MAN VS MACHINE: BEATING HUMANS IN A MULTIPLAYER CARD GAME WITHOUT LOOKAHEAD

Lappeenranta-Lahti University of Technology LUT

Master's Program in Computational Engineering, Bachelor's Thesis

2023

Ilmari Vahteristo

Examiners: Andreas Rupp, Teemu Härkönen, Heikki Haario

ABSTRACT

Lappeenranta-Lahti University of Technology LUT
School of Engineering Science
Computational Engineering

Ilmari Vahteristo

Man vs machine: Beating humans in a multiplayer card game without lookahead

Bachelor's thesis
2023
30 pages, 6 figures, 1 tables, 2 appendices
Examiners: Andreas Rupp, Teemu Härkönen, Heikki Haario

Keywords: Moska, card game, hidden information, neural network, simulation, multiplayer

Games offer a fun and challenging test bed for algorithms and artificial intelligence, due to their well-defined environment, potentially large complexity, and '*man vs machine*' dichotomy. Since vacuum-tube computers, scientists have used games to test how intelligently they can make a computer play and to benchmark progress in hardware, computer science, and mathematics.

In this study, a representation for 'Team Machine' in the hidden information card game 'Moska' is created. Moska is a multiplayer, non-sequential, shedding-type card game with a large game tree complexity. It has similarities with the popular Russian card game 'Durak'.

We simulate Moska games and gather data from them to train a deep neural network to evaluate how good a position is from the player's perspective, i.e. with hidden information. We use this neural network for move selection with a look-ahead of one and test the performance of the agent by launching a website where people can play against the agent. We observe the agent successfully employ multiple strategies used by humans and reliably outplay even advanced human players.

TIIVISTELMÄ

Lappeenrannan-Lahden teknillinen yliopisto LUT
School of Engineering Science
Laskennallinen tekniikka

Ilmari Vahteristo

Ihminen vastaan kone: Yli-ihmisen tasolla pelaaminen monen pelaajan korttipelissä

Kandidaatintyö
2023
30 sivua, 6 kuvaa, 1 taulukkoa, 2 liitettä
Tarkastajat: Andreas Rupp, Teemu Härkönen, Heikki Haario ja

Hakusanat: Moska, korttipeli, piilotettu informaatio, neuroverkko, simulaatio, moninpeli

Pelit tarjoavat hauskan ja haastavan testialustan algoritmeille ja tekoälylle, koska niillä on hyvin määritelty ympäristö, mahdollisesti suuri kompleksisuus ja '*ihminen vastaan kone*'-dikotomia. Jo tyhjiöputkitietokoneista lähtien tutkijat ovat käyttäneet pelejä testatakseen kuinka älykkäästi he saavat tietokoneen pelaamaan, sekä seuratakseen edistystä laitteistossa, tietojenkäsittelytieteessä ja matematiikassa.

Tässä tutkimuksessa luomme pelaajan 'Tiimi Koneelle' piilotetun tiedon korttipelissä nimeltä 'Moska'. Moska on moninpeli, jossa pelivuorot eivät kulje järjestyksessä ja jossa pelaajat pyrkivät eroon korteistaan. Moskalla on erittäin suuri pelipuu ja jo yhdestä aloitus konfiguraatiosta voi saada saman verran pelejä kuin koko universumissa on atomeja.

Simuloimme Moska-pelejä ja keräämme niistä dataa syvän neuroverkon kouluttamiseksi arvioimaan sitä, kuinka hyvä tilanne on pelaajan näkökulmasta - eli piilotetun tiedon kanssa. Käytämme tätä neuroverkkoa siirtovalinnassa vertaamalla jokaisen mahdollisen siirron hyvyyttä tutkimatta pelipuuta sen pidemmälle. Tutkimme luodun pelaajan taitoja tekemällä verkkosivun, jossa ihmiset voivat pelata tätä tekoälyä vastaan. Havaitsemme agentin käyttävän menestyksekkäästi useita ihmisten käyttämiä strategioita ja pelaavan paremmin kuin jopa kokeneet ihmispelaajat.

ACKNOWLEDGEMENTS

I would like to thank my friends at Algebros, and the Lateksii guild room personnel, who gave insights into many problems I faced and got me interested to start, and kept me motivated to keep going. Also, a big thank you to my girlfriend Hanna, who put up with my constant talk about Moska, and pretended to be interested. Also, a special thank you to all who contributed by playing against the machines, which brought very valuable insights!

I would also like to thank my supervisors for offering good feedback, guidance, and discussions about different approaches and academic writing, as well as access to powerful computing resources, without which this wouldn't have happened.

Lappeenranta, May 15, 2023

Ilmari Vahteristo

CONTENTS

Appendix 2: Introduction to Moska

1 INTRODUCTION

1.1 Moska

Moska is a Finnish card game, which has likely originated from the more popular Russian card game Durak. Moska is more complex than Durak since it is played with a full deck, the target can play directly from the deck, the attacking turns are not sequential, and the target player can attack themself. In this section, we will briefly go through the main rules of the game. For a more comprehensive guide to Moska, see the Appendix.

1.1.1 Overview

Let us clarify some terminology for the game:

- **Target** is the player to whom cards are played.
- **Attacker** is a player that plays cards to the target from their own hand.
- **Killing** is the target's way to get rid of cards that have been used in an attack. Card A can 'kill' card B, if card A has a higher rank than card B and they are of the same suit or if card A is trump and B is not.
- **Trump suit** is the suit of cards, that can kill any card despite its rank, unless the card to kill is also of trump suit.
- **Bout** is the set of turns taken by players where the target does not change. The first move of a bout is the initializing play by the player on the target's right-hand side. The last move of a bout is when no attacker wants or can attack, and the target chooses to end the bout.
- **Table** is the place where all the cards played on the current bout are. It consists of not-killed and killed cards (which also contain the cards that were used to kill).

Each player is dealt 6 cards at the beginning of the game. The first target is chosen randomly (or the last game's loser is the first target). The player on the target's right-hand side initiates the game by playing cards to the target such that: There are at least two cards of the same rank for each unique rank unless only a single card is played. For example, valid initial plays could be: $(\heartsuit 2)$ *or* $(\spadesuit 14)$ *or* $(\spadesuit 5, \clubsuit 5)$ *or* $(\heartsuit 8, \spadesuit 8, \diamondsuit 8, \spadesuit 10, \clubsuit 10)$.

After the player on the target's right-hand side has attacked, every player can freely (non-sequentially) attack, given that a card with the same rank is already on the table, and the number of not-killed cards on the table is not greater than the number of cards in the target's hand. If any attacker attacks, they must always fill their hand to six from the deck, if there are cards in the deck.

The target can also attack themself if there was deck left when the bout began, even if that would mean there are more not-killed cards than there are in the target's hand. The target does not fill their hand to 6 cards until they end the bout.

The target can freely kill not-killed cards. The card can either be played from the top of the deck or the target's hand. If the card is played from the deck, and it can't kill any card on the table, the card is left to the table as a not-killed card, and the target can't play from the deck again until they kill the card with a card from their hand.

When the target doesn't want to kill any cards on the table (or there are none), and none of the attackers want to attack anymore, the target can end the bout by picking the not-killed cards (can be empty), or picking every card on the table. After ending the bout, the target fills their hand to 6 cards from the deck if they have less than six cards and there are cards in the deck. If the target doesn't lift any cards from the table one rotation is performed, so the player on the target's left-hand side is the new target, and the target becomes the new initial attacker. Two rotations are performed if the target lifts any cards from the table, so the second player from the target's left side becomes the target.

The objective of the players is to not be the last person with cards in their hand. The specifics of the rules are fairly complex, but the general idea is quite simple.

1.1.2 Complexity

Game tree complexity is the size of a tree search needed to evaluate the initial position, and can be estimated by calculating the average number of possible moves from a position (branching factor) and the average length of the game. For example, the estimated game tree complexity of Chess is $35^{80} \approx 10^{123}$ [1].

Moska is fairly complex but isn't comparable to chess in terms of game tree complexity. From simulations, we observed that the game tree complexity goes down, as the number of players is increased, which is expected. Figure 1 shows the game tree complexity for 2

to 8 players.

Four-player games, which are the focus of this thesis, had an average of 81 moves per game, with a branching factor of 5.4, giving a game tree complexity of $5.4^{81} \approx 10^{59}$, which is a lot, but nothing compared to chess' 10^{123}. The complexity of two-player Durak, on the other hand, was calculated to have a branching factor of 2.17 and an average length of 44, which results in a game tree complexity of $2.17^{44} \approx 6 \times 10^{14}$ [2].

The complexity results from the simulations do not account for the non-sequential nature of the game, where often multiple players want to play cards, but only one of them can, which would increase the complexity. Also, these results are hardly comparable, since for example four player Moska has up to $\binom{52}{6} + \binom{46}{6} + \binom{40}{6} + \binom{34}{6} + 28! \approx 10^{29}$ different starting configurations, whereas chess has just one.

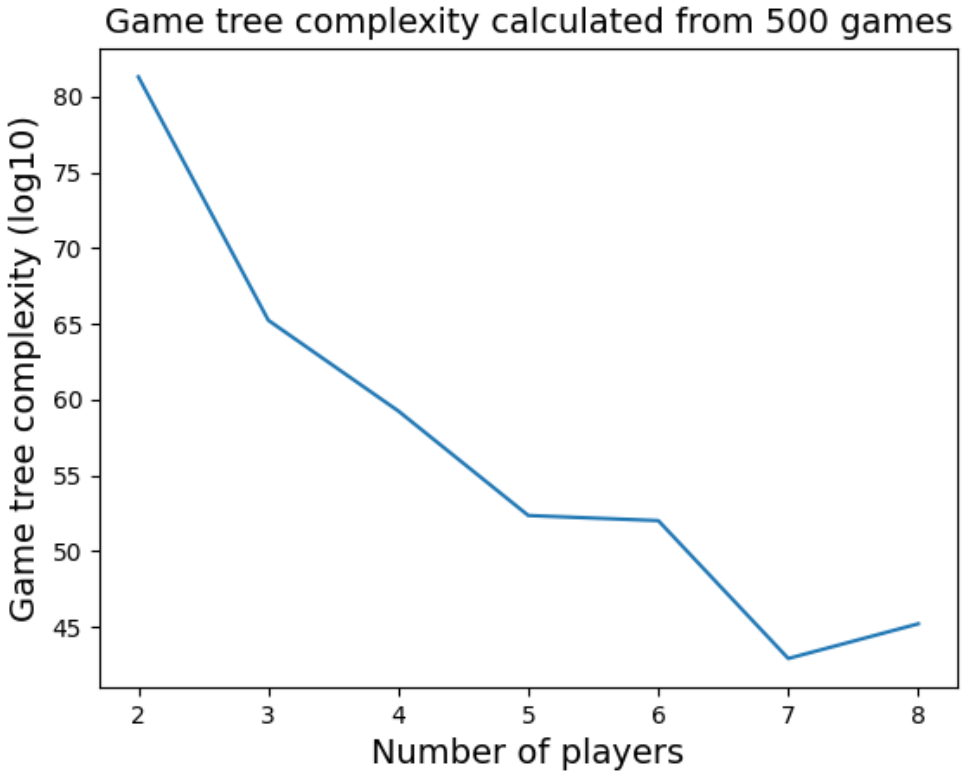

Figure 1. The game tree complexity of playing Moska with different amounts of players. Two-player Moska has a game tree complexity of $\approx 10^{81}$.

1.1.3 Literature

There are no publications about Moska, but we identified studies about Durak. Most notably, about analysis and playing algorithms by Zarlykov (2023) [2] and complexity of Durak by Bonnet (2016) [3].

Zarlykov compared the performance of different heuristic evaluations, Monte Carlo, and Minimax methods in playing Durak. It also discusses analytical and logical results of Durak. For example that a game must terminate, and hence the same state can not occur twice in a game.

Bonnet showed that in perfect information (PIF) two-player version of Durak, whether an attack was defensible, was NP-hard to solve, and generalized PIF two-player Durak was polynomial space (PSPACE) -complete, similarly to Chess and Go.

1.2 AI in games

Humans like playing games for a variety of reasons. Some play for enjoyment, others competitively, and some for socializing. In computer science and mathematics, games are studied because they provide a set of rules, clear goals, and are often hard to model. The strategies used by players can be very complex and require abstract thinking. Machines have a hard time thinking, so often they have to decide moves by some combination of learning to evaluate game states, searching the game tree and simulating games. Multiple game problems are computationally complex making games appealing to humans and a fruitful test bed for studying algorithms and Artificial Intelligence (AI) [4].

In the early days of computer science playing games was often done with a hand-crafted algorithm, because it wasn't possible to perform complex optimizations, or run multiple simulations. Nowadays, AI in games is often about searching the game tree and simulating games. The most common algorithms are Minimax and Monte Carlo. Perhaps the simplest version of the Monte Carlo tree search, called pure Monte-Carlo tree search, plays some number of random games from the current state to termination for each legal move, and chooses the move which led to the highest number of wins. Minimax on the other hand, in its simplest form, searches all possible combinations of moves up to some number of moves ahead and then evaluates the state of the game in each of the nodes and chooses the move resulting in the best evaluation. In these simplest forms, these algorithms can be very slow in complex games, and often different methods are used to

reduce the game tree to make better decisions with fewer computations.

1.2.1 State evaluation functions

A state evaluation function is a function, that provides an evaluation of a state with respect to a goal. As humans, we naturally evaluate how good some positions are. In popular culture, a famous state evaluation happens in "Star Wars - Revenge of The Sith", where during a duel against Anakin, Obi-Wan declares that he has "the high ground" because he physically reached a higher positioning and consequently had the figurative "high ground", i.e. a better position. In the context of humans and games, Holding et. al (1982) observed that a distinguishing ability in good and bad chess players was the efficiency and ability to evaluate a game state [5].

Evaluation functions can be hand-made functions, optimized functions, or be based on simulations. These strategies can be used in conjunction with each other to complement each other's flaws. For example, OpenAI used a neural network to get an initial approximate evaluation for states to better target an MCTS tree search in Go [6].

Hand-made evaluation functions are game specific and can be hard to make. A simple hand-made evaluation function can be made by choosing a set of features $\boldsymbol{S}$ and corresponding weights $\bar{w}$ for the features. A simple approach is then to evaluate a state $\bar{s}$ (values for each selected feature in $\boldsymbol{S}$) as a linear combination of the state and respective weights $\bar{s} \cdot \bar{w}$.

Evaluation based on simulations is a strategy that performs so-called roll-outs and uses results from multiple simulations (from the given state to the terminal state) to evaluate the state. For example, the evaluation of a state can be the number of games won when simulating random play to a terminal state between players, starting from some state. There are multiple different ways to do and prioritize the simulations and infer knowledge from them. This class is known as Monte-Carlo Tree Search (MCTS) in games [7]. One of the simplest ones is pure Monte-Carlo tree search, which for some random turn games (for example Hex) converges to optimal play as the number of simulated games tends to infinity [8]. It is not clear whether it converges to optimal play in Moska.

Neural networks are also frequently used to computationally evaluate a position. Such neural networks can be trained by supervised learning from a dataset of (possibly human) games. The neural network learns to associate positions as either a winning position or a

losing position.

1.2.2 Tree search optimization

Tree search algorithms are used to look ahead and evaluate the possible future states deeper in the game tree to account for an imperfect evaluation function and the *horizon effect* where moves that appear good at first sight, are not good when evaluated further [9]. This is a problem especially with hand-crafted evaluation functions, for example threatening a king in chess is good, but bad if you get eaten on the next turn. Traditional tree-search methods, such as minimax, require an evaluation function to provide information at the leaves of the tree-search. A notable use of minimax, with a hand-made static evaluation function is the Stockfish chess AI [10].

In this study, a greedy algorithm (where depth is one) is used instead of a tree search, due to the large complexity of tree search in hidden information games [11] and the computational simplicity of this method. This method can be thought of as approximating the results of Monte-Carlo rollouts, where the neural network gives an approximation of how often the player will win given a position, without actually having to simulate the games.

1.3 Neural networks

Artificial Neural networks (NN) are a collection of connected nodes and weights. In this thesis, we use supervised learning with a feed-forward deep neural network to evaluate how good a player's position is from their perspective.

In a feed-forward neural network, data flows only in one direction, and there is no recurrence. Each input tensor goes through the network, and the network gives an output tensor. To predict a Moska state, the input is a vector with 442 elements, representing a game state from the perspective of some player (See 2.2.1), and the output is a real value between 0 and 1 describing how good the position is from the player's perspective.

Training a neural network by supervised learning is done with multiple examples. An example consists of some measured features, and the observed value of the target variable (can be a class or a real value). A measurement is given to the neural network, and the error between the model's output and the observation is measured with a loss function. The weights in the neural network are then updated with an optimization algorithm

minimizing the loss function, typically some version of gradient descent.

Most neural network optimization methods adjust the weights based on batches of data to speed up and stabilize the optimization process. Since neural networks are not the focus of this paper, we only introduce these concepts for single sample optimization, which is the core concept.

Convolutional neural networks (CNN) [12] have been more or less the standard type of NN to evaluate a game state. CNNs have been succesfully used for example in Chess [13] [14], Go [15] and Hex [16]. In this study, CNNs are experimented with, but no improvement was observed compared to regular deep neural networks.

1.3.1 Loss function

The loss function is the function, that measures how incorrect the model's prediction is with respect to the observed value. In this thesis, we use the binary cross entropy loss (also known as a logarithmic loss) function, which is often used for binary classification [17]. The formula for binary cross entropy is given by:

$$L(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})] \tag{1}$$

where y is the true label (0 or 1) and $\hat{y}$ is the predicted probability of the positive class (range from 0 to 1). The binary cross entropy loss penalizes predictions that are further from the true labels with higher values, illustrated in Figure 2.

1.3.2 Stochastic gradient descent

Stochastic gradient descent (SGD) is a stochastic approximation of gradient descent optimization. In gradient descent (GD), the gradient is calculated for the entire dataset, but in SGD an approximation of the actual gradient is calculated based on a single sample. Stochastic gradient descent calculates the effects of each neuron's weights and adjusts them based on how much they affected the calculated loss. There is also a parameter called 'learning rate' associated with optimizers, which is a real value between 0 and 1, used to adjust the step size taken towards the gradient. The SGD update rule for the

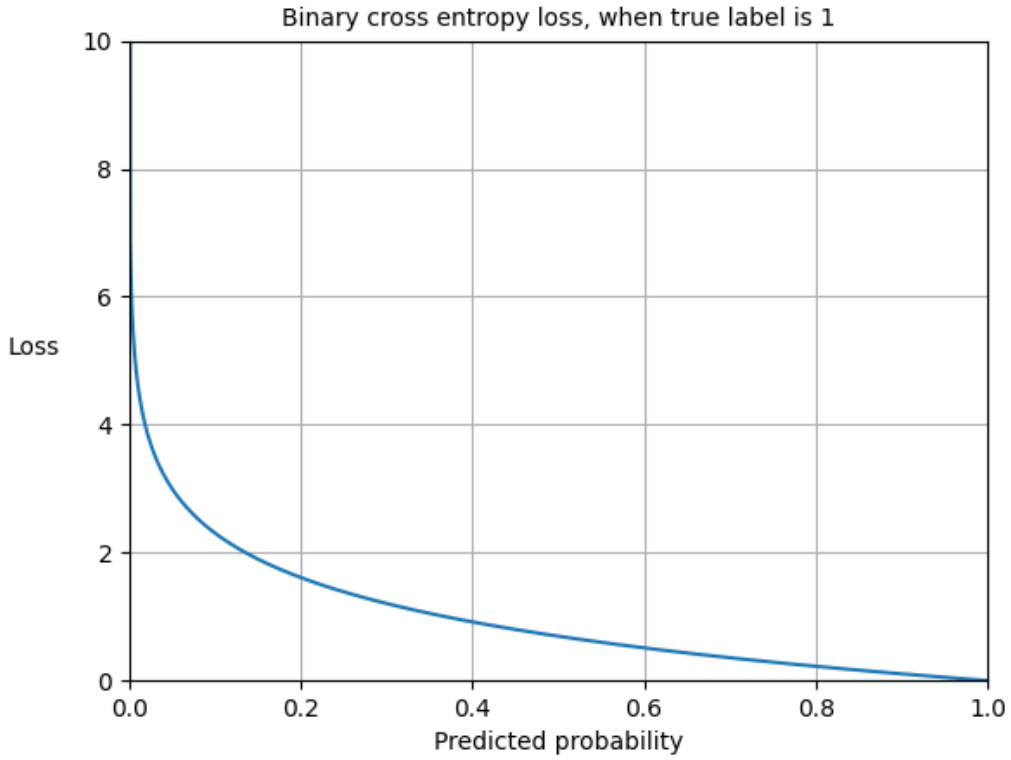

Figure 2. Binary cross-entropy loss (equation 1), when the actual label is 1 ($y = 1$).

weights W at iteration t is given by:

$$W_{t+1} = W_t - \alpha \nabla J(W_t, x_i) \tag{2}$$

where α is the step size and $J(W, x_i)$ is the loss function evaluated on a single training sample x_i.

SGD is an integral part of most neural network optimization algorithms (optimizers), albeit with multiple different variations.

Optimizer is the algorithm used to adjust the weights of the neural network to minimize the selected loss function during training. There are multiple available optimizers. In this thesis, we use the Adam (Adaptive Moment Estimation) optimizer, since it is a powerful, widely used optimizer [18]. Adam uses a combination of stochastic gradient descent, exponential moving average, and adaptive learning rate to tune each weight of the neural network.

1.3.3 Regularization

Regularization is frequently used in machine learning to balance the trade-off between variance and bias. A high variance (overfit) means that a model has too closely fit the training data (e.g. 10th degree polynomial on 10 data points), and a high bias (underfit) means that the differences between the model and training data are large (e.g. linear fit on quadratic data). Both mean, that the model will do poorly on new data. Regularization makes it possible to use more complex models, which can better capture complex relationships, without the model overfitting. Overfitting is not usually a problem if the training dataset is large enough with respect to the complexity of the underlying relationships. Overfitting can usually be mitigated by regularization methods, such as L1, L2, dropout, and many others [19].

Shortly, L1 and L2 regularization add a term to the loss function, that is proportional to the regularized weight thus discouraging large weights, while a dropout layer randomly sets weights to 0, making the model depend less on single connections.

2 METHODOLOGY

In this section, we introduce the methodology used to simulate games, gather data, and train the neural network. We also apply the methodology to the game of Tic-Tac-Toe to show the overall methodology in a simple game. The general approach is to use a multi-agent simulation with hand-crafted agents and gather data from the simulations. In this thesis, hand-crafted agents are used instead of starting from random play to essentially skip some of the early iterations when starting from random play. The gathered data contains perfect card counting information, including which cards each player has publicly lifted to their hand, and which cards have been discarded from the game. An evaluation function is then modeled to the gathered data, where each state is labeled with 1 (didn't lose) and 0 (lost). The evaluation function is then implemented to an agent using a greedy algorithm, which evaluates each possible next state of the game (to a depth of one), and plays the move resulting in the highest state evaluation.

2.1 Software

The simulation is a parallelizable multi-agent system implemented in Python. The simulation implements a game engine and different player interfaces. The used software is open source and available in GitHub [20]. A simplified UML graph can be found in Figure A1.2.

Simulation speed can be anywhere from 0.1 - 10 games per second on a single Intel(R) i5 core (12th Gen), largely depending on the parameters of the simulation. The simulation scales well, and has good CPU core utilization as seen in Figure 3.

The software is oriented towards simulations, rather than playing as a human. The effect of a player's speed is not studied in this thesis, and players can't affect when they play, because the turns are assigned randomly. The game engine works so that each player will get a turn before the current target can end the bout if after they have last played something, another player has made a move other than skip. The only known exception to the regular rules in Moska is that players can only play to themselves if there is deck left. In normal Moska, players can play to themselves also, if the deck ran out when they are the target, until the end of their role as target.

The software allows for 2-8 player games, but in this thesis, we only discuss four-player

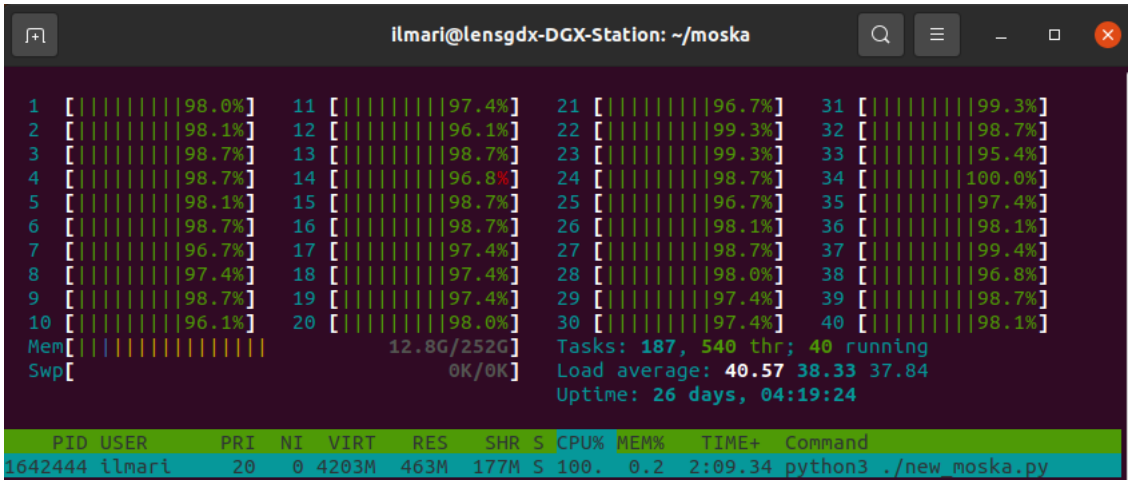

Figure 3. CPU utilization when running 40 simulation process batches on 40 Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz processors.

games, unless otherwise specified. Four-player games are selected, as a middle ground between the length of the preparation phase, and the end game (See Appendix 2.4).

Next, we discuss the different interfaces that are used to implement agents to the game engine.

2.1.1 AbstractPlayer

The **AbstractPlayer** interface is the base class for all playing agents. It implements the necessary logic to use an agent with the game engine. Subclasses must implement logic to choose which class of moves to play (for example, play to self, kill from hand, etc.), and the logic to select the cards to play after deciding which class of moves to play. The AbstractPlayer interface allows full customization of the playing logic and possibly very fast simulations. For example, MoskaBot3 (Bot3) uses this interface.

2.1.2 AbstractEvaluatorBot

The **AbstractEvaluatorBot** (MIF) interface implements logic to generate the possible legal moves, with fairly efficient combinatorial- and graph algorithms. This implements the greedy logic (i.e. no look-ahead) of the main approach. This interface only requires an instance to implement an evaluation function, which takes in a list of game state objects and returns the evaluations corresponding to the states. If this interface is used directly,

the agent will have extra information about the future state of the game. For example, when thinking of attacking the target, this interface gives extra information to the evaluation function about the next state, which a human wouldn't have. A similar interface is the **AbstractHIFEvaluatorBot** (HIF), which hides the excess information by sampling several possible immediate next states for each move and using the average evaluation of those samples as its evaluation for the move. The MIF interface is faster and designed for simulations, but has more information than a human would making the game unfair.

Two important classes, that use this logic, are NNHIFEvaluatorBot (NN-HIF) and NNEvaluatorBot (NN-MIF). They are designed to use a neural network to evaluate a state.

An agent using either one of these interfaces has a `max_num_states` parameter (defaults to 1000), which controls the maximum number of different moves to consider. The generation of all possible moves has an exponential worst-case complexity and it would be infeasible to create all the possible plays and evaluate them.

The HIF interface also has a `max_num_samples` parameter (defaults to 100), which controls the number of random samples to sample from the deck to create the average evaluation of a move.

Algorithm 1 Move Selection Logic of NN-HIF Player: In this algorithm description, *MV_IDS* contains the identifiers of all classes of moves, the *get_possible_moves(move_id)* returns all the legal moves that the player can make that belong to the class of *move_id*, *get_next_states(move)* returns a list of the possible next states after playing a move, and *evaluate_state(state)* returns an evaluation of the state. The player plays the move that results in the highest evaluation (or highest average).

$best\_moves \leftarrow \{\}$
for $move\_id$ in MV_IDS **do**
    $moves\_to\_scores \leftarrow \{\}$
    for $move$ in $get\_possible\_moves(move\_id)$ **do**
        $states \leftarrow get\_next\_states(move)$
        $eval \leftarrow 0$
        for $state$ in $states$ **do**
            $eval \leftarrow eval + evaluate\_state(state)$
        end for
        $eval \leftarrow eval/len(states)$
        $moves\_to\_scores[move] \leftarrow eval$
    end for
    $best\_move, best\_eval \leftarrow max(moves\_to\_scores)$
    $best\_moves[move\_id] \leftarrow (best\_move, best\_eval)$
end for
$Play(max(best\_moves))$

2.2 Data collection

In a game, each player records all the game states from their perspective. After each game, all the player's recorded states are gathered. All the loser's states are taken, and the same amount of randomly selected states from not-losers are selected to reduce data and balance the data set labels. The states are then shuffled, and each state is written to a randomly named file in a numeric vector format, along with a binary value (the label) representing whether the player lost (0) or didn't lose (1).

Two iterations of neural networks were trained, and hence two datasets were created. Both datasets were created by randomly selecting agents and their hyperparameters to reduce bias in the dataset. The used agents ranged from random play to best play so far (Bot3 for the first dataset and NN-MIF with a neural network based on the previous dataset for the second dataset). The datasets had 50 M labeled states, gathered from 1M games.

2.2.1 Vector representation

The state vector contains all the information about the current state of the game from the player's perspective. The vector contains general information, for example, who is the target, and what is the trump suit, as well as lists of cards. A single list of cards is encoded as a vector of length 52, where each index corresponds to a specific card in a reference deck, which is sorted from values 2 - 14 and suits in order ♣, ♢, ♡, ♠. Each index in the card vector is marked as 0 if the card is not known or is not in the game and else it is marked as 1.

The perspective vector includes:

- Number of cards still in the deck (including trump card at the bottom) : integer
- How many cards each player has in their hand : 4 x integer
- Which players have played (and won't play again until someone plays a move other than skip) : 4 x boolean
- Which players are still in the game : 4 x boolean
- Whether there is a card played from the deck in unfallen cards : boolean
- Which player is the target : 4 x boolean

- Which suit is the trump suit : 4 x boolean
- Whose perspective this vector is : 4 x boolean
- Which cards are still in the game : 52 x boolean
- not-killed cards : 52 x boolean
- killed cards, and cards used to kill : 52 x boolean
- Each player's known cards; the cards the player has publicly lifted from the table : 4 x 52 x boolean
- The player's cards : 52 x boolean

This vector representation is simple, and there is no information, that a real player couldn't have. There are no implied connections (for example encoding cards according to their strength) in the vector. In chess, this type of encoding (bitmap) has led to better chess evaluations than algebraic representation, where the pieces are encoded according to their strength [14].

2.3 Neural network

Multiple neural network architectures were non-systematically experimented with. The dataset was split into train/validation/test with 92/6/2 % split and the models with the lowest binary cross entropy (equation 1) loss on the test dataset were benchmarked to see the percent of games they win.

The neural networks were very prone to overfitting the noise in the data since it is impossible to know whether a player will win in the early stages of the game. To address these issues of noise and bias, a very large batch size of 16384 was used in an attempt to essentially cancel out the noise.

To gain 100-fold performance improvements on neural network inference and to improve parallelism, the trained TensorFlow models were converted to TensorFlow Lite models.

2.4 Example on Tic-Tac-Toe

In this section, we discuss the results of this methodology, when applied to the game of Tic-Tac-Toe, and the lessons learned and applied to Moska.

For Tic-Tac-Toe this method was applied from scratch, starting from random play. For each iteration, 5000 games were simulated between a random player, and a neural network player (initially random), and a neural network was trained on data from the simulations. Only data from the most recent iteration was used.

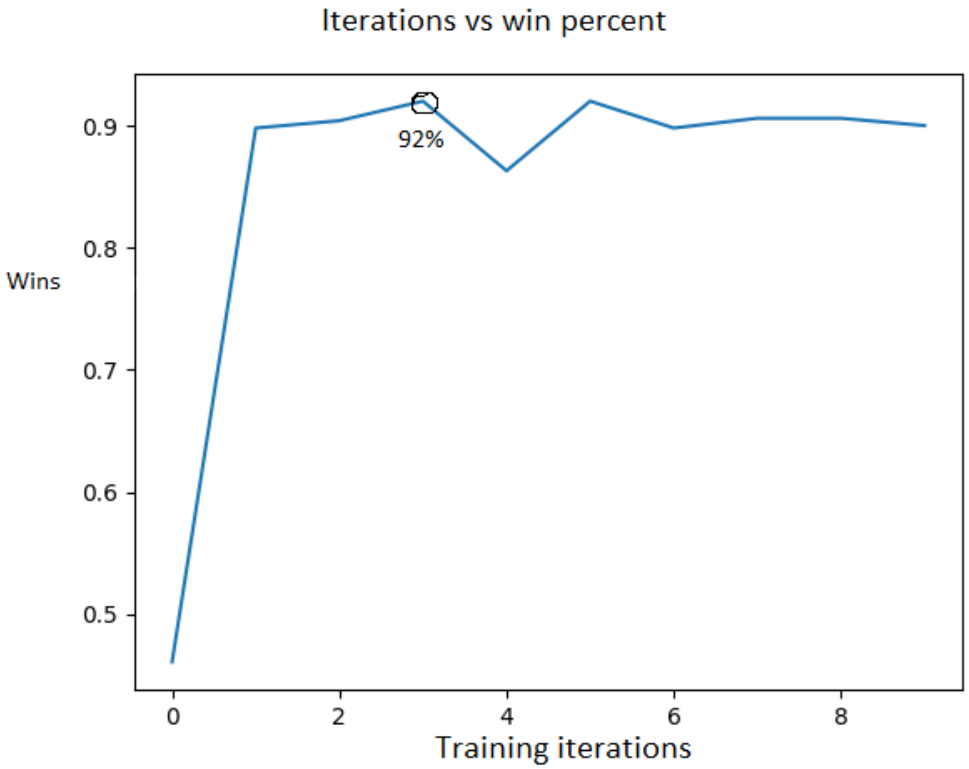

Figure 4. The win portion of the trained Tic-Tac-Toe model against random play after some amount of iterations.

In Tic-Tac-Toe, a CNN highly outperformed a regular NN. The highest win percentage against random play was 92% with two convolutional layers and 2 dense layers, and only 65% for 4 dense layers.

It was also noticed, that adding randomness when gathering data is essential for good performance because it reduces bias in the dataset. Hence in each iteration, the trained model played a random move 20% of the time when gathering data. If no randomness is added,

the new dataset will be biased and the model will have easy exploits, since the trained model only makes certain moves and there is no exploration to discover its weaknesses. For Moska, this exploration was achieved by varying the agents and parameters of agents in the dataset.

It is important to note, that we only applied this approach to Tic-Tac-Toe to demonstrate it's usage. We didn't vary many parameters, try multiple different neural networks, and only used a small number of games. Undoubtedly, this approach can create stronger agents to Tic-Tac-Toe than observed here.

3 RESULTS

In this section, we present results from training the neural network and the performance results of implementing the neural network on an agent for the card game of Moska.

3.1 Neural network performance

A neural network's performance is initially measured by the loss on a test dataset, and later by using it as an evaluation function on the AbstractEvaluatorBot (MIF) and AbstractHIFEvaluatorBot (HIF) interfaces to create a playing agent.

Surprisingly, the traditional neural networks, with no convolutions, consistently outperformed the more sophisticated methods.

The neural network architecture had a batch normalization layer, 4 hidden layers, and three dropout layers with rates of 0.4, 0.4, and 0.35 after the first three hidden layers, totaling roughly 1 million parameters (See Figure A1.1). The default learning rate of 0.001 seemed to overfit too quickly, and so 0.0001 was chosen along with a very large batch size of 16384.

Going forward, we will use these names for agents using the MIF or HIF interface with the evaluations provided by a neural network with the aforementioned architecture:

- **NN1-MIF** and **NN1-HIF** use the first iteration of the trained neural network (NN1). NN1-HIF uses the HIF interface (HIF referring to 'hidden information) and NN1-MIF uses the MIF interface (MIF referring to 'more information').
- **NN2-MIF** and **NN2-HIF** use the second iteration of the trained neural network (NN2), so the data used to train this model contained simulated games with NN1-MIF.

The NN2 neural network could predict the winner from the test dataset with 73.5% accuracy and 0.504 binary-cross-entropy loss, containing states anywhere from the beginning of the game (hard to predict) to the end of the game (easier to predict).

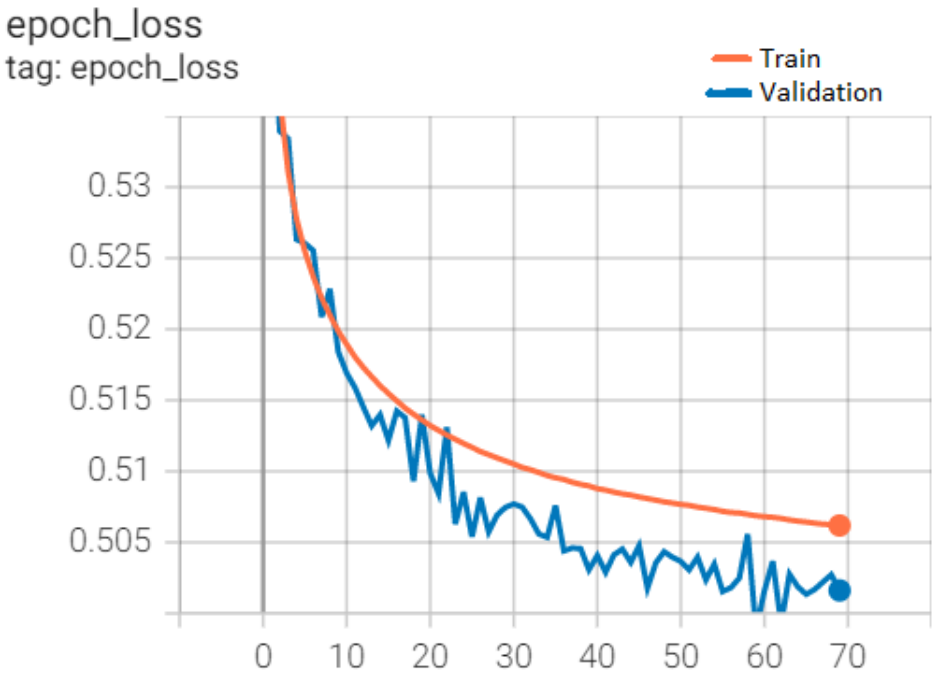

Figure 5. Loss and validation loss during the training process of NN2 for 70 epochs. Due to the dropout layers, the loss on the validation dataset is lower than the loss on the validation dataset. Further training did not reduce validation loss anymore.

3.2 Agent performance

The performance of an agent is assessed with three methods: standardized benchmarks, games against humans, and individual game analysis.

3.2.1 Benchmarks

Benchmarking is a fast and objective measure of an agent's performance. A benchmark consists of a set of predefined agents, either hand-crafted or previously trained models. The agent to benchmark is defined, and 2000 games are simulated between the predefined agents and the agent to benchmark. The benchmark then reports the percent of losses the benchmarked agent had.

The agents in the benchmarks, and the agent to be benchmarked, used the default `max_num_states` (1000) and `max_num_samples` (100) if applicable to the agent.

The first benchmark has three of the strongest *hand-crafted* agents (Bot3). The third benchmark has three NN1-MIF agents, so they have some additional information making

them stronger. The third benchmark has one agent from the first and second benchmarks and a weaker hand-crafted agent. The fourth benchmark only has random players.

The loss percentages for some agents in each benchmark can be seen in table 1. From the table, it can be seen that both NN1 agents are better than Bot3, which was the best agent used in training the first iteration, and respectively both NN2 agents are better than NN1-MIF, which was the best agent used in the training of NN2.

Table 1. The percent of lost games out of 2000 played games on each benchmark for some players. On each benchmark, there are four players, and one of the players plays against the benchmark. Hence, if the benchmarked player is as good as the other three players, the expected portion of games the player loses is 25%.

Player	Benchmark 1	Benchmark 2	Benchmark 3	Benchmark 4
NN2-MIF	13.65	15.35	12.65	0
NN2-HIF	14.25	18.45	15.65	0.05
NN1-HIF	23.8	26.3	21.15	0.25
Bot3	25.15	27.55	24.75	0.4
Random	90.75	90.2	87.7	24.95

3.2.2 Tests on humans

Tests on humans reveal the actual performance of the trained NN models, and whether *Team machine*, or humans win this battle.

Tests on humans were conducted against the NN2-HIF model. The tests were conducted by creating a simple website, where people could go and play against the agents. The humans could play anonymously, or by registering and rating their Moska skills as beginner, intermediate, or advanced before playing.

Unfortunately, the period to allow playing was poorly timed due to student events and exams, and only seven unique people played a total of 260 games. 85% of the games were played by self-rated 'advanced' players.

From all the games, humans lost 29.8 % of the games, indicating that the NN2-HIF agent is stronger than even advanced human players! Due to the relatively small number of games played, we estimate the 95% confidence interval for the true percentage of human losses to be between 25% and 35% with the Agresti-Coull formula [21].

3.3 Game analysis

The NN2-HIF agent makes very reasonable moves and employs many of the strategies used by human players such as gathering combinations, luring other players to attack with high cards at the preparation phase, delaying possible plays to have opponents make decisions with less information, sparing high combinations and trump cards until the end, preferring to kill with ranks that are already on the table, as well as using all standard possible moves at the agent's disposal.

Contrary to our own strategies with Moska, individual agents seem to preferentially bully people with weaker hands, most likely to weaken one opponent and ensure at least one player has a worse hand. Sometimes the agent plays unlike a human if they have a basically guaranteed win or loss because the evaluations for any move are equal - it isn't in a hurry to lose or win.

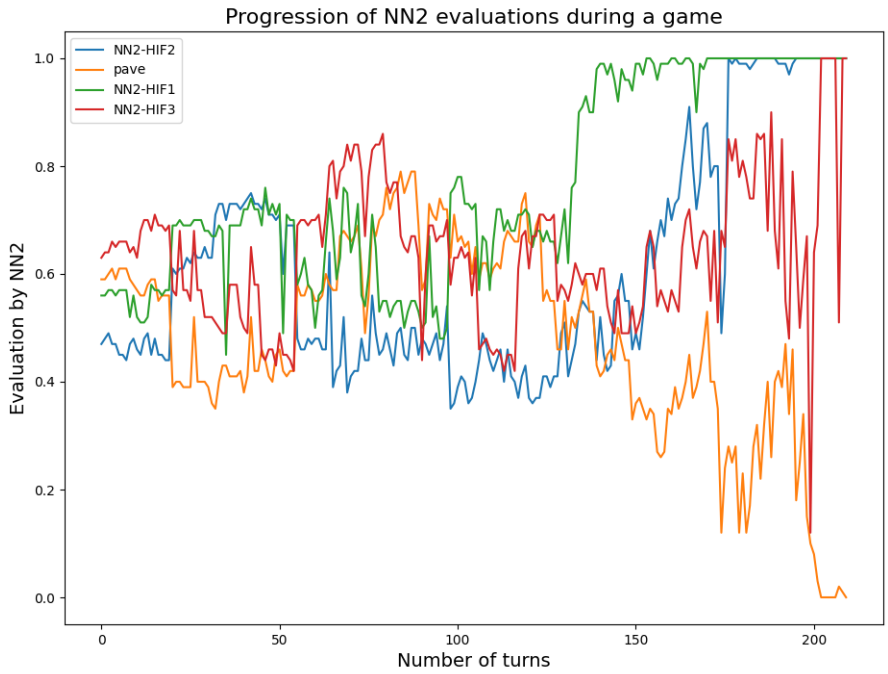

Figure 6. Player evaluations as a function of the number of turns played. In this figure, the line labeled 'pave' is a human player. The human player has a good initial hand, but blunders in the face of superior AIs.

We also spoke with the people who played against the agents and asked them for feedback

about the agent's behavior and the overall playing experience with Moska. The participants hadn't found any consistently exploitable playing styles (as is also evident from the loss percentage) and generally considered it a well-made and a realistically playing agent. They also didn't report any inconsistencies about the rules or the general setup, except for the minor rule deviation discussed in 2.1 and a rare rule exception that is used in some versions of Moska, which allows people to attack from the deck.

4 DISCUSSION

4.1 Improving results

There are many possible improvements for the agent, so we limit ourselves to only consider improvements that would be generalizable and not limited to a particular game.

The single most influential addition would likely be the addition of a tree search that traverses the game tree further than one move ahead. For example, the raw neural network behind AlphaGo Zero had an Elo rating of just 3055, but with the addition of MCTS it reached a staggering 5185 rating [6]. Both minimax and MCTS methods should work for Moska, however, it is not clear how to implement a minimax tree search for Moska due to the hidden information.

Also, more data, more iterations, and trying different neural network architectures would very likely improve the evaluation function, and thus agent performance.

There was also a very minor flaw discovered in the distribution of turns in the game engine, which effectively increased the effect of chance on the result of the game. This lead to worse players doing slightly better, and good players doing slightly worse. The results described in this thesis do not take this fix into account, but it is reasonable to assume (and preliminary results show) that humans do worse when this issue is fixed, since humans are worse than machines, and thus beneficiaries of the bug.

4.2 Generalization to other games

Our approach can be used for any number of players and many other games. It is also versatile and robust for some rule changes since it simply evaluates a game state and doesn't explicitly care what are its current choices. Implicitly it does matter, since a model trained on games with some set of rules, assumes other players follow those rules.

This method can be applied to games with discrete options, but for games, where the options are continuous some conversion is required. For example, the betting amount in Poker is a continuous value, and to use this method the betting amounts should be bundled, where for example only bets with a $100 difference are evaluated. Furthermore, this method requires the generation of possible moves, which can be computationally very

complex and programmatically hard. In the game of Moska generating initial plays, or kill combinations from hand both have exponential complexity and use a form of depth-first search.

4.3 Conclusions

The literature on multiplayer hidden information games is not vast, and successful applications are still quite rare. This thesis shows an approach to creating agents for multiple types of games, including hidden information and multiplayer games, by creating a static evaluation function from simulations without requiring human expertise about the game, or data about human games.

Our application of the method to the multiplayer card game of Moska is successful and can outperform even advanced human players.

Our method can be used as a standalone method to purely evaluate each possible next state, or as an assisting method in conjunction with other methods, for example as an initial guide to direct a tree search.

As is ever more common, we bow to Team Machine for winning this battle, and as humans, we prepare to bow down to our ever more capable AI overlords.

REFERENCES

[1] Louis Victor Allis et al. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.

[2] Azamat Zarlykov. Artificial intelligence for the card game durak. 2023.

[3] Édouard Bonnet. The complexity of playing durak. In *25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, pages 109–115, 2016.

[4] Aviezri S Fraenkel. Complexity, appeal and challenges of combinatorial games. *Theoretical Computer Science*, 313(3):393–415, 2004.

[5] Dennis H Holding and Robert I Reynolds. Recall or evaluation of chess positions as determinants of chess skill. *Memory & Cognition*, 10:237–242, 1982.

[6] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[7] Jean Mehat and Tristan Cazenave. Monte-carlo tree search for general game playing. *Univ. Paris*, 8, 2008.

[8] Yuval Peres, Oded Schramm, Scott Sheffield, and David B Wilson. Random-turn hex and other selection games. *The American Mathematical Monthly*, 114(5):373–387, 2007.

[9] Hans J Berliner. Some necessary conditions for a master chess program. In *IJCAI*, volume 3, page 77, 1973.

[10] Shiva Maharaj, Nick Polson, and Alex Turk. Chess ai: competing paradigms for machine intelligence. *Entropy*, 24(4):550, 2022.

[11] Daniel Whitehouse. *Monte Carlo tree search for games with hidden information and uncertainty*. PhD thesis, University of York, 2014.

[12] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[13] Hitanshu Panchal, Siddhant Mishra, and Varsha Shrivastava. Chess moves prediction using deep learning neural networks. In *2021 International Conference on Advances in Computing and Communications (ICACC)*, pages 1–6. IEEE, 2021.

[14] Matthia Sabatelli, Francesco Bidoia, Valeriu Codreanu, and Marco A Wiering. Learning to evaluate chess positions with deep neural networks and limited lookahead. In *ICPRAM*, pages 276–283, 2018.

[15] Chris J Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. *arXiv preprint arXiv:1412.6564*, 2014.

[16] Kei Takada, Hiroyuki Iizuka, and Masahito Yamamoto. Reinforcement learning for creating evaluation function using convolutional neural network in hex. In *2017 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 196–201. IEEE, 2017.

[17] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, pages 1–26, 2020.

[18] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] Claudio Filipi Gonçalves Dos Santos and João Paulo Papa. Avoiding overfitting: A survey on regularization methods for convolutional neural networks. *ACM Computing Surveys (CSUR)*, 54(10s):1–25, 2022.

[20] ilmari99. moska: A repo for simulating moska games, and trying different play algorithms. `https://github.com/ilmari99/moska`, 2023.

[21] James F Reed III. Better binomial confidence intervals. *Journal of Modern Applied Statistical Methods*, 6(1):15, 2007.

[22] Lutz Roeder. Netron, visualizer for neural network, deep learning, and machine learning models, 2017. If you use Netron in your research, please cite it using these metadata.

[23] Moska – korttipeliopas. `https://korttipeliopas.fi/moska`. Accessed: 29.04.2023.

Appendix 1. Images

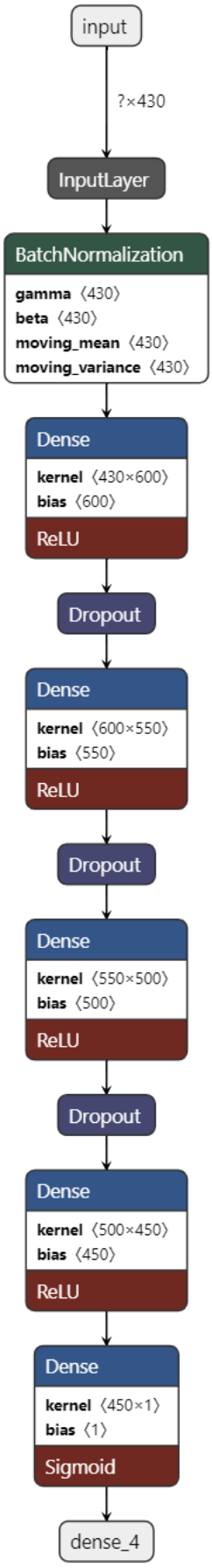

Figure A1.1. The architecture used in both NN1 and NN2 neural networks. Created with Netron [22].

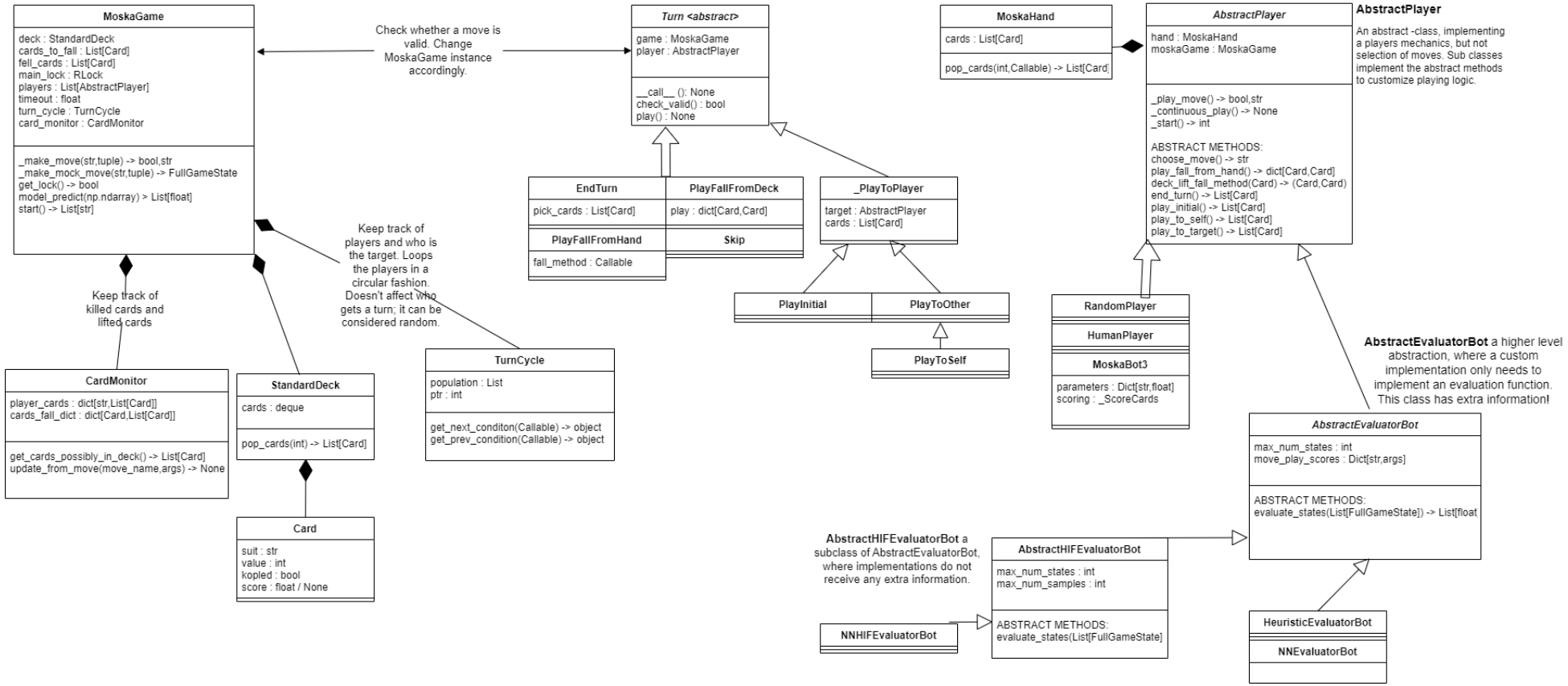

Figure A1.2. A simplified graph of the implemented software; Game engine and players.

Appendix 2. Images

The rules introduced here, are the rules, that are played most commonly at Lappeenranta University of Technology among Computational Engineering students.

The Finnish rules which are described in *'Korttipeliopas'* are slightly different [23].

Here we will use the same terminology as defined in 1.1.1.

2.1 Goal

The goal of the game is to not be the last player with cards in hand. The winners are equal winners, and it only matters if a player is the loser or not.

2.2 Dealing

At the beginning of the game, each player is dealt 6 random cards from the deck. The next card in the deck will be lifted and the suit will be chosen as the trump suit. If a player has the 2 of the trump suit in their initial hand, he can switch it with the trump card of the deck (getting a larger trump card in exchange), before the game starts. The ace is the highest card (14), and 2 is the lowest card.

2.3 Flow of the game

2.3.1 Initiating a bout

At each stage, one player is the target. The player on the target's right-hand side is the initiating player. The initiating player can play any single card, multiple cards of the same rank, or combinations of multiple cards. For example valid moves could be: $(\heartsuit 2)$ *or* $(\spadesuit 14)$ *or* $(\spadesuit 5, \clubsuit 5)$ *or* $(\heartsuit 8, \spadesuit 8, \diamondsuit 8, \spadesuit 10, \clubsuit 10)$.

2.3.2 Attacking

After the bout is initiated, every player can attack the target, but the cards played MUST have the same rank as some card currently on the table.

Appendix 2. (continued)

Attackers can play cards to the target, as long as there are at most as many not-killed cards on the table, as the target has cards in hand.

Each attacker fills their hand to 6 cards from the deck (if there is a deck left) immediately after playing the cards. The target will not fill their hand.

2.3.3 Defending

Killing a card

The target can defend himself by killing a card on the table, with a greater card of the same suit, or any card of the trump suit unless the card to kill is also trump suit.

Playing cards to self

The target can also play cards to themselves to get rid of bad cards and limit the number of cards others can play. The target can ONLY play cards to themselves if there was a deck left at the beginning of the bout. Also, similar to attackers they can only play cards to themself if a card with the same rank already exists on the table.

Playing from deck

The target can also choose to pick the top card of the deck to kill a card on the table. If the picked card can kill any card on the table, the picked card must be used to kill a card. If the picked card can not fall any card on the table the card is left on the table as if an attacker had played it. The player can not pick the top card of the deck if they have already picked a card from the deck during the bout and the card is still on the table as a not-killed card.

2.3.4 Ending the bout

The target can end the bout once every player has confirmed that they do not want to play more cards. At the end of the bout, the target must pick either the not-killed cards (could be empty) or the full table. The target must also fill their hand to 6 cards from the deck if they have less than 6 cards in hand after ending the bout. If the target doesn't have to pick any cards, the player on the target's left-hand side is the new target and the current

target is the new initiating player. If the target has to or chooses to pick cards, the player on the target's left-hand side is the new initiating player. The initiating player is always the player on the target's right-hand side.

2.3.5 Ending the game

When the deck is finished, the game continues as before, except playing from deck or filling is not possible. Also playing to self as a target is not allowed. In this stage, players attempt to play their cards intelligently, so they will not be the last player with cards in hand. The last person with cards in hand is the loser.

2.4 General techniques

The game can be thought of as having 2 phases: The preparation phase and the end-game phase. A game is in the preparation phase when there are still cards in the deck. In the preparation phase, players try to get a good hand for the end game. A good hand is a hand with high cards and combinations. This allows the player to kill cards played to them, and play multiple cards simultaneously to others. Personally, in this phase, there are many variables to consider, and I just try to assess whether a move likely results in a better hand. The evaluation process is purely an approximation based on previous experience. The end of the preparation phase is also important since the trump card is the bottom card on the deck (This is visible to all players). So the player who gets to lift the last card is guaranteed to have the trump card in their hand for the end-game.

An example of a simple evaluation process in the preparation phase:

Q: I could play 2 10s, neither is trump suit. Should I play them to get 2 random cards from the deck?

A: Usually, if I don't have a trump card, I would play them for a chance at a trump card. If I have at least two trump cards, I would probably not play them, since I would rather have a hand with this combination. This also depends on whether 10s have already been played and several other factors.

The end-game phase starts when the deck is empty. In the end-game phase, players try to get rid of their cards. In this phase, more logic, and knowledge from previous moves

Appendix 2. (continued)

can be used (for example which cards a player has lifted), especially when the number of players left in the game is small (2-4) and there are few cards left. Players generally try to assess the future of the game in a tree Search manner. This can be quite hard since there is hidden information and the number of plays is usually too large for humans.

For a simple example,

Q: All other players are finished, except me and 1 other. I have a trump 10, and 2 5s (not trump). The other player has 1 card. He is the target. How should I play my cards?

A: I can only play one card, since he only has one card in hand. If I play a card, that he can fall, then I lose. If I play my trump 10 and he cant fall it, I win, because he has to lift the card, leaving him with 2 cards in hand. I can then play my pair of 5s and win.