General Game Playing AI for Java

Project Report

Group 16 - Java Artificial General GamER (JAGGER)

Thomas van den Berg
s2532743

Daniel Botnarenco
s2386593

Dominik Myśliwiec
s2545411

Thom Harbers
s2621533

Caz Saaltink
s2511878

April 21, 2023

Abstract

Modern knowledge and advancements in Artificial Intelligence inspired our client Tom van Dijk, who became interested in a Java deep-learning-based general game playing AI. Based on DeepMind's research, we developed JAGGER—an AlphaZero-like AI in Java, using Deeplearning4J (DL4J). Our system is based on Monte Carlo tree search using predictions from the neural network as the main element of its heuristic. Self-play in each iteration of training using the tree search allowed for the generation of a dataset, to which the neural network was fitted. After training the network, it should be more accurate in its predictions, enhancing the used search heuristic. Despite the poor documentation of the used DL4J library, we were able to create an AI capable of learning how to play games that implement the given Java interface. In our testing, we found that the AI learns how to play the game, but it might become worse over time due to overfitting. We suspect that the performance of JAGGER could be improved by the appropriate tuning of the training hyperparameters. Unfortunately, considering the staggering amount of time it took to complete enough training of the AI for a relevant result, we did not have time for that. We believe it would be beneficial to find a way of tuning the parameters reliably, experiment with them, and research whether it is worth using JAGGER for playing simple games considering the performance drawbacks over non-machine-learning-based solutions.

Keywords — General Game Playing, Java, AlphaZero, JAGGER, Monte Carlo tree search, Deeplearning4J

Preface

This report presents the results of our design project in the ”Design Project” module of the Technical Computer Science bachelor at the University of Twente. The project was guided by our project supervisor, Tom van Dijk. As a team, we would like to thank Tom, who guided us in the project, promptly responded to our concerns, and showed interest in our work. His feedback was tremendously helpful throughout this process. Tom also grounded us when necessary and made sure we stayed on track and met our targets, which we are grateful for.

We would also like to thank the team of Module 11 led by Rom Langerak, who spent their time on crucial organization tasks that go barely noticed.

Contents

1 Introduction **3**

2 Domain Analysis **4**
2.0.1 Introduction to the Domain . . . 4
2.0.2 Client, users and interested parties . . . 4
2.0.3 Software Environment . . . 5
2.0.4 Existing Solutions . . . 6
2.0.5 Conclusions . . . 8

3 System Specification and Project Proposal **9**
3.1 Requirements Capturing . . . 9
3.2 Implementation Trajectory . . . 9
3.3 Deliverables . . . 11
3.3.1 Project Proposal . . . 11
3.3.2 Minimum Viable Product . . . 11
3.3.3 Final Product . . . 11
3.3.4 Project Report . . . 11
3.3.5 Poster . . . 12
3.3.6 Presentation . . . 12
3.4 Planning . . . 12
3.4.1 Activity 1: Design & Research (Weeks 1–3) . . . 13
3.4.2 Activity 2: Implementation (Weeks 3–8) . . . 13
3.4.3 Activity 3: Documentation (Weeks 3–10) . . . 14
3.4.4 Activity 4: Completing Poster and Report (Weeks 8–10) . . . 14
3.5 Roles and Responsibilities . . . 15
3.6 Procedures . . . 16
3.6.1 Daily Stand-ups . . . 16
3.6.2 Weekly Meetings with Client . . . 16
3.6.3 Penalties and Rewards . . . 16
3.6.4 GitLab Issues . . . 17
3.6.5 Reviews . . . 17
3.7 Risk Analysis . . . 17
3.7.1 Areas of Risk . . . 17
3.7.2 Risk Assessment . . . 18

4 System Design **25**
4.1 High Level System Design . . . 25
4.1.1 Introduction . . . 25
4.1.2 Initial Class Design . . . 25
4.1.3 Design Choices and Trajectory . . . 26

4.1.4 Final Class Design . . . 31
4.2 Neural Network Design . . . 33
4.2.1 Description . . . 33
4.2.2 Initial Design . . . 33
4.2.3 Design Choices . . . 35
4.2.4 Design Trajectory . . . 37
4.2.5 Current Design . . . 38
4.3 Monte Carlo Tree Search Design . . . 45
4.3.1 Description . . . 45
4.3.2 Initial Design . . . 45
4.3.3 Design Choices . . . 46
4.3.4 Design Trajectory . . . 47
4.3.5 Current Design . . . 50

5 Manual **51**
5.1 Adding a New Game . . . 51
5.1.1 Implementing the New State Interface . . . 51
5.1.2 Constants . . . 52
5.1.3 ConfigUtils . . . 53
5.1.4 Trainer Class . . . 53
5.1.5 Trainer Arguments . . . 54
5.2 Training a Game . . . 55
5.3 Resuming Training . . . 55
5.4 Global settings . . . 55
5.5 Using JAGGER . . . 56
5.6 Troubleshooting . . . 56

6 Testing **57**
6.1 Test Plan . . . 57
6.1.1 Unit Tests . . . 57
6.1.2 Integration and System Tests . . . 58
6.2 Test Execution . . . 58
6.2.1 Unit test execution . . . 58
6.2.2 Integration And System Test Execution . . . 60

7 Performance **61**
7.1 Parallelization . . . 61
7.2 Caching . . . 61
7.3 Parallel Inference . . . 62

8 Evaluation **66**

9 Conclusions **68**

10 Future Work **69**
10.1 Tuning Hyperparameters . . . 69
10.2 Reducing Overfitting . . . 69
10.3 Rollout vs. Neural Network Prediction . . . 70
10.4 Caching . . . 70
10.5 Symmetries . . . 71
10.6 Single Monte Carlo Tree Search Parallelization . . . 71
10.7 Monte Carlo Tree Search Data Structure . . . 71

11 Reflection **72**

11.1 Planning . . . 72
11.2 Contributions . . . 73

A Meetings with the Client **75**
A.1 Week 1 (Physical) . . . 75
A.2 Week 2 (Teams) . . . 76
A.3 Week 3 (Teams) . . . 77
A.4 Week 4 (Physical) . . . 78
A.5 Week 5 (Online) . . . 78
A.6 Week 6 (Online) . . . 78
A.7 Week 7 (Online) . . . 79
A.8 Week 8 . . . 79
A.9 Week 9 . . . 79
A.10 Week 10 . . . 79

B Sprint reports **81**
B.1 Sprint 1 (February 20 - March 12) . . . 81
B.2 Sprint 2 (March 13 - March 26) . . . 81
B.3 Sprint 3 (March 27 - April 9) . . . 81
B.4 Sprint 4 (April 10 - April 21) . . . 81

C Class Diagrams **83**

Chapter 1

Introduction

The Software Systems course takes place in Module 2 of the Technical Computer Science Bachelor program at the University of Twente. In this course, students will learn object-oriented programming (OOP) in Java. Each year, during the final two weeks of the course, students will develop a board game in Java that can be played in a network. They also have to create an AI player for this game. Tom van Dijk, the module coordinator for Module 2, is the project's major stakeholder. Instead of having to design a new AI for each game every year, Tom wants to create a general game-playing AI that can be used across various games and uses deep learning. The result will save time and effort that would otherwise be spent on developing a new AI. Our team was charged with building and implementing this general game-playing AI, which can be trained and deployed in any two-player turned-based game. This will require designing a customizable neural network in Java. The design report will go over the steps our team took to construct this general game-playing AI, including the research stage of the project, the design decisions we made, and the system's implementation. We will also assess the performance of our AI system by running it on several different games and comparing its results to those of other AIs. Ultimately, we want to build a strong and successful general game-playing AI that can be employed in a large variety of games and should make use of a neural network that can be trained.

Chapter 2

Domain Analysis

2.0.1 Introduction to the Domain

The domain our system is designed for concerns the development of AIs specialized in learning how to play games. Often, developers in this space will create, train, and optimize their AIs for a single game in particular. If one would want to use the same AI for a different game, they would have to remake the AI from scratch, making this an expensive and time-consuming process. Our system aims to provide a solution to this problem by providing a general game playing AI that can be trained to play any two-player deterministic game. It does not require any prior knowledge of the game, besides the rules. The AI will learn by playing against itself and gains knowledge by encountering new situations and making mistakes.

This domain is an interesting area of research in the field of artificial intelligence, as it requires the development of algorithms and techniques that can learn from experience and adapt to new situations. Therefore, it is not limited to the development of AIs for games, but it can also be applied to different domains.

2.0.2 Client, users and interested parties

The system is developed for our client, Tom van Dijk, who is the coordinator of Module 2. Tom is a member of the Formal Methods and Tools (FMT) group at the University of Twente. The purpose of the project is mainly for his personal interest.

He has tried to develop a system similar to ours but has not been able to finish it due to time constraints. Thus, he is knowledgeable about the domain and has a good understanding of the requirements. Tom is also the one who has provided us with a part of his system used in Module 2. This included some simple Java classes and game implementations, which we could use to build our system.

2.0.3 Software Environment

During the first meeting with our client, we discussed the software environment that we would be using for the project. We had the option to use either a combination of Java and Python or just Java, with a preference for the latter, as all the other system components are written in Java.

From a historical perspective, there have been a wide variety of programming languages and environments used for research and development in the field of machine learning. Python, however, has seen an immense increase in popularity in recent years, with its overall usage growing from 10% to 30% between 2018 and 2020, according to a study from GitHub (Roper & Richter, 2020). The scientific computing community has played a large role in this, as Python has become the most popular language for data science and machine learning with nearly 70% market share (Carraz et al., 2019).

A reason for this is that Python is an easy-to-learn, high-level programming language, yet it is also very powerful and flexible. On top of that, it has a large community of developers, which has led to a large number of libraries and frameworks being developed for it. Machine learning often makes use of linear algebraic operations on multidimensional arrays, which are very well-supported in Python, by libraries such as NumPy and SciPy (Harris et al., 2020; Virtanen et al., 2020). These libraries make use of the C, C++, and Fortran programming languages, which are very fast and efficient, but are not as easy to use as Python. The abstraction provided by Python allows for the use of these libraries without having to worry about the underlying implementation. This makes it a very attractive language for machine learning and data science, as it allows for quick and powerful prototyping and experimentation (Raschka et al., 2020).

Most recent deep-learning and machine-learning libraries are Python-based, such as TensorFlow, PyTorch, and Keras.

TensorFlow (Martín Abadi et al., 2015) is an open-source library developed by Google, and is very popular for deep learning, as it is very flexible and has a large community of developers. The core of the library is written in C++ and makes use of CUDA technology to make use of the GPU for computations. The main principle of TensorFlow is the use of data flows. The program is built up of computational blocks which are associated with each other through a directed graph, called a computational graph. Data is passed from one block to another and is processed there. This architecture makes it very easy to parallelize computations and is well-suited for building neural networks, as each neuron can be represented by a computational block (Gevorkyan et al., 2019).

PyTorch (Paszke et al., 2019) is another open-source library, initially developed by Facebook, and is also popular for deep learning. It is very similar to TensorFlow, as it too is built with C++ and CUDA, and uses data flows and computational graphs. The biggest difference is that PyTorch's computational graphs are dynamic, while TensorFlow's are static. This allows for more flexibility, as nodes can be added and removed from the

graph at runtime, whereas in TensorFlow, the graph has to be defined before the program is run (Gevorkyan et al., 2019).

Keras (Chollet et al., 2015) is a library that can be used on top of TensorFlow and other technologies. The main purpose of Keras is to make it easier to build neural networks, as it provides a high-level API that is easy to use. The library allows you to describe your neural network in terms of layers, e.g. neural layers, optimizers, convolutional layers, activation layers, etc. These layers are then combined to form a model, which can be trained and evaluated (Gevorkyan et al., 2019).

Java, however, also has some libraries and frameworks available that can be used for machine learning, such as weka, Deep Java Library (DJL), and Deeplearning4j.

Weka ("Machine Learning at Waikato University", n.d.) is an extended Java machine-learning library, containing a collection of algorithms. It enables users to carry out tasks like data pre-processing, classification, regression, clustering, etc. Multiple advanced tools are available, such as support vector machines, bayesian networks, decision trees, etc. The library also contains a graphical user interface, which can be used to easily visualize the data and the results of the algorithms. Users can then quickly analyze datasets and results, without the need for extensive knowledge about machine learning.

DJL ("DJL - Deep Java Library", n.d.) is a Java library for deep learning, developed by Amazon. Their goal was to provide an open-source tool, to create and train deep-learning models, targeted at Java developers. The library is built with Java concepts in mind, on top of existing frameworks, abstracting away from the underlying complexity (Vasudevan, 2019). MXNet (Apache, n.d.) is the primary framework used, which has been created by Apache and is designed with efficiency and flexibility in mind. It contains a dynamic dependency scheduler, allowing for automatic parallelization of operations, as well as a graph optimizer, making execution fast and memory efficient.

Lastly, you have Deeplearning4j ("Deeplearning4j Suite Overview", n.d.), a library developed by Konduit. It is a tool that provides the ability to run deep learning on the Java Virtual Machine (JVM). It lets you train models, while it interacts with the Python ecosystem under the hood, allowing for Python execution. DL4J also includes useful submodules, such as ND4J (Team, 2016), which is very similar to NumPy but also contains operations from both TensorFlow and PyTorch.

We decided to use Deeplearning4j for the project. Picking Java will lead to a smoother integration with the other system components, and it was also the preferred choice of our client. DL4J was in this case the best option, as it is the most popular Java library for deep learning, allowing for better support and documentation, while also utilizing the Python ecosystem.

2.0.4 Existing Solutions

Many people have already researched the field of general game play. A popular study, which has become one of the standards in the field, is the game description lan-

guage (Thielscher, 2011). GDL tries to describe games by using a set of first-order logical clauses, which are combined to form a description of the game. Though, this approach has some drawbacks, as it causes high-level algorithmic challenges, especially when it comes to the Monte Carlo tree search. On top of that, describing games in this way is not intuitive, and is extremely time-consuming. People who have little knowledge of logic and first-order logic will have a hard time understanding a description of a game and will have to spend a lot of time learning how to describe a game in GDL. Additionally, once a game has been described, it is hard to change aspects of the game, as the rules are often intertwined, and changing one rule can cause other rules to break. Lastly, processing GDL descriptions is computationally expensive, because logic resolution needs to be applied.

Another approach is Ludii, an ERC-funded ("European Research Council", 2023) general game system, which tries to be an efficient tool to be used by AI researchers, as well as people in related fields like game designers, historians, etc. It aims to model the most traditional games and combines them all inside a single system. This database can then be used to try and find relationships between these and their components. The games modeled in Ludii describe not only board games, but also card games, tile games, and dice games. It does this by trying to define games by the use of so-called ludemes, which are high-level concepts containing game-related information, allowing for exact and understandable descriptions. The Ludeme Project describes these ludemes to be advantageous, as it allows them to 'distinguish between a game's *form* (its rules and equipment) and its *function* (its emergent behavior through play)'. This separation allows for the possibility of an evolution analysis, as the ludemes make up the "DNA" of the game (Piette et al., 2020b).

But the most famous solutions have been created by DeepMind, who created AlphaZero and MuZero (Schrittwieser et al., 2020; Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, et al., 2017). These two systems have a generic algorithm applied and can reach superhuman performance without any additional knowledge about the game besides the rules. Compared to AlphaGo Zero, AlphaZero does not depend on handcrafted knowledge and domain-specific augmentations. Together with Monte Carlo tree search it plays against itself and continuously increases its performance. MuZero takes the same approach, but instead of needing a way to simulate the game MuZero uses a learned model of the game. This model means that MuZero can play a greater variety of games and even achieve better performance in some (like MuZero did for Go) (Schrittwieser et al., 2020).

Based on the available tools found in Section 2.0.3, and on the Deepmind papers mentioned above, people have created open-source implementations (Di Pasquale, 2021; Duvaud, 2019; Evolutionsoftswiss, 2018; Nair, 2017; Song, 2017). The most popular ones use Python (Duvaud, 2019; Nair, 2017; Song, 2017), but we also found some using Java (Di Pasquale, 2021; Evolutionsoftswiss, 2018). We can use these implementations and DeepMind papers to get a general game playing AI working quickly. The Java projects can show us how to implement the specifics (like the neural networks) in Java,

while the Python projects provide high-quality implementations that can serve as a reference for understanding how the general system should be structured.

2.0.5 Conclusions

By analyzing the current domain, we were able to identify the current technological tools available, as well as the existing solutions. Based on this information, we were able to make decisions on which tools to use for the project and which existing solutions to use as a reference. We decided to use Java for the project using Deeplearning4j as the deep learning library. Additionally, using Deepmind's papers as a reference, together with the open-source implementations, we were confident that we could develop a system with good principles behind it.

Furthermore, the role and interest of our client were identified, and we were able to keep this in mind when making decisions during development.

Chapter 3

System Specification and Project Proposal

3.1 Requirements Capturing

Capturing requirements is one of the most critical steps in the software development process. This is the foundation for any project and hugely impacts its success. It provides clarity because the objective and the expected outcome are established. It will be clear to both the developers and the client what the end goal is and allow for a smooth process. Furthermore, it reduces scheduling and requirement risks. This improves the overall quality of the product, as time can be spent more efficiently and effectively. On top of that, it allows for better project planning. Thanks to this process, the development team has the necessary information to estimate time and resources, resulting in a realistic project plan.

We capture the requirements for our project by talking to our client, which in our case is also our supervisor, Tom van Dijk. Before meeting him for the first time we did some research on the topic of the project. This way we could already start asking questions and discussing the direction in which the project is heading. We took notes of this meeting and based on what we heard and what we found we came up with requirements. These requirements will be proposed to Tom in the subsequent meeting. There we can agree on and determine whether the requirements would satisfy the client's objective. Thanks to regular communication with Tom, our client, we can modify or reassess existing requirements during development, although we will attempt to avoid this by making the initial requirements as specific as possible.

3.2 Implementation Trajectory

Starting from week 3, we will work with two-week sprints.

	Requirements	MoSCoW
A	The AI is trained using deep learning	Must
B	The AI can make a legal move in less than a given amount of seconds	Must
C	The amount of time the AI can use to make a move must be configurable	Must
D	The AI can pick a legal move if given a Java implementation of the Game interface	Must
E	The AI can play any deterministic two-player turn-based game that follows the Game interface	Must
F	The AI can be trained on a GPU using CUDA	Must
G	The AI's performance increase is noticeable over the course of at most three days on an NVIDIA GTX3060Ti	Should
H	The AI can play any game that follows the Game interface	Could
I	The AI's information and state will be visualized with a GUI	Won't

Table 3.1: Requirements and their MoSCoW categories

Sprint	Weeks
Sprint 1	Weeks 3-4
Sprint 2	Weeks 5-6
Sprint 3	Weeks 7-8
Sprint 4	Weeks 9-10

Table 3.2: Overview of sprints

In the first sprint, we will finish a minimum viable product (see Section 3.3.2).

During the next sprint, we will focus on implementing the theory and principles behind DeepMind's AlphaZero (Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, et al., 2017). This sprint will be considered done if the AI can beat a random agent using those methods. Our definition of "beating" is given in Section 6.1.2.

Sprint 3 will focus on improving learning efficiency and speed. Because we do not have access to a thousand TPUs like DeepMind, efficiency is very important. By increasing the learning speed, we can achieve much better performance in the same time frame. In this sprint, our goal will be to increase the AI's performance. We may achieve this by implementing some of the improvements DeepMind made with MuZero (Schrittwieser et al., 2020). A MuZero-like implementation would also be less limiting for future games, as it can also handle games with more players, games that are not turn-based, and games with randomness and external factors. However, it is arguably not strictly necessary to create a MuZero-like implementation in this scenario, as Module 2 has only ever used turn-based two-player games, therefore improving the AlphaZero-like implementation is a priority. This sprint will be considered done if the AIs can beat Tom's current minimax implementations.

The last sprint will be used as an emergency sprint for finishing up the project. This includes fixing bugs and finalizing documentation. As can be seen in planning, if implementation goes as intended this sprint will not be needed.

3.3 Deliverables

3.3.1 Project Proposal

The project proposal contains the following items:

- Requirements capturing
- Functional and quality requirements
- Implementation trajectory
- Planning
- Test plan
- Risk analysis

This document is one of the required deliverables for the project, but may also be used as an agreement for the product between the client and the team.

3.3.2 Minimum Viable Product

The MVP for this project consists of a trainable general game playing AI that can play the following three games:

- Collecto
- Pentago
- Othello

To be considered a minimum viable product, AI must only play legal moves, and it must not take more than a configurable amount of seconds per move. The AI will work based on the given implementation of the Game interface. That means it should also work for other games than the four games listed as long as there is an implementation for it. These conditions fulfill all our "Must" requirements from the MoSCoW classification (see Table 3.1).

3.3.3 Final Product

What is understood as the final product, is the AI requested by the client with the specified requirements (Table 3.1). The design and iterative implementation of this product over the sprints is the main purpose of the Design Project.

3.3.4 Project Report

The project report is the detailed documentation of the process of design, research and development in the project, as well as a description of the product itself. According to the provided manual, it must include:

- A requirement specification
- A global design
- A detailed design with a justification of the design choices
- A test plan and test results, and (pointers to) source code and a manual.
  It should be made clear what are the individual contributions of each student.

3.3.5 Poster

A poster showcasing the project. It should be an aesthetic presentation of the product and its design. The poster will primarily consist of the following:

- Project title and logo graphic
- Graphic overview of how the product works
- Description or visualization of the purpose of the product
- Visualization or short statistical summary of the results achieved by the AI

3.3.6 Presentation

At the end of the project, a presentation must be given, which will portray the process behind the project, start-to-finish, as well as a description of the final product and its usage.

3.4 Planning

The product is in need of a well-structured plan to ensure its successful delivery. The project will be divided into four activities: Design & Research, Implementation, Documentation, and Completing Report and Poster.

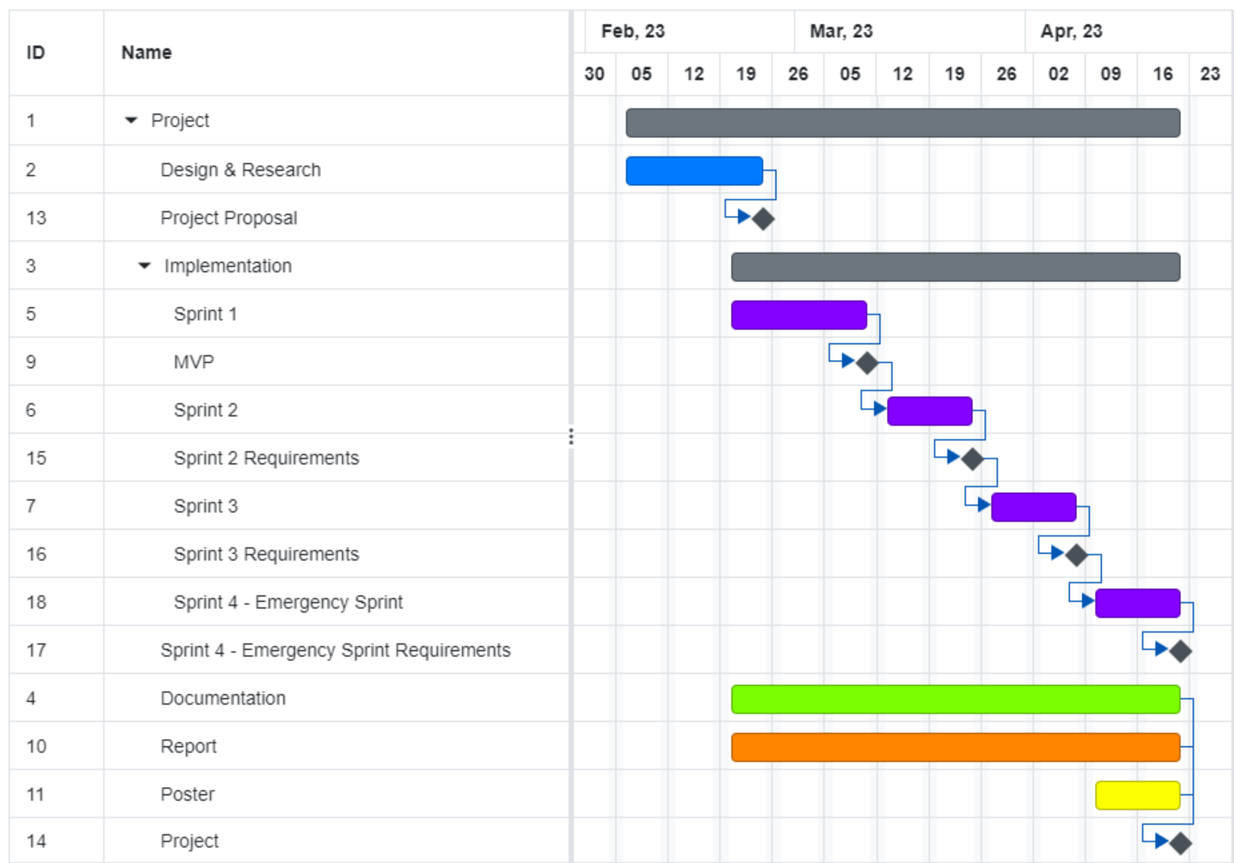

Figure 3.1: Planning

3.4.1 Activity 1: Design & Research (Weeks 1–3)

The Design & Research would mean obtaining the project requirements from the client, specifying the intended functionality of the product, and researching relevant literature and technology. This phase will help in ensuring that the project is created according to the predefined requirements and needs of the client.

Deliverables:

- Project Proposal, including requirements (see Section 3.3.1)

3.4.2 Activity 2: Implementation (Weeks 3–8)

The product is created in Implementation based on the requirements and research done in the first part. This process will keep the entire team engaged in the project, and frequent progress updates will be presented to the client. Implementation will be divided into more sprints with specific deliverables.

Deliverables:

- Minimum Viable Product (see Section 3.3.2)
- Final product delivery (see Section 3.3.3)

Communication is essential to the project's success. To keep the client informed of project progress and modifications, a communication plan has been determined. The following methods of communication will be used:

- **Daily Scrum Meetings:** To discuss progress and any issues that need to be resolved, the project team will hold daily scrum meetings. These meetings will help the project team be up to date with each other's work.
- **Coordination with client:** The client will be updated on the status of the project during weekly meetings held on Friday, and feedback will be asked from the client to make sure the project is meeting their requirements and expectations.

3.4.3 Activity 3: Documentation (Weeks 3–10)

In combination with the weekly client meetings, this will bring more important feedback and insight that will help the project development based on the client's needs. While working on the actual implementation of the product the report would be improved with in-depth information about different aspects such as technical details.

3.4.4 Activity 4: Completing Poster and Report (Weeks 8–10)

During this phase, the team should be in the finishing stage of the project where everything about the product is documented and a report can be finalized. The explanation of the product with the research done and the conclusions that have been drawn are important for the scientific community. Therefore, a special phase has been created to allocate enough time in detailing all the findings. Additionally, according to the project specification, a scientific poster will be created to showcase the process and result of the project.

Deliverables:

- Scientific Poster presenting results (see Section 3.3.5)
- Final Project Report (see Section 3.3.4)

From the planning table of our team, we define each sprint with specific milestones. These milestones represent how our requirements will be implemented throughout the implementation part of the project. Sprint 1 has as its milestone the Minimum Viable Product (MVP) which consists of the following defined requirements:

- The AI is trained using deep learning
- The AI can make a legal move in less than a given amount of seconds
- The amount of time the AI can use to make a move must be configurable
- The AI can pick a legal move if given a Java implementation of the Game interface

- The AI can play any deterministic two-player turn-based game that follows the Game interface
- The AI can be trained on a GPU using CUDA

Sprint 2 will implement this requirement:

- The AI's performance increase is noticeable over the course of at most three days on an NVIDIA GTX3060Ti

Sprint 3 will address the next requirement:

- The AI can play any game that follows the Game interface

Sprint 4 will be used as an Emergency Sprint in which the team will spend time to improve or work on the requirements of the project in case the previous 3 sprints were not sufficient.

The project team is certain that by following the planning plan, the project will be completed on time and with a high value. Sprints for the team have been set from week 2, each sprint lasts for two weeks and each sprint has its own goals. The most important deliverable will be the Minimum Viable Product, which should be done by the end of week 4.

3.5 Roles and Responsibilities

There are many necessary steps and procedures in successfully completing a software development project. To ensure that all the steps are completed successfully, and no important aspects of the project are omitted, it is important to have defined rules and responsibilities. While all team members will first and foremost fulfill the role of developers, each team member also has more specific responsibilities, which help organize project work on a higher level:

- Team leader - Dominik Myśliwiec:
  - Ensuring goals are set, tasks are delegated, and deadlines are set and met.
  - Keeping track of project activities and deliverables. (see Section 3.3)
  - Communication between client/supervisor and team.
  - Resolving team conflict and encouraging collaboration between team members
  - Ensuring the project is heading in the right direction
- Testing supervisor - Thomas van den Berg:
  - Supervises testing efforts of all team members
  - Defines high-level testing strategy
  - Ensures all features are tested according to the predefined testing strategy
  - Assures quality of end product
- Documentation supervisor - Daniel Botnarenco:
  - Ensures documentation is done according to given specifications and requirements.

  - Encourages and supervises team efforts in documenting code and writing deliverables (see Section 3.3)
  - Assures quality of documentation
  - Ensures that team members are aware of good documentation practices and follow them
- Version control and quality supervisor - Caz Saaltink:
  - Responsible for appropriate usage of version control by team members
  - Ensures team members follow common conventions for coding, version control, etc.
  - Ensure that the product not only works but is easily understood and potentially improved in the future
- Research supervisor - Thom Harbers
  - Ensures extensive research is done for the purpose of design and implementation
  - Ensures that the product design follows existing resources
  - Oversees used sources and their validity in the documentation.
  - Evaluates the quality of research done by all team members.

3.6 Procedures

3.6.1 Daily Stand-ups

During the project, our team has daily stand-ups at 11:00. We keep each other up-to-date about what we are working on and we can ask each other for help if needed. We also use the daily stand-ups to shift workloads between each other if needed.

3.6.2 Weekly Meetings with Client

We also have weekly meetings with our client. These will usually be on Fridays. For these meetings, we prepare an agenda which contains the topics we want to discuss. We work on the agenda throughout the week; if we come up with questions, we immediately add them to the agenda for the upcoming meeting so we do not forget them. During the meeting, we take detailed notes so that we can discuss the results of the meeting and how to move forward.

3.6.3 Penalties and Rewards

If someone in the team repeatedly fails to honour agreements, we can decide to hand out strikes. After three strikes, they have to get a cake for the rest of the team. If they still do not change their behavior, they will receive a red card after three more strikes.

At the end of the project, we can also give a green card to one of the team members. The green card will be awarded to someone who put an exceptional amount of effort into the project.

3.6.4 GitLab Issues

For task division and tracking we use GitLab issues. Issues will be assigned to at least one (but mostly just one) team member. Large tasks will be broken up into smaller tasks to make sure progress is noticeable from one daily stand-up to the next. A large task is defined as a task that is expected to take more than 24 hours (three working days).

We track progress with the GitLab issue board. The board has four lanes:

1. To do
2. In progress
3. Reviewable
4. Done

The lanes are self-explanatory. Our review process will be explained in the following section.

3.6.5 Reviews

When an issue is reviewable (Section 3.6.4 item 3), the work done on that issue will be reviewed by other team members. To facilitate this process, we use GitLab's Merge Request feature. The purpose of reviewing is not only to minimize errors but also to ensure that reviewers are up-to-date with the work of other team members. In order to ensure that everyone agrees with the contents of the report, the entire team must approve merge requests before they can be merged. For the code repository, two other team members must approve merge requests. We require fewer approvals here because we do not need to be as strict and want to allocate more time to implementation.

3.7 Risk Analysis

3.7.1 Areas of Risk

For the purpose of risk analysis, typical areas of system development as potential sources of risks (Higuera and Haimes, 1996) will be considered from highest to lowest likelihood of occurrence and impact, where the impact of the risks is a time cost, rather than a financial cost, contrary to typical business risk management.:

1. Schedule: the most limiting factor, but also the most valuable resource in this project is time. Since the project is restricted to the time of this module, scheduling risks have a high likelihood of occurrence, as well as a high impact.
2. People: in any group project, group dynamics and communication are key factors in successfully achieving the main goal. Risks in this area have a lasting effect on the project, potentially decreasing the team members' motivation and productivity.
3. Software: what is meant by software, in this case, is the main product - the AI for general gameplay. The risks to software will mostly be caused by poor development

and human error, therefore can not possibly be completely avoided, but should be mitigated as much as possible through testing and documentation.

4. Technology: the technologies used in the project will include Java libraries and packages used - especially DL4J (Black et al., n.d.) which will be at the core of the project. These libraries and packages cannot be considered failproof, and pose a constant risk to the project. This means the effects and potential dangers of the libraries have to be monitored constantly and carefully.
5. Hardware: the hardware in this project will be used to train the AI. This can be a time-intensive task, which means potential risks could cause scheduling issues and a less effective final product. That being said, the goal of the project is not to train the AI but rather to develop the AI, which, if successful, for the purpose of this project can be trained with less computing power, so high-impact risks in this area are less likely to occur.
6. Cost: risks in this area can be mostly disregarded for the purposes of the project. The aim is not to earn money, and so far there are no expected expenses. Risks to other areas also do not demand financial action.

It is important to characterize these areas beforehand, which makes it easier to then identify and assess potential risks in the next steps of risk analysis.

3.7.2 Risk Assessment

A common structured approach to identifying and assessing risks is a Risk Matrix (Garvey and Lansdowne, 1998). Each risk will first be identified and classified, then its probability (see Table 3.3) and impact (see Table 3.4) will be determined. Based on these two factors, the risk will be rated according to the risk rating table (see Table 3.5). This step is crucial, and achieves the following:

- Facilitates discussion on risks to the project
- Encourages risk mitigation
- Helps in prioritizing mitigating high risks over low risks
- Promotes awareness of potential risks
- Helps in identifying actions helpful in mitigation
- Shows when mitigation might be ineffective and an action plan post-risk might be needed

Probability Range	Interpretation
0-10%	Very Unlikely to Occur
11-40%	Unlikely to Occur
41-60%	May Occur About Half of the Time
61-90%	Likely to Occur
91-100%	Very Likely to Occur

Table 3.3: Probability of Occurrence Illustrative Interpretations

Impact Category	Definition
Critical (C)	An event that, if it occurred, would cause project failure
Serious (S)	An event that, if it occurred, would cause major schedule costs. Secondary requirements may not be achieved
Moderate (Mo)	An event that, if it occurred, would cause moderate schedule costs, but important requirements would still be met
Minor (Mi)	An event that, if it occurred, would cause only small schedule costs, and important requirements would still be met
Negligible (N)	An event that, if it occurred, would have no effect on the project

Table 3.4: Risk Matrix Impact Assesments

	Negligible	**Minor**	**Moderate**	**Serious**	**Critical**
0-10%	Low	Low	Low	Medium	Medium
11-40%	Low	Low	Medium	Medium	High
41-60%	Low	Medium	Medium	Medium	High
61-90%	Medium	Medium	Medium	High	High
91-100%	Medium	High	High	High	High

Table 3.5: Risk Rating Scale

With the existing impact and probability assessments, as well as the rating scale for this project introduced, the next step is to identify, and then analyze each risk using a Risk Matrix with the following fields:

- Risk: a short description of the risk itself
- Source: the source of the risk (area of the project, or more detailed technology, hardware, etc.)
- P: the probability of this risk according to Table 3.3
- I: the impact of this risk according to Table 3.4
- R: the rating of this risk according to Table 3.5
- Mitigation plan: actions that can help in mitigating this risk

Many of the listed risks are risks common software risks (see Hoodat and Rashidi, 2009), although in different sources they are classified using different categories.

The approach taken in risk management for this project relies on risk mitigation, as opposed to risk avoidance (Oren, 2001). That is because while we try to avoid the risks as much as possible, in software engineering, there are many risks that have to be accepted as almost unavoidable. Those risks can, and sometimes *will* happen, but there are possible steps that can mitigate their impact on the project. Therefore, what will

be understood as risk mitigation is taking steps to minimize the probability of the risks occurring, and, in case they fail, decreasing the impact they might have on the project, allowing it to be successfully completed despite them.

The initial risk analysis provided in Table 3.6 is a fundamental element of risk management, however, it is only the first step - this process will last the entire project. Appearing risks must be identified and mitigated according to this specification or must be identified and reassessed if they have not been considered initially.

Table 3.6: Risks of the project and mitigation plans

Risk	Source	P	I	R	Mitigation plan
Ambiguity of requirements	Requirements	11-40%	Mi	Low	Making specific requirements, communicating with client
Impossible requirements	Requirements	0-10%	C	Medium	Making specific requirements, researching thoroughly
Inadequate requirements	Requirements	41-60%	Mo	Medium	Researching thoroughly, brainstorming on requirements together
Change of requirements	Requirements	11-40%	S	Medium	Ensuring specific early communication with Client
Lack of agreement between customer and developers	Requirements	0-10%	S	Medium	Ensuring specific and regular communication with client
Lack of testing	Software	11-40%	C	Medium	Adhering to a good testing strategy, following testing policy
Human errors	Software	91-100%	Mo	High	Adhering to the testing strategy, following testing policy, using version control, and following good version control practices
Complexity of architecture	Software	41-60%	S	Medium	Following good coding practices, writing detailed documentation
The difficulty of implementation	Software	11-40%	S	Medium	Researching thoroughly, communicating between team members, communicating with client
Inadequate documentation	Software	41-60%	S	Medium	Regularly documenting during implementation, adhering to our team documentation policy

Lack of skill or domain knowledge	Software	11-40%	C	High	Communicating between team members, researching thoroughly and learning
Lack of design documentation	Software	11-40%	C	High	Regularly updating design report, checking other team members' progress and contribution
Lack of good estimation	Schedule	41-60%	S	Medium	Discussing with the team about the schedule, maintaining a good implementation trajectory, communicating about difficulties
Unrealistic schedule	Schedule	11-40%	S	Medium	Specifying partial goals, prototyping, having a detailed project planning
Slow development	Schedule	11-40%	S	Medium	Having a daily standup, communicating about personal circumstances and progress, prioritizing the implementation
Hardware failure	Hardware	0-11%	S	Medium	Training AI on multiple devices, prioritizing development and design before training, prioritizing the MVP (see Section 3.3.2)
Lack of good hardware	Hardware	41-60%	S	Medium	Training on multiple devices, optimizing software, having partial goals and prototyping
Lack of collaboration between developers	People	41-60%	Mo	Medium	Having daily standups, using agile, communicating about implementation and difficulties, asking for help
Unavailability of developers	People	0-11%	S	Medium	Having detailed planning, communicating between team members and with client
Unavailability of Client	People	11-40%	Mo	Medium	Having detailed planning, communicating early as well as regularly with client

Lack of motivation/commitment	People	11-40%	C	High	Communicating regularly, having daily standups, communicating about difficulties, maintaining a healthy atmosphere
Lack of roles and responsibilities definition	People	11-40%	S	Medium	Planning, communicating about responsibilities, committing to the project
Disagreement between team members	People	11-40%	S	Medium	Having a healthy attitude, communicating regularly, openly and honestly, managing each others' expectations
Difficulties with understanding libraries	Technology	11-40%	S	Medium	Researching thoroughly, collaborating between developers, writing high-quality documentation
Difficulties with library integration	Technology	11-40%	S	Medium	Defining a documentation policy, reading documentation, collaborating between developers, sharing knowledge between developers
Lack of technology documentation	Technology	11-40%	Mo	Medium	Collaborating between developers, sharing knowledge between developers, researching thoroughly
Technology change	Technology	0-11%	Mo	Low	Discussing needed technologies before implementation, maintaining detailed documentation
Not completing MVP on time	Planning	11-40%	Mo	Medium	Discussing with the team about the schedule, maintaining a good implementation trajectory, moving MVP deadline by a week

Not meeting sprint 2 requirements	Planning	11-40%	Mo	Medium	Discussing with the team about the schedule, maintaining a good implementation trajectory, moving the unsatisfied requirements deadline by a week
Not meeting sprint 3 requirements	Planning	11-40%	Mo	Medium	Discussing with the team about the schedule, maintaining a good implementation trajectory, using the emergency sprint to try to satisfy requirements, documenting failures

Chapter 4

System Design

In this chapter, the full technical description of the system is provided and explained. This project aims to create a general game playing AI in Java, using Deeplearning4j. The AI will be able to play any deterministic two-player turn-based game that follows a given Game interface. The system is designed to meet a set of requirements, including the most crucial requirement: training the AI using deep learning. The project's client and supervisor, Tom van Dijk, is the main stakeholder and has expectations for the system's functionality and performance. In this system design, we will describe the architecture and components of the system, including the Monte Carlo tree search and the neural network used by the AI.

4.1 High Level System Design

4.1.1 Introduction

In Section 4.1 we will give an overview of all parts of our system except for the parts that are described in Sections 4.2 and 4.3. Additionally, we will go over some impactful design decisions that were made.

4.1.2 Initial Class Design

In this subsection, we will shortly discuss the initial design our team has created for the system. Figure C.1 shows the structure and class diagram of the initial design.

AbstractNeuralNet The class is an abstract class implemented such that we can develop multiple neural networks for different games.

Arguments The class is used to easily pass hyperparameters to the training of the AI, in our case the "Coach" class.

neuralnet/Arguments The class is used to pass multiple arguments to the neural networks.

Coach This class takes care of training the neural network. That includes generating game data using MonteCarloTreeSearch, fitting the training network to the data, setting up tournaments between the training network and the current best network using Mafdet, and saving every iteration to a file.

JaggerAI The class contains the implementation of the AI interface given to us by our client. It can be used as the end product with a trained model, as well as during training in combination with Mafdet to evaluate the model's training results. The JaggerAI class uses the Monte Carlo tree search algorithm to make decisions of moves, making use of the customizable arguments passed by the Arguments class.

Mafdet The class is named after Mafdet (also Mefdet, Maftet), a goddess in the ancient Egyptian religion which was the deification of legal justice. The name "Mafdet" was chosen for this class as it represents the concept of competition. This class initializes a match or multiple matches between two AIs and returns their results after the games are completed. It is used in the tournament part of a training cycle.

MonteCarloTreeSearch The class contains the Monte Carlo tree search implementation. The algorithm performs a search on the game tree of states and selects nodes according to the Upper Confidence Bound (UCB) formula. For each node, the current state of the game is evaluated by the neural network to determine the policy vector and the expected scores.

NeuralNet The class was created to implement the neural network.

PolicyVector The class was designed to store the policy vector part of a prediction from the NeuralNet. The probability of a move can be changed with the `update` method.

Prediction The class is used for combining a PolicyVector and Value into one object.

Result The class is used to have better management of the results of the games that are being played between two AIs.

Value The class is used to represent a state's value, which is an estimate of how good a state is for a player in the range between -1 and 1.

4.1.3 Design Choices and Trajectory

While developing the general game playing AI, we focused on three key design aspects:

Usability Our main target user for the project is our client, Tom. He indicated in our meetings (Appendix A) that the main purpose of the project is his own interest in game-playing AIs but lack of time for implementations. With that in mind, we supplied detailed guidelines to help Tom set up the AI.

Performance We designed the system so that it's possible to run it on a personal computer, although depending on the game and network complexity, the performance

may differ. We explored several system implementations to improve performance, and we observed that employing CUDA cores enhances training time. We make use of a neural network architecture that can improve the quality of decisions. Furthermore, we have developed a Monte Carlo tree search, which guides the learning process.

Interoperability We developed the system to be interoperable with other games because the AI would be utilized in many Java games. This is a logical result of the fact that the interface used by the Monte Carlo tree search and neural network can accommodate any two-player deterministic turn-based game.

In the following subsections, we will explain the impactful design choices we made and discuss their implementations.

4.1.3.1 Parallelization

The self-play and tournament stages take up the vast majority of a training cycle, making them ideal candidates for performance enhancements. Self-play and the tournament both run many games. Parallelization is very reasonable since each game operates independently. The two stages are implemented by `Trainer` and `Mafdet`, respectively, both utilizing Java work-stealing thread pools ("Java Development Kit Version 17 API Specification — Executors.newWorkStealingPool", n.d.) for efficient execution.

For each game in self-play and the tournament, a task is submitted to the thread pool. A set number of threads work on tasks from the thread pool until all tasks are completed. Because games may have different durations, we chose to use a work-stealing thread pool because threads that finish early can "steal" tasks from other threads, maximizing efficiency.

By default, work-stealing thread pools use the number of available processors as the thread count. However, because we use parallel inference, we found that the optimal thread count is the number of games, so that all games can be played in parallel (Section 7.3). `Trainer` and `Mafdet` automatically use the number of self-play games and the number of tournament games as their thread count, respectively. Section 7.1 studies the performance of different thread counts before we used parallel inference.

4.1.3.2 Caching

Caching is the process of storing copies of data in a cache, or temporary storage location so that they can be accessed more quickly. When our system started using a larger neural network architecture, it took more time to run the application. Low CPU usage, even with multiple threads, suggested that our training process contained a bottleneck. Profiling data made it clear that inferences from the neural net were to blame. One attempt at diminishing this bottleneck was building a cache for the neural network's output. We implemented a cache by using a hash map, mapping states to a network result if this state had already been encountered.

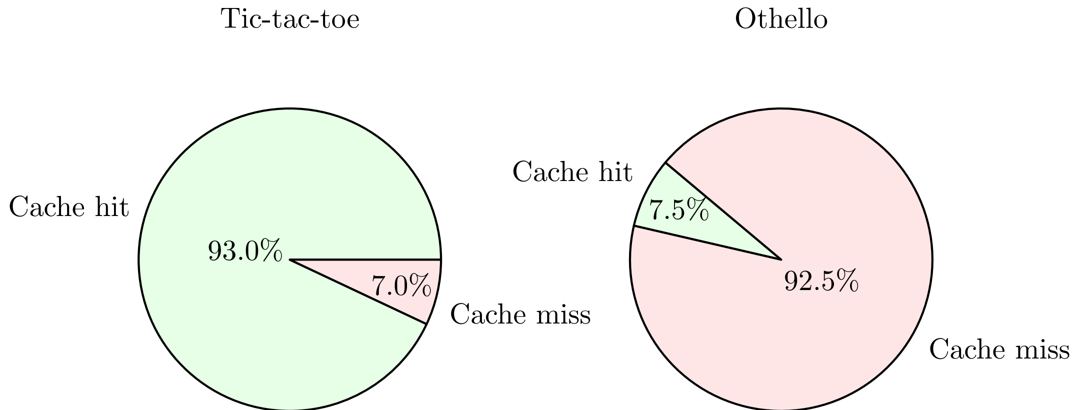

Figure 4.1: Cache hits and misses during one iteration of tic-tac-toe and Othello self-play. Relevant hyperparameters: `monteCarloIters=250, cPuct=1.5, numSelfPlayGames=100`. `dirichletValue` was set to `0.9` and `0.6` for tic-tac-toe and Othello, respectively.

In Section 7.2 we see that for tic-tac-toe, caching improves the overall performance of our system by almost 40%. Caching results in a noticeable speedup to the process in tic-tac-toe but unfortunately the same cannot be said for Othello. When training Othello with a cache, the cache quickly grew too big, causing the process to run out of memory, resulting in a crash. Additionally, because Othello has many more different states than a small game like tic-tac-toe, the cache hit ratio was very low (Figure 4.1). The low amount of cache hits made us decide that it was not worth it to spend time trying to fix the memory problem, ultimately leading us to drop caching completely.

4.1.3.3 Parallel Inference

At first when we parallelized `Trainer` and `Mafdet`, we only noticed a slowdown. As explained in Section 4.1.3.2, requesting predictions from the neural network formed a bottleneck. While caching helped for tic-tac-toe, it did not scale well for larger games such as Othello. Another change we had to make in order to make multithreading worthwhile was cloning the networks for each thread. Because Deeplearning4j networks are not thread-safe, multiple threads requesting inferences from the same network caused a bottleneck (Konduit, n.d.-c). Cloning the networks for each thread prevents this from happening.

We later found out that, according to Konduit (n.d.-c), another solution would be to use DL4J's `ParallelInference` (Konduit, n.d.-b). After running some experiments, we found it achieved great results. See Section 7.3 for more information.

4.1.3.4 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA that enables software developers to access the parallel processing power of NVIDIA graphics processing units (GPUs) for general-purpose computing. CUDA can be used to accelerate the computation of the computationally intensive tasks involved in training deep learning models, such as convolutions, matrix multiplications, and backpropagation. By offloading these tasks to the GPU, which is specifically designed for parallel processing, significant speedup can be achieved compared to running the same computations on a CPU.

Deeplearning4j supports the use of CUDA through one of their ND4J backends. ND4J has backends for both CPUs and GPUs. By utilizing CUDA, Deeplearning4j can perform matrix operations and other computations in parallel on NVIDIA GPUs, significantly accelerating training time for deep neural networks.

The first time we tried to use ND4J's CUDA backend, we actually observed a strong increase on time spent in self-play (Figure 4.2), which was unexpected. It turned out that the time it took for the neural network to return a prediction for a given state increased significantly when using CUDA. Presumably, the reason is that it takes a large amount of time to transfer data between the CPU and GPU, outweighing any benefit from lower inference times. For this reason, we refrained from using CUDA for a long time. Figure 4.2 shows that using CUDA does decrease fitting time. However, on the intermediate network, the absolute difference is negligible compared to the amount of time that is spent on self-play.

With the arrival of the final neural network, fitting time on a CPU increased significantly Figure 4.2. This sparked the idea of using CUDA for fitting only, while still using the CPU for self-play and the tournament. Unfortunately, Deeplearning4j does not support using different backends in the same process (Gibson, 2021). This meant we had to spawn a different process to fit the data with the GPU. With this, we keep the relatively low times for self-play and tournaments, while also benefiting from GPU performance during fitting. (Note: for some GPUs, inference may be faster than on a CPU, see Section 7.3 for more information.)

In addition to using CUDA for fitting, we also looked into options for using the GPU's parallel power to run Monte Carlo tree search. We looked into different options for using a GPU in Java. "JCuda" (n.d.) provides a way to interact with the CUDA runtime from Java, allowing the execution of CUDA kernels. This would require converting all game logic to kernel code which was deemed too time-consuming.

There are also options that can execute Java code on a GPU ("Aparapi", n.d.; Fumero et al., 2019; "TornadoVM", n.d.). However, it turned out that only a limited subset of Java code is supported and we could not use this without a significant rewrite of the game logic. For these reasons we decided to opt for CPU parallelization only.

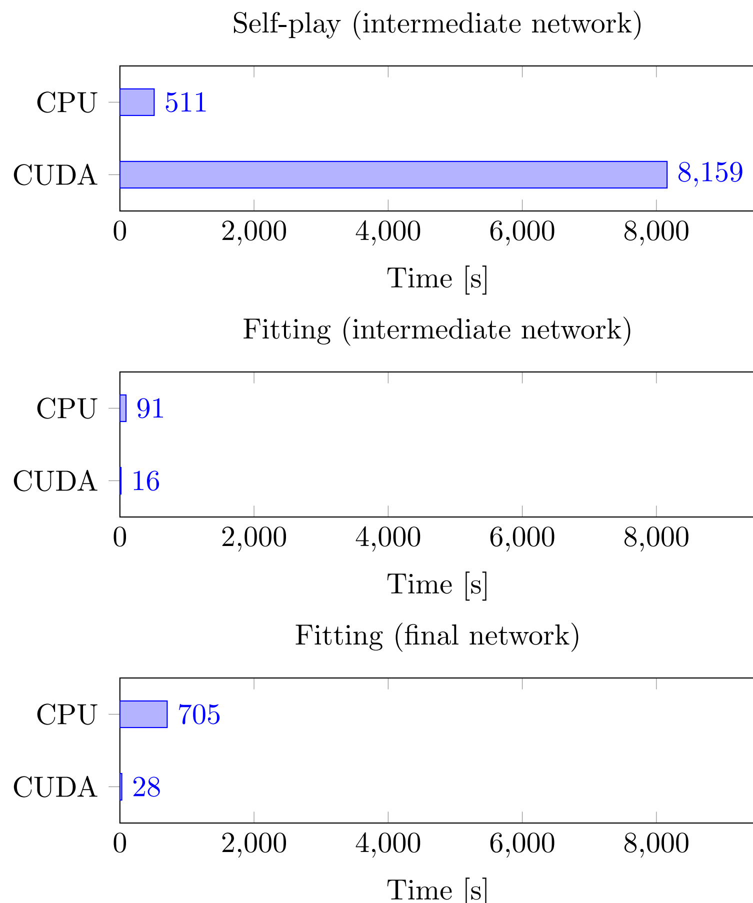

Figure 4.2: Time comparison of different backends for self-play and fitting (Othello). Relevant hyperparameters: `monteCarloIters=100, numSelfPlayGames=50, numEpochs=20`. CPU: Intel Core i7-10750H @ 2.60GHz. GPU: NVIDIA Quadro T1000 with Max-Q Design.

4.1.3.5 Dataset Partitioning

After switching to CUDA for fitting, we ran into crashes because Deeplearning4j ran out of memory. As our network design grew larger, it took up a larger part of the available memory. We had to fix this by splitting the dataset into smaller partitions. The number of partitions the dataset should be split into can be set with `Fitter.NUM_PARTITIONS`. The number of needed partitions depend on the user's GPU, the neural network size, the game, and the value of `numSelfPlayGames`. For optimal performance, `Fitter.NUM_PARTITIONS` should be set as low as possible (but at least 1).

Deeplearning4j also offers DataSetIterators for the purpose of fitting a dataset into smaller chunks. Unfortunately, all attempts at using those still caused the program to crash.

Presumably, DL4J still uses more memory with a DataSetIterator than with manually split datasets.

4.1.4 Final Class Design

Our last version of the system has changed drastically since the start of the project. Most notably, the AbstractNeuralNet was removed. It was included in the initial design because we planned to have two different networks, one for policy and one for value, just like AlphaGo (Silver et al., 2016). As described in Section 4.2 we later moved on to a single network with two heads. This meant we only needed one network class: JaggerNetwork.

We also split up the arguments classes. We now have three different classes: NetworkArguments, MCArguments, and TrainerArguments. MCArguments is a subset of TrainerArguments which makes it easy for Trainer to pass its TrainerArguments to the constructor of MonteCarloTreeSearch.

Lastly, we created two distinct AI implementations: IterationBoundJagger and TimeBoundJagger. Initially, we only had a time-based AI (now TimeBoundJagger), but when we introduced multithreading this became unreliable due to varying thread execution times. Thus, we decided to use an implementation that always executes the same number of iterations, ensuring reliability. This implementation is utilized in both self-play and tournaments, while TimeBoundJagger remains accessible for our client's use.

In the appendix, Figure C.2 shows a class diagram of our final implementation. For a complete overview of changes made, see the list below.

AIGenerator A functional interface for creating an AI. It is used in Mafdet to generate a new AI for each game. Before this interface was created, Mafdet would clone the AI instances it received but this became problematic when we implemented Collecto, as each game can have a different starting state.

CollectoTrainer, OthelloTrainer, PentagoTrainer, TicTacToeTrainer
Distinct Trainer classes were created for each game to easily train different games. The classes only contain a `main` method which instantiates Trainer with the appropriate parameters for that specific game.

Edge A class to store an edge between a State and a Move. Used by MonteCarloTreeSearch.

Fitter A class that fits a JaggerNetwork to a dataset. Its functionality is located inside the `main` method because this class is used in a separate process. See Section 4.1.3.4 for more information.

IterationBoundJagger An extension of Jagger that returns the best move after a given amount of `monteCarloIters`. This AI is used during the self-play and tournament stages of the program.

Jagger The class contains an abstract implementation of the AI interface given to us by our client. It takes care of the instantiation of MonteCarloTreeSearch and JaggerNetwork and it keeps track of the current State. It is extended by IterationBoundJagger and TimeBoundJagger.

JaggerNetwork Renamed from NeuralNet. For more information see Section 4.2

Mafdet This class was changed to support playing different games in parallel.

MCArguments This record class is used to pass multiple arguments to MonteCarloTreeSearch and, transitively, to Jagger.

MCArgumentsI An interface with the same methods as MCArguments. It is implemented by TrainerArguments (and MCArguments). It was created so that Trainer can pass its received TrainerArguments to MonteCarloTreeSearch and IterationBoundJagger.

MonteCarloTreeSearch This class was not changed from the initial design. For more information see Section 4.3.

NetworkArguments This record class is used to pass multiple arguments to JaggerNetwork.

PolicyVector This class is used to represent a policy vector returned by JaggerNetwork. We removed the `update` method and made the class immutable to make our code less error-prone.

Prediction This class was unchanged from the initial design.

Result This class was unchanged from the initial design.

StateGenerator A functional interface for State generation. It was introduced when support for Collecto was implemented because Collecto games have different starting states. It is used by Trainer and Mafdet to set up self-play and tournaments, respectively.

TimeBoundJagger An extension of Jagger which takes a `timeToMove` argument and returns the best move in the given time. This AI is not used in the training process but it can be used by our client when playing on the server.

Trainer Renamed from Coach. The class now supports parallel game generation.

TrainerArguments This record class is used to pass multiple arguments to Trainer.

Value This class was unchanged from the initial design.

4.2 Neural Network Design

4.2.1 Description

One of the most crucial elements of our system is the neural network. The way our game AI can outperform average game players relying on time-consuming searches is by training this network ahead of time to get the most accurate predictions possible. The goal of a network is to, for any input game state, return a prediction of the state's value and policy vector. What is understood by the "value" of a state is how good it is for the player to be in that state when having to make a move, ranging from -1 to 1, where -1 means the position is losing in all possible playouts, 1 is winning, and 0 is equal or drawing. The policy vector is a distribution between all the moves in a game's action space indicating which move is advisable. It can be thought of as the probability for each move (including illegal moves as long as they are in the action space) that it is the best move to make from the current state.

The neural network must improve through training to a point where its predictions allow for searches using the Monte Carlo tree search that give a better result on fewer iterations, essentially providing a tradeoff of spending time on training by significantly improving the search's heuristic.

It must also be stated that, although the purpose of the project is to design a general game playing AI, it does not mean that every game would use the same network. This would, first of all, require the state of any game to be represented by a vector of the same length (or in a convolutional layer stack of the same size and depth). Additionally, training the network must be specific to the game, due to different patterns, rules, game conditions, etc. which would make previously adjusted biases for another game irrelevant. However, especially due to our further specification of only deterministic, two-player board games, it is certainly feasible to create a network which follows a similar architecture, with the only differences being its size, number of layers as well as input and output layers.

4.2.2 Initial Design

In the first version of the system (the MVP), there were two separate neural networks, both of which were very simple.

The first neural network was the value network which consisted of an input layer of a variable size, followed by a fully connected layer, also of a variable size, followed by a single output layer which returned a prediction for the value. The input layer's size would be derived from the state, for example, 9 for tic-tac-toe or 64 for Othello. This layer would also use a rectified linear activation function. Initially, the output layer used a cross-entropy loss function and a sigmoid activation, outputting a number between 0 and 1.

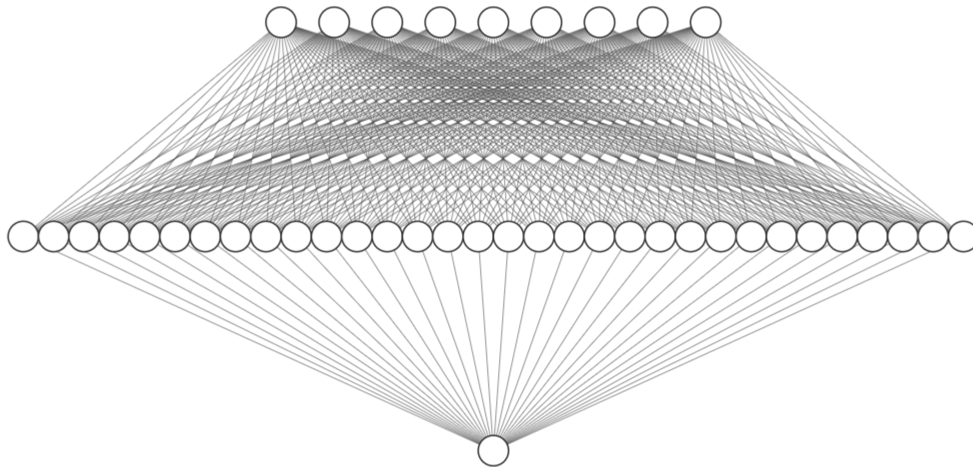

Figure 4.3: Example Value Network for tic-tac-toe with initial architecture and 32 hidden nodes

The policy network, in the beginning, was very similar in architecture. It also consisted of an input layer, the size of which was derived from the state, followed by a fully connected layer of variable size, followed by an output layer. In this case, the output layer consisted of several nodes equal to the action space of the game. For that purpose, we also added a function to the state interface we were given in the beginning, which returned the action space of the game. Additionally, due to the purpose and nature of the policy network, it used a multiclass cross-entropy loss function and a softmax activation function, which is often used in classifiers, due to the output being a probability for each node, with their sum always being 1.

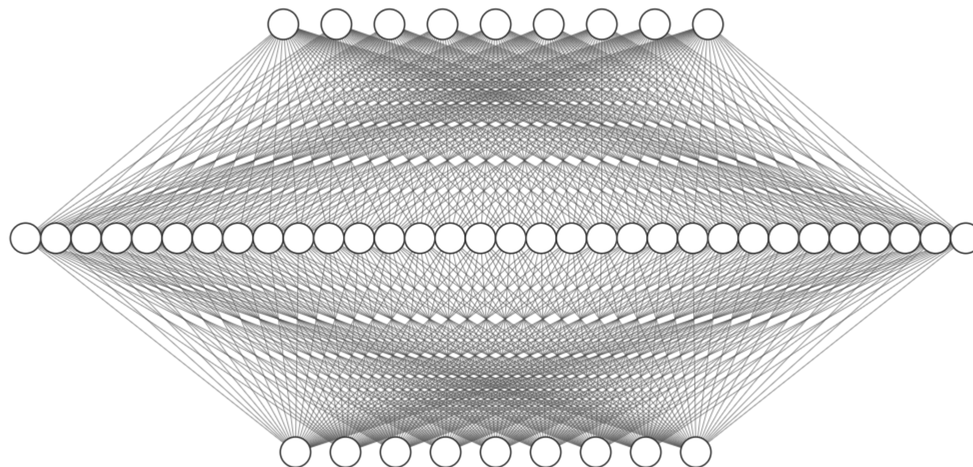

Figure 4.4: Example Policy Network for tic-tac-toe with initial architecture and 32 hidden nodes

Although it seemed like these networks worked for a minimum viable product, after further research and learning about neural networks, we noticed many potential errors, which came into sight when further testing the system. This included a lack of scalability, poor predictions, and overfitting for more complex games, which are the main domain of the project.

4.2.3 Design Choices

Many choices had to be made during development. Although we knew that we would like to follow the design of the neural network of DeepMind's AlphaZero, we had to take into account the limitations of Deeplearning4j as well as the difference in complexity between games implemented by students for the purpose of module 2, and chess, Shogi and Go, which were the main domain of AlphaZero. Another important consideration was that the hardware available to the users of our product was orders of magnitude less performant than that available to DeepMind.

4.2.3.1 Separate or Dual Networks

The first important choice to be made was between two separate networks or one dual network. For the MVP, the first version of our project consisted of two networks. This was due to a lack of familiarity with implementing more complex networks in DL4J, which was necessary for a network with shared layers but separate outputs. Our decision for later stages however was to use a combined dual network, which meant more experience with DL4J was necessary. There were two reasons for this choice:

Research In "Mastering the game of go without human knowledge" (Silver, Schrittwieser, et al., 2017), there is a comparison provided between the two options. Through their evaluation, they found that a model based on two separate convolutional networks had a higher accuracy for predicting professionals' moves. However, this does not have much value, since for many of the games JAGGER would be trained to play there are no professional players. Additionally, a well-trained network will often be able to make predictions better than a professional player, meaning that this statistic does not necessarily translate to better play. A dual residual network had the highest ELO rating, as well as the lowest mean squared error of the outcome of professional games. Both of these statistics are more relevant since they indicate that this type of network outperforms two separate networks in play, which is the most important factor for us as well. DeepMind also used this reasoning in their choice to use a dual residual network in their next iteration of the product - AlphaZero (Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, et al., 2017).

Performance Since JAGGER is unlikely to be trained on professional hardware, such as multiple GPUs, or even TPUs, the amount of memory needed for training the networks was a very important factor. In testing, we found that even on the best hardware available to us (an Nvidia RTX 3060Ti GPU, Intel Core i7-10700 CPU

and 16GB RAM), in training there were often issues with memory allocation, which caused the program to crash. This leads us to believe, that a single network with two heads will allow for better performance since it uses fewer resources. Additionally, the longest part of the training cycle was the self-play stage, the majority of which was computing the network's predictions. Due to this, generating a dataset from a single game played using Monte Carlo tree search scaled almost exponentially with the size of the network. Although we were unable to identify the exact reason for this, we believe that this is due to the steps DL4J takes to receive a prediction, which is computation heavy. Unfortunately, when using a CUDA-based backend for the neural networks, although we believed receiving an output from the network would be faster, it scaled even worse with it's size. Potentially, it was also being slowed down due to memory allocation, though we did not have enough time or resources to investigate this further.

4.2.3.2 Pooling

Another design choice that we made during implementation was the addition of pooling in the residual block, although we removed it for the final implementation. Pooling is used to take clusters of neurons and apply a function to them, which in turn generalizes the features and decreases the output size. After pooling, we apply batch normalization again, which will improve the learning of the last convolutional layer and help optimize the network. Although DeepMind did not use this element in their networks, there is evidence that pooling can increase the performance of the network in learning (Clausen et al., 2021). There was confusion, however, on the details of the pooling implemented. The two most popular types of pooling are average-pooling and max-pooling. Max-pooling is often believed to be more useful in residual networks used for image recognition since they highlight features in higher contrast. Average-pooling, on the other hand, can be described as "smoothing out" the features. We tried both kinds of pooling in our implementation and found average-pooling to give better results. However, interestingly, after testing what we believed to be the final implementation, we found that by removing pooling from the network, although the training was slower, the loss value decreased more. We believe this might be due to pooling simplifying and reducing the network, which was already quite small for training on somewhat average personal computers. If more time was available, it would certainly be beneficial to explore this further, perhaps by using an even bigger network with added max-pooling and global average-pooling.

4.2.3.3 General Design

Convolution The reason for using convolution in the network was the similarities between board game boards and images. Board game states often consist of an X by Y board with a certain type of feature on each field (a channel). This is also what an image is, and convolution is designed to find patterns in this type of input using filters (O'Shea and Nash, 2015). In an image, this might be edges, clusters of pixels, or perhaps shadow and light. In games, this could be patterns such as

clusters of free spaces, or a number of features meaning one of the players is close to a victory.

Residual blocks Residual blocks use a special skip-connection (in DL4J terminology, an add-vertex). A skip-connection is used in residual networks (often called resnets) to "re-add" the original input after a set of convolutional filters have been applied. This approach reduces accuracy degradation, as well as training error (He et al., 2016). It is a crucial step in network scalability, allowing each layer to learn on a similar level, no matter how deep into the network it is.

Batch normalization Batch normalization is a crucial element of the network. Since every time the network is fitted, it most likely receives a different batch of states and predictions, the output of the layers should be normalized. This allows for the usage of higher learning rates, which means faster learning (Santurkar et al., 2018). For that reason, the batch normalization step is used after every convolutional layer in the network.

ReLu activation The ReLu activation function is used due to its advantages over other common activations. The ReLu activation does not activate every neuron, but for every activated neuron, the gradient is always equal to 1. This activation function enabled huge breakthroughs in the world of deep learning (Krizhevsky et al., 2017).

4.2.4 Design Trajectory

The first steps of improving the initial network can mostly be considered bug fixes. Firstly, the weight initialization of the initial network was changed to a ReLu type initialization, as opposed to the initially used normal distribution with a mean of 0. This was a mistake due to a lack of experience with machine learning and meant that on a simple neural network, there was a risk of many neurons not having an impact on the output. The second fix was related to the output activation of the value network. Initially, the output activation was a sigmoid function, which meant that the value prediction lay between 0 and 1 and was then translated into a range of -1 and 1 using another function. This also meant that the prediction for training had to be given in the range of a sigmoid. This was also unintentional and was easily fixed by changing the activation function to a tanh activation.

The next, bigger step, was to combine the two networks. This was very difficult and problematic because the Deeplearning4j documentation is quite poor and often outdated. If we were more aware of how big of an issue this would be, we would most likely opt for a different implementation that would not rely on this library. The combination of the networks led us to an intermediate version of the network. This version mostly differed in size and configurability. Since the final version of the network is described in detail in Section 4.2.5, this section is mostly focused on explaining the differences between the two versions.

The intermediate version was a residual network consisting of two convolution blocks (Figure 4.6), with one residual block (Figure 4.7) in between, followed by the value and policy heads (Figure 4.8, Figure 4.9). The number of residual blocks was not yet configurable, which meant the network was not scalable depending on the game. The number and size of filters used in the initial convolution block could be specified by the user in a configuration per game. The other convolution block and residual block had double the number of filters of the first block, but in the convolutional layer of the value and policy head there were 6 and 8 filters respectively. Additionally, both heads had a dense layer consisting of the same configurable number of nodes, unlike in the final design. The residual block in this network also used average-pooling, the reason for which can be found in Section 4.2.3.

This transitional version of the network, although not performing great in practice, was a good foundation for the final version. The goal of improvements made to this version was to make it more scalable, more precisely configurable, and better optimized for training and more complex games. This was done by using a configurable number of residual blocks, removing the dense layer from the policy head, and reducing the number of filters in the heads' convolutional layers. Additionally, pooling was removed to due poor performance, and lack of time for improvements to include it in the network.

4.2.5 Current Design

Through many iterations of implementation, testing, training and evaluation, we have reached a more configurable and flexible network architecture. The latest network architecture is also a lot more similar to that used by Google's DeepMind in AlphaGo Zero and AlphaZero (Silver, Schrittwieser, et al., 2017, Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, et al., 2017).

The network is a residual network, very similar to the ones used for image recognition. It consists of a single convolution block, followed by a variable number of residual blocks, then split into two heads, one returning the value, and one returning the policy victory. An example of this architecture can be found in Figure 4.5. To help understand it better, diagrams are provided from "AlphaGo Zero explained in one diagram" (Foster, 2020) which provides an easy-to-understand summary of DeepMind's project. Some of the diagrams have been modified according to the differences between AlphaGo Zero and our implementation.

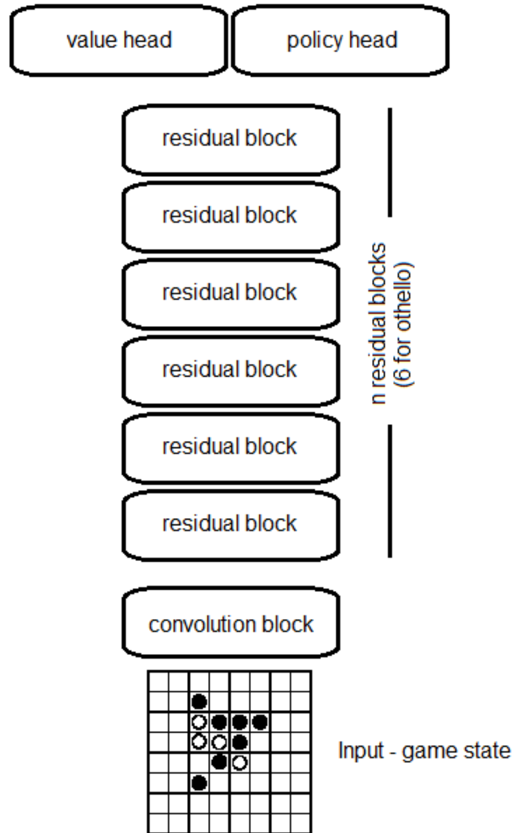

Figure 4.5: Example of a network used for training an Othello player with 6 residual blocks

---

This description is still only a rough overview of the network architecture on a high level, so to gain a better understanding of the network architecture and summarise the reasons for each element, we will describe all the mentioned parts in more detail. The precise description and purpose of each block of the network can be found below.

4.2.5.1 Convolution Block

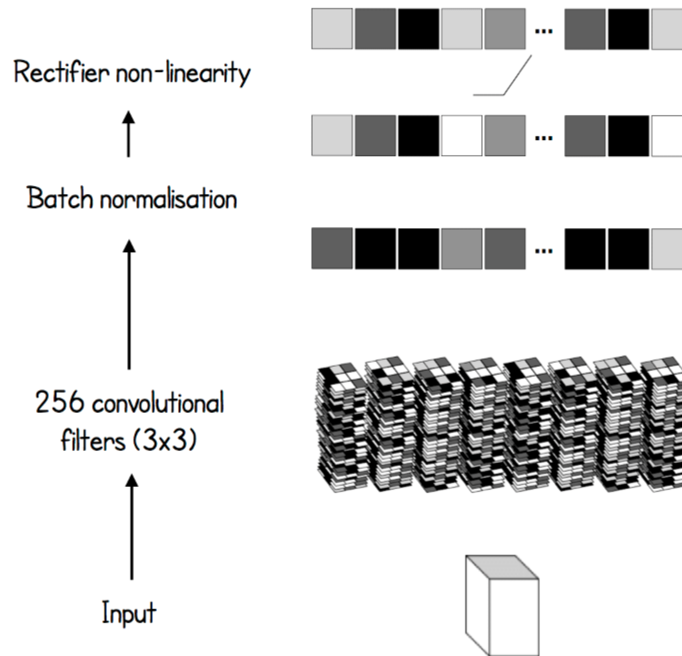

Figure 4.6: Example convolution block with `numFilters=256` and `kernelSize=3`

The first block in the network is always a convolution block. It consists of 3 elements:

Convolutional layer This layer consists of filters of a configurable size. These filters shift around the input state with a stride of 1 in both axes and always calculate the padding so that its output dimensions (apart from depth which is equal to the number of filters) are the same as those of the input.

batch normalization Batch normalization is applied after every convolutional layer in the network. For the exact reasons why, see Section 4.2.3.

ReLu activation The ReLu activation function is a crucial element of the block and network, as it outputs the activation for each neuron of the previous layer. This introduces non-linearity to the output allowing the network to learn and perform more complex tasks.

4.2.5.2 Residual Block

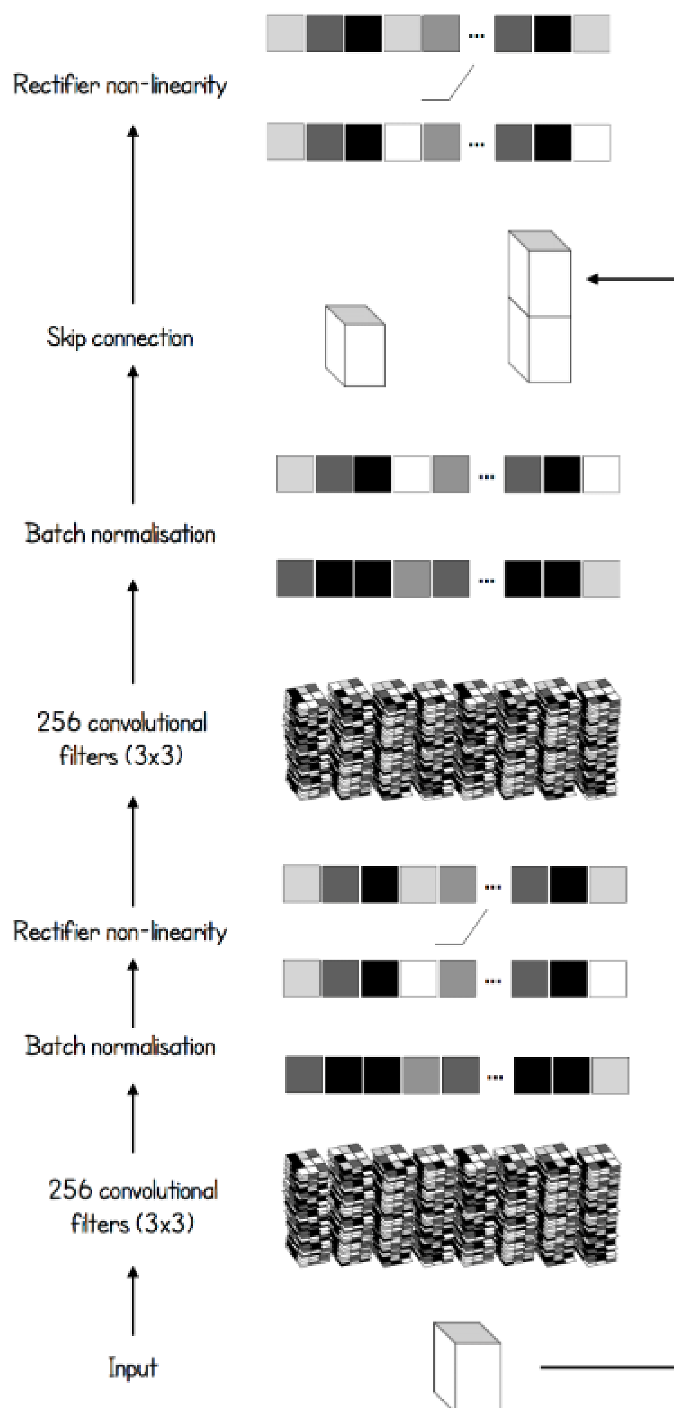

Figure 4.7: Example residual block with `numFilters=256` and `kernelSize=3`

The residual blocks contribute the most to the size of the network. The block can be described as an extension of a convolution block (see in detail in Section 4.2.5.1), because its convolutional layers, batch normalization, as well as activation function, are the same.

The residual blocks are followed by another convolutional layer, the same as the first one. Next, we use the most crucial element of a residual block — the skip-connection. A skip-connection works by adding the original input of the block back to its output before the last activation. This layer reduces accuracy degradation, as well as training error, which allows for the learning of every block equally from the original input, no matter how deep it is into the network. All that's left is the final activation of the block, the output of which will be passed on to the next residual block, or the value and policy heads.

4.2.5.3 Value Head

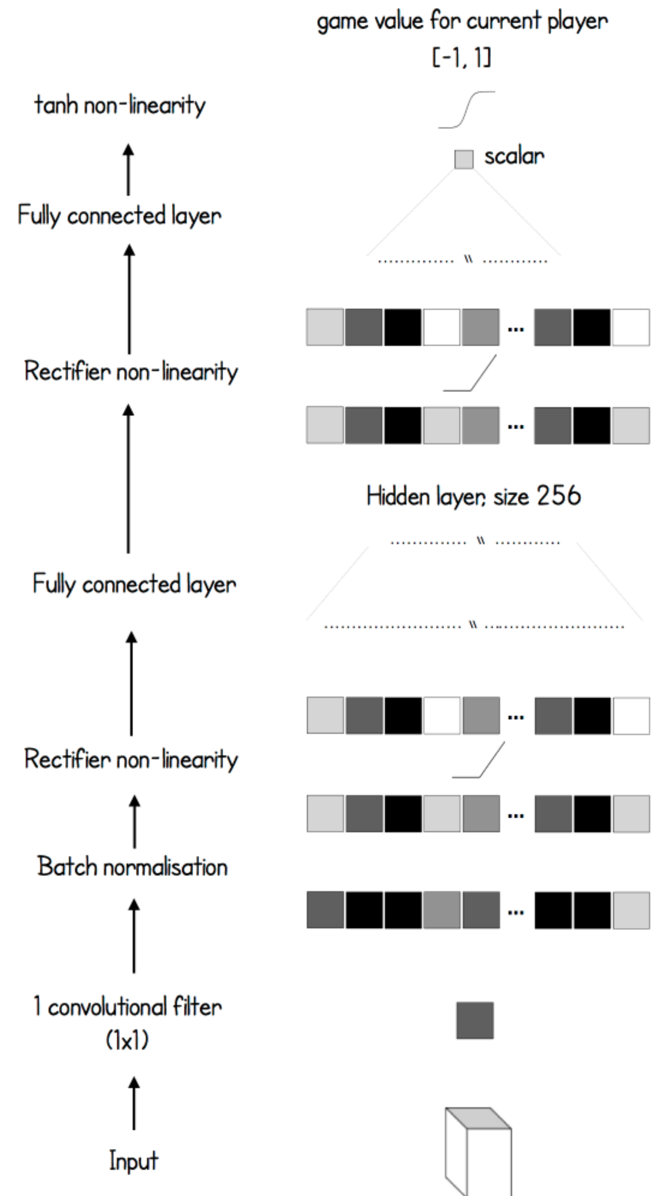

Figure 4.8: Example value head with hidden layer size of 256

The value head is a series of layers directly following the residual blocks. It is parallel to the policy head due to them working on the same input but having outputs with different sizes and different purposes.

The value head starts with a convolutional layer with a single 1 x 1 filter. The role of this filter is to use the output of the last residual block, which in the case of the example given in Figure 4.7 would consist of 256 channels, and translate it onto one channel. This output, just like that of all the previous convolutional layers, will be passed on to batch normalization and a ReLu activation function. For higher accuracy of the value output, this head consists of another fully connected layer with a configurable size, followed by another ReLu activation and a fully connected output layer. Since the expected value output is between -1 and 1, the activation used for the output layer is a tanh function, which has results in the same range.

4.2.5.4 Policy Head

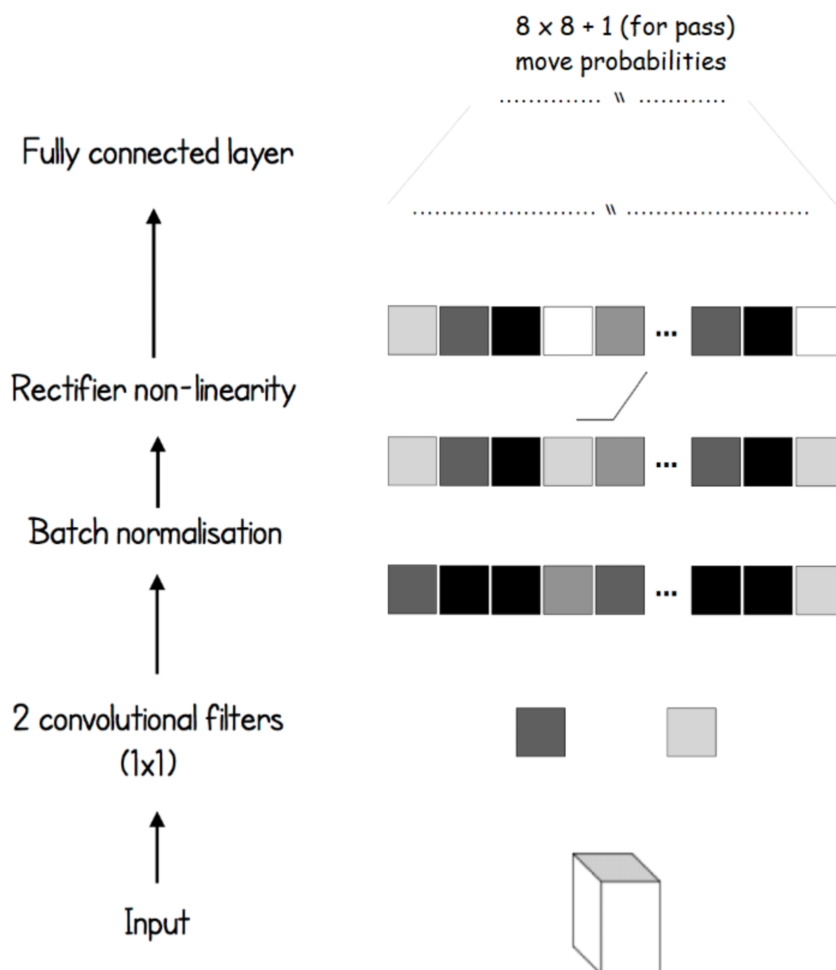

Figure 4.9: Example policy head for Othello

The policy head, just like the value head, directly follows the last residual block. It starts with a convolutional layer, again using 1 x 1 filters to reduce the number of channels between the output of the last residual block and the input of the last fully connected layer. Unlike the value had, the policy head uses two such filters, which helps with the more complicated policy vector output. The two channels will be passed to the batch normalization and a leaky ReLu activation function. The head ends with a fully connected output layer which uses a softmax activation function since the output is a probability distribution that must sum to 1.

4.3 Monte Carlo Tree Search Design

4.3.1 Description

Monte Carlo tree search is one of the most important parts of our system. The search travels through the state trees of the games until a certain iteration count is reached. This traversal is done in such a way that iterations find a balance between exploring new states, and visiting states which already have been estimated to have a favorable outcome. If this balance between exploration and exploitation is done well, more iterations should result in better approximations of which moves are good. MCTS is a heuristic search algorithm commonly used in decision-making processes as we can see from *Monte-carlo tree search* by G. M. J.-B. C. Chaslot (2010). Their tree works by creating a search tree from a single node representing the current state of the game. From this node, the algorithm then has to follow 4 steps: Selection, Expansion, Simulation, and Back Propagation.

The Selection step is defined by finding the best child node for a specific game state (node) in the tree. Expansion step starts after a node is selected as the parent node. The algorithm expands the parent node by adding multiple child nodes that represent possible game states from the selected node. Simulation is the step where the algorithm plays out a game from the newly added nodes by choosing random child nodes until a terminal state is reached. This is also called the rollout step. Back-propagation is where the algorithm updates the information of the visited nodes during the simulation.

Our specific implementation does not use simulation, rather it uses the value from the neural network to estimate how good a position is. Theoretically, if getting an output from the network was fast enough this would save time by not performing the rollout, while also getting a more accurate estimate if the network is trained well.

4.3.2 Initial Design

Our first working implementation of Monte Carlo tree search was based on the work by Nair (2017). This implementation does not model a tree, rather it uses hash maps to store all the necessary information.

It was a quick way to get an implementation we could use and improve later. While writing it, we already thought of ways we could improve the performance. Our main ideas at the time were to add parallelization and to find better data structures than hash maps.

4.3.3 Design Choices

As stated before, we had two main ways of improving the Monte Carlo tree search performance that we already knew of before the first implementation was working. Later on we also discovered the Java Native Interface (JNI) (Oracle, n.d.), which could be used to run code from other languages, possibly allowing further speedup.

Based on a paper by G. Chaslot et al. (2008) we have discovered that there are multiple ways to parallelize the Monte Carlo tree search. These consist of:

Leaf parallelization
: This method has been introduced by Cazenave and Jouandeau (2007) and is one of the easiest ways to parallelize MCTS. This works by having one thread that traverses the tree and adds nodes until a leaf node is reached. Then for each node starting from the leaf node games are simulated using multiple threads. Afterward, the results are back-propagated by one single thread.

Root parallelization
: G. Chaslot et al. (2008) define root parallelization as "building multiple MCTS trees in parallel, with one thread per tree." This method works by creating multiple MCTS trees at the same time, where each tree is searched by one thread. After the time to decide a move has elapsed, all the MCTS trees are merged, and the scores of the games are calculated. The best move is selected based on the scores of all the games played.

Tree parallelization
: This method works by having one shared tree from which simulations of the games are played. The idea is that each thread could modify the information in the tree. This method requires the implementation of mutexes, a synchronizer, to ensure no concurrent modifications happen which can corrupt the data. In addition, we see that there are two ways to improve this method: mutex location and virtual loss.

Initially, no parallelization was being used, meaning that it was not able to explore multiple options simultaneously within the game tree. This limitation results in the algorithm finding a less optimal move or taking longer to run down the tree search. By implementing parallelization of MCTS we can speed up the process of traversing the tree and explore more outcomes, leading to better move choices and reduced time to search the tree. In addition to the parallelization of the MCTS, if we would make use of the tree parallelization method, we should develop a virtual loss function to help the search perform better as we can see from the paper by Mirsoleimani et al. (2017).

We also researched the paper by Enzenberger and Müller (2009) where they discuss how the efficient parallelization of MCTS is achieved using an increased number of threads and a lock-free parallel MCTS that improves the overall scalability of a Fuego GO program. This is interesting research, but we did not get to experiment with it, because we had performance issues much more important than the speed of the mutexes and locks.

With regards to the mutex location, the main factor in optimizing this is ensuring that as many calculations as possible are moved out of the code section that requires the mutex.

A paper by Mirsoleimani et al. (2017) discusses how virtual loss encourages exploration of the search space when using tree parallelization. The tree in MCTS is defined by nodes and edges. When a node is expanded, a child node is added and virtual loss is added to the parent node. This encourages the other threads that are running MCTS to explore other paths in the search tree, rather than selecting the same parent node and its child. Consequently, the tree search would explore more alternative paths in the game tree.

When testing with tic-tac-toe, we could see while profiling the code that a lot of time was spent retrieving information from the hash maps. This is why we looked into using another internal data structure: non-binary trees, hopefully leading to better performance.

To test if using the JNI to run native code would make a performance difference we also tested a Rust implementation of Monte Carlo tree search.

4.3.4 Design Trajectory

In this section, we will look into how we can improve our initial Monte Carlo tree search implementation based on the ideas from Section 4.3.3.

4.3.4.1 Parallelization

There are multiple ways to parallelize Monte Carlo tree search. These can be divided into two categories: (i) parallelization strategies that parallelize a single Monte Carlo tree search (like the approaches mentioned in Section 4.3.3); and (ii) parallelization strategies that run multiple Monte Carlo tree searches at the same time. The first approach has a lot more overhead and implementation effort than the second approach, but does have the advantage that it improves a single Monte Carlo tree search. Speeding up a single search allows the AI to perform better in a competitive setting where a single game is being played, and the AI has a set amount of time to make a move. During training the second approach works just as well as the first. Training requires a lot of Monte Carlo tree searches to occur in order to play games, whether it be for self-play or tournaments against previous generations. These games are independent, meaning we can just play the games on different threads, rather than use concurrency to play single games. The second approach should even be faster in a lot of cases because it doesn't need to account for virtual-losses or mutexes.

Currently, we have implemented the second approach to speed up the training. We do have some work done on implementing the first approach as well to improve the performance during actual game-playing, but this has not been validated to work, and initial testing of this implementation has shown no clear improvement by adding more threads (Figure 4.10, Figure 4.11), leading us to believe there are some major issues with the untested implementation.

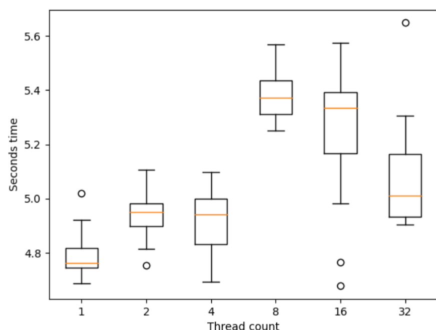

Figure 4.10: Time to play multiple tic-tac-toe tournaments of 2 games, each tournament increasing the amount of Monte Carlo iterations by a factor of 2 up to 5000. CPU: AMD Ryzen 7 1800X @ 3.60GHz.

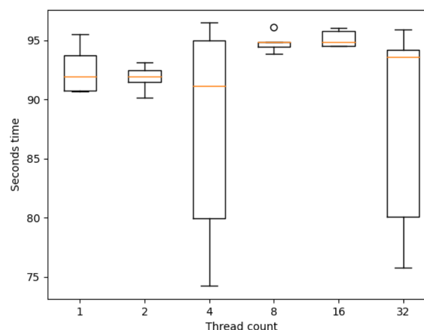

Figure 4.11: Time to play multiple tic-tac-toe tournaments of 2 games, each tournament increasing the amount of Monte Carlo iterations by a factor of 16 up to 1048576. CPU: AMD Ryzen 7 1800X @ 3.60GHz.

4.3.4.2 Data Structure

To test whether a different data structure could make a difference we reimplemented Monte Carlo tree search using a tree based data structure, basing the math on AlphaZero (Silver et al., 2018). We tested this new implementation against our original by running multiple tic-tac-toe tournaments of 2 games, each tournament increasing the amount of Monte Carlo iterations by a factor of 16 up to 1048576 (Figure 4.13). This test showed a drastic difference in performance, the new implementation was 9 times as fast in a single test. However, upon further inspection, we discovered that this difference would not be reasonably achievable in real world scenarios (Figure 4.12). Our new implementation is a lot quicker in tree traversal, where the data structure matters a lot. Though, when creating new parts of the tree, it did not make such a large difference, since adding a new game state uses a prediction from the neural network. Our initial test ramped up to an iteration count that was a more than an 25 times greater than the total amount of legal states in tic-tac-toe Van Cranenburgh et al., 2007. When we decrease the amount of iterations to be smaller than the amount of legal tic-tac-toe states, the advantage disappears.

Our new implementation is only much faster when running more iterations than the amount of legal game states, which is not reasonably achievable in most games, and if

Monte Carlo tree search ever gets so far, it should not drastically improve further by running more iterations. Because the advantage is minimal, and it would take a lot of time to test, we have chosen to keep our original data structure.

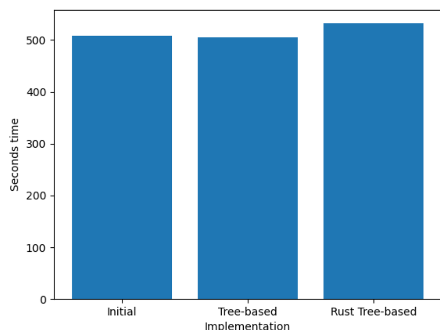

Figure 4.12: Time to train a small network, while using different Monte Carlo tree search implementations to generate self-play games for training. CPU: AMD Ryzen 7 1800X @ 3.60GHz.

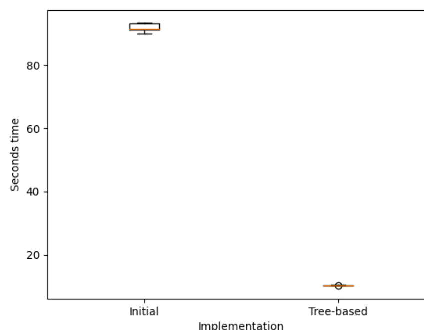

Figure 4.13: Time to play multiple tic-tac-toe tournaments of 2 games, each tournament increasing the amount of Monte Carlo iterations by a factor of 16 up to 1048576. CPU: AMD Ryzen 7 1800X @ 3.60GHz.

4.3.4.3 Java Native Interface

We found out that Java code can leverage compiled native code by using the Java Native Interface (Oracle, n.d.). To see if this would make a difference in the performance of Monte Carlo tree search we rewrote our tree based implementation in Rust and compared it to the Java version. The tree-based version was chosen because it would be hard to implement a hash-map-based implementation due to the fact that we do not have the state objects in Rust.

To test the Rust implementation against the Java implementation we tested how fast each implementation could train a small network, while using different Monte Carlo tree search implementations to generate self-play games for training. In this test, the Rust implementation took 5% longer than the Java implementation (Figure 4.12). We expect this to be due to the overhead added to the Rust implementation whenever it has to communicate with the Java VM (for example when requesting the prediction from the neural network).

Similarly to our finding from the previous section Section 4.3.4.2, if the neural network would be able to make predictions faster, the difference might have been more significant. Monte Carlo tree search only constitutes a small part of the overall JAGGER process when excluding getting predictions from the neural network, so any gains that can be made by running native code would be small. There is also the overhead of communicating with the JVM whenever we need to add new states to the tree, which makes the process

take longer proportionally to the amount of nodes we add to the tree. If we combine these two factors we get a situation where using Rust has minimal gains, and quite a bit of cost. If the Monte Carlo tree search made up a larger part of the JAGGER process it might have been worth it, but that is currently not the case.

4.3.5 Current Design

Our current design of Monte Carlo tree search is very similar to our original implementation, mainly being changed by some bug fixes. Internal parallelization could improve the situation while playing games, but would not make a difference in training. Sadly, our parallel implementation did not speed up how many iterations could be run in a certain time, but this could be due to an error on our part. There is potential to speed up our implementation by using another data structure, but this potential is less than 5% when choosing a reasonable iteration count. Because we found some bugs in our Monte Carlo tree search quite late into the project, and validating another implementation would take a lot of time, we have decided to stay with our hash-map-based implementation. The Java Native Interface was interesting to use, but did not result in a performance improvement. One way to possibly achieve this would be to find a fast way to communicate with the neural network, by also porting this to native code.

Chapter 5

Manual

In this chapter, we will explain how to use JAGGER. In different sections, we will explain how to train the AI, how to add a new game, how to change hyperparameters, and what machine-specific settings you can adjust.

5.1 Adding a New Game

This section will go over the necessary steps to add a new game.

5.1.1 Implementing the New State Interface

The first step to adding a new game is to implement the new `State` interface. To use `State`s as input for our network, we had to define a standard for encoding a state to an array of numbers. We added a method `stateAsRank4Array` to the `State` interface. In the implementation of this method you have to encode the game state in a 4-dimensional `NDArray` (Konduit, n.d.-a). You can use the already-implemented games as inspiration.

Similarly to AlphaZero, for the already implemented games, we implemented `stateAsRank4Array` by creating the array from the perspective of the current player (Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, et al., 2017). For example, a 1 in the array means a piece of the current player, rather than a piece of player 1. Similarly, a -1 in the array might mean a piece of the opponent of the current player, rather than a piece of player 2. This removes the need for explicitly encoding whose turn it is. If the game is not equal for both players, it is also necessary to encode the player's color. In Go, for example, the white player receives an extra point as compensation for playing second.

For tic-tac-toe, Pentago, and Othello, we only used one channel, i.e., the 4-dimensional array has size 1 on the first two dimensions. For Collecto, we used additional channels to encode the balls each player had in their possession. After the channel that encodes

the board, we use an additional channel per color, per player, resulting in $6 \cdot 2 = 12$ additional channels. The first 6 channels represent the balls of the current player and the second 6 of the opponent. The channel is filled with the number of balls the player has of that color.

5.1.2 Constants

The second step in adding a new game is to add the appropriate constants for it. In `Constants.java` you will find a list of constants, grouped by game. For your new game, you will have to add the same constants, possibly with different values. You can easily do this by copying and pasting the constants of any existing game and then adjusting the names of the pasted constants to reflect the new game. See an explanation of all constants below.

`LEARNING_RATE` This is a learning rate schedule. It is based on a map that maps an iteration to a learning rate. It is important to note that the iterations used here do not align with the iterations JAGGER uses (also called: training iteration, training cycle). Instead, the iterations in the learning rate schedule represent the number of times something was fitted to the neural network. This counter is maintained by DL4J. Every call to `JaggerNetwork.fit` increments the counter by one. That means that the number of iterations in the learning rate schedule divided by the number of epochs and by `Fitter.NUM_PARTITIONS` results in the number of JAGGER training iterations.

`C_PUCT` The c_{puct} used inside `MonteCarloTreeSearch`. The constant is used to determine the level of exploration during the Monte Carlo tree search. See the papers on AlphaGo and AlphaGo Zero for more information (Silver, Schrittwieser, et al., 2017; Silver et al., 2016).

`DIRICHLET_VALUE` The value α to generate Dirichlet noise $Dir(\alpha)$. It should be scaled in inverse proportion to the typical number of legal moves in any position of the game. See the papers on AlphaZero and AlphaGo Zero for more information (Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, et al., 2017; Silver, Schrittwieser, et al., 2017). You can estimate a good value by comparing the typical number of legal moves of the new game to the already existing games and the games used in the AlphaZero paper.

`HIDDEN_NODES` The number of hidden nodes in the non-output dense layers. Currently, there is only one such layer in the value head. If the game has a large number of states for which values vary a lot, this value may need to be raised for the predictions to be more accurate.

`DIRECTORY` The directory where models will be saved during the training of the game.

`X` The x size of the network's convolutional layer. Usually, this will be equal to the x dimension of the game board. Since NDArray dimensions use zero-based numbering, this number should be equal to the size of dimension 3 of the 4-dimensional array returned

by `State.stateAsRank4Array`, i.e., `state.stateAsRank4Array().size(3) == X` should hold.

`Y` The y size of the network's convolutional layer. Usually, this will be equal to the y dimension of the game board. This number should be equal to the size of dimension 2 of the 4-dimensional array returned by `State.stateAsRank4Array`, i.e., `state.stateAsRank4Array().size(2) == Y` should hold.

`CHANNELS` The number of channels of the network's convolutional layer. For board games with just two different colored pieces, the number of channels can usually be set to 1. This is the case for tic-tac-toe, Pentago, and Othello, for example. For Collecto, we used additional channels to represent the number of balls a player has for each color. Each channel represents a different color, where the channel is filled with the number of balls the player has of that color. The number of channels should be equal to the size of dimension 1 of the 4-dimensional array returned by `State.stateAsRank4Array`, i.e., `state.stateAsRank4Array().size(1) == Y` should hold.

`KERNEL_SIZE` The kernel size of the convolutional layers in the network. This value should always be lower than the board size of the game, but high enough to detect important patterns. For example, a value of 2 for tic-tac-toe would be appropriate, but a value of 3 or 4 would be better suited for Othello.

`FILTERS` The number of filters of the convolutional layers, i.e., the number of outputs of those blocks. This value should be lower for games with simple patterns, but higher for games where patterns, or in deep learning terms "abstractions", might be quite complex (for example Chess).

`RESIDUAL_BLOCKS` The number of residual blocks of the network. This should scale to the complexity of the game. If the accuracy of predictions seems not to improve from a certain point and is not yet satisfactory, a higher number should be used.

5.1.3 ConfigUtils

Thirdly, you have to adjust `ConfigUtils.java` to support the new game. `ConfigUtils` is used to easily get the right constants for a given `State` class. To support the new game, add an `else if` statement to each of the methods in `ConfigUtils`. The `else if` statement should contain the appropriate superclass or interface to check whether the given state is of that type, and it should return the appropriate constant. The already existing code is self-explanatory and can be easily extended for new games.

5.1.4 Trainer Class

The fourth step is creating a new trainer class. For convenience, each game has its own trainer class, all located inside the `jagger.games` package. For the new game, create a new package containing the new trainer class. The class can pretty much be copied from

any of the other trainer classes. If the new game has the same starting state in each game, you can refer to the tic-tac-toe, Pentago, or Othello's trainer class as an example. On the other hand, for games where each game begins with a different state, Collecto's trainer class can serve as a suitable example.

5.1.5 Trainer Arguments

Lastly, the `TrainerArguments` have to be configured. Most of these arguments depend largely on the amount of time you are willing to spend on training. All arguments are described in detail below.

`monteCarloIters` The number of Monte Carlo iterations to execute per move. A higher number results in higher-quality data, but also takes more time. We mostly used a value of 50, however, we turned it down to 30 for Othello because it is a relatively long game compared to the others, and we could not afford to run it with 50 Monte Carlo iterations for each move.

`cPuct` The c_{puct} value. It is explained in Section 5.1.2. You will usually just use the `getCPuct` method from `ConfigUtils`.

`dirichletValue` The Dirichlet value. It is explained in Section 5.1.2. You will usually just use the `getDirichletValue` method from `ConfigUtils`.

`numTrainerIters` The number of training iterations to run. One training iteration consists of self-play, fitting, and the tournament. Make sure not to set this number too low if you do not want it to finish in the middle of the night when you intend to let it train overnight. Setting the value too high is no problem because the process can be stopped at any time, and the last models will automatically be saved to your file system.

`numEpochs` The number of epochs to fit the data with. We mostly used a value of 10.

`numSelfPlayGames` The number of games to play in the self-play state. We mostly used a value of around 100. However, after many iterations, you might find that the best network and training network win the same number of games against each other, indicating that the training network does not learn enough. At that point, it might be wise to increase the number of self-play games and resume the training.

`tournamentGames` The number of games to play in the tournament. We recommend setting this to an even number, to make sure both players can play first the same number of times. We often used a value of 40, except for Othello where we decreased it to 18 to save time.

`tournamentScoreThreshold` The number of wins by the training network minus the number of wins by the best network must be strictly larger than this threshold in order to declare the training network as the new best network.

5.2 Training a Game

Once set up, training a game is very straightforward. You only have to run the appropriate trainer class. The class can easily be run using IntelliJ. From a command line, it is easy to use the Maven `exec:java` goal. More detailed instructions can be found in the README.

5.3 Resuming Training

Because JAGGER saves the best and training networks after every iteration, it is very easy to resume training if the process was stopped. To do this, you can use the `Trainer(StateGenerator, JaggerNetwork, JaggerNetwork, int)` constructor inside the trainer class. In addition to the other constructor, it takes the best network, training network, and starting iteration.

If you stopped the process after iteration 50 finished, your model output directory will contain iteration directories up until `iter_50`. Assuming the model output directory is `./models/Othello`, the following code snippet illustrates how to resume training from iteration 51.

```
OthelloLongsState state = new OthelloLongsState();
Trainer t = new Trainer(() -> state,
        JaggerNetwork.loadNetwork(
                "./models/Othello/best_network.zip"),
        JaggerNetwork.loadNetwork(
                "./models/Othello/iter_50/network.zip"),
        51);
```

5.4 Global settings

Other than the game-specific configurations we discussed in Section 5.1.2 and Section 5.1.5, there are also some global settings that can be changed. This section will be used to explain them.

`Fitter.NUM_PARTITIONS` sets the number of partitions that should be made before fitting a dataset. As discussed in Section 4.1.3.5, fitting too much data at once, causes DL4J to run out of memory. Depending on the game, the number of self-play games, and your GPU, this value must be set high enough to prevent crashes. For reference, an NVIDIA RTX 3060 TI can handle about 10 Othello games per fit, which means 10 partitions when `numSelfPlayGames=100`.

`MonteCarloTreeSearch.NOISE_QUOTIENT` sets the percentage of Dirichlet noise that should be used. Just like in AlphaGo Zero and AlphaZero, it is set to 0.25 (Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, et al., 2017;

Silver, Schrittwieser, et al., 2017). This value remains constant during the entirety of the training. However, when using JAGGER outside of training, the quotient should be set to 0 to always select the best decision.

5.5 Using JAGGER

Of course, JAGGER can be used as an AI, outside of training. You can use either `TimeBoundJagger`, or `IterationBoundJagger`—the difference between the two is explained in Section 4.1.4. Let us say you spent some time training an Othello network and now want to use the AI on the module 2 server, giving it 10 seconds per move. You can create the AI like this: `var jagger = new TimeBoundJagger(new OthelloLongState(), 10000, "./models/Othello/best_network.zip");`.

5.6 Troubleshooting

During the project, we sometimes woke up seeing that training crashed in the middle of the night. Usually, we saw a `cudaMalloc` error, and other times some other CUDA-related error. Almost always, the problem was caused by the GPU running out of memory during fitting. Possible solutions to this problem are increasing `Fitter.NUM_PARTITIONS`, or decreasing `numSelfPlayGames`.

Chapter 6

Testing

6.1 Test Plan

6.1.1 Unit Tests

We have decided to write unit tests for our project and use the Test Driven Development philosophy where appropriate. We take this approach because certain parts (like creating the initial AI that plays legal moves) are much more testable than others (like training a model). Sometimes TDD is perceived as cumbersome, but we feel that using it where feasible would be a good approach. Firstly, an advantage of TDD is that it allows us to experiment more freely because of the increased certainty that we do not break the parts created using TDD. Fast experimenting is especially useful for us because we are making an AI, which will require us to test a lot to increase the performance of the AI. Secondly, TDD forces us to keep on top of testing, preventing us from neglecting unit tests or leaving them until the last moment. Finally, it allows us to check that our code works as intended. If test writing occurs after coding, a risk exists that the test is written with the code in mind (like forgetting edge cases not addressed by the code).

We decided that this hybrid approach is appropriate for JAGGER. The fundamentals of deep-learning AIs can be quite math-heavy, making it easier to write the tests for development, while the fine-tuning of the AI does not suit TDD. The requirements for unit tests are not final and will change according to the design. Our testing requirements are as follows:

- Every class of a component using TDD should have unit tests.
- Every public method of a component using TDD should have unit tests.
- TDD should be considered as a testing approach for all components.
- All commits on components using TDD must work towards implementing a test.

6.1.2 Integration and System Tests

We can create two types of tests to test a general gameplay AI. The first type is pass/fail tests, which can test functionality or milestones. The other type of test is one where the AI receives a score instead of a pass/fail. The scored tests can measure more complex things like the improvement of an AI over time.

When it comes to JAGGER, we have thought of the following tests to pass as milestones:

1. Play a legal game against an opponent using the Java interface.
2. Beat a random move AI.
3. Beat the provided strong AIs.
4. Beat strong AIs from the internet.

Some tests are required to deliver a functioning product, while others are more ambitious (like beating a strong Othello AI from the internet). We consider our AI to beat other AIs if it wins more games than it loses with a set number of matches to be specified once we gain more experience and knowledge after designing and implementing the MVP.

Aside from these milestones, there is also the question of how strong the JAGGER AI generally is. To measure this, we first considered AlphaZero's approach to measuring AI strength (Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, et al., 2017). AlphaZero's strength was measured by matching the AI after a certain amount of training steps against another AI with a known ELO rating. Then based on how it performed, they could estimate an ELO rating relative to the baseline AI. After some consideration, we decided not to use this approach. We do not expect the games we will be playing to have well-established ELO scores. Furthermore, assessing the ELO rating of an AI takes a large number of matches, which we cannot play due to time constraints (giving each side of the board five seconds to make a move in a game of forty total moves would take more than two days to play a thousand games).

Instead, we have decided to use the results of the matches to evaluate the AI directly. We would try to increase the percentage of wins and decrease the percentage of losses. There are multiple AIs we can use for comparisons. These options include: (i) random move AIs, (ii) AIs provided by the client, (iii) less trained versions of JAGGER, and (iv) AIs from the internet. We plan to test against at least (i) and (ii), and more if time permits.

6.2 Test Execution

6.2.1 Unit test execution

6.2.1.1 Test coverage

As seen in Table 6.1, we have written unit tests for most of the code we have written. Generally speaking, 80% coverage is a good target to aim for. We managed to achieve

Package	Class	Method	Line
arguments	MCArguments	100% (1/1)	100% (1/1)
↪	NetworkArguments	100% (1/1)	100% (1/1)
↪	TrainerArguments	100% (1/1)	100% (1/1)
mafdet	Mafdet	80% (4/5)	75% (31/41)
↪	Result	100% (6/6)	100% (18/18)
mcts	Edge	100% (3/3)	100% (8/8)
↪	MonteCarloTreeSearch	50% (4/8)	65% (52/79)
neuralnet	Trainer	83% (10/12)	51% (71/137)
utils	ConfigUtils	100% (4/4)	90% (36/40)
↪	ListUtils	100% (2/2)	100% (9/9)
↪	OSUtils	100% (1/1)	66% (2/3)
	PolicyVector	100% (16/16)	98% (59/60)
	Prediction	100% (1/1)	100% (1/1)
	Value	100% (5/5)	100% (11/11)

Table 6.1: Unit test coverage

an average of 94% coverage for the methods and 89% coverage for the lines. We are satisfied with these results, as this is on the higher end of the spectrum where the system is properly tested and potential issues can be quickly identified and fixed. Complete coverage was not our goal, as we did not want to spend too much time writing tests, especially in the short amount time we had for this project. This way we also had time to focus on testing and evaluating the AI as described in Section 6.2.2.

Some deficiencies in the coverage can be explained by parts of classes that are not easily testable, or that are tested manually. For example, in the `Trainer` class, all basic (helper) functions are tested automatically, but it also contains the `train` method. This function is not easily testable with unit tests, but it does take up a major part of the class, heavily affecting the coverage. `Mafdet`'s coverage also suffers from this, as it contains a `main` method.

6.2.1.2 Test Driven Development Execution

Overall, we managed to keep up with our test driven development (TDD) plan described in Section 6.1.1. Test driven development was considered for each component of the system, and in some cases it was decided that it was not the best approach. Every class where we decided to use TDD has a corresponding test class with unit tests. On top of that, all public methods in these classes have at least one unit test, as can be seen in Table 6.1 (with some exceptions, see Section 6.2.1.1). Lastly, when we were writing the

initial implementations of the components, most commits aimed at passing the written unit tests.

6.2.2 Integration And System Test Execution

As mentioned in our test plan (Section 6.1.2), integration and system tests of the AI were done by playing games against other AIs. We did not use any ELO-rating system but instead used the results of the matches to evaluate the AI directly.

In the same section, we set ourselves 4 milestones to test our system and see to what extent we have achieved our goals, with a pass or fail result for each milestone.

The first milestone was to play a legal game against an opponent using the Java interface. This was already achieved early on in the development phase, as this was an integral part of the system that was needed to achieve any results.

The following two milestones were to beat a random move AI and the provided strong AIs. Whether we managed to achieve these milestones is up for discussion. More on this can be read in chapter 8, where we go into more detail about these results.

Lastly, we wanted to beat strong AIs from the internet. We did not manage to achieve this milestone, as we did not get the chance to test our AI against these systems. Most of our time was spent on the previous two milestones, as they were more important to the project. Without being able to pass those, there was no point in testing against the internet AIs.

Chapter 7

Performance

This chapter dives into the performance enhancements we made in the project.

7.1 Parallelization

Initially, we tested the performance of the game tic-tac-toe, because it is the smallest and easiest game our client provided. When we parallelized self-play, we expected a strong time decrease, especially because the parallelization did not involve any synchronization. The first bar chart in Figure 7.1 shows the results. While two threads complete the self-play stage faster than one thread, further increasing the number of threads only works adversely. As discussed in Section 4.1.3.3, multiple threads requesting predictions from the same network caused a bottleneck. When we clone the networks to prevent this bottleneck, we can see the results look more like what we would expect (see second bar chart of Figure 7.1), however, the time already increases with more than 4 threads. In this case, 4 threads are the optimal number of threads.

Figure 7.2 shows the results for the same experiment done on Pentago. In contrast to tic-tac-toe, the time does increase further with more than 4 threads, all the way up until 12, the number of logical processors of the CPU the experiment was run on.

7.2 Caching

As discussed in Section 4.1.3.2, caching was implemented to combat the bottleneck caused by sharing a neural network. Figure 7.3 shows the results for caching. In our first test, which was without cloned networks, we observed a strong decrease in self-play time. The strongest decrease of 68.1% can be seen when using 8 threads. For the sake of completeness, we also ran the experiment with cloned networks. As expected, cloning the neural networks has an adverse effect. This is caused by the fact that the cache is not shared across the different network clones. Interestingly, it still performs a bit better

than the cloned networks version without caching (second bar chart of Figure 7.1). This shows that a prediction is requested more than once for some states, in which case the cache does help. Unfortunately, we were not able to run this experiment for any other games than tic-tac-toe because the cache grew too large for the amount of memory we had available (see also Section 4.1.3.2).

7.3 Parallel Inference

Previously, we cloned the neural networks to allow getting predictions in multiple threads. According to Konduit (n.d.-c), DL4J's `ParallelInference` (Konduit, n.d.-b) could be used instead. In this section, we will research the performance of `ParallelInference`.

The results for this experiment can be found in Table 7.1. We started testing with `monteCarloIters=10` for less waiting time. First, we set a baseline; we ran 100 games of self-play with 8 threads. We then ran it with parallel inference and a batch limit of 100—the number of self-play games. This resulted in an insignificant time decrease. The idea was that when a thread asked for a prediction, the `ParallelInference` would make the thread wait because the batch limit was not yet satisfied, which would then let the thread switch to another task in the thread pool until it reached the `ParallelInference` again.

In hindsight, this did not work out that way, and DL4J would only handle 8 inferences at a time, one from each thread. In order to make full use of the batch limit, we tried running it with 100 threads now, which resulted in a significant performance boost: a 64.2% time decrease compared to not using parallel inference. To verify that this boost was not just caused by increasing the number of threads, we also ran it with 100 threads but no parallel inference. This resulted in a slight increase compared to 8 threads without parallel inference, confirming that parallel inference does make a difference.

Lastly, we tested whether this solution scaled well with a larger number of Monte Carlo iterations. We set `monteCarloIters=50` and compared running 8 threads without parallel inference to 100 threads with parallel inference. A similar time decrease of 63.0% was observed, proving that this solution scaled well with respect to the number of Monte Carlo iterations.

We also tested whether batching inferences has a positive effect on GPU inference times, since our past experience with GPU inference has shown disappointing results (Section 4.1.3.4). On an NVIDIA Quadro T1000 with Max-Q Design, the test with 100 threads and a batch limit of 100 did not end within the time it took on the CPU, so the experiment was terminated. We later found, however, that GPU inference on an NVIDIA RTX 3060 Ti did get faster as a result of parallel inference (compared to an Intel Core i7-10700 @ 2.90GHz). We did not run more experiments on this GPU because it was occupied with other tasks. Let this serve as a suggestion to consider trying running all of

JAGGER with the CUDA backend—rather than just the fitting process—if you have a good GPU.

`monteCarloIters`	**No. threads**	**Batch limit**	**Parallel inference**	**Time [s]**
10	8	–	no	321
10	8	100	yes	312
10	100	100	yes	115
10	100	–	no	379
50	100	100	yes	590
50	8	–	no	1595

Table 7.1: Performance of 100 games of Othello self-play. CPU: Intel Core i7-10750H @ 2.60GHz.

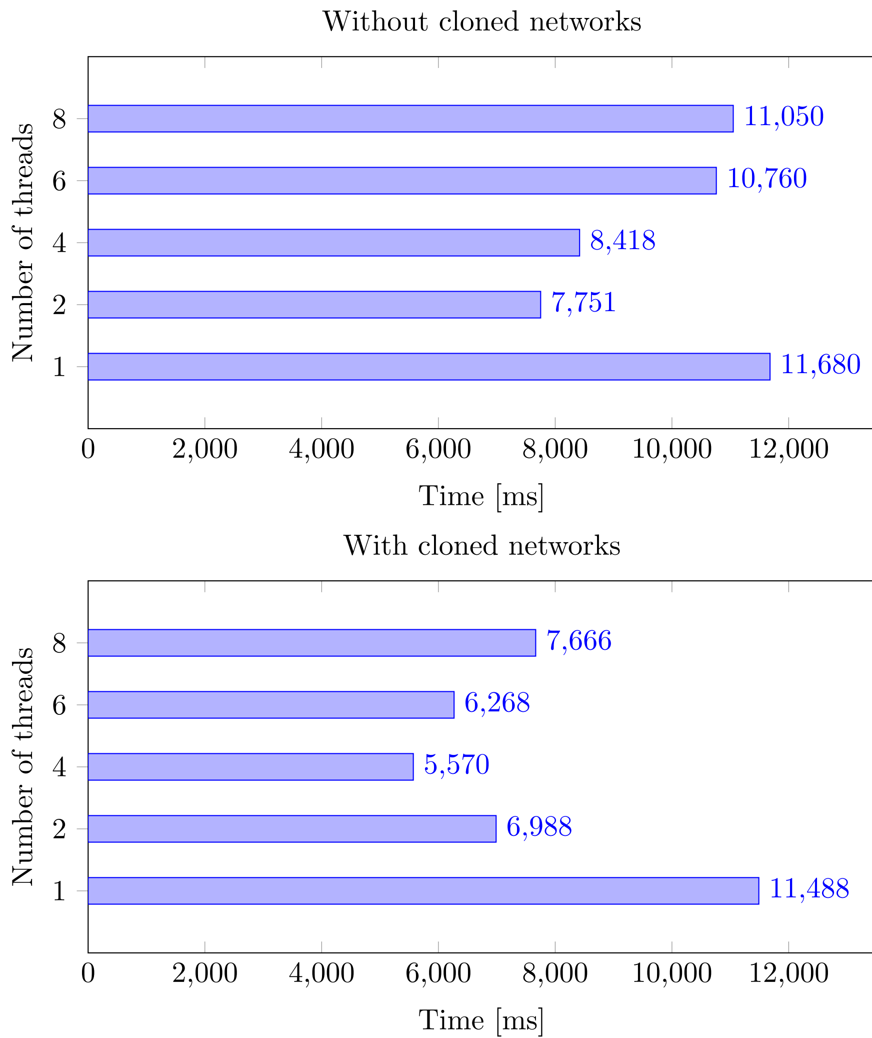

Figure 7.1: Tic-tac-toe self-play on different numbers of threads with and without cloned neural networks. Relevant hyperparameters: `monteCarloIters=250`, `cPuct=3.5`, `numSelfPlayGames=1000`. No Dirichlet noise. Initial neural network architecture. CPU: Intel Core i7-10750H @ 2.60GHz.

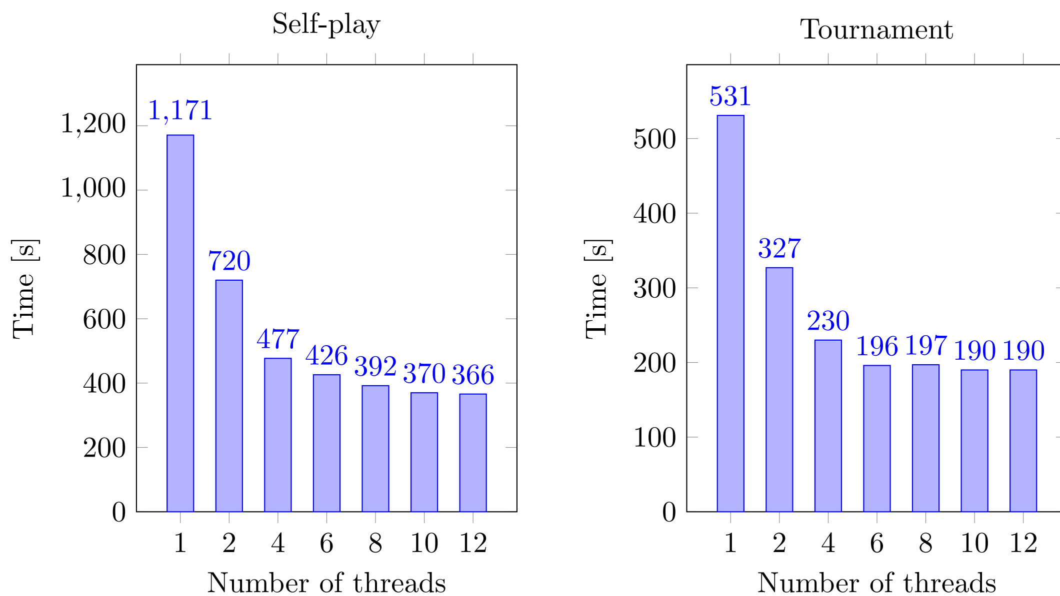

Figure 7.2: Pentago self-play and tournament on different numbers of threads with cloned neural networks. Relevant hyperparameters: `monteCarloIters=50`, `cPuct=1.4`, `dirichletValue=0.2`, `numSelfPlayGames=100`, `tournamentGames=40`. Final network. CPU: Intel Core i7-10750H @ 2.60GHz.

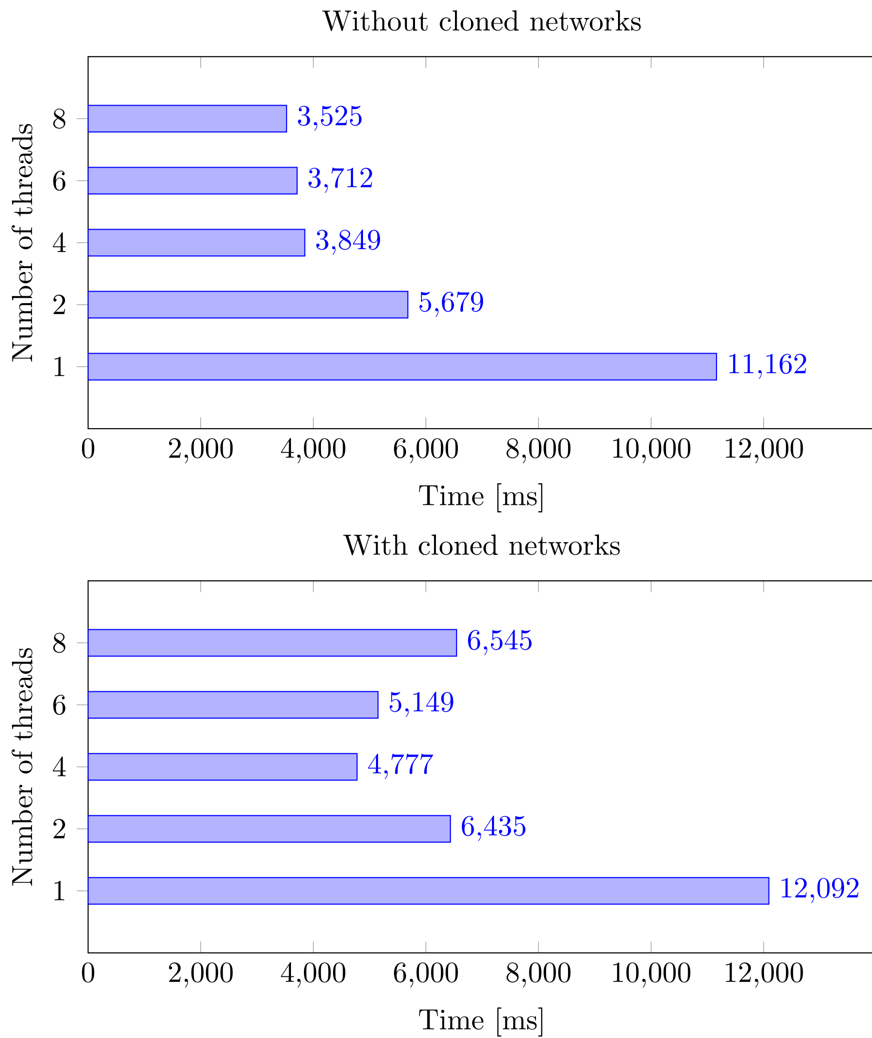

Figure 7.3: Tic-tac-toe self-play on different numbers of threads with cache, with and without cloned neural networks. Relevant hyperparameters: `monteCarloIters=250`, `cPuct=3.5`, `numSelfPlayGames=1000`. No Dirichlet noise. Initial neural network architecture. CPU: Intel Core i7-10750H @ 2.60GHz.

Chapter 8

Evaluation

The evaluation of our project consisted mainly of training and evaluating models of the neural network for the games available to us. The models were trained on suboptimal parameters due to a lack of time for hyperparameter optimization, which is visible in the results. Despite that, it is clear that all elements of the system and the system altogether work as intended. The networks show improvement in certain stages of the training process, and even when they show regression in the later stages, this is due to the parameters rather than system functionality. It is also clear that the system behaves differently with different parameters, which makes it possible to tune parameters and train a competitive AI for a game implementing the interface.

We evaluated the three trained models we created, for Pentago, Collecto and Othello. All models were evaluated through a tournament of 30 games against an AI implemented by our client, Tom. The tournament was fair, JAGGER was the first player in 15 games, and the second in the other 15. Both the AIs were given three seconds per move in each game. Pentago and Othello were played against Tom's implementation of a Monte Carlo tree search AI, but due to missing implementation of parts of the necessary framework, Collecto was played against a minimax-based AI.

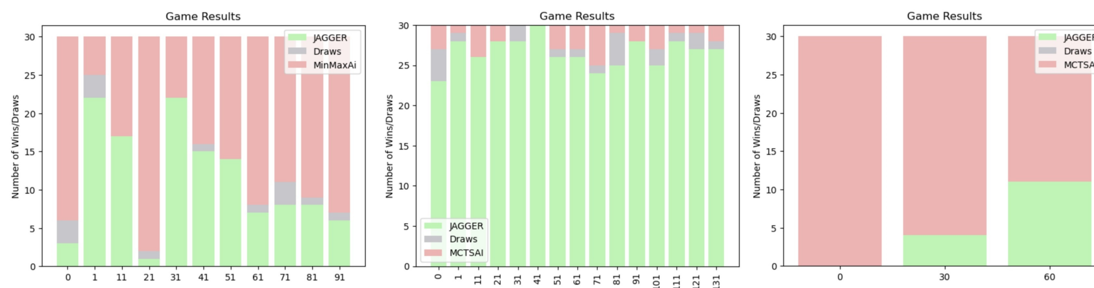

Figure 8.1: Results of 30 games of Collecto played by JAGGER against a minimax algorithm, and 30 games of Pentago and Othello against a Monte Carlo tree search AI respectively.

The models show various results. The sample of 30 games is very small considering the randomness involved, but due to time constraints, it is the largest sample size we were able to run. That means that the results cannot be considered conclusive, but they provide valuable insight regardless. In Pentago and Collecto, it seems that the models improve early, but at a certain iteration begin to become worse over time, most likely due to overfitting. We can also see a very significant improvement in the Othello model. The lack of regression in Othello can be attributed to two elements. The first reason may be the shorter (iteration-wise) training of just 60 iterations. It is difficult to predict whether the model might show regression in later stages. The other reason is that we had already learned valuable lessons from training other models early, and training an Othello model for 60 iterations became possible very late in the implementation process, due to big improvements in training speed. This meant that the Othello model was trained on hyperparameters that were suited much better for the training, showing promising results over the relatively short training considering the complexity of the game of Othello.

Chapter 9

Conclusions

To conclude, our system fulfills all requirements agreed upon by the team and our client. We were able to create a working framework for creating and training AIs for any two-player, deterministic, turn-based board game. JAGGER can now be used by our client in any way he pleases, whether it is training to play new implementations of games, matches against AIs designed by students, or as a tool in future research projects. The setup for a new game is simple, and the system is quite flexible, allowing for different sizes of networks and complexities of games, and is usable on an average personal computer. As mentioned, we did not deliver well-trained, competitive models for any of the games, but this was not in the scope of our project. The evaluation of trained models suggests that this is not due to system faults, but rather due to overfitting. We believe that this is caused by inappropriate training parameters. The fact that clear trends can be seen in the model performance suggests that it is possible to tune these parameters through analysis of training with different variables in a controlled environment. Our client was aware of the high risk of us not being able to train models in time and did not require models that would win most games. That being said, the JAGGER project is fully capable of achieving that goal with enough time and experimentation, although one must be aware that this is a lengthy process. We are satisfied with the delivered product, considering our previous lack of knowledge and experience in the domain of general game playing and deep learning.

Chapter 10

Future Work

10.1 Tuning Hyperparameters

The best way to achieve better performance of the AI in play is to train it on better hyperparameters. Many parameters have a significant impact on training speed and results. Due to this, optimization is a lengthy and troublesome process. In order to find the best parameters, one must execute multiple planned experiments, such as ones conducted by Wang et al. (2019). For example, when looking for the appropriate number of Monte Carlo tree search iterations to train an Othello AI, one could plan an experiment, where the training process is tried 3 times on different values with no other variables changing. To achieve results of any relevance, multiple iterations of the training cycle must be completed for each option of the tested variable. It is therefore no surprise, that to run such experiments for Othello, it may take multiple weeks to find the best training parameters. Additionally, the variables might affect one another in unexpected ways, for example, although making a network bigger might mean it will begin to plateau at a lower score, it also negatively impacts the time it takes to run more Monte Carlo iterations. Since, as we discussed with our client, it was not in the scope of our project to train a competitive model for each game, we decided to prioritize other aspects of the project rather than spending the majority of time optimizing training parameters. We believe, however, that this would be an interesting aspect of the project to investigate, and might be crucial when attempting to train a competitive AI.

10.2 Reducing Overfitting

In our testing and evaluation, we find that the networks reach a plateau when trained, after which it is visible that some iterations are becoming worse at playing the game. We suspect that this is due to overfitting, which causes the network to become very good at estimating the value and policy for the states it has been trained on in the last iteration but perform poorly when receiving "new" states as input. Although this can

potentially be largely improved by tuning parameters, several other steps can be taken to avoid overfitting. In their article, Wang et al. (2019) explain how in an AlphaZero-like approach, overfitting can be reduced by training on previously generated data, as well as adding dropout in the neural network. In our testing, we ran into many issues with available memory already, which would only be amplified by training on previous datasets. This is simply due to a lack of hardware with enough memory to store amounts of data large enough to use this method appropriately. Since the purpose of the system is to be trained on a personal computer, we decided that it would not be optimal to implement this feature. As to adding a dropout rate in the neural network, although it would most likely reduce overfitting, it could also increase the training time. We only trained some models for testing and evaluation, and time was a valuable resource. That's why we opted not to use dropout in training to achieve better results when stopping training somewhat early.

10.3 Rollout vs. Neural Network Prediction

The main assumption behind the idea for the AlphaZero algorithm is that using a neural network prediction as opposed to a rollout in the Monte Carlo tree search can be faster and more accurate. However, for many games, this assumption might not hold. It is possible that in the time it takes the neural network to output a prediction, the rollout can be performed numerous times, giving a relatively accurate heuristic. This is possible because, due to the nature of DL4J, predictions take a long time. Additionally, some networks might be too big for games with lower complexity, making this scenario more likely. With this in mind, two new research questions can be posed: **"For game X, after which iteration of training does the accuracy of predictions of the neural network outweigh the time it takes to receive an output?"** and **"How complex does a game have to be to justify the usage of deep learning?"**. These questions could be answered through further research and experimentation using our system.

10.4 Caching

As explained in Section 4.1.3.2, caching for larger games cost too much memory. Caching was then dropped to focus on other, more promising improvements. We still believe caching may result in a slight performance increase. Figure 4.1 shows a cache-hit percentage of 7.5% for Othello. With a working cache implementation, we can expect a speed boost of about 7.5%.

To prevent the program from running out of memory, the cache should have a size limit. Additionally, different states could have different "caching priorities".

In general, for games where the number of pieces on the board increases—such as tic-tac-toe, Pentago, and Othello—the cache should prioritize keeping states from the beginning stages of the game, rather than end-game stages. States from the beginning of a game are more likely to occur multiple times than states near the end of a game.

Conversely, games where the number of pieces on the board decreases—such as Collecto—may perform better if the cache prioritizes end-game states, as those states are more likely to occur multiple times.

10.5 Symmetries

In the paper of Silver et al., 2016, symmetries are used to get a better prediction from the neural network. This is realized through two methods:

- During the self-play stage, all symmetries of states are also generated, leading to more training data for the neural network.
- When MCTS asks a prediction of the neural network it uses a random symmetry so that the evaluation is averaged over different biases.

Because of the enlarged training dataset, we expect that by using symmetries the time it takes to train the system will be lowered as more states will be checked. To test this hypothesis we implemented symmetries for tic-tac-toe and Othello. Because parallel inference exploits large number of simultanous games, lowering the number of games did not have as much effect as we hoped. Dividing the number the number of games by 8 (the number of symmetries), for example, only reduced the time it took by 30–40%. Considering we had little time left at this point, we decided to focus on other tasks, postponing further investigation and testing of symmetries. For future work, we do expect symmetries to be a valuable enhancement.

10.6 Single Monte Carlo Tree Search Parallelization

Because of our initial bad experiences with parallelization of a single Monte Carlo tree search, we have not been able to implement this as part of our project. Later experiments have however shown that using parallel inference, a significant speedup could be achieved by allowing parallel tree searching. Unfortunately, this was discovered in the last days of the project, so we were unable to add this feature, but it should not be difficult to do so. This only affects the performance when using JAGGER for game playing, not when training as discussed in Section 4.3.4.1.

10.7 Monte Carlo Tree Search Data Structure

Even though we have run some tests showing small potential when using other data structures for the Monte Carlo tree search, we chose not to use the new data structure due to time constraints. If the neural network's performance can be increased, the potential gain of a better data structure grows. It might therefore be beneficial to explore these data structures further to improve performance.

Chapter 11

Reflection

In this chapter, we will provide an overall reflection on different aspects of the project. It's important to reflect on the actions taken during the project to determine if there were better ways to manage the project. The aspects we will discuss in this chapter are: planning and the contributions of each team member.

11.1 Planning

At the beginning of the project when we were creating a project proposal for Tom van Dijk we had to come up with a schedule for the project. This schedule as seen from Figure 3.1 was divided into 4 activities, each of them with its own scope and sub-tasks. We believe that we made the right choice of having 4 activities set up like this, because of the division of important activities that have to be done to implement this project.

The first activity that we made for us was Design & Research. In this period we intended to spend this time Researching the different technologies we can use and start the design of the system. This phase went as planned. Research was divided and conducted accordingly and with enough detail to begin implementation in the next phase. The second activity on our list was the implementation step. We divided this activity into 4 separate sprints as the time frame of this step was exceptionally big, about 8 weeks long. The first sprint was one of the most essential sprints because of its crucial goal: delivering a Minimum Viable Product(see Section 3.3.2). We managed to accomplish this goal, determining that we had chosen a reasonable time frame to gather knowledge and start working on the product. The second and third sprints were less strict in regard to deliverables. These sprints consisted of improving the overall performance and architecture of the system. For each of these sprints, we managed to implement new features, as well as enhance the performance of the system altogether by fixing bugs and optimizing existing elements. The last sprint was an emergency sprint, allowing us to finish the project on time even if complications arose during implementation. The sprint proved itself useful and was

used to fix various bugs, improve performance and train and evaluate models, parallel to writing the project report.

We are satisfied with the planning we had done before the implementation phase of the project. It allowed us to accomplish the goals of the project within our time limit. Additionally, thanks to careful planning we were able to stay aware of our progress and adjust the efforts put into different aspects of the project when necessary.

11.2 Contributions

To increase efficiency, we divided responsibilities and tasks in the project. Every member of the team had their own role, which motivated all members to be active in the project. The roles and tasks were divided based on experience, knowledge, and interest. The contributions of all members were as follows:

Thomas van den Berg Testing supervisor in charge of defining a high-level testing strategy; wrote the majority of tests, implemented the Monte Carlo tree search, implemented symmetries, and worked on improving the performance of the algorithm.

Daniel Botnarenco Document supervisor in charge of overseeing project documents; conducted model evaluation and documentation.

Dominik Myśliwiec Team Leader, implemented and improved the neural network(s) architecture as well as the initial version of the Trainer class, created the poster and wrote and gave presentations.

Thom Harbers Research supervisor improving quality of research; implemented training checkpoints, contributed to early documentation.

Caz Saaltink Version control and quality supervisor, greatly improved the performance of the system by implementing parallelism, separate fitting process and several others, contributed to model evaluation, and enforced good version control practices.

Additionally to the responsibilities of their roles (as found in Section 3.5) all team members contributed to the research, as well as training networks to play different games in the last sprint, and testing.

To conclude this chapter, based on the above-mentioned points, we strongly believe that, given the duration of this project and the difficulty of the task, we are satisfied with meeting all the deadlines previously set. As a team, we did not run into major difficulties with communication, and the project work was done in a good atmosphere.

Regardless, some aspects of our planning could be improved based on our experience. We now recognize, that although the Design & Research phase was critical, we should have attempted to finish it sooner, giving us more time for the implementation of the system. We were unaware of how many problems would arise due to the lack of resources on the Deeplearning4J library we used, which created a high risk of delays. That being

said, in the later stages, we realized how much domain-specific research we had missed in the research phase that would have made implementation much simpler. Perhaps it would have been a good decision to dedicate a portion of the time in the implementation phase to conducting additional research.

Apart from that, there are no other major changes we would have made. Considering that all the Must, Should and Could requirements (as seen in Table 3.1) were met, it is safe to say we were able to overcome the difficulties we faced and deliver a satisfactory product to the client.

Bibliography

Apache. (n.d.). Apache mxnet for deep learning. https://github.com/apache/mxnet

Aparapi. (n.d.). Retrieved March 17, 2023, from https://aparapi.com/

Black, A. D., Gibson, A., Kokorin, V., & Patterson, J. (n.d.). Deeplearning4j suite overview. https://deeplearning4j.konduit.ai/

Carraz, M., Stichbury, J., Schuermans, S., Crocker, P., Korakitis, K., & Voskoglou, C. (2019, February). State of the developer nation 16th edition.

Cazenave, T., & Jouandeau, N. (2007). On the parallelization of uct.

Chaslot, G., Winands, M., & Herik, H. (2008). Parallel monte-carlo tree search, 60–71. https://doi.org/10.1007/978-3-540-87608-3_6

Chaslot, G. M. J.-B. C. (2010). *Monte-carlo tree search* (Vol. 24). Maastricht University.

Chollet, F., et al. (2015). *Keras.* https://github.com/fchollet/keras

Clausen, C., Reichhuber, S., Thomsen, I., & Tomforde, S. (2021). Improvements to increase the efficiency of the alphazero algorithm: A case study in the game 'connect 4'. *Proceedings of the 13th International Conference on Agents and Artificial Intelligence.* https://doi.org/10.5220/0010245908030811

Deeplearning4j suite overview. (n.d.). https://deeplearning4j.konduit.ai/

Di Pasquale, D. (2021). Daniel-dipasquale/java-ml. https://github.com/daniel-dipasquale/java-ml

Djl - deep java library. (n.d.). https://djl.ai/

Duvaud, W. (2019). Werner-duvaud/muzero-general: Muzero. https://github.com/werner-duvaud/muzero-general

Enzenberger, M., & Müller, M. (2009). A lock-free multithreaded monte-carlo tree search algorithm, 14–20. https://doi.org/10.1007/978-3-642-12993-3_2

European research council. (2023, April). https://erc.europa.eu/

Evolutionsoftswiss. (2018). Evolutionsoftswiss/alpha-zero-learning: Java based alpha zero reinforcement learning. https://github.com/evolutionsoftswiss/alpha-zero-learning

Foster, D. (2020, July). Alphago zero explained in one diagram. https://medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0

Fumero, J., Papadimitriou, M., Zakkak, F. S., Xekalaki, M., Clarkson, J., & Kotselidis, C. (2019). Dynamic Application Reconfiguration on Heterogeneous Hardware. *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments.* https://doi.org/10.1145/3313808.3313819

Garvey, P. R., & Lansdowne, Z. F. (1998). Risk matrix: An approach for identifying, assessing, and ranking program risks. *Air Force Journal of Logistics*, *22*(1), 18–21.

Gevorkyan, M., Demidova, A., Demidova, T., & Sobolev, A. (2019). Review and comparative analysis of machine learning libraries for machine learning. *Discrete and Continuous Models and Applied Computational Science*, *27*, 305–315. https://doi.org/10.22363/2658-4670-2019-27-4-305-315

Gibson, A. (2021, September). Using multiple backends. Retrieved March 28, 2023, from https://community.konduit.ai/t/using-multiple-backends/1607/2

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., ... Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Higuera, R. P., & Haimes, Y. Y. (1996). *Software risk management.* (tech. rep.). Carnegie-mellon univ pittsburgh pa software engineering Inst.

Hoodat, H., & Rashidi, H. (2009). Classification and analysis of risks in software engineering. *International Journal of Computer and Information Engineering*, *3*(8), 2044–2050.

Java Development Kit Version 17 API Specification — Executors.newWorkStealingPool. (n.d.). Retrieved March 20, 2023, from https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/Executors.html#newWorkStealingPool()

Jcuda. (n.d.). Retrieved March 17, 2023, from http://javagl.de/jcuda.org/

Konduit. (n.d.-a). Nd4j reference. Retrieved April 20, 2023, from https://deeplearning4j.konduit.ai/nd4j/reference

Konduit. (n.d.-b). ParallelInference.java. Retrieved March 28, 2023, from https://github.com/deeplearning4j/deeplearning4j/blob/97b8aeb2dd938d116e8c2f65ec647e9637e7eedf/deeplearning4j/deeplearning4j-scaleout/deeplearning4j-scaleout-parallelwrapper/src/main/java/org/deeplearning4j/parallelism/ParallelInference.java

Konduit. (n.d.-c). Performance Issues. Multithreading. Retrieved March 28, 2023, from https://deeplearning4j.konduit.ai/multi-project/explanation/configuration/backends/performance-issues#step-6-ensure-you-are-not-using-a-single-multilayernetworkcomputationgraph-for-inference-from-multip

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90.

Machine learning at waikato university. (n.d.). https://www.cs.waikato.ac.nz/ml/index.html

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat,

Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Jia, Y., Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, … Xiaoqiang Zheng. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems [Software available from tensorflow.org]. https://www.tensorflow.org/

Mirsoleimani, S. A., Plaat, A., Herik, H., & Vermaseren, J. (2017). An analysis of virtual loss in parallel mcts, 648–652. https://doi.org/10.5220/0006205806480652

Nair, S. (2017). Suragnair/alpha-zero-general. https://github.com/suragnair/alpha-zero-general

Oracle. (n.d.). Java native interface. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

Oren, S. (2001). Market based risk mitigation: Risk management vs. risk avoidance. *Proceedings of a White House OSTP/NSF Workshop on Critical Infrastructure Interdependencies held in Washington DC, June*, 14–15.

O'Shea, K., & Nash, R. (2015). An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems 32* (pp. 8024–8035). Curran Associates, Inc. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Piette, É., Soemers, D. J. N. J., Stephenson, M., Sironi, C. F., Winands, M. H. M., & Browne, C. (2020a). Ludii – the ludemic general game system. In G. D. Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. Bugarín, & J. Lang (Eds.), *Proceedings of the 24th european conference on artificial intelligence (ecai 2020)* (pp. 411–418, Vol. 325). IOS Press.

Piette, É., Soemers, D. J. N. J., Stephenson, M., Sironi, C. F., Winands, M. H. M., & Browne, C. (2020b). Ludii - the ludemic general game system.

Raschka, S., Patterson, J., & Nolet, C. (2020). Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, *11*(4), 193. https://doi.org/10.3390/info11040193

Roper, W., & Richter, F. (2020, March). Infographic: Python remains most popular programming language. https://www.statista.com/chart/21017/most-popular-programming-languages/

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? *Advances in neural information processing systems*, *31*.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., & Silver, D. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, *588*(7839), 604–609. https://doi.org/10.1038/s41586-020-03051-4

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman,

S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. https://doi.org/10.1038/nature16961

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, *362*(6419), 1140–1144.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., & Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. https://doi.org/10.48550/ARXIV.1712.01815

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., & et al. (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354–359. https://doi.org/10.1038/nature24270

Song, J. (2017). Junxiaosong/alphazero_gomoku. https://github.com/junxiaosong/AlphaZero%5C_Gomoku

Team, E. D. D. (2016). ND4J: Fast, Scientific and Numerical Computing for the JVM. https://github.com/eclipse/deeplearning4j

Thielscher, M. (2011). The general game playing description language is universal. *IJCAI International Joint Conference on Artificial Intelligence*, 1107–1112. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-189

Tornadovm. (n.d.). Retrieved March 17, 2023, from https://www.tornadovm.org

Van Cranenburgh, A., Samid, R., & van Someran, M. (2007). Tic-tac-toe.

Vasudevan, R. (2019, November). Introducing deep java library (djl). https://towardsdatascience.com/introducing-deep-java-library-djl-9de98de8c6ca

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., … SciPy 1.0 Contributors. (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, *17*, 261–272. https://doi.org/10.1038/s41592-019-0686-2

Wang, H., Emmerich, M., Preuss, M., & Plaat, A. (2019). Hyper-parameter sweep on alphazero general. *arXiv preprint arXiv:1903.08129*.

Appendix A

Meetings with the Client

A.1 Week 1 (Physical)

On Friday of the first week of the module, we had our first meeting with the client. As the description of the project was quite minimal, we had a lot of questions about what was expected from us. Therefore, we decided to research the topic to get a better understanding of the possibilities and limitations in this existing area. This would allow us to ask specific questions to Tom, which would help us to get a better understanding of the project, and to get a better idea of what we would be working on.

First of all, we discussed meeting times and the way we would communicate with Tom. We decided to have a meeting with our client every week to discuss our progress and to ask questions about the project. For each meeting, we would send an agenda beforehand, containing the topics we wanted to discuss. He could then start thinking about the answers to our questions and could prepare for the meeting if needed. Most of these meetings were held on Microsoft Teams, as traveling made it difficult to meet in person. However, we did have a few meetings in person, as we were able to meet at the university because of some other things we had to attend nearby. Tom did mention that he would be on vacation near the end of the project, so we had to make sure that we had the most important things done before that time. This way we could still have feedback from him before he left, and make sure the project was still on track.

Secondly, we discussed the project itself. From the research we had done, we found different directions the project could go in. We were not sure what kind of game input we would be working with. One example we found was Ludii (Piette et al., 2020a), which is a general game system containing traditional games. They created so-called ludemes, which are high-level game descriptions that can be used to define any game. More on this can be found in Section 2.0.4. However, Tom mentioned we would be working with a game interface he had created, rather than using a game description language. This

interface contained a few methods that we could use to get information about the game, such as the current state of the board, the possible moves, and the current player.

We then talked about what we would do with the extracted game information. We were wondering if the AI would be live-timed, or be trained beforehand. Tom explained that the idea was to make an evaluation function using deep learning together with Monte Carlo tree search (MCTS). He suggested doing more research on AlphaZero, AlphaGo, and MuZero (Foster, 2020; Schrittwieser et al., 2020; Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, et al., 2017), and looking into the differences in efficiency.

Additionally, we were interested in what the project would be used for. Would it be used for benchmarking, a reference for Module 2 students, or something else? Tom mentioned that it was mostly for his pleasure and fun.

Now that we had a better understanding of the project and the direction it should go into, we could start thinking about what we would need to do to complete the project. We agreed to present a project proposal at the next meeting containing requirements, where we could discuss and refine them if needed.

A.2 Week 2 (Teams)

The main topic of this meeting was discussing the project proposal. We had finished our initial proposal and wanted to see if we were on the right track.

During the meeting, Tom went over the document and gave us feedback on things he spotted that could be improved, without going through the entire file. He mentioned that our current planning could be a lot more specific. The current Gantt chart was vague, as it was lacking details about requirements and deliverables. Continuing on this, he suggested that we should add a section about the deliverables, containing a brief and clear overview of what would be delivered and when. This would also allow us to reference each deliverable when talking about them. Tom also wanted to see a part about existing projects and repositories, describing what they are, and talking about what we are making and why it is different Lastly, we asked what was meant by a section about procedures, as this was not clearly described in the rubric. He explained that it would be a section that contains information about how we would work together, how meetings are approached, and what to do when someone is late or doesn't show up.

He also suggested some changes to the requirements. Initially, we had a requirement that the AI should be able to beat other AIs. Instead of failing the requirement if the AI lost, Tom mentioned it would be more interesting to see why it lost, suggesting that we should remove this requirement. There was also a requirement stating that the AI would make a move within 5 seconds, as Tom mentioned something similar during our first meeting. Though, a better approach would be to make the number of seconds a parameter so that this could easily be changed for each game.

There were also some other small remarks or concerns regarding the proposal. We inquired Tom's opinion on the usage of "we", and he responded by saying he was fine with this and motivated us to write in active voice. He also pointed out that we should try to avoid using bullet point lists without any details, as this would make the document look unfinished.

Besides the project proposal, we also discussed the project itself. During the first meeting, Tom mentioned a couple of different board games that were used during Module 2. We forgot which ones exactly, so he refreshed our memory. The games that he had implemented using the Game interface were tic-tac-toe, Othello, Collecto, and Pentago. He proposed he would send us files containing an implementation of the games so that we could use these for training our models.

We also discussed ratings and how we would evaluate the AI. We were curious about how he implemented the ratings for the games in Module 2. Tom explained that he simply used an ELO calculation mentioned on Wikipedia. We asked if we should use this for our project, and he said that we could, but that he would also be fine with using the number of wins.

A.3 Week 3 (Teams)

The agenda points for this meeting mostly revolved around the project proposal. The previous week Tom had given us a lot of useful feedback, which we tried to incorporate into the proposal. Therefore, we mentioned inside our agenda which parts of the proposal had been updated since last time.

For the requirements section, we seemed to have missed an `\hline` which caused our table to look bad. On top of that, it was also too small, so the table should be bigger in the final proposal. Tom suggested we should also add a requirement that our AIs must be able to be trained on a GPU using CUDA.

Additionally, Tom had some other ideas about sections he would want to see added or improved. He suggested that we should mention that we are using GitLab issues, and how we plan to assign them. On top of that, he wanted to see our related works get expanded by explaining how we could use these existing projects to our advantage, as well as adding consideration of frameworks to this section, and talking about different deep-learning frameworks.

Lastly, some visual and minor improvements could be made to the proposal. For example, references should have links for the entire name and not just the year. Other improvements were using `enumitem` to make the lists look better, capitalizing the first letter when referring to tables, and adding colors to the risk assessment table to make it easier to read. There were also some small grammar mistakes that we should fix and some abbreviations that we should expand to make the document less ambiguous.

A.4 Week 4 (Physical)

Week 4 was the final week of sprint 1, where at the end we should have a working prototype of the AI. For this meeting, we planned to discuss the progress we had made so far for the minimum viable product (MVP), as well as show our finalized project proposal.

For the project proposal, we again incorporated the feedback we had received from Tom during the previous meeting. This time Tom had very few remarks, as he was satisfied with the progress we had made. Though, he did mention that besides DL4J, we should still talk about other frameworks we could use.

In terms of the minimal viable product, we created an initial class diagram, which showed our initial idea of how the project would be structured. But at the time we presented it, changes in the structure had already been made while working on the project, so this needed to be updated. Still, it was a good starting point to illustrate our ideas.

The last point on the agenda was to ask for some Java classes that were needed for the project, as they were missing from the previous files Tom had sent us.

A.5 Week 5 (Online)

At this point in the project, we had finished our MVP and started working on the next sprint. The purpose of the second sprint was to implement the theory and principles from DeepMind's AlphaZero paper into our project.

As we were still early in Sprint 2, the meeting was mostly about discussing and showing our MVP. Tom was happy with the progress we had made and asked some questions about how we had implemented the different parts of the project.

We also mentioned that all of us tried to make CUDA work, as it can be finicky to get working. Tom rightfully pointed out that we should not spend too much time on this, and try to see if one or two of us could get it working, and then help others.

Lastly, he was also interested in seeing how it would do against a minimax AI, using a neural network as an evaluation function.

A.6 Week 6 (Online)

Week 6 was the final week of sprint 2, where we should have a working AlphaZero-like implementation. To get to a working implementation, we had to change some things in our project.

First of all, we changed the architecture of the neural network. We added convolutional layers to the network, allowing it to learn more complex patterns.

We also already tried to increase the speed of the training process. To do this, we mainly focused on trying to parallelize the Monte Carlo tree search. As the algorithm spends a lot of time in the tree search, we thought that this would be a good place to start.

Lastly, when training the network on Othello, we noticed that we were getting weird results. It seemed that the `OthelloLongState.getWinner` function was returning the opposite of what it should. Here, Tom gave us some advice on what could be causing this, and how we could fix it.

A.7 Week 7 (Online)

Week 7 was the start of sprint 3, where our main goal was to improve the performance and speed of the AI.

There was not much to discuss during this meeting, as training the network was taking a lot of time and major changes were not made to the project. We did however have a couple of questions for Tom.

We had to ask for the `positions.txt` file for Pentago, as we had not received it yet. Besides that, we also asked if Tom knew about any presentation plans for the FMT board, as we had heard from Rom that this may have been arranged already. He said that he would ask around and get back to us.

A.8 Week 8

Week 8 did not have a meeting, due to Easter holidays.

A.9 Week 9

Week 9 did not have a meeting, due to Tom's vacation.

A.10 Week 10

Week 10 was the last week of the project, where we were finalizing our project and preparing for the presentation. As we had not spoken with Tom for a while, we started the meeting by bringing him up to speed on the progress we had made.

After this, we asked him a couple of questions. Firstly, we were curious about the style of presentation that we should use. Tom explained that it would be a relatively short presentation of about 15 minutes and that we should highlight the most important parts of the project. We went through the presentation rubric together to identify what we should focus on.

We then showed him the poster we had made and asked for his feedback. He said that it looked good, but that we should probably add our names to the poster.

Lastly, we showed our presentation slides, where Tom thought we should cut back on the number of slides. Another thing he rightfully mentioned was that we should never end on a thank-you slide, but rather show our conclusions.

Appendix B

Sprint reports

B.1 Sprint 1 (February 20 - March 12)

The project proposal was made, together with researching the domain and establishing the requirements. Subsequently, the system specification was made, together with the system design.

The first version of the system was implemented.

B.2 Sprint 2 (March 13 - March 26)

The system was tested and evaluated.

The system was improved by restructuring the network architecture.

Certain aspects of the system were made immutable.

B.3 Sprint 3 (March 27 - April 9)

To speed up the system, self-play and tournaments were parallelized.

Missing new methods for feature extraction for Pentago and Collecto were implemented, to allow for more games to be trained.

A lot of training was done, to try to improve, optimize and evaluate the system.

A separate fitting process using the CUDA backend was added.

B.4 Sprint 4 (April 10 - April 21)

A new version of the network has been made, with more scalability and configurability.

Parallel Inference was added.

More training has been done to further improve the system.

The code has been cleaned up and refactored where needed, as well as documentation was improved.

For the reflection component, the ethics report about the project was made.

The last sections of the report were written, after which the whole report was checked.

The poster for the presentation was made, and the final presentation has been prepared.

Appendix C

Class Diagrams

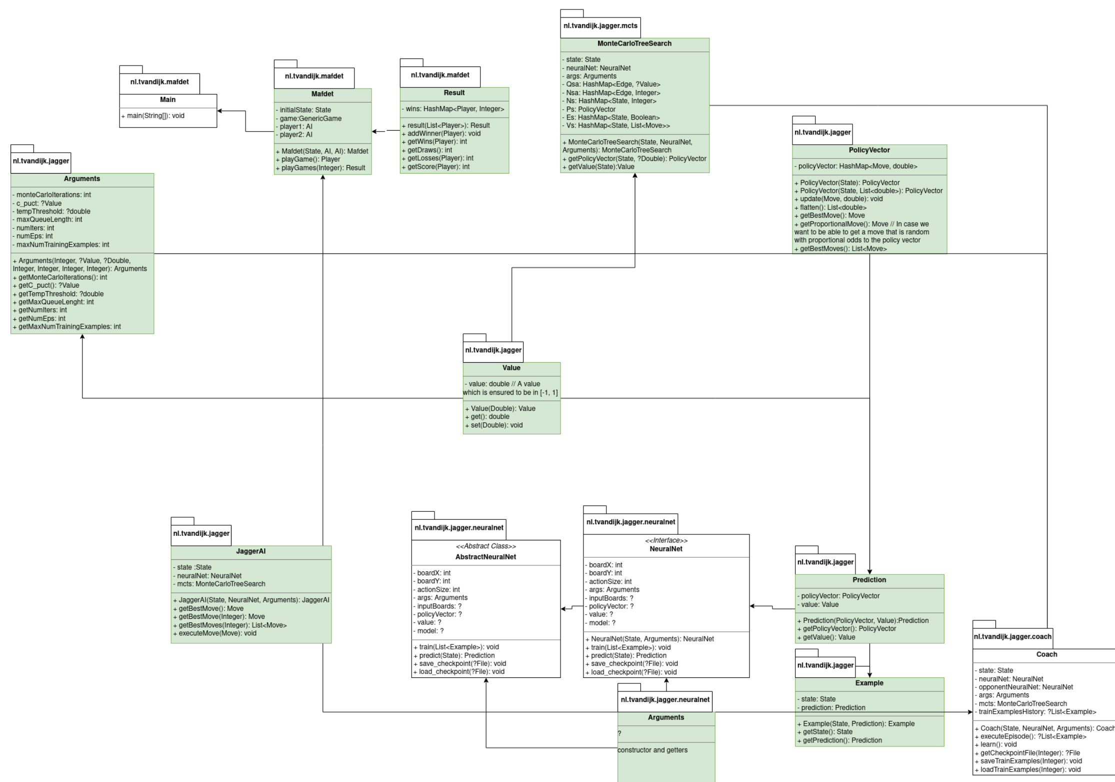

Figure C.1: Class diagram of initial design

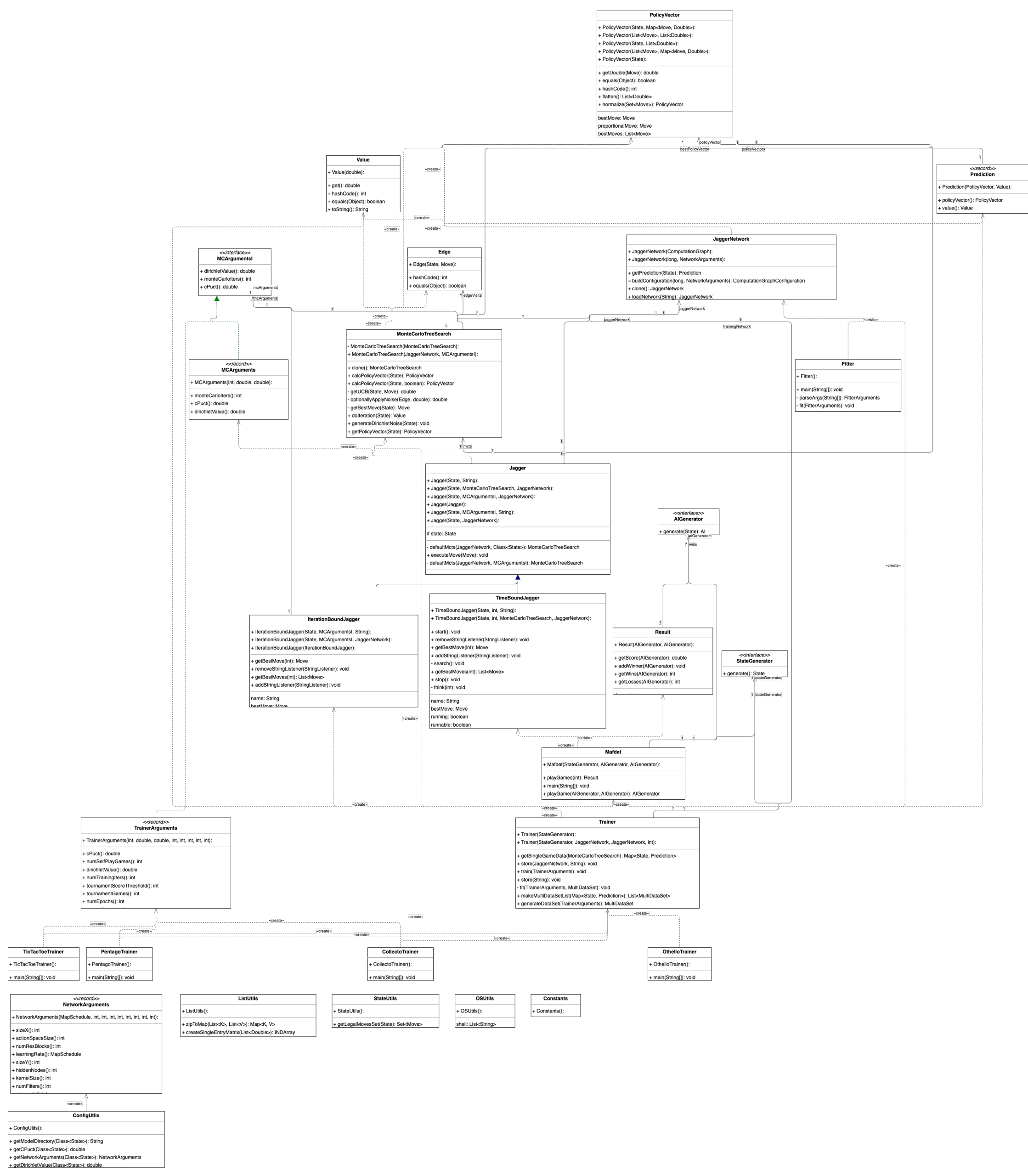

Figure C.2: Class diagram of final implementation