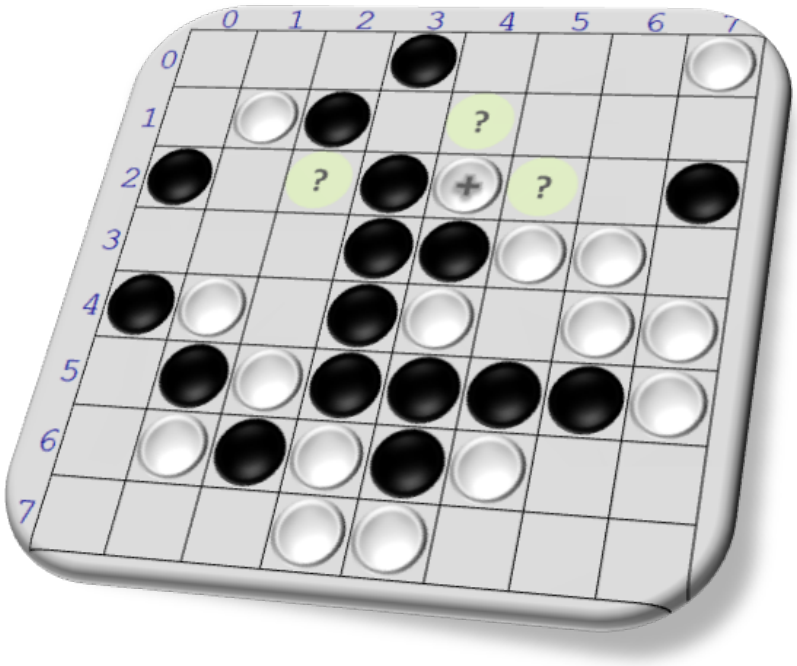
Monte Carlo Tree Search with Neural Network

Alexander Mønnike Hansen [almh@itu.dk]
Mikkel Hooge Sørensen [mhso@itu.dk]
Frank Andersen [fand@itu.dk]

Bachelor Thesis, Software Development
IT-University of Copenhagen

May 2019



IT UNIVERSITY OF COPENHAGEN

Contents

1 Abstract

In 2017 DeepMind introduced *AlphaZero*, a program for mastering Chess, Shogi, Go, and with the capabilities of mastering any game with similarly defined rules and game states. AlphaZero accomplishes this using Deep Reinforcement Learning with no prior encoded knowledge, and learns optimal strategies entirely from self-play. Self-play and training is powered by specialized Tensor Processing Units (TPUs), each generating game state data for a convolutional neural network. In this paper, we wish to investigate whether the approach used by AlphaZero can be implemented, and utilized, on consumer grade hardware, and produce interesting results within a reasonable time frame. In doing so, we detail our experiences of implementing the algorithm, and of testing different parameters and configurations. We do so in order to construct a program that is as optimized as possible. Throughout the paper we provide advice on the important optimizations, and possible pitfalls, that we experience along the way. Our results show that it is feasible to train an algorithm, that uses the same techniques as AlphaZero, on consumer grade hardware within a reasonable amount of time. This is only made possible by the limited complexity of the specific games chosen. More complex games, like Go, will likely not be feasible with non-specialized hardware, unless techniques such a crowd-computing or alternate approach are used.

2 Preface and Introduction

This project is written as a product of our bachelor thesis, at the IT-University of Copenhagen, spring term of 2019. The project is titled *Monte Carlo Tree Search with Neural Network*, and aims to examine the combined use of Monte Carlo Tree Search (MCTS), and a Convolutional Neural Network (CNN), to achieve mastery at deterministic games with perfect knowledge. The project has been done under the supervision of Troels Bjerre Lund, Associate Professor at the IT-University of Copenhagen.

We would like to thank our supervisor Troels Bjerre Lund for assistance, as well as the groups of fellow students working on similar problems.

This report assumes a thorough understanding of programming, specifically in Python[1], as well as a basic understanding of the Monte Carlo Tree Search algorithm [2], Neural Networks, specifically Convolutional Neural Networks[3], and basic knowledge of game theory. This paper is mainly aimed at readers that are interested in implementing a program that uses the techniques that AlphaZero has popularized, as well as readers interested in learning about the inner workings of AlphaZero.

Problem definition

In recent years self trained algorithms making use of Monte Carlo Tree Search in conjunction with neural networks have emerged. Most notably, AlphaZero from Google's DeepMind has proven that such algorithms can consistently beat specifically tailored algorithms in games such as Chess, Go and Shogi. These self trained algorithms, such as AlphaZero, make use of specialized hardware to significantly speed up their training.

Can algorithms like AlphaZero realistically be made to achieve better results, for instance measured by winrate, when run on consumer grade hardware, or are traditional algorithms, tailored for specific games, better suited when no specialized hardware is used?

AlphaZero consist of two central parts: training the neural network, and playing games with that network, in order to generate further data for it to train on. Generating data for training requires a large amount of computational complexity, and this is where AlphaZero utilized specialized hardware. However, when the network has finished training, and is used for playing games, it needs less computational power and resources than more conventional game-playing programs, like Stockfish for Chess. This was noted by the Deepmind team when testing AlphaZero[4](p. 59)

Therefore, we mainly wish to investigate whether training the network, can be done on consumer grade hardware, since utilizing a previously trained network is trivial and can certainly be done on most hardware. The vast majority of computational complexity lies with the feedback loop of generating game data and training the network on this data. This will therefore be the primary focus of our efforts.

2.1 Naming the program

Every good program needs a proper name. We considered naming our program in the style of AlphaZero, such as OmegaZero or BetaOne, but ultimately decided on the name *Katafanga*. Katafanga is a small island in the South Pacific Ocean, close to Fiji, and the name means *Smiling Beach*. As AlphaZero was once a dream for the team at Deepmind, so is the island of Katafanga a dream for our team, and hopefully one that can someday be fulfilled.

2.2 Motivation

AlphaZero is purely self-trained, knowing only the legal moves according to the rules of the game that it is playing. Despite this, it is a very strong algorithm, having beaten top of the line algorithms that employ complex hand-crafted heuristics, such as Stockfish[5] for Chess, and Elmo[6] for Shogi. However, AlphaZero was trained using specialized hardware[4], which allowed it to train its neural networks at a rate, which no commonly available hardware could match. If the techniques used in AlphaZero were to become commonplace, and if the algorithm could be optimized enough to be used on regular hardware, the algorithm could benefit areas, that would otherwise be excluded from using it, because of a lack of specialized hardware. The fact that AlphaZero operates in a relatively domain-agnostic way opens the door for many opportunities for potential real-world applications. If a problem can be stated in terms of rules, and if potential solutions to the problem can easily be verified, AlphaZero could feasibly tackle it, without requiring modifications to the core algorithm.

As shown in the Deepmind research[7] the self learning approach can contribute to finding new perfect game-solutions, or confirm what is thought to be ultimately known as the best moves already. If AlphaZero becomes readily available, it could become the go-to advisor for professional players in many games, one example being Chess. In Chess, it would most likely replace Stockfish as the leading chess engine, and could provide new insights into the already vast opening books of top Chess players. It might also be used to propose new and innovative ways to play the game, as it already has in the few show matches it has played.

3 Background and Description of the Problem

In this section we will expand on what we wish to investigate with this project, followed by a brief introduction to the background of AlphaZero. Lastly we will discuss the requirements for the program that we wish to implement to solve the problem presented above.

3.1 Historical context

Historically, algorithms for playing games at a competent level has been heavily knowledge based, where years of human experience has been explicitly encoded in complex heuristics functions, designed to help the AI navigate the game tree of the specific game. This required not only a thorough understanding of proper strategies for the given game, but would ultimately be of very little value when designing a separate AI for another game. IBM's famous Chess playing program *Deep Blue* convincingly beat chess grandmaster[8] Garry Kasparov, but would have very little success if faced with the challenge of playing Checkers.

AlphaZero sought to change this. The principles behind AlphaZero are not necessarily groundbreaking in isolation, but the way in which they are combined is innovative and interesting. AlphaZero is built on ideas which were already conceptualized in the 1950's and 60's. This includes Markov Decision Processes, Reinforcement Learning (specifically Q-learning), and Neural Networks. Only recently has a new field within Reinforcement Learning emerged, called Deep Reinforcement Learning. Here, evaluating the reward of performing certain actions within the state space defined by the problem at hand, is done by a neural network. This has only become possibly in the last few decades, as hardware has become more and more powerful. This, combined with the fact that AlphaZero managed to create an algorithm that is highly game agnostic, is what makes the algorithm interesting.

By utilizing ideas from Reinforcement Learning, the proficiency that AlphaZero achieves in the games it plays is achieved purely through self-learning. This means that no human knowledge, beyond the limitation set by the game rules, is encoded into the algorithm. While this could be seen as a disadvantage, it turns out that without being influenced by what humans understand about a game, AlphaZero is free to explore and develop tactics beyond human capabilities. It is even able to go beyond the capabilities of its predecessors like AlphaGo[9] and AlphaGo Zero[10]. This independence from game specific knowledge also means that AlphaZero can play a myriad of different games, without changing anything in the core algorithm.

The ideas and techniques that AlphaZero built upon first came to the world's attention when AlphaGo played Go[11] against Lee Sedol[12], considered the very best Go player, and won 4-1. During this breakthrough the algorithm employed a number of innovative tactics not commonly used, and showed what could be conceived as creative play. This is made even more impressive by the fact, that Go is a very complex game that is virtually impossible to play at a competent level for a classical AI algorithm like minimax[13]. From there AlphaGo became AlphaGo Master which became AlphaGo Zero and finally AlphaZero. Each generation built upon the previous with AlphaZero using many similar techniques of self-learning and evaluating game states as previous generations.

3.2 What we wish to investigate

We wish to investigate whether or not it is feasible to implement, and utilize, an algorithm, that is built on the techniques used by AlphaZero, meant to run on consumer grade hardware. AlphaZero used highly specialized hardware which is not only expensive, but is currently not commercially available. Specifically, for generating game state data, AlphaZero used 5000 first generation Tensor Processing Units (TPUs). Additionally, 75 second generation TPUs were used to train the neural network. Both generations of TPUs are built with machine learning operations in mind, and are specifically optimized for performing operations[14][15] such as matrix operations, convolutions, activation functions, etc, foregoing more general functions such as texture mapping, rasterization, multithreading and so on. At a very rough estimate, just one of these TPUs (from either generation) has the potential to deliver five times the throughput, compared to the the hardware we have available when executing

operations related to machine learning. An outline of the hardware used during this project, as well as a comparison to that used in AlphaZero, can be found in A.2.
Furthermore, we wish to investigate whether the approach used by AlphaZero can produce an AI that can outperform more traditional algorithms, when using common hardware, within a reasonable amount of training time. These traditional algorithms includes Monte Carlo Tree Search and alpha-beta search algorithms, specifically Minimax.

3.3 Requirements for solving the problem

The program should be able to play and learn two-player, turn-taking, deterministic games. Furthermore the program should be able to save data about its progress and evaluation. It should feature ways to monitor the progress of self-learning and training through evaluation against alternate game-playing algorithms as described above, as well as metrics for training loss within the network. The network itself should be implemented in way where it is flexible enough to accept inputs from different types of games, and should be easily re-configured if we decide to change its structure or parameters at a later date.

Earlier we explained that we would like our program to outperform traditional algorithms, in a reasonable amount of time. The word "reasonable" is very open to interpretation. There are several factors affecting training time. These include the desired game-playing competence of the AI, the complexity of the game, and the hardware used. For a decent level of competence in a somewhat complex game, we would consider a reasonable amount of time to be 2 weeks of training time, at the longest.
Another factor is the limited time frame of our project. We need to first implement, test, and then hopefully reach proper results within this time. If training takes much longer than 2 weeks, the success of this project becomes an unrealistic premise. If the program is used for games of higher complexity, or to get a higher level of competence, the training time would of course increase. As such this time requirement is still very much a tentative measure. Further details of the program, and the time needed for it to produce interesting results, will be further explored later in this paper.

3.4 Designing the program

These requirements mean that the program should feature a flexible structure where different *agents*, i.e. game-playing AIs, can be made to run, indifferent of the structure of the rest of the program. This also applies to how games are implemented. Since AlphaZero is meant to function on any game with the specifications described earlier, the games should also be interchangeable and feature an interface that the rest of the program can communicate with. These parts of the program should work as efficiently as possible, since they are the backbone of the program. If the games, or the agents playing the games, are slow, the entire process will suffer. Adding on to this, during training, the different *actors* can be made to run in parallel. An actor is an AI, using two agents to play games against itself. This process of playing against itself is called *self-play*. Many of these actors can be run simultaneously, since the data they generate are local to the specific actor, and can later be combined and trained on. This should create a large boost in performance. Most essentially, the performance of the games and agents should not hinder the process of training and learning, since this is where interesting results and optimizations can be found.

The neural network is the core of the program. It should be generic enough that it can accept input from any game, regardless of how the game plays, and how it is structured. An example of different types of input, also found in AlphaZero, is Chess where there exists both a *from* and a *to*, when moving a piece on the board. This is not the case for Go, where pieces (or stones) are not moved, but are placed, and therefore information about the *from* part is not relevant. The neural network needs to account for this, among other things.

Having the ability to save the progress of the network and the generated self-play data to files is very valuable, since training will take a considerable amount of time. If the program experiences a crash midway through execution, a lot of time waste could be prevented, if the data is saved every once in a while. This is also useful for saving data from runs with different configurations. If for example we

decide to run tests with different network architectures, we have the ability to save all the data for the different runs, and easily compare results of each.

4 Problem Analysis

In this section we will discuss the design choices that we have made during the implementation of our program, as well as the reasoning behind these choices. We will go over what games we considered, and what characteristics are important in a game. We also will go over how we have structured our version of AlphaZero, how we evaluate the performance of our program, what optimizations we have made, and considerations about how to structure and save the data generated by the program.

4.1 The games

Important game characteristics
In order to develop this program, we needed at least one game that the algorithm could learn, and attempt to master. Going beyond this, we wanted to implement at least two, in order to test and showcase the flexibility of the approach. We wanted to show that when implementing a new game, the game should require only an isolated implementation containing the game specific rules. If implementing a new game instead required a rewrite of aspects of the core algorithm, or required the use of band-aid solutions, it would fail the goal of being a game-agnostic algorithm. When choosing a game to use, several aspects of the game had to be considered. The game should be deterministic, so that the algorithm would know the possible outcomes of an action. It should be zero-sum in order to lower complexity, and it should be turn-based, in order to simplify when the players were able do specific actions. Lastly the game should have perfect knowledge, so that the AI would know all relevant information about the state of the game, and nothing would be hidden from view.

These attributes are important, but beyond these, the complexity and specific rules of the games are also very relevant. The complexity of the games should be at a level where no optimal solution exists or where the game has been "solved". However, the games should not be too complex, resulting in the AI not being able to train, and become competent, within the time and hardware constraints put on us by the premises of the project. Furthermore, the possible length of a game is also an important factor. If a game has a fixed max length, like Tic-Tac-Toe with nine maximum moves, the resulting game tree and state space will be less complex. The worst case game time will also be shorter, compared to a game with no fixed length like Chess.

The chosen games
During this project we have explored different games. Our decisions were based on the above described game characteristics, as well as our own initial knowledge of the games, and are supported by sources detailing their complexity[16]. Three games of varying styles and complexities were chosen: Latrunculi, Othello, and Connect Four.

Latrunculi was the first game we implemented and worked with. It is an ancient roman game that has recently been rediscovered and reconstructed by the french professor Ulrich Schädler[17]. The game has two phases. In the first, the players place their respective pieces on the board, and in the second phase, they try to capture the opponents pieces. We agreed to only work with the second phase, and generated random starting positions for the pieces using specific seeds for random number generation. Even so, this game proved to be too complex for our purposes. One issue was a high branching factor. More importantly though, is the fact that the game length is not fixed, and most games would take upwards of hundreds of turns to complete. This was especially prevalent if the AI's playing were making sub-optimal moves, or were very closely matched. We ended up largely discarding this game, and did not use it for training the network, because of these challenges which resulted in very slow training, and running times. Working with a game that is unknown to most people did make it interesting to implement, but it also added some complexity, as the only available rules did not cover certain edge-cases for the game. We expanded and clarified the rules where needed, to the best of our ability. These clarifications can be found in A.4.

Othello, a modern version of Reversi[18], was the second game we implemented. It is a less complex game, compared to Latrunculi, because it has a smaller branching factor of only 10, as well as 58

turns on average per game. Furthermore, after pieces are placed on the board, they are never removed again, unlike in Latrunculi. This means that the maximum length of any game is fixed, and is equal to the number of total squares on the board. This game is what most of our data in this paper is based upon. It is of fitting complexity for our program. It is complex enough to pose a challenge, while still being simple enough that our algorithm has a chance at becoming proficient at it, within the given time frame.

Finally, we also implemented Connect Four. It is a significantly simpler game, with both a smaller branching factor, and a fixed game length. This game has been solved[19] (for the classic 7x6 board), which we believe make it a less interesting game to use as a basis for testing our program. However, since the rules and complexity of this game is so simple, we made use of it early on, when attempting to get the program to a functional state. Because Connect Four is a solved game, we were able to compare the moves that our algorithm prioritized with the moves that statistical correspond to "perfect play". This is shown in 7.10. Connect Four was mostly used for our own testing, and the majority of our results in this paper is based on Othello.

Modular implementation of games
To make sure the implementation of games were modular, meaning each game could easily be replaced by another when needed, we took inspiration from the high level game API described in Artificial Intelligence - A Modern Approach[20]. It describes a number of methods that all the games should implement, to allow AI agents to access them, without the algorithm needing to adjust for the inner workings of the specific game.

4.2 Choosing which parameters to explore

Setting up and implementing a machine learning algorithm, introduces a lot of different parameters and configurations, needed for the learning process to function in an optimal manner. To figure out what parameters to focus our attention on, we first studied the AlphaZero paper, and found that some parameters were adjusted differently based on the complexity of the three games they researched. This included the exploration noise added during MCTS simulation. This was adjusted in relation to the legal actions in the game, and because we worked with other games the noise constant should be adjusted to match our games[4] (p. 17).

Most of the choices for parameters in the AlphaZero paper stems from the fact that they are designing an algorithm intended for use on specialized hardware. For instance, their value for total training steps (times the network will train over the duration of the programs runtime) was set to 700.000. We would never be able to match this number, so instead our task was to find reasonable alternatives for these values. This meant finding trade-offs where we would still be able to get proper results, but where the values would be more realistic for our purposes. This was also the case for the amount of game data being stored at once, since we only had a fraction of the hardware memory available to AlphaZero.

Some parameters were chosen to be explored and tested precisely because they had not been mentioned in the paper. For these, we simply wanted to learn more about how the algorithm behaved, and we wanted to see if we could discover new and interesting configurations that might produce better results. Many of the parameters were simply left untouched, and was deemed to be less impactful, or not interesting enough to explore in our limited time frame. These included the learning rate for the network, the weight decay and momentum for the gradient descent optimizer, and the weight initializer values.

4.3 Evaluating performance

To evaluate the performance of our program, we play it against other AIs, at regular intervals during the training of the network. The newest version of the network plays a certain number of games against each opponent, half being played as player 1 and, half as player 2. We play as each side to reduce the effect of either player possibly having an advantage, or to detect if our algorithm develops

a bias towards either player. The first opponent we play against is a simple AI that takes a random move each turn, naturally making it a very easy opponent. Evaluating against a random AI is desirable, because it is very fast, and will quickly gives us an indication of whether our algorithm actually learns during training.

After the network has trained for a bit, the rest of the opponents are introduced to the evaluation. The main one of these is the standard MCTS AI which implements the basic Monte Carlo Tree Search algorithm with random rollouts. Basic MCTS is relatively simple to implement and its strength can be easily adjusted by lowering or raising the number of simulations that it runs. It is also game agnostic, and does not rely on game specific heuristics in order to function properly. This is highly desirable since our goal is to utilize it on several different games.

Furthermore we also test the newest version of the network against several older generations. We refer to these older generations as *macro networks*. These networks are saved at every hundredth training epoch, and is evaluated against to see how the network improves compared to previous generations. If the network consistently wins against these older networks, it is a clear indication that it is learning.

Finally, minimax algorithms have been implemented for each game. We initially planned to evaluate against these as well, but found that they ran very slowly when several evaluation games had to be played in succession. Because of this, we do not use minimax for evaluation. Another reason for leaving it out is that the win rate of Katafanga when matched against it usually corresponds quite closely with the win rate against MCTS. Finally, minimax requires proper heuristics for each game, which would need to be properly verified in order for the strength of the AI to be properly quantified. This is not the case with MCTS, where a single number (the number of simulations) determine its strength. These reasons led us to avoid using minimax when evaluating, as we did not believe the benefits outweighed the drawbacks.

Graph-data representation
For visually representing the data gathered during evaluation the Pyplot library[21] was used. We wanted to include all relevant data points that could indicate a progression of the capabilities of the algorithm during training. This included not only the evaluation described above, but also loss data from the network.

We realized that having a total of four different graphs, with real time data, during training would be highly beneficial. One would contain the evaluation results, with a plot of the ratio of wins against the different alternate AIs. This graph is the most useful to indicate an improvement of the algorithm over time, and as such this was the main graph that we used to convince ourselves that the algorithm worked. Additionally, three other graphs were used, each monitoring the state of the network. These three graphs represented the overall loss, policy loss, and value loss, acquired when training the network. These figures gave an indication of how well the network was able to fit the data it was given, and how accurately it was able to predict the outcome of games, as well as the best actions to take in particular states. Splitting the loss into three separate graphs helped us diagnose errors with the data given to the network more easily. If for example the target values were incorrect, it could be detected quickly in the value loss graph, but would not necessarily be reflected in the other loss graphs.

4.4 Chosen language and frameworks

Python 3 was chosen based on our prior experience with the language, as well as its track record within the field, and its ability to handle machine learning through its libraries and frameworks. These frameworks consisting primarily of Keras[22] with the TensorFlow[23] backend.

Keras is built as a high level API for deep learning with neural networks. It supports CNTK, Theano, as well as the framework which we chose to work with: TensorFlow. Tensorflow allows for numerical computation on CPU, GPU or TPU devices, and is developed by Google.

The Python language has branched into version 2 and version 3 which are not exactly compatible.

Version 2 is slightly faster on some machines and in some environments, but has less functionality, and package development have drifted between the two major versions. We decided to stay on version 3 as speed testing showed little to no major improvement for our code, and we believed using this version supported our needs and was therefore the right choice.

4.5 Optimization

The main goal of this project was to implement a program that used the techniques of AlphaZero, to achieve interesting results within a reasonable amount of time. This meant that optimizing all aspects of our program was of utmost importance. The following sections describe our thoughts and efforts in making Katafanga run as fast as possible.

NumPy
The NumPy library is almost synonymous with standard python. It is a powerful package for working with multidimensional arrays and matrices, among other things. The logic of Numpy operations are implemented in C++, which means it is significantly faster than interpreted Python. It is used widely across the code base to take advantage of the faster arrays, lists and matrices, as well as being used in conjunction with the neural network, acting as tensors with data for the network to train and predict on.

Numba
The Numba library provides the option of adding just-in-time (JIT) compilation of snippets of python code. This enables compilation to machine code, making execution of the relevant snippet noticeably faster. We made use of this when implementing the rules for the games, and when implementing the minimax algorithm. Making Numba work required some refactorization of the code, as Numba is limited in its capability to optimize and compute more advanced data types and structures. This meant that third party libraries and classes would not work with Numba (with the exception of Numpy) and workarounds had to be made. We solved this by isolating the functions that would benefit the most from JIT compilation, moving them outside their respective classes, if they existed in any. We then made sure that these functions only operated on primitive data structures. An example of the speedup Numba provided could be seen when we implemented the rules for Othello. When running one round of MCTS, with basic rollouts and without the aid of the neural network, the execution time was around 50 seconds with 800 simulations. We then tried extracting the code for calculating possible actions, as well as the result of these, and for calculating whether a terminal game state was reached, to functions of their own. When using JIT compilation on these functions, the time spent dropped to just over 4 seconds.

Threads vs processes
The utilization of the available hardware is important, as a lot of the running time of the program is spend on generating data for the network to learn from. So a need for parallel execution is evident. In python, threads are running on the same CPU core as the process which spawns them. This means that on a modern CPU with multiple cores, the workload is not spread across the cores, and potential speedups from multi core execution is not gained. Threads are therefore not useful at the extent we need them to be. Luckily python contains a library for creating and managing processes. Superficially, they operate almost identically to threads, but allow for creation of separate processes within the underlying operating system. These processes contain their own interpreter as well as copies of all relevant objects and variables used in the program. This in turn allows for the processes to run on separate CPU cores which lets the program utilize the full power of the CPU.

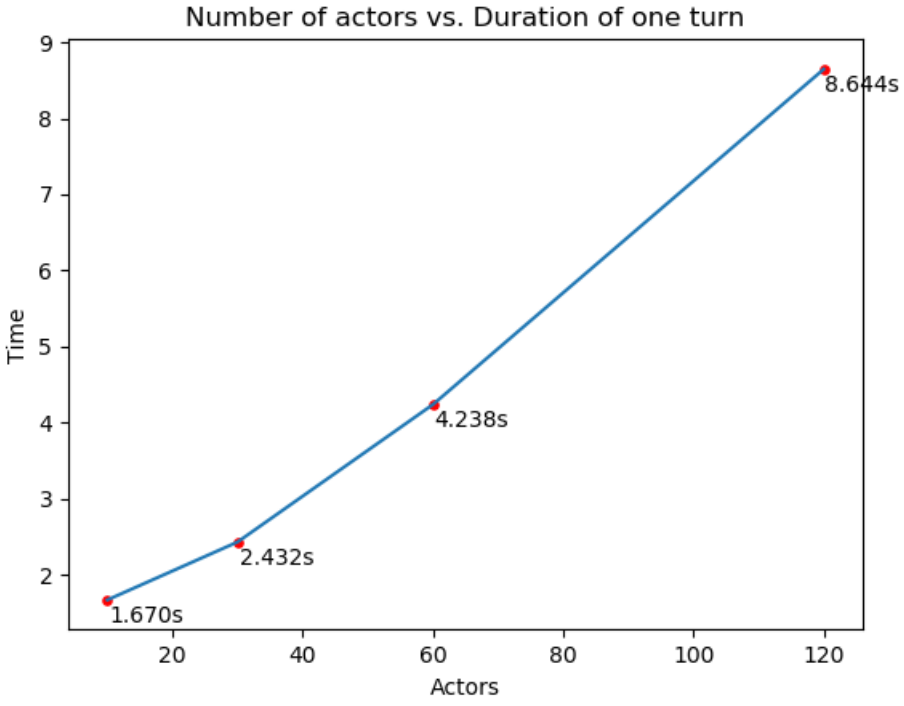
However, the use of processes pose challenges that does not occur when using threads. The main challenge is sharing data between them. This is more complicated, because processes by design are run in complete isolation. To enable sharing of data, we made use of the *Pipe* class, which is built into python. Pipes act like two way communication channels between processes. This way, each process can independently generate data and send it through pipes to a main process, which then stores the data. The main process also stores the neural network models, and the processes can request evaluation from the network through its pipe. This way the generation of data is parallelized over multiple

CPU cores, and much more data can be generated in a shorter duration. The main process then has the responsibility of storing and manipulating the given data, which includes organizing the data into batches for the network to train on.

Batch-play and GPU utilization
We realized very quickly that utilizing the GPU for training, and for predicting with the network, would be essential. GPUs can drastically speed up most operations used in machine learning, since a lot of these are based on vector and matrix operations, which is the bread and butter of GPUs. This is especially beneficial with convolutional neural networks where the majority of work is done with matrices, and where the most used operation is the dot-product between the convolutional kernels and the 2D input data. A small example showing the speed gain of using a GPU can be seen below. This shows the time taken for a single turn during self-play with 200 MCTS simulations, i.e. the time taken for 200 evaluations from the network. To see which GPU was used in this test, consult A.2.

Time taken for 200 network evaluations
With only CPU: 107.6 seconds.
With GPU: 6.0 seconds.

Evidently, using the GPU is absolutely essential. This was made easy by Tensorflow's excellent support for this. The challenging part was making the most of the high parallelization that a GPU offers. The most powerful GPU at our disposal had 2432 cores, which means a lot of the workload of the neural network could be distributed among these. To fully utilize this, we implemented the logic for self-play in a way that enabled games to be played in batches. The games were already being played on multiple processes, as described above, but each process also contains a number of actors. Each of the actors controls two self-playing agents, which continually generates game data. This data can then be combined for each actor, and later for each process, and prediction using the network can then be done in batches. Choosing the right amount of actors determined how much value we could squeeze out of the GPU. At a certain point, too many actors causes a bottleneck on the GPU, and do not provide any further benefit. Below is a graph showing how long taking one turn in a game of Othello took, for different amount of actors, where all actors are spread equally among the self-playing processes.



The duration per turn between 10 and 30 actors is less than doubled, while the amount of data

generated by the actors are tripled. This shows the benefit of using batches during self-play. The sweet spot is found at around 60 actors. Going any further than that, and the duration for a turn grows at a faster rate than the generation of data does.

4.6 Persistent storage

Using our own computers to train the network will result in some fairly long training sessions, and since we do not have equipment that can be dedicated solely to training the network, this presents a problem. Because of this, some manner of more permanent storage for the training data was needed, in order to save the progress between training sessions. There are several ways to go about this:
We could store training data and progress locally on each computer, allowing us to stop a training session and resume it later on, and potentially move this data manually to a different device to continue the training there. Alternatively, we could store this data remotely in a database, giving us all the benefits from local storage with the added benefit of more easily sharing this data between devices. Furthermore, we could expand this to allow us to have several devices simultaneously working towards training a single shared network.
We opted for the first solution, local storage. The reason for this is mainly because of the effort required to implement the second option. The remote storage solution would present additional challenges and complexity. We did implement a small scale solution which showed that it was possible to implement, but we still judged that a full scale implementation would not deliver sufficient benefit for the effort it would require.

4.7 Arena based learning

One addition to the training that could potentially be used, is the addition of an Arena function. This function would pit the current network against a number of earlier generation networks, and whichever network proved to be the most competent would be the one we trained moving forward. The intention behind this would be to insure that Katafanga is improving by discarding networks that have become worse at playing the game, compared to its previous iterations. This would potentially ensure a more steady progress towards a more competent game-playing AI.
However, we have chosen not to use this tactic for a number of reasons. First of all, we are unsure what effect this tactic would have, as AlphaZero does not do this, and whether or not that effect would be beneficial or detrimental. As an example, a network might train towards a strategy that would be an improvement in the long term, but has a negative effect to begin with. In this situation it would lose in the arena, be replaced by a previous generation, and the longer term advantage would never be attained. Furthermore, we worry whether this violates the pure self-play approach, which is one of AlphaZero's strengths, as we are affecting the development and learning of the network, beyond the self-play. With all of these concerns, we were not willing to invest the time and effort into testing out this tactic.

5 User's Guide and Examples

In this section, we will quickly go over how to run our program, describing the different possible parameter that can be used.

To execute the program, a python (version 3.6.8) environment needs to be installed, the environment consist of a number of libraries, most significant are Keras and Tensorflow, a detailed list of the environment can be seen in appendix A.1 The program is launched through a console, the execution differs depending on the arguments given. The arguments are structured as shown below:

python main.py [player1] [player2] [game] [board_size] [rand_seed] [<options>]

Player options: Human, Random, Minimax, MCTS, MCTS_Basic
Game options: Latrunculi, Connect_Four, Othello
Board size options: game specific (8 is generally the standard board size).
Random seed option: Used for testing, locks random generator for a known seed value e.g. 42, only used for playing Latrunculi.
If the argument given for a specific setting is ".", the standard setting will be used.

Other options: ['-s', '-l', '-lg', '-ln', '-v', '-t', '-c', '-g', '-p', '-ds', '-dl'] Where:

-s enables saving game data and network data to disk while playing
-l enables loading of learned data from disk, and continues from state
-lg load only games
-ln load only network
-v verbose debug information
-t activate timing
-c load additional configs, from custom config.txt file (loaded from resources/config.txt)
-g shows visual graphical interface of gameplay
-p shows statistical plots
-ds saves game data and network data to sql database while playing (unsupported)
-dl loads learned data from sql database and continues from state (unsupported)

Examples of standard runnable strings: Run Latrunculi as human player vs. Minimax on a 8x8 board with graphics

python main.py Human Minimax Latrunculi 8

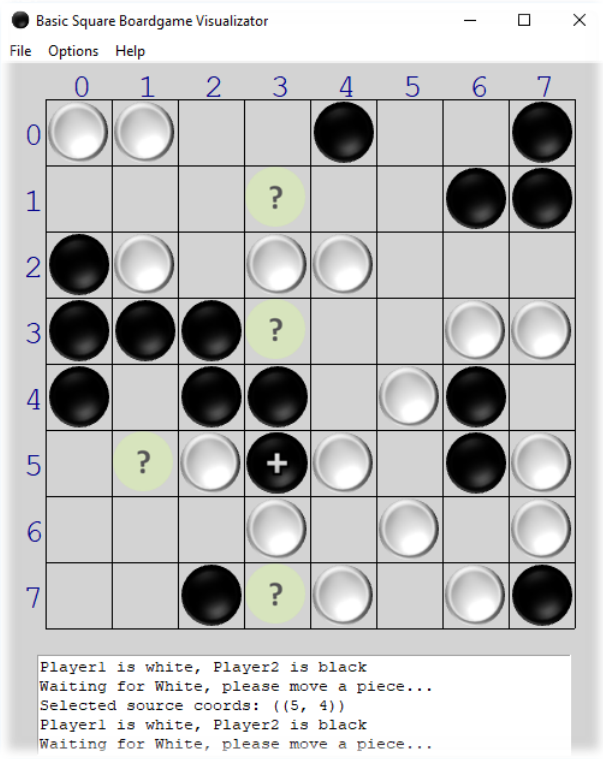


Figure 1: Graphical user interface for board games.

Run Connect Four in "AlphaZero" mode with training, game data saving, plotting statistics and additional configs on a 6x6 board without graphics.

```
python main.py . . Connect_Four 6 -s -p -c
```

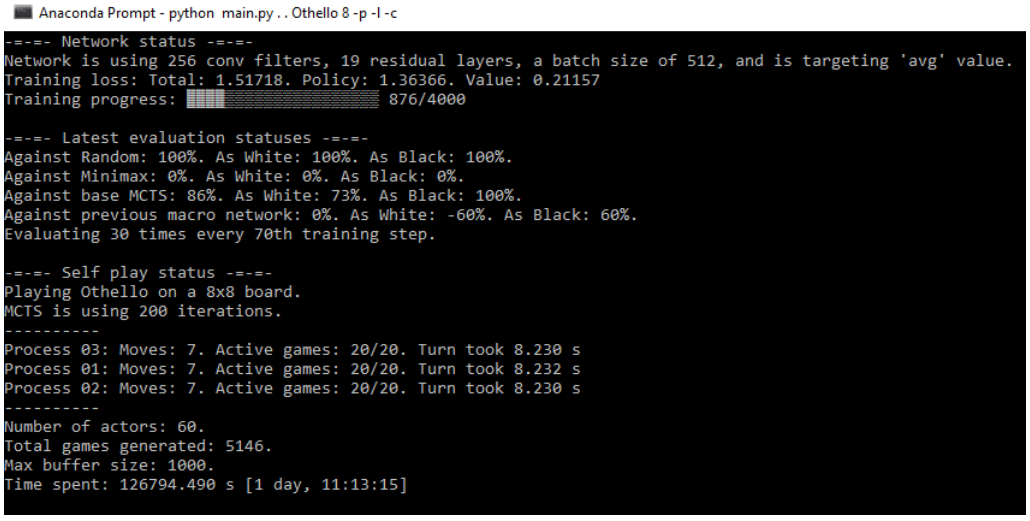


Figure 2: Console information about the network, self play and training progress.

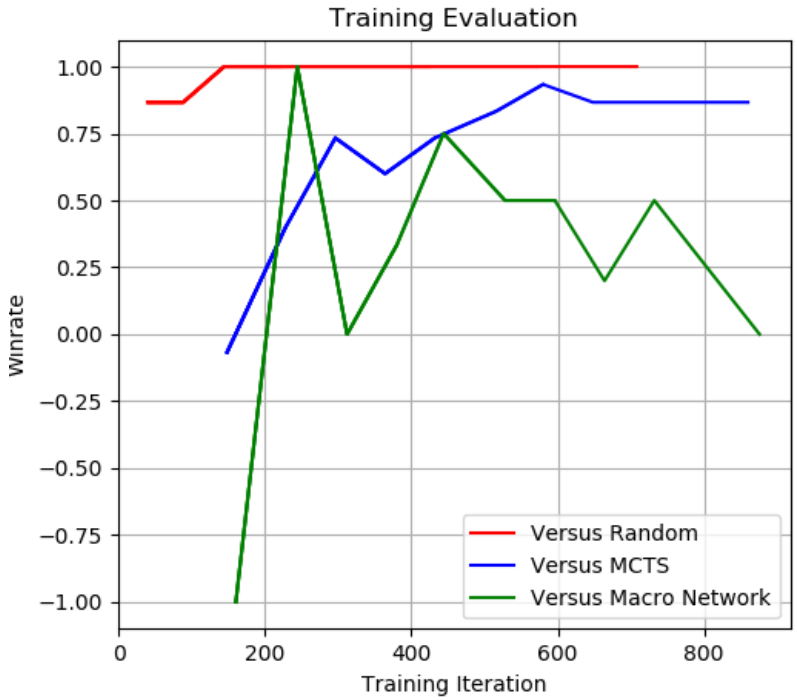


Figure 3: Plot data showing information about Policy loss, Average loss, Value loss and Training Evaluation.

To quit the program use "CTRL+C" on Windows/Linux.

Saved game data, plot images and report result are available in the directory *../resources/[game name]* Data will be overridden if a new game is initiated.

6 Technical Description of the Program

In this section we will discuss the technical details of our program. We will go over the key components and describe how they function and communicate.

Just below a diagram can be seen, describing the general structure of the program. When the Main module is called, it creates a number of new processes for Self_play, where the games will be played. It also starts Monitor and any GUI that might be needed. As the Self_play processes run, they ask the Monitor for evaluations from the network, which is passed on to the neural network. Games and networks are saved regularly. Once the training is done, the program terminates.

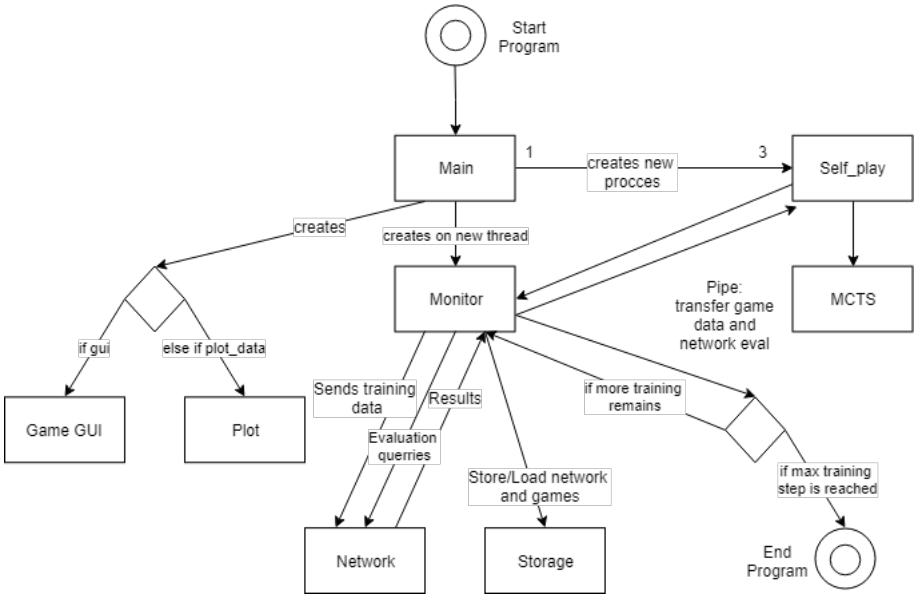


Figure 4: Activity diagram outlining the control flow between the subsystems of Katafanga

The program consists of the following main modules:

- controller - game logic, game agents, self-play, and training.
- model - neural network, states, and storage of networks/self-play data.
- view - GUI for visualizing game states and playing against the AIs using a visual interface, and graph logic for plotting telemetric data, about various parts of the program, during runtime.
- testing - unit tests for various parts of the program.

Additionally, files exist outside these modules. These include:

- main - loads command line arguments and initializes the program.
- test_main - runs all unit tests and summarizes the results.
- config - contains variables and constants used in various parts of the program, that can all be overridden in config.txt file.

6.1 Game

As previously discussed, several games have been implemented. These games share an abstract superclass called Game which contains a number of abstract methods. All games make use of a class called State which contains the information of a specific state. This information includes the game board, which player has the turn, and the location of pieces on the board. Each game overrides the following methods with their specific game rules.

- *actions(state)* - return all possible actions that can be made from a given state.
- *result(state, action)* - return the state resulting from taking a given action in a given state.
- *terminal_test(state)* - test whether a given state is a terminal state, meaning the game has ended.
- *utility(state, player)* - return the point reward for the given state for the given player, 1 = player won, 0 = draw (or state is not terminal), -1 = player lost.

Additionally, the games are responsible for transforming data from/to the neural network. The first three methods that do this are game specific, overridden by subclasses of Game.

- *structure_data(state)* - transform a state into a set of 2D arrays (tensors) containing binary data of where the current player pieces and opponent pieces are on the board. This method is overridden for each game, as each game might have a different number of specific piece types.
- *map_actions(actions, logits)* - mask out logits (which are returned from the network) that do not represent the given legal actions, and maps the actions to the relevant logits. Finally, normalizes these logits over all remaining legal logits.
- *map_visits(visits)* - structure visits of all actions for a given state (acquired by calling make_target(state)) into a form that can be accepted by the network. For Othello/Connect Four this is simply a vector of probabilities of selecting each action.

The following methods are defined in Game and are not overridden.

- *make_target(state_index)* - returns a tuple of expected value and policy output, which is used to train and calibrate the network with. The expected value is either the terminal value of the game, or MCTS' perceived value of the given state, which is partly based on the terminal value and partly on the values from the network, or an average of both of these. The policies are the visits that MCTS made for each action available in the given state.
- *store_search_statistics(node)* - stores information about a node used in MCTS. This includes the visits of that node's children (actions) and MCTS's perceived value of the state represented by the node (q-value).

6.2 Agents

The game playing agents also share a superclass called GameAI. This class only has the one abstract method execute_action(state), which is overridden by the specific game agents, where they calculate the best action and return the resulting state. The existing agents are outlined below.

- *human* - is used in conjunction with GUI game window when a human plays as either player. Adds a listener to the GUI and blocks until the user has selected an action.
- *random* - simply selects a random action for each state.
- *minimax* - superclass for the three game specific minimax algorithms. Implements alpha-beta pruning and transposition table. The heuristic function is overridden in game specific implementations.
- *minimax_othello*, *minimax_cf*, *minimax_latrunculi* - Overrides heuristic function for minimax as described above. Some also override cutoff function.
- *MCTS_Basic* - Implements MCTS with random rollouts.
- *MCTS* - Implements MCTS using the AlphaZero approach, with neural network evaluation etc.

6.3 MCTS

The version of MCTS used in conjunction with our network varies on some key points from a standard MCTS, like the one we evaluate against. The standard version uses four phases to achieve its goal. The first phase, *Selection*, is where nodes that should be explored by the algorithm are selected, starting from the node corresponding to the current game state, until a leaf is reached in the MCTS tree. The nodes are selected according to how they maximize the UCB formula:

$$v_i + C\sqrt{\frac{lnN}{n_i}} \tag{1}$$

Where v_i is the expected value of the node, n_i is the visit count of the node, N is the parent visit count, and C is the *exploration parameter*, used to control how highly the UCB formula weighs the exploration of new actions. This formula balances exploitation of high value nodes, with exploration of relatively unknown nodes.

If the leaf node that has been selected is not a terminal node, i.e. does not represent a game state where the game is over, *Expansion* creates the child nodes of the chosen node and chooses one of them. *Simulation* then runs a rollout from the chosen node, till the end of the game, choosing random moves. *Backpropagation*, takes the result of the game, found in the rollout, and propagates that from the chosen node, up to the current node. Updating value and visit count on each node on that path.

The modified version uses an evaluation from the neural network, rather than simulating rollouts. It is this value from the network that is then propagated back. The only exception is when a terminal state is hit. In this case, the utility value for the player at that node is used, instead of the value from the network. It should be noted, that when a value is received from the network it is inverted ($value * -1$) before being propagated up the MCTS tree. This is done so that the value reflects the perspective of the player that made the move which led to the state that is being evaluated. Furthermore, we use an extended version of UCB to select nodes called PUCT which stands for Polynomial Upper Confidence for Trees. This is the function that is also used in AlphaZero:

$$Q(s,a) + U(s,a) \tag{2}$$

Where $Q(s,a)$ is the average value for the node gained by taking action a in state s (expected value/visits). (s,a) is found by the following function:

$$U(s,a) = C(s) * P(s,a) * \frac{\sqrt{N(s)}}{1 + N(s,a)} \tag{3}$$

Where $P(s,a)$ is the prior probability of selecting action 'a' in state 's', which is acquired from the policy values form the network. $N(s)$ is the visit count of state s, and $N(s,a)$ is the visit count of the node gained by taking action 'a' in state 's'. $C(s)$ is found by the following function:

$$C(s) = ln(\frac{1 + N(s) + C_{base}}{C_{base}}) + C_{init} \tag{4}$$

Here C_{base} and C_{init} are both constants. This is how AlphaZero calculates the $C(s)$ value. We have, instead chosen to keep C(s) as a constant we call *explore_val*. While the function shown above is used in AlphaZero, they do say that it most often behaves as a constant: *"C(s) is the exploration rate, which grows slowly with search time... ... but is essentially constant during the fast training game"* [4] (p. 17).

Like UCB, PUCT also seeks to balance exploration with exploitation while including new factors like P(s,a). However, in order to further encourage exploration *Dirichlet noise* is added to the root node representing the state to take an action in, before MCTS simulations are run, but after the root

node has first been expanded. The noise is added to the prior probabilities for each child of this node. As mentioned, the prior probabilities are predicted by the network when expanding a node with new children. The Dirichlet noise is calculated by first acquiring a number of values, equal to the number of new children for the node, drawn from a *gamma distribution* [24]. This is done using the Numpy library method *numpy.random.gamma(shape, scale, size)*, where scale is set to 1 and size is the number of new children. Shape, which is called *NOISE_BASE* in our program is both game specific and board size specific. It depends on the average amount of possible actions in for a specific game or, in other words, the average branching factor. For Othello 8x8, this value is set to 0.7 which is roughly equal to $\frac{avg\_actions}{10}$, where avg_actions is ∼14. If more average actions are present the *NOISE_BASE* value will be smaller, and the gamma distribution will be more concentrated. After the values are drawn, the following formula is used for every child of the root node, in order to apply the noise to their prior probability.

$$P(s, a) = p * (1 - f) + g * f \tag{5}$$

Where p is the prior value for that node, g is the value drawn from the gamma distribution matching that node, and f, which in our program is called the *NOISE_FRACTION*, is the *concentration parameter*[25] of the Dirichlet distribution. This value specifies how concentrated or spread out the noise should be across the child nodes. This value is set to 0.25 in our program.
Finally, to further encourage exploration, softmax sampling[26] is used to add a small random chance of MCTS choosing a sub-optimal action when all simulations have been run. This only happens early in the game. Specifically, in Othello 8x8, during the first six moves. If MCTS is at a further point in the game than this, *argmax* is instead used on the action that was visited the most during MCTS simulation. Softmax sampling is mainly done because calculating the expected outcome of selecting specific actions early in the game is difficult. The sampling helps the program explore opening moves that might not seem appealing from its perspective. The formula for calculating the probabilities for each node to be selected from the softmax distribution can be seen below.

$$p(i) = \frac{e^{\beta * v_i}}{\sum_{i=1}^{K} e^{\beta * v_i}} \tag{6}$$

Where v_i is the visit count of node i, and β is the *temperature* which determines how high the variance should be between the probabilities. Higher temperature means nodes with higher visit counts are more likely to be picked, while flattening the distribution of nodes with smaller visit counts. A low temperature means all nodes are almost equally likely to be picked.

The code for MCTS can be found in the *controller/mcts.py* file. The code for calculating PUCT is found in the method named *ucb_score(node, parent_sqrt)* which takes as arguments a child node to be selected for or against, and the square root of its parents visit count. The code for adding noise to the priors of children of a root node can be found in the method *add_exploration_noise(node)* and the code for softmax sampling is found in *softmax_sample(child_nodes, visit_counts, temperature)*.

6.4 Neural network architecture

The neural network is implemented using the *Keras* framework in conjunction with *Tensorflow*, and resides in the file *model/neural.py*. It uses a sequential model where layers feed into each other, one at a time, except at the two outputs, where the network splits into the value and policy "heads". The overall structure of the network is based on the one used in the official AlphaZero implementation[4] (p. 18). We will not repeat how the network is structured as a whole as it is stated quite clearly in the aforementioned paper. To see an overview of the complete network consult appendix A.3. We will however describe in more detail the parameters we ended up using as well as alterations we made. These alterations are in part inspired by an implementation of AlphaZero for Connect Four[27].
The network consists of a total of 42 convolutional layers. The vast majority of these are used as part of residual blocks. The *residual blocks* are implemented based on those described in the paper about *Deep Residual Learning for Image Recognition* by Microsoft[28]. Each block contain two convolutional layers, each followed by a batch normalization and a *leaky relu* layer. The final leaky relu activation

is applied after the blocks and the skip connection has been merged together.

The construction of the network is made to be dynamic, so that game specific elements, such as the dimensions of input and output data, can be easily modified and new games can be added without much difficulty. These methods include *input_shape(game)*, *policy_head(game, prev_layer)*, and *value_head(game, prev_layer)*. If for example the network is constructed with Latrunculi in mind, the shape of the input data will be different from Othello. As with all aspects of our program, almost all parameters in the network can be configured using config files to allow for easier testing and tuning.

Using the network to predict or train on data is done on the GPU. This is made possible by using the GPU compatible versions of Tensorflow and Keras, and by setting Tensorflow specific configs, which control the amount of memory that Tensorflow is allowed to use. We implemented a method which make use of Tensorflows native profiling tools for computing how large a model is, which is determined by the amount of parameters (weights, kernel values, biases etc.) that can be trained. This method also prints the FLOPS (floating point operations per second) of the network, when it is used to predict values. FLOPS are a good indication of the throughput of the network, i.e. how many calculations per second can be made using the network. If the network can be made to efficiently run in parallel on the GPU, when predicting values, the throughput goes up.

6.5 Self-play

Self-play is the file where all types of games with all types of agents are played. This takes place in a method called *play_games* which takes as input a list of games, two lists of players (player 1's and player 2's), a config file, and some optional arguments. Because the program is structured around parallel play, the games are played in "batches". This means that at each turn an action is taken for every game in the given list, using the corresponding player 1/player 2 agent for that game. Playing games in batches is primarily used in conjunction with MCTS, because parallelization on the GPU is then fully utilized, as described earlier. When not running the program in training mode, i.e. if one or both of the players are not managed by MCTS, batch play is not used, and a single game is played. When batch play is used, a function called *play_as_mcts* is used to handle many different MCTS actors playing at once. This is done by splitting up the different MCTS operations into the following methods utilized by *play_as_mcts*.

- *create_roots(batch_data)* - For each game in batch, calls MCTS function *create_root_node(state)* where state is the current game state. This initializes the MCTS tree and returns the root of that tree.
- *expand_nodes(batch_data, nodes, policies, values)* - For each game in batch, calls MCTS function *set_evaluation_data(node, policy, value)* where node, policy, value is a node to expand, with the given policy logits and value, acquired from the network.
- *prepare_actions(batch_data, roots)* - For each game in batch, calls MCTS function *prepare_action(root)* where root is the root acquired from *create_roots*. This calls MCTS which adds the exploration noise to the prior probabilities of the children of root.
- *select_nodes(batch_data, roots)* - For each game in batch, calls MCTS function *select(root)* where root is the root created in the method above. This runs MCTS *selection* on the roots, and returns the nodes that should be simulated by MCTS.
- *backprop_nodes(batch_data, nodes, values)* - For each game in batch, calls MCTS function *backpropagate(node, node.state.player, -value)* where node is a selected and expanded node, and value is acquired from the network.

6.6 Storage

The storage module handles saving/loading of game data and of the networks during self-play and training. This happens in the *ReplayStorage* and *NetworkStorage* classes. When a game is over, the *Game* class is serialized and sent to the monitor method. Here it is passed along to the *ReplayStorage*

instance and saved in memory. If the program is set to save to disk, the game will also be saved as binary data on the disk. The same occurs when the network has finished a training epoch. This happens directly in the monitor method, where the network is sent to *NetworkStorage* and saved in memory. Again, it is also saved to disk if this is enabled. The folders where data is saved depends on the game being played. If for example Othello is being played, the data will be saved to */resources/Othello/replays* and */resources/Othello/networks*, for games and networks respectively. Equivalent methods exist for loading this data, and are used if the program is set to load from disk on startup. These methods are called *load_replay* and *load_network_from_file*. Additionally, *NetworkStorage* also handles saving and loading of *macro networks*, which are older networks, used when evaluating the current network, to see if it has become smarter than previous generations.

Finally, *ReplayStorage* contains another essential method called *sample_batch*. This method selects random states from random games saved in the game buffer located in the class. It does so by first selecting a number of games equal to the desired batch size. The chance of selecting each game is determined by how long the history of the game is which is equal to the amount of moves that were made in that game. Then, for each game chosen, a random game state in that game is chosen to be part of the training data. The training data then consists of the input to the network for each state (acquired by calling *game.structure_data()*), as well as the expected outputs for the network to match up against. The outputs are acquired by calling *game.make_target()*.

7 Results

In this section we will go over a number of different test runs where we train Katafanga with different parameters and configurations. First we will go over the standard parameters used by each test. We will then go over the specific runs. In each run, we will use a variation on the standard set of parameters, describe what we expected from that test run, and what results we actually got back. In the end of the section, we will sum up our collected results.

AlphaZero is, as mentioned, an algorithm with a vast reliance on constants and hyperparameters that all need specific tuning for the algorithm to provide optimal results. Taking into account the sheer amount of possible combinations of parameters, as well as the time needed to produce useful results, it would not be feasible to test them all. This meant that some parameters had to be left "as is", and only those that was deemed to have a large effect on the results of the algorithm was tested. Below are explanations of what the parameters are. Indicated in square brackets are the default values for the specific parameter, which can be assumed to not change unless explicitly specified in the test.

- **Game [Othello][8]:** The game being trained on, as well as the board size used.
- **MCTS simulations [200]:** How many iterations of selection, expansion, evaluation, and backpropagation MCTS uses before selecting an action to make.
- **Residual layers [19]:** The amount of residual blocks used in the network.
- **Convolutional filters [256]:** The amount of filters in each convolutional layer.
- **Batch size [512]:** The amount of training examples that the network learns from at each training epoch.
- **Games per training [5]:** The amount of games that are played between each epoch of training.
- **Evaluation games [30]:** The amount of games being played against alternate AI's at each evaluation point. 15 are played as player 1, 15 as player 2.
- **Game storage [1000]:** The amount of games saved at one time during training. Training batches are drawn from random game states from these games.
- **Dirichlet noise base [0.7]:** Indicates the amount of noise to add to the prior probabilities of nodes. Roughly estimated by dividing the avg number of actions in a given state by 10.
- **Target value [avg]:** Indicates how to calculate the value of a game state, which is what the value head of the network trains on. The possibilities for target value includes:
  - z: The terminal value of the game (1 for win, 0 for draw, -1 for loss) is used as target value. This is also called the z value.
  - q: The expected value of a state, calculated by MCTS in conjunction with the neural network, is used as target value. This is called the q value.
  - avg: The average of z and q is used.
  - mixed: A linear falloff between z and q is used. This means that z is weighed less and less as training goes on (up to a maximum of 600 training epochs), and q is weighed more and more. After 600 epochs, only the q value is used.

Excerpts of the test runs are shown below, indicating what parameters were tested, as well as the expected outcome and actual results of the tests. The images depict three graphs with three respective colors, indicating these three evaluations:

- Red: Win rate against an AI making random moves.
- Blue: Win rate against standard MCTS with random rollouts (using the same amount of iterations as the modified MCTS).

– Green: Win rate against five previous generations of the neural network, called macro networks. For example, at training epoch 700, the algorithm would evaluate against networks at generation 600, 500, 400, 300, and 200.

7.1 Preface concerning the results

The following tests all contain a flaw with the way evaluation against MCTS is done. Sometimes Katafanga will lose to the basic implementation of MCTS during evaluation. We were not able to reproduce this at all when observing these algorithms playing games, one at a time, against one another. Not even if the basic MCTS was given 2000 simulations, and Katafanga only used 200. Katafanga would still confidently beat MCTS in all the games we observed, both as the starting player, and when playing second. Therefore we theorize that a flaw with the way batch-play is handled in our program causes this error. Perhaps some data gets switched around, or perhaps older data is not properly deleted, which confuses the algorithm and causes it to select a suboptimal move. In any case, this is important to keep in mind when observing the graphs in this section. Luckily, this error seemed to be equally prevalent in all test runs, so comparisons between the performance of the different runs should still be valid.
Apart from this, partway through conducting the tests we also discovered two additional less impactful errors. The first error concerned the first test and half of the second, in which the data underlying the graph for evaluations against macro networks was not correct. This means that the macro evaluations for test one as well as those before training iteration 600 in test two, are not usable. After these two tests, the error was fixed, and the data for the macro networks should be correct.
The other error was found after the sixth test. The error had to do with the noise applied to prior probabilities of the root node's children, before MCTS simulations. This noise was greater than intended. As mentioned, the noise applied is mainly controlled by the *NOISE_BASE* parameter, which was erroneously set to 2.5, instead of the intended 0.7, for Othello 8x8. We believe that this may have caused Katafanga to have a hard time solidifying a proper game playing strategy, and has most likely also influenced the results to some degree.

Finally, it is worth noting that the hardware used during testing vary between each test. We had to use every available resource, in order to get a proper sample of results in the limited time we had. This means that if a test is shorter in duration, it does not necessarily mean that an improvement or optimization was made in our algorithm. The duration could simply be because of variations in the hardware used. Even though these errors and varying factors apply, we still believe the results of the tests are worth discussing. The error with batch play was present during all tests, and the increased noise was present during all tests, except test 7 and 8. The tests would mostly have been affected equally by the errors, and the varying power of our hardware only impacted the duration of the tests, not the results.

The duration of test runs were determined by how many training iterations the network would complete. We chose to aim for at least 1000 iterations, as this assured that the network had time to stabilize itself, and allowed enough time to evaluate against marco networks and MCTS. This is not nearly the 700.000 iterations that was done in the original AlphaZero training, but since just 1000 iterations took upwards of 1-2 days to complete, we had to set certain limitations.

7.2 Test 1: First functional run on Othello 8x8

Figure 5: Test 1: Evaluation results during first functional run on Othello 8x8

Test duration
2151 training steps.
2 days and 20 hours.

Special parameters
None.

Purpose
This was the first long run after we were fairly confident that the program worked as intended. We wanted to test the stability of the program, to see whether 100% win rate against MCTS could be reached.

Results
This test run shows that Katafanga is capable of learning by self-play, and is eventually able to beat an MCTS AI which uses the same constraints that it does. After the graph peaked at 100% win rate against MCTS, it fluctuated at around 75-90%. We wondered whether this could be because of our limited game storage. After 1000 games are saved, the oldest are deleted, which could mean that the neural network "forgets" older strategies by re-adjusting weights that previously encoded such information (see test #7, where we investigate this hypothesis). The reason for these fluctuations is most likely due to the error with batch-play and evaluation, as mentioned previously, but we were not aware of this error when first conducting this test.

7.3 Test 2: Smaller network

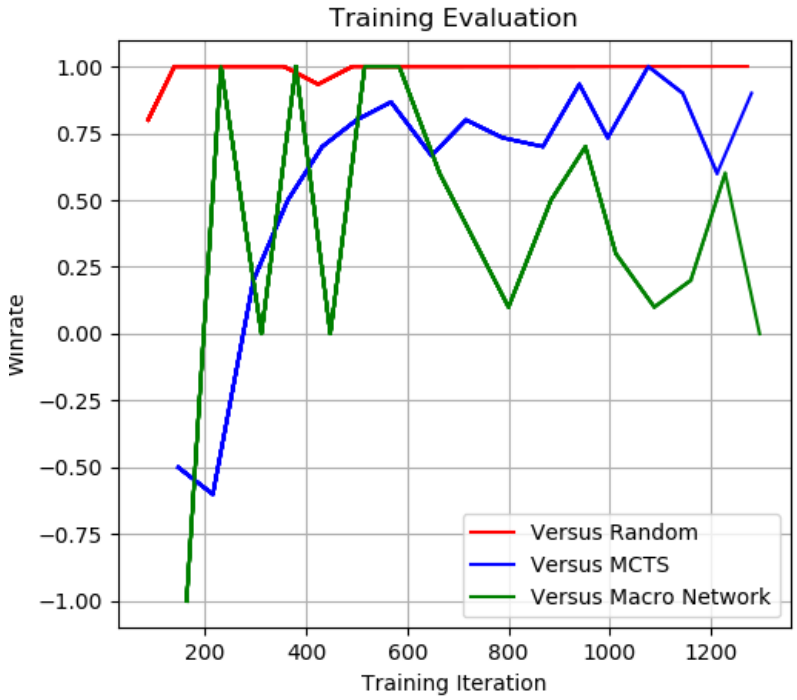


Figure 6: Test 2: Evaluation results for the smaller network

Test duration
1296 training steps.
1 day and 1 hour.

Special parameters
Smaller network with 11 residual blocks and 128 convolutional filters.

Purpose
This test was meant to show the tradeoff of using a smaller network, where training it would be faster, but where performance might be worse. We wanted to see if this tradeoff was worth it. If training was significantly faster and performance was only slightly worse, or not noticeable at all, the smaller network would be wiser to use for our purpose.

Results
To get an idea of the difference in scale between this network and the full size one, we utilized the method described earlier for computing the trainable parameters. For the bigger network the parameters numbered 134.8 million, while the smaller one reduced this number to 39.2 million. This means that the smaller network has a lot less work to do, when adjusting its weights which naturally lower the time needed for training. The potential downside is that a lot less information can be encoded in the weights. For games with a large amount of legal game states, this might mean that information is lost and the network might not be able to converge on a proper strategy. In our case, this does not seem to be the case. The network learns rather well and even hits 100% win rate against MCTS after around 1000 epochs. This suggests that Othello is simple enough that a smaller network is sufficient to encode the necessary information.

7.4 Test 3: Exclusively using Q-value

Figure 7: Test 3: Evaluation results for only using q value

Test duration
1100 training steps.
1 days and 3.5 hours.

Special parameters
Using only q-value as the target value.

Purpose
This was to see the effect of only using the q-value as a target value. We were interested in whether this would improve or decrease the learning rate, or if it would have no effect, compared to the standard setting, which is to use an average of q- and z-value.

Results
This test shows a significantly worse performance, when using q-value alone, compared to the average. By step 1000 we have about 25% win-rate against standard MCTS, compared to about 75% win-rate at step thousand, in test 1, where we use the average target value. However, Katafanga does continue to get better, as can be seen by its 75% win-rate against earlier networks. This is especially clear when compared to the macro network graph, seen in the next test for Z-value alone, where the win-rate against the macro networks is significantly more erratic. When using q-value exclusively, we see a much more stable and consistent improvement compared to previous networks.

7.5 Test 4: Exclusively using Z-value

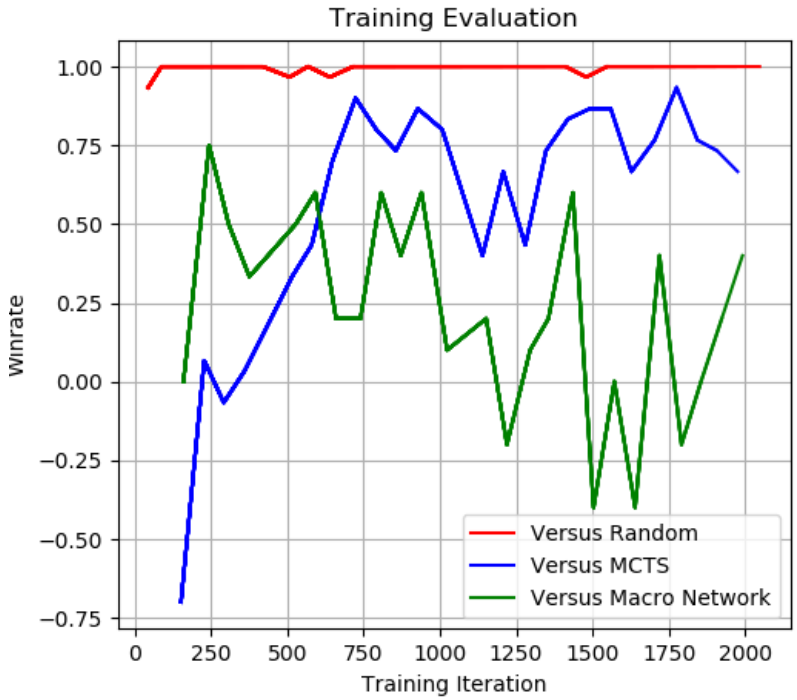


Figure 8: Test 4: Evaluation results for only using z value

Test duration
2052 training steps.
2 days and 6.25 hours.

Special parameters
Using only z-value as the target value.

Purpose
This was to see the effect of only using the z-value as a target value. We were interested in whether this would improve or decrease the learning rate, or if it would have no effect, compared to the standard setting which is to use an average of q- and z-value.

Results
This test shows a significantly quicker learning rate than using q-value alone, and is comparable to using the average value. This test reaches an 80% average win-rate against MCTS after 700 training steps which is very similar to using the average target value. However, the performance of Katafanga fails to stay consistently high, even dipping below 50% win-rate. The win-rate against the macro networks also varies wildly, and it suggests that the network seems to learn very little after step 1000. It seems that there does seem to be some advantages to using z-value exclusively in the early training. However, when the network becomes more competent, some of the other settings tested might be more efficient, such as using the average of z- and q-value.

7.6 Test 5: 400 MCTS simulations

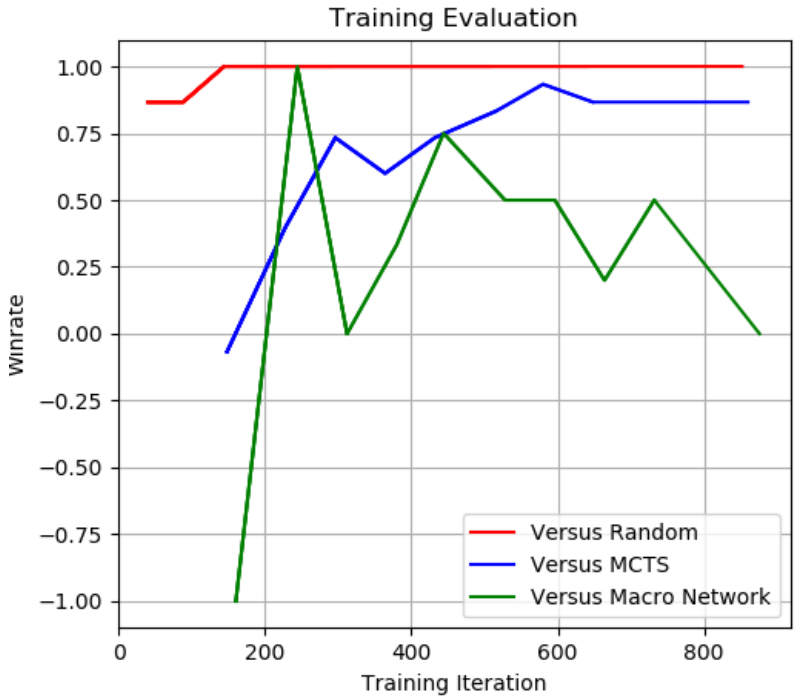


Figure 9: Test 5: Evaluation results for using 400 MCTS simulations

Test duration
876 training steps.
1 day and 11 hours.

Special parameters
Using 400 MCTS simulations, instead of 200.

Purpose
This test was supposed to show whether the trade-off between spending more time running MCTS simulations, and generating less data to train on, was worth it. The idea was that with more simulations being run, the statistical likelihood of finding a good action would be higher.

Results
The test shows a higher initial win rate against MCTS than previously. This seems to indicate that the algorithm learns more with fewer training epochs. However, as expected, doubling the MCTS simulations also doubled the time needed for each training epoch. Interestingly, the performance of the network seemed to flatten out at around 600-700 training steps. This might indicate that the network reached an equilibrium of sorts, and can't learn any more with the data it has available, or it could be a coincidence. It is worth noting that the basic MCTS being evaluated against also uses 400 iterations for this test, so that both AIs plays under similar conditions.

7.7 Test 6: q/z-values with linear falloff

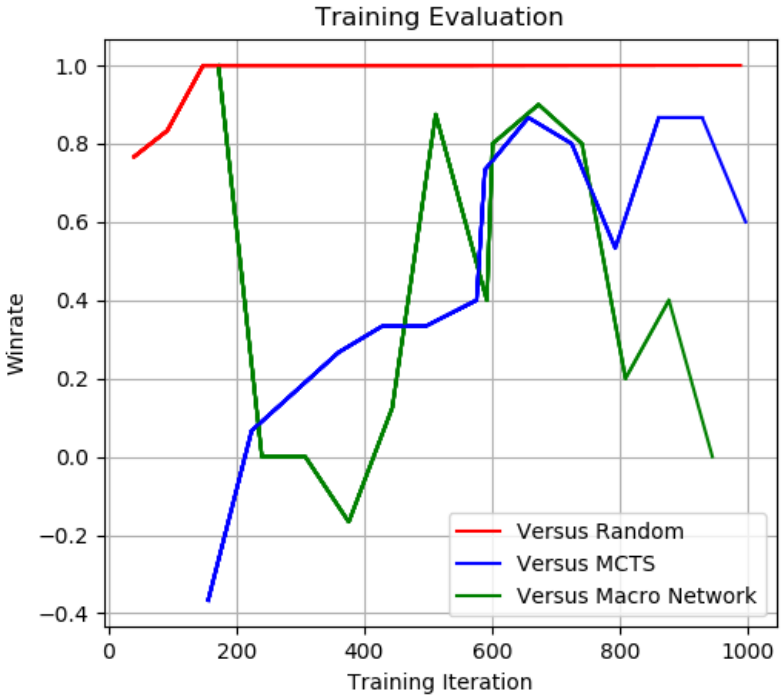


Figure 10: Test 6: Evaluation results for using q/z linear falloff

Test duration
1000 training steps.
1 day and 10 hours.

Special parameters
Using linear falloff between z and q-value (called mixed in parameter explanation).

Purpose
At this point we realized that using only q as target value was hindering performance at the start of the training, and it took awhile for the program to get good results. When using only z the results were better early in the training, and less so later in the training. We therefore decided to test a mix between z-value and q-value. The training starts out using only z-value, which then gets weighted less and less as the training goes on. Compared to the q-value which gets weighted higher and higher as the training progresses. After 600 epochs, only q-value is used as the target value.

Results
During the first 600 iterations, the win-rate against MCTS climbs slowly while the win-rate against the macro-networks is very unstable. After the 600 steps, the win-rate gets better against MCTS, though the win-rate against the macro-networks drops drastically. This suggests little to no improvements at each epoch, towards the end of this training session. This was not a great success. The very unstable learning rate is likely due to the target value constantly changing which is supported by the significant jump in win-rate at the moment when the training switches to pure q-value. Better results might be gained from changing when it goes to pure q-value, or perhaps have the progression end when the target value is equal to the average of q- and z-value. However these results are significantly worse than just using the average, so we choose not to examine the mixed target further.

7.8 Test 7: Growing game buffer

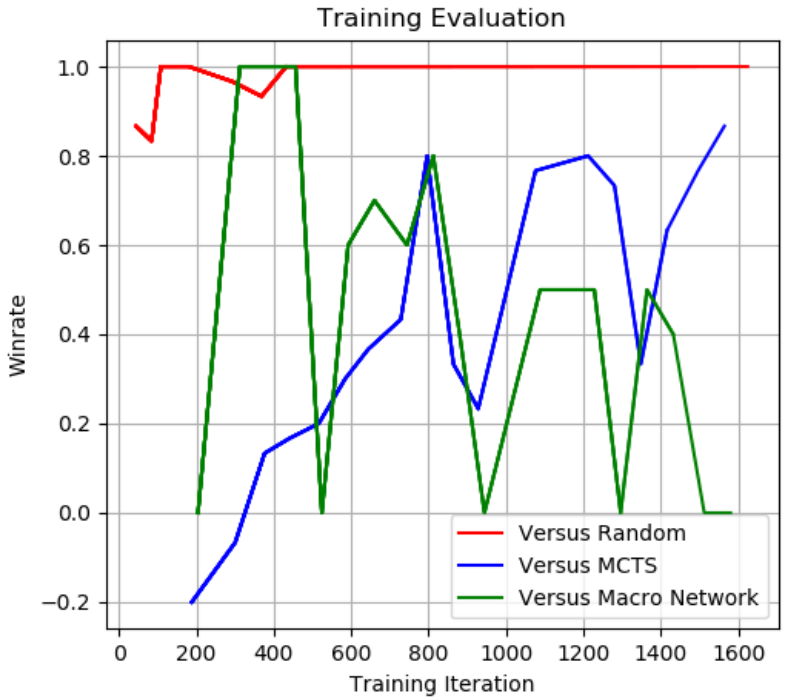


Figure 11: Test 7: Evaluation results for utilizing a growing game storage

Test duration
1652 training steps.
1 day and 6 hours.

Special parameters
Gradually increasing the amount of game data stored as well as using a smaller network, also used in test #2.

Purpose
This test was meant to investigate whether storing more game data as the training went on was beneficial. The idea was that as the network got smarter, the data that the network helped generate would become of higher "quality" and would be worth saving. We also had the thought that maybe the network "forgot" older viable strategies, and that saving the data for longer could help alleviate this.

Results
The test did not seem to improve how well the program performed, and on the contrary it seemed that the network had a harder time learning a proper strategy which is made evident by the large fluctuations of the win rate. However, the win rate against MCTS does seem to spike at the end and reach the 83% which is equal to the highest spike that most of the tests reach, so perhaps training is simply slower using this method, but might reach the same performance as previous tests.

7.9 Test 8: Noise base value at 0.7

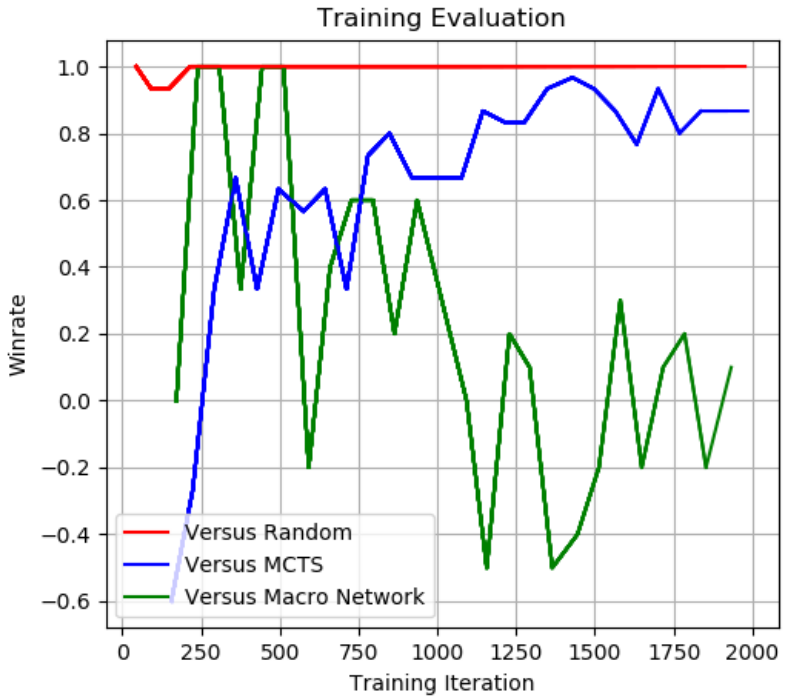


Figure 12: Test 8: Evaluation results for using a noise base value of 0.7

Test duration
1997 training steps.
1 day and 14 hours.

Special parameters
NOISE_BASE parameter set to 0.7 as well as making use of a smaller network identical to the one used in test #2.

Purpose
As described earlier, because of a mistake made, our noise was set to 2.5 and not 0.7 as expected. This test tries to make up for it by using the intended noise value.

Results
The test is fairly fast when it comes to winning against random and show increasing and valid results against MCTS. But somehow the learning falls does not seem stable when it comes to the uneven result of the macro network. Here, the algorithm seems to perform sub-par against earlier generations of the network.

7.10 Validating against "perfect play"

Verifying Katafanga against existing AIs gives a good idea of the capabilities of the algorithm. In order to get a slightly more theoretical and statistical foundation on which to evaluate the strength of our program, we decided to compare the way Katafanga plays against the theoretical "perfect" way to play. This would give an idea of how optimal the choices made by Katafanga really were. Since Connect Four is the simplest of the games we implemented, we did the test with this game. We made use of the Masters Thesis, titled *A Knowledge-based Approach of Connect-Four*[19] as well as the online game solver for Connect Four[29] to compare against, and to understand what the perfect game in Connect Four would look like.

It should be noted that all Connect Four solvers, theoretical and functional, are based on the classic 7x6 board. However since non-square board sizes are not well supported by our program, we went with the 7x7 option. The optimal game might be slightly different for a 7x7 board, but we still believed the test provided a good indication of whether our program was on the right track.

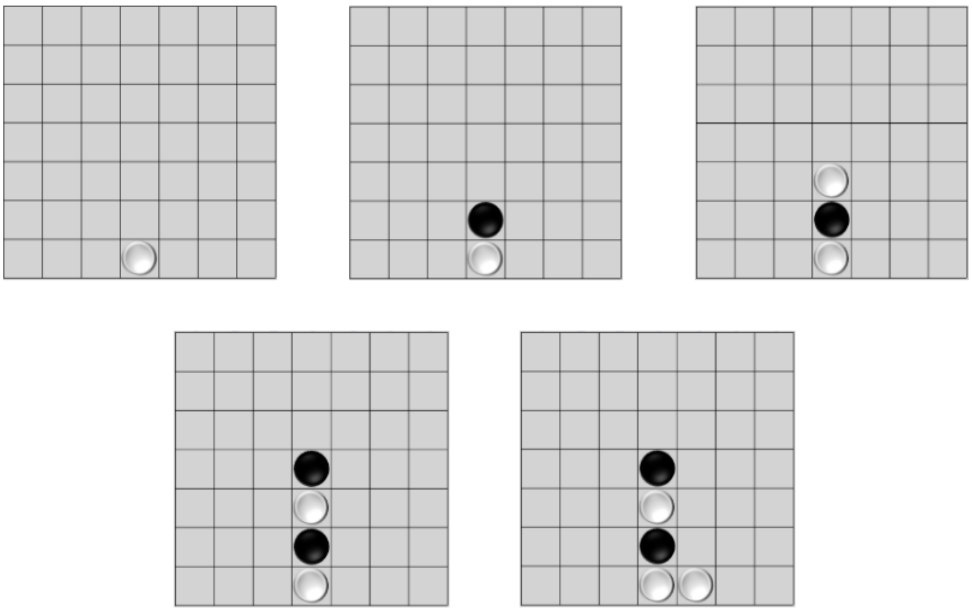


Figure 13: Sequence of moves by Katafanga in Connect Four.

The images above show the progression of a game where Katafanga plays against itself on a 7x7 board. The first four moves correspond nicely with what the Connect Four solver believes are the most optimal moves and will (with optimal/perfect play from each side) result in white winning. Inspecting the output from the algorithm reveals that values from the neural network backs up the assumption that white will win. For example, when white has the first turn, the expected value of placing a piece in the middle column is 0.22. The network correctly assesses that white has the upper hand, and believes that white has a roughly 61% chance of winning the game after placing its first piece. On the other hand, at the second move, when black places its first piece, the expected value of the move is -0.20. Again, this backs up the fact that black should lose if the game continues on this path.

At the fifth move, the algorithm no longer recognizes the optimal move which would have been to place a piece in the middle column. When this test was run, the algorithm had only trained for around five hours, and had most likely not encountered this game state long enough to accurately assess where to place the next piece.

7.11 Summary of results

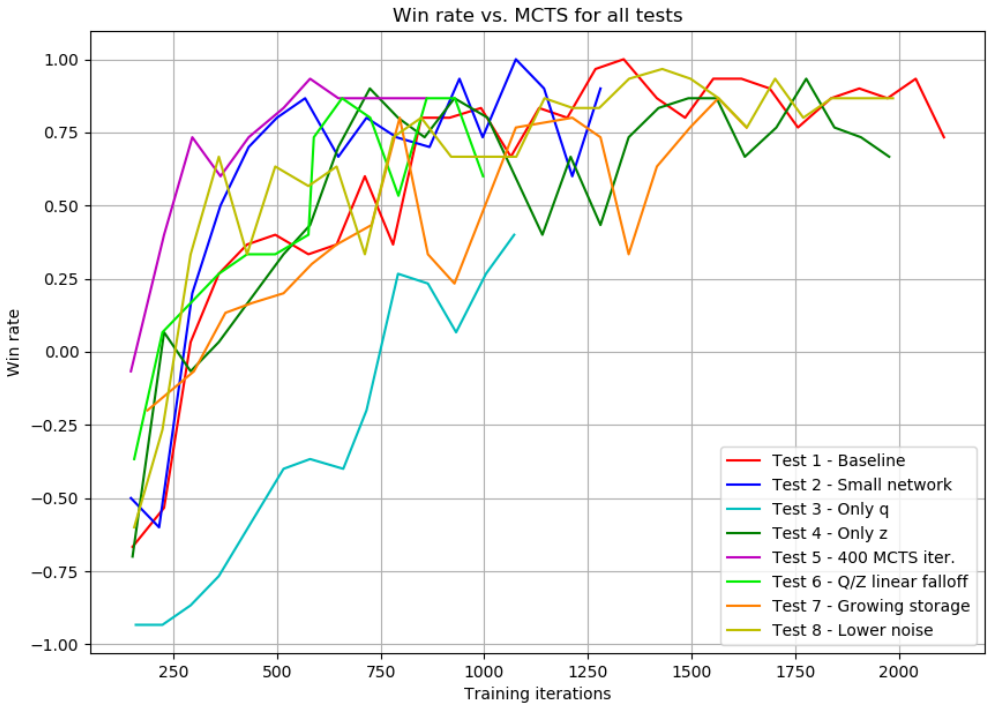


Figure 14: Combined win rates for all tests.

As can be seen from the graph above, all our tests produces fairly similar results. The most noticeable outlier being the test using only q value as target. It has a significantly worse performance, compared to the average. As might be expected, the best early win-rate is achieved by the test using 400 MCTS iterations, compared to the standard 200 iterations. It gets to search through a significantly larger portion of the state space, and as such, makes decisions on more reliable information. This results in a better performance, even though the standard MCTS is also using 400 iterations.

The tests described in this section barely scratches the surface of the thousands upon thousands of possible values for the parameters of AlphaZero, and the combinations between them. Because of this, and because of our limited time frame, we attempted to test parameters which we deemed would have the most impact on the results of running the program. We have also tried creating tests that target different parts of the program, instead of focusing more deeply on one or two particular aspects of the algorithm. We believe this was the best way to create varied and meaningful data within a short amount of time, and served as a way to better learn how the different parts of the program would be affected by different configurations. These tests are by no means conclusive, but they still provide an insight into how versatile AlphaZero is.

8 Discussion

In this section we will discuss the evaluation of Katafanga's performance, as well as future and related works.

8.1 Evaluating the performance of Katafanga

When training, we evaluate against at least one classical algorithm, that being Monte Carlo Tree Search, or MCTS. Against our implementation of this algorithm, Katafanga does fairly well, having an average win rate of about 80%. However, it should be noted that we mostly run MCTS with only 200 simulations, which is a relatively low amount, and a higher number of simulations might give different results. We have also tested against minimax, even if it is not included in the regular evaluation because it proved to be too slow. These results show that the win rate for minimax usually corresponds quite closely with the win rate of MCTS, but MCTS was generally sligthly stronger, and Katafanga would beat minimax consistently after a moderate amount of training time. It should be noted, that this is our own version of minimax, and the handcrafted heuristics it uses could most likely be improved.

8.2 Future work

Some of the future work that might be conducted includes further code run time optimization, perhaps utilizing tools like *profiling* for diving deeper into the parts of the program that might act as bottlenecks. This would require changes to the code as threading / parallelism makes profiling difficult, something we experienced during the project.
For solved games with "perfect play", like Connect Four, we would have liked to create an additional way to evaluate our algorithm. For each training iteration, one could automatically run a few evaluation games where a database of moves that make up perfect play, would be matches against moves made by Katafanga. This would give a clear indication that the algorithm converges towards optimal play.

As far as the neural network goes, a lot of work could be put into testing a wider variety of different parameter settings, trying to find more optimal settings. In the limited time frame, there was many parameters that we did not get a change to test.

8.3 Related work

Since the public introduction of AlphaZero, a lot of people have taken interest in replicating their work. This includes professionals as well as amateurs.

Oracle Developers did some work and shared their experience with implementation and exploration of AlphaZero in the article[30] at "Medium.com", it's written by Aditya Prasad (Jun, 2018). We took inspiration from this article for several of our tests, and based some of our ideas about which parameters to explore, on work described in this article.

Leela Zero[31] is an open-source project[32] with the focus of creating and training a re-implementation of AlphaZero for Go (alt. version for Chess also exist). Their self-play and training use "crowd-computing", where volunteers offer their devices to generate data. This data is then collected and trained on, and the overall algorithm becomes stronger. An article about Leela Zero can be found at *Medium.com*[33].

A walk-through of implementing AlphaZero using custom TensorFlow operations and a custom Python C module can be found at *Towards Data Science*[34]

9 Conclusion

In this section we will conclude on how well we solved the task proposed by this project, and what findings we can draw based on our problem statement. Below is a recap of the problem statement where we state what we set out to examine.

> *Can algorithms like AlphaZero realistically be made to achieve better results, for instance measured by winrate, when run on consumer grade hardware, or are traditional algorithms, tailored for specific games, better suited when no specialized hardware is used?*

Within the time frame of this project we have managed to train our algorithm to an extent where it plays competently in Othello and Connect Four. The particular algorithms include a random AI, MCTS AI, as well as the previous generations of the same network. The results shows that our program does indeed learn and improve when training the network on a relatively simple game. However, this does not mean that it is feasible to train our program to play Chess or even Go, like the DeepMind team did with AlphaZero. The most complicated game we have implemented, to run our program on, is Latrunculi. By our estimate the complexity of this game is comparable to Chess, if slightly less complex overall. We have to use our own estimate, as it is a reconstructed game, and not much information is known about it. The problem is that training our algorithm on Latrunculi takes a while. We do not have much recent data on this, as we abandoned the game halfway through the project in favor of Othello and Connect Four. However, what data we do have suggests that it would take significantly longer to train our network to play Latrunculi than Othello and Connect Four. It then follows that this increase in training time would be even more pronounced when training the algorithm to play Chess or especially Go. This could mean that attempting to train our program for more complex games might take too long to be practical, even if it might be possible.

For games of moderate complexity, our program is able to confidently beat traditional algorithms, and as such we can confidently say that when using the AlphaZero approach, it is indeed viable to train and run the algorithm on consumer grade hardware. However, if the goal is to become better than the best computer program currently available, or even the best human player, the situation might be different. Since the progress of learning becomes slower and slower the more the network trains, getting to a high enough level, even in simple games, might not be feasible within a reasonable amount of time.

In direct answer to our problem statement, AlphaZero like algorithms can be made to achieve good results against traditional algorithms when trained on consumer grade hardware. However, for more complex games the training would likely be too slow. Furthermore, while we have beaten our own traditional algorithms, using Katafanga, it might still lose to algorithms that use more competently crafted heuristics. This shows that in some specific situations, an AlphaZero like algorithm can indeed achieve better results than more traditional algorithms, even when trained without specialized hardware.

10 References and Bibliography

References

[1] P. S. Foundation, "Python," May 2019, accessed: 13-05-2019. [Online]. Available: https://www.python.org/

[2] S. Sharma, "Monte carlo tree search," August 2018, accessed: 13-05-2019. [Online]. Available: https://towardsdatascience.com/monte-carlo-tree-search-158a917a8baa

[3] P. Jason Brownlee, "A gentle introduction to convolutional layers for deep learning neural networks," April 2019, accessed: 13-05-2019. [Online]. Available: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks

[4] e. a. David Silver, Thomas Hubert, "A general reinforcement learning algorithm that masters chess, shogi and go through self-play," *Science*, 2018, accessed: 13-05-2019. [Online]. Available: https://deepmind.com/documents/260/alphazero-preprint.pdf

[5] D. Yang and T. Romstad, "Strong open source chess engine," May 2019, accessed: 13-05-2019. [Online]. Available: https://stockfishchess.org/

[6] M. Takizawa, "Strong open source evaluation function and book file for computer shogi," January 2011, (Japanese). Accessed 13-05-2019. [Online]. Available: https://github.com/mk-takizawa/elmo_for_learn

[7] L. Baker and F. Hui, "Innovations of alphago," April 2017, accessed 13-05-2019. [Online]. Available: https://deepmind.com/blog/innovations-alphago/

[8] "Wikipedia: Grandmaster (chess)," May 2019, accessed: 14-05-2019. [Online]. Available: https://en.wikipedia.org/wiki/Grandmaster_(chess)

[9] Deepmind, "Alphago overview," January 2016, accessed 27-04-2019. [Online]. Available: https://deepmind.com/research/alphago/

[10] D. Hassabis and D. Silver, "Alphago zero introduction," October 2017, accessed 13-05-2019. [Online]. Available: https://deepmind.com/blog/alphago-zero-learning-scratch/

[11] "Wikipedia: Go (game)," April 2019, accessed: 13-05-2019. [Online]. Available: https://en.wikipedia.org/wiki/Go_(game)

[12] "Wikipedia: Lee sedol," April 2019, accessed: 13-05-2019. [Online]. Available: https://en.wikipedia.org/wiki/Lee_Sedol

[13] "Wikipedia: Minimax," May 2019, accessed: 13-05-2019. [Online]. Available: https://en.wikipedia.org/wiki/Minimax

[14] D. P. Kaz Sato, Cliff Young, "An in-depth look at google's first tensor processing unit (tpu)," May 2017, accessed: 14-05-2019. [Online]. Available: https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[15] "Central processing unit (cpu) vs graphics processing unit (gpu) vs tensor processing unit (tpu)," May 2019, accessed: 14-05-2019. [Online]. Available: https://iq.opengenus.org/cpu-vs-gpu-vs-tpu/

[16] "Wikipedia: Game complexity," May 2019, accessed: 13-05-2019. [Online]. Available: https://en.wikipedia.org/wiki/Game_complexity

[17] U. Schädler, "Latrunculi, a forgotten roman game of strategy reconstruction," May 2019, accessed: 13-05-2019. [Online]. Available: http://history.chess.free.fr/papers/Schadler%202001.pdf

[18] "Wikipedia: Reversi is a strategy board game," April 2019, accessed: 13-05-2019. [Online]. Available: https://en.wikipedia.org/wiki/Reversi

[19] V. Allis, "A knowledge-based approach of connect-four," October 1988, accessed: 13-05-2019. [Online]. Available: http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf

[20] S. J. Russell and P. Norvig, *Artificial Intelligence A Modern Approach Third Edition.* Upper Saddle River, New Jersey 07458: Pearson Education, Inc, 2010.

[21] Matplotlib, "Pyplot tutorial," May 2019, accessed: 13-05-2019. [Online]. Available: https://matplotlib.org/tutorials/introductory/pyplot.html

[22] "Keras: The python deep learning library," May 2019, accessed: 13-05-2019. [Online]. Available: https://keras.io/

[23] "Tensorflow core - keras guide," May 2019, accessed: 13-05-2019. [Online]. Available: https://www.tensorflow.org/guide/keras

[24] "Wikipedia: Gamma distribution," May 2019, accessed: 13-05-2019. [Online]. Available: https://en.wikipedia.org/wiki/Gamma_distribution

[25] "Wikipedia: Concentration parameter," May 2019, accessed: 13-05-2019. [Online]. Available: https://en.wikipedia.org/wiki/Concentration_parameter

[26] M. Lee, "Softmax action selection," April 2005, accessed: 13-05-2019. [Online]. Available: http://incompleteideas.net/book/ebook/node17.html

[27] D. Foster, "How to build your own alphazero ai using python and keras," Januar 2018, accessed: 13-05-2019. [Online]. Available: https://medium.com/applied-data-science/how-to-build-your-own-alphazero-ai-using-python-and-keras-7f664945c188

[28] S. R. J. S. Kaiming He, Xiangyu Zhang, "Deep residual learning for image recognition," *Microsoft Research*, 2015, accessed: 13-05-2019. [Online]. Available: https://arxiv.org/pdf/1512.03385.pdf

[29] gamesolver.org, "Connect 4 solver," May 2019, accessed: 13-05-2019. [Online]. Available: https://connect4.gamesolver.org

[30] A. Prasad, "Lessons from implementing alphazero," June 2018, accessed: 13-05-2019. [Online]. Available: https://medium.com/oracledevs/lessons-from-implementing-alphazero-7e36e9054191

[31] "Leela zero website," May 2019, accessed: 13-05-2019. [Online]. Available: http://zero.sjeng.org/home

[32] "Leela zero github," May 2019, accessed: 13-05-2019. [Online]. Available: https://github.com/leela-zero/leela-zero/blob/master/README.md

[33] B. Schultz, "How a chess program called 'leela' is breaking new ground in chess and ai," April 2018, accessed: 13-05-2019. [Online]. Available: https://medium.com/@benschultz_57614/how-a-chess-program-called-leela-is-breaking-new-ground-in-chess-and-ai-bad9125f9458

[34] C. 2007, "Alphazero implementation and tutorial," December 2018, accessed: 13-05-2019. [Online]. Available: https://towardsdatascience.com/alphazero-implementation-and-tutorial-f4324d65fdfc

[35] "Cpuboss - intel core i7 4790," May 2019, accessed: 14-05-2019. [Online]. Available: http://cpuboss.com/cpu/Intel-Core-i7-4790

[36] T. et al, "Nvidia geforce gtx 1070 ti," May 2019, accessed: 13-05-2019. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-gtx-1070-ti.c3010

A Appendix

A.1 Program python 3.6.8 environment information
A.2 Hardware difference between AlphaZero and Katafanga
A.3 Diagram of neural network
A.4 Rules of Latrunculi
A.5 Rules of Othello (Reversi)
A.6 Rules of Connect Four

A.1 Program python 3.6.8 environment information

Name	Version	Build	Channel
_tflow_select	2.1.0	gpu	anaconda
absl-py	0.7.0	py36_0	anaconda
astor	0.7.1	py36_0	anaconda
astroid	2.2.5	py36_0	
blas	1.0	mkl	anaconda
ca-certificates	2019.1.23	0	
certifi	2019.3.9	py36_0	
colorama	0.4.1	py36_0	
cudatoolkit	9.0	1	anaconda
cudnn	7.1.4	cuda9.0_0	anaconda
cycler	0.10.0	pypi_0	pypi
gast	0.2.2	py36_0	anaconda
grpcio	1.16.1	py36h351948d_1	
h5py	2.9.0	py36h5e291fa_0	anaconda
hdf5	1.10.4	h7ebc959_0	anaconda
icc_rt	2019.0.0	h0cc432a_1	anaconda
intel-openmp	2019.1	144	anaconda
isort	4.3.8	py36_0	
keras	2.2.4	pypi_0	pypi
keras-applications	1.0.6	py36_0	anaconda
keras-preprocessing	1.0.5	py36_0	anaconda
kiwisolver	1.0.1	pypi_0	pypi
lazy-object-proxy	1.3.1	py36hfa6e2cd_2	
libprotobuf	3.6.1	h7bd577a_0	anaconda
llvmlite	0.27.0	pypi_0	pypi
markdown	3.0.1	py36_0	anaconda
matplotlib	3.0.2	pypi_0	pypi
mccabe	0.6.1	py36_1	
mkl	2019.1	144	anaconda
mkl_fft	1.0.10	py36h14836fe_0	anaconda
mkl_random	1.0.2	py36h343c172_0	anaconda
mysql-connector	2.1.6	pypi_0	pypi
numba	0.42.0	pypi_0	pypi
numpy	1.15.4	py36h19fb1c0_0	
numpy-base	1.15.4	py36hc3f5095_0	anaconda
openssl	1.1.1b	he774522_1	

A.1 table Continued

Name	Version	Build	Channel
pip	19.0.1	py36_0	
protobuf	3.6.1	py36h33f27b4_0	anaconda
pylint	2.3.1	py36_0	
pyparsing	2.3.1	pypi_0	pypi
pyreadline	2.1	py36_1	anaconda
python	3.6.8	h9f7ef89_1	
python-dateutil	2.8.0	pypi_0	pypi
pyyaml	3.13	pypi_0	pypi
scipy	1.2.0	py36h29ff71c_0	anaconda
setuptools	40.8.0	py36_0	
six	1.12.0	py36_0	anaconda
sqlite	3.26.0	he774522_0	
tensorboard	1.12.2	py36h33f27b4_0	anaconda
tensorflow	1.12.0	gpu_py36ha5f9131_0	anaconda
tensorflow-base	1.12.0	gpu_py36h6e53903_0	anaconda
tensorflow-gpu	1.12.0	h0d30ee6_0	anaconda
termcolor	1.1.0	py36_1	anaconda
typed-ast	1.3.1	py36he774522_0	
vc	14.1	h21ff451_3	anaconda
vs2015_runtime	15.5.2	3	anaconda
werkzeug	0.14.1	py36_0	anaconda
wheel	0.32.3	py36_0	
wincertstore	0.2	py36h7fe50ca_0	
wrapt	1.11.1	py36he774522_0	
zlib	1.2.11	h62dcd97_3	anaconda

A.2 Hardware difference between AlphaZero and Katafanga

The hardware used by Google Deepmind for AlphaZero consisted of:
5000x 1. gen TPUs for game generation
75x 2. gen for training.

Google's reason for building their a new processing unit, is based on the growing need for machine learning tasks, used for their own services like Gmail, Translate, Image Search, their datacenter (cloud) customers and more. Regular CPUs are made for general purpose computations, whereas GPUs are made for vector computations. While CPUs and GPUs can be used for training neural networks, they are not nearly as efficient as a TPU. TPUs are more efficient in the comparison between power-consummation vs. computation power, compared to GPUs or CPUs, which for Google is a major factor in their large datacenters.

What differentiates TPUs (matrix) from CPUs (scalar) and GPUs (vector) is the way it is able to handle matrix operations, making it extremely efficient for machine learning operations.

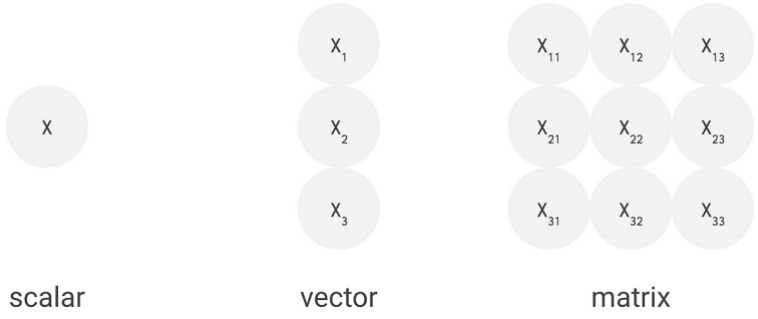


Figure 15: Computation of scalar, vector and matrix [14]

Our own consumer grade hardware setup a variety of laptops and workstations, all containing Intel CPUs and dedicated graphics cards (GPUs). The most powerful of our devices being a workstation containing an Intel I7 4790 CPU[35] and a NVIDIA GTX 1070 TI[36] mid-highend GPU.
In our implementation the CPU is used for MCTS simulations and the GPU is used for network evaluation and training. As seen in the table from Google below, our computation power is far lower, even if you only compare it to a single TPU, not even considering that Google used thousands of units.

Number of operations per cycle between CPU, GPU and TPU.[14]

	Operations per cycle
CPU	a few
CPU	(vector extension) tens
GPU	tens of thousands
TPU	hundreds of thousands, up to 128K

To be able to make a direct comparison between our setup and the TPU, we would have to be able to execute our code, on Google hardware. This would allow us able to calculate the throughput of our program in floating points operations per seconds (flops). As this is not possible, we can only conclude based on the above, that AlphaZero utilizes hardware that are several orders of magnitudes more powerful than what Katafanga has available.
From Google whitepaper - In-Datacenter Performance Analysis of a Tensor Processing Unit[14].

A.3 Diagram of neural network

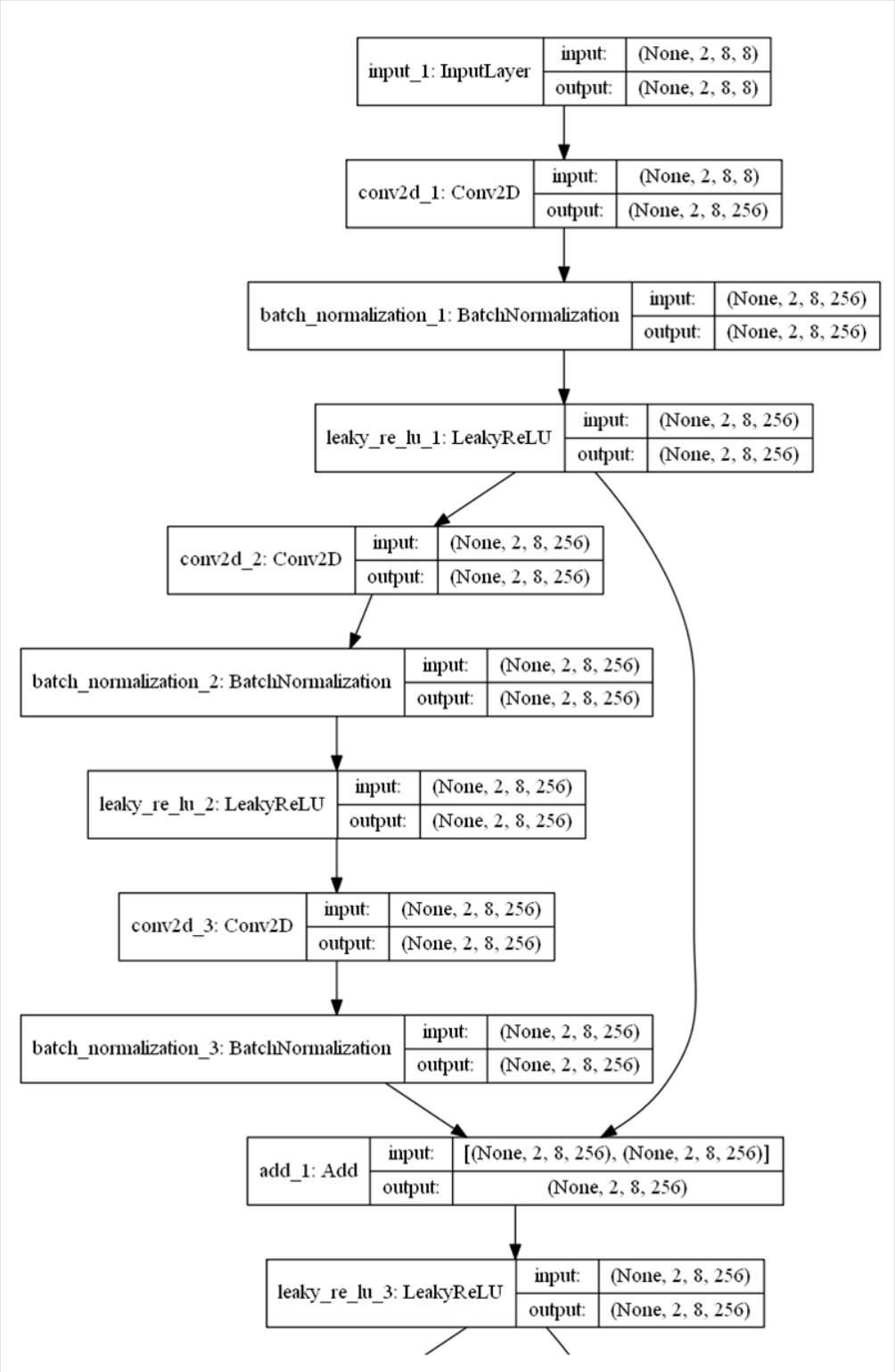


Figure 16: Neural network (input and first residual block

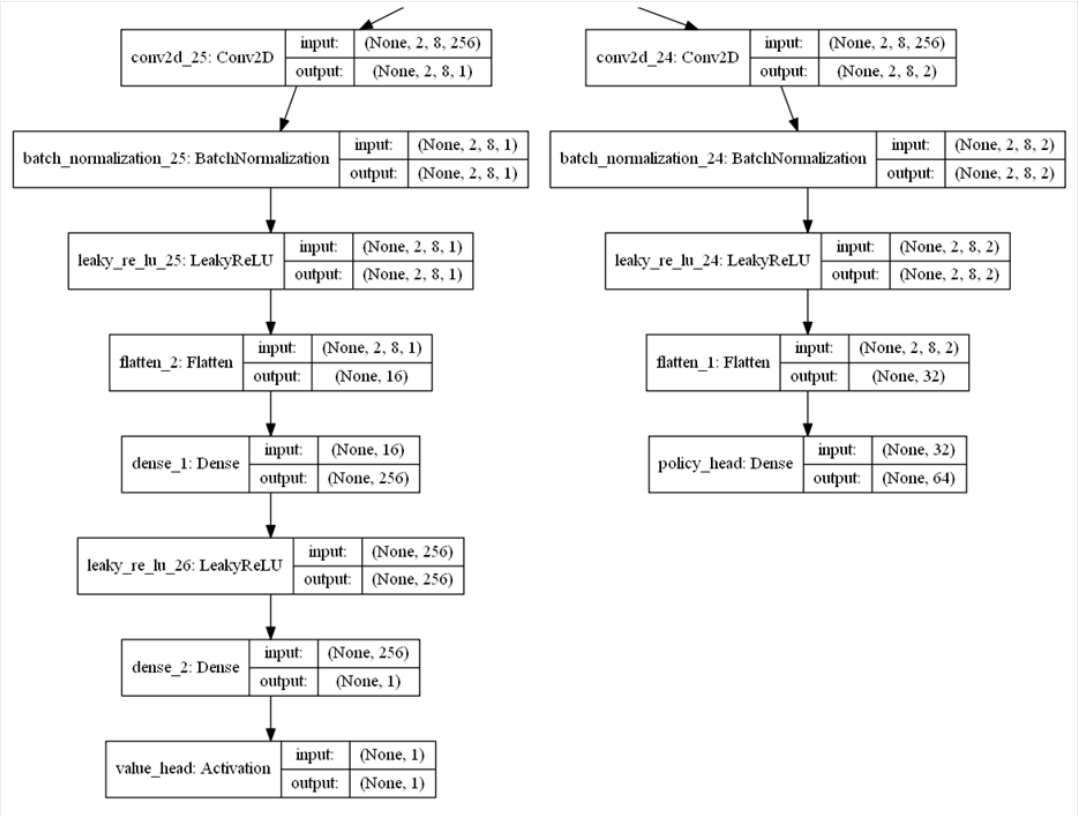


Figure 17: Neural network (value and policy heads))

A.4 Rules of Latrunculi

We have based our implementation of this game on the rules outlined in the paper titled *Latrunculi – A Forgotten Roman Game of Strategy Reconstructed*, written in 2001 by *Ulrich Shädler*. This paper outlines the basic rules and outcome of different moves. These rules are paraphrased below.

- Players take turns placing their initial pieces. Usually a total of 16 pieces are placed for each player.
- Players take turns moving their pieces. Only orthogonal moves are allowed. A piece can jump over one other piece of any color, if the square it would land on is clear. Several jumps in succession in one turn is allowed.
- If a piece gets trapped between two pieces of a different color (a piece above and below or to the left and right) then that piece will become "captured", and can't move.
- If a piece is captured, the player that captured it can choose to use their turn to remove that captured piece from the board, provided that piece is still being captured. A captured piece can be freed if any of the pieces capturing it are moved or captured themselves.
- A player can move a piece between two opposing pieces (called "suicide") only if this would result in one of those pieces being captured.
- If a player has only one piece left on the board, that player loses the game.

The number of starting pieces is between 1/2 and 1/3 of the total squares on the board. For our purpose, we start the game with a number of pieces matching half he number of squares on the board.

Not all outcomes of moves are documented, in the given article. Several moves will lead to cases, where ambiguity might occur. We cover these cases in the following section, where we make our own choices as to the result of these moves. We generally choose the outcome that would be the simplest to implement.
Special cases Case #1

	B1	
B2	W1	B3
	B4	

Figure 18: Case #1

Either of the four black pieces can move or be captured, and $W1$ will still remain captured. As long as at least two adjacent pieces remain $W1$ will stay captured.

Case #2

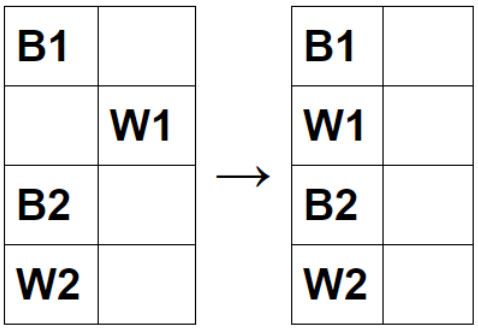


Figure 19: Case #2

When $W1$ moves, it, and $W2$, will capture $B2$, even though this move would not normally be possible, because $B1$ and $B2$ would capture $W1$. Several alternatives of this case exist, but in all of them the moving/attacking piece wins the "tie" and captures the relevant piece(s).

Case #3

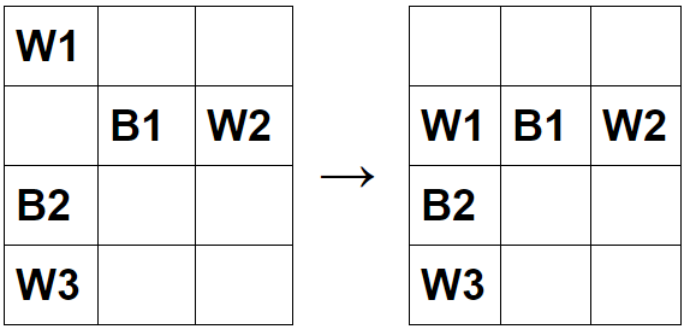


Figure 20: Case #3

When $W1$ makes this move, both $B1$ and $B2$ will become captured.

A.5 Rules of Othello (Reversi)

The rules of our Othello implementation is based on the modern version of Reversi found at the article of the game at Wikipedia[18]. But with a small difference as the while / black colors are inverted in our game, so that player 1 is white, and player to is black.

A.6 Rules of Connect Four

Connect Four is played on a board with 6 rows and 7 column, where two players takes turn placing their own colored piece in a row, resulting in the piece getting layered from the bottom up in that row. A player can win by getting his or hers four pieces placed side-by-side, horizontal, vertical, or diagonal.