The Influence of the Size of the Search Space on Learning to Play Chess using Deep Reinforcement Learning Algorithms

Aksel Hakim Zakuto

[1]

Abstract

The current state-of-the-art solutions for playing Chess, are created using deep reinforcement learning. AlphaZero, the current world champion, uses 'policy networks' and 'value network' for selecting moves and evaluating positions respectively. However, the training of these networks are done using reinforcement learning from games of self-play [10]. There are many factors which determine the learning speed of reinforcement learning algorithms [2], where the size of the search space is a main one. In this research, we have tried to see the effect of the size of the search space on the time it takes the reinforcement learning agent to learn.

1 Introduction

1.1 Background of the research

Chess is a game that is played for centuries, and can arguably be considered the most popular board game in the history of human civilization [11]. It always stars from the same position where both players are equally likely to win. There is no luck or randomness involved which makes it a very skill-based game. Because of this fact, throughout its existence, it is used as a metric for intelligence.

The rules of Chess are very simple but knowing the game doesn't make you a good player, there is always a room for improvement. Unless you have almost infinite compute power and calculate all possible moves which branch exponentially until they all result in a finished game state, you can never be sure of what the outcome of your move is going to be. In this case, where you don't have the compute power to calculate all possible scenarios, you have to decide on which scenarios you have to calculate in the limited time you have, before making a move. Deciding which positions to consider and evaluating their advantage is what separates a good and a Chess player, just like how it separates a good and a bad Chess AI(engine).

Even before there was the term 'artificial intelligence', humans were interested in automating the game of Chess. The first Chess programs were only able to solve Chess puzzles or play endgames. In 1956, Paul Stein and Mark Wells developed the first fully automatic Chess playing program using logic [6]. At this point, the power of the program was very

[1] "Supervised by Greg Neustroev (G.Neustroev@tudelft.nl) and Mathijs de Weerdt (M.M.deWeerdt@tudelft.nl)"

weak and could easily be defeated by a human. After that, the dominance of mankind only lasted for 40 years and in 1996, Deep Blue beat the world champion Garry Kasparov in a game of regular time controls, becoming the first program to beat a reigning world champion. Deep Blue was using an artificial intelligence model where it was trained with tons of data of previous games from Chess masters and grandmasters [5]. Nowadays, the Chess programs are so much stronger than the humans that the meta of the game is shaped by new coming artificial intelligence programs. In 2017, Alpha Zero beat Stockfish in a 100-game match becoming the current Chess champion. Unlike the previous models, this program did not use any of the previous games and learned its strategy only with self-play, using reinforcement learning techniques [10].

Reinforcement learning is especially useful in situations where there is a environment, and an agent and a reward for every possible action the agent performs. Environment is influenced by every decision agent makes, which can either influence positively or negatively the reward it receives. At first, the agent starts with no knowledge of the domain, and thus makes random moves. As it receives positive/negative rewards, it learns what kind of moves to make in certain situations. This approach is very applicable to Chess where the environment is the board, the agent is the player (Chess engine) and rewards are the evaluation of the board. So, the agent starts learning by making moves in every situation which either puts its position in a more or less winning position. Chess is a zero sum game in nature, which means there is net benefit or loss in the system and one can only gain advantage from the loss of another [1]. Because of this, the game starts with equal position and when a player has a more advantageous position, this implies that its opponent has lost advantage and now is less likely to win the game. Through reinforcement learning, the agent learns to find more advantageous ideas and thus is more likely to be victorious.

There are several factors which make Chess a difficult game to learn and play for humans as well as for computers. Most common factors which makes problems hard are usually stochasticity and sparsity of rewards in the action space. These factors are not the main factors making Chess hard. Current approaches for playing Chess include some probabilistic calculation, especially while trying to predict the future moves and positions. However, by its nature, Chess is a fully deterministic game. On the other hand, the rewards are also not necessarily sparsely distributed. There are many metrics to look at trying to evaluate a position and thus the reward. Even though the final goal is to check-mate, some other indicators such as material advantage, piece activity and king safety are also good representatives the current situation of the game. Even when the possible mating sequences are very far away, using these metrics, it is still possible evaluate the position without computing all possible variations until all mating lines.

The two main factors making Chess a difficult game are the huge number of local optima and even the greater size of the search space of possible moves. Firstly, it is obvious that in most situations, it is not possible to evaluate all possible moves fully into the end of the game. In this case, we are forced to evaluate a limited number of steps in the future and evaluate the positions and thus chose the move which is more likely leading to a winning position. However this assumption may lead the agent to miss the global optima. It is possible to evaluate until certain steps into the future and choose a move which doesn't lead to a winning position but with another move there was a forced mating sequence, but further in the future. However, even this problem is not as huge as the size of the search space for Chess. In average, there are 30 legal moves that can be played in every position. Even though this number doesn't look too scary at first glance, it is crucial to understand how a player evaluates a position. In order to evaluate one more step, in other words, find

the best possible move the opponent player can make for every possible move, you have to evaluate 30x30 moves(30^2).The general formula for total number of positions is:

$$NumberOfPossibleMoves^{Depth}$$

where "depth" is how deep we want to analyze. Given the average game lasts 80 moves, the number of possible moves are 35^{80} or 10^{123}. Considering there are 10^{82} atoms in the observable universe, this number is indeed huge.

2 Methodology

The hypothesis is that, for a certain problem, as the size of the search space increases, the time time it takes to perform at a certain elo (The Elo rating system is a method for calculating the relative skill levels of players in zero-sum games such as Chess) also increases. Specifically for Chess, the required training is shorter for an engine when the search space is smaller. For instance, finding the best move or sequence of moves is harder when there are dozens of possible moves for both sides but much easier if there are only a few. This is a very intuitive hypothesis and also holds with the way humans play Chess. To test the correctness of it for reinforcement learning agents, there are a few steps to take before going further into evaluation.

First, there is a need for a dataset of Chess positions. In between these positions, we are only interested in fixed number of moves until there is a forced check-mate. By this way, we can control the size of the search space and observe its influence on the performance of the agent. By only considering such positions, we are able to sort the positions by their size of search space (number of possible unique positions that are reachable from current position until the check mate). Secondly, we need a Chess engine using current state-of-the-art methodologies including deep reinforcement learning. We will train this agent for different amounts. After acquiring our trained Chess engines and evaluation positions, we can start the research.

For each trained iteration, evaluate the agents performance using all our selected positions. Using our Chess positions of different sizes of search space, we are able to see if the performance improves significantly faster for positions with a smaller search space. If this is the case, this does indeed conclude that the size of the search space directly influences the training time of the agent. For Chess we have also showed that the size of the search space grows exponentially. However, this does not directly imply that the training speed also has to be linear with the growth of the size of the search space (exponential in this case). The current state of the art algorithms do not naively go through every possible move but just like a human, try to estimate which positions can lead up to a better position in the future and focus on those positions more. That is why, growth rate of learning with respect to the growth of the size of search space is also a very important factor for looking into.

3 Experimental Data and Evaluation

Experimental work

Before going into the algorithm, one main problem is the evaluation. For comparing different algorithms or the same algorithm trained for different levels, we must first answer, how to see which one performs better in different circumstances. The first solution that comes to

mind is making two trained agents playing against each other. This is a very direct approach on finding who turned out to be a better player. However, this approach has its limitations. When comparing the two in different circumstances (performance in situations where there is big/small action search space), the evaluation gets more complicated. In general, the size of search space starts high and steadily increases as the position gets more complicated, and as we reach the end game (where only a few pieces for each player are left on the board), it decreases rapidly. So in general, to see the performance in situations where there needs to be calculation of moves in a big search space, it is better to look at the beginning or the middle of the game and for small search space evaluation, focus on the end of the game. The main problem however is that, the early, middle and the end game are connected. If one agent outperforms the other in the early game and creates an advantage for itself, this advantage is carried further into the game and influences the middle and the end games directly. So if one agent is better in early game and the other is better in the late game, it is very difficult to see a better performance from the one that performs better in the late game. This is due to the fact that, the game gets so one-sided that, the other does not even have the opportunity to show its ability.

Algorithm 1: Evaluation of different levels of the agent using problem of different sizes

```
download dataset;
separate the dataset into mate in one, two and three positions;
further label the data into groups of different size search spaces;
initialize the environment, agent and reinforcement learning framework;
for each iteration i ← 0 to n do
    train the agent for n iterations of self-play;
    for each group of dataset d with similar size of search space do
        numberOfSolvedProblems ← 0;
        for Each problem in d do
            if agent solves the problem successfully then
                numberOfSolvedProblems++;
        end
        accuracyOfAgentInThisDataset ←
          numberOfSolvedProblems/numberOfProblemsInDataset;
    end
end
```

Because of this reason, a better way for evaluation is to look for mate-in-one, two or three positions (a mate in two position means that there is a forced checkmate in two moves despite the best play from the opponent). Firstly, this approach is also useful when comparing the performances of several agents. This is done by providing a high number of problems for both agents and comparing their accuracy (how many percent of the problems did the agent find the correct sequence of moves which lead to a check mate). On top of this, it is also much more convenient to label a position by its size of search space. For a mate in two position, the size of the search space can be calculated very easily by looking at every possible move combination of 3 consecutive moves (2 moves for the agent finding the mate in two and one move for the opponent to play in between). It is simpler because, we have to let the agent play for a known number of moves for each position and thus we

can explore the whole search space beforehand. Most importantly, the problems evaluating the accuracy for position with big and small search spaces can be compared independently. Unlike before, where each the late game was just the middle game played further, in this approach, the middle game does not affect anything for the end game.

The current Chess AI bechmark is set on combining deep neural networks, reinforcement learning frameworks and combining these with Monte Carlo Tree Search (MCTS) algorithms (The working of MCTS algorithm is graphically descibed in Figure 1). The deep neural network is used to focus on the more important branches of the tree search. The board position (including some previous board positions for better seeing the long term plan of the opponent) is given as input. For the output, we get possibilities of winning for each possible move. MCTS is just an algorithm for focusing more on the predictions made my the neural network. If the neural network predicts that one certain move is a good move, MCTS spends more time exploring that branch compared to other moves. However, there is still some space left for exploration, and good moves are not exploited to the maximum extent. For deciding on which position is more advantageous, there is a separate value network, which is trained using the reward function of the self-play from reinforcement learning. This approach is very different than the previous benchmarking Chess AI algorithms, where the development was made by small tweaks and ideas implemented by many different people over decades, telling the AI how to behave in certain specific situations and always keeping the best version over the time. This does not only make the algorithm extremely complex and ugly, but it forces it to be very domain specific.

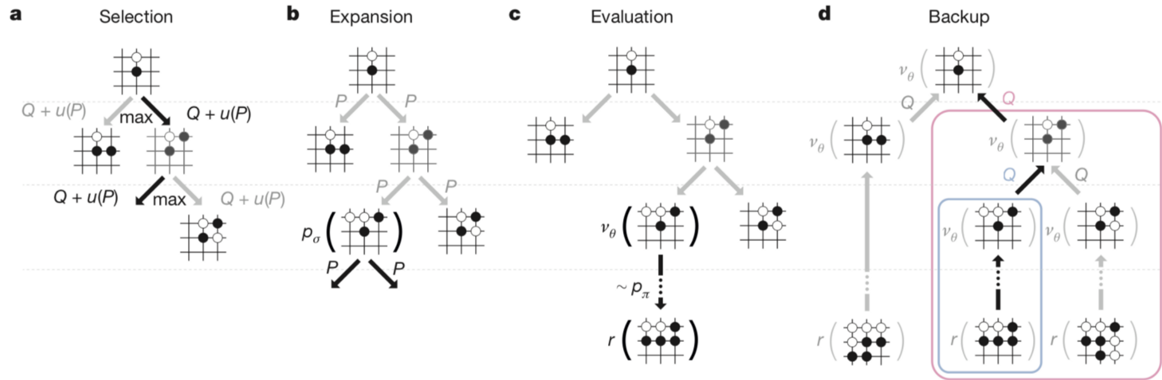

Figure 1: Monte Carlo Tree Search [9]

For the experimental research, we also used an implementation which mimics the current benchmark approach. For this approach, all of the training is done by self-play and the more it plays, the better it becomes. To see the direct effects of size of the search space, on the growth of the accuracy, we simply have to label the positions with respect to their size of their search space, then train the model for different levels and calculate their accuracy separately. Finally, observe the rate of the improvement of the performance for different sized problems and quantify.

Improvement of an idea

It is normal to expect to see the agent perform better as it gets more training. It is also very expected to see it perform better in cases where it is dealing with a smaller search space. The first and the simple task is to verify these claims experimentally. First, train the agent for different levels and evaluate it using the same bunch of problems. Verify that in the long run, training does improve performance. Secondly, when we have to see the effects of the size of the search space, we used different sized problems for evaluating each trained agent and compare their performance. From the hypothesis, we expect to see the agent starts performing better in smaller problems more quickly and being good in bigger problems takes more time.

If this is indeed the case, and we verify that as the size of the search space increases, the training time also increases, we have to quantify this claim. To what extent does it effect? Firstly, there are two ways we can determine the size of the search space. For each position in the tree, the breadth size is equal to the number of possible moves the player can make. The depth size however, is determined by how many moves into the future does the player want to/is able to evaluate (Figure 2). For good practice, it is possible to use both these factors interchangeably to see the effects in more detail.

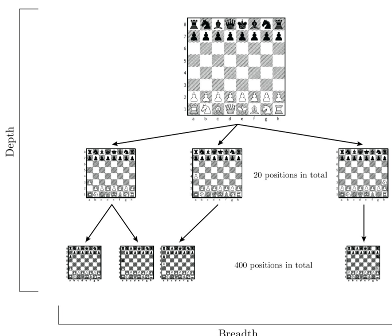

Figure 2: Definition of Breadth and Depth in the context of Chess

We can start by setting the breadth as our independent variable. This means the depth should remain constant throughout this evaluation. We focus on mate in one, two and three moves separately. Starting with mate in one problems, we label each problem with the number of possible moves. The number of possible moves is normally distributed with a mean of 30-35. This means, most of the positions have number of possible moves that are close to the mean and as we move a few standard deviations away,the number of problem drop very fast. For simplicity reasons, we labeled positions with respect to number of possible moves with slices of size 10. This means all the positions that have a mate in one, and have 0-10 possible moves are labeled together, while number of positions of 10-20, 20-30 etc. are also labeled together. By this way, for mate in one positions we see that for example, the size of the search space of 30-40 is $\frac{3.5}{4.5}$ $(\frac{7}{9})$ of the positions labeled 40-50 (when we assume moves with 30-40 label have 35 and 40-50 have 45 possible moves in average). However for mate in two and three positions, we have to take the square or cube of these numbers since this number of moves exists but two or three times in sequence. Using this method, we can quantify the growth of the search space for different positions and compare this growth with the growth of the accuracy performance of each agent. If this growth rate

is very similar to the growth of the performance of the agent, we conclude that they have a linear relationship.

Likewise, we can use the depth as our independent variable. When we use all positions of mate in one, two and three without labeling, we can assume that they all have similar number of possible moves for each position (around 35). Thus using the averages, we can say that for mate in one positions, the size of the search space is 35 while for mate in two, it is 35^2. We see that when the depth is independent, the size grows exponentially. Using this, we can try to look at the accuracy performance rate of the agent for problems of different depths and see if the accuracy grows logarithmically (since the search space grows exponentially, the problem grows exponentially and the growth rate is inversely correlated with the accuracy growth rate of agent, we can expect to see a logarithmic growth).

4 Experimental Setup and Results

Tools and Dataset

The initialization of Chess games and the agents, training and the evaluation as well as the data processing pipelines are all programmed using Python, in Google Collab, a Jupyter Notebook structure where the tasks can be executed faster and in parallel on the cloud. For the Basic game functionality, Pythons "Chess" library is used. This library allows us to set the board in arbitrary positions, making legal moves, showing possible moves and displaying the state of the game (check, checkmate, stalemate etc).

For the deep reinforcement learning algorithm used by the agents, we used Arjan Groen's "RLC" (Reinforcement Learning Chess) repository [3]. His approach is done trying to mimic current state-of-the-art Chess AI's, combining deep neural networks with reinforcement learning and monte carlo tree search. Due to hardware and scientific limitations, his version is not exactly same as the current state-of-the-art versions. The layers of the networks are more arbitrary and there are small differences in the learning methodologies. However, the main principles are very similar and the implementation is done with the same fundamentals. A more detailed explanation of his approach can be found in his Kaggle blogs[4] as well.

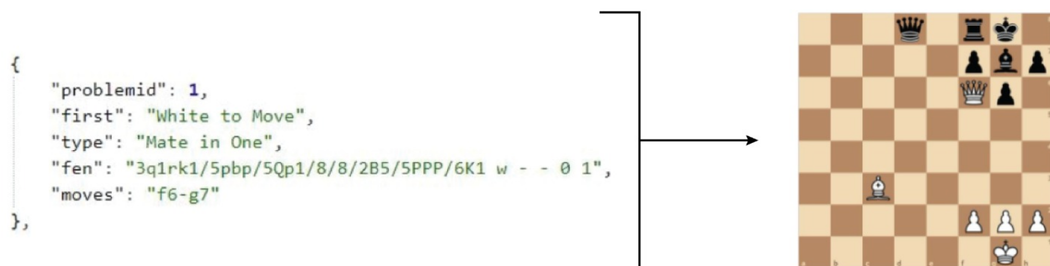

Figure 3: Example data of one position from the data set with a direct translation from the given Forsyth-Edwards Notation to the board position

For the evaluation of the agents, we used the problems that are written by the famous Chess teacher Laszlo Polgar in his book "Chess: 5,334 Problems, Combinations, and Games". Of these 5,334 problems, 4,462 of them are mate in one, two and three positions.

These problems are represented in a JSON format in Daniel Romeo's repository "4462-Chess-problems" [8] in the format shown in figure 3. For each item in the dataset, there is an index, a data storing who's turn it is, a type information representing either it is a mate in one, two or three position. On top of these, there is a FEN (Forsyth-Edwards Notation), which can directly be translated into a Chess board position, visible in Figure 3. Lastly, there is a solution for all possible moves. We don't actually need solution for evaluation since with the help of the Chess framework for Python, we can easily verify if the position is a checkmate and the winner is the first player to move.

Experimental Results

We started the experimentation with just disregarding width size of problems and only focusing on the depth size. This means, we only categorized the problems with respect to how many moves it required until checkmate (mate in one, two or three).

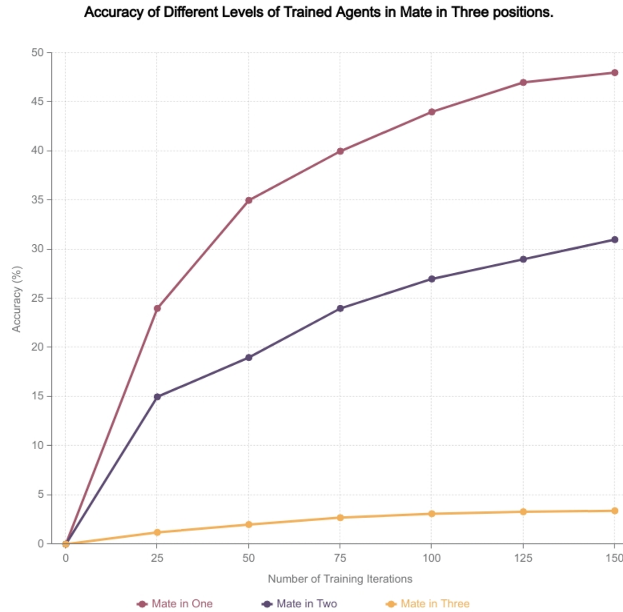

Figure 4: Performance growth of agents in mate in one, two and three positions for different amounts of training

Using this metric, the first thing that is easily visible is that the number of required moves affects the time of learning exponentially. For example, looking at performances at 25 iterations in the Figure 4, we see that the mate in one has an accuracy of 24 percent and mate in two having 15 and while mate in three having around 2 percent. So when going from mate in one to two, the accuracy only goes to around half but from two to three reduces the accuracy by $\frac{1}{8}$. This makes sense, since the number of positions grow exponentially as well. Mate in one positions have an average of 35 moves each. For the mate in two, the number grows exponentially to 1225 (35^2) and mate in three goes to 42,875 (25^3). From these results, we see that the the time needed for learning is not directly linear with the growth of the search space, although they all grow exponentially.

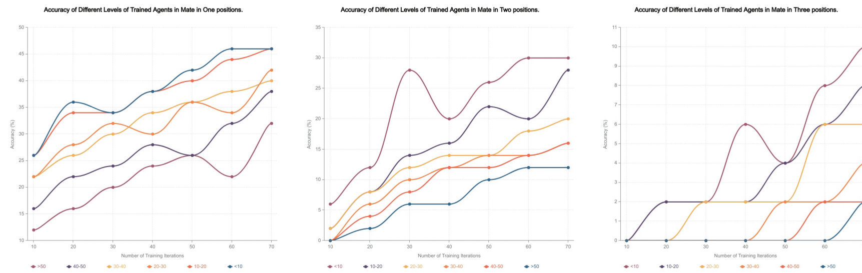

Figure 5: Accuracy of Different levels of trained agents in mate in one, two and three positions, with different sizes of search space. The ranges mentioned under the table are the number of possible moves in the current position

For the second experiment, we further divided all mate in one, two and three positions into number of possible moves (Figure 5). The first thing that pops to the eye, is how much faster the mate in one starts performing better compared to mate in two and three. For all of the labeling in general, mate in one starts performing with an accuracy of 0.1-0.3, and goes up to 0.5 threshold. On the other hand, the mate in two positions starts much lower, around 0.04 but again goes as high as 0.3. Lastly, the mate in 3 starts with 0 accuracy and at the final, can barely hit 0.1.

This is a very concrete representation of the effect of the exponential growth of the search space. Even though the depth grows 1 to 2 to 3, the growth of accuracy does not decrease with 3x to 2x to 1x, with a constant value x. Instead if we look at a concrete point, like the accuracy of the smallest label (0-10) with the best training, in mate in one figure, it has a 0.46 accuracy. When we look at the same data point but for mate in two, we see that the accuracy dropped to 0.3 and for the mate in three, it is 0.1. So we see that for some value x, we say that $x^1 = 0.46, x^2 = 0.3$ and $x^3 = 0.1$. This equation holds for mate in two and three positions, where transitioning from two to three grows the search space to the same degree in which it also increases its performance. However, for the transition between the mate in one to mate in two, the equation has some error and with the accuracy of the mate in one, if the hypothesis was correct, we would assume to have a accuracy of 0.22 whereas it is 0.3.

We repeat this evaluation process for all data points and calculate the average deviation from the theoretical of the hypothesis. The trend we have discussed continues and shows that the growth of the accuracy is exponential. There are some factors which introduce possible errors, and makes the observation difficult, which are explained in the following sections of this article.

One other observation is that, the growth of the accuracy is more constant and predictable for positions that have fewer moves until forced checkmate. One explanation is that, as the number of possible moves grows very hard, the room for luck and stochasticity is introduced into the training of the agent. When the number of possible moves is little, choosing the correct move is more achievable with a good trained model. As the number of possible moves grow, even the best trained model starts having a hard time and thus creates a randomness.

5 Responsible Research

The method we have followed and the experiments we have conducted are easily reproducible. Both the data we used for the evaluation of agents and the deep reinforcement learning algorithm we used are all publicly available also including the Chess library we used as our environment. The tests and the experiments we conducted are documented in detail[Algorithm 1], which involves the training, performance and the evaluation of the agent.

Although it is easily possible to create the same environment and create the same experimental setting, the results can and probably will be different. This is due to the fact that, the agent starts learning by making random decisions. These random initial decisions, shape how it learns the game and can drastically change the pace and performance of its learning. However, if the experiment is again done using a large data set with many iterations, although the results will be different, the trend is going to remain. In other words, it easily possible to conduct the same experimental study and come to the same conclusions but using a slightly different result set.

6 Discussion

The effects of the size of the search space has been considered by many scientists. However, these research was done by comparing different games of different sizes of search space [2]. In general it has been observed that an agent is able to learn to play a game like Checkers faster, compared a Chess or especially Go. From the figure 6, it is possible to see that Checkers has a much smaller search space than Chess or Go. By combining results like these, it is easy to come up with a conclusion stating a larger search space requires more training time.

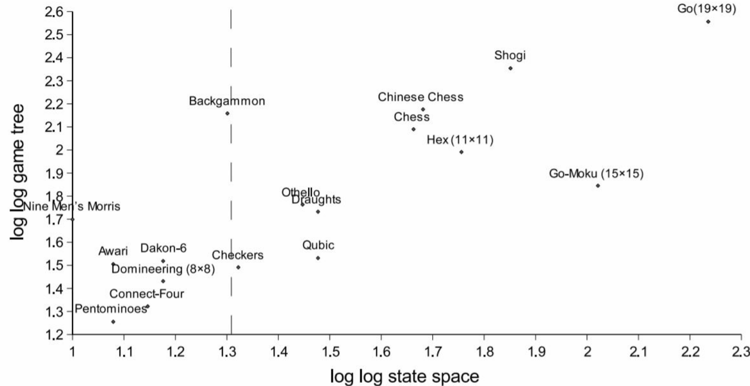

Figure 6: Size of game three and search space for different popular board games [2]

Even though this conclusion may be correct, studies like these are not sufficient to back it up. There are many factors which result in the training iterations needed for playing a game at a certain level. These factors, are discussed in more detail in the introduction section. To be able to make a conclusion like this, somehow, the other factors should be

negated. There are two possible ways of achieving this. One option is that, thousands of different board games should be used, and sorted with respect to their sizes of search space. When the number of board games a much higher, individual differences cancel out and the trend is more clear.

However, this approach has some very clear limitations. Even if there are thousands of different board games that are implemented and ready for use, it is a very labour intensive work to adapt the reinforcement learning agent to play in thousands of different environments, let alone its evaluation metrics. The much simpler and straightforward way is to analyze the same game but considering different positions of different sizes of search space. By this method, it is not possible to get misled by the individual differences in different games. Also, when the size of the search space is the independent variable and can be tracked for every position, it is possible to not only see the trend but also to see the exact factor of its effect.

7 Conclusions and Future Work

Reinforcement Learning is used in situations when there is an environment and inside there needs to be consecutive actions taken. By simulating and learning by trying, the agents finds better and better strategies, trying to improve its performance. This description is very applicable to board games, including Chess and thus, certain board games are used extensively for conducting research in the field. The time it takes the agent to perform well is not same for every domain and the factors determining this imbalance are studied. One main factor is the size of the search space for the given domain. In this research, we focused exactly to this, firstly by verifying that it indeed does affect the time of learning. Then, tried to answer to what extent it affects. For these given problem, we have found that the time needed for training grows exponentially just like the growth of the search space.

Even though, the method we used is very straightforward, there is still some points of improvement. Firstly, for the evaluation of each agent, we only used 50 problems (different set of 50 for different levels of evaluation), while measuring its performance. All the problems that were used for evaluation are not the same difficulty, and when the number of problems used are not very large, it is more likely to have different levels of problems for each set. As the number of problems increase, the sample average difficult approaches the average of all the set and thus different sets have same difficulty. In other words, the results would be more accurate if the evaluation was done using a larger set of problems.

One other point of improvement exists because of the random initial actions made by the agent. In the current procedure, the evaluation is done once for different levels of the agent. However, there is a factor of luck since the agents starts by making random decisions and thus the outcome can be different for different trials. To prevent this, it is more accurate to train the agent to same level many times, evaluate each time and take the average as the performance for that level. By doing this approach, we can get rid of the luck factor we have mentioned.

The conclusions found were descriptive enough for seeing a pattern. Due to many experimental trials and looking at the bigger trend, it is possible to come up with some clear conclusions. However, when looked at the small picture, there are some stochasticity that can be reduced further by following the procedures mentioned above. This would further prove the results and maybe some other conclusions could be made.

References

[1] David Balduzzi, Marta Garnelo, Yoram Bachrach, Wojciech Czarnecki, Julien Perolat, Max Jaderberg, and Thore Graepel. Open-ended learning in symmetric zero-sum games. In *International Conference on Machine Learning*, pages 434–443. PMLR, 2019.

[2] Imran Ghory. Reinforcement learning in board games. *Department of Computer Science, University of Bristol, âŠ*, 2004.

[3] Arjan Groen. Rlc. `https://github.com/arjangroen/RLC`, 2019.

[4] Arjan Groen. Reinforcement learning chess 5: Tree search. *Kaggle*, 2021.

[5] Feng-hsiung Hsu. Ibm's deep blue chess grandmaster chips. *IEEE Micro*, 19(2):70–81, 1999.

[6] Allen Newell, John Calman Shaw, and Herbert A Simon. Chess-playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2(4):320–335, 1958.

[7] Xuewei Qi, Yadan Luo, Guoyuan Wu, Kanok Boriboonsomsin, and Matthew Barth. Deep reinforcement learning enabled self-learning control for energy efficient driving. *Transportation Research Part C: Emerging Technologies*, 99, 2019.

[8] Deinal Romeo. 4462-chess-problems. `https://github.com/denialromeo/4462-chess-problems`, 2021.

[9] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[10] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362, 2018.

[11] Arie van der Stoep. A chess legend. *Board Game Studies Journal*, page 107.