KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Machine Learning

Second cycle, 30 credits

Exploring the effects of state-action space complexity on training time for AlphaZero agents

TOBIAS GLIMMERFORS

Stockholm, Sweden, 2022

Exploring the effects of state-action space complexity on training time for AlphaZero agents

TOBIAS GLIMMERFORS

Degree Programme in Computer Science and Engineering
Date: July 4, 2022

Supervisors: Jeannie He, David Sandberg, Tor Kvernvik
Examiner: Ming Xiao
School of Electrical Engineering and Computer Science
Host company: Ericsson
Swedish title: Undersökning av påverkan av spelkomplexitet på träningstiden för AlphaZero-agenter

Abstract

DeepMind's development of AlphaGo took the world by storm in 2016 when it became the first computer program to defeat a world champion at the game of Go. Through further development, DeepMind showed that the underlying algorithm could be made more general, and applied to a large set of problems. This thesis will focus on the AlphaZero algorithm and what parameters affect the rate at which an agent is able to learn through self-play. We investigated the effect that the neural network size has on the agent's learning as well as how the environment complexity affects the agent's learning.

We used Connect4 as the environment for our agents, and by varying the width of the board we were able to simulate environments with different complexities. For each board width, we trained an AlphaZero agent and tracked the rate at which it improved. While we were unable to find a clear correlation between the complexity of the environment and the rate at which the agent improves, we found that a larger neural network both improved the final performance of the agent as well as the rate at which it learns.

Along with this, we also studied what impact the number of Monte-Carlo tree search iterations have on an already trained AlphaZero agent. Unsurprisingly, we found that a higher number of iterations led to an improved performance. However, the difference between using only the priors of the neural network and a series of Monte-Carlo tree search iterations is not very large. This suggest that using solely the priors can sometimes be useful if inferences need to made quickly.

Keywords

Deep learning, Reinforcement learning, AlphaZero, Monte-Carlo tree search, Environment complexity

Sammanfattning

DeepMinds utveckling av AlphaGo blev ett stort framsteg året 2016 då det blev första datorprogrammet att besegra världsmästaren i Go. Med utvecklingen av AlphaZero visade DeepMind att en mer generell algoritm kunde användas för att lösa en större mängd problem. Den här rapporten kommer att fokusera på AlphaZero-algoritmen och hur olika parametrar påverkar träningen. Vi undersökte påverkan av neuronnätets storlek och spelkomplexiteten på agentens förmåga att förbättra sig.

Med hjälp av 4 i rad som testningsmiljö för våra agenter, och genom att ändra på bredden på spelbrädet kunde vi simulera olika komplexa spel. För varje bredd som vi testade, tränade vi en AlphaZero-agent och mätte dens förbättring.

Vi kunde inte hitta någon tydlig korrelation mellan spelets komplexitet och agentens förmåga att lära sig. Däremot visade vi att ett större neuronnät leder till att agenten förbättrar sig mer, och dessutom lär sig snabbare.

Vi studerade även påverkan av att variera antalet trädsökningar för en färdigtränad agent. Våra experiment visar på att det finns en korrelation mellan agentens spelstyrka och antalet trädsökningar, där fler trädsökningar innebär en förbättrad förmåga att spela spelet. Skillnaden som antalet trädsökningar gör visade sig däremot inte vara så stor som förväntad. Detta visar på att man kan spara tid under inferensfasen genom att sänka antalet trädsökningar, med en minimal bestraffning i prestanda.

Nyckelord

Djupinlärning, Förstärkande inlärning, AlphaZero, Monte-Carlo tree search, spelkomplexitet

Acknowledgments

I would like to thank my supervisors at Ericsson who have helped me during my work on this thesis. Tor Kvernvik and David Sandberg, have helped me come up with ideas for me to explore and have given me access to some of Ericsson's resources, among other things. I would also like to thank my supervisor at KTH, Jeannie He, who has helped me with some of the formalities regarding this thesis and helped to give feedback.

Stockholm, July 2022
Tobias Glimmerfors

Contents

List of Figures

List of Tables

List of acronyms and abbreviations

MCTS	Monte-Carlo tree search
USCF	United States Chess Federation

Chapter 1

Introduction

This chapter introduces the main topics that this thesis will cover and describes the problems related to them. This chapter also describes the purpose and goals, and outlines the structure of the thesis.

1.1 Background

In the spring of 2016 the DeepMind's AlphaGo [1] became the first computer program to defeat the world champion in a match setting in the game of Go on a 19×19 board. AlphaGo combined supervised and unsupervised learning to train a neural network capable of predicting which moves are most likely to be good, when provided with a board state. This neural network, coupled with a Monte-Carlo tree search algorithm outperformed any pre-existing algorithm and gained world-wide attention for its success.

In 2017 DeepMind released a paper describing AlphaGo Zero [2], which was able to master Go completely through self play. AlphaGo Zero built on the methods of the AlphaGo paper, and after 72 hours of training, it managed to beat a version of AlphaGo which had been trained for several months by a score of $100 - 0$ [2].

Further advances also came in 2017 when DeepMind released a new paper titled *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm* [3]. In the paper, they showed that the same methods which had been used to train AlphaGo Zero could be used on a variety of games, rather than being restricted to only learning to play Go. These advances showed great success, giving hope to solving problems previously thought to be too hard for classical algorithms.

1.2 Problem

The AlphaZero [3] algorithm from DeepMind has shown that an agent can learn good policies for a variety of environments. However there seems to be little information available on how the training is affected by different parameters during training. While DeepMind has shown that their agents could learn to play games such as Go, shogi and chess, these games are vastly different, making it difficult to get a general sense of what aspects of the games make them easier or harder to learn. This thesis aims to investigate the effect that some key parameters have on the training time and strength of an AlphaZero agent.

One of the key factors which is likely to affect the training of an agent is the complexity of the environment, with an agent likely needing more time to learn a more complex environment. In addition to this, there are some other hyperparameters which are interesting for AlphaZero in particular, such as the neural network which is used and the number of iterations of the Monte-Carlo tree search algorithm.

1.2.1 Original problem and definition

This thesis aims to answer the following questions:

- How does the state-action complexity of an environment affect the rate at which an AlphaZero agent learns?
- How does the size of the neural network affect the rate at which and AlphaZero agent learns and its playing strength?
- How does the number of Monte-Carlo tree search iterations affect the strength of an already trained AlphaZero agent?

1.2.2 Scientific and engineering issues

The main engineering issues of this project are to implement a fast-running Monte-Carlo tree search (MCTS) algorithm as well as a a good training pipeline for collecting training data.

1.3 Purpose

The purpose of this project is to investigate the effect that key hyperparameters have on the training of an AlphaZero agent. With this, we hope that future

researchers can understand the agents better when modelling a problem, allowing them to choose a good set of parameters.

1.4 Goals

The goal of this project is, primarily, to implement an MCTS algorithm which is able to run on a GPU to allow for better performance. Along with this, another goal is to optimize the code for the algorithm as much as possible in compute-heavy areas to make training feasible without the need for an excessive amount of resources.

1.5 Research Methodology

In order to carry out experiments, part of this project consists of a complete implementation of the AlphaZero algorithm. While there are some open source implementations such as LeelaZero [4] which could be used and modified, this may make customisation difficult and provide us with less control during training. Using a custom implementation of the AlphaZero algorithm, we will conduct several training runs from which we will be able to gather the data necessary to answer our research questions.

1.6 Delimitations

While this projects is deeply connected to the AlphaZero [3] algorithm by DeepMind, there will be no further advances to the algorithm itself in this thesis. The work in this thesis will instead consists of an implementation of the original algorithm, which will be the basis of all experiments.

Additionally, this thesis will only discuss sustainability and ethics to a limited degree. The are two main reasons for this. Firstly, this project consists mainly of experiments which are simulated with the help of code. Secondly, the AlphaZero [3] algorithm, which we use throughout our experiments, was developed several years prior to our work. As a result of this, sustainability and ethics become less relevant for this project.

1.7 Structure of the thesis

This thesis is structured into 6 chapters, excluding this chapter. First, we will explain the core concepts needed to understand our work and give some details regarding previous work. Thereafter we will explain what methods were used to gather results and how our research was conducted. We will also provide some implementation details which may provide useful if attempting to recreate our results.

Towards to end of this thesis we will present our results along with an analysis of them. The thesis then ends with a discussion chapter, and finally a conclusion.

Chapter 2

Background

This chapter provides background information about the topics covered in this thesis. It will start by defining the key terms which will used throughout the thesis. Afterwards, some previous work will be described.

2.1 Elo

Elo is a system which assigns numerical rating values to players as an estimate of player strength, with higher values corresponding to a better playing strength [5]. The strength of two players, A and B, are approximated by R_A and R_B respectively, and in a match between players A and B, the expected score of player A is calculated according to the following formula [5]:

$$E_A = 1/(1 + 10^{(R_B - R_A)/400})$$

For a game where there are no draws, this is equivalent to the probability that A wins against B. Note that only the absolute difference between R_A and R_B is important for this calculation. As a result, all ratings could be adjusted by adding or subtracting a fixed amount x to all players without invalidating any individual calculation.

When applied to a game played between human players, one common approach for mainting Elo ratings is to assign new players a fixed rating before their first game. This is, for example, done on major chess websites such as chess.com [6] and lichess.org [7].

To update player ratings, there are numerous algorithms which exist. Here we summarise the system used by the United States Chess Federation (USCF), as described in [5]. In the system used by the USCF, ratings are adjusted at the

end of a tournament to more closely match the observed results. The formula for the adjustment of a player's rating is given as [5]:

$$r_{post} = r_{pre} + K(S - S_{exp})$$

where r_{post} gives the updated rating of the player, r_{pre} gives the pre-tournament rating of the player, S is the player's total score in the tournament, S_{exp} is the expected total score of the player in regards to the opponents that the player faced, and K is a factor controlling by how much the rating should be adjusted.

In general, this formula, as well as other formulas employed by other Elo rating systems work under the principle that a player who is performing at a strength above their rating should increase their rating. Similarly, a player that is performing at a strength below their rating should decrease their rating.

One advantage of the Elo system used by the USCF is that it allows for updates in the ratings of players if they have improved or gotten worse. For an algorithms used by a computer program however, it should be obvious that the associated Elo rating should remain fixed, as long as all parameters affecting the computer remain fixed. As such, other methods for calculating Elo may be desired. In Section 2.1.1, we briefly discuss a method for approximating the Elo ratings of set of players, based only on the outcomes of a set of matches.

2.1.1 Bayesian inference

One idea in Estimating the Elo from a list of results is to calculate the ratings of the players which maximise the likelyhood of the results which were observed. i.e. maximising [8]

$$p(\gamma|\boldsymbol{G}) = \frac{P(\boldsymbol{G}|\gamma)p(\gamma)}{P(\boldsymbol{G})}$$

where γ is the ratings of the players, $\boldsymbol{G}$ is the observed game results. $p(\gamma)$ is the prior distribution which is assumed to follow a normal distribution, and $P(\boldsymbol{G})$ is a normalizing factor.

In this paper, we will make use of a pre-existing program, BayesElo [9], for Elo estimations as outlined in their paper [8].

2.2 AlphaZero algorithm

In this section, we will give a detailed description of how the AlphaZero algorithm works. The information in this section consists mainly of a summary of the methods used in the AlphaGo Zero [2] paper as well as the AlphaZero [3]

paper. We have altered some minor parts of the explanation in order to more closely match our implementation, but the definitions which we bring up in this section come from the papers by DeepMind [2], [3].

At a high level, AlphaZero consists of two main components: a neural network and a MCTS algorithm. The neural network, denoted as f_θ with parameters θ, takes a representation s of a board position along with its history as input, and outputs a vector of move probabilities as well as a value: $(\boldsymbol{p}, v) = f_\theta(s)$. The move probabilities $\boldsymbol{p}$ represent the probabilities that the agent will select any given move, i.e. $\boldsymbol{p}_i$ is the probability of selecting move i. The value v represents the expected outcome of the game, where a value of 1 represents a win for the first player, a value of -1 represents a win for the second player and a value of 0 represents a draw. The composition of the neural network is outlined Chapter 3.

Given a state s, AlphaZero uses a series of MCTS iterations together with the neural network to produce an improved vector of move probabilites $\boldsymbol{\pi}$. MCTS iteratively builds a search tree, exploring the set of possible moves. This search tree consists of nodes representing board states s, and edges (s, a) representing moves which can be taken in state s. Each edge in the tree stores a prior probability $P(s, a)$, a visit count $N(s, a)$, an action-value $Q(s, a)$ and a total action value $W(s, a)$. The variant of Monte-Carlo tree search which is used is by AlphaZero can be divided into 3 main phases, outlined in Section **??**.

2.2.1 Monte-Carlo tree search phases

2.2.1.1 Selection

The first phase of an MCTS iteration consists of a simulation started in root of the search tree s_0. At each time step t, a move a_t is chosen among the possible moves at state s_t. If the resulting state of applying action a_t at state s_t exists in the search tree, we traverse the edge (s_t, a_t) and reach the resulting state s_{t+1}. This is repeated until we select an action which we have not previously explored, which would result in a state s_L that does not exist in the search tree. It it worth noting that s_L does not need to be a terminal state of the game. When selecting a move a_t, it is chosen according to the following formulae, taking the move statistics saved in the tree into account:

$$U(s, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

$$a_t = \underset{a}{\text{argmax}}(Q(s_t, a) + U(s_t, a))$$

where c_{puct} is a constant which determines the level of exploration. Intuitively this prioritises moves with a high prior and low visit count at first but asymptotically prefers moves with a higher action value.

2.2.1.2 Expansion

After selection, the unexplored node s_L is added to the tree along with new edges (s_L, a) for each legal move a at state s_L. Using the neural network, we calculate $(\boldsymbol{p}, v) = f_\theta(s_L)$ and initialise the edges (s_L, a) to $\{N(s_L, a) = 0, W(s_L, a) = 0, Q(s_L, a) = 0, P(s_L, a) = pn_a\}$, where pn_a is a normalized probability vector of a such that the sum of the probabilities for the legal moves sum to 1. After this is done, the value v is backed up.

2.2.1.3 Backpropogation

For each of the edges $(s_t, a_t), t < L$ traversed during the selection phase, the edge statistics are updated as follows:

$$N(s_t, a_t) = N(s_t, a_t) + 1$$

$$W(s_t, a_t) = W(s_t, a_t) + v$$

$$Q(s_t, a_t) = \frac{W(s_t, a_t)}{N(s_t, a_t)}$$

After a series of MCTS iterations the visit count of the root node is used to construct an improved probability vector $\boldsymbol{\pi}_a = \frac{N(s_0,a)}{\sum_b N(s_0,b)}$. AlphaZero then plays a move a with a probability proportional to $\boldsymbol{\pi}_a^{1/\tau}$, where τ is a temperature parameter controlling the level of exploration. Setting the value of τ close to 0 will result in greedy play where the agent selects the most visited action, whereas a higher temperature value allows for more exploration.

2.2.2 Parameter optimisation

In order to optimise the parameters θ of the neural network f, AlphaZero continuously plays games against itself, selecting a move according to the MCTS procedure outlined above. During the beginning of the game, a temperature of 1 is used to allow for some exploration. Thereafter a temperature value arbitrarily close to 0 is used for the temperature. During a single game, each of the states s_i which arise on the board are saved together with $\boldsymbol{\pi}(s_i)$. At the end of the game, the result of the game $r \in \{-1, 0, 1\}$ is

recorded. Each of the positions s_i are then added to a replay buffer as the inputs with expected outputs $(\boldsymbol{\pi}(s_i), r)$. The parameters θ of the neural network are optimised through stochastic gradient descent, with batches $(s, \boldsymbol{\pi}, z)$ sampled uniformly from the replay buffer, to minimize the loss as per:

$$(\boldsymbol{p}, v) = f_\theta(s)$$

$$l = (z - v)^2 - \boldsymbol{\pi}^T log p + c||\theta||^2$$

I.e. the neural network is trained to minimize both the mean squared error between the predicted value v and the recorded values as well as to minimize the cross entropy loss between the predicted probability vector $\boldsymbol{p}$ and the improved probability vector $\boldsymbol{\pi}$. On top of this there is also L2 weight regularisation to prevent overfitting, with a parameter c controlling the amount of regularisation.

2.3 Related work

In this section, we present some previous work related to our thesis. There seems to be little research regarding the effects of the environment complexity on training time. Therefore, this section will instead focus on work that deals with large action-spaces as well as work building on AlphaZero.

2.3.1 Procgen Benchmark

In the paper *Leveraging Procedural Generation to Benchmark Reinforcement Learning* [10], 16 open source environments were created for the benchmarking of reinforcement learning algorithms. The aim of the benchmark is to see how well algorithms are able to generalise, rather than memorizing specific trajectories.

They used these environments to investigate the effects of scaling of the model size in RL. They found that larger architectures significantly improve both sample efficiency and generalization.

2.3.2 Muzero

Muzero [11] is an algorithm similar to AlphaZero. Developed by DeepMind, it encorporates many of the same techniques, including a nerual network and an MCTS algorithm. The main change introduced in Muzero is that the agent is

able to learn policies for environments without the need for a transition model, instead using a neural network which learns which transitions are possible.

2.3.3 Sampled Muzero

The introduction of Sampled Muzero [12] proposed a way for RL-agents to learn policies in environments with a large action space without the need to sample all actions. It builds on the methods of MuZero [11] showing that, with modification to the algorithms, algorithms such as MuZero can be used to learn highly complex environments. In addition to this, Sampled Muzero is also able to learn a policy for continuous environments, something which AlphaZero and MuZero are not able to do.

2.3.4 Offline learning with Muzero

In the 2021 paper titled *Online and Offline Reinforcement Learning by Planning with a Learned Model* [13], the authors show that it is possible to learn in an offline manner using the Muzero model. Due to there being a large number of similarities in the AlphaZero and Muzero algorithms, this would suggest that AlphaZero can also be used in offline learning. For a real-world use case this could prove very useful as data collection could be made separate from the training of the model, possible speeding up training.

2.4 Summary

In summary, there has been prior work which shows that algorithms such as AlphaZero can be used even in complex environments. However, there seems to be very little investigation into how the complexity of an environment affects the learning ability of the agents. This motivates the need for this thesis and further research.

Chapter 3

Methods

The purpose of this chapter is to provide an overview of the research method used in this thesis. Since this project builds upon the results of primarily AlphaZero [3], many of the methods in this project take inspiration from their paper.

Section 3.1 describes what factors we consider for comparing the complexities of two different environments. Section 3.2 describes what environments we used for the training of the agents. Section 3.3 details the architecture of the neural network used. Section 3.4 explains how training-data was gathered, what hyperparameters were used and why. Section 3.5 describes how our results were gathered from training runs and how they were analysed.

3.1 Complexity of environments

There are numerous ways to measure the complexity of an environment. In this report, we are interested in both the size of the state space as well as the size of the state-action space. Here we define the size of the state-space as the number of reachable states, starting from an initial state. We define the size of the state-action space as the number of pairs (s, a), such that a is an action that can be taken in state s, summed over all reachable states $\mathcal{S}$. Since it can sometimes be difficult to calculate the exact size of the state space or the state-action space, it can also be interesting to look at the size of the action space. We define this as the total number of unique actions which are possible in at least one state each.

Throughout this thesis, we do not include any exact calculations for the sizes of the state space or the state-action space. We also do not include any

approximations of these metrics, nor any upper bounds etc. The reason for this is that we are interested in ranking the environments according to their overall complexity. As explained in Section 3.2, we are able to do this trivially without the need of any calculations.

3.2 Environment

For our purposes, we want to be able to train an AlphaZero agent on similar, yet non-identical, environments where we can vary the complexity of the environment. For this, we chose the board game Connect 4. Connect 4 has been proven to be a theoretical win for the first player by the likes of James Dow Allen. However, there seems to be no trivial strategy applicable to all states, and the game is complex enough such that training a reinforcement learning based agent remains interesting.

Connect 4 is also a game which can be modified easily to alter the complexity of the game. For example, by increasing the height of the board, the state space and state-action space complexities increase while the action space remains fixed. Alternatively, by increasing the width of the board, the state space and state-action space complexities once again increase, however this also increase the size of the action space.

Moreover, using a two-player board game of perfect information makes the training of agents more straightforward. It is possible to use AlphaZero on other types of environments, however for those use-cases slight modifications need to be made to the algorithm as it was originally used for board games.

Since we want to be able to alter the overall complexity of the game, we trained separate agents for different board sizes by varying the width of the board. In total 8 agents were trained, each one on a different board size. The height of the board was kept constant at 6 to match that of the original game while we varied the width, using each of the widths: $4, 5, 6, 7, 8, 9, 16, 32$.

3.3 Neural network composition

We make use of the same neural network type as outlined in AlphaGo Zero [2]. We make use of the same type of layers, but have decreased the number of layers and the number of filters as the environments we train on are arguably less complex than Go. The model takes, as input, a matrix with dimensions $(batch\ size, 3, 6, w)$, where w is the width of the board. I.e. for each batch element, 3 planes of the same size as the board are stacked together. The first

of these planes contains a value of 1 at each location occupied by player 1. The second plane contains a value of 1 at each location occupied by player 2. The rest of the values in the first two planes are left at 0. The third plane consists of a value of 1 at each location if it is player 1's turn. Otherwise it consists of a value of -1 at each location. In AlphaGo Zero [2] the number of planes is greater to accommodate for a brief history of the game. In games such as Go and chess, this is necessary as the entire state of the game cannot be deduced solely from a board position. This is not the case in Connect 4, and therefore we can resort to having 3 planes of input. The neural network we use consists of a main body, whose output is fed to a policy head and a value head to produce the prior and value predictions respectively.

The layers of the main body of our neural network are as follows:

- A convolutional layer with 64 filters of kernel size 3×3.
- A batch normalization layer.
- A ReLU layer.
- 8 residual blocks, each consisting of:
  - A convolutional layer with 64 filters of kernel size 3×3.
  - A batch normalization layer.
  - A ReLU layer.
  - A convolutional layer with 64 filters of kernel size 3×3.
  - A batch normalization layer.
  - A skip connection, adding the input to the output of the previous layer.
  - A ReLU layer.

The output of this is fed into a policy head and a value head producing the priors and values respectively. The policy head consists of the following layers:

- A convolutional layer with 2 filters of kernel size 1×1.
- A batch normalization layer.
- A ReLU layer.
- A fully connected layer producing an output with dimensions $(batch\ size, w)$ representing the logit probabilites of each move.

The value head consists of the following layers:

- A convolutional layer with 1 filter of kernel size 1×1.
- A batch normalization layer.
- A ReLU layer.
- A fully connected layer producing an output with dimensions $(batch\ size, 256)$.
- A ReLU layer.
- A fully connected layer producing an output with dimensions $(batch\ size, 1)$.
- A tanh layer producing a scalar in the range $(-1, 1)$.

3.4 Training details and hyperparameters

For the training of the agents, we used many of the same parameters outlined in AlphaGo Zero and AlphaZero. Using randomly initialised neural network parameters, we continuously simulated self-play games. For the first 6 half-moves of the game we used a temperature set to 1 in order to get a variety of openings. Thereafter we greedily selected the action with the highest visit count. This is similar to choosing a temperature value arbitrarily close to 0 but has the added benefit of avoiding numerical errors. At the end of each game, each of the encountered positions were added to a replay buffer along with the improved move probabilities and the outcome of the game. The replay buffer stores the $100,000$ most recent positions encountered during training. Each time the end of a game is reached, a random batch of 2048 positions is sampled from the replay buffer, and gradient descent is performed on this batch. Each 5000 training iterations a checkpoint file is saved containing data such as the neural network parameters and the state of the optimizer.

We used an Adam optimiser with the same learning rate which is used as in AlphaGo Zero. For the first 4×10^5 training iterations, the learning rate is kept at 10^{-2}. The learning rate is decreased by a factor 10 after 4×10^5 and 6×10^5 and training iterations. Thereafter it is kept at a constant value of 10^{-4}.

Just as in AlphaGo Zero, we also add Dirichlet noise to the priors of the root node to increase exploration. Using a value of $\epsilon = 0.25$ we set the priors of the root:

$$(\boldsymbol{p}, v) = f_\theta(s_0)$$

$$\boldsymbol{p} = (1 - \epsilon)\boldsymbol{p} + \epsilon \cdot noise$$

Here noise is a vector of the same size as p sampled from a Dirichlet distribution with a value $\alpha = 10/w$, where w denotes the width of the board since that is the size of the action space.

3.5 Evaluation

In order to evaluate the performance of our agents after(and during) training, we make use of the saved checkpoint files(called players here for simplicity). By having the players play against each other in a tournament, we are able to estimate the Elo ratings of the players accordingly. Each pair of opponents facing each other play 200 games with each color. Noise is added to the root nodes during each MCTS iteration, but moves are greedily selected according to visit count rather than using the temperature from before.

Ideally, a round robin tournament would be simulated, where each player faces each other player a large number of times. However, this leads to a very large number of games needing to be played, and as a result we need to filter out some of these matchups. This is done by only letting two players face each other if the number of training games which seperate them is less than a certain threshhold. In our experiments we set this threshhold such that each player faces at most 24 opponents. This will result in less confident values for the Elo ratings but should not affect them too much. As stated by [14], many matchups from a round robin tournament would not be interesting either way since they would be too lop-sided. We are thus effectively operating under the assumption that checkpoint files created after a similar amount of games should hold a similar Elo rating, while checkpoint files created with a large gap will have a larger Elo gap and thus their matchups will not be as interesting.

The results of this tournament are saved to a file and imported into the BayesElo [9] program. This is the same program which was used by AlphaGo Zero to compute the Elo ratings.

The only factor which makes two players play differently are the parameters of the neural networks that they use. I.e. two distinct checkpoint files will contain different neural network parameters. Calculating the ratings of each of the players is thus similar to calculating a relative rating of the agent at different times during training. This allows us to plot the Elo of our agent over time, thereby allowing us to see what progress the agent is making - a faster increase in Elo would suggest that the agent is learning to play faster.

Since BayesElo only calculates relative Elo ratings, we transform all ratings outputed by the program by a constant such that the randomly initialised neural network has an Elo rating of 0. Comparing the graphs of

Elo over time allows us to see if there is a correlation between the board game size and the rate of improvement of the agents.

Chapter 4

Implementation Details

This chapter outlines some of the decisions which were made in our code implementation and why they were made. We will first explain some of the issues which we faced during early stages of development. Thereafter we will outline how we were able to overcome these challenges.

4.1 Issues with initial implementation

One of the main aspects to consider when implementing a version of AlphaZero is speed of training. Since a large number of training games need to be played in order for the agent to become good at the game, a fast implementation is crucial. During the early phases or development, a single game was played at a time, using PyTorch for the neural network. The representation of the search tree for MCTS was done through instances of Python classes, one instance being necessary for each node and one for each edge. This however proved to be too slow, requiring several seconds to compute a single move.

AlphaGo Zero [2] incorporates multithreading into the MCTS, conducting multiple tree searches in parallel. Several techniques for using multithreading in MCTS are described in [15]. Three types of parallelization techniques are brought up; namely leaf parellelization, root parallelization and tree parallelization. Each of these techniques have their downsides. Primarily they are more difficult to implement than a standard MCTS algorithm, especially when a very large speedup is required. Considering this, we went for another type of parallelism: we simulate a batch of independent games in parallel. Since none of the games need to share information, this method can be implemented without the need for locks and mutexes etc. This method also

scales well with the number of threads since each thread can be used to compute moves for a separate game.

The implemenation of our code was to a large extent written using Jax as the main framework. Jax has the advantage that it can perform fast matrix operations on the GPU without much modifcation to the code. For example, expressing the move-making logic in terms of matrix operations can thus speed up the code drastically as a batch of moves can be computed in parallel on the GPU. Using this, we conduct training by simulating a batch of 256 self-play games in parallel on each of our available GPUs. Unless otherwise specified, our experiments used 5 GPUs for training. Moves are made on all boards simultaneously, and once a game finishes, it is added to the replay buffer, after which a new game is started. Running on 5 GPUs, this approach allows for roughly $20,000$ training iterations per hour, thereby requiring roughly 2 days to run $800,000$ training iterations.

The neural network was implement using Haiku, a neural network framework which supports Jax functionality. For training, an Adam optimizer from the optax library was used with a learning rate schedule described earlier but shown in Table 4.1 for clarity.

Thousans of steps	Learning rate
$0 - 400$	10^{-2}
$400 - 600$	10^{-3}
> 600	10^{-4}

Table 4.1: Learning rate schedule used during training.

4.2 Mone-Carlo tree search implementation

In this section we explain some of the details behind our MCTS implementation which allows it to run efficiently with a batch of games on a GPU. Our implementation draws inspiration from some of DeepMind's work on MCTS such as Mctx, part of the DeepMind Jax Ecosystem [16]. Therefore our implementation shares a lot of the same core ideas and functionality as Mctx.

One of the first ideas is that in a lot of implemenetaions of MCTS, objects are created dynamically during each of the tree search iterations. However, by knowing the number of tree search iterations ahead of time, we know the amount of space that needs to be allocated to save all edge data such as visit

counts. This allows us to preallocate any space that we will use. This is done by creating a matrix for each separate variable that we need to keep track of across edges in the search tree. This means that we need to store one matrix for the visit counts, a separate matrix for the action values etc. The dimensions of the first two axes of the matrices are equal to the batch size and the number of MCTS iterations respectively. For some of the matrices, an additional axes needs to be added with a size equal to the number of actions. More specifically, consider the search tree that is being created. The number of nodes in this tree is at most equal to the number of search iterations, with fewer nodes being possible in the event of a terminal node being visited more than once. In our version, these nodes are saved as a matrix of dimension (batch size, #search iterations). The entry in the matrix at index (i, j) gives the state which was added to the search tree during the jth search iteration of the ith game. Thus, for example if a new node is to be added, instead one simply needs to update the entry at a specific index.

In a similar way, there are at most $\#iterations \times \#actions$ edges in a single search tree. As a consequence, we need to preallocate a matrix of dimension (batch size, #search iterations, #actions) for each entry that is saved in an edge. As an example, the entry at index (i, j, k) of the action value matrix represents the action value of taking action k in the state saved at index (i, j) in the state matrix.

In addition to this, we also need to save the transitions between states such that we are able to traverse edges. This is done by saving an additional matrix of dimension (batch size, #search iterations, #actions). The entry at index (i, j, k) represents the index at which we have saved the state which would result from applying action k to the jth state of the ith game.

Chapter 5

Results and Analysis

In this chapter, we present our results and analyse them. We will start by explaining some points about how we analyse the results to make it clear to the reader what we look at. Thereafter, we will present some results to verify that our code works, followed by a presentation of the results which we have divided up into three section. Section 5.3 covers the results of our research regarding the effects of the environment complexity along with an analysis of the results. Section 5.4 covers the results of varying the neural network size along with an analysis of the results. Lastly, Section 5.5 we present our results for varying the number of Monte-Carlo tree search iterations along with an analysis of the results.

5.1 Introductory points

A large amount of the results which we present in the later portions of this chapter concern the Elo ratings of the agents which we train. Any time we compute the Elo ratings of two or more agents we do this by making use of the BayesElo [9] program. As mentioned in Chapter 2, we adjust all of the outputed ratings by a constant k. This constant is chosen independently for each training run such that the lowest rating achieved during training is 0. This allows us to more easily judge the rate of improvement of our agents.

We also want to stress that the aim of this thesis was, at no point in time, to develop the best playing AI for the game Connect 4. As such it is not entirely relevant what exact rating changes are observed for our agents. Instead we will be looking at the rate of improvement among our agents, as we find that this should generalise to a greater extent when looking at different types of environments. This also has the added benefit of being able to compare agents

which were trained in different environments i.e. different board widths.

It should also be noted that the Elo ratings which we compute for our agents can only be compared directly to agents which were trained in the same environment. I.e. the strength of an agent can only be compared to the same agent at different times during training for a fixed board width. This is because there is no natural way to have an agent play against an agent which was trained on a different board width, thus there is no obvious way to establish relative ratings. This however is not an issue in our work, since we are interested in the rate of improvement rather than the playing strength of our agents.

5.2 Validation of our results

In order to make sure our main algorithm worked, a lot of inspection was made into what was happening during training. For example, we tested that the MCTS algorithm converged to the theoretical best moves when many iterations were allowed. This was done by selecting states which were close to the end of the game in order to manage the size of the game tree from each position. Further tests were also conducted to verify that each of the games which is played consists of a legal series of moves. This was done partly by manual inspection of a large set of played games along with a script that checked it.

Before conducting most of our training runs, we first conducted a smaller run on the game with a width of 7. Since this is the standrad width, it allowed us to later test the performance of our agent against an online AI. By doing so, we were able to verify that one of our trained agents could occasionally win against a theoretically perfect AI if our agent was allowed to move first. In our experiments our trained agent won, lost and draw roughly and equal portion of test games against the perfect AI.

For the plots we present in this chapter in which we show the Elo change of our agents over time, we have simulated a tournament between each version of the agent. In Chapter 3 we explained in greater detail how many matches the tournament contains. We stated that we did not need to simulate a full round robin tournament and are able to save computation time by not doing so. A round robin tournament would likely provide slightly more accurate results than the tournaments which we simulate. However, as we explained, the difference between the results of these tournaments should not be large. This is further backed up Figure 5.1 and Figure 5.2. Figure 5.1 shows the Elo rating change when using a partial tournament. Figure 5.2 shows the Elo rating change when using a full round robin tournament. As the plots show, there is very little difference. To make it even more obvious that they yield

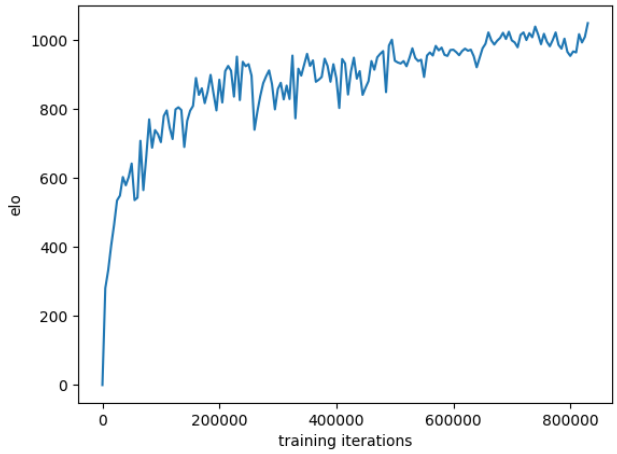

Figure 5.1: Elo change of agent during training, evaluated with incomplete round robin tournament. A player created after t training iterations faced each player created within $60,000$ training iterations of t, 400 times.

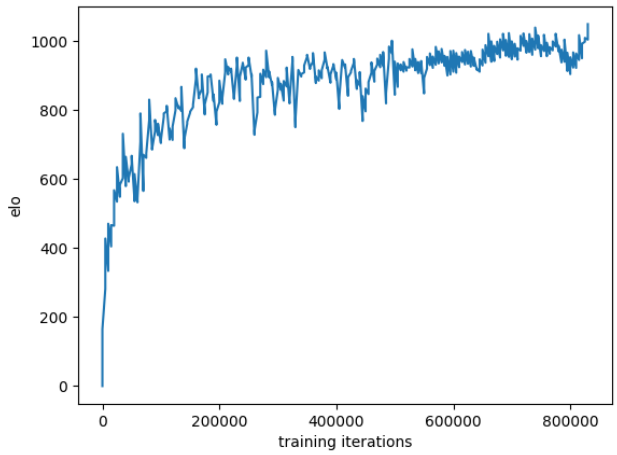

Figure 5.2: Elo change of agent during training, evaluated with a full round robin tournament. Each pair of players faced each other 400 times, each player starting an equal number of games.

nearly identical results, Figure 5.3 shows a joint composition of these plots.

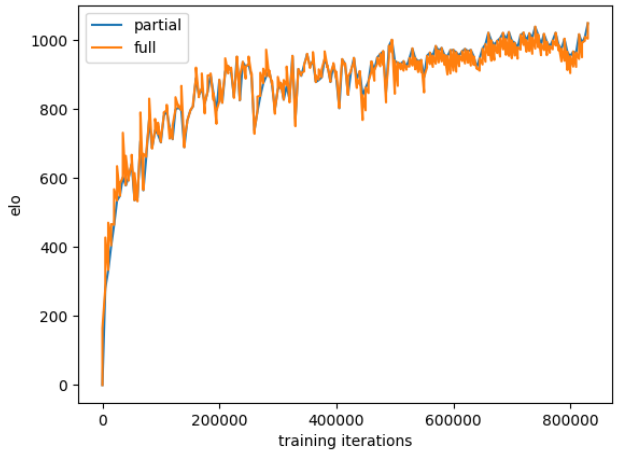

Figure 5.3: A joint composition of Figures 5.1 and 5.2.

5.3 Environment complexity

In this section we present the results from all our training/evaluation runs which explore the effects of a change in the state-action complexity. In total, 8 different board game sizes were used. We will first present how well each of our agent performed in each of the environments and thereafter, we will compare them.

5.3.1 Width 4

Avoiding a loss in a game of Connect-4 with a board width of 4 is nearly trivial for most board states. In a majority of cases it is as simple as checking whether the opponent can win on their next move if you were to make a move, and if so, make another move. Forcing a win on the other hand is seemingly hard as you often need to create two simultaneous threats in order to win. Due to the small size of the board this becomes a challenge. As a result of this, the amount of knowledge which the agent can learn is not very high. This is reflected in Figure 5.4. As shown, the Elo rating of the agent rapidly jumps up to approximately 370 and stays nearly constant throughout the rest of the training. This shows that the agent is quickly able to gather some knowledge about the environment, but thereafter there is little progress to be made.

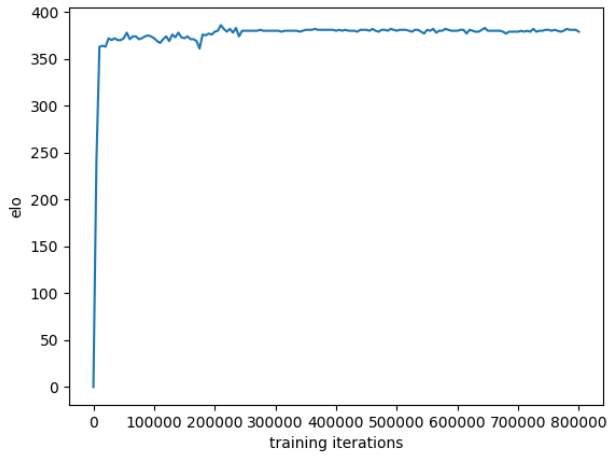

Figure 5.4: Improvement over time with a board of width 4. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

5.3.2 Width 5

Figure 5.5 shows the improvement of the agent over time for an agent trained on a board of width 5. Again, we see a very rapid increase in Elo rating at the start, which represents a quick rate of learning. However, this time a slight increase of about 100 Elo points can be observed during the first $300,000$ training iterations, when disregarding the initial jump. This seems to suggest that the agent is able to learn more knowledge about the game when a width of 5 is used, when compared to that of a width of 4. The rate at which the agent learns is very rapid during the start of training, but slow throughout the rest of the training.

5.3.3 Width 6

Figure 5.6 shows the improvement of our agent over time when training on a board of width 6. We once again notice an initial jump in rating. Thereafter the agent is able to improve its Elo rating by approximately 200 points (from 600 to 800) in under $200,000$ training iterations. A further 200 point improvement is achieved throughout the rest of training, although this requires an additional $800,000$ training iterations to accomplish.

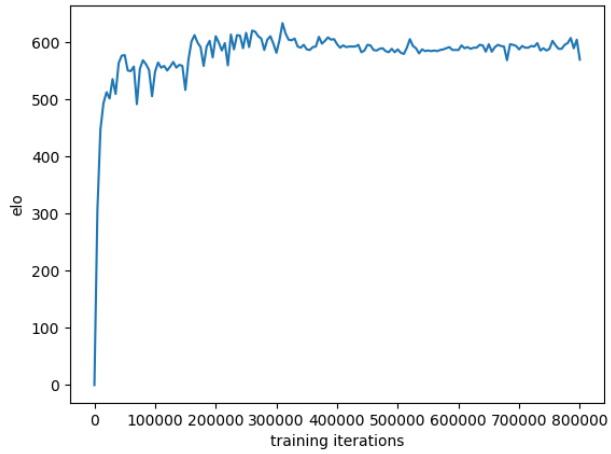

Figure 5.5: Improvement over time with a board of width 5. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

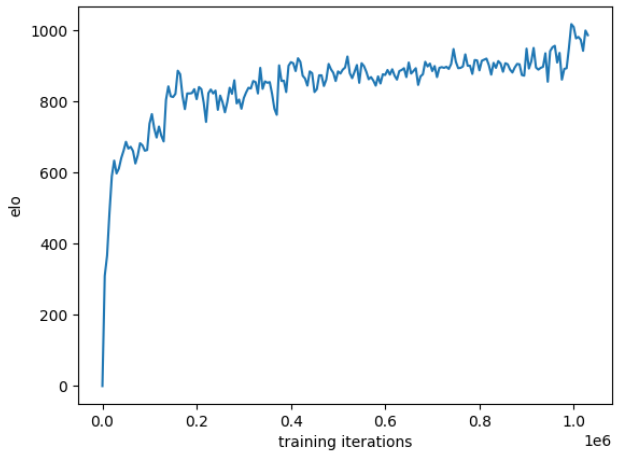

Figure 5.6: Improvement over time with a board of width 6. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

5.3.4 Width 7

In Figure 5.7, the improvement of our agent is shown for a board width of 7. Again there is a clearly noticeable jump at the start of training. This time however, the jump is not as steep, requiring a greater number of training

iterations to achieve an Elo rating of 600. An improvement of approximately 200 Elo points is achieved within the first $200,000$ training iterations, although the fluctuations in the graph suggest that the improvement of the agent is not entirely stable. After approximately $300,000$ training iterations, the agent achieves an approximate Elo rating of 950. Thereafter, there are some fluctuations in the rating of our agent, but there is little improvement, with the agent maxing out its rating at just over 1000 Elo points.

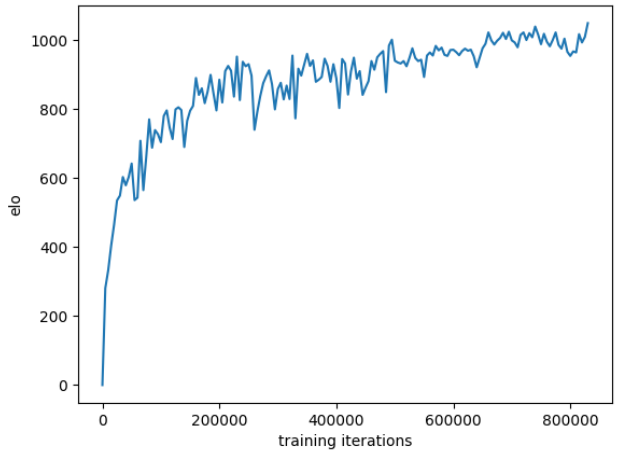

Figure 5.7: Improvement over time with a board of width 7. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

5.3.5 Width 8

The improvement of our agent for a board of width 8 can be seen in Figure 5.8. Once again, there is an initial leap in ratings from 0 to 600 at the very start of training. Thereafter, the agent steadily improves its rating for approximately $600,000$ training iterations, peaking at a rating of over 1200. This is the first board width for which the agent is able to maintain a stable rate of learning for a prolonged period of time. Since the plot of for the Elo rating has not plateaued completely, it is possible that the agent would continue to improve its rating further, if given more training time.

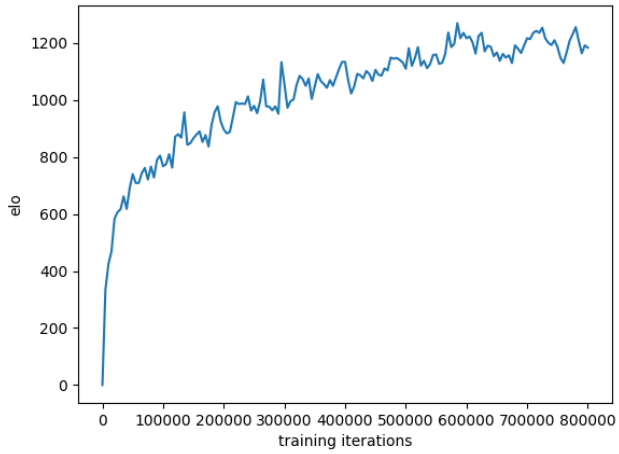

Figure 5.8: Improvement over time with a board of width 8. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

5.3.6 Width 9

Figure 5.9 shows the improvement of our agent for a board of width 9. As usual, there is an initial jump in rating from 0 to approximately 600. This time, the agent manages to cross the 1000 mark after approximately $200,000$ training iterations, showing that it is still able to learn quickly despite the increased environment complexity. However, in the time period from around $300,000$ to $450,000$ training iterations, the rating of the agent drops by close to 200 Elo points, before rising back up again and surpassing its previous peak. Once again, it seems that this agent could learn more, but that the training time was cut short.

5.3.7 Width 16

The training improvement of our agent for a board width of 16 is shown in Figure 5.10. As for all the other board widths, there is an initial jump to a rating of 600. After this, there is a period of around $400,000$ training iterations, during which the Elo rating of the agent fluctuates between 600 and 800. Thereafter, the Elo rating of the agent drops for the remaining of the training session, ending at a rating of approximately 400. This behaviour is unexpected, but is probably caused directly or indirectly by the large increase in environment complexity.

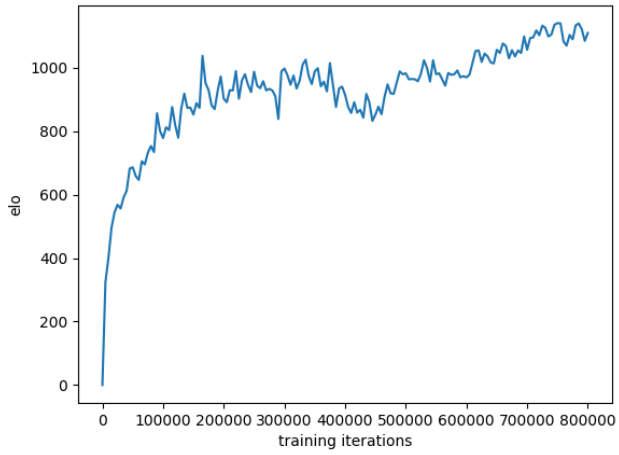

Figure 5.9: Improvement over time with a board of width 9. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

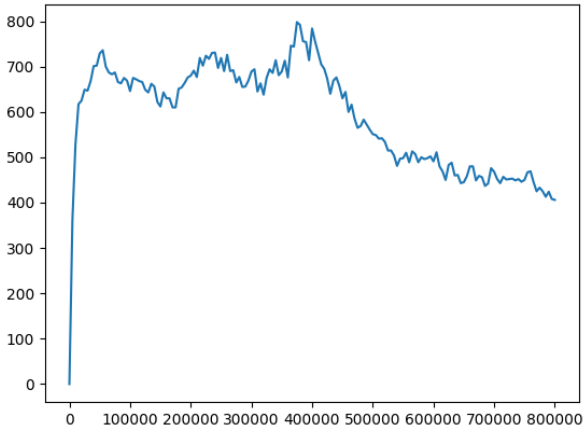

Figure 5.10: Improvement over time with a board of width 16. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

5.3.8 Width 32

The improvement of our agent during training on a board of width 32 is shown in Figure 5.11. From the figure it is clear that the agent was unable to learn much from its training, with the Elo rating of the agent fluctuating

between 0 and 100 throughout training. It would seem that the large increase in environment complexity proved too hard for the agent to learn, when provided with the same parameters as for the other board widths.

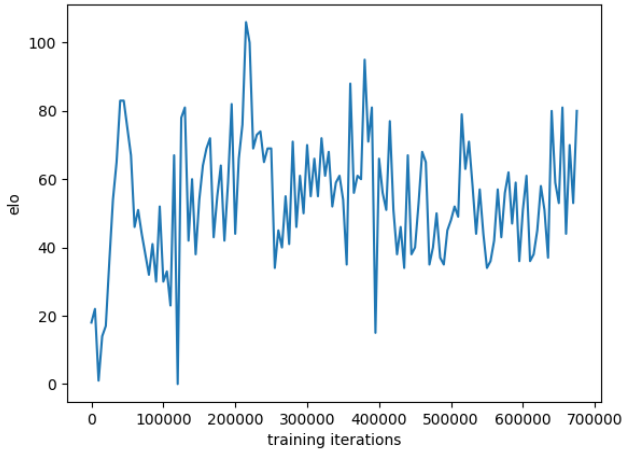

Figure 5.11: Improvement over time with a board of width 32. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

5.3.9 Analysis of board width results

Figure 5.12 shows the improvements of each of the agents, for each of the widths used during training. While it is difficult to analyse which of the agents plays the most perfectly, the plots provide an intuitive and somewhat clear way of investigating the rate at which they learn.

The first thing to note about the plots is that most of them are similar to a very large degree for the beginning of training. For the first $100,000$ training iterations, all agents except the one which was trained on a board of width 4 and 32 look nearly identical. There is some variation for the different agents, however the differences are not sufficiently large to be able to state what is causing these differences.

The agents which were trained on boards of widths 4 and 5 nearly only show improvement during the beginning of training. This is likely due to the fact that these are the simplest environments in terms of complexity. As such there is not much which can be learnt about the game. With this said, it seems that these two agents were the agents which improved the most quickly, as they seemingly can not improve much further.

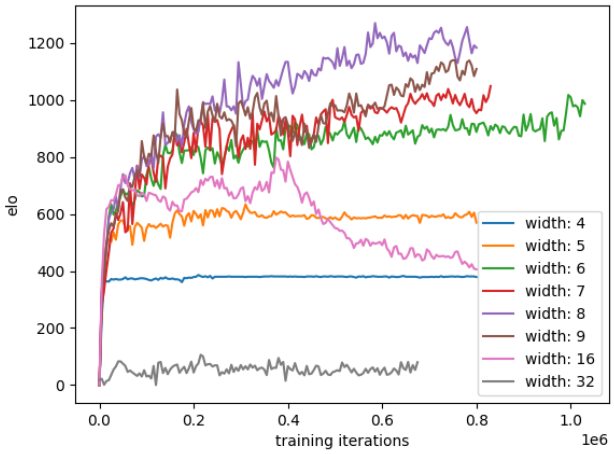

Figure 5.12: Improvement over time for all different board widths.

For the rest of the agents, it is during the later stages of training which some larger distinctions can be made. The agents which were trained on boards of widths 6, 7, 8 and 9 continue to improve, usually at a slow rate, for the remainder of the training. As seen in Figure 5.12, the agent that was trained on a board width of 8 has a higher Elo rating for a large portion of the training. This suggests that this agent was the one which was able to improve the quickest, since there is portion of the training where any other agent significantly outperforms it in Elo rating.

The agents which were trained on boards of widths 6, 7 and 9 match up very closely after $400,000$ training iterations, and it is only after this point which a clear distinction in rating is made. Out of these three agents, the agent which was trained on a board of width 9 finishes with the highest rating, followed by the agent trained on a board of width 7 and thereafter the agent trained a board of width 6.

The agents which were trained on a board of width 16 and 32 are clear outliers when looking at the plots. It is immediately evident that these agents performed significantly worse than any of the other agents in terms of improvement. For the agent which was trained on a board of width 16, it seems that the agent was able to improve at first, but not throughout the entire training session.

The reason for this is not entirely clear. One possible explanation for this is that the game is too complex for the agent to learn, when provided with the same set of parameters as the other agents. Perhaps it would be able to

improve at a better rate if the learning rate was adjusted or if the size of the neural network was increased.

Another possible explanation for the results is that the agent was able to improve its policy at the start of training, but thereafter got stuck exploring the same types of states for the remainder of training, leading to overfitting. However, we think that this is not the likliest cause, given that this did not happen for smaller board sizes.

For the agent which was trained on a board of width 32, it is probably fair to state that the agent did not learn much at all, judging by Figure 5.11. The reason for this is likely that the size of the board was too large to be able to be learnt by the smaller neural network that we used. Since the width of the board is 32, we did not have enough convolutional layers for a filter to capture information about all parts of the board, which may have affected the results.

5.4 Neural network size

In this section we present our results and analysis of the training runs which explore the effects of the size of the neural network. In order to vary the size of the neural network, each run used a different number of filters in each of the convolutional layer. The architecture of each neural network is the same as outlined in Chapter 3. The only factor which we modify is the number of filter used. Across our runs, we have tested neural networks with 1, 2, 4, 8, 16, 32 and 64 filters. For each of our runs in this section, a board of width 7 was used for $800,000$ training iterations.

5.4.1 1 Filter

Figure 5.13 shows the improvement over time of an agent trained using 1 filter per convolutional layer. The agent improves its rating quite steadily throughout training with a few occasional fluctuations.

5.4.2 2 Filters

Somewhat surprisingly, our agent seems to learn more slowly when using 2 filters. As shown in Figure 5.14, there is little improvement during the first half of training. During the second half, there is a significant improvement of approximately 200 Elo points. Overall the total improvement and the rate of improvement is lower than that of the agent with 1 filter though.

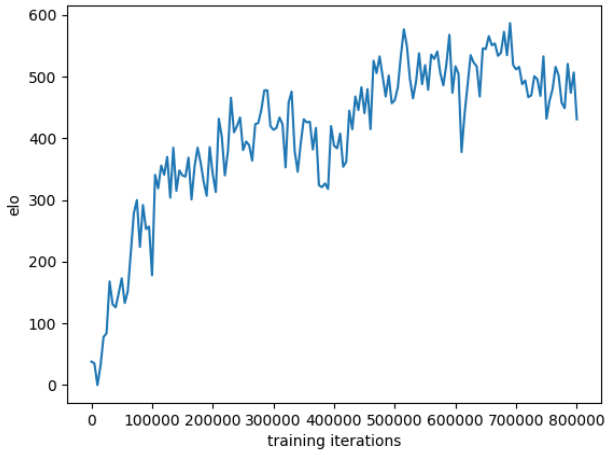

Figure 5.13: Improvement over time with a board of width 7, using 1 filter for each convolutional layer. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

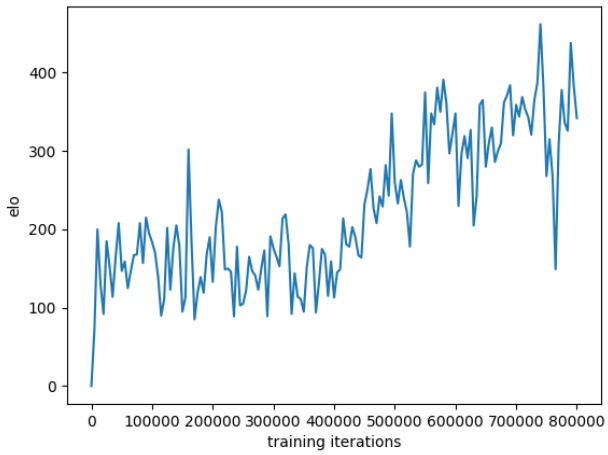

Figure 5.14: Improvement over time with a board of width 7, using 2 filters for each convolutional layer. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

5.4.3 4 Filters

The agent which uses 4 filters for each convolution layer is depicted in Figure 5.15. The rate of improvement is more steady for this agent when compared to that of the agents with 1 and 2 filters. It gains a total of 600 Elo points which is similar to that of the agent with 1 filter, however since the plot never plateaus, it is possible that it could improve further.

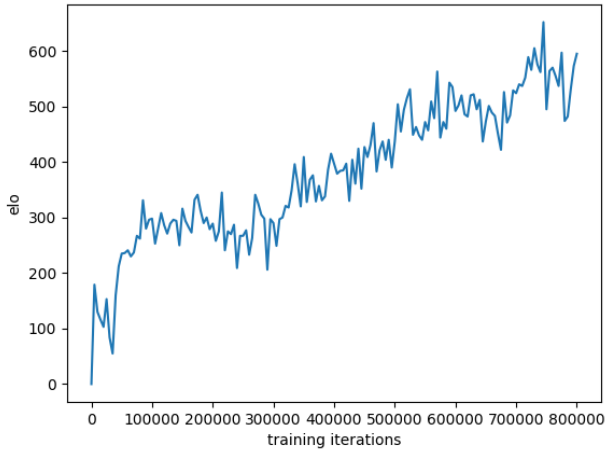

Figure 5.15: Improvement over time with a board of width 7, using 4 filters for each convolutional layer. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

5.4.4 8 Filters

Figure 5.16 shows the improvement of the agent trained using 8 filters per convolutional layer. For the first time, we see a large jump at the start of the training, similar to what we saw when training on varying board sizes. After the initial rating jump, there is a slow but steady improvement rate for the rest of training, also reaching an Elo rating of 600.

5.4.5 16 Filters

Figure 5.17 shows the improvement rate of the agent trained with 16 filters per convolutional layer. There is once again a large initial jump in Elo at the start of training, reaching a rating of 400. Thereafter the agent is able to improve

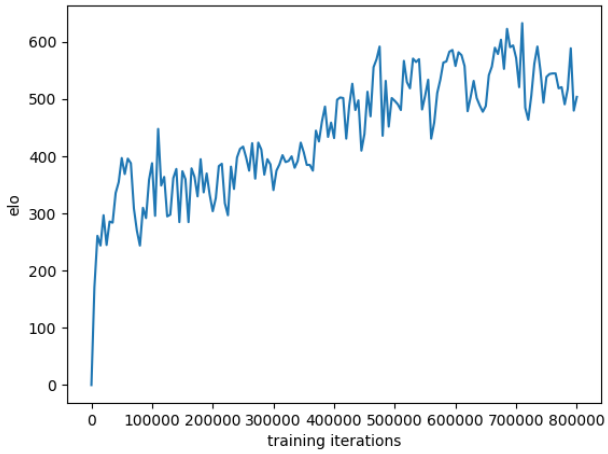

Figure 5.16: Improvement over time with a board of width 7, using 8 filters for each convolutional layer. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

further, peaking at close to 800 Elo points. This agently is seemingly learning faster than the previous agents which had fewer filters. This agent is also able to learn more since it achieves a higher Elo rating.

5.4.6 32 Filters

Figure 5.18 shows the improvement of an agent trained using 32 filters per convoluational layer. Again, there is a large jump in ratings at the start of training. However this jump is noticebly larger than previously, immediately reaching a rating of approximately 600. Thereafter the agent slowly nears a rating of 1000, seemingly learning even more than the agent with 16 filters.

5.4.7 64 Filters

Figure 5.19 shows the agent which was trained using 64 filters per convolutional layer. The plot is similar in many ways to that of the agent which was trained using 32 filters. The main difference here is that using 64 filters results in a somewhat steeper curve, suggesting that it learns more quickly, even if it becomes equally good in the end.

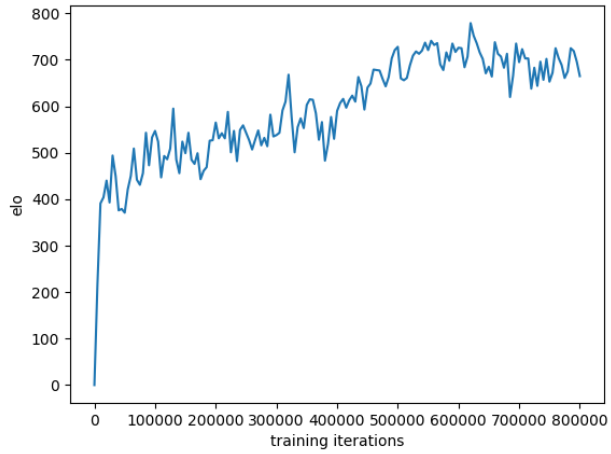

Figure 5.17: Improvement over time with a board of width 7, using 16 filters for each convolutional layer. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

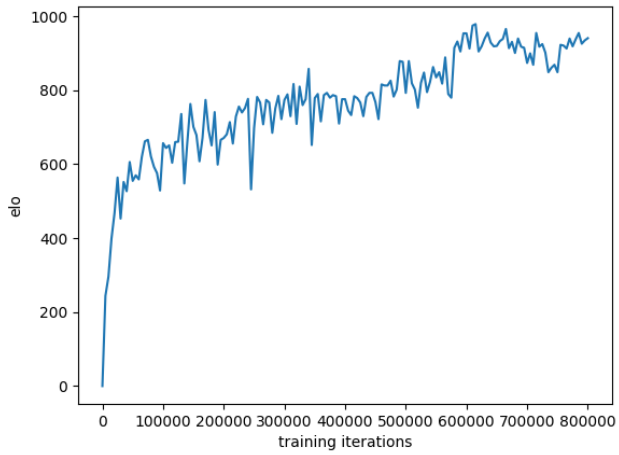

Figure 5.18: Improvement over time with a board of width 7, using 32 filters for each convolutional layer. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

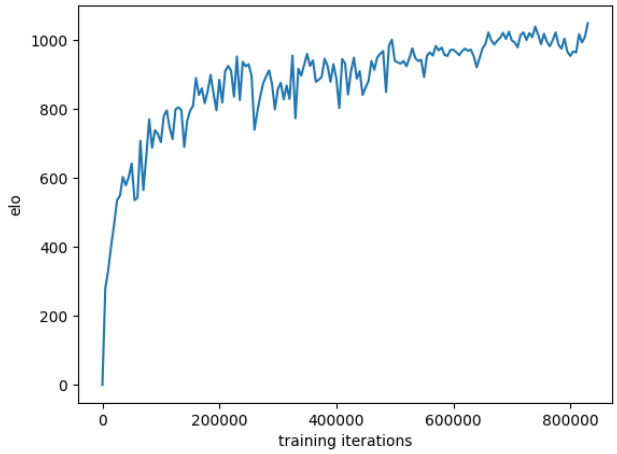

Figure 5.19: Improvement over time with a board of width 7, using 64 filter for each convolutional layer. The data points for the curve were generated using the BayesElo [9] program with match results from a tournament of agents, saved each 5000 steps during training.

5.4.8 Analysis of neural network size

From the figures presented, there is a clear difference in results for different numbers of filters. In general it seems that more filters lead to a higher Elo rating, meaning the agent is able to learn more. This is also what is to be expected since a neural network with more parameters should be able to fit data better. On top of this it also seems that a model with more filters is able to learn more quickly as well.

In Figure 5.20, we join the plots which we presented for each of the different number of filters. The agents with 1, 4 and 8 filters are very similar and thus it is hard to draw any conclusions from them. The agent with 2 filters is a clear outlier, performing the worst. These four agents are arguable the most susceptible to variation though since their neural networks have the fewest number of parameters. However, their plots suggest that the agent is not able to learn very well when a small number of filters is used.

Using 16, 32 and 64 filters results in a much greater improvement. It is also clear from the plot, that using 16 filters results in a slower rate of improvement than using 32 filters. Similarly using 32 filters results in a slower rate of improvement than using 64. The total improvement is also higher when using a greater number of filters.

In conclusion it seems that using a higher number of filters results in our

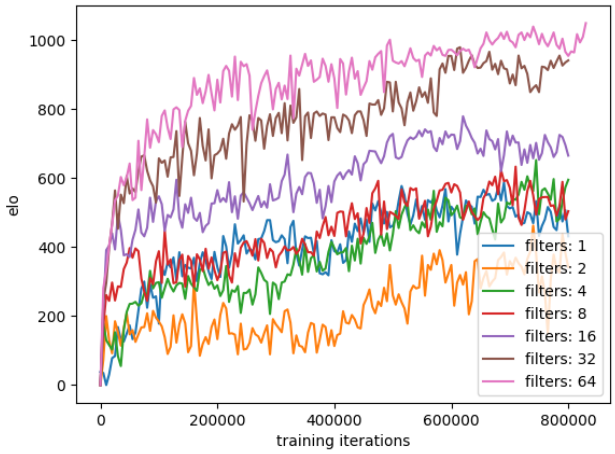

Figure 5.20: A joint plot of all agents which were trained on a board of width 7 using different number of filters.

agents learning more quickly and improving more overall, when compared to using fewer filters. Using a number of filters close to 0 allows for little improvement and also seems to have a risk of not performing well due to variance.

5.5 Monte-Carlo tree search iterations

In this section we explored the effects of varying the number of MCTS iterations for an already trained agent. We present our findings in Figure 5.21. It shows the results from a round robin tournament played between the same agent, varying the number of MCTS iterations. Each pair of players (different players meaning different number of MCTS iterations) faced each other 1000 times in total, each player being the first player in 500 of the games. The x-axis of the plot shows the logarithm of the number of MCTS iterations which were used. The number of iterations used were $1, 2, 4, 8, 16, 32, 64, 128, 256, 512$. From the figure we see that there is a clear trend in the playing strength of the agent, with more MCTS iterations correlating to a higher Elo rating. There is only one outlier to this trend which is when using one MCTS iteration.

Using only one MCTS iteration is effectively the same as only using the priors of the neural network for deciding on which move to play. Using a large number of MCTS is expected to improve upon this, and this is reflected in Figure 5.21. However, it is somewhat unexpected that a worse performance

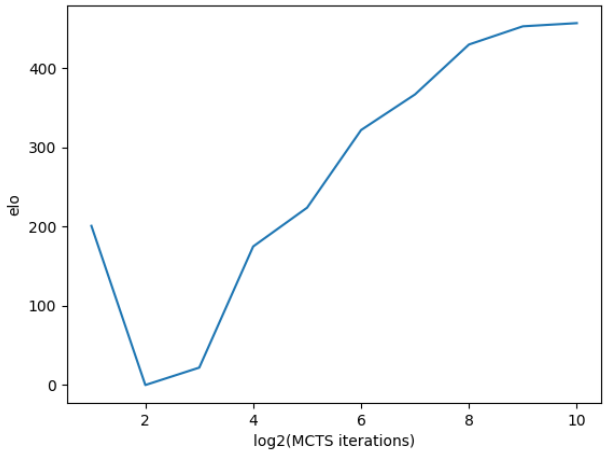

Figure 5.21: A comparison of player strengths for the same agent, when using a different number of MCTS iterations.

should be seen with 2 and 4 MCTS iterations. We believe that the reason for this is the following: consider using 2 MCTS iterations. In our code, the first iteration will search the move which has the highest prior. For the second iteration, it could explore either the move with the second highest prior or the same move as in the first iteration. If it were to explore the same move as in the first iteration, that move will have all of the visits from the simulation and will therefore be played. This will thus result in the same behaviour as having a single MCTS iteration. However, if another move is searched, this move will be played with a 0.5 probability since there are two moves which have been visited an equal number of times. Since the neural network is already trained, it is more likely that the move with the highest prior is the best move, and as such, using 2 MCTS iterations could actually affect the performance of the agent negatively.

The difference in playing strengths between the agent which uses 1 MCTS iteration, effectively only using its priors, and the agent which uses 512 MCTS iterations is smaller than we predicted. The rating difference between the two is approximately 250 Elo points. This correlates to an approximate win rate of 80% for the agent using 512 MCTS iterations. Taking Figure 5.7 into account, we observed a 1000 Elo improvement during the training of our agent. Since only using the priors results in an Elo rating of approximately 250 less, we can conclude that only using the priors, results in a performance of 750 Elo points higher than the initial parameters equipped with 512 MCTS iterations,

correlating to a near 99% win rate.

Chapter 6

Discussion

In this chapter we will discuss our findings. We will bring up some factors which may have affected our results and also discuss how our results could have been improved.

6.1 Hyperparameters

In our work, we used many of the same hyperparameters as were used in the AlphaGo Zero paper [2]. The main parameter which was changed was the number of residual blocks in our neural network. This was done for two main reasons: to speed up training time, and to reduce the number of neural network parameters. The reasoning behind changing this parameter was that the environment we are exploring, namely Connect 4, is a less complex game than Go. Go is played on a 19 by 19 board, while Connect 4 is played on a 6 by 7 board. Due to this we felt that the high number of layers used in AlphaGo Zero was not necessary. This is something that may have affected our results, since we later moved on to testing larger boards for Connect 4, our largest board size being 7 by 32. This being said, the main importance was to keep the parameters the same during the different training runs, such that it was only the environment which was changing. We made sure to do this and, as a result, the smaller neural network was used throughout our work.

Another parameter which we would have like to experiment more with was the size of the replay buffer. During our training runs, we used a buffer with a capacity of $100,000$ positions. This is far fewer than the $500,000$ games which were saved in AlphaGo Zero [2]. The only reason we decreased the buffer size by such an amount was due to memory constraints. During our runs, we were unable to save larger buffers in memory. A capacity of approximately $100,000$

was the largest we could use, and therefore is what we chose.

We believe that the choice of buffer capacity may have affected some of our results. The figures from our training runs on board widths of size ranging from 4 to 9 matched what we expected to see. However for our training run on a board width of 16, we saw a decrease in performance during the second half of training. One possible explanation for this is that the buffer was too small, making the agent *forget* about some of its past experiences. This is something which becomes more relevant for games with greater complexity, and we believe that this is why it only appeared in our results for the most complex environment.

6.2 Low Elo ratings

From the figures which we presented in Chapter 5, the highest Elo improvement for any of our agents was approximately 1200. This correlates to having a 1 in 1000 chance of losing to the initial agent. While this improvement is significant, and perhaps a larger rating increase could have been observed with a longer training period, it is lower than could be expected. For example, in the game of chess, one of the best computer programs, Stockfish 15, possess an Elo rating of 3541 according to CCRL [17], which provides a list of ratings for the top chess engines. In AlphaGo Zero [2], their trained agent achieves a rating of over 5000, more than a 7000 improvement over the agent with its initial parameters.

One thing to note here is that a rating of 0 does not necessarily correlate to the same playing strengths in any of these games as they can not directly be compared to one another. With this said, there is a still a very large difference in the rating gain of our agent and the one used by DeepMind. There are two interesting points to discuss regarding this. The first is the hyperparameter tuning, and the second is environment complexity.

As we have stated in this work, our goal was not to create an AI with the highest possible playing strength. As a result of this, we did not focus on hyperparameter tuning. This is something which was done in AlphaGo Zero, which explains part of their success in achieving a higher Elo rating than our program.

Secondly, Connect 4 is a far less complex game than Go. The number of squares on a Go board is 361. When compared to the 42 squares on a Connect 4 board, this leads to Go having a much greater state complexity, along with a greater action complexity. This should equate to there being more knowledge to learn about Go, and hence a bigger rating improvement can be observed in

Go. Intuitively this also seems reasonable. In games such as Go and chess, the more experienced player often ends up on top, even for players of nearly equal strength. In Connect 4 however, games are often closer, with most players scoring close to 50% against one another.

Additionally, draws are possible in Connect 4, whereas they are not in the game of Go. As a result of this, it should be expected that, even close games should favor the more experienced Go player, whereas in Connect 4, a close game will likely end a in a draw.

Lastly, we would also like to note that Connect 4 has been proven to be a win for the first player (when playing on a 6 by 7 board). During the early stages of training we often observed a near equal amount of wins for the first and the second player with some occasional draws. However, during the later stages of training, we often saw that the first player was winning far more often than any other result. This would suggest that our agent was able to become sufficiently good at the game to force a win in most cases. This is also backed up by our results from the games against a perfect AI, as mentioned in Chapter 5. After an agent is able to learn a policy which is close to optimal, little further Elo gain can be observed, and this is another reason that we believe the rating improvement of our agent was low.

6.3 Choice of environment

We motivated our choice of environment in Chapter 3. One of the key reasons we chose to use Connect 4 was that it is easy to alter the size of the board without needing to change the rules of the game. One downside of us choosing this environment is that the game is not very complex. Due to this, it might have been more interesting to use another testing environment. For example, in Connect 4 most moves are legal for a majority of the game, and as a result, the neural network probably has an easier time predicting the priors. However, in other games such as Go and chess, moves are often illegal. This would have added an extra layer of complexity to the training which would have been interesting to investigate.

During early stages of development, Othello was considered as the game to use. Othello starts with a nearly empty grid, with four central stones, two white, and two black. During a game, stones can only be placed next to already placed stones (with some additional restrictions from the rules). This would have satisfied our interesting in having moves be illegal, but the reason we chose not to investigate Othello further was that we thought that the same strategies should be good regardless of the board size, since the board size

is only relevant once the stones are placed near the edges of the board in the case of Othello.

Chapter 7

Conclusions and Future work

In this chapter, we present our conclusions and some possible future work, which could be researched further.

7.1 Conclusions

From our results, we were unable to find a clear correlation between the complexity of an environment and the improvement rate of our agents. During the initial stages of training, the rate of improvement was similar for all of our agents, whereas during the later stages of training, agents training on more complex environments showed more promise in terms of overall improvement.

Although we do not have a way of verifying which of our agents learned the most optimal policy in terms of theoretically playing optimal moves, we believe that the agents which were trained on more simple environments came close to learning an optimal policy.

We can however, conclude that having a larger neural network improves the playing strength of an AlphaZero agent. Along with this, the rate of improvement is also higher for agents using a large neural network, when comparing by the number of games played.

For an already trained agent, we showed that, in general, the higher the number of MCTS iterations used, the better the agent performs during inference time. However, the gap in performance between using a single MCTS iteration and using 512 was not huge. As a result of this, if inference time is of importance, the number of MCTS iterations could be kept at a low number.

Even though we were not able to concretely answer all of our research questions, we feel we have achieved our goals. We have shown that

implementing a fast-running AlphaZero algorithm is possible without the need for an excessive amount of computational power. We also feel that we have made it possible for future work to be carried out, which builds upon our work and which explores the topic in even more depth.

Throughout this project we have gained a greater understanding for how AlphaZero agents are able to learn. Two very important factors are the speed of executing the Monte-Carlo tree search as well as the quality of the collected data. For future researchers, we recommend that some special attention be given to these two areas. Optimising the MCTS algorithm is perhaps the most important part of the code, since without it you can not train an agent in a reasonable amount of time without expensive hardware. For data collection, it can be a good idea to spend more time on each move during the self-play games, such that you can calculate a more accurate set of target priors and values to save in the replay buffer.

If we had a second chance to do this project, we would like to save more data from the training. For example; which the most common first moves are during different stages of training or how often the played move differs from what the highest prior would suggest as the best move. This kind of information would have been interesting to have in order to see how the agent's behaviour develops during training.

7.2 Limitations

After several of our training runs had been successfully completed, we noticed that there was a large similarity between many of the plots which we produced. As a result we decided to increase the board width to 16 and 32 in order to get board widths which differed by powers of two. This however limited the number of training runs which we were able to conduct as the bigger board widths increased the duration of training by a large factor. As a result of this we have fewer training runs than we could have otherwise had, which may have limited our ability to gather results.

7.3 Future work

Due to the complexity of our research questions, we were only able to answer some of our questions. In this section we will bring up some issues which should be addressed in future work.

7.3.1 More training runs

The first thing which should be explored further, is to compare several more training runs. In this thesis, we have not explored the board widths from 10 to 15. On top of this, using environments other than Connect 4 would also be helpful as they would offer more variation.

7.3.2 Comparing agents across environments

In our current work, there is no direct way for us to compare agents which were trained in different environments. We are able to judge by how much an agent has improved compared to its initialisation. However, it would be helpful to see how good the agent is compared to a *perfect* agent. This would require knowledge about which moves are the theoretically best, which comes with an added challenge, but it would give rise to a more interesting way to analyze the performance of the agents.

7.3.3 Training a single agent in multiple environments

For certain use-cases, it may be of interest to be able to train an agent which learns a good policy for several environments simultaneously. In our case of Connect 4, we had the idea of training a single agent which would be able to take input boards of any dimension. Our idea is that the agent would play games against itself on boards of varying dimensions during training. During evaluation, we could then partly test its ability to play the games on board sizes which it had seen training, but also on board sizes which it had not seen previously. We did not implement this training procedure as it was too far away from our research questions. However, we feel that this is something which would be very interesting to research and which could also be useful. For example, consider the case of modelling a real world problem as a game. If one is unsure about which action space your environment should have, one could let the agent learn in multiple environments at once. At the end one could then check which of the environments the agents performs best in and use that as your modelled environment.

7.4 Reflections

Due to the nature of this thesis, there are not any large social, environmental or ethical aspects of this work. This is due to this thesis consisting mainly of

experiments which are simulated with the help of code. There is for example no need to collect a labelled data set which could otherwise require a lot of human work. Also, since AlphaZero [3] is an algorithm which existed several years before this project was carried out, any ethical or environmental concerns were already present before this work. I.e. we argue that we have not raised any new concerns with this project.

The main economic aspect of this work is that Ericsson will have access to the code which was used for all the experiments. Ericsson could thus potentially stand to benefit from this through further research. Other than this, there should not be too great of an interest in this project from an economic view point. Again, this stems from the fact that the AlphaZero [3] algorithm already existed, thus this work mainly affects Ericsson, who are allowed to keep the code.

References

[1] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 01 2016. doi: 10.1038/nature16961 [Page 1.]

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct 2017. doi: 10.1038/nature24270. [Online]. Available: https://doi.org/10.1038/nature24270 [Pages 1, 6, 7, 12, 13, 17, 41, and 42.]

[3] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *CoRR*, vol. abs/1712.01815, 2017. [Online]. Available: http://arxiv.org/abs/1712.01815 [Pages 1, 2, 3, 6, 7, 11, and 48.]

[4] "leela-zero," accessed: 2022-04-26. [Online]. Available: https://github.com/leela-zero/leela-zero [Page 3.]

[5] M. E. Glickman and A. C. Jones, "Rating the chess rating system," *CHANCE-BERLIN THEN NEW YORK-*, vol. 12, pp. 21–28, 1999. [Pages 5 and 6.]

[6] "Chess.com," https://www.chess.com/, accesssed: 2022-04-28. [Page 5.]

[7] "Lichess," https://lichess.org/, accessed: 2022-04-28. [Page 5.]

[8] R. Coulom, "Whole-History Rating: A Bayesian Rating System for Players of Time-Varying Strength," in *Computer and Games*, ser. Lectures Notes in Computer Science, van den Herik, H. J., Xu, Xinhe, Ma, Zongming, Winands, and M.H.M., Eds., vol. 5131. Beijing, China: Springer, Sep. 2008, pp. 113–124. [Online]. Available: https://hal.inria.fr/inria-00323349 [Page 6.]

[9] "Bayesion elo rating," https://www.remi-coulom.fr/Bayesian-Elo, accessed: 2022-04-22. [Pages xi, xii, xiii, 6, 15, 21, 25, 26, 27, 28, 29, 30, 33, 34, 35, 36, and 37.]

[10] K. Cobbe, C. Hesse, J. Hilton, and J. Schulman, "Leveraging procedural generation to benchmark reinforcement learning," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 2048–2056. [Online]. Available: https://proceedings.mlr.press/v119/cobbe20a.html [Page 9.]

[11] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. P. Lillicrap, and D. Silver, "Mastering atari, go, chess and shogi by planning with a learned model," *CoRR*, vol. abs/1911.08265, 2019. [Online]. Available: http://arxiv.org/abs/1911.08265 [Pages 9 and 10.]

[12] T. Hubert, J. Schrittwieser, I. Antonoglou, M. Barekatain, S. Schmitt, and D. Silver, "Learning and planning in complex action spaces," 2021. [Online]. Available: https://arxiv.org/abs/2104.06303 [Page 10.]

[13] J. Schrittwieser, T. Hubert, A. Mandhane, M. Barekatain, I. Antonoglou, and D. Silver, "Online and offline reinforcement learning by planning with a learned model," *CoRR*, vol. abs/2104.06294, 2021. [Online]. Available: https://arxiv.org/abs/2104.06294 [Page 10.]

[14] B. P. Wise, "Elo ratings for large tournaments of software agents in asymmetric games," *CoRR*, vol. abs/2105.00839, 2021. [Online]. Available: https://arxiv.org/abs/2105.00839 [Page 15.]

[15] G. M. J. B. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel monte-carlo tree search," in *Computers and Games*, H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands, Eds. Berlin, Heidelberg:

Springer Berlin Heidelberg, 2008. ISBN 978-3-540-87608-3 pp. 60–71. [Page 17.]

[16] I. Babuschkin, K. Baumli, A. Bell, S. Bhupatiraju, J. Bruce, P. Buchlovsky, D. Budden, T. Cai, A. Clark, I. Danihelka, C. Fantacci, J. Godwin, C. Jones, T. Hennigan, M. Hessel, S. Kapturowski, T. Keck, I. Kemaev, M. King, L. Martens, V. Mikulik, T. Norman, J. Quan, G. Papamakarios, R. Ring, F. Ruiz, A. Sanchez, R. Schneider, E. Sezener, S. Spencer, S. Srinivasan, W. Stokowiec, and F. Viola, "The DeepMind JAX Ecosystem," 2020. [Online]. Available: http://github.com/deepmind [Page 18.]

[17] "Ccrl," https://ccrl.chessdom.com/ccrl/4040/, accesssed: 2022-06-01. [Page 42.]

TRITA-EECS-EX- 2022:543