a draw is removed, resulting in an action space of size 4672.

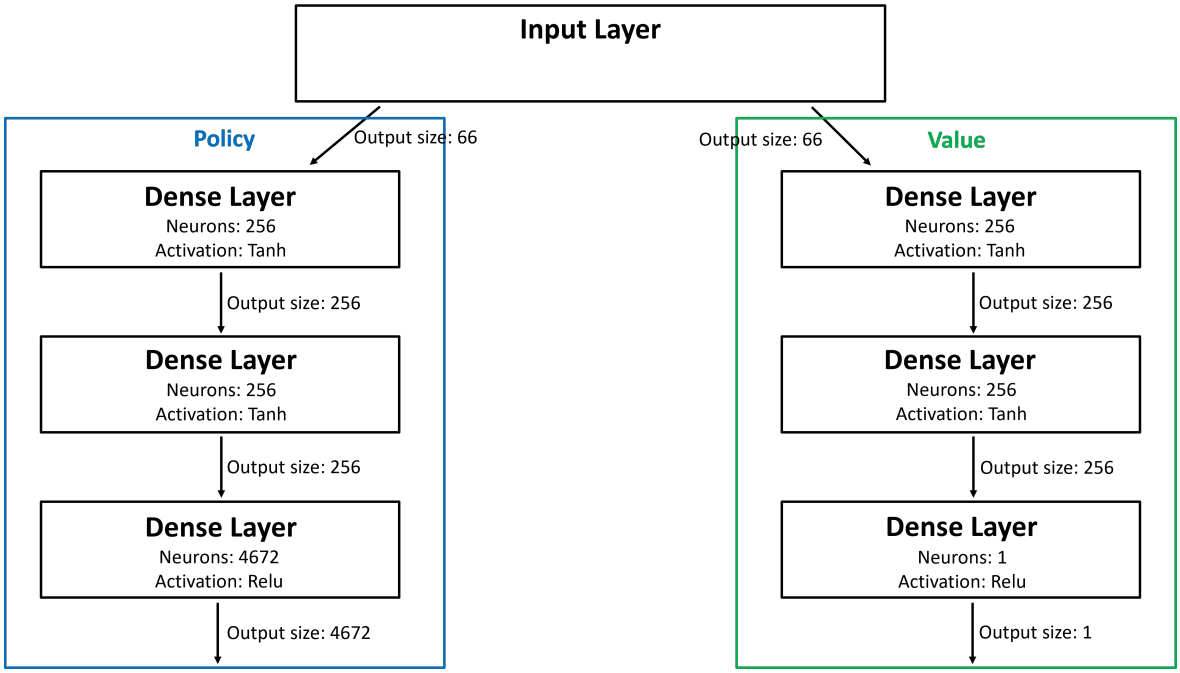


Figure 3.3: Architecture of the used neural network

3.1.5. Rewards

Another parameter of an environment that influences an agent's training performance is the reward. The environment gives a reward only if the game is over. So, when a player makes the last move that leads to the end of the game, both players/agents get a reward according to the game's result. If one agent has won the game, it receives a reward of $+100$, the other agent that has lost a reward of -100. In case of a draw both, agents get rewarded a 0. On all other, not game ending, moves, both agents receive a reward of 0. This reward structure is changed through the different versions of the agents. In version 2, the reward for a win/loss is increased to $+1000/-1000$, but it remains 0 otherwise. For version 3 of the agent, the opening book and endgame table are implemented and used to give interim rewards. In addition to the $+1000/-1000$ when the game is won/lost, a reward of $+10$ is rewarded for every move that is present in the opening book or endgame table. How it gets decided when a move is in the opening book or endgame table and should be rewarded is explained in the next section. These rewards are given to an agent directly after its move and not accumulated at the end of the game, so that it can better and faster learn which moves are good. The big difference between the 1000 at the end of the game and only 10 for a good move is

chosen so that the agent still mainly focuses on winning a game (because this reward is so much higher), but also has some direct feedback so that hopefully the training performance can be improved. For every other move, a reward of 0 is still kept.

3.1.6. Opening book and endgame table

To give the agent some help, especially during the early stages of the training, an opening book and endgame table are used to evaluate the actions of the agent. The opening book chosen for that task is M11.2[31]. This book contains games of players ranked between 2600 and 2700 in the FIDE ranking. For comparison, the best player in the world, Magnus Carlsen, has a rating of 2853 as of April 2023[32]. So, this opening book contains games of some of the best players in the world. The opening book is used in a binary format to save storage space and increase speed. It can easily be read by "python-chess" and used without any needed wrappers. To give a reward for a move based on the opening book, it is checked if the move is part of the opening moves contained in the opening book. If so, the interim reward for that move is given.
As an endgame table, the Syzygy table for three to five pieces is used. A Syzygy table is chosen, because they are much more space-efficient than other endgame tables[12]. Because of the enormous sizes of the six- and seven-piece Syzygy tablebases, only the one for three to five pieces is used. The endgame table can be read by "python-chess" as well and also does not need any further wrappers. But because the endgame table does not save the best move for a position and just information about who would win and how many moves it takes to get there, it cannot be simply checked if a move is in the endgame table. Instead, it is checked if the current player can win in the current position. It is also stored what the last number of moves was to get there. If the agent can win and reduced the needed number of steps for its win, it is assumed that the move was useful and got it closer to its win and thus the move is rewarded with the interim reward. This way, both good opening moves and endgame moves are rewarded to help the agent learn a good strategy for the opening and endgame faster so that it can better focus on the more complex part, the middle game in between.

3.1.7. Stockfish integration

To test the trained agents against Stockfish, Stockfish needs to be implemented in the chess environment. The current version of Stockfish is 15.1, and it can be downloaded for a variety of operating systems[23]. It can then be imported by "python-chess" and used to compute the next move for a position. To make all agents interchangeable in the environment, a common interface for the agents is defined (see section 3.2.1

for more details). Because the Stockfish interface offered by "python-chess" does not provide this interface, a wrapper is implemented to comply with the defined interface.

3.2. Agents

3.2.1. Parameters

As mentioned in the last section, a common interface is defined for agents in the environment. This is shown in figure 3.4.

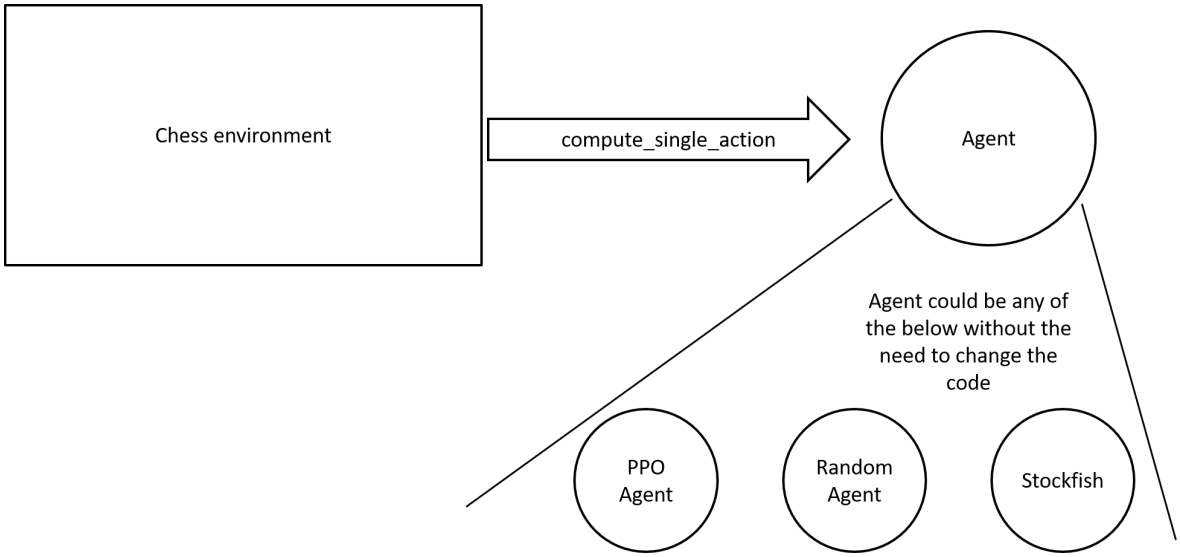


Figure 3.4: Defined interface for the agents

Every agent has to offer the method "compute_single_action" which takes the state of the environment and computes the next action and returns it. This method is chosen because it is offered by the PPO agent of RLlib, so that these agents do not need a wrapper. For the training, a random agent is needed, against which the PPO agent can play. A random agent is here defined as an agent that chooses one of the possible and allowed actions randomly. This is changed in version 4 to use self-play. To be able to use the random agent in the training, it needs the described interface and hence it is implemented that way. The interface is not only needed for the training, but also makes the agents interchangeable during the evaluation process and when they play against each other or against a human.
For the training process it is important, that the agent plays both sides (as white and as black), otherwise it would only learn to play as one side and cannot be used for the other. To ensure that, it is chosen randomly for every episode (game) if the agent that is being trained plays as white or black. It therefore has games where it learns to play the white player and other games where it improves on being the black player.

3.2.2. Training

For the training, an agent is trained for 1000 iterations. Every 100th iteration, a checkpoint is made of the agent to analyze the progress during the training. These agents are evaluated, and the results are presented in chapter 4. The hardware used for the training is as follows:

- Graphics card: Nvidia RTX 2060 with 6 Gigabytes VRAM
- Processor: Intel I7-9750H
- RAM: 16 Gigabyte DDR4

For the training, 75 % of the graphics card capacity are used, and the rest is done on the processor. The training is performed in a single process, multiple workers for the collection of training samples or improving the agent are not used. During the training, the computer was sometimes used for other computing-light tasks as well. But this was distributed equally over all versions and did not have a notable influence on the training time.

3.2.3. Evaluation

To evaluate the agents and being able to compare the different versions of it, an evaluation metric is needed. Assuming an agent starts out as a random agent when initialized, a random agent is chosen as the baseline for the evaluation. The trained agent plays against the random agent and gets points depending on the outcome of the game:

- +1 point for a victory
- 0 points when draw
- -1 point if defeated

By doing so, the final evaluation value assigned corresponds to the difference of games the agent has won and the games it has lost. Thus it can be said, that when the evaluation value is close to zero, the agent has the same strength as a random playing agent. The bigger the evaluation value, the better it is than the random agent, and the smaller the worse. To account for statistical anomalies in the evaluation process, the agent plays 1000 times against the random agent and the score from that is taken. It might also be possible, that an agent is better playing one color than the other, so 500 of the 1000 games the agent plays as white and the other 500 as black. The results

shown in chapter 4 only show the evaluation score over the whole 1000 games and not separated by color. If there are any interesting differences in the agent's strength of the two sides, it will be mentioned.

4. Results

In the following sections, the different versions of the PPO agent trained are presented, as well as the results these achieve. The agents were implemented and trained in this order and build on the ones that were trained before that. Up to version 3, the changes between the versions are changes to the environment. In version 4 the training method was adjusted and from version 5 to 10 the changes are made to the PPO agent and its (hyper-)parameters. The changes made in version 7 to 9 are based on an empirical study about on-policy reinforcement learning and what (hyper-)parameters are advisable[33].

4.1. Version 1

The first version of the agent was trained using all the settings described in chapter 3. From the evaluation results (figure 4.1) can be seen that an improvement is achieved over the training period. After the training, the agent wins 3.7 % more of the 1000 games than it loses against the random agent. Even though it performs better than the random agent in the evaluation, the margin is not big enough to clearly indicate that the agent is better. This agent can be used as the baseline for the comparison with the next versions. Furthermore, figure 4.1 shows that the agent does not improve consistently during the training. Ideally, the training progress would be more or less consistent so that there are no big changes for the worse (such as at iteration 600).

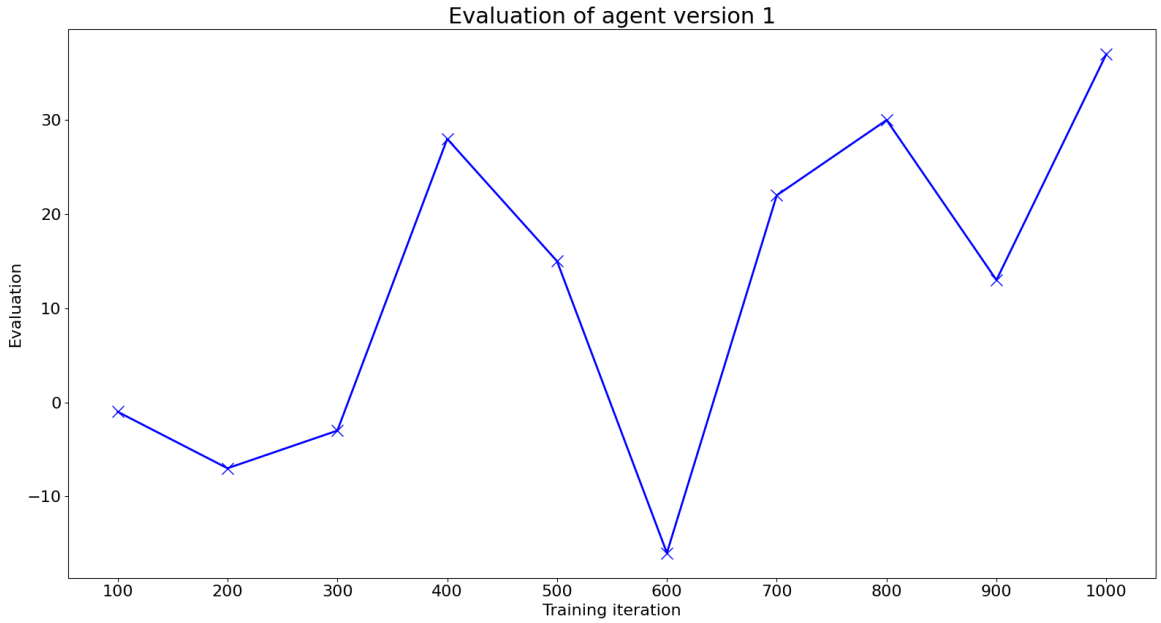


Figure 4.1: Evaluation of agent version 1

4.2. Version 2

For version 2, the reward for a win/loss is increased from 100 to 1000. When discounting the rewards, a bigger final reward should lead to a bigger influence on the earlier moves. Moreover, the observation space and action space are changed so that draws are automatically claimed whenever possible by the environment, instead of the agent having to claim it manually. Also, the automatic draw after 400 moves rule is removed from the environment. This is a rule not part of the FIDE chess rules, but necessary for the representation of the old observation space. By removing it, the environment becomes aligned with the FIDE rules. This change could cause the games sampled during the training to become longer on average. That can lead to the earlier moves having a smaller reward, because the final reward is more moves into the future. But the average length of the episodes increased only slightly from 324.2 to 326.44. This does therefore do not influence the training performance negatively. The results of version 2 are shown in figure 4.2.

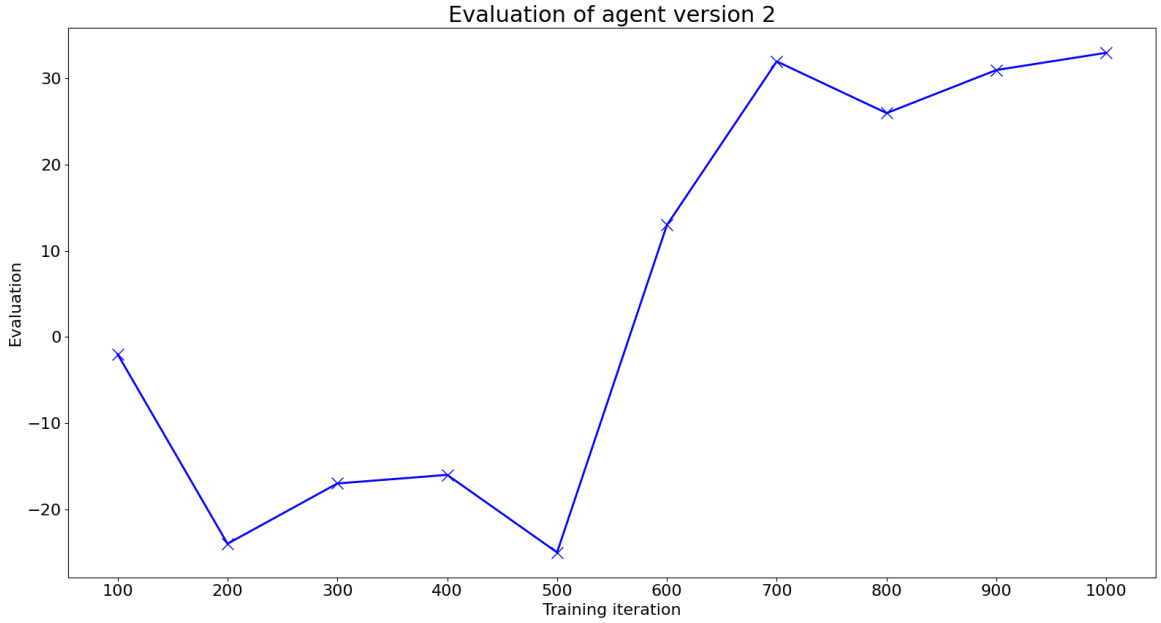


Figure 4.2: Evaluation of agent version 2

The training is more consistent than that of version 1, but the first 500 iterations it is always worse than a random agent. After iteration 500, it makes a jump in performance and almost reaches the same score as version 1 (figure 4.3). Even though the new agent takes longer than version 1 to make any progress, the changes to the environment are

kept for the next versions. They do not bring an improvement besides a more stable training, but the environment is now congruent with the FIDE rules, which is a good reason to keep the changes. Additionally, the now smaller observation space might be useful for later versions of the agent.

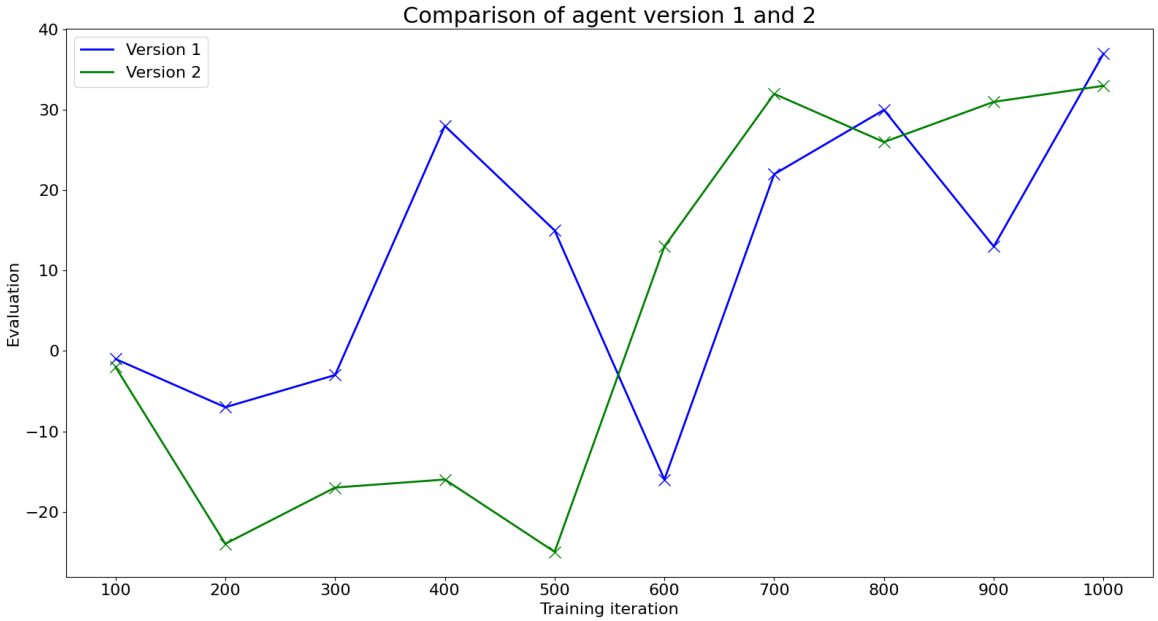


Figure 4.3: Comparison of agent version 1 and 2

4.3. Version 3

In version 3, the opening book and endgame table are introduced into the training. By using these two to reward individual useful moves (as described in section 3.1.6), the agent should be supported during the training. With these changes in place, the agent trained much better and is improving most of the time (figure 4.4).

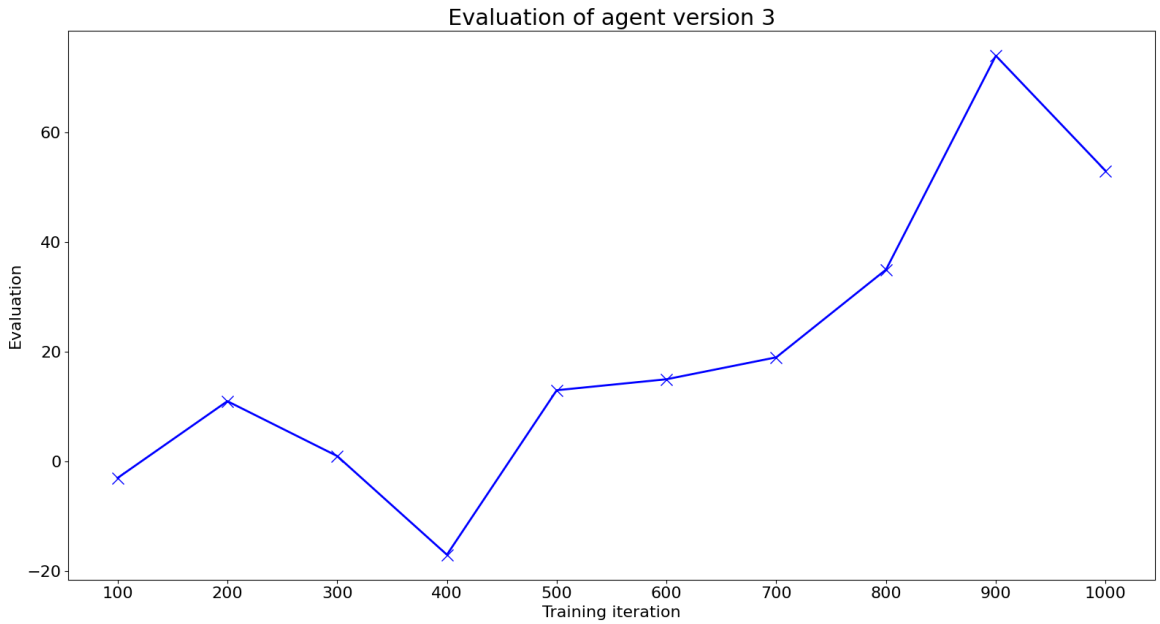


Figure 4.4: Evaluation of agent version 3

From the comparison to the other two versions (figure 4.5) can be seen, that the version 3 agent is also able to achieve a better end result. By doing so, version 3 becomes the new best version and is the baseline for new adaptions of the agent. After 900 iterations, version 3 wins 7.4 % more games than it loses, it can therefore be concluded that this checkpoint of version 3 is better than the random agent. This shows, that the usage of interim results can significantly improve the training performance of a PPO agent.

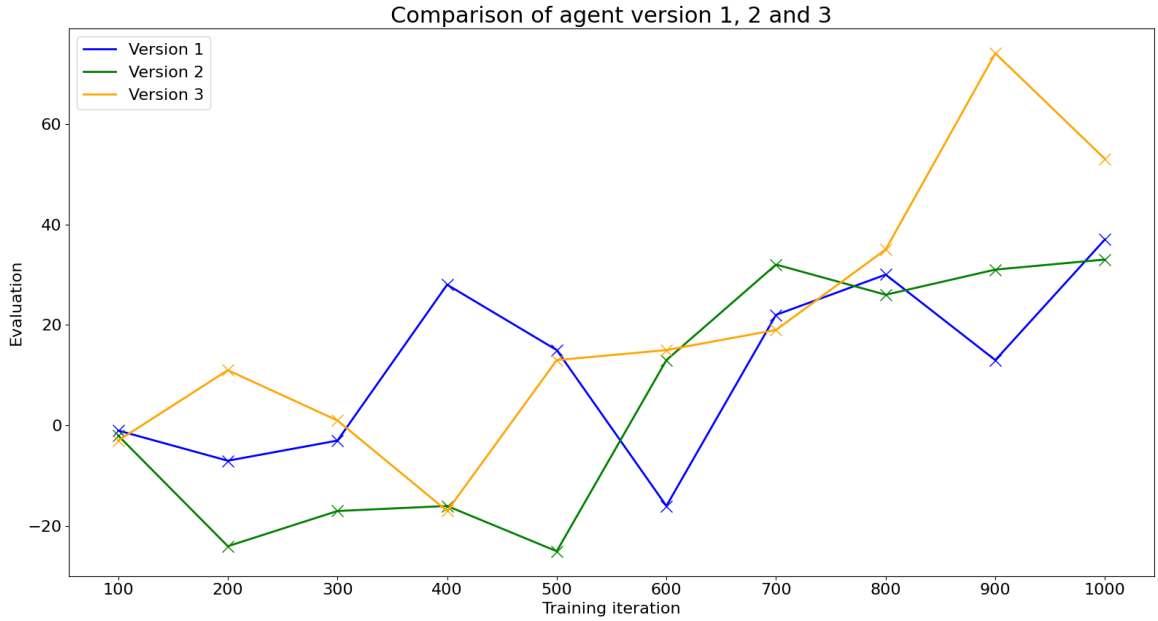


Figure 4.5: Comparison of agent version 1, 2 and 3

4.4. Version 4

The usage of the opening book and endgame table from version 3 are kept. In the first three versions, the agent played against a random agent during the training. In version 4, this is changed to a self-play approach. After every iteration, the opponent is updated to the current weights of the agent. Doing this, new training samples are always collected with the current version of the agent as an opponent. But the training samples that were collected in a previous iteration and are still being used for training, were collected using an older version of the agent. But because the changes to the agent should not be too massive, this should not cause any problems. Of course, the older a training sample, the less useful it might become. This is avoided by removing old games from the training samples. Figure 4.6 shows the evaluation results of version 4. It can be seen that after iteration 400, the training speed becomes slower, and it is a periodical process of improving, then getting slightly worse for 200 iterations and improving again afterwards.

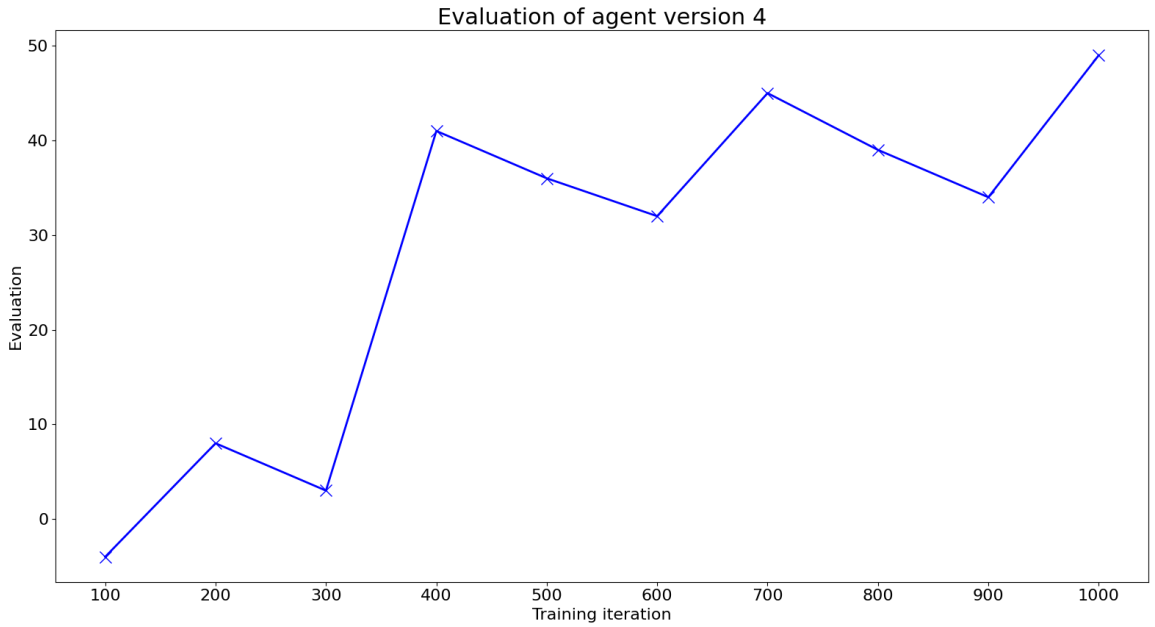


Figure 4.6: Evaluation of agent version 4

In comparison to the older version, version 4 does not fall below 0 (except the early checkpoint at iteration 100), which means it is continuously better than the random agent. Version 3 and especially the spike of it at iteration 900 is better than version 4, but with the advantages mentioned in section 2.3.5 and the more static learning

progress, the self-play is kept in the training process. Furthermore, the self-play reduced the time the training took drastically (figure 4.20). When training an agent, this is a huge advantage, because then the agent can train more in the same time and most likely get better because of that in the same time period. In this experiment the number of iterations is fixed though (for comparison's sake), so this benefit does not apply to the trained versions in this project.

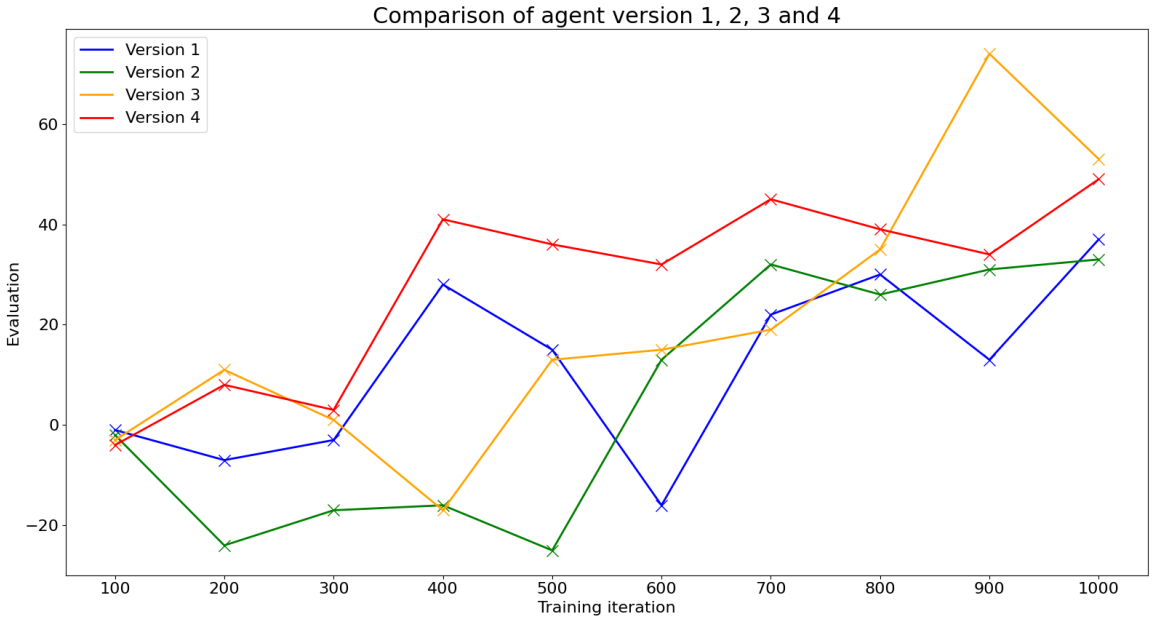


Figure 4.7: Comparison of agent version 1, 2, 3 and 4

4.5. Version 5

As described in section 2.3.3, part of the error that PPO strives to minimize can be the KL divergence. By default, RLlib adds the KL divergence to the error with a factor. This factor changes automatically throughout the training, but is always bigger than 0 (meaning the KL divergence is always added to the error). This tries to avoid too big changes to the policy to reduce the risk of worsening the agent. Because of the used invalid action masking (and a different set of allowed actions for every move), the policy results change drastically every prediction. Depending on the used representations in Python for the (almost) zero per cent probability of not allowed actions, the KL divergence might even become infinite. But that does not mean that the policy has changed much, thus the KL divergence is not a good metric for calculating the difference between the policies in this case. To train version 5, the coefficient of the KL divergence in the error is set to zero and hence the KL divergence is ignored and not added to the error. The results of that agent are shown in figure 4.8.

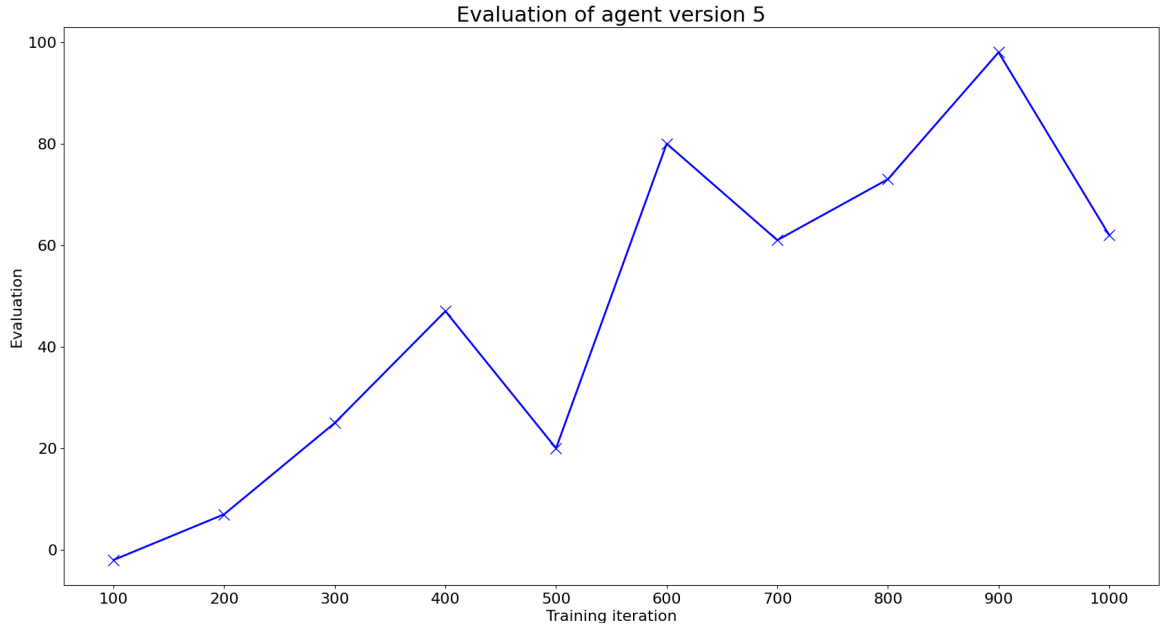


Figure 4.8: Evaluation of agent version 5

The training was much more effective, which can be seen especially in the comparison with the other versions in figure 4.9. Version 5 achieved a level of skill that is unmatched by any of version 1 to 4. After 900 iterations, version 5 wins 9.8 % more games than it loses to the random agent. That is a 33.3 % increase in comparison to version 3

after 900 iterations, the best version so far. Another interesting thing visible in the comparison is that version 5 (except at iteration 500) is always the best agent after iteration 200. Version 5 does not just achieve the best results, it therefore also learns faster than the previous versions. Hence, it is advisable to always disable the KL divergence part of PPO's error when working with invalid action masking.

Figure 4.9: Comparison of agent version 1, 2, 3, 4 and 5

4.6. Version 6

Version 6 builds on version 5 but changes the entropy coefficient of it. As explained in section 2.2.2, the entropy is a measurement for how equally a probability distribution is distributed. RLlib does not add the entropy part of PPO to the error by default. The coefficient of this is set to 0 and has been in versions 1 to 5. To increase the exploration of the agent in the first half of the training, the coefficient is set to 0.005 at iteration 0 and is linearly decreased till it becomes 0 at iteration 500[34]. This way, the policy should be more equal at the beginning, meaning it explores more and then slowly decrease the exploration factor and focus on finding the optimal policy after iteration 500. The results of that experiment can be seen in figure 4.10.

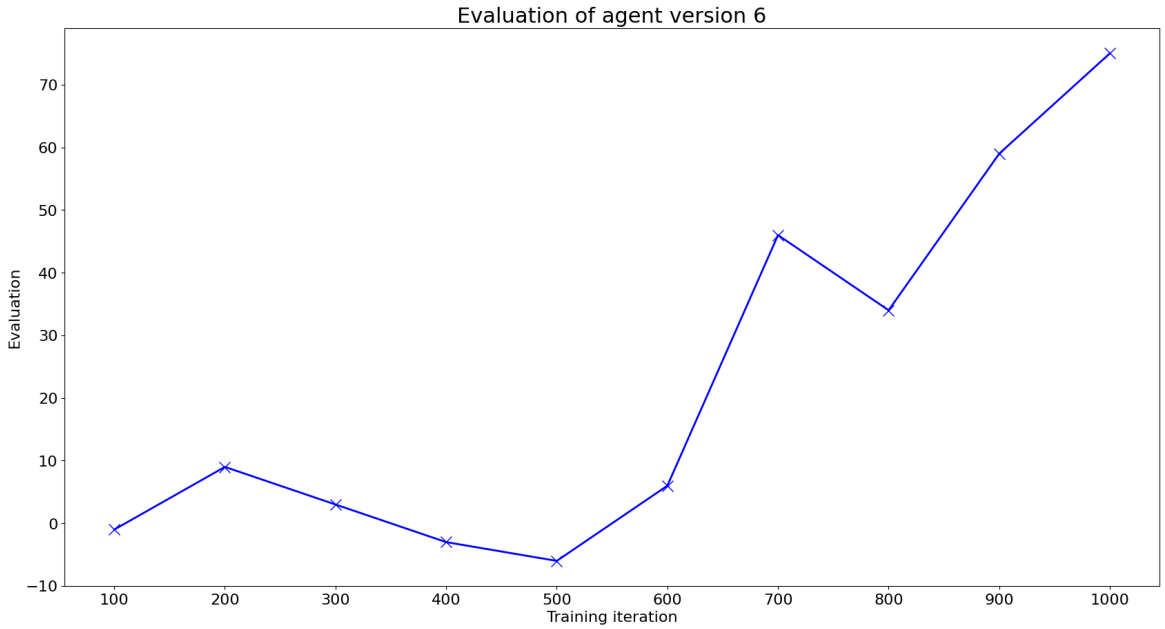


Figure 4.10: Evaluation of agent version 6

The graph shows, that the first 500 iterations the agent does not make progress or becomes stronger. This was to be expected and because it has explored more, it should learn much faster in the second 500 iterations. The results seem to support this theory. But if compared to the other versions (figure 4.11), especially version 5, it can be seen that even though it learns faster, it never catches up with version 5 at 900 iterations. This approach might be beneficial for longer training times where the agent can benefit for more iterations from the learned information in the increased exploration phase. But for this setting it results in a worse agent, and thus this is

removed for the following versions. Versions 7 to 10 have the entropy coefficient set back to 0.

Figure 4.11: Comparison of agent version 1, 2, 3, 4, 5 and 6

4.7. Version 7

An empirical study recommends setting the clipping parameter of PPO to 0.25[33]. In the PPO paper, a value of 0.2 is mentioned[20]. The default clipping parameter used by RLlib is 0.3 and was used for the previous versions. Version 7 is trained with a clipping parameter of 0.25, but otherwise the same parameters as version 5.

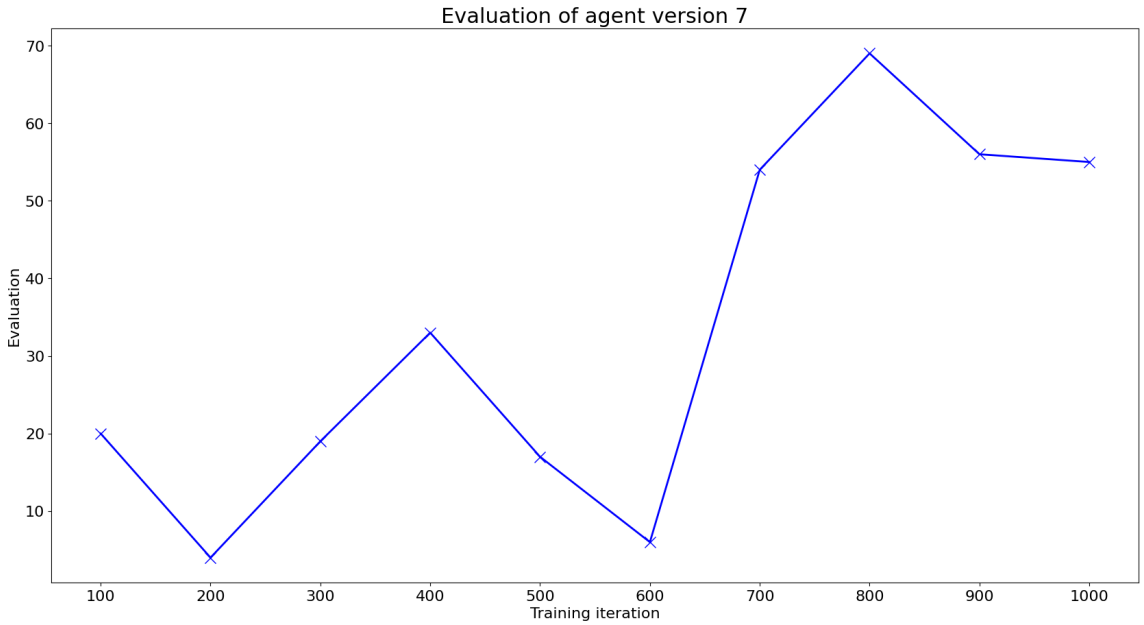


Figure 4.12: Evaluation of agent version 7

The change to the clipping parameter negatively affects the training, as shown in figure 4.12. Two times during the training it goes down almost back to 0 (iteration 200 and 600) and never gets above an evaluation score of 70. The comparison in figure 4.13 shows, that not just version 5 reached a better result, but also versions 3 and 6. The clipping parameter will therefore be reverted to the default value of 0.3 and the change is not kept for the next versions.

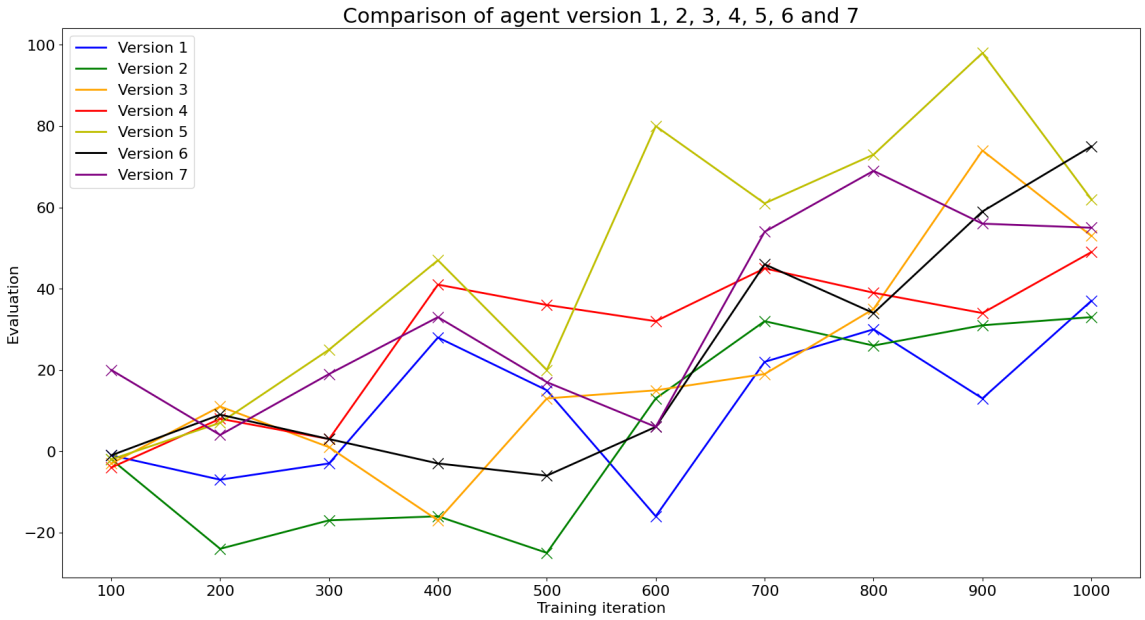


Figure 4.13: Comparison of agent version 1, 2, 3, 4, 5, 6 and 7

4.8. Version 8

Another thing that is mentioned and recommended by the empirical study is the use of Generalized Advantage Estimation (GAE) as the advantage estimator with the parameter λ set to 0.9. GAE tries to reduce the needed number of samples for the training by reducing the variance of the policy gradient estimation[35]. This increases the bias, but the reduction of the variance should be bigger than the increase in the bias, and thus theoretically improving the training. The second issue GAE addresses is the problem that the progress often times is not stable and steady. This is also visible from the training history of the different versions of the agent. By using a trust region optimization procedure for the policy and value function, this problem is addressed. The results of the training with the GAE are presented in figure 4.14.

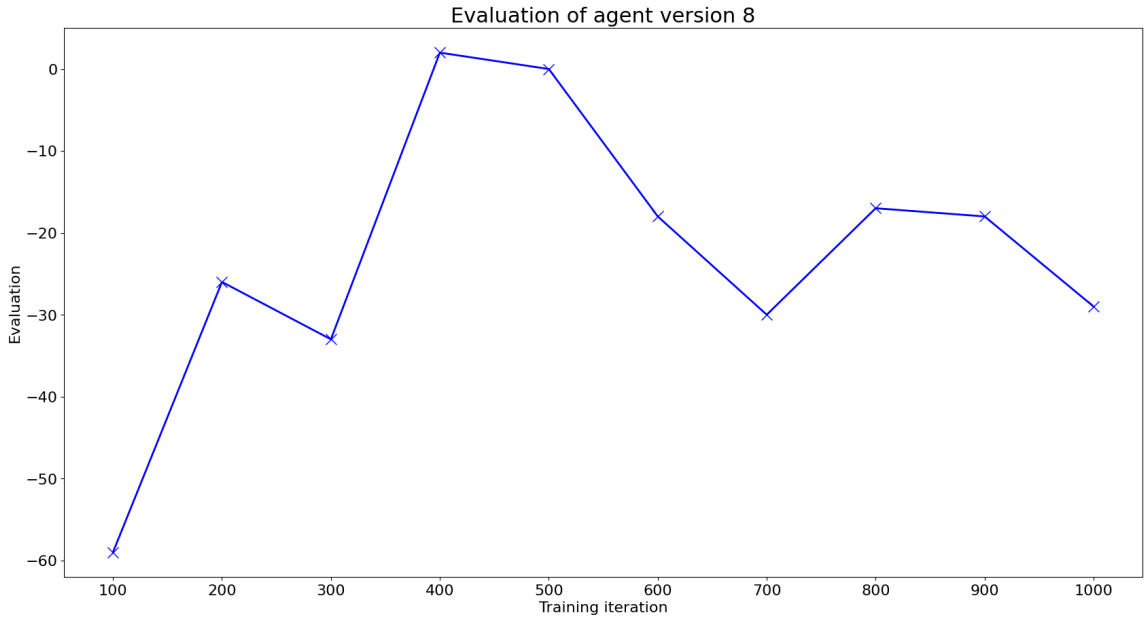


Figure 4.14: Evaluation of agent version 8

The results demonstrate that the use of GAE is not useful for this environment. The agent peaks at 400 iterations and becomes worse after that again. At 400 iterations, it is furthermore the only time that the agent wins more games than it loses against the random agent. Interestingly, the agent is always worse when playing white than black after iteration 200 which can be seen in table 4.1.
Especially from the comparison with the other versions (figure 4.15) can be seen that version 8 is the worst version trained so far. Except at iterations 400 and 500, the

version 8 agent has always the worst performance. Thus it can be concluded, that the usage of GAE is not advisable for this kind of task and will therefore be removed in the next versions 9 and 10.

Iteration	Evaluation white	Evaluation black
100	-22	-37
200	-13	-13
300	-17	-16
400	-12	14
500	-19	19
600	-17	-1
700	-19	-11
800	-23	6
900	-21	3
1000	-20	-9

Table 4.1: Evaluation of version 8 by side

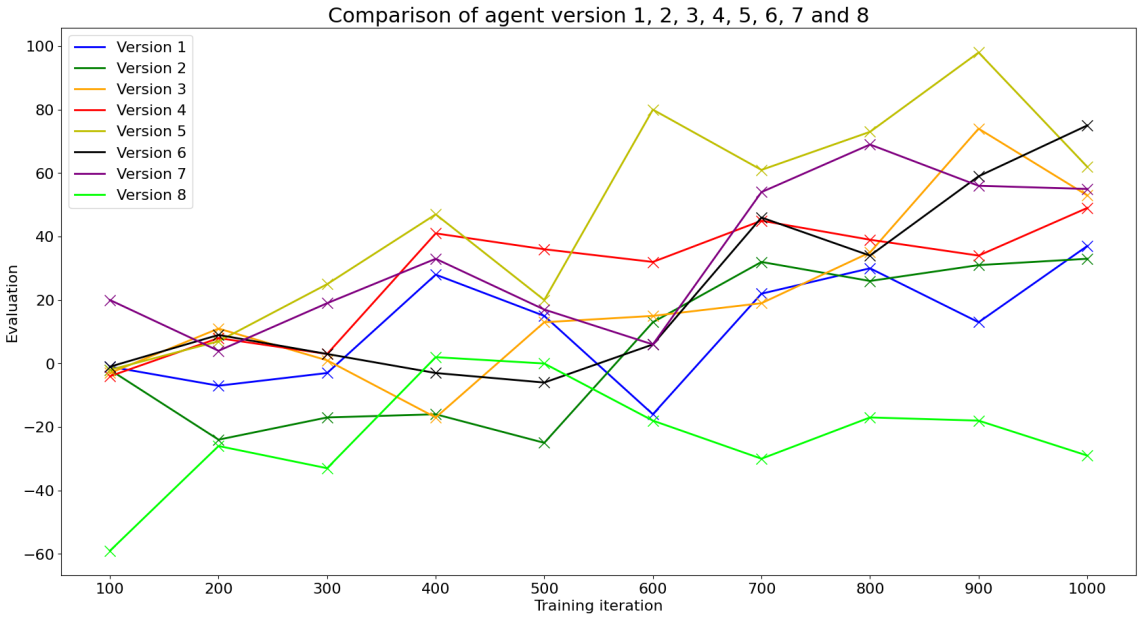


Figure 4.15: Comparison of agent version 1, 2, 3, 4, 5, 6, 7 and 8

4.9. Version 9

In comparison to big companies and software projects, the amount of training done for one agent is very small. To compensate for that, the learning rate is increased. The learning rate controls the size of changes done to the policy, a bigger learning rate causes the policy to be changed more in the direction of the gradient, whereas with a smaller learning rate the steps are smaller along the gradient. This could have the downfall, that changes away of the optimal policy are also bigger. The default learning rate of the PPO implementation of RLlib is 0.00005. This is increased to 0.0003 which is mentioned as a good starting point by [33]. With the bigger learning rate, version 9 gets the results shown in figure 4.16.

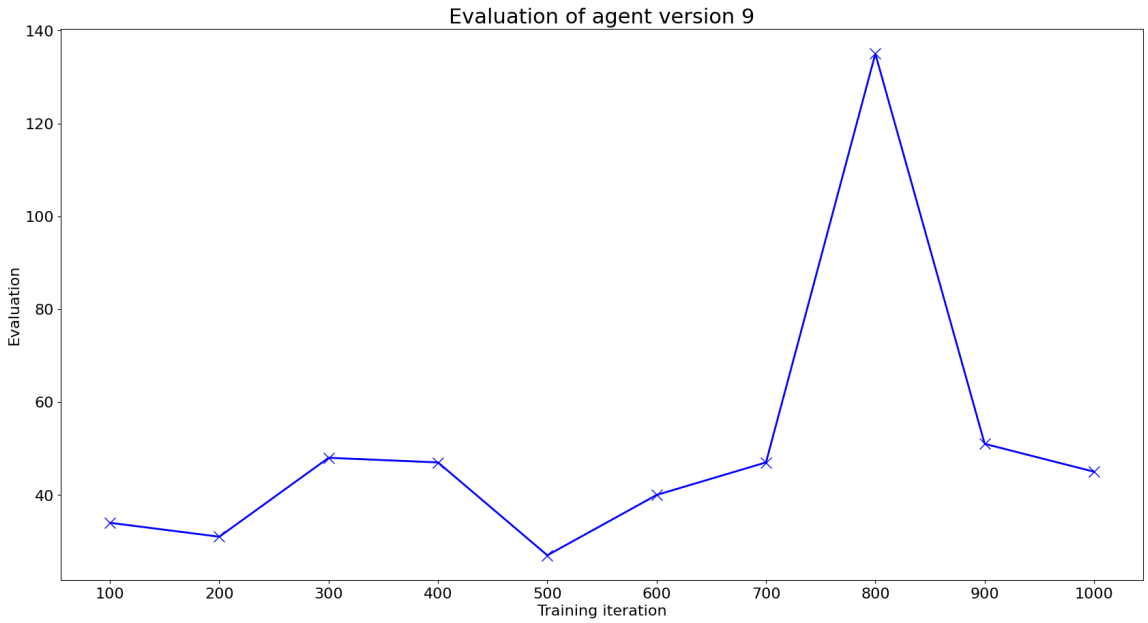


Figure 4.16: Evaluation of agent version 9

The strength of the agent appears to be relatively stable, except for one jump at iteration 800. At this point, the agent increases from an evaluation score of 47 at iteration 700 to 135 and falls back to 51 at iteration 900. The increase can be explained by the fact that the agent learned the scholar's mate and a variation of it between iteration 700 and 800. At iteration 800 it always tries to open with scholar's mate or variations of it and the random agent cannot defend against it, doing the needed moves only with a low probability. This causes the enormous spike at iteration 800, because the agent can win a lot of games with the scholar's mate. The agent is more

successful with this opening against the random agent when playing white. During the evaluation, the agent could win 116 games and only losing 24 games as white of the 500 games, whereas it only won 75 with black and lost 32. At iteration 800, the agent does not know how to defend against the scholar's mate though, so while using it as its standard opening, it can be easily defeated using the scholar's mate. The defense against the scholar's mate is learned between iteration 800 and 900. At iteration 900 it knows how to defend against scholar's mate and because it is playing against itself during the training, it also stops using it because it became less successful. This is the explanation why the evaluation score drops again after iteration 800. Which does not mean the agent became weaker, because it now has learned to defend against scholar's mate, the chances that the random agent tries it are so little, that these improvements cannot be seen in the chosen evaluation metric.

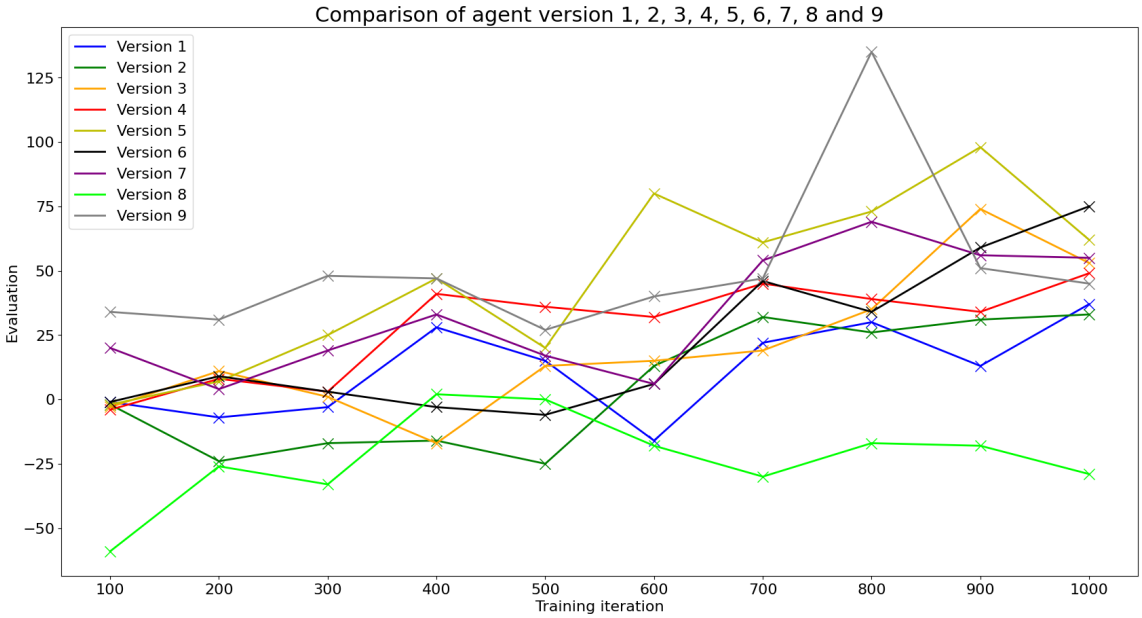


Figure 4.17: Comparison of agent version 1, 2, 3, 4, 5, 6, 7, 8 and 9

When compared to the other versions (figure 4.17), version 9 starts off as the best agent for the first 400 iterations and with its spike at iteration 800 achieving the best evaluation score of all versions. By purely relying on the evaluation score, version 9 at iteration 800 is the best agent (at least when playing against a random agent). From this experiment can be concluded, that it can be beneficial to increase the learning rate when the training time is restricted and comparably short.

4.10. Version 10

The discount factor (gamma) of a reinforcement learning algorithm defines, how much future rewards should be weight in relation to instant rewards[36]. It therefore has a big influence on the optimal policy that is found by the agent. An empirical study found, that a discount factor of 0.99 is a good starting point for many environments[33]. 0.99 is also the default discount factor in RLlib's PPO implementation and has been used for the training of all previous agents.
A game of chess can have a lot of moves till it is finished, but in the environment designed for the training the number of moves has to be finite. This is because of the rule that after a position occurring three times, an automatic draw is claimed. The number of fields and pieces is finite; thus, the number of possible positions has to be finite as well. It is very large, but finite. Hence, the third repetition of a position has to occur after also a finite number of moves. It can therefore be argued, that the task of a chess game (in this environment at least) is an episodic task. For episodic tasks a discount factor of 1 (so no discounting and simply summing up all future rewards) is advisable[37]. Because otherwise the agent learns to end (ideally win) the game in as little moves as possible[38]. This is not problematic, as a quick win is as good as a slow one, but it increases the bias of the agent towards quicker wins which might cause it to avoid a slow win for a less optimal solution. The results of the training with a discount factor of 1 are shown in figure 4.18. The results are very unstable, with two drops in performance at iteration 600 and 900. It is interesting to note that version 10, in contrast to version 9, did not learn any opening tactics. This could be caused by the increase of the discount factor, because relative to the reward at the end of a game, the rewards after a move are bigger in version 9 (because the end reward is discounted). It is therefore more important for agent version 9 to learn the opening moves than it is for version 10. Because of the big difference in reward size (end reward is 1000 whereas interim reward only 10), version 10 is mainly focused on winning the game in any way possible to maximize the return. But with only a sample size of one each, it cannot be said with certainty that the reason for that behavior is the increase of the discount factor. It might as well be, that version 9 only learned the opening by chance or that another trained version with discount factor 1 will not learn an opening within the first 1000 iterations. As shown by the comparison (figure 4.19), version 10 does not achieve the same evaluation score as version 9, but is still one of the best agents trained. This supports the theory that the settings of version 9 and 10 are superior to the earlier versions. But it also demonstrates, that the success of the training is dependent on luck and an agent trained with the same settings will not result in the same agent with the same strength. This demonstrates that it is not always simple to

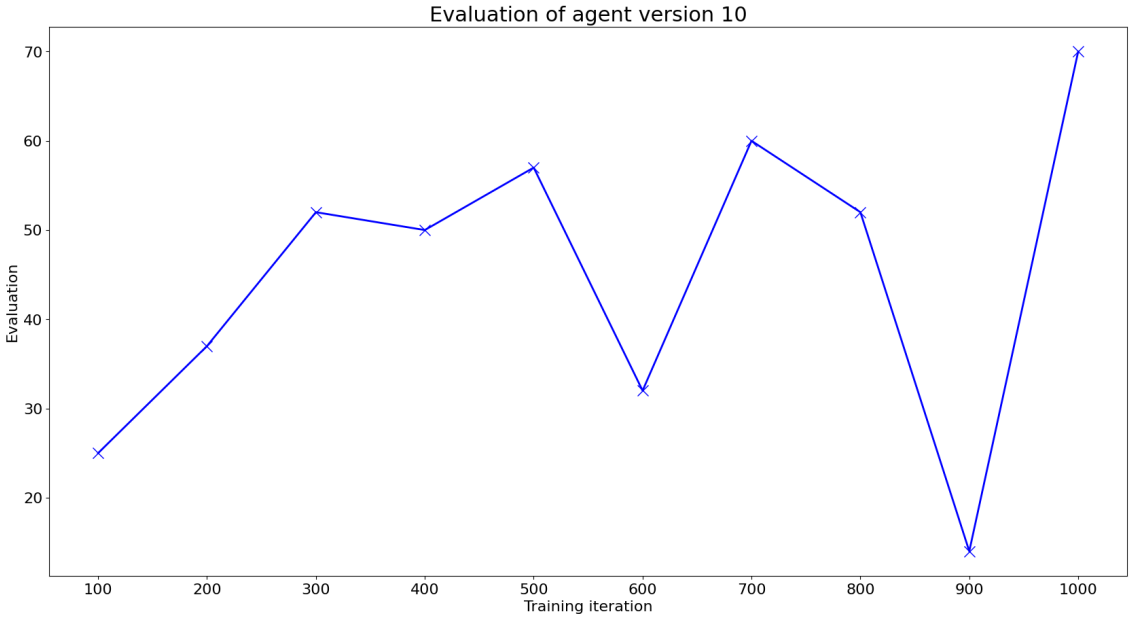


Figure 4.18: Evaluation of agent version 10

compare the effectiveness of training settings on the base of one trained agent. But for this project it is the most reasonable and viable solution.

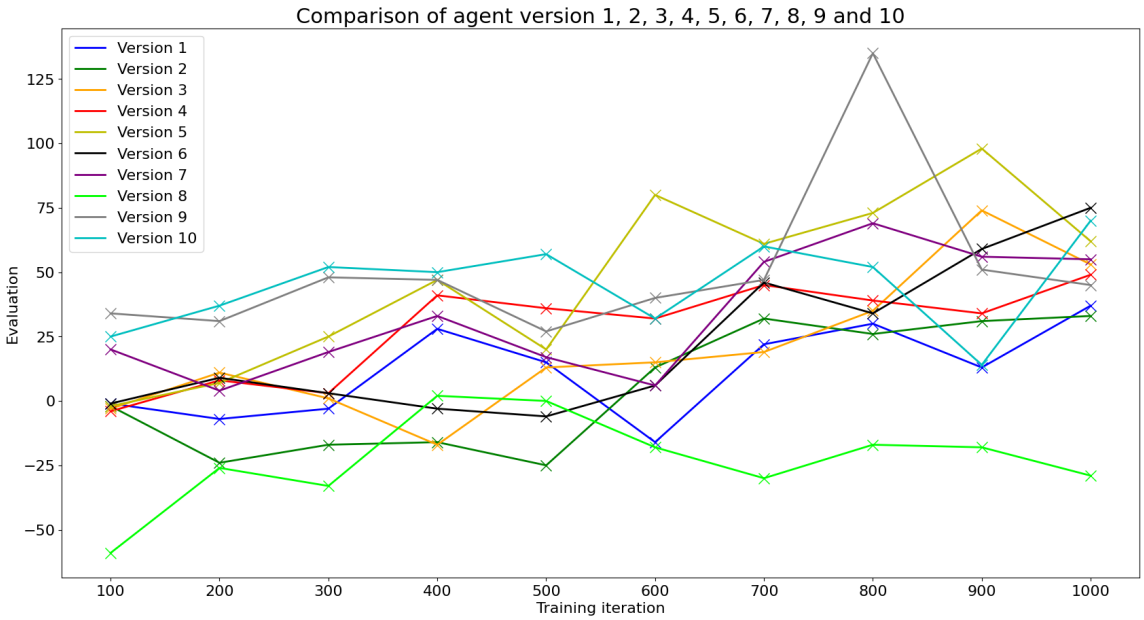


Figure 4.19: Comparison of agent version 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10

4.11. Training time

Figure 4.20 shows the time that the different versions needed for training for 1000 iterations. It shows, that the change from using a random agent as the opponent during the training to self-play decreased the training time drastically. This can be explained by the calculations of PPO being faster than randomly deciding for an action, and thus the sampling time for the training samples is much shorter. This is another benefit of the self-play, and one of the reasons why it has been kept throughout the other versions. After version 4 there are only small changes in the training time, which might have been caused by other factors than the changes applied between the versions.

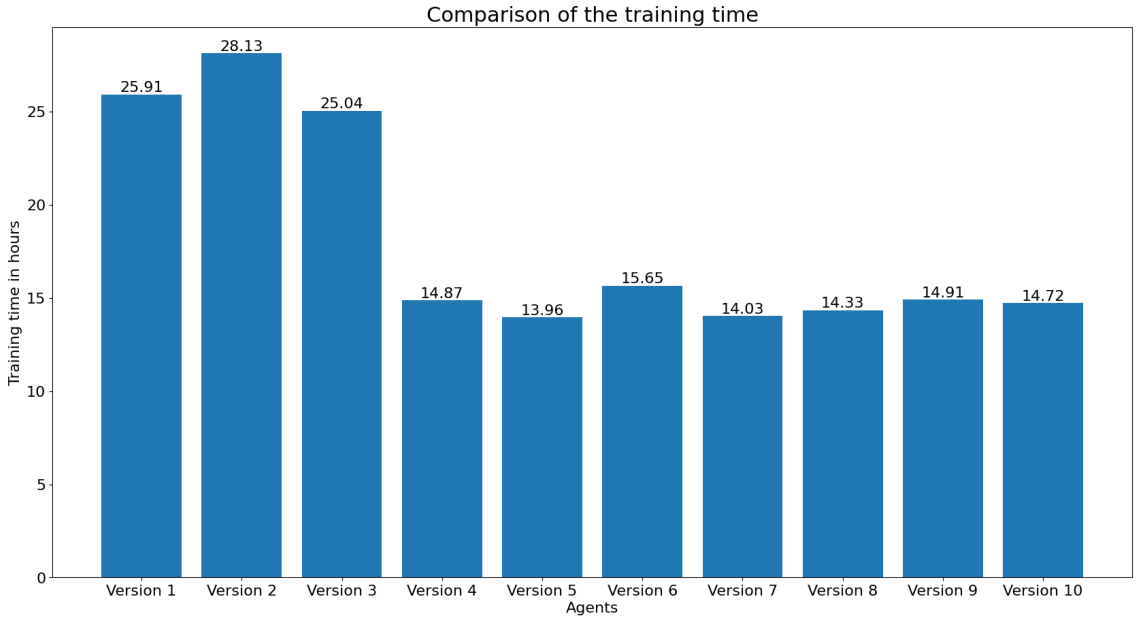


Figure 4.20: Comparison of the training time

5. Discussion

The best agent trained (at least according to the chosen evaluation metric) is version 9 at iteration 800. It can be seen from the results that this agent is clearly better than an agent randomly choosing an allowed move. But the question arises how strong it is comparably to a real chess program such as Stockfish. Stockfish has won 100 games of 100 against the random agent and appears to be stronger (Stockfish's strength depends on the time limit it was given for a move, for this test a limit of 0.1 seconds was used). In a direct comparison of agent version 9 at 800 iterations and Stockfish, Stockfish won all games disregarding of the time limit given. Even with a very small time limit, Stockfish easily defeated the trained agent. This shows that the trained agent is far from being a good chess player and this can be seen from playing against it, it is relatively easy to win against it even as a not very good player. Projects such as AlphaZero and Leela Chess Zero have been able and Leela Chess Zero is still using a reinforcement learning approach to produce some of the best chess programs. There are two main explanations for the difference in strength between for example Leela Chess Zero and the trained agents here. Firstly, these projects are most likely using a better reinforcement learning method and have their training parameters better tuned. But this statement cannot be derived directly from a comparison between the strength of these agents with the ones trained in this project, because of the different training conditions used to produce the agents. The different agents and methods would need to be trained equally under the same conditions to have a meaningful comparison and evaluate which architecture is better fitted for chess. This leads to the second, even more important, explanation: the difference in used hardware for the training. For the training of AlphaZero a total of 5064 TPUs were used, whereas for the training of these agents only a single graphics card was used. For Leela Chess Zero, a software is offered with which the agent can be trained on many computers of private individuals as well as the hardware already used by the project. These differences explain why these agents are able to reach performances far beyond the agents trained on a single graphics card. Assuming reasonable training settings, an agent almost always profits from more training and becomes better. And when the difference in the training hardware and

therefore the amount of played games is that massive, it leads to a very big difference in the strength of the agents.

6. Future Work

The trained agents are not optimal, there are many hyperparameters left to fine-tune and improve the agent and its performance. Even the fine-tuned hyperparameters can, given enough time, be improved further. One useful approach would be to use an automated hyperparameter optimization approach. The problem with these is, that they need a lot of time and/or resources to be used. For example in grid search, a wide variety of different agents have to be trained to see which hyperparameter combinations work best. Population-based hyperparameter optimization also needs a lot of computational resources, even newer and less computational demanding approaches[39]. If the resources are available, these would be good ways to increase the performance of the agent. If not, then the hyperparameter optimization has to be continued to be done manually. This might be less effective and time-consuming, but keeping the goal of training the agent on a "normal" computer does not allow for more complex approaches. Besides the hyperparameter of the agent, there might be changes to the structure of the agent that can enhance its training efficiency. It might be beneficial to explore different network structures for the policy predictions and see which work best and if other, then the here chosen architecture, might be better suited to the task.
It might also be interesting to see, if training the agent for longer will result in a much stronger agent or if it is nearly impossible to achieve a superb agent using such basic hardware. For that, the agent would probably need to be trained for weeks or even months, to make an accurate prediction about the limitations imposed by the hardware. Regarding the limitations of the training hardware, an interesting experiment would be to train the PPO agent and the AlphaZero agent on the same (simple) hardware for the same period of time to see which of these two is conceptually better for the task of chess. This would also show, how good an agent (that has been shown to be able to outperform humans) can get by being trained on hardware an average person might have. This would provide further information about if, in the field of reinforcement learning, it is only possible to make groundbreaking progress by using enormous amounts of computation power, which only a few companies or organizations can provide.

7. Conclusion

To train the agents and compare the results to agents trained by big organizations with access to a lot of computation power, a chess environment is implemented. This environment enables the configuration of many parameters of the environment and therefore makes the fine-tuning of the hyperparameters easier. This chess environment offers the gym interface and is thus compatible with most reinforcement libraries and algorithms offered by them. To make the testing of the agents simpler, a GUI is offered with which one can play against the agents. The trained agents use the PPO algorithm implemented by the RLlib library. The environment rewards the agents at the end of the game if it wins and punishes it for a loss. Because chess has many different possible actions to take and not all actions are valid in every state, invalid action masking is used to force the agent to only choose a valid action. To make the trained agents comparable among each other, all are trained for 1000 iterations each on 75 % of a Nvidia RTX 2060. During the project, a total of ten different agent versions were trained and evaluated. The evaluation is done by letting the agent play 1000 games (500 as white and 500 as black) against a random agent (an agent that always chooses an action randomly from the allowed ones in that state). Over the course of the different versions, some parameter and training settings have been found to perform better than others for this project. But some of these are also recommendable for similar projects, and not just for this environment and task. One of the things that improves the training performance drastically is the use of interim rewards. This is advisable for all projects where one big reward is given only at the end of one epoch, because this can increase the speed with which the agent learns, especially at the beginning of the training. For this environment, interim rewards are implemented by using an opening book and endgame table to reward moves that have been shown to work better than others. Furthermore, the use of self-play enhances the training and, in this environment, decreased the needed training time drastically (in comparison to using a random agent as the opponent). Another recommendation that derives from the training is the elimination of the KL divergence from the error of PPO when invalid action masking is used. Some implementations of PPO add the KL divergence between the new and old policy to the

error to minimize. But when invalid action masking is used, this value does not give meaningful information about the distance of two probability distributions (policies) and therefore skews the error. This leads to not optimal training updates and should therefore be disabled when invalid action masking is used. Another thing that showed to be useful is increasing the learning rate when the training time and resources are less. Additionally, because a chess game is a episodic task, using a discount factor of 1 is beneficial. The experiments have also shown some (hyper-)parameter to cause no improvement or even make the agent worse. These are primarily the adding of the entropy to the error function of PPO, reducing the clipping parameter and using GAE as the advantage estimation.
The best trained agent is better than a random agent, winning 13.5 % more games against it than it loses. But against Stockfish it does not have a chance, Stockfish defeats it in every game disregarding of the time limit given to Stockfish. Even an average playing person can easily win against the agent. This shows that it is possible for a single person with a rather "normal" hardware to achieve some results using reinforcement learning. But in contrast to a big organization with enormous computation power, the results achievable are not sensational or may even be not good enough for many (complex) applications. This evokes the impression that reinforcement learning has become a topic which is only truly relevant and useful for big organizations with a large amount of computation power. Especially for more complex projects and applications, this computation power is needed to achieve the results wanted and/or required.

Bibliography

[1] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712.01815 [cs.AI].

[2] OpenAI et al. *Dota 2 with Large Scale Deep Reinforcement Learning*. 2019. arXiv: 1912.06680 [cs.LG].

[4] Sehoon Ha et al. *Learning to Walk in the Real World with Minimal Human Effort*. 2020. arXiv: 2002.08550 [cs.RO].

[7] "FIDE LAWS of CHESS". In: (Nov. 2008).

[8] Claude E. Shannon. "XXII. Programming a computer for playing chess". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), pp. 256–275. DOI: 10.1080/14786445008521796. eprint: https://doi.org/10.1080/14786445008521796. URL: https://doi.org/10.1080/14786445008521796.

[9] Nenad Tomašev et al. "Assessing Game Balance with AlphaZero: Exploring Alternative Rule Sets in Chess". In: (Sept. 2020).

[13] S. Kullback and R. A. Leibler. "On Information and Sufficiency". In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86. DOI: 10.1214/aoms/1177729694. URL: https://doi.org/10.1214/aoms/1177729694.

[14] Solomon Kullback. *Information Theory and Statistics*. 1959.

[15] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.

[16] Edwin T. Jaynes. "Prior Probabilities". In: *IEEE Transactions on Systems Science and Cybernetics* 4.3 (1968), pp. 227–241. DOI: 10.1109/TSSC.1968.300117.

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

[18] Yuxi Li. *Deep Reinforcement Learning*. 2018. arXiv: 1810.06339 [cs.LG].

[19] Kai Arulkumaran et al. "Deep Reinforcement Learning: A Brief Survey". In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38. DOI: 10.1109/MSP.2017.2743240.

[20] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[21] Shengyi Huang and Santiago Ontañ ón. "A Closer Look at Invalid Action Masking in Policy Gradient Algorithms". In: *The International FLAIRS Conference Proceedings* 35 (May 2022). DOI: 10.32473/flairs.v35i.130584. URL: https://doi.org/10.32473%2Fflairs.v35i.130584.

[22] Trapit Bansal et al. *Emergent Complexity via Multi-Agent Competition*. 2018. arXiv: 1710.03748 [cs.AI].

[33] Marcin Andrychowicz et al. *What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study*. 2020. arXiv: 2006.05990 [cs.LG].

[34] Zafarali Ahmed et al. *Understanding the impact of entropy on policy optimization*. 2019. arXiv: 1811.11214 [cs.LG].

[35] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018. arXiv: 1506.02438 [cs.LG].

[36] MyeongSeop Kim et al. "Adaptive Discount Factor for Deep Reinforcement Learning in Continuing Tasks with Uncertainty". In: *Sensors* 22.19 (2022). ISSN: 1424-8220. DOI: 10.3390/s22197266. URL: https://www.mdpi.com/1424-8220/22/19/7266.

[37] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[38] Paul Ecoffet et al. "Policy Search with Rare Significant Events: Choosing the Right Partner to Cooperate with". In: (Mar. 2021).

[39] Jack Parker-Holder, Vu Nguyen, and Stephen J Roberts. "Provably Efficient Online Hyperparameter Optimization with Population-Based Bandits". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 17200–17211. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/c7af0926b294e47e52e46cfebe173f20-Paper.pdf.

Websites

[3] "OpenAI Five defeats Dota 2 world champions". In: (). URL: https://openai.com/research/openai-five-defeats-dota-2-world-champions (visited on 04/18/2023).

[5] Marco Ramponi. "How ChatGPT actually works". In: (Dec. 2022). URL: https://www.assemblyai.com/blog/how-chatgpt-actually-works/ (visited on 04/18/2023).

[6] "Leela Chess Zero Website". In: (). URL: https://lczero.org/ (visited on 04/18/2023).

[10] revoof. "7-piece Syzygy tablebases are complete". In: (Aug. 2018). URL: https://lichess.org/blog/W3WeMyQAACQAdfAL/7-piece-syzygy-tablebases-are-complete (visited on 04/18/2023).

[11] Rocky64. "Eight-piece tablebases – a progress update and some results". In: (Oct. 2021). URL: https://www.chess.com/blog/Rocky64/eight-piece-tablebases-a-progress-update-and-some-results (visited on 04/18/2023).

[12] "Syzygy Bases". In: (). URL: https://www.chessprogramming.org/Syzygy_Bases (visited on 04/18/2023).

[23] "Stockfish 15.1". In: (Dec. 2022). URL: https://stockfishchess.org/blog/2022/stockfish-15-1/ (visited on 04/18/2023).

[24] "Komodo Dragon Website". In: (). URL: https://komodochess.com/ (visited on 04/18/2023).

[25] "TCEC Results". In: (). URL: https://tcec-chess.com/ (visited on 04/18/2023).

[26] "Python-chess documentation". In: (). URL: https://python-chess.readthedocs.io/en/latest/ (visited on 04/18/2023).

[27] "RLlib documentation". In: (). URL: https://docs.ray.io/en/latest/rllib/index.html (visited on 04/18/2023).

[28] "OpenAI Gym GitHub". In: (). URL: https://github.com/openai/gym (visited on 04/18/2023).

[29] "Announcing The Farama Foundation". In: (Oct. 2022). URL: https://farama.org/Announcing-The-Farama-Foundation (visited on 04/18/2023).

[30] "Pygame documentation". In: (). URL: https://www.pygame.org/docs/ (visited on 04/18/2023).

[31] "New chess opening book: M11.2 (bin and ctg)". In: (Jan. 2021). URL: https://chessengines.blogspot.com/2021/01/new-chess-opening-book-m112-bin-and-ctg.html (visited on 04/18/2023).

[32] "Top 100 Players April 2023". In: (Apr. 2023). URL: https://ratings.fide.com/ (visited on 04/18/2023).