A Decision Heuristic for Monte Carlo Tree Search Doppelkopf Agents

Alexander Dockhorn, Christoph Doell, Matthias Hewelt, and Rudolf Kruse
Institute for Intelligent Cooperating Systems
Faculty of Computer Science, Otto von Guericke University Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany
Email: {alexander.dockhorn, christoph.doell, rudolf.kruse}@ovgu.de, matthias.hewelt@st.ovgu.de

***Abstract*—This work builds up on previous research by Sievers and Helmert, who developed an Monte Carlo Tree Search based doppelkopf agent. This four player card game features a larger state space than skat due to the unknown cards of the contestants. Additionally, players face the unique problem of not knowing their teammates at the start of the game. Figuring out the player parties is a key feature of this card game and demands differing play styles depending on the current knowledge of the game state.**

In this work we enhance the Monte Carlo Tree Search agent created by Sievers and Helmert with a decision heuristic. Our goal is to improve the quality of playouts, by suggesting high quality moves and predicting enemy moves based on a neural network classifier. This classifier is trained on an extensive history of expert player moves recorded during official doppelkopf tournaments. Different network architectures are discussed and evaluated based on their prediction accuracy. The best performing network was tested in a direct comparison with the previous Monte Carlo Tree Search agent by Sievers and Helmert. We show that high quality predictions increase the quality of playouts. Overall, our simulations show that adding the decision heuristic increased the strength of play under comparable computational effort.

I. Introduction

Card and board games are a suitable testbed for developing artificial intelligence (AI). They often pose unique problems, which demand specific adaptations of standard algorithms. Our work is motivated by recent successes in the games Go [1] and Poker [2].

The first game, Go, is a 2-player full information game and consists of an enormous state space ($\approx 10^{172}$ [3]). Recently, the program Alpha-Go by Silver et al. [1] reached top-level play and was able to beat the world champion. One component of their work was the prediction and suggestion of expert moves by a neural network. While many other techniques were involved for improving the play strength of the AI, we will focus on the next-move prediction.

Our second motivational example, Poker [4], is a game which combines gambling, strategy, and skill. The probably most famous variant of poker is Texas Hold'em. Here, each player is dealt two cards face down. After that, five community cards are dealt face up in three stages. Before and after each of the stages the players need to estimate their chances of winning the current hand. Without information of their opponents hand, each player needs to place his bet or fold. During the game the player needs to continuously reevaluate his chances of winning to either continue the bet or stop his participation in the current round. Despite possible bluffs, each player interaction results in additional information, which can be used for reevaluation. Nonetheless, first algorithms managed to stand against humans in no-limit poker just recently. Libratus, a poker AI created by Noam Brown and Tuomas Sandholm [2], improved due to new techniques for endgame solving in imperfect information games. Standard techniques for full game analysis involve Monte Carlo Tree Search (MCTS), which were widely applied in previous scientific analysis [5]–[8].

In this work we create an AI for doppelkopf, which is a trick-taking card game for four players. Its popularity is slightly lower than skat, which is a well-studied card game in the AI community [9]–[12]. Doppelkopf not only involves analyzing a larger state space than skat, but includes the unique feature of unknown player parties, which are determined by the cards in each players hands. Normally, the two players holding the queens of clubs play against the remaining two players. As every player only sees his own cards, the teams are unknown, until a player plays his queen or reveals his party through an announcement. Players can make such announcements to increase their risk and reward for winning the current round. During the game it is necessary to collaborate with your teammate, which is why players need to continuously reevaluate the current game state to infer the parties of each player. The combination of a large state space and imperfect information positions our research in line with other popular research topics, such as Go and Poker.

We directly address the work of Sievers and Helmert [13], in which they developed an MCTS agent. While their work already proved to be capable of developing normal level human play, in this work, we extend the simulation phase of the MCTS by using a neural network for expert move prediction. Such a prediction scheme proved to be successful in Go [1]. We want to test if similar techniques can be successfully applied to imperfect information games with large state spaces.

In the next section (Section II) we summarize the most important rules for playing doppelkopf, followed by a review of the work by Sievers and Helmert in Section III. Therefore, we shortly discuss the MCTS algorithm as well as its popular extension upper confidence bounds applied to trees (UCT) first, and secondly give an overview of the actual implementation in the context of doppelkopf. The following Section IV highlights

our process for developing a decision heuristic based on neural networks. Our evaluation in Section V is split into two parts: we first validate the predictive capabilities of our trained nets in Section V-A, and second review the influence of the decision heuristic for simulations in MCTS on the strength of play in Section V-B. We end our analysis in Section VI by shortly summarizing the results of our work, as well as suggesting topics for future analysis.

II. Doppelkopf

In this section we give a short overview on the most important rules for playing doppelkopf. Since there are multiple playing variants we based our work on the standard tournament rules published by the Deutscher Doppelkopf Verband (*German Doppelkopf Association*, DDV) [14]. This summary only contains those rules, which are important for the development of our model.

A set of doppelkopf cards consists of 48 cards. Typically a shortened french deck is used, including four suits, namely clubs (♣), spades (♠), hearts (♡), and diamonds (♢). Each suit consists of 12 cards, precisely two cards per ace (A), ten (10), king (K), queen (Q), jack (J), and nine (9). Each card type has an assigned point value. A nine is worth 0 card pips, jacks are worth 2 card pips, queens 3, kings 4, tens 10, and aces 11 card pips. Therefore, the total sum of card pips is 240. Additionally, cards are ordered into trumps and non-trumps. In a normal game all ♢ cards, all jacks, queens, as well as both ♡ tens form the trump suit. The remaining cards form the non-trump suit and are ordered as follows: aces, tens, kings, nines. Figure 1 shows the order of cards in a normal game.

The game starts by dealing each player 12 cards. Cards need to be hidden from other players and it is forbidden to communicate card ownerships. After the dealing phase, the type of game needs to be settled on. Based on the tournament rules of the DDV two types of games are distinguished, namely normal and solo games, each consisting of a re and a kontra party. In a normal game players holding the ♣ Q form the re party. In case a player has both ♣ Q, he can either play a solo or a marriage. In a marriage one of the other three players joins the re party during the game. Card ownership is usually not known at the start of the game, but played card can be used to infer the player parties.

Due to the generally low frequency of solo games, we do not go into any detail for specific rules. The interested reader is referred to the full description of tournament rules by the DDV [14].

Card pips are gained by winning tricks. Each trick is a playout of one card per player according to the following rules:

- one player starts by playing a card
- clockwise players need to add a card of the same suit
- in case, they cannot follow the played suit (because they do not own an appropriate card) they can choose freely
- the player who plays the highest card wins the trick and starts the next trick

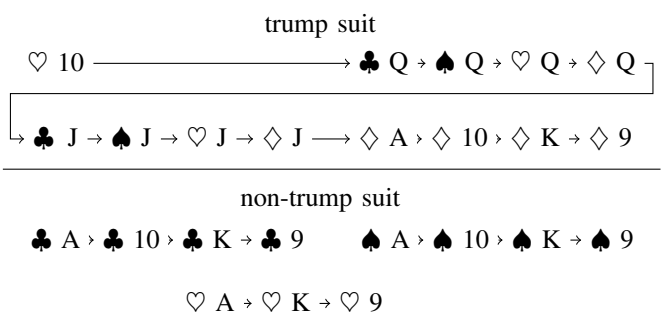


Fig. 1: Card ranks in normal games.

After 12 tricks the game is finished and every team counts the card pips of the tricks they won. The re party has to score at least 121 card pips to win the game. In case the re party does not succeed, the kontra party wins.

The winning threshold can be shifted by making announcements during the first rounds of a game. Making announcements increases the value of the game. Each party can announce that the opponent party will not manage to score 90 (60, 30, 0) card pips. In case the other party reaches the goal, the announcing party loses the game. Additionally, players can announce their party by calling ”re” or ”kontra”.

Winning a game without further announcements is worth 1 point. In case, players announced their party 2 points are scored. Achieving a score of more or equal than 150 (180, 210, 240) card pips awards one extra point per stage. A correct announcement that the opponent party will not manage to score 90 (60, 30, 0) card pips is worth extra +1 (+2, +3, +4) points. Therefore, announcements can hugely increase the value of a game, while bearing the risk of losing the same amount of points in case of a wrong announcement. Additional points are rewarded in case of certain events during the card playing phase, such as

- winning against the elders: winning against the re party
- catching a fox: winning a trick consisting of a ♢ A played by the opposing party
- making a doppelkopf: winning a trick including at least 40 card pips
- winning the last trick with a ♣ J

The goal of our AI is to choose the card that maximizes the chance of winning the game and the value of the win at each trick. Two features of doppelkopf pose unique demands on possible solutions. First, doppelkopf features a large state space, which would be infeasible to be explored at every trick. The start of the game marks the phase with the highest complexity. When all players were dealt 12 cards, the number of possible games can be approximated by Equation 1 [13]:

$$\sum_{i=0}^{48} \prod_{j=0}^{3} \binom{12}{\lfloor (i+j)/4 \rfloor} \approx 2.4 \cdot 10^{13} \tag{1}$$

However, this formula disregards that players do not have full information of their opponents cards. This second feature further increases the number of possible games. We will further

assume that a player received 12 unique cards. Therefore, the second player receives 12 out of 36 cards, from the 24 cards left the third player gets 12 as well and the fourth player gets the remaining cards. This distribution can be modeled by binomial coefficients (ignoring double occurences of cards), which results in a rough upper bound for the number of possible card distributions:

$$\binom{36}{12} \cdot \binom{24}{12} \cdot \binom{12}{12} \approx 3.4 \cdot 10^{15} \tag{2}$$

Combining equation (1) and (2), right at the start of the game the game's complexity is approximately:

$$2.4 \cdot 10^{13} \cdot 3.4 \cdot 10^{15} = 8.2 \cdot 10^{28} \tag{3}$$

This estimate still disregards other game modes, namely solo and marriages, and does not take player's announcements during the game into account. However, it highlights both components, base complexity and lack of information, which need to be addressed specifically in the AI agent.

III. Monte Carlo Tree Search

In this section we will cover the basics of Monte Carlo Tree Search (MCTS) to review the approach by Sievers and Helmert. MCTS is a heuristic search algorithm, which lead to notably successes in the field of computational intelligence in games [15].

In MCTS the agent explores the game tree by (1) selecting a node under consideration (2) choosing any legal move for expanding the search tree (3) performing (random) playouts for estimating the value of this node and (4) updating the value estimate of all visited nodes. Node values are determined by repeatedly simulating games from the selected expansion node till the end of the game. The outcome of such an episode is used for backpropagation over all visited nodes. Hereby, for each node the chances of winning the game by visiting this node is calculated by dividing the number of winning episodes by the total number of episodes that visited this node.

A. Upper Confidence Bounds applied to Trees

During the search phase two important concepts need to be considered, namely exploration and exploitation. Exploratory moves choose nodes, which were only rarely visited during previous simulations. In contrast, exploitation chooses nodes again, which are already known to yield high chances of winning the game, to further increase the confidence in our estimate. In MCTS exploration and exploitation need to be balanced during the search.

The work of Levente Kocsis and Csaba Szepesvri [16] addressed this problem with their proposal of the upper confidence bounds applied to trees-method (UCT). Here, the formula for the upper confidence bounds 1 is used to determine the expansion candidate. During the second phase of MCTS, UCT chooses the node, which maximizes the value given by:

$$\underbrace{R(s')}_{\text{Exploitation}} + C \underbrace{\sqrt{\frac{log(V(s))}{V(s')}}}_{\text{Exploration}} \tag{4}$$

where s' is a child node of s, $R(s')$ is the average success after choosing node s', and $V(s)$ counts the number of visits of state s during previous episodes. We refrain from using the convex combination equation of the UCT score for directly comparable results to the work of Sievers and Helmert.

Parameter C balances between exploration and exploitation of the game tree. Small values favors the first half of the equation, such that a high number of wins per move results in a repeated simulation the same node. Higher values increase the influence of the exploration term, therefore, leads to simulations of rarely explored nodes.

B. UCT for Doppelkopf

For doppelkopf the MCTS faces the problem of not knowing the current state of the game. This includes two sources of uncertainty, (1) the hidden hand-cards of all other players and (2) their moves. The first needs to be assumed for a simulated playout of the game. For this purpose, the implementation by Sievers and Helmert generates multiple possible card distributions and repeats the simulation for all of them. This leads to a better estimate of the current winning chances.

The number of possible card distributions (see Equation 2) is too large for a complete simulation. Nevertheless, the rules of the game can limit the number of possible distributions. The following rules were used for card distribution inference:

- a card that already has been played cannot be assumed to be on a player's hand
- in case a player is playing a marriage both ♣ Qs are on his hand
- in a normal game both players of the re party own one ♣ Q, while the kontra party cannot own this card
- in case a player does not play the current suit of a trick, he cannot have any other cards of this suit. This also takes the trump-suit into account.

Only reliable information is used for limiting considered card distributions. Therefore, at the start of the game cards are assumed to be distributed at random. After several tricks, accumulated information is used to infer our opponent's cards, such that the final moves only need to consider few remaining distributions.

IV. Neural Network-Based Decision Heuristic

The current UCT implementation by Sievers and Helmert tests many card distributions for determining which card should be played. Using random playouts each of these card distributions is in need of multiple simulations to achieve a stable value estimate for each possible card. In our work we want to decrease the number of simulations per card distribution, while achieving similar or better play strength. Therefore, we implement a neural network-based decision heuristic, with the task to replace random playouts by prioritizing high value moves. Figure 2 shows how our approach is integrated in the previous implementation by Sievers and Helmert. In the following subsections we explain the details of our neural network design.

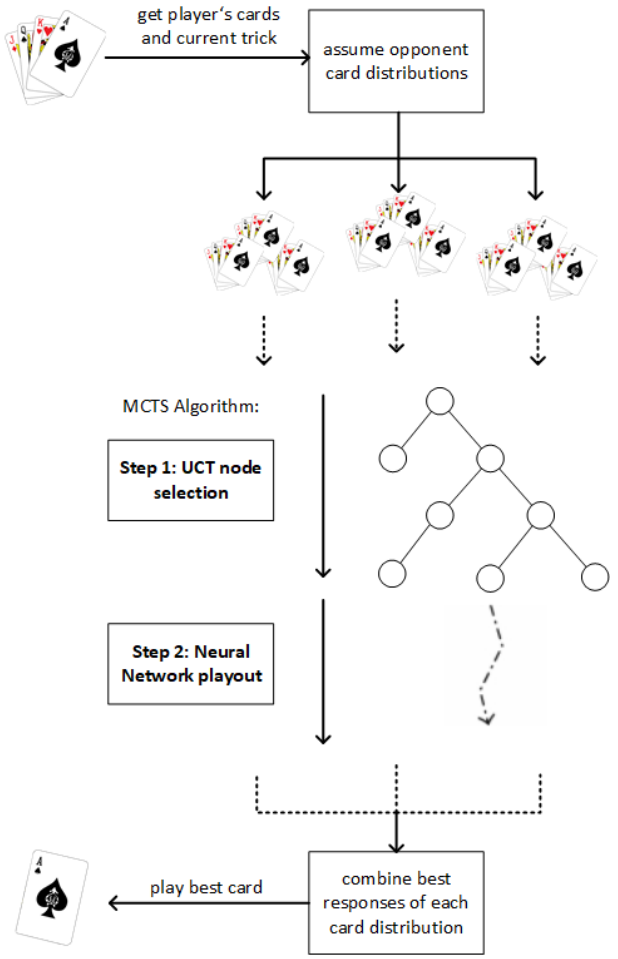


Fig. 2: Combination of a neural network-based decision heuristic and a MCTS agent for doppelkopf

A. Doppelkopf Data Set

The task of the neural network is to predict high value moves for the current game state. For our study we use 34 911 tournament game records downloaded from a German doppelkopf online platform [17] to train the network. Each game consists of 12 tricks, which can be played from four positions. This yields a total of 1 675 728 game states to be evaluated.

Table I summarizes the number of games per game mode. In general a neural network classifier needs a reasonable amount of data to learn useful patterns. Due to the low frequency of solos in our dataset we excluded solo games from our evaluation. The remainder (31 448 games, 1 509 504 game states) was split into three parts. We used 60% of the games (905 702 game states) to train the model, 20% (301 901 game states) for validation and another 20% (301 901 game states) for testing. The implementation was performed using the Keras framework [18] with the Tensorflow [19] backend. Results and source code are available on our website [20].

TABLE I: Distribution of games per mode in the provided game record data set.

Game Mode	Total Games	Share
normal game	24 548	$\approx 70.32\%$
announced marriage	6900	$\approx 19.76\%$
jack solo	1263	$\approx 3.62\%$
queen solo	763	$\approx 2.19\%$
ace solo	1086	$\approx 3.11\%$
♢ solo	88	$\approx 0.25\%$
♣ solo	85	$\approx 0.24\%$
♠ solo	85	$\approx 0.24\%$
unannounced marriage	51	$\approx 0.15\%$
♡ solo	43	$\approx 0.12\%$

B. Coding the current state of the game

Since our network predicts the next move, we first need a suitable coding of the current game state to serve as input for our neural network models.

The following information needs to be represented:

a) the currently played game mode
b) the current position in the trick
c) cards played during the current trick
d) history of previous tricks
e) *cards per player
f) *the party the player belongs to
g) *the parties of other players

Entries marked with an asterisk are bound by available information of the current game situation. Since the MCTS evaluates multiple card distributions, we create full information games and code the current card distribution under consideration. Generally, we use a binary vector as input. Each element is one-hot-encoded [21], meaning each possible value is represented by its own binary variable. This ensures an equal weighting between included information parts.

a) game mode (10 dimensions) For the purpose of coding the current game mode, we need ten dimensions, of which each represents one of the following modes: normal game, announced marriage, unannounced marriage, queen solo, jack solo, ace solo, ♢ solo, ♡ solo, ♠ solo, and ♣ solo. We currently do not include other than normal games and announced marriages. Nevertheless, the coding should be insusceptible to changes in the utilized training data.

b) current position (4 dimensions) Four neurons represent the player's position in the current trick.

*c-e) card related information (384 dimensions, 48 cards*8 possibilities)* Most of the neurons are needed to encode the current position of each of the 48 cards. For each card eight possible positions are considered: (1-4) hand of player one/two/three/four, (5-7) first/second/third card played in the current trick, or (8) played during an earlier trick. Each is represented by one dimension per card. Because the doppelkopf

deck consists of two cards of every kind, the first instance is always represented by the earliest occurrence of a card. For 48 cards we need a total of 384 input dimensions.

f-g) player parties (8 dimensions) Two dimensions per player are used to encode the player's party. Exactly one of these two dimensions per player is active. Note that even if the player is usually not aware of each player's party membership, we assume complete information during the simulation phase of the MCTS.

C. Design of the Neural Network

In this section we further specify our neural network model by describing the model of our hidden layers, as well as our configuration of training parameters. Concatenating the dimensions from above leads to a total number of 406 input dimensions. Generated neural networks will have 24 output neurons, one for each card type in the doppelkopf deck.

A feedforward neural network, also called a multilayer perceptron, is a classifier, which defines a mapping of input x to a category y (here: $x =$ game state, $y =$ card to be played). Its basic components are neurons, which are ordered in multiple layers. Each neuron consists of a net-input function, an activation function, and an output function. Information flows forward through the net, such that the output of a layer is fed as an input to the next layer. The number of layers is called the depth of a network, while the number of neurons per layer is called the width.

Supervised learning can be used to adjust the parameters of a neural network. Weights can be adapted with the backpropagation algorithm and its optimized adaptation AdaDelta, using the current error-rate of the network. A detailed description of neural network basics can be found in the book "Computational Intelligence" by Kruse et al. [22].

In the following sections we describe a network by the number of neurons per layer. Input and output layers will stay consistent in all tested network structures, since they are bound by the classification task. In our experiments we iteratively adjusted the number of hidden layers and the neurons they contain to optimize for the prediction accuracy of the network. Additionally, we used dropout layers [23]. For each neuron with a certain probability, called dropout rate, those layers deactivate this neuron during the training phase. Therefore, neurons hold partially redundant information, which increases generalization and decreases the chance of overfitting for the whole network. The final network structure is coded by the number of neurons per layer and the dropout rate of interlaced dropout layers. For example the network architecture name "406-100-0.2-24" describes a network of 406 input neurons, one hidden layer consisting of 100 neurons, a dropout layer with a dropout rate of 0.2, and an output layer of 24 neurons.

V. Evaluation and Discussion

Two evaluation phases were added for ensuring the validity of our results. First, we compare different network structures depending on their prediction accuracy. In our second evaluation we use the best neural network for enhancing MCTS simulations. The following subsections explain the details of both evaluations.

A. Next Card Prediction Evaluation

First tests on the prediction of expert moves were performed in the masterthesis of Matthias Hewelt [24]. Simple densely connected neural networks were used for the prediction of expert moves. Based on a rule of thumb by Heaton, the number of hidden neurons in a densely connected network should by between the number of inputs and the number of outputs [25]. Therefore, each hidden layer consisted of 215 neurons. The network was trained using batches of 1000 cards and stochastic gradient descent as optimization function.

Table II shows accuracy rates averaged over ten training repetitions. In the context of the given prediction task we distinguish Context Free (CF) and Context Sensitive (CS) evaluations. The Context Free evaluation directly compares the highest ranked card predicted by the neural network with the true card in the test sample. However, this disregards the fact that this card may not be playable. Therefore, we add the context bound evaluation, in which only the highest rated card, which also needs to be playable, is compared to the true outcome. Classification accuracies on each position were between approximately $40 - 50\%$.

Since we were not satisfied by those prediction rates we continued this analysis and tested other training parameters as well as multiple network structures. Highest accuracy gains were achieved by the cross-entropy loss function [21] and the AdaDelta Optimizer [26]. Furthermore, rectified linear units were used during our following experiments [27]. Thus, we reduced the vanishing/exploding gradient problem and were able to test bigger network sizes.

The results of this second phase are recorded in Table III. Here, we tested the influence of an interlaced dropout layer, by adjusting the dropout rate. We can see, that our second generation of networks achieved much higher accuracy rates of about 70%. Those networks were trained on the whole dataset.

Based on the previous results we used a fixed dropout rate of 0.2 in the remainder of our experiments. The last evaluation of our network architecture compares prediction rates of networks with different number of layers and nodes per layer. Results can be seen in Table IV.

After our first evaluation series, the best performing network was NN2. We tried to push the limits even further by optimizing the size of each layer and including batch normalization layers. The final result is network NN7, which outperformed all other networks during the training phase. It consists of two hidden layers with 700 and 406 neurons, an interlaced dropout layer, and two batch normalization layers [28]. The results of our other networks suggest, that increasing the number of layers may result in even better accuracies. As an example our largest network, which included seven hidden layers, performed equally well on most positions, and outperformed the two layer network on the next card prediction for the fourth position. While adding more layers may increase the prediction accuracy, it also increases the required time for a prediction. The networks

TABLE II: Comparison of classification rates. For each position we trained and tested each network structure. Further, a separate network was trained on the whole data set. 1 hidden layer: 406-215-24, 2 hidden layers: 406-215-215-24, 3 hidden layers, 4 hidden layer: 406-215-215-215-215-24, 5 hidden layers: 406-215-215-215-215-215-24

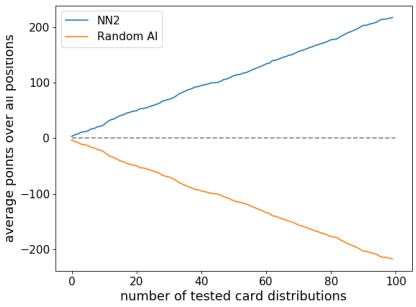
Network Architecture	All Positions		Position 1		Position 2		Position 3		Position 4	
	CF	CS	CF	CS	CF	CS	CF	CS	CF	CS
1 hidden layer	0.3294	0.4157	**0.3537**	**0.4986**	**0.3865**	**0.4778**	**0.3273**	**0.4364**	**0.3501**	**0.4908**
2 hidden layers	**0.4066**	**0.4767**	0.3498	0.4696	0.2440	0.3667	0.2680	0.4167	0.2855	0.4469
3 hidden layers	0.4044	0.4701	0.2686	0.4160	0.2185	0.3974	0.2377	0.4026	0.2346	0.4063
4 hidden layers	0.3479	0.4252	—	—	—	—	—	—	—	—
5 hidden layers	0.2969	0.3994	—	—	—	—	—	—	—	—

TABLE III: Comparison of dropout rates 0, 0.2, and 0.5 by mean accuracy and standard deviation of 10 training evaluations. The model was trained on all positions. We trained the models on different network structures with different number of hidden layers. 1 hidden layer: 406-100-24, 2 hidden layers: 406-812-406-24, 6 hidden layers: 406-1624-812-406-203-100-50 24, 7 hidden layers: 406-3248-1624-812-406-203 100-50-24

Network Architecture	dropout rate $= 0$		dropout rate $= 0.2$		dropout rate $= 0.5$	
	CF	CS	CF	CS	CF	CS
1 hidden layer	0.5997 ± 0.0148	0.6284 ± 0.0095	0.6135 ± 0.0022	0.6409 ± 0.0028	0.4965 ± 0.0044	0.5597 ± 0.0023
2 hidden layers	$\mathbf{0.7159} \pm 0.0036$	$\mathbf{0.7175} \pm 0.0035$	$\mathbf{0.7293} \pm 0.0030$	$\mathbf{0.7296} \pm 0.0030$	$\mathbf{0.7194} \pm 0.0023$	$\mathbf{0.7196} \pm 0.0023$
6 hidden layers	0.7136 ± 0.0004	0.7153 ± 0.0038	0.7186 ± 0.0129	0.7206 ± 0.0113	0.7069 ± 0.0095	0.7080 ± 0.0083
7 hidden layers	0.7125 ± 0.0026	0.7146 ± 0.0026	0.7240 ± 0.0139	0.7261 ± 0.0130	0.7176 ± 0.0027	0.7188 ± 0.0025

TABLE IV: Comparison of classification rates. Each model was separately trained and evaluated on the whole training data set and on the subset of each single position.

ID	Network Architecture	All Positions		Position 1		Position 2		Position 3		Position 4	
		CF	CS	CF	CS	CF	CS	CF	CS	CF	CS
NN1	406-100-0.2-24	0.6135	0.6410	0.5703	0.5751	0.6238	0.6526	0.6405	0.6721	0.6806	0.7145
NN2	406-812-0.2-406-24	0.6675	0.7293	0.6014	0.6022	0.7042	0.7072	0.7215	0.7256	0.7657	0.7698
NN3	406-3248-0.2-406-24	0.6896	0.6952	0.5913	0.5934	0.6803	0.6893	0.6991	0.7098	0.7450	0.7562
NN4	406-6496-0.2-406-24	0.6910	0.6954	0.5917	0.5939	0.6838	0.6921	0.6964	0.7072	0.7439	0.7554
NN5	406-1624-0.2-812-406-203-100-50-24	0.7186	0.7206	0.5945	0.5975	0.6919	0.7005	0.7082	0.7174	0.7577	0.7667
NN6	406-3248-0.2-1624-812-406-203-100-50-24	0.7240	0.7261	0.5900	0.5943	0.6928	0.7034	0.7091	0.7192	0.7601	0.7710
NN7	406-700-0.2-bn-406-bn-24	**0.7376**	**0.7378**	**0.6044**	**0.6050**	**0.7236**	**0.7250**	**0.7393**	**0.7411**	**0.7869**	**0.7887**



(a) neural network guided MCTS vs. Random AI

(b) neural network guided MCTS vs. MCST only using UCT

Fig. 3: Comparing the strength of play of different doppelkopf agents. Games included two players per agent type and were repeated on all six possible agent positionings. Each value represents the average points per round per agent type.

NN1, NN3, and NN7 were used during game simulation in our final experiments, in which we test our network in simulated games against other doppelkopf agents. This was done to evaluate a diverse set of networks, while adjusting the number of layers and neurons per layer. Networks NN4 to NN6 took too long for computation of the results and will be included in our extended results on our website [20].

B. Evaluating Player Performance

In our second evaluation series we compared the strength of play of our neural network enhanced MCTS agent with a random AI and the provided UCT implementation by Sievers and Helmert [13]. In order to eliminate the possibility of one player receiving better cards by chance, we repeatedly tested each card distribution on all six possible agent positionings. By averaging a players total points per round, we can assure a test under fair conditions. Since two players of the same type are undistinguishable, we took the sum of points per agent type.

Figure 3a shows the result of NN2 playing against two random players. Those lost nearly every game and did not stand a chance versus our developed agent. A more interesting comparison is shown in Figure 3b. Here, we see the performance of our neural network-based MCTS agent using networks of differing size and structure playing versus Sievers' and Helmert's UCT agent. Networks NN1 and NN3 were able to consistently beat their UCT agent. Despite dominating results in the card prediction task, NN7 was not able to beat the other's networks results during simulated play. We assume this is caused by overfitting during the training process on the used data set.

We also evaluated the run-time of our simulation. Average simulation times for networks NN1, NN3, and NN7 are 53s, 296s, and 161s per game. Therefore, an increased number of layers and hidden neurons per layer cause higher computation times.

VI. Conclusions

In this work we discussed the card game doppelkopf and its unique demands on the application of computational intelligence methods. These include the players uncertainty of the opponents cards as well as unknown player parties at the beginning of a game, which leads to an overall larger state space than comparable games like skat. Our work was motivated by reducing the state space under consideration during the MCTS by only simulating high value moves.

Based on a MCTS agent by Sievers and Helmert we introduced a neural network based decision heuristic for rollout simulations. The network was trained on a large database of tournament records to effectively predict expert moves. While the UCT algorithm is used to decide which nodes to expand during the MCTS, the neural network predicts probable player moves depending on the current state of the game. Thus, we were able to increase the quality of performed simulations leading to a higher strength of play.

During our work we trained different network architectures and analyzed their prediction accuracies. Only few layers were necessary to predict expert moves reasonably well. Small dropout rates further increased the prediction accuracy to 73%. The best net consisted of two hidden layers and one interjacent dropout layer, with a dropout rate of 0.2. For this purpose, we directly compared our decision heuristic enhanced agent with the previous implementation by Sievers and Helmert [13] by simulating 400 games. We ruled out positioning bias by repeating our simulation with the same card distributions dealt to every possible positioning of two agents each. Our evaluation demonstrates that our decision heuristic leads to an increased strength of play. Details on our results as well as our source code are available at [20].

In the future we plan to explore the applicability of other network architectures to further improve the prediction accuracy. In our current encoding time dependent models like simple recurrent neural networks [29] as well as long short-term memory networks [30] are not applicable. A time dependent encoding would only contain a history of all previous moves and the set of cards on the player's hand. However, those might need a larger database of game records to cover the time dependency of the players' moves. Additionally, using reinforcement learning to iteratively train two competing neural network based players may further improve the prediction accuracy.

References

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[2] N. Brown and T. Sandholm, "Safe and Nested Subgame Solving for Imperfect-Information Games," no. Johanson, may 2017.

[3] L. V. S. Allis, *Searching for Solutions in Games and Artificial Intelligence*, 1994.

[4] J. Rubin and I. Watson, "Computer poker: A review," *Artificial Intelligence*, vol. 175, no. 5-6, pp. 958–987, 2011.

[5] G. Van den Broeck, K. Driessens, and J. Ramon, "Monte-Carlo Tree Search in Poker Using Expected Reward Distributions," 2009, pp. 367–381.

[6] M. Ponsen, G. Gerritsen, and G. M. J.-B. Chaslot, "Integrating Opponent Models with Monte-Carlo Tree Search in Poker," *Proc. Conf. Assoc. Adv. Artif. Intell.: Inter. Decis. Theory Game Theory Workshop*, no. February, pp. 37–42, 2010.

[7] M. Lanctot and K. Waugh, "Monte Carlo sampling for regret minimization in extensive games," *Advances in Neural ...*, pp. 1–9, 2009.

[8] N. Brown, C. Kroer, and T. Sandholm, "Dynamic Thresholding and Pruning for Regret Minimization," 2017.

[9] J. Schäfer, "Monte Carlo Simulation im Skat," Ph.D. dissertation, Otto-von-Guericke-University Magdeburg, 2005.

[10] M. Buro, J. R. Long, T. Furtak, and N. Sturtevant, "Improving state evaluation, inference, and search in trick-based card games," *IJCAI International Joint Conference on Artificial Intelligence*, pp. 1407–1413, 2009.

[11] S. Kupferschmid and M. Helmert, "A Skat Player Based on Monte Carlo Simulation," *Proceedings of the Fifth International Conference on Computers and Games (CG 2006)*, pp. 135–147, 2006.

[12] T. Keller and S. Kupferschmid, "Automatic bidding for the game of skat," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5243 LNAI, pp. 95–102, 2008.

[13] S. Sievers and M. Helmert, "A doppelkopf player based on UCT," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9324, 2015, pp. 151–165.

[14] Deutscher Doppelkopf Verband e.V., "Homepage of the deutscher doppelkopf verband e.v." accessed: 2017-07-11. [Online]. Available: http://www.doko-verband.de

[15] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A Survey of Monte Carlo Tree Search Methods," *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 4, no. 1, pp. 1–43, 2012.

[16] L. Kocsis and C. Szepesvari, *Machine Learning: ECML 2006*, ser. Lecture Notes in Computer Science, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 4212.

[17] Patrick Rochol, "Doppelkopf website," accessed: 2017-07-12. [Online]. Available: https://www.online-doppelkopf.com

[18] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[19] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[20] "A Decision Heuristic for MCTS Doppelkopf Agents," http://www.is.ovgu.de/Team/Alexander+Dockhorn.html.

[21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[22] R. Kruse, C. Borgelt, C. Braune, S. Mostaghim, and M. Steinbrecher, *Computational Intelligence*, 2nd ed., ser. Texts in Computer Science. London: Springer London, 2016.

[23] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[24] M. Hewelt, "Masterthesis: Decision Heuristic Extension for the UCT-Algorithm," Ph.D. dissertation, Otto von Guericke University of Magdeburg, 2017.

[25] J. Heaton, *Introduction to Neural Networks for Java, 2Nd Edition*, 2nd ed. Heaton Research, Inc., 2008.

[26] M. D. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," dec 2012.

[27] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," *Proceedings of the 27th International Conference on Machine Learning*, no. 3, pp. 807–814, 2010.

[28] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 448–456. [Online]. Available: http://proceedings.mlr.press/v37/ioffe15.html

[29] S. Tokui and P. Networks, "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks," *arXiv:1512.05287*, no. Nips, pp. 1–9, 2015.

[30] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural computation*, vol. 9, no. 8, pp. 1735–80, 1997.