HAMK
Hämeen ammatti-
korkeakoulu

Artificial Intelligence Solution for Gomoku Using Neural Networks

Bachelor's thesis
Information and Communication Technology
Spring 2024
Mikko Moilanen

HAMK
Hämeen ammatti-
korkeakoulu

Tieto- ja viestintätekniikka Tiivistelmä
Tekijä Mikko Moilanen Vuosi 2024
Työn nimi Artificial Intelligence Solution for Gomoku Using Neural Networks
Ohjaaja Petri Kuittinen

Tämän työn tarkoituksena oli luoda neuroverkoilla toimiva tekoälyratkaisu viiden suora -pelille. Tekoäly hyödynsi vahvistusoppimista, missä tekoälyä palkittiin hyvästä suorituksesta ja tekoäly pyrki maksimoimaan palkkion, mikä puolestaan johti toivottuihin oppimistuloksiin.

Opinnäytetyössä käytettiin ohjelmointikielenä Pythonia ja tekoälykirjastoksi valikoitui PyTorch. Tekoälyn neuroverkkorakennetta testattiin useilla eri kokoonpanoilla ennen kuin sopiva malli löytyi. Neuroverkon koko oli suppea, mutta se oli erittäin suorituskykyinen. Tekoälyä koulutettiin yhteensä 3000 kertaa siten, että se pelasi toista tätä työtä varten luotua tekoälyä vastaan. Tämä TestAI:ksi nimetty tekoäly ei hyödyntänyt neuroverkkoja vaan algoritmipohjaista ratkaisua ja pääasiallinen syy tämän tekoälyn käytölle oli se, että sen avulla oli mahdollista suorittaa suuria määriä testauksia lyhyessä ajassa. Ihmispelaajan kanssa kouluttaessa tuhansien pelikertojen pelaaminen useaan kertaan olisi vaatinut kohtuuttoman suuren aikapanostuksen. Tämän jälkeen tekoälyä koulutettiin vielä 1000 kierrosta siten, että se pelasi itse itseään vastaan.

Tekoälyn suorituskyky oli hyvä. Tekoäly pystyi yli 75 prosentin todennäköisyydellä ennustamaan oikean siirron ja voitti TestAI:n yli 80 prosenttia kerroista koulutuksen jälkeen. Tekoälyn suorituskykyä testattiin TestAI:ta vastaan, samoin kuin itseään vastaan ja myös ihmispelaajaa vastaan.

Työn tavoitteena oli luoda tekoäly, joka vastaa suorituskyvyllisesti arviolta hyvää perustasoa olevaa ihmispelaajaa ja suorituskykymittausten perusteella työ saavutti tämän tavoitteen.

Avainsanat Tekoäly, neuroverkot, vahvistusoppiminen, abstraktit pelit, Gomoku.
Sivut 40 sivua ja liitteitä 1 sivua

HAMK
Hämeen ammatti-
korkeakoulu

Information and Communication Technology Abstract
Author Mikko Moilanen Year 2024
Subject Artificial Intelligence Solution for Gomoku Using Neural Networks
Supervisor Petri Kuittinen

The purpose of this work was to create an artificial intelligence solution for the game of Gomoku using neural networks. The AI utilized reinforcement learning, where the AI was rewarded for good performance and aimed to maximize its rewards, leading to desirable learning outcomes.

The thesis used Python as the programming language and PyTorch was selected as the machine learning library. The AI's neural network structure was tested with several different configurations before a suitable model was found. The neural network was compact, but it was highly efficient. The AI was trained a total of 3000 times, playing against another AI created for this work. This AI, named TestAI, did not utilize neural networks but an algorithmic solution, and the primary reason for using this AI was that it allowed training on a large scale to be conducted over a short period of time. The AI was further trained for 1000 rounds by playing against itself.

The AI's performance was good. The AI was able to predict the correct move with a probability of over 75 percent and win the TestAI in over 80 percent of the games after training. The AI's performance was tested against the TestAI, as well as against itself and also a human player.

The goal of the work was to create an AI that matches the level of a good average human player in terms of performance, and based on the performance measurements, this goal was achieved.

Keywords Artificial intelligence, neural networks, reinforcement learning, abstract games, Gomoku
Pages 40 pages and appendices 1 page

Table of Contents

Figures

Appendices

1 Introduction

In recent years, the concept of artificial intelligence (I will refer to artificial intelligence in this text as AI) has emerged as a new megatrend, offering solutions to complex problems and new ways to optimize human labour, while at the same time causing concerns about what implications artificial intelligence has for the future of humanity. Regardless of these concerns, it is rather safe to say that artificial intelligence is here to stay, and its outreach will only grow over time. The field of AI is rapidly evolving and thus content also becomes outdated quickly, which is relevant to keep in mind for any future readers of this paper.

The field of artificial intelligence development is rapidly gaining momentum as more powerful solutions become available and as the processing power of computers is rapidly growing. Moore's Law, named after entrepreneur Gordon Moore, states that the number of transistors per silicon chip doubles every two years (Britannica, 2024). Similarly, OpenAI, the company behind one of the most infamous Large Language Model (LLM) ChatGPT, has stated that the performance of AI doubles every 16 months (OpenAI, 2020). The most recent developments in LLM artificial intelligence have come close to passing the Turing Test, a classical test to measure whether a machine could trick a human into believing that they are conversing with a human (Goldstein & Kirk-Giannini, 2023). In our discussions with my thesis supervisor Petri Kuittinen, he offered his insights on the latest AI development by saying that AI development is developing at an exponential rate, although no scientific consensus has been reached on the speed of development. It is only a matter of time before Turing's Test is passed, but LLMs are not the only field of AI development in recent years.

Google's DeepMind team has developed an AI that is capable of playing StarCraft 2, a famous real-time strategy game, by using generic machine learning techniques, such as neural networks, reinforcement learning as well as learning directly from data. I will go through different machine learning techniques later on in this thesis, but according to the DeepMind team's research paper published in 2019, the AlphaStar AI managed to reach a Grandmaster level and was ranked above 99.8 percent of the active players ladder. (The AlphaStar Team, 2019) This is a prime example of the potential a neural network development can have on the video games industry.

The original idea for this thesis was to build an AI for a snake game that could beat the game with a perfect score, meaning that the snake would be able to fill the entire screen and then, eating the last fruit, grow itself larger than the game area. However, such solutions already

exist, and there is even at least one GitHub page containing the source code for it (Chrispresso, 2019). Therefore, another game needed to be selected.

After some consultation with the thesis supervisor, I set out to research three potential games on which an AI solution could be built: Space Invaders, Checkers, and Gomoku. Each of these games already had pre-existing artificial intelligence solutions available, but none were “perfect” in the same sense as was the case with the Snake AI, meaning that it could perform an unbeatable game round. This somewhat stems from the fact that these games are different. Snake is a single-player game that has a theoretical ending that is always possible to achieve, whereas Gomoku and Checkers are multiplayer games where players compete against each other, trying to reach a clearly defined victory condition. Space Invaders is a single-player game that has an infinite game loop, at least in theory, so there is no end condition that a player could reach should they run a perfect game.

Original Space Invaders would need to be emulated, whereas Checkers and Gomoku engines could be built so that the artificial intelligence would have direct access to the input and output data of the game itself. Based on my research, I concluded that the Checkers had the most advanced artificial intelligence solution already built out of these three games. Therefore, I concluded that Gomoku and Space Invaders were the most suitable games for this thesis. Since I was more familiar with Gomoku, I finally chose that as the game for this thesis.

There has been a wide range of artificial intelligence solutions developed for Gomoku over the years. This thesis aims to build an artificial intelligence for Gomoku using neural networks and reinforcement learning, resulting in a solution that is capable of playing the game at a level that matches the level of a semi-competent amateur player. The performance is measured with a custom-built TestAI and a human player, myself. Out of the scope of this thesis is testing the AI against other Gomoku AIs, other than the TestAI specifically created for this project. The project is also not intended to be compiled into an executable or to be published as a stand-alone product. The source code will be publicly available on GitHub, the link to the GitHub page can be found at the end of this thesis in Appendix 1. Also, the user interface is built only to a functional level without any extra visual elements. The solution will consider only errors that are directly necessary for the performance of the solution. This means for example an out-of-bounds error when trying to access a grid value outside of the board, as well as other types of value errors and runtime errors. The solution does not take into account for example a situation where the saved model file would be tampered with.

I will start this thesis by going through the theoretical framework that helps to understand the core concepts used in this thesis. After that, I will describe how the project was implemented, and finally, I present the results and reflect on how the model performed.

2 Theory

2.1 A Brief Introduction to Gomoku

Let us start by defining what is the game in question. Gomoku, also known as Five in a Row, is a classic board game that originated in Japan and is played with black and white stones typically on a 15x15 or 19x19 board. The goal of the game is to create a row of five stones horizontally, vertically, or diagonally. The first player plays with black pieces, and the second player with white pieces.

According to gomokuworld.com, the name Gomoku comes from the Japanese language, in which it is referred to as “gomokunarabe”. “Go” means five, “moku” is a word for pieces, and “narabe” means line-up. The earliest published book on Gomoku dates back to the late Edo period, around 1850. Gomoku is considered to be related to a game called Gobang. (gomokuworld.com, n.d.)

The player who plays the first move is typically considered to have an advantage over the second player in Gomoku. Therefore, in professional Gomoku games, there are restrictions on the first player's moves to balance out the advantage of the first player. The implementation for this project will not take into account these professional Gomoku rules. Furthermore, in a professional Gomoku ruleset, a situation where there are more than five stones in a row is not considered to be a winning situation, but an “overdraw”. (gomokuworld.com, n.d.) In the context of this thesis, this rule is also not taken into account. Training the neural network AI to take these complex rules into account would be unnecessarily difficult. Since the AI is not intended for professional use, a simpler version without professional ruleset consideration is sufficient.

2.2 Artificial Intelligence and Machine Learning

To begin the discovery of artificial intelligence and machine learning, it is important to first define what is meant by these concepts. According to Copeland, who has written an article “Artificial intelligence” on Britannica in 2024, artificial intelligence is the ability of a computer

or a machine to perform tasks that are commonly associated with intelligent beings. The Council of Europe 2020 publication “History of Artificial Intelligence” dates the Artificial Intelligence discipline to be some sixty years old, but it has experienced a boom in interest since 2010. Both articles conclude that artificial intelligence is still far away from reaching a level of human intelligence or processing power, although computers are highly capable of performing certain problem-solving tasks at inhumane speeds. (Copeland, 2024; Council of Europe, 2020)

Copeland defines intelligence to be a combination of different traits: learning, reasoning, problem-solving, perception, and using language. He outlines two different approaches in AI research: symbolic and connectionist. A symbolic approach to AI aims to replicate intelligence independent of how a human brain would function, whereas a connectionist solution is to mimic the neural networks that exist in the human brain to simulate intelligence. He describes the difference between these two approaches with an example of building a system that recognizes alphabetic letters. A systemic approach would be to build a system that compares the shapes and geometric descriptions of letters to categorize them, whereas a connectionist approach would be to present individual letters for a neural network and gradually improve its performance by tuning it to recognize different letters. (Copeland, 2024) Therefore, it is to be concluded that the artificial intelligence solution for this thesis is connectionist by nature.

Machine learning is a field of artificial intelligence, first defined in the 1950s by Arthur Samuel, according to Sara Brown in her article “Machine Learning, explained”, published in 2021 on the MIT Sloan School of Management website. Machine learning can be utilized to perform tasks with computers that traditional computer applications would not be able to do, for example, image recognition. She describes that machine learning starts with data that is gathered and categorized for the training of the machine learning model. More data leads to a better program. The programmer chooses a machine learning model that feeds the data and lets the computer train itself. Sometimes tweaking the model might be in order. The trained model is then tested with a validation data set, which is usually a partition of the original data that was excluded from the training set to test out how the AI performs with a completely new dataset after the training has finished. (Brown, 2021)

Brown identifies three use cases for machine learning programs. They can be descriptive, predictive, or prescriptive by nature. A descriptive model tries to describe based on the data what has happened. A predictive model uses data to predict what will happen. A prescriptive model uses data to make suggestions about what action to take. She further describes three

subsets of machine learning: supervised, unsupervised, and reinforcement learning. Supervised learning means labeled data sets, which makes the model grow more accurate over time. Labeling is done by humans and over time the machine would learn to distinguish between the labeled categories on its own. Unsupervised learning learns from unlabelled data. It tries to find patterns that were not explicitly told to look for. Finally, reinforcement learning is a model of learning where the machine goes through trial and error to find the best action. Reinforcement requires a reward system. (Brown, 2021) Reinforcement learning is the model that is most well-suited for video game artificial intelligence, therefore it is the chosen learning method for this thesis as well. Reinforcement learning and its suitability as a video game artificial intelligence solution is explained in more detail in Chapter 2.4.

Sambit Mahapatra defines machine learning in his article “Why Deep Learning over Traditional Machine Learning?” as a set of algorithms that parse data and learn from them to make intelligent decisions. He then defines deep learning as a subset of machine learning, where the algorithm can learn high-level features from data incrementally. According to him, traditional machine learning algorithms require that the input data is processed so that the patterns become more easily visible for the algorithm to learn from, whereas deep learning can learn from data that is unstructured. He also describes how deep learning algorithms are more efficient in problem-solving as they can solve problems end-to-end, whereas traditional machine learning algorithms need to do them one by one. Deep learning algorithms take a long time to train due to their complexity and require significant hardware to do so, whereas machine learning algorithms are faster to train. But Mahapatra explains that when testing the algorithms, deep learning performs far better than most of the machine learning algorithms. He concludes that deep learning is useful when processing large amounts of data and when it is utilized for complex problems, such as image classification and speech recognition. (Mahapatra, 2018)

2.3 Neural Networks

To understand the type of artificial intelligence being built for this thesis, it is vital to gain knowledge of what neural networks are and how they function. According to Ronald T. Kneusel in his book “Practical Deep Learning: A Python-based Introduction”, the neural network is a graph that represents a series of computational steps that connect an input feature to an output feature via possible hidden layers. The connection is represented using lines that are called weights. The output feature is typically a probability. The map is read from left to right and a single individual node is called a neuron. Each neuron receives an

input from the previous layer, calculates a new output based on the inputs, and passes on the output value to the next layer until an output layer has been reached. Each neuron can have multiple inputs, and in a complex neural network, there can be hundreds of inputs to each single neuron. (Kneusel, 2021, pp. 170 - 171)

A neural network is called a fully connected feed-forward neural network if all neurons of the previous layer are connected as inputs and all neurons from the current layer are connected as outputs to the next layer, the flow is from left to right and there is no feedback loop (Kneusel, 2021, pp. 170 - 171). This type of neural network is one-dimensional. A convolutional neural network on the other hand is multi-dimensional. It shares much of the same attributes as the fully connected neural network. For example, convolutional networks are usually also feed-forward. I will go through both of these network types to gain a better understanding of how neural networks function. I will be using Kneusel's book as the main source for neural networks as he does the topic justice by describing it clearly and concisely, although the topic is fairly complex. This is a clear indication of his deep understanding of neural networks.

2.3.1 Fully Connected Feed-Forward Neural Network

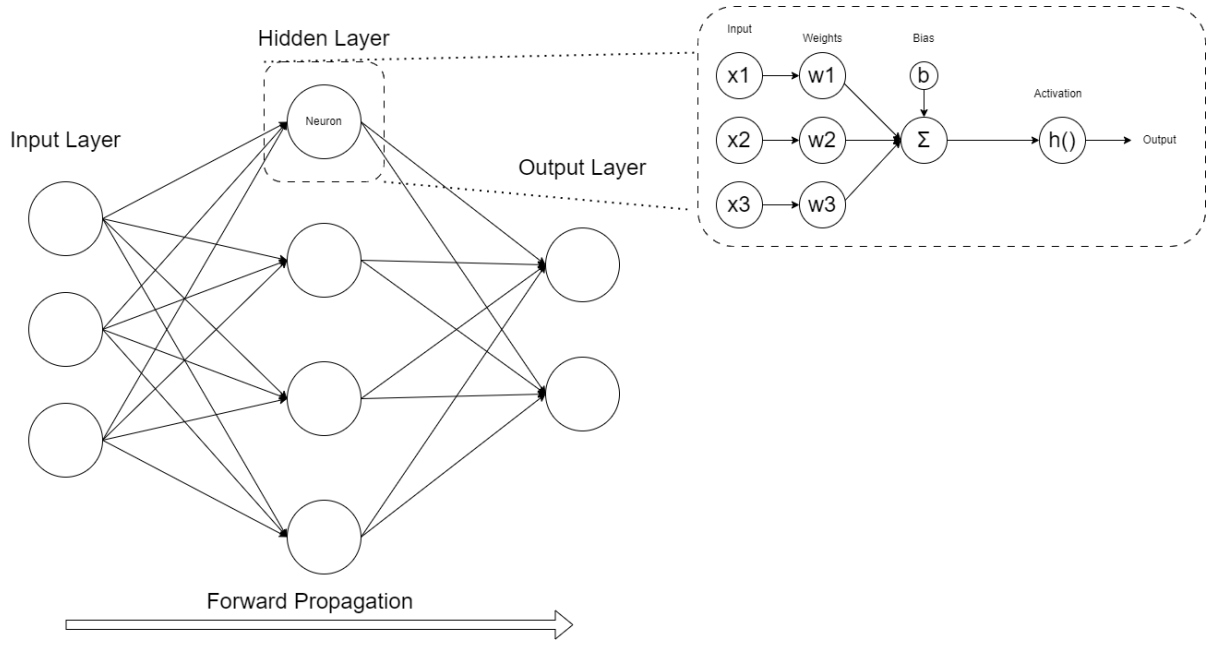
A node in a neural network consists of inputs, weights connecting the inputs to the neuron, and an output. According to Kneusel, this is called an activation function, which calculates the output of the node, which is a single number. The inputs, represented by x_n, where n is the index number of the input, are multiplied by their connecting weights, represented by w_n, with the same indexing applied to them as the input they are connecting, after which all of the inputs are summed and then fed to the activation function of the node, which is represented with *h*. The output is called *a*, and finally, there is the bias value, represented by a letter *b*. Bias is an offset value that is used to adjust the input range to make it more suitable for the activation function. With these values, Kneusel presents a formula that is used to produce a scalar output value:

$$\mathrm{a} = \mathrm{h}(\sum_{i=0}^{n} w_i x_i + \mathrm{b})$$

Simplified, this is how to build a neural network according to Kneusel: add together a bunch of nodes and link them appropriately. He points out that in reality there is more to it and that creating a well-functioning neural network is a complicated task, but once it has been properly created and trained, its use is rather straightforward. (Kneusel, 2021, pp. 171 – 172)

Below is an image of an illustration of a neural network and the structure of a single neuron to further illustrate how the network is constructed and how different parts of the neural network link to each other.

Figure 1. An illustration of a neural network structure.



The activation function needs to be non-linear. This means that the input(s) of the function must not be proportional to the output of the function. Kneusel points out that functions, when plotted in graphs and producing non-straight lines, are non-linear by nature. These include trigonometric functions, exponential functions, and so on. If the network has a linear activation function, it can learn only linear mappings, which usually are insufficient for neural networks. According to Kneusel, traditional neural networks use either sigmoid or hyperbolic tangents as non-linear activation functions. Both of these functions produce an “S” shape when plotted on a graph with a different range: the sigmoid function runs between 0 and 1, while a hyperbolic tangent runs between -1 and 1.

The sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The hyperbolic tangent function:

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Kneusel mentions that more recently these two functions have been replaced with a rectified linear unit, or ReLU for short.

The ReLU function:

$$ReLU(x) = \max(0, x) = \begin{cases} 0, if\ x < 0 \\ x, otherwise \end{cases}$$

Regardless of its name, ReLU is not a linear function, rather it is piecewise: it removes negative values and replaces them with 0. It is computationally faster to calculate than sigmoid and hyperbolic tangent which is beneficial, especially in huge neural networks with thousands of nodes. (Kneusel, 2021, pp. 172 – 176)

While neural networks are usually built in layers, it is not strictly necessary according to Kneusel, but computationally smart and helpful training-wise. In a fully connected network with hidden layers, the outputs of the previous layer link to all nodes in the next layer, hence the name fully connected. Kneusel points out that this too is not strictly necessary but simplifies the implementation. A neural network with a single hidden layer in between input and output layers is capable of learning any function mapping, however, this might not be the most practical solution in every case. As the number of nodes grows, so does the number of biases and weights, and this in turn requires the training data to grow as well. The neural network architecture has a huge impact on the learning capabilities of the model. While the exact parameters for an ideal network architecture vary case by case, Kneusel gives a few rules of thumb to follow when designing a neural network architecture:

1. If the input has definite spatial relationships (such as image recognition), consider using convolutional neural networks.
2. Use as few hidden layers as necessary, but no more than three.
3. The number of nodes in the first hidden layer should be greater than or equal to the number of inputs.
4. The number of nodes in the following layers should be a value between the number of nodes in the previous layer and the number of nodes in the next layer.

The output layer is usually either a single node or multiple nodes and it outputs a decision value. If the output value is binary, the final node is sigmoid where the value is then rounded to either 0 or 1 based on the threshold value that is applied to it. On the other hand, if there is more than one output node, then a Softmax operation is applied to these outputs and the output with the largest Softmax value is selected. (Kneusel, 2021, pp. 176 – 180)

The Softmax function:

$$p_i = \frac{e^{a_i}}{\sum_j e^a j}$$

Kneusel points out that if the *a* value in the Softmax function is large, then the e^a value will inevitably be very large, which is not computationally wise as it deteriorates precision. If all the other *a* values are subtracted from the largest *a* value before calculating the Softmax, it should reduce the risk of having a large exponent without affecting the end result p_i values. (Kneusel, 2021, pp. 176 – 180)

2.3.2 Convolutional Neural Network

According to Kneusel, convolutional neural networks have several advantages over traditional neural networks. It requires fewer parameters due to convolution, which applies parameters in each layer to small subsets of the input rather than to the entire input. Convolutional network is also able to detect spatial relationships in input irrespective of where they appear. Kneusel describes how this is useful, especially in image recognition tasks. A convolutional network can learn that a sub-section of an image can appear at any point in the input canvas, whereas a traditional network would learn that a sub-section may only appear where it has been present in training content. This presents a clear advantage in multi-dimensional inputs such as images, which are two-dimensional, as the input is not flattened into one dimension. However, convolutional neural networks can also be used in one-dimensional data, according to Kneusel, although he concludes that it might not be an ideal network to use in such a case. (Kneusel, 2021, pp. 283 – 284)

Convolutional neural networks use small kernels to scan an input. Kneusel describes the kernel as the thing we ask the convolutional layer to learn during the training. In the context of two-dimensional input, it's a small array that is moved over the input and slices a smaller sub-array equal to the size of the kernel. Then the network applies a matrix math operation on the sub-array to produce an output number. The operation is a multiplication of the kernel with the sub-array, after which the values are added together, producing the final output value. The output array might be smaller depending on the kernel size. This is called the exact or valid approach according to Kneusel, as it only outputs data that is exactly outputted by the operation. There are ways to retain the input size, such as zero-padding, where the kernel considers values outside of the input array to be 0 and then performs the same matrix operation as previously mentioned. The kernel moves through the input array in what Kneusel calls strides, which means the amount that the kernel moves in the array between

each scan. A stride is most commonly a value of one, meaning that the kernel moves one unit further each time. (Kneusel, 2021, pp. 285 – 287)

Kneusel describes that, unlike a traditional neural network, a convolutional neural network has various different layers: convolutional, ReLU, pooling, dropout, flatten, and dense. Kneusel uses Keras names for these layers. Keras is a Python library which is described in more detail in chapter 2.6. A convolutional layer can be considered as a stack of multi-dimensional arrays. Convolutional layers have filters, which are a stack of kernels, that are applied on top of the inputs and then used to process an output. The functionalities of a ReLU-layer have been described above in the previous chapter. The pooling layer reduces the number of dimensions based on a defined rule. Kneusel mentions that the most common rule is max, which means that the highest input value is outputted, and the rest is discarded. A dropout layer randomly selects a portion of the outputs and sets them to zero. According to Kneusel, this helps the network to learn meaningful representations of the data. The flatten and dense layers work in tandem. Flatten layer turns an input into a vector and passes it to a dense layer, which implements a traditional fully connected neural network layer. The output is, according to Kneusel, usually a four-dimensional array and it is passed to another dense layer or a softmax output layer. In addition to these layers, Kneusel points out that it is possible to create your own layers as well. He also mentions that Keras supports even more layers than defined here. (Kneusel, 2021, pp. 289 – 293)

2.3.3 Optimizer Functions

Optimizer functions are crucial to machine learning. According to Maciej Balawejder, who writes in their article “Optimizers in Machine Learning”, the optimizer tries to find the optimal parameters that will reduce the amount of error as much as possible. There are different types of optimizers and it’s not feasible to go through all of them within the scope of this thesis, but I will be introducing the ones that I will be experimenting with: Stochastic Gradient Descent (SGD) and Adaptive Moment Estimation, known also as Adam. Before diving into these two algorithms, I will briefly introduce the Gradient Descent function, on which both of these algorithms are based on. The original Gradient Descent function is as follows:

$$\theta = \theta - \alpha \times g_t$$

Where θ is the model parameters, α is the learning rate, and g_t is the gradient of the cost function. In practice, this means that gradient descent is processing a function to find its lowest point, starting from a random point somewhere on the function, according to

Aishwarya V Srinivasan in her article “Stochastic Gradient Descent — Clearly Explained !!”. Both she and Balawejder point out however that standard Gradient Descent is slow when datasets become large due to the large amounts of calculations they need to perform to find the lowest point. Therefore, stochastic methods are more suitable for optimization for large or complex datasets. Stochastic means “random” in common terms, according to Srinivasan. (Balawejder, 2022; Srinivasan, 2019)

Stochastic Gradient Sample, SGD, calculates a random sample of points in a function to find the lowest point. This has an immense impact on computation times. According to Balawejder, this has the benefit of being computationally simple, but at the same time, it is more dependent on the learning rate. He also notes that often in machine learning the functions don’t have a single low point but are non-convex with multiple local minimum values. In these scenarios, SGD might incorrectly discover a local minimum value, which would still not be the global minimum value. He notes, however, that nowadays SGD refers to a Mini-Batch Gradient Descent, which uses data batches instead of data points to optimize the model. This utilizes the accuracy of a standard Gradient Descent, while retaining the speed of SGD, according to Balawejder and Srinivasan. (Balawejder, 2022; Srinivasan, 2019)

The Adam optimizer, as described in the paper "Adam: A Method for Stochastic Optimization" by Diederik P. Kingma and Jimmy Lei Ba, is an algorithm for the first-order gradient-based optimization of stochastic objective functions. It leverages adaptive estimates of lower-order moments to achieve efficient optimization, particularly suited for large-scale problems in terms of data or parameters. Adam is considered to be simple and has low memory requirements. The algorithm computes individual adaptive learning rates for different parameters from estimates of the first and second moments of the gradients, hence the name adaptive moment estimation. This technique combines the advantages of two other optimization methods: AdaGrad, which performs well with sparse gradients, and RMSProp, which is effective in non-stationary settings. The Adam optimizer adjusts its learning rates based on the moving averages of the magnitude of the gradients, making it invariant to gradient scaling. (Kingma & Lei Ba, 2015, pp. 1 – 2) According to Balawejder, Adam is the most powerful optimizer algorithm currently available for machine learning and is suitable for most scenarios (Balawejder, 2022).

2.4 Reinforcement Learning

Reinforcement learning is a subset of machine learning. Ruth Brooks writes in their article “What is reinforcement learning?” that reinforcement learning is an AI-driven system, referred to sometimes as an agent, that learns through trial and error with feedback from its own actions. It mimics natural intelligence and attempts to maximize its reward function. (Li, 2018, p. 10; Brooks, n.d.)

Reinforcement learning is somewhat different from other machine learning categories. Brooks writes that reinforcement learning uses mapping between input and output like supervised learning does, but otherwise, they have little in common. While all machine learning models can be trained with a reward system, reinforcement learning is sequential, and it usually considers long-term accumulative rewards instead of instant rewards that supervised and unsupervised learning strive for, according to Li and Brooks. Reinforcement learning uses sequential decision-making and is usually model-free. This means that the agent learns via trial and error and applies the experience it has learned directly to its behavior. According to Li, the point of reinforcement learning is to find a balance between under-fitting and over-fitting models that perform best in any given task. He writes that according to the no free lunch theorem, there is no universally best model that could fit into any given situation and that for some problems, a machine learning solution might not even be the best option. (Li, 2018, pp. 10 - 11, 13; Brooks, n.d.)

A reinforcement learning model has a reward system at its core, although it is not strictly necessary for a reinforcement learning model. Li writes that those are mathematical functions that provide feedback for the agent to make decisions and when designing a reinforcement learning model, a reward policy needs to be carefully thought out. He emphasizes that a poorly specified reward might yield poor training outcomes for the model. (Li, 2018, p. 36)

Brooks points out three different types of reinforcement learning: policy-based learning, value-based learning, and model-based learning. Li has extensively explained these different policies, and I will briefly summarize them here. Policy-based learning aims to maximize the cumulative reward. Li highlights that a major issue regarding policy-based learning is that it can suffer from sample inefficiency where there are large variances in output estimates. Various solutions have been proposed to tackle this issue, and different policy methods have been researched to find a way to improve the sample efficiency while keeping stability and unbiasedness. (Li, 2018, 30; Brooks, n.d.) Value-based learning evaluates the potential benefits of each state or state-action pair within a given context. According to Li, Temporal

difference learning and Q-learning are classical representations of value-based learning. Value-based learning tries to maximize its given value function, and from there, it can derive optimal policy. (Li, 2018, 24) Finally, model-based learning is a method where the agent learns in an environment created for learning purposes. The learning uses the experience gained from the environment, and the agent then learns to perform within those constraints. (Li, 2018, 18; Brooks, n.d.)

A fundamental weakness in reinforcement learning is, according to Li, the dilemma between exploration and exploitation. The model faces a dilemma of either employing the information gathered so far to make the best available decision (exploitation) or gathering more information (exploration). Exploration might not produce the highest reward short term but is essential for better learning outcomes in the long term. If a model starts exploiting too early, it might stick with suboptimal actions. (Li, 2018, p. 42)

A subset of reinforcement learning is a concept called deep reinforcement learning, which uses hidden neural network layers in between inputs and outputs to calculate outcomes. Those were introduced in the chapter 2.3. in more detail. Li writes that the opposite of a deep reinforcement learning model is called a “shallow” model, which is used for example in linear functions and decision trees. (Li, 2018, p. 21)

Reinforcement learning is widely used in various different fields and has recently gained popularity. Li finds games and robotics as two widely used fields but also lists various other applications, such as natural language processing, machine translation, dialogue systems, computer vision, finance and business management, healthcare, education, energy, transportation, security, science, and art. For the scope of this thesis, I will mostly focus on the utilization of reinforcement learning in gaming.

According to Li, games provide an excellent platform for artificial intelligence algorithms. He pointed out that there already exist some phenomenal achievements of artificial intelligence performance in games. He uses Google’s AlphaGo as an example of this. Go is a game with a perfect information horizon, meaning that a player can theoretically calculate the outcome of every move. Such is the case in Gomoku as well. However, the search space is usually gigantic and thus unfeasible. For example, Li mentions that the search space of Go is 250^{150}, which makes estimating the entire game board extremely challenging. Reinforcement learning can provide solutions, such as in the case of AlphaGo, that are more performant than a simple tree search. According to him, AlphaGo is built with deep convolutional neural networks and reinforcement learning. In addition to board games and games with perfect

information horizons, Li mentions that there has been significant development in games with imperfect information horizons, such as card games like Texas Hold'em, as well as video games with complex inputs and varying degrees of information horizon availability. (Li, 2018, pp. 76 – 81)

Finally, reinforcement learning has been employed extensively in the field of robotics, according to Li. Robotics systems can greatly benefit from reinforcement learning. However, robotics has its particular challenges when it comes to reinforcement learning. One major challenge is the training of robotics systems. Li points out that it is easier to train a robot in a simulated environment than in real life. Reinforcement learning is sample intensive, meaning that the agent needs to run multiple training runs before it has reached a satisfactory level of performance. However, a simulation is usually not an entirely accurate representation of reality and the challenge of bridging the gap between the simulated environment and the real environment is a continuous challenge for reinforcement learning in the field of robotics. The reward specification also becomes even more crucial in real-world robotics applications, as a badly defined reward system for a model can yield unsatisfactory or undesirable outcomes. (Li, 2018, p. 82)

2.5 Previous Implementations for Gomoku-AI

Gomoku is usually considered a very enticing option for artificial intelligence development due to the game's relative simplicity. Two players play in turns, placing a stone on a grid-shaped board. The board is traditionally either of size 15x15 or 19x19. The player has only one type of movement to do, the rules are straightforward without exceptions to them, and the board size is limited. On the other hand, the game has a deep layer of strategies on how to play. This creates a wide variety of options to implement different kinds of solutions for developing artificial intelligence solutions for the game. Over time, a multitude of solutions have been proposed and there is a tournament for different AI solutions to be played against called Gomocup. Gomocup has been organized since 2000 and is currently the largest AI Gomoku tournament in the world. Gomocup utilizes rules found in professional Gomoku tournaments, such as restrictions on first-player opening moves and the fact that more than five in a row is not considered a win. (Gomocup.org, n.d.) For the scope of this thesis, the built solution will not be tested against any pre-existing artificial intelligences, nor will it be played in Gomocup. As mentioned earlier, this solution also does not take into account the rules found in professional Gomoku tournaments.

To gain a more comprehensive understanding of the wide range of different solutions, I will go through three recent solutions for Gomoku artificial intelligence. All of them produce a somewhat different solution with varying degrees of success. Measuring success varies between the solutions, so a qualitative analysis between the solutions is not feasible within the scope of this thesis. Instead, these solutions function as a tool that can be used to detect differences and similarities with the solution proposed in this thesis, as well as to learn about the best practices and avoid the most obvious pitfalls. In the implementation phase, I will outline in more detail my proposed solution for Gomoku artificial intelligence and compare it with the solutions presented here.

2.5.1 A Minimax Search Tree Solution

Dr. M. Ranjitha et al. have written a paper in Journal of Physics an article “Artificial Intelligence Algorithms and Techniques in the computation of Player-Adaptive Games”, in which they outline solutions for different classical board game artificial intelligences, one of them being Gomoku. For Gomoku, they presented a minimax search tree solution accompanied by an alpha-beta pruning algorithm. They noted that even with the alpha-beta pruning algorithm the search space during each move was still considerable due to the fact that the player may place the stone on any unoccupied cell on the grid. (M. Ranjitha et al., 2020, p. 2)

Minimax and alpha-beta pruning are relatively old concepts in computer science. S. H. Fuller, J. G. Gaschnig, and J. J. Gillogly explain these two concepts in their paper published already in 1973. According to them, a minimax search is a search tree approach to finding the most optimal move in a board game with two players by minimizing the possible loss. An opposite concept is a so-called maximin, where instead the minimum gain value is attempted to be maximized. They point out that this creates an issue where the search algorithm has a lot of possibilities to go through. For example, a game of checkers has 10^{40} and chess 10^{120} possible games, each with its own unique sets of moves. Back in the 70’s, Fuller et al. considered processing all of these moves to be impossible. With today’s processing power such limitations are no longer an issue, but searching through such an immense amount of potential moves would still take too long to be practical. Alpha-beta pruning is according to Fuller et al. similar to the minimax in the sense that it arrives to the same outcome of the best possible move, but faster. This is achieved thanks to the fact that an alpha-beta pruning algorithm takes only into account branches that have the highest payoff, thus reducing the needed number of computations to reach the most optimal move. (Fuller et. al., 1973, 1 – 7)

Ranjitha et al.'s solution used a heuristic function to discover the empty cell within the board with the highest payoff for each move. The value is a sum of all the threats associated with the cell position. According to Ranjitha et al., the AI was then also introduced with more offensive capabilities, however, the paper does not specify the methods used to improve such capabilities. (M. Ranjitha et al., 2020, 2-3) The solution described in the paper indicates that the AI was relatively defensive in its play style. It is also noteworthy that the AI is not using any machine learning mechanisms, however according to Ranjitha et al., the AI was designed so that it has eight degrees of difficulty and that it could scale the difficulty based on how well the opposing player is playing against it. The scalability mechanism implementation was also not further discussed in the paper. Ranjitha et al. write that the AI was play-tested against 50 human players. They reported that 43 playtesters said that they had enjoyed playing against the AI once it had scaled its difficulty to match the player's skill level. (M Ranjitha et al., 2020, p. 3)

2.5.2 Convolutional and Recurrent Neural Network Solution

Rongxiao Zhang writes in his paper "Convolutional and Recurrent Neural Network for Gomoku" about his machine learning approach for Gomoku artificial intelligence. He acknowledges in his paper that solutions based on search-tree algorithms can provide a solution for the game, but due to the nature of the game, the scalability of such systems is poor due to the massive amount of legal moves especially at the beginning of the game. (Zhang 2016, 225) Instead, he proposes a solution that uses self-teaching behavior enabling convolutional neural networks, accompanied by multi-dimensional recurrent neural networks. These two neural networks perform two different tasks in Zhang's solution: the convolutional neural network was assessed by its ability to detect human-recognizable winning patterns and the multi-dimensional recurrent neural network for its ability to play on different-sized boards. According to Zhang, the neural networks were not designed extremely deep nor complicated. (Zhang, 2016, p. 225, 227)

The convolutional neural network was trained to recognize different Gomoku patterns by using a large dataset from the RenjuNet database, which according to Zhang contains more than a million moves, therefore the convolutional neural network used pattern recognition to solve Gomoku moves. Some of these moves were filtered to avoid duplicates, to focus on moves with less than 5 consecutives, and to improve training time. The neural network consists of three convolutional input layers, and three fully connected layers, which are then fed to a Softmax classifier that filters out illegal moves and updates the move weights with a small randomizer to break symmetry. (Zhang, 2016, pp. 225 – 227)

Zhang trained the recurrent neural network with the same dataset as the convolutional neural network to do an image classification test. The neural network is constructed so that it does four sweeps throughout the game board matrix in four diagonal directions, covering the entire board. As with the convolutional neural network, the move is decided using a Softmax classifier. According to Zhang, this network is scalable in the sense that it can be trained using a smaller board, and then scaled up to a full-sized board for testing. (Zhang 2016, pp. 226 – 227)

The solution was tested against a tree search, alpha-pruning AI built on a custom Gomoku engine that Zhang assessed would be approximately playing on a human amateur level. Both networks had different metrics for evaluation. The convolutional neural network AI did not win a single game out of 1000 games against the tree search, while the win rate for the recurrent neural network was three percent. The recurrent neural network did manage to accurately validate 50 percent of the training data, but it could not transfer this to successful moves during testing. According to Zhang, the convolutional neural network could occasionally play good moves but was also inconsistent in its decision-making. Zhang identified issues with the training data. According to him, only a limited number of the features in the data were relevant to the game itself. In addition to that, a lot of moves were fixed in nature, especially when evaluating the starting moves. These issues can, according to Zhang, severely impact the relevance of the training data, even if the dataset itself is fairly large. As a concluding remark, Zhang points out that a self-playing implementation could bypass these issues and would be potentially quick to learn when implementing with a multi-dimensional recurrent neural network. (Zhang, 2016, pp. 227 - 230)

2.5.3 Move Prediction Using Deep Learning

Kun Shao, Dongbin Zhao, Zhentao Tang, and Yuanheng Zhu have published an article “Move prediction in Gomoku using deep learning”, in which they train deep convolutional neural networks using supervised learning to predict the moves made by expert Gomoku players from RenjuNet dataset. According to Shao et al., deep learning is a form of machine learning that employs multiple layers of neural networks to model a high-level abstraction of massive data, allowing it to learn very complex functions. Deep learning is especially useful in discovering complex features from high-dimensional data. The proposed solution from Shao et al. uses deep learning in conjunction with move prediction to improve the intelligence of the artificial intelligence agent. (Shao et al., 2016, pp. 1 – 2)

The success of Google's AlphaGo has worked as an inspiration to Shao et al. According to them, Google has successfully used move prediction on AlphaGo by training it with a very large dataset to maximize the likelihood of human-like moves in any particular state of the game. They conclude that move prediction and deep learning are closely connected, and can support each other very well in complex, non-smooth functions such as the Gomoku game. Shao et al. use Keras and TensorFlow, one of the most popular deep learning libraries, to train their artificial intelligence. The RenjuNet library they used contains over 1.2 million moves to train the AI with. Much like in the solution discussed in chapter 2.2.2, this creates a pattern classification problem for the AI to solve. (Shao et al., 2016, pp. 3 – 4)

The tests were conducted with varying degrees of depth in neural networks, ranging from one to ten layers. Shao et al. concluded that networks with more depth were performing better than shallow ones, although, after a depth of four, the performance gain became marginal and hardly discernible. The prediction accuracy increases rapidly on the first epoch, slowing down gradually after each successive epoch passes. Move accuracy is highest between approximately 30 and 70 moves. Shao et al. suggest that the move accuracy is affected differently at different stages of the game. In the beginning, there are too many empty positions which negatively affects the accuracy, while when the number of moves increases it obfuscates the potential moves, making it difficult for the network to detect patterns, thus also reducing accuracy. The overall expert move prediction Shao et al. managed to produce with their solution was 42 percent, which according to them reached the level of an expert Gomoku player. (Shao et al., 2016, pp. 1, 5 – 7)

2.6 Technologies and Libraries Used in This Project

Arguably, any object-oriented programming language could be used to create an artificial intelligence solution for Gomoku. However, some programming languages are better suited for this task than others. The most commonly used languages for artificial intelligence creation are Python, C++, R, and Java.

I ended up choosing Python as the development language for this thesis because it has a wide range of helpful libraries and frameworks for artificial intelligence development and machine learning. Python is also simple to read and versatile, making it a good choice for a complicated programming task such as this. Furthermore, Python has a library called "Pygame" which makes it easy and straightforward to implement a simple custom game that can be then interfaced with the artificial intelligence. I will next define the libraries I have chosen to use for the development. I will start with the Pygame.

Pygame is a free Python library that is designed for writing video games. Pygame is perhaps the most common library for Python game development, and according to pygame.org it has been downloaded millions of times and there are over 600 published titles created using the library. Pygame is lightweight and supports multithreading, which makes it well-suited for this project. (Pygame.org, n.d.)

For artificial intelligence development with Python, the most common options are Keras, TensorFlow, Sklearn, and PyTorch libraries. These libraries are all capable of producing neural network solutions, however, they all differ slightly from each other and have different use cases. After evaluating these different libraries, I ended up choosing PyTorch as the library used in this thesis.

Sklearn is meant for unsupervised and supervised learning, but as of writing this thesis has no built-in support for reinforcement learning, therefore rendering it unsuitable for the purposes of this project (Scikit-learn.org, n.d.). Keras was integrated as a part of TensorFlow in 2017, but can still be used independently of TensorFlow, if another back-end solution is selected, such as PyTorch. Keras cannot be run independently, therefore I found the need for using Keras debatable, although Keras promises to be a “human readable solution” (Keras.io, n.d.). TensorFlow is developed by Google and is an open-source library for artificial intelligence. TensorFlow has limited support for Windows platforms since version 2.10, which also sets restrictions on the Python version to be used (Tensorflow.org, n.d). Due to these limitations, I concluded that TensorFlow would not be the best solution for my project either. PyTorch has a good level of documentation and there were multiple game-related AI solutions created with PyTorch. PyTorch enables defining neural networks on the fly, which enables fast prototyping and exploration, is compatible with the most recent Python version, and is simple to use (Pytorch.org, n.d.). Therefore, choosing PyTorch was a simple choice.

The NumPy library was used to process data in this thesis project. According to NumPy documentation, NumPy is a Python library that offers an assortment of tools for operating with arrays and scientific computing. It supports n-dimensional arrays and is computationally fast and pre-compiled in C code. (Numpy.org, n.d.) For data visualization, I used the Matplotlib library, which supports creating static visualizations of data, such as graphs.

In addition to that, I used various other libraries to support the development process. A library called Tkinter was used to build a user interface for the game menus. While this project does not have elaborate user interface elements, it is still necessary to consider the usability of the

program to a degree that it does not hinder the work. Therefore, a rudimentary user interface was developed. A picture of the user interface can be found in the chapter 3.2.1. where I will discuss in more detail the implementation of the main menu.

3 Implementation

3.1 Design and Definitions

Based on the performance of some of the models I concluded that my artificial intelligence solution should not include image recognition solutions, as they seem to have had mixed results. A Gomoku board can be represented as a matrix that the network can convert to inputs, and then process the data in order to find the most optimal output. This, I believe, will improve the accuracy of the model.

Another method I decided to exclude was to use the training data from a publicly available professional Gomoku player game history. The data samples were relatively small to produce satisfactory results and were a major culprit in reducing the accuracy of Zhang's solution. Instead, I utilized self-learning. This happened in two phases: firstly, I trained the network with a custom-built Gomoku-AI algorithm that roughly equals an amateur-level Gomoku player. This algorithm is referred to as the TestAI. It utilizes an algorithm search to find the highest-scoring board cell. The TestAI favors a slightly defensive play style over an offensive style to make it suitable as a training partner for the neural network AI. The TestAI implementation will be described in more detail in chapter 3.2.3. The goal of training the network against the TestAI was that it would eventually learn to match and outplay the TestAI. Once this level was reached, the network played against itself for further training. The minimum goal for the neural network AI was to learn to play Gomoku better than the TestAI. A secondary goal was to improve the margin of how much better the neural network AI is in comparison to the TestAI. After the training had been completed, a validation round against the TestAI and a human opponent, myself, was played to evaluate the model's performance. These results will be discussed in chapter 3.5.

A classic Gomoku board is a grid pattern of size 15x15 or 19x19. In this project, I used a board of size 15x15, although the game was built so that a grid of any size can be used. In Gomoku, two players play against each other, taking turns to place stones on the grid trying to form a vertical, horizontal, or diagonal line of five stones with their own color. In this project, player 1 (in code the player is referred to as player 0) plays with black stones, while

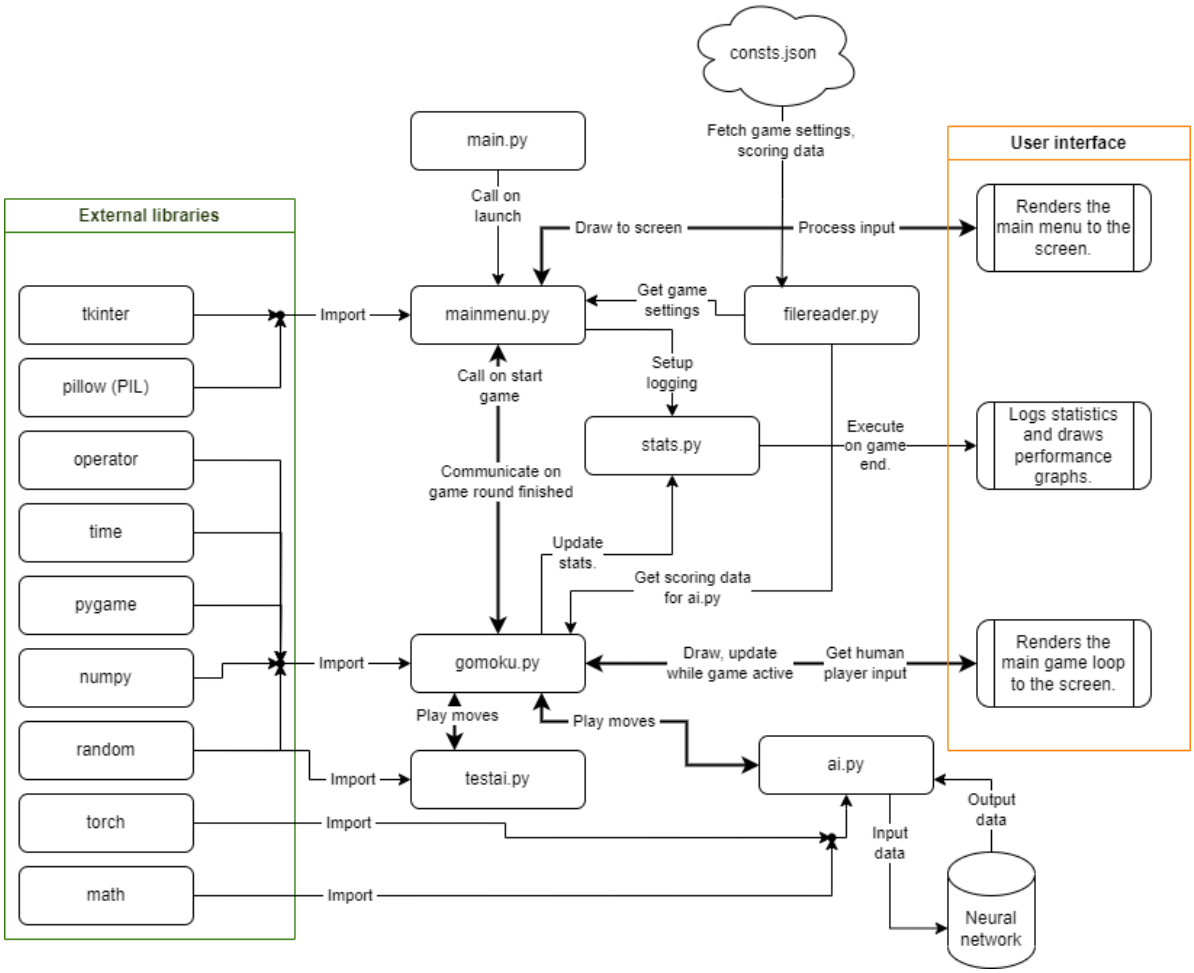
player 2 (in code referred to as player 1) plays with white stones. Player moves are scored so that the maximum score possible is the number of cells in the grid board, and the minimum score is the negative value of the cell amount.

During the game setup, the number of rounds can be specified. This is called the game session. A game session is used for both the training and evaluation phases. In a training phase, thousands of rounds needed to be played in order to accumulate a sufficient amount of training data for the model to process and learn from. The validation phase is when the trained model is tested against itself, the TestAI, and a human opponent. This validation phase is described in more detail in chapter 3.4. Model Performance.

3.2 Building the Solution

The implementation process was not straightforward. It involved a lot of preplanning to make sure that all the components would work together as seamlessly as possible. I will go through the process briefly here and discuss the results after the training process is completed. I will first talk about the supplementary modules for the project, such as the menu, file management, and plotting interface. Figure 2 illustrates how the different modules are related to each other.

Figure 2. Project flow chart.



In the flow chart above, the general flow of the project is defined. On the left are the external libraries used, encapsulated inside the “External Libraries” box. I have defined the most relevant external libraries for this project in section 2.7. In the middle are the modules built for this project and how they are related to each other. A one-way line represents a relation where data is flowing only towards the direction of the arrow, whereas a two-way thick arrow represents a relation where data is flowing both ways. On the right is the visual output the user can see during the execution, encapsulated inside the “User Interface” box.

3.2.1 Implementing the Main Menu and Other Supplementary Modules

In this section, I will go through the following Python scripts and their functionalities: mainmenu.py, filereader.py, and stats.py. I will also explain the functionality of consts.json file. In order to better understand the game and AI implementation, an understanding of the underlying supplementary modules is in order.

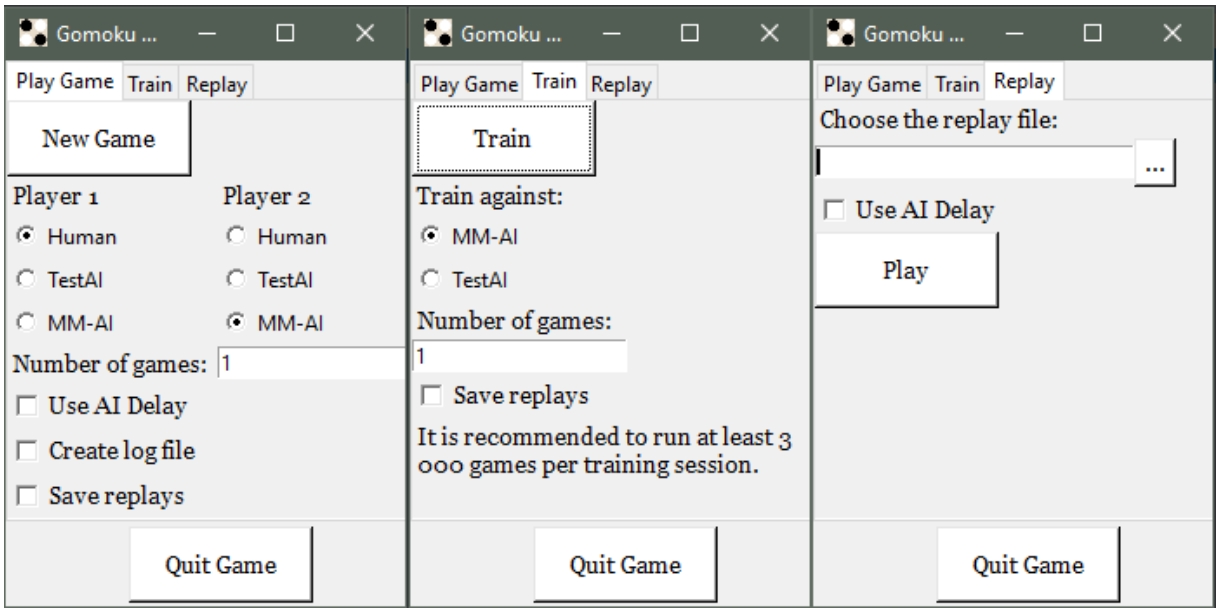
The Main Menu

The Main Menu is launched at the start of the application. Its main function is to provide the user with a user interface for selecting the right mode and executing it. The visual look of the Main Menu UI design can be found in Figure 3. There are three modes available: play, train, and replay. The play mode is where the user can select to play a game either against another human player, the TestAI, or the neural network AI (MM-AI). The user can define the number of games being played, and choose whether to log data, whether to use AI delay, and whether to save replays of the games being played. The main purpose of this mode is to test the trained model but it can also function as the “normal” play mode.

The train mode contains fewer settings. The training mode always runs the AI in training mode so that player 1 is MM-AI. The statistics are automatically recorded and the model performance is plotted on graphs, which will later be discussed in detail at the performance of the model.

The final Main Menu tab is the replay mode. If replays are recorded, they can be replayed here to analyze AI moves during a single game. Since the games can run really fast, they can be replayed here to observe singular moves more carefully and in detail. The need for this mode became evident during the evaluation of the model and was added toward the end of the production.

Figure 3. The UI design for all three main menu tabs.



Statistics

The purpose of the statistics script is to record statistics and to plot model performance graphs. The main menu defines whether statistics are being recorded. In the case of a training round, statistics and plotting are automatically turned on.

The File Reader and Consts.json

The file reader gets data from the consts.json file. The file contains settings for the game: the grid size, cell size, piece, background, and line colors. In addition, it contains the scoring data for the neural network AI model. Scoring will be discussed in more detail later on in chapter 3.3 Building the AI, but the reason scoring was implemented this way was to better take into consideration the various different cases the Gomoku board can be played effectively. Trying to implement all the necessary checks that can affect scoring would create a very complex and hard-to-read code, therefore creating a Json list of scoring patterns is more human-readable and easier to balance.

3.2.2 Implementing the Game

The base game is built using the pygame library. The Main Menu calls a run function from Gomoku.py that contains the game loop that is then active until the last game round is over. The run function first checks whether the game board is full. In such a scenario, a game would end in a tie unless there is a five in a row for either player.

The game then considers the move of a currently active player. Players are initialized by the Main Menu and depending on the settings, the player can be either human or AI. If the player is human, the game waits for a player input, which is a mouse click on any unoccupied grid cell. The input is considered a player move, and it ends the turn of the player in question. The game then proceeds to check whether the player won the game. If yes, the game exits the main gameplay loop and checks whether the round was the final round. If yes, then the game exits, and depending on whether logging was enabled, logs are printed and graphs are rendered. If the game is not over, a new round begins. If the player does not win the round, the game moves on to the next player and waits for its move.

In case of an AI move, it calls the AI to make its evaluation and run the decision-making process to output a grid coordinate on which the AI’s piece is then placed. The neural network AI decision-making process is further described in the following chapters. The neural

network AI is trained against the TestAI created for this project, which does not implement neural networks, but is rather more akin to a minimax algorithm, although it does not perform a full minimax search pattern. My assessment is that this AI functions roughly as well as a low-skill level human player would. I will briefly explain how this algorithm works.

3.2.3 TestAI

As part of the game implementation, I built an AI to test the neural network AI against. Training neural networks might take thousands of game rounds before the AI has gained a sufficient amount of learning data. Training AI against a human player would therefore take a very long time, thus the need for an automated solution. The neural network AI could also be playing against itself from the very beginning, but that might lead to a situation where it would learn behaviors that are sub-optimal. In other words, the training data would not be of sufficient quality to effectively train the model. Therefore, building another artificial intelligence solution to run the training is necessary. I call this AI the TestAI.

The TestAI functions so that it checks the entire game board to find all the possible moves. It then scores these available slots and chooses the cell with the highest score value. The score is calculated by evaluating adjacent cells up to a depth of five cells in eight different directions and then giving points based on whether there are player moves present. The score is increased if the cell is not empty and it has the same piece as the previous cell. If the piece is different than the previous one, then the score is reduced, as it is an indication of a closed row. The TestAI scores slightly higher if it manages to block the opposing player's moves, making it play slightly more defensively. After the score has been calculated, moves with the highest score are singled out of all the available moves, and one of the best moves is randomly selected, if there exists more than one such move.

Based on my testing of the TestAI, it performs reasonably well on a game board that is not very full of played pieces. As the board gets fuller, the TestAI struggles to find the best move that a human player could easily discover. However, for the purposes of training the neural network AI, the performance of the TestAI is sufficient.

3.3 Building the AI

The building process for the neural network AI was multi-staged and took a considerable amount of research to implement. There were multiple different neural network solutions

explored, as well as different implementations of data input for the network for move prediction and training. For clarity, the neural network AI model was named “MM-AI”. The name is a small wordplay, as neural networks are usually abbreviated as “nn”, and my initials are “MM” which is close to the neural network abbreviation, making a small reference to the author while retaining the likeness of the neural network abbreviation.

The idea from the very beginning was to build a convolutional neural network and as such, in each neural network implementation, the network always featured at least one convolutional network layer. The first layer was always a convolutional network layer, but the last and second last layer was also experimented with linear layers to see how they affect model performance. Convolutional network layers also ranged from one to three, making in total the range of neural network layers in the experiment phase to range between two and five. Each research was accompanied by experiments with input and output sizes, kernel, padding, and dropout values, as well as different activation, optimizer, and loss functions and adjusting the learning rates.

I will explain the functionality of the final version of the model, as well as the reasoning behind each choice in the following chapters. The text follows the order in which the program runs the AI during the gameplay. This I believe is a logical and reader-friendly way of explaining this rather complex model. The process starts with implementing the network and the AI class that implements the network. After that, I will discuss how the AI gets its action using the network. Then I will discuss how the scoring system works and finally, I will go through the training for short and long memory.

3.3.1 Implementing the Convolutional Neural Network

During the research phase, I experimented with ReLU and Sigmoid activation functions. For each layer, I applied different optimizers: Adam, Adamax, and Adagrad, as well as Stochastic Gradient Descent (SGD) and Averaged Stochastic Gradient Descent (ASGD) optimizers. For loss functions, I experimented with MSEloss and CrossEntropyLoss.

After the experimentation, I identified the neural network solution that yielded the most optimal results. The network has an input layer of size 3, a hidden layer of size 30, and an output layer of size 225. That is not to say that it is the most optimal solution, but it was the most optimal solution I was able to identify during the research phase. The neural network is constructed as a class called ConvNet and is as follows:

```
class ConvNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(ConvNet, self).__init__()
        self.layer1 = torch.nn.Sequential(
            torch.nn.Conv2d(3, hidden_dim, kernel_size=5, stride=1,padding=2),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=5, stride=1),
            torch.nn.Dropout(p=0.05))
        self.layer2 = torch.nn.Sequential(
            torch.nn.Conv2d(hidden_dim, output_dim, kernel_size=5, padding=2),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(kernel_size=5),
            torch.nn.Dropout(p=0.2))
```

While this might produce shallow neural networks that might not be optimal for deep learning, it was the best-performing model that I was able to identify. The network is used with a function called forward, which receives the input for the first layer as input value. This is the standard implementation when using PyTorch:

```
    def forward(self, x):
        out = F.relu(self.layer1(x))
        out = F.relu(self.layer2(out))
        return out
```

The ConvNet class has also functions for loading and saving the model for reusability. The network outputs values, from which the model then picks the most optimal one. The ConvNet class is inherited by GomokuAI class, which is initiated as follows:

```
class GomokuAI:
    def __init__(self, _board_size=15):
            self.game = None
            self.learning_rate = 0.00075
            self.board_size = _board_size
            self.gamma = 0.2
            self.epsilon = 0.25
            self.memory = deque(maxlen=MAX_MEMORY)
            self.model = self.build_model(self.board_size)
            self.optimizer = optim.Adam(params=self.model.parameters(),
            lr=self.learning_rate)
            self.criterion = nn.MSELoss()
            self.loss = 0
```

The class receives on initialization as an input the size of the grid. The grid is always assumed to be square and it has a default value of 15. The first variable game holds data from the game board and is refreshed at the beginning of each move.

A learning rate of 0.00075 was found to be a well-balanced value. Too high of a learning rate leads the AI to skip a lot of learning and thus not learning well enough or learning suboptimal behaviors. Conversely, too low a learning rate means that the learning is too slow and the AI might not be able to catch essential information during the training.

The gamma value is used as a discount factor when calculating the reward function. The value range is between 0 and 1, and lower values usually lead the AI to prioritize short-term rewards instead of long-term rewards. The reverse is true for values closer to 1.

The epsilon value defines the probability for exploration when getting the AI move. Exploration means that the AI explores alternative moves instead of the most optimal one it would otherwise get outputted from the network. This is useful especially early on in the training when the model does not have a good understanding of an ideal move. It helps the AI to discover different possibilities. There is also an epsilon decay rate, which is set to 0.999, meaning that after each turn the epsilon is multiplied by the decay rate, slowly decreasing the likelihood of the AI exploration, and instead moving towards full exploitation, which means choosing the best move the model is able to output. Epsilon values are only used in training mode, in evaluation the AI always chooses exploitation.

Since Gomoku is a game where a random singular move can have a drastic effect on the whole game, I implemented a solution where the exploration would select a random move from the top percentile of available moves the AI has evaluated from the game board, instead of pure random. However, as a stopgap, I also implemented a fully random-based move selection of all available moves if neither regular exploration or exploitation produced any legal moves.

The memory stores the AI’s memory until the maximum memory has been reached. For this model, the memory size is set to 1 000 000. Using memory helps the model to learn from its past experiences, however too large of a memory pool can slow down the processing speed.

Optimizer is used to calculate model weights based on the gradients computed during backpropagation. After some experimentation, Adam was found to be the most optimal optimizer function for this model. I have discussed optimizer functions in detail in chapter 2.3.3.

The loss function measures the difference between the predicted values and the target values. Different loss functions use different algorithms and mean squared error provided

loss data that reached closest to zero for this model. The loss value reflects how well the model's prediction matches with expected future rewards.

3.3.2 How the Model Performs Its Move

The action for the neural network AI is called from the Gomoku.py script when it's the AI's turn to make a move. Before that, however, the board is converted into a so-called "one-hot board". This means that each state is represented by an array of binary values of 0 or 1. Jason Brownlee explains the one-hot encoding in his article on Machine Learning Mastery "Why One-Hot Encode Data in Machine Learning?". He describes that one-hot is necessary when dealing with labeled data, for example when classifying pets to "dogs" and "cats". Machine learning models cannot directly handle categorical data, and it must be converted to numeric values instead. Brownlee explains that sometimes a simple integer encoding might be sufficient, if it concerns data where there is a direct relation between the values, such as when labeling "place": "first", "second", "third" and so on. However, when such a relation does not exist in data, this kind of labeling might be misleading for the AI, and in these situations one-hot encoding is useful. One-hot encoded data is represented as an array of binary values, which removes the potential of AI doing value assumptions when those are not required. (Brownlee, 2020)

The Gomoku board in my solution is by default represented with zeros, ones, and twos. Zero means that the space is unoccupied, one means that it is occupied by the first player, and two means that it is occupied by player two. This information might be confusing for the AI for the aforementioned reasons. Instead, after converting to a "one-hot board", each board cell is represented with an array consisting of three binary values. An empty board cell is [1, 0, 0], player one cell is [0, 1, 0] and player two cell is [0, 0, 1]. This way the AI will not make any value assumptions of any individual cell. The neural network input layer is defined to be of size three, which is exactly for this reason.

The get_action function is as follows:

```
def get_action(self, state, one_hot_board):
            valid_moves = self.get_valid_moves(state)
            current_state = torch.tensor(self.get_state(one_hot_board),
            dtype=torch.float)
            action = None
        with torch.no_grad():
            prediction = self.model(current_state)
```

```
if random.random() < self.epsilon:
    num_moves_to_select = max(int(len(valid_moves) * .025), 1)
    if num_moves_to_select > 0:
        try:
    top_moves_indices = torch.topk(prediction.flatten(),
    k=num_moves_to_select-1).indices
    action =
    self.id_to_move(top_moves_indices[torch.randint(len(top_moves_indic
    es), (1,))].item(), valid_moves)
        except RuntimeError:
            action = None
else:
    state_concat = torch.cat((self.get_state(one_hot_board),
    scores_tensor), dim=1)
    pred_possible_moves = int(torch.max(prediction))
    pred_indices = np.where(np_scores == pred_possible_moves)
    if len(pred_indices[0]) > 0:
        idx = random.randint(0, len(pred_indices[0])-1)
        action = (pred_indices[0][idx], pred_indices[1][idx])
     else:
        action = None
while the action is None:
    # if no action, switch to exploration
    action = self.id_to_move(self.get_random_action(state),
    valid_moves)
self.adjust_epsilon()
return action
```

Before the AI plans its move, the valid moves are fetched by using the current board state as input. Valid moves are moves that do not overlap with any of the already played cells. If the AI would propose moving to an occupied cell, it would result in the action being invalid and the AI would instead revert to an exploration move, to prevent conflicts and infinite loops.

After that, the board state is converted into a tensor object that the neural network can process. Then a prediction of the next move is made, which outputs an array representing the prediction of the next game state. From this game state, the best value is selected to represent the next move the AI will make and it is returned as an action, which is a grid coordinate that the Gomoku.py can then use to place the game piece on the board. After the move has been made, the score of the AI's move is evaluated and used for the short memory training.

3.3.3 How Scoring is Calculated

There are two scoring models for the AI: short score and long score. Short score is intended to be an immediate score for the AI to evaluate its latest move with. The long score is used for the AI to evaluate its performance during a game round, giving feedback on whether the AI managed to win the game and how well the game round went overall. I will first discuss the short score, and then the long score.

As discussed earlier, the consts.json file contains the scoring data. This data is meant for the short score. The scoring is done by looping through the adjacent tiles of each tile in the board. The adjacent tiles are then checked against the scoring data in consts.json. If there is a match, the scoring value is picked from the json file, otherwise, the score is zero. The entire board is scored at the beginning of the AI turn. For each cell, the adjacent tiles are assessed. The assessment is done in eight directions and up to five cells deep. Opposing directions are then added together, and that value is added to the final tally of the cell's score value. That forms the final score for a single cell. This is illustrated in Figure 4. The score of each cell is indicated on top of the cell. Cells that already have a piece are of value -1. Scores in this image are not yet normalized. The model does not receive the scoring data of the entire board, instead, it will receive the value of its own move scaled between zero to one, one being the best move possible and zero worst move possible.

Figure 4. An example of a scoring scheme.

0	0	0	1	0	0	0
0	1	0	4	0	0	0
1	1	3	13	1	1	0
0	3	6	-1	4	1	0
0	1	-1	-1	3	0	0
0	1	4	15	2	1	0
0	1	1	4	1	1	0
0	0	0	1	0	0	0

To simplify and reduce the number of scoring scenarios in the const.json file, scoring is reformatted so that the first occupied occurrence in a row to be measured is always considered as 1, and any opposing occupied occurrence is then considered as 2 in the scoring. The conversion and score calculation are done in pairs, where both directions, for example up and down, are calculated at the same time, and as such both directions consider

the first occupied occurrence with the same value, thus creating useful calculations. The scoring data is formatted so that the key is the pattern in question, and the value is its final score. A snippet of scoring can be found below to further illustrate the solution:

```
"scores" : [
    {
        "partial" : [
            {
                "[]": 0,
                "[0]": 0,
                "[1]":  2,
                "[0, 0]": 0,
                "[1, 0]": 2,
                "[1, 1]": 13,
                "[0, 1]": 1,
                "[1, 2]": 1,
                "[0, 2]": 0,
                "[0, 0, 0]": 0,
                ...
```

Finally, the score is normalized to values between 0 to 1, so that it functions better when training the AI. The score is only used for AI, so it is not visible to the player during gameplay.

The long score is calculated with a more simplistic formula. The AI is rewarded for winning the game and punished for losing the game. The function of the long score is to give AI feedback on how the game round went so that it can adjust its long-term strategies better. Long score is calculated with an equation:

$$s_{long} = \pm 1 \times g^2 \pm n_{moves}$$

Where *g* is the grid size (by default, and in the context of this thesis, this is 15) and *n* is the number of moves the player has made during the game round. If the AI is victorious, the grid size is a positive value, while if the AI fails the round, the grid size is multiplied by -1 to negate its value. Then if the AI was victorious, each played move is reduced from its final score, and if the AI did not win, each move is added to its score. In case of a tie, the score is zero. The hypothesis is that this kind of scoring encourages the AI to play more aggressively and try to win quickly. The score is finally normalized to values between -1 to 1.

3.3.4 Training the AI Memory

After the AI has made its move the AI will immediately train its short memory. This means that the AI will assess its latest move and learn from it.

```
def train_short_memory(self, state, reward, next_state):
        self.model.train()
        state = torch.tensor(state, dtype=torch.float).unsqueeze(0)
        next_state = torch.tensor(next_state,
        dtype=torch.float).unsqueeze(0)
        reward = torch.tensor([reward], dtype=torch.float)
        q_pred = self.model(state)
        q_next = self.model(next_state)
        q_new = q_pred + self.learning_rate * (reward + self.gamma *
        torch.argmax(q_pred - q_next))
        loss = self.criterion(q_pred, q_new)
        loss.requires_grad_(requires_grad=True)
        loss.backward()
        self.loss = loss.detach().numpy()  # for logging purposes
        self.optimizer.step()
        self.optimizer.zero_grad(set_to_none=False)
```

The model is set to a training state so that it is capable of learning. The input values are converted to tensor objects so that the model can handle them as inputs and then the reward function is used to calculate the new q value, which is used to calculate the loss value. The reward function is described well by Maurio Comi's article on the towardsdatascience.com website "How to teach AI to play Games: Deep Reinforcement Learning". In his article, he describes the reward function as follows:

$$NewQ(s,a) = \mathrm{Q(s,a)} + \alpha[R(s,a) + \gamma max Q'(s',a') - Q(s,a)]$$

The function states that the new q value is the current q value plus the learning rate multiplied by the reward plus gamma multiplied by the maximum predicted reward minus the current q value. (Comi, 2018) Comi presents this equation in his article for a snake game reward function, but in my opinion, the same function can apply to Gomoku as well.

Once the game round has been concluded, the AI will train for its long memory. For this, the AI considers all the moves it has made during the game round and the final result of the game (win, loss, or tie). It then performs a similar function as presented for the short memory training. During training, move accuracy, final score, and loss value for both short and long-

memory training are recorded for model performance evaluation, which will be discussed next.

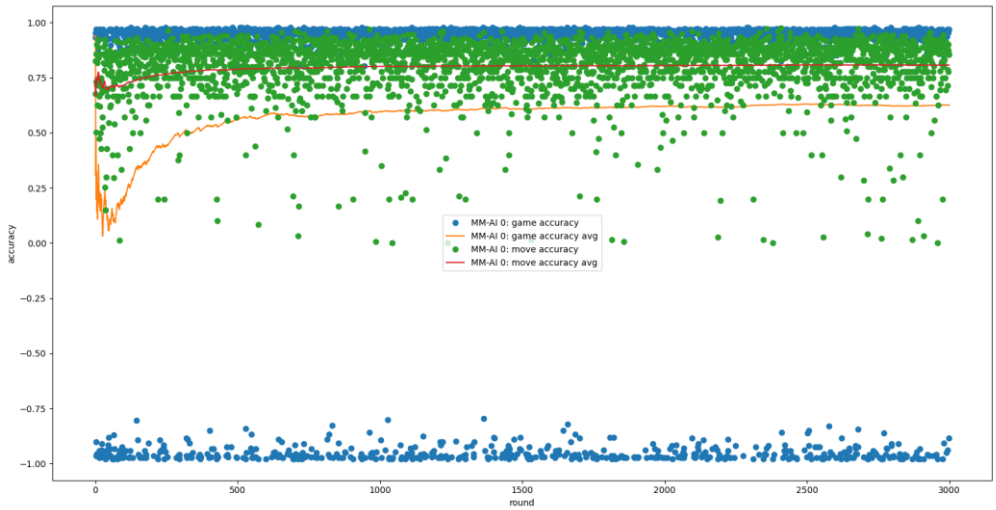
3.4 Model Performance

Once a well-performing model was identified, the results were speaking for themselves. The MM-AI was able to consistently win the TestAI, and after three thousand rounds of training the MM-AI was performing better than the TestAI. I will go through the results next and present the data with few diagrams.

Based on these results, the AI seems to reach a peak performance value after approximately 1000 games. After this point, the AI can predict the move with roughly 75 % accuracy. There is only an incremental increase in prediction quality after this point, however, the final score seems to be slowly increasing. The first 500 rounds witnessed a dramatic change in model performance, after which the increase was slower but noticeable. During the training phase, the MM-AI managed to win 83 percent of the games. In the next chapter, I will go through the validation phase, where I test the trained model against the TestAI, itself, and a human opponent.

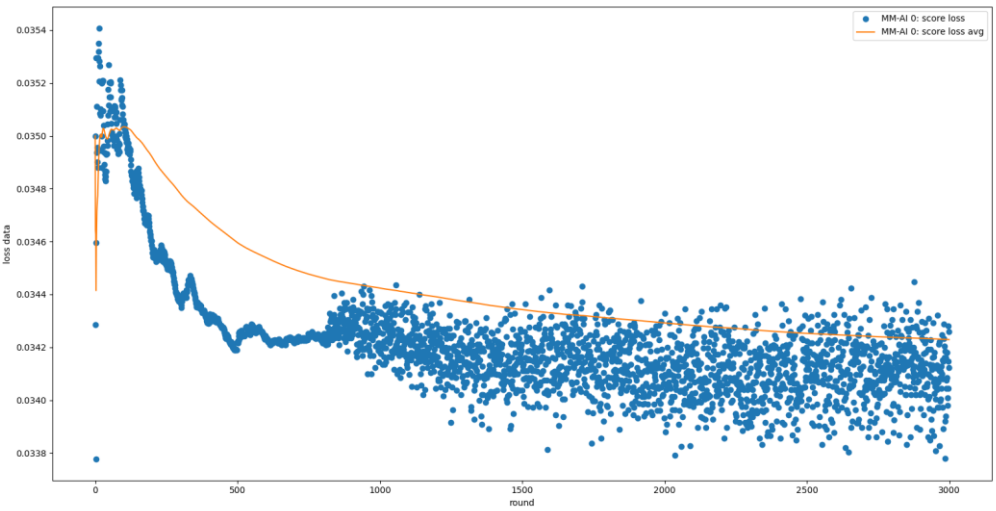
Based on the results portrayed in Figure 5, it seems that the neural network AI was playing mostly rather short games: it either won or lost quickly. The closer the score would be to zero, the longer the game was. Since most of the final scores are located close to the extremes, it indicates that the games were usually short and did not reach a tie situation once. This is consistent with the hypothesis that the long memory training formula encouraged AI for short and aggressive games instead of long-winded ones. In Figure 5, blue dots represent the final score of a game round, normalized between -1 to 1, -1 representing a complete failure, and 1 representing a perfect game. The orange line represents the rolling average of the final game scores. Green dots represent the average accuracy of moves during a game round, and the red line represents the rolling average of move accuracy. Accuracy is measured between 0 to 1, 0 meaning completely inaccurate moves, and 1 meaning only perfect moves.

Figure 5. The accuracy data for the training set.



The loss variance during the training for long memory loss is very small. In Figure 6, the extremities lie within a range of 0.002, the average being around 0.0342. As far as loss values are concerned, this amount is decent. As with the average game score, the average loss score kept on improving over the course of the training, albeit at a slower pace than in the beginning. Here, blue dots represent the loss amount for each round, and the orange line represents the rolling average.

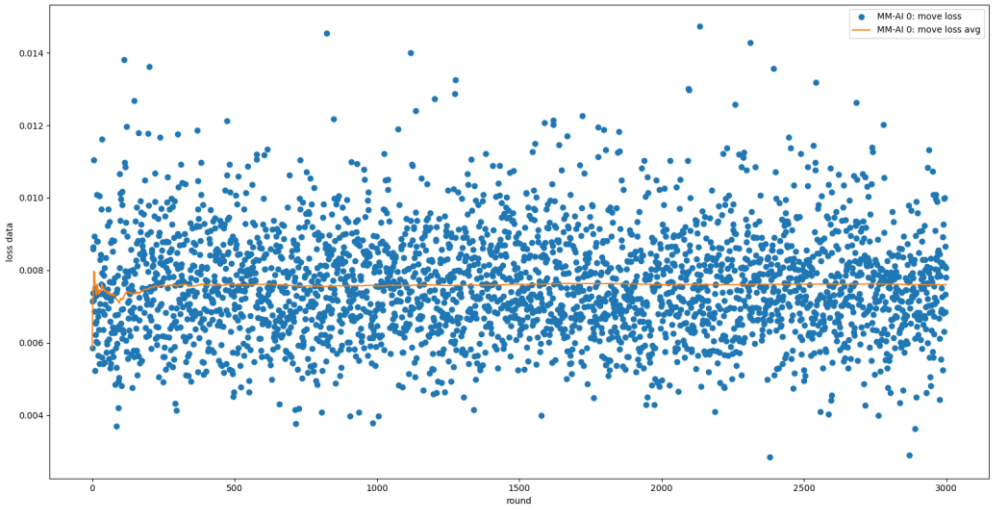
Figure 6. The loss data for overall scoring for the training set.



Move loss presented in Figure 7 indicates the model's ability to predict future rewards from individual moves, using its short memory. The values remained remarkably consistent

throughout the training. The rolling average is slightly below 0.008, which is a decent loss value, meaning that the AI was able to accurately predict future rewards.

Figure 7. The loss data for moves for the training set.



3.4.1 Validating the Results

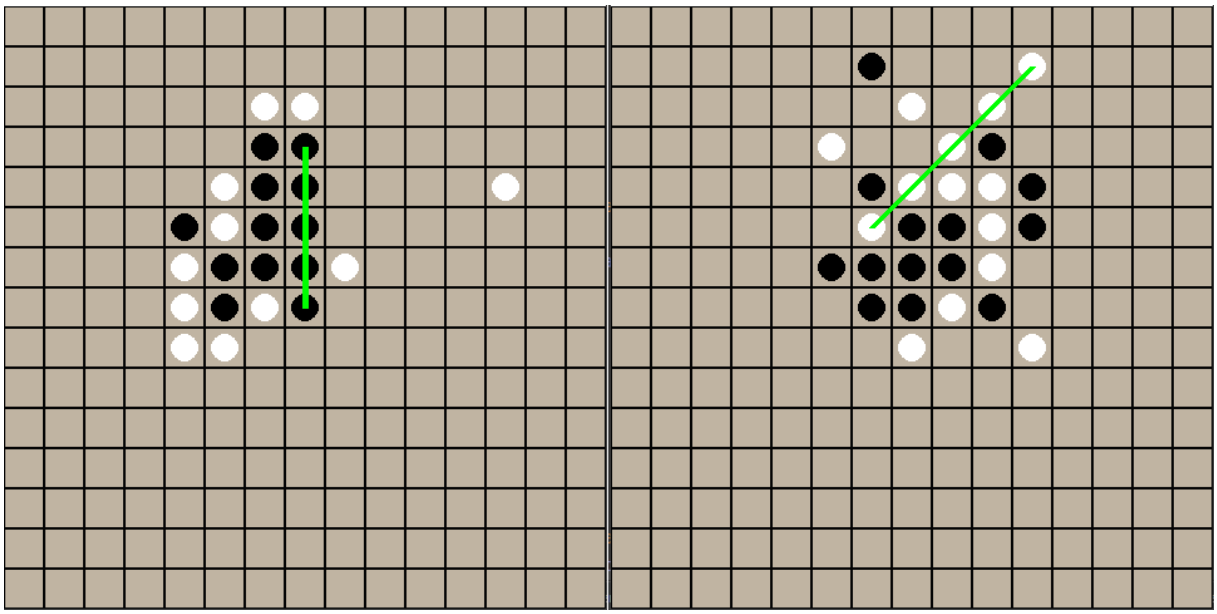
The model was tested against itself, the TestAI, and a human player, myself, to validate the learning results. When pitted against itself, the results were unsurprisingly tight. The model played 200 rounds against itself. The first player was able to win 46 percent of the games, while the second player won 45.5 percent of the games, showing that the first player had a slight edge over the second player, as mentioned in Chapter 2.1. This result was consistent through multiple runs. The rest of the games were ties. These games were longer than against the TestAI or the human player, lasting on average 42 turns per player.

Against the TestAI, 400 validation rounds were played so that the model was playing 200 rounds both as the first and the second player. As discussed earlier, in Gomoku, the first player will always have a slight advantage over the second player, but since mitigating that advantage has not been taken into account within the scope of this thesis, my prediction was that the model can perform somewhat better as the first player, than as the second player. After playing those 400 validation rounds, the results reflected this by showing that the model was able to win 83.67 percent of the games as the second player, and 94.33 percent of the games as the first player, meaning that it performed considerably better as the first player than as the second player. Against the TestAI, the average round lasted for 11.3 (as the first

player) and 16.6 (as the second player) moves per player, meaning that the game was considerably shorter than when the model played against itself.

The model was able to create a genuine challenge against the human opponent as well, winning three rounds out of ten. The round length was approximately 16 moves per player. An interesting observation that I made during the validation rounds was that the MM-AI was capable of a very offensive play style if it was allowed to do that. On the other hand, if the human player was playing offense, the AI adopted a more defensive strategy, trying to block the human player's advances. In the case of a long game, the AI's prediction quality started to slightly diminish, as there were multiple "good" moves to obfuscate the perfect move. The Figure 8 portrays two game situations that occurred while playing against the human opponent. In both instances, the human player is playing with black pieces and the AI is playing with the white pieces.

Figure 8. Two game rounds of human vs. MM-AI.



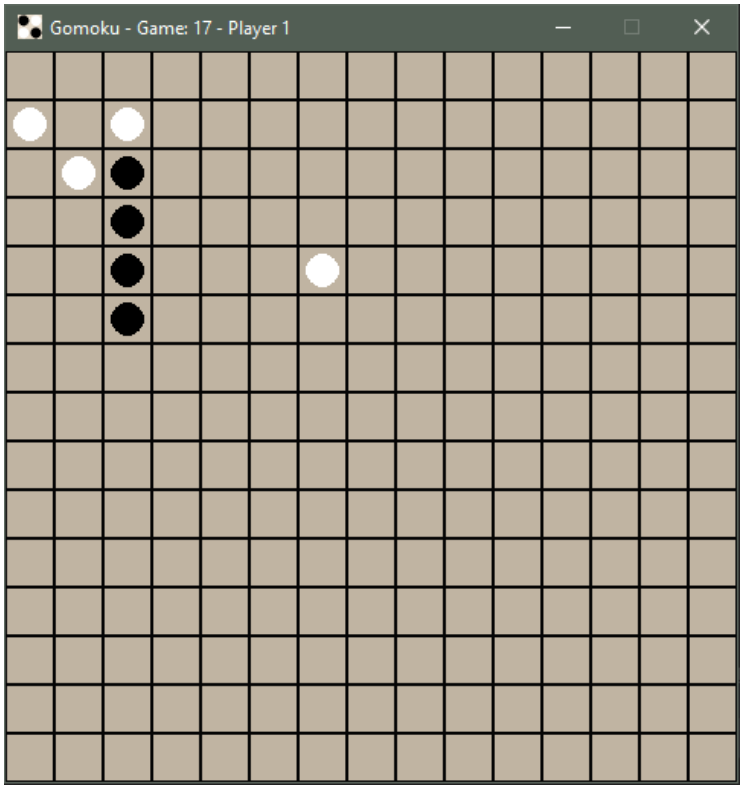
3.4.2 AI Hallucination

During the validation rounds, it became evident that the model performed some "hallucination" games every now and then. Although this concept is mostly used when talking about Large Language Model artificial intelligence, the same definition also matches the behavior this model sometimes exhibits. According to the IBM webpage, an artificial intelligence hallucination means behavior that appears inaccurate or nonsensical for a human observer. The term is used metaphorically in the context of AI. According to IBM, AI

hallucinations may occur for example due to model overfitting, inaccurate or biased training data, or if the model is overly complex. Hallucinations might also appear if the AI is not constrained enough with its output, giving the AI too many unnecessary choices. IBM suggests using data templates and human oversight as well as to continuously tweak the model to mitigate AI hallucination (IBM, n.d.)

Based on my experiments, the model was hallucinating more if it was trained with a small training set (1000 game rounds or less). The more it was trained, the less frequent the hallucinations became. At most, the model was tested with 10,000 rounds of training, so it is possible that if the training had been continued further, at some point the model would have started to overfit, which could again increase the amount of hallucination. While I did not measure the actual amount of hallucinations, after a training of three thousand games I estimate that the model hallucinated less than one game out of one hundred. Figure 9 shows an example of AI hallucination. The AI is playing as the white pieces and the current turn is for the player one (black pieces), resulting in the victory of player one. The AI lets the player win seemingly unopposed, making moves that for a human observer make little sense.

Figure 9. An example of AI hallucination within the context of this solution.



4 Conclusion

Overall, the project was a success. The neural network implementation was relatively straightforward thanks to the PyTorch library, which is easy to learn, and has well-constructed documentation to refer to. The model performance was also satisfactory and managed to reach the expectations set for the model.

While the network architecture is shallow, it can still offer a satisfactory level of challenge for an average player. To improve on the model's performance, a more complex neural network architecture would need to be implemented, along with some improvement on the scoring system. This implementation did not use adjustable learning rate nor did it change the optimizer function during the training. These steps could be used to further optimize the model: lowering the learning rate as the learning progresses would enable the network to learn more minute details and switching the optimizer function from Adam to SGD could further improve prediction accuracy over time.

A surprising discovery was how quickly the model was able to learn, although the learning rate was set to a relatively low value. Due to the shallowness of the network, the model reached its peak capacity very quickly, in just a little over 1000 game rounds. The more complex networks I tested did not show significant development even at 10,000 rounds of play. Finding a network that can produce results the model can learn from is an intricate and tricky business that requires a lot of trial and error, as well as a lot of time especially when training the network after each incremental change to see how it affects the performance.

At the beginning of this thesis project, I was contemplating including a genetic algorithm as well, but due to time constraints, it had to be excluded from the scope. A genetic algorithm is an algorithm that can tweak certain parameters of the model to help discover the most optimal solutions. Common parameters to be tweaked are usually learning rate, exploration rate as well as network layer sizes. In addition to this, the genetic algorithm could be used to optimize the scoring table to a solution that would yield the most optimal results.

With the aforementioned considerations in mind, I firmly believe that this implementation has a lot of potential and could be used to further develop a Gomoku AI that could perhaps compete against other AI solutions as well. In addition to having success with the model, I strongly feel that I have been able to learn a great deal about neural network development during the process of this project. Before this project, I had tackled the topic through some

courses, but this thesis has given me a more hands-on experience of what it means to construct a neural network solution from scratch and to make it function as intended.

References

Balawejder, M. (2022). *Optimizers in Machine Learning.* Medium.com. https://medium.com/nerd-for-tech/optimizers-in-machine-learning-f1a9c549f8b4

Britannica. (2024). *Moore's Law*. The Encyclopedia of Britannica. https://www.britannica.com/technology/Moores-law

Brooks, R. (n.d.). *What is reinforcement learning?* University of York. https://online.york.ac.uk/what-is-reinforcement-learning/

Brown, S. (2021). *Machine learning, explained*. MIT Sloan School of Management. https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained

Brownlee, J. (2020). *Why One-Hot Encode Data in Machine Learning?* Machine Learning Mastery. https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/

Chrispresso (2019) *SnakeAI.* A GitHub repository. Last accessed 17th of March 2024. https://github.com/Chrispresso/SnakeAI

Comi, M. 2018. *How to teach AI to play Games: Deep Reinforcement Learning*. https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a

Copeland, B.J. (2024). *Artificial Intelligence*. The Encyclopedia of Britannica. https://www.britannica.com/technology/artificial-intelligence

Council of Europe. (2020). *History of Artificial Intelligence*. https://www.coe.int/en/web/artificial-intelligence/history-of-ai

Goldstein, S. & Kirk-Giannini, C. D. (2023). *AI is closer than ever to passing the Turing test for 'intelligence'. What happens when it does?* The Conversation. https://theconversation.com/ai-is-closer-than-ever-to-passing-the-turing-test-for-intelligence-what-happens-when-it-does-214721

Gomocup.org. (n.d.). https://gomocup.org/

Gomokuworld.com. (n.d). *The History of Gomoku*. http://gomokuworld.com/gomoku/3

Google DeepMind. (2019). *AlphaStar: Grandmaster level in StarCraft II using multi-agent reinforcement learning*. https://deepmind.google/discover/blog/alphastar-grandmaster-level-in-starcraft-ii-using-multi-agent-reinforcement-learning/

Fuller, S. H., Gashnig, J. G., Gillogly & J. J. (1973). *Analysis of the Alpha-Beta Pruning Algorithm.* Carnegie-Mellon University

Keras.io. (n.d.). *Getting Started with Keras*. https://keras.io/getting_started/

Kingma, D. P. & Lei Ba, J. (2015). *Adam: A Method for Stochastic Optimization*. Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego. https://doi.org/10.48550/arXiv.1412.6980

Kneusel, R. T. 2021. *Practical Deep Learning: A Python-based Approach*. First Edition. No Starch Press, Inc.

Li, Y. (2018). *Deep Reinforcement Learning*. Cornell University.

Mahapatra, S. (2018). *Why Deep Learning over Traditional Machine Learning?* Towards Data Science. https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063

Numpy.org (n.d.). *What is NumPy?* https://numpy.org/doc/stable/user/whatisnumpy.html

OpenAI. (2022). *AI and efficiency*. https://openai.com/research/ai-and-efficiency

Pygame.org. (n.d). *About – Pygame wiki*. https://www.pygame.org/wiki/about

Pytorch.org (n.d). *PyTorch documentation*. https://PyTorch.org/docs/stable/

Ranjitha M. et al. (2020). Artificial Intelligence Algorithms and Techniques in the computation of Player Adaptive Games. *Journal of Physics: Conf. Ser. 1427 012006*.

Scikit-learn.org. (n.d.). *Why is there no support for deep or reinforcement learning? Will there be such support in the future?* https://scikit-learn.org/stable/faq.html#why-is-there-no-support-for-deep-or-reinforcement-learning-will-there-be-such-support-in-the-future

Srinivasan, A. V. (2019). *Stochastic Gradient Descent — Clearly Explained !!* Towards Data Science. https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31

Tensorflow.org. (n.d.). *Install TensorFlow with Pip*. https://www.tensorflow.org/install/pip

Zhang, R. 2016. *Convolutional and Recurrent Neural Network for Gomoku.* Stanford University Press.

Appendix 1. Source Code Location

The source code and a trained model can be found on GitHub:
https://github.com/Mig26/gomoku-thesis-proj