Freie Universität Berlin

Bachelorarbeit am Institut für Informatik der Freien Universität Berlin

Implementing a Doppelkopf Card Game Playing AI Using Neural Networks

Johannes Obenaus
Matrikelnummer: 4581314
jogo@zedat.fu-berlin.de

Betreuer: Maikel Nadolski
Eingereicht bei: Prof. Dr. Dr. Raúl Rojas
Zweitgutachter: Prof. Dr. Tim Landgraf

Berlin, September 14, 2017

Abstract

Recently, neural networks achieved major breakthroughs in many areas, which had been unfeasable for classical algorithms, e.g. in speech and image recognition, as well as in game playing AI. To apply this concept to the so far unsolved German card game Doppelkopf, we implemented an AI that combines the classical upper confidence bounds applied to trees (UCT) algorithm with a special type of recurrent neural networks, the long short-term memory (LSTM). We hypothesized this combination to be a promising approach, as the large search space of Doppelkopf would be reduced by using additional information from this neural network, as done in Go or Poker, for instance.
As a result, our LSTM predicted the next card in a Doppelkopf game at an average human level, and thereby, improved the tree search observably in a sense that promising nodes were visited earlier. Despite the impressive prediction accuracy of the LSTM, the combined UCT+LSTM player achieved a score of only 0.04 points per game on average against the same UCT player without LSTM, indicating that the LSTM did not improve the UCT player significantly.
As an outlook we propose several possibilities to improve this approach.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

September 14, 2017

Johannes Obenaus

Acknowledgement

Während meiner Bachelorarbeit erhielt ich viel Unterstützung, für die ich mich an dieser Stelle bedanken möchte. Der größte Dank gilt meinem Betreuer Maikel, ein guter Doppelkopfspieler, ausgezeichneter Doppelkopfprogrammierer und noch besserer Freund. Darüberhinaus danke ich meinem guten Freund Zacharias für anregende Diskussionen und viele konstruktive Ideen. Ich danke meiner Familie und vor allem meiner Freundin, Vivian, die mir jederzeit den Rücken freigehalten hat.

Contents

1 Doppelkopf **2**
1.1 Basic Rules . . . 2
1.2 Card Deck . . . 2
1.3 Taking Tricks . . . 3
1.4 Game Type Determination . . . 3
1.5 Game Evaluation . . . 4

2 Upper Confidence Bounds Applied to Trees **6**
2.1 Monte Carlo Tree Search . . . 6
2.2 UCT Weights . . . 7
2.3 Two Versions of the UCT Player . . . 8

3 AI Research in the Game of Go **10**
3.1 Early Approaches of Go Playing AIs . . . 10
3.2 AlphaGo . . . 10

4 Recurrent Neural Networks and LSTMs **12**
4.1 Basic Concepts of RNNs and LSTMs . . . 12
4.2 Character-level LSTM . . . 15
4.3 Adjusting the Character-level LSTM for Doppelkopf . . . 16
4.4 Does our LSTM learn how to play Doppelkopf? . . . 16
4.5 Adjusting the Weight Updating in the UCT Algorithm . . . 19

5 Experiments **20**
5.1 Prediction Accuracy . . . 20
5.1.1 Prediction Accuracy of the Random Player . . . 20
5.1.2 Prediction Accuracy of a Human Player . . . 20
5.1.3 Prediction Accuracy of the LSTM . . . 21
5.1.4 Prediction Accuracy per Trick . . . 22
5.2 LSTM Player versus UCT Player . . . 22
5.2.1 LSTM versus Random and UCT . . . 23
5.2.2 UCT+LSTM versus UCT . . . 24

6 Conclusion **25**

Bibliography **26**

Introduction

Doppelkopf is a German trick taking card game that is not very well known outside of Germany and thus, hasn't received that much attention from AI researchers so far. That might be the reason why there is no Doppelkopf AI playing on top human level, even though the complexity of the state space is lower than in other games, like Chess or Go. Since Doppelkopf has many rules of thumb, it is not very hard to implement a decent Doppelkopf AI that is based on rules and heuristics, but somewhat limited in situations that do not fit to those rules (e.g. the open source Doppelkopf AI FreeDoko[1]). To overcome this limitation, this thesis investigates a more general approach that uses no hardcoded rules or heuristics about the game, but only learns from human played games. To achieve this, we combined the UCT algorithm, which was implemented for Doppelkopf by Sievers in 2012 [13], with a special type of recurrent neural networks, the long short-term memory (LSTM). The LSTM is designed to reduce the search space significantly by predicting the next card. And indeed, our LSTM improved the tree search observably in a sense, that promising nodes were visited earlier. Using neural networks led to recent successes in many areas, e.g. in speech and image recognition as well as in game playing AI [7], [16], [15], [11], [12].

The first chapter of this thesis explains the basic rules of Doppelkopf. If you know the game already, it's safe to skip this chapter. In chapter 2, we are introducing the Monte Carlo tree search and UCT algorithms which are widely used not only in board and card games. Chapter 3 is a very brief digression to the history of Go playing AIs which motivates our approach to use neural networks for Doppelkopf. Recurrent neural networks and especially the LSTM that we used will be introduced in chapter 4 and is the major part of this thesis together with the experiments in chapter 5. In chapter 6 we are presenting our conclusion, whether this approach seems to be promising or not and how it could be improved.

[1] `http://free-doko.sourceforge.net/de/FreeDoko.html`

1 Doppelkopf

Doppelkopf is a German trick taking card game. Literally, "Doppel" means double and relates to the fact that every card exists twice in the deck. This chapter shall introduce the official Doppelkopf tournament rules, but only to an extend that is necessary for this thesis. A detailed description of all rules can be found on the DDV (Deutscher Doppelkopf Verband) website [2] (only in German).

1.1 Basic Rules

A Doppelkopf game is played by four players seperated into two teams, the Re- and Kontra-team. Each player gets twelve cards and before the actual game starts, there is a game type determination phase. There are three different game types: the regular game, the solo, and the marriage. In a regular game, which is the most common game type, the teams are not known at the beginning of the game, but determining this is a crucial point in Doppelkopf.
Before we are talking about how to determine the game type and the teams, I will introduce the card deck used for Doppelkopf.

1.2 Card Deck

There are 48 cards which are divided into trump and non-trump cards. Remember, that every card exists twice. In a regular game the trump cards in descending order are as follows

$$\heartsuit 10, \clubsuit Q, \spadesuit Q, \heartsuit Q, \diamondsuit Q, \clubsuit J, \spadesuit J, \heartsuit J, \diamondsuit J, \diamondsuit A, \diamondsuit 10, \diamondsuit K, \diamondsuit 9.$$

The non-trump cards in descending order are

$$\clubsuit A, \clubsuit 10, \clubsuit K, \clubsuit 9 \qquad \spadesuit A, \spadesuit 10, \spadesuit K, \spadesuit 9 \qquad \heartsuit A, \heartsuit K, \heartsuit 9,$$

while each non-trump suit is equivalent to each other.
As shown in Table 1, each card is worth a certain number of points which are independent from the suit and only determined by the card type. The basic goal in a Doppelkopf game is taking tricks with as many card points as possible but note that these points are always counted per team and not per player.

[2] http://www.doko-verband.de/Regeln__Ordnungen.html

Card Type	Points
Ace	11
Ten	10
King	4
Queen	3
Jack	2
Nine	0

Table 1: Amount of points for each card type.

1.3 Taking Tricks

The player who played the highest card (remember above mentioned order of cards) in a trick will take it and has to play the first card in the next trick. The first card played in a trick forces every other player to follow suit if possible. For example, if the first card in a trick is non-trump, the other players have to play a non-trump card of the same suit. If the first player plays a trump card, everybody else has to play trump. In case a player cannot follow suit, he can play any card.
All trump cards are higher than non-trump cards. If there are two highest cards in a trick, the one earlier played will take the trick.
Table 2 shows some concrete examples.

Player 1	Player 2	Player 3	Player 4	Winner of the Trick
$\clubsuit A$	$\clubsuit K$	$\clubsuit 9$	$\clubsuit A$	Player 1
$\heartsuit A$	$\diamondsuit 10$	$\heartsuit 9$	$\heartsuit K$	Player 2
$\spadesuit 10$	$\spadesuit A$	$\spadesuit 9$	$\heartsuit A$	Player 2
$\spadesuit J$	$\diamondsuit 9$	$\clubsuit Q$	$\heartsuit 10$	Player 4

Table 2: In each example player 1 plays the first card, which determines the suit to be followed. Example 1: Clubs is followed by every player, player 1's ace is higher than player 4's ace. Example 2: Player 1's $\heartsuit A$ is trumped by player 2's $\diamondsuit 10$, player 3 and player 4 have to follow suit, nevertheless. Example 3: Player 2 and 3 follow suit, player 4 has no spades and plays another non-trump, which cannot take the trick but is a lot of points (usually this makes sense, when player 2 and 4 are playing in the same team). Example 4: Trump has to be followed, player 3 plays the second highest trump, which is trumped by player 4's $\heartsuit 10$.

1.4 Game Type Determination

Before the actual game starts, each player has the opportunity to announce beeing "healthy" (gesund) or having a "reservation" (Vorbehalt). If all players are healthy, a

regular game will be played. In a regular game, the two players holding the $\clubsuit Q$ are playing in a team (Re-team) and the players without $\clubsuit Q$ are the Kontra-team. During the course of the game the players of a team need to find each other by deductions made from the playing style of the other players.
If one player has both $\clubsuit Qs$, he has a marriage. Now, this player has the option to announce reservation in the game type determination phase. When a marriage is announced, the partner will be whoever of the other three players takes a trick first. In case the player holding the marriage takes the first three tricks, he will play alone. And if this player does not announce reservation but instead healthy, he is playing alone without having the others to know this beforehand - a so-called silent solo.

Solo is the game type in which one player is playing against three others. I will omit the details, since we have implemented our AI only for regular games and marriages so far.

1.5 Game Evaluation

After the game type is determined, every player can make an announcement claiming that his team will win or reach a specific goal (bidding). The first announcement can be done until holding 11 cards or more in your own hand. The second announcement would be "Keine 90" and claims that the opponents team will make less than 90 points and can be announced until having at least 10 cards on your hand. In the same way it is possible to announce "Keine 60", "Keine 30" and "Schwarz" (black).
Its important to note, that by making an announcement you are, at the same time, revealing which team you are belonging to (Re or Kontra) and increasing the game value. Table 3 shows how to evaluate the game value.
There are 240 possible card points in one game and Kontra needs 120 points to win, except only Kontra was announced, in that case Re wins with 120:120.

Lets give an example to make the game evaluation process clear: Re won the game (+1 for Re), but Kontra is not under 90 and Kontra made a trick with 40 points (-1 for Re) and captured a $\diamondsuit A$ (-1 for Re). That leads to a final game score of -1 for Re or +1 for Kontra.
In case a team consisted only of one player, his points are multiplied with three, so that the sum of all four players points is zero at any time.

Criterion and Special Situations	**Score Points**
Won	+1
Opponent under 90	+1
Opponent under 60	+1
Opponent under 30	+1
Opponent has no trick	+1
Re announced	+2
Kontra announced	+2
Under 90 announced	+1
Under 60 announced	+1
Under 30 announced	+1
Black announced	+1
Won against $\clubsuit Qs$	+1
Every trick that is worth at least 40 points	+1
Captured opponents $\diamondsuit A$	+1
$\clubsuit J$ took the last trick	+1
Captured opponents $\clubsuit J$ in the last trick	+1

Table 3: Evaluating score points. Points for met criterions are added (or subtracted) at the end of the game and this sum will be the score of the game. There are also points for special incidents in a game, as mentioned in the lower part of the table.

2 Upper Confidence Bounds Applied to Trees

In this section we are going to present the upper confidence bounds applied to trees (UCT) algorithm [10] adapted for Doppelkopf. A detailed despriction can be found in [13] and [14]. We implemented the algorithm in the same way as it was done in [13], except for a small modification: We did not include biddings in the tree search, but only the cards that were played in the game.
As implied by the name, UCT is an algorithm that is meant to improve the performance of tree searches, in particular we will focus on Monte Carlo tree search (MCTS).

2.1 Monte Carlo Tree Search

MCTS is a heuristic search algorithm for decision making. It can be applied to various kinds of decision problems, in our case the decision problem is choosing which card to play next. To be presice, given a current game state, game history and a player to move we are interested in answering the question which card is best to be played next.

The basic idea of MCTS relies in repeatedly executing the four steps illustrated in Figure 1.

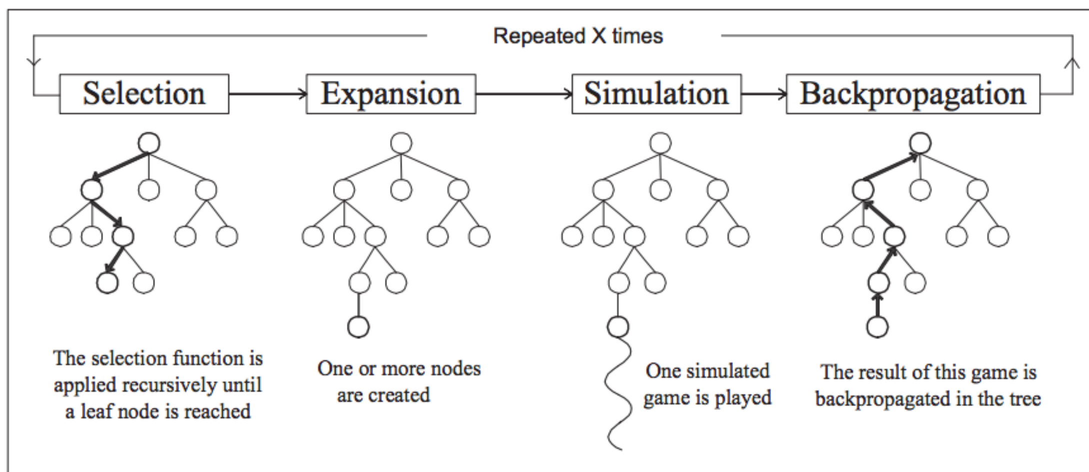

Figure 1: MCTS (source: [5]). 1.) Select the leaf node with highest weights. 2.) If it does not finish the game, expand it with child nodes. 3.) Perform a Monte Carlo simulation to compute a game score. 4.) Use this game score to update the weights.

We are starting from the root node and then walking down the tree along nodes with highest weights until we *select* the current most promising leaf node. Note, that each node in our search tree represents a game state and every legal next move can be represented as a child node. In the next step, we are *expanding* the leaf node with child nodes and then perform a random *sampling* of the rest of the game (this is called a Monte Carlo simulation (or just simulation)). It is also possible to compute more than one simulation every step and then average in order to get more reliable UCT rewards. The UCT reward is determined by the expected game points and expected

score points generated from this sampling and then, be used to update the weights in our tree (*backpropagation*). These four steps is what we call a rollout. The selection of a leaf node in the first step is crucial, that means the method by which to compute the weights for each node. It is quite obvious that weighting only by generating random samples is in many cases not good enough. It is especially problematic to get a good balance between exploiting deep variants of high value nodes and, at the same time, exploring many possible nodes (with lower weights), known as the exploration-exploitation dilemma. The UCT algorithm addresses exactly this problem and gives a formula for computing the weights.

2.2 UCT Weights

The weight w_i of a node i is computed with the following formula [4]:

$$w_i = v_i + C \cdot \sqrt{\frac{\log N_i}{n_i}} \tag{1}$$

where v_i is the estimated value of the node (UCT reward), as described above, n_i is the total number of times the node has been visited and N_i the total number its parent node has been visited. C is a bias parameter to guide the amount of exploration.

- The UCT reward v_i is often computed as the relative number of wins on a particular paths. However, in Doppelkopf it is not important whether to win or not, but to maximize the score points. Thus the following formula for computing the UCT rewards makes sense (details can be found in [13]):

  $$v_i = 500 \cdot s + e \tag{2}$$

  where s are the score points of the game (without bidding s will be between -8 and 8 realistically) and e the game points (between 0 and 240). The factor 500 is necessary to weight the score points stronger than the game points. In each node v_i is the average over the UCT rewards of the child nodes.

- $C \cdot \sqrt{\frac{\log(N_t)}{n_t}}$ is the exploration term. In case of normalized UCT rewards between 0 and 1, $C = \sqrt{2}$ is the theoretically optimal parameter [3], which we chose later. However, when not normalizing the UCT rewards, but instead using Equation 2, Sievers [13] chose $C = 16000$ as optimal hyperparameter.

UCT combines MCTS with the update formula given in Equation 1, and this is exactly the formula we want to improve by using a recurrent neural network (RNN). This RNN gets a game history as input and returns a vector with probabilities for the next card to be played, with the aim that this RNN assigns higher probabilities to cards that seem natural to be played next. Using this information we can adjust the exploration factor and visit promising nodes more often. More about this in subsection 4.1.
Let's have a look at two possible implementations of the UCT Doppelkopf player, first.

2.3 Two Versions of the UCT Player

In this chapter we want to explain two variants of implementing the UCT Doppelkopf player and how to use machine learning approaches to improve the playing strength. First of all, it is very important to notice that UCT, as described above, can only be applied for games with perfect information. Obviously, Doppelkopf is not a perfect information game, but we can attack this problem by assigning random, but (with the game history) consistent cards to the remaining players. This will be done several times and for each card assignment a UCT tree can be computed. At the end, we average over all those trees. Details about the card assignment problem can be found in [13].
For now, we are considering a given card assignment algorithm and introduce two variants how to execute the UCT algorithm.

Ensemble UCT

This version generates a certain number of fixed card assignments and for each card assignment, computes a new UCT tree. In our examples each of these trees often consists of 1000 rollouts.
Remember, that generating UCT trees is computationally expensive and thus the ensemble UCT player has the disadvantage that we are computing only 10 different card assignments and this is done randomly. When choosing bad card assignments, even the best selection policy will not give accurate results. On the other hand, computing UCT trees is computationally very expensive and therefore, it is not wise to increase this parameter alone. Generating a better card assignment, that is based also on implicitly deducted information from the game history, would improve this version more effectively.

Single UCT

The single UCT player uses only one UCT tree (with 10000 rollouts) and, in contrast to the ensemble player, it does not assume a fixed card assignment, but computes a new consistent card assignment at each rollout instead.
For this version, we have the advantage of generating a huge number of card assignments, what automatically leads to smoother averages of expected scores and weights, but on the other hand the UCT tree becomes exceedingly larger (wider), because the number of consistent child nodes increases drastically. Thus, for this version a policy that narrows the number of interesting child nodes would be more effective.

Another advantage of the single UCT player is that it is more likely to have nodes which stay consistent with the game history and therefore, many parts of the tree can be reused for the next turn.

We implemented a policy network that predicts the next card together with the ensemble UCT player and not the single UCT player, what sounds a bit weird after these explanations. The reason is that our very first approach was implementing a neural network, that predicts a good card assignment for a given game history and this ap-

proach suits the ensemble player better. However, we weren't very successful with generating card assignments, that are significantly better than random and therefore changed our strategy to predicting the next card. Of course, it would be very interesting to see, how our network works together with the single UCT player.

Before we are introducing RNNs, we want to present approaches of Go playing AIs, especially the recent success of AlphaGo, since there are major similarities to our project.

3 AI Research in the Game of Go

The recent results of Go playing AIs served as a source of inspiration for this project and should briefly be discribed in this chapter. We will see that there are many similarities to how the game of Go was attacked by AI developers as we are doing it for Doppelkopf. Go is a board game for two players, alternating placing black and white stones on a 19×19 board. Like in chess, all information is apparent to each player from the beginning. Hence, it is not necessary to guess or approximate any missing information. However, the state space in Go is exceedingly bigger than in Doppelkopf and therefore, a lot more difficult to approach.
In Doppelkopf the size of the state space can be computed as a product of the number of legal card deals, $\binom{48}{12} \cdot \binom{36}{12} \cdot \binom{24}{12} \cdot \binom{12}{12} \cdot \approx 2.4 \cdot 10^{26}$ and the number of possible game states for a fixed card deal, $\sum_{i=0}^{48} 4 \cdot \binom{12}{\lfloor (i+3)/4 \rfloor} \cdot \binom{12}{\lfloor (i+2)/4 \rfloor} \cdot \binom{12}{\lfloor (i+1)/4 \rfloor} \cdot \binom{12}{\lfloor (i)/4 \rfloor} \approx 2.4 \cdot 10^{13}$ [14]. Together this yields a rough estimate of $5.6 \cdot 10^{39}$ for the size of the state space.
On the other hand Go has already $2 \cdot 10^{170}$ legal board positions[3] and to get a very rough estimate of the state space: One game lasts more than 200 moves on average and has more than 100 legal moves each turn, yielding a state space in the dimension of 10^{400}.
Still, there is a top human level Go AI.

3.1 Early Approaches of Go Playing AIs

Some notable results in Go playing AI were achieved about a decade ago, when computer programs like MoGo or CrazyStone reached an average amateur level [6]. Both programs used basically the UCT algorithm described above. However, considering the enormous state space of Go, it is not surprising that this approach didn't yield in a top human level AI. Similar to Doppelkopf, the game evaluation can only be done at the end of the game. That means it is necessary to compute every rollout until the end in order to obtain a value for the UCT reward. Remember that a single game on the 19×19 board is more than 200 moves long and at each node we have a few hundred possible successors. Simply relying on improving the computational power of supercomputers seems hopeless in this case.
However, in 2016 Silver et al. [15] introduced some methods to drastically reduce the state space, that needs to be explored.

3.2 AlphaGo

AlphaGo is the first computer program beating a professional Go player in 2015 [15]. There are two major concepts that paved the way for this success. First, AlphaGo uses a deep convolutional neural network to predict the next move, the so-called policy network, and second, the so-called value network that is able to evaluate the current position. AlphaGo combines the MCTS algorithm with those networks. Simplified, the weights are computed with above mentioned UCT formula:

$$w_i = v_i + C \cdot \sqrt{\frac{\log(p_i)}{1 + n_i}} \tag{3}$$

[3] http://tromp.github.io/go/legal.html

where v_i is the action value, that is computed by a combination of a random simulation until the end of the game and the value network. p_i is the probability for the child node during expansion of a leaf node, and is given by the policy network. n_i is the number of visits.
AlphaGo's policy network, which is the exact analogon to our network, was trained with human professional games and learned to predict the next move with 55% accuracy. That is about the level of a top amateur player. This impressively reduces the number of moves which need to be explored.
In conclusion, AlphaGo is a successful mixture of classical tree search algorithms combined with different neural networks which reduces the search space.

4 Recurrent Neural Networks and LSTMs

Machine learning is the ability of a machine to acquire knowledge by extraction of patterns from data.
Commonly, a neural network consists of an input layer, hidden layers and an output layer, where each layer consists of a certain number of nodes and is often fully connected with the next layer via a non-linear activation function (e.g. sigmoid or ReLU). The purpose of a neural network is learning a function for a given problem. The most challenging part is always to find a suitable architecture for the given problem, i.e. an architecture that is expressive enough for the problem, but still efficiently trainable.
Ordinary neural networks are stateless, i.e. they get a fixed size input (e.g. an image) and return a classification. However, this is not what we need in our case. We want to have a neural network that gets a game history and the current state as input and then returns a probability distribution for the next card. "Remembering" and using the information from this history is exactly what recurrent neural networks and especially long short-term memory (LSTM) are made for. We will start with some basic information about RNNs and then introduce the RNN we used for Doppelkopf.

4.1 Basic Concepts of RNNs and LSTMs

In contrast to ordinary neural networks, the dimensions of input and output of the RNN are not fixed. We can feed sequences of variable length (e.g. written text or a game history) into the RNN and also get sequences as output. RNNs proved to be very effective for many applications like speech or handwriting recognition [7], [16]. Let's give some examples to compare situations where we are dealing with variable length input and output to situations with fixed length:

1. **Fixed length input and fixed length output**. Image classification with vanilla neural networks.

2. **Fixed length input and variable sequence output**. Describing an image (input) with a sequence of words (output). This is a fundamental problem and just two years ago, in 2015, Vinyals et al. [16] achieved a big improvement by using LSTMs.

3. **Variable sequence input and fixed length output**. Predicting the next card (output) in a game of Doppelkopf with the history of the game given (input).

4. **Variable sequence input and variable sequence output**. Predicting a game of Doppelkopf until the end (output) with the history of the game given (input).

In order to achieve the flexible behavior of an RNN, it combines an input vector with an internal state to produce a new state:

$$h_t = f(W_h \cdot h_{t-1} + W_x \cdot x_t + b_h) \tag{4}$$

$$y_t = W_y \cdot h_t + b_y \tag{5}$$

where h_t is the current and h_{t-1} the previous internal state, b_h and b_y are bias vectors, x_t is the input, and y_t the output. Also note the different weight matrices W_h, W_x

and W_y, in contrast to an ordinary neural network, in which the output in a forward step is simply calculated by $y = f(Wx)$ (f is a non-linear function, typically tanh or sigmoid). Equations 4 and 5 are executed in each node on the forward path.
As usual, when working with neural networks it is all about learning good weight matrices.
Figure 2 illustrates how an RNN works.

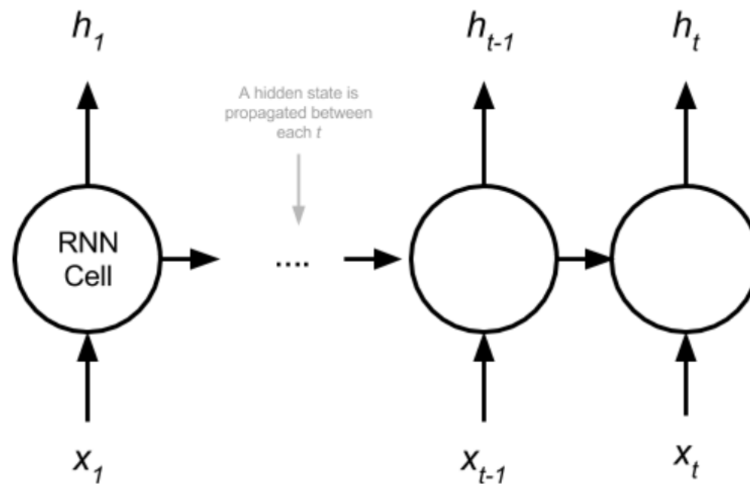

Figure 2: Source: `https://medium.com/datalogue/attention-in-keras-1892773a4f22`. This diagram shows a very simple view of an unrolled vanilla RNN cell in an RNN layer. During these t steps the RNN cell receives an input sequence of length t and passes a sequence of hidden states to the RNN cells in the next layer until the output layer returns the predicted sequence. Note, that the unrolled cells need to share the same weight matrices.

An LSTM is a special type of RNNs which is usually more efficient in learning to store data over an extended time interval [8] than common RNNs. LSTMs try to overcome the problem that with conventional methods, the error signal during backpropagation tends to either blow up or to vanish. Details about this can be found in [8].
Figure 3 is a close-up of what was called RNN cell in Figure 2 (with the difference, that we are introducing the more complex LSTM cell in Figure 3 now).

The basic idea of the LSTM cell is the same as for the RNN cell but uses some additional concepts to efficiently remember information. In addition to the hidden state h_t, an LSTM cell also uses the cell state c_t and propagates these values while unrolling the LSTM cell. Moreover, the LSTM is able to add and remove particular information to the cell state, what is controlled via "gates". These gates are not binary operations, but instead a sigmoid function to carefully control the amount of information that passes the gate. All gates are functions of the current input x_t and the previous state h_{t-1}. Lets have a look at the three different gates shown in Figure 3:

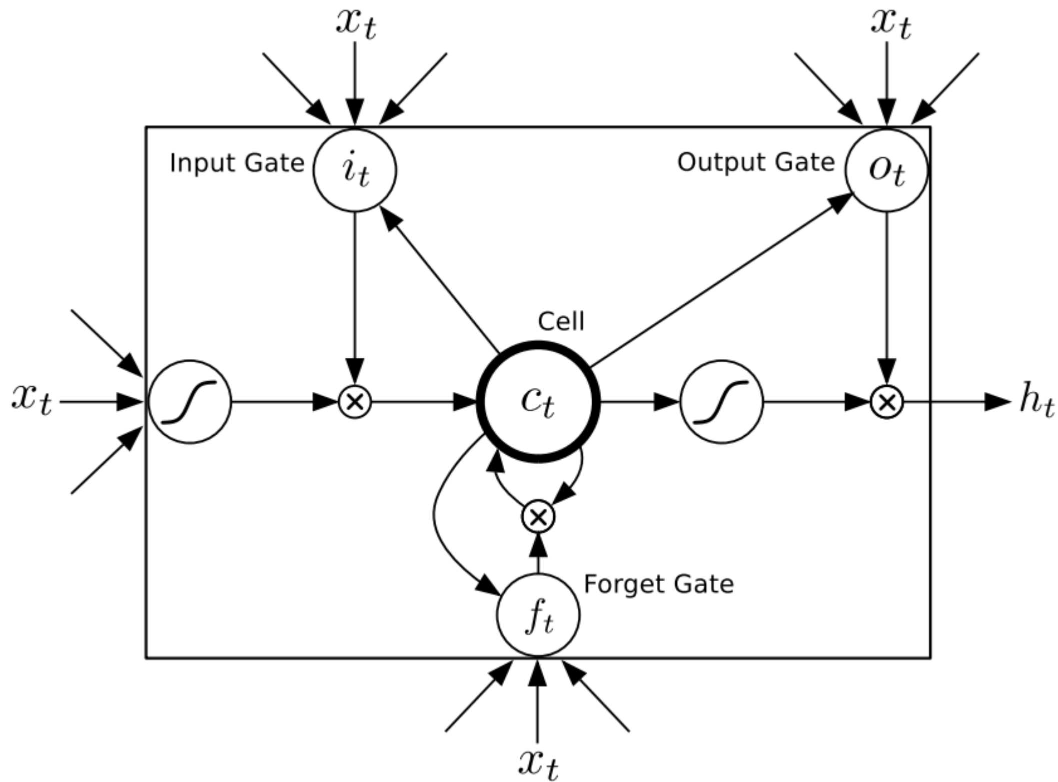

Figure 3: LSTM cell (source: [7]). In this diagram, x_t is the input at time step t that "enters" the LSTM cell via different gates together with the previous hidden state h_{t-1} and previous cell state c_{t-1}. We have three different gates i_t, f_t and o_t, each having different purposes and weight matrices learned during training. $\otimes$ denotes elementwise multiplication and $\int$ is the non-linear function tanh.

1. The forget gate f_t is supposed to remove information about previous cell states from the current cell state. For example, when a new Doppelkopf game in our training set starts, we want to forget the information about the previous game. The gate is implemented as follows:

$$f_t = \sigma(W_{fx} \cdot x_t + W_{fh} \cdot h_{t-1} + b_t)$$

2. The input gate i_t then controls which information to add to the new cell state. For instance, in a Doppelkopf game, it is a highly relevant information, when a $\clubsuit Q$ was played. To achieve this, we need to combine the sigmoid gate with a tanh layer, that creates a vector g_t containing the new information:

$$i_t = \sigma(W_{ix} \cdot x_t + W_{ih} \cdot h_{t-1} + b_t)$$
$$g_t = \tanh(W_{gx} \cdot x_t + W_{gh} \cdot h_{t-1} + b_t)$$

3. The output gate then determines which information are passed to the next cell and how to update the hidden state:

$$o_t = \sigma(W_{ox} \cdot x_t + W_{oh} \cdot h_{t-1} + b_t)$$

The input and forget gates are used to update the old cell state c_{t-1} to the new cell state c_t:

$$c_t = f_t \otimes c_{t-1} + i_t \otimes g_t$$

And the output gate then decides which parts of the cell state to propagate in the new hidden state:

$$h_t = \tanh(c_t) \otimes o_t$$

Lets summarize all equations:

$$i_t = \sigma(W_{ix} \cdot x_t + W_{ih} \cdot h_{t-1} + b_t) \quad (6)$$
$$f_t = \sigma(W_{fx} \cdot x_t + W_{fh} \cdot h_{t-1} + b_t) \quad (7)$$
$$o_t = \sigma(W_{ox} \cdot x_t + W_{oh} \cdot h_{t-1} + b_t) \quad (8)$$
$$g_t = \tanh(W_{gx} \cdot x_t + W_{gh} \cdot h_{t-1} + b_t) \quad (9)$$
$$c_t = f_t \otimes c_{t-1} + i_t \otimes g_t \quad (10)$$
$$h_t = \tanh(c_t) \otimes o_t \quad (11)$$

σ is the sigmoid function, W's denote weight matrices, x_t input at a given time step, h_t the hidden state, b's are bias vectors and $\otimes$ denotes elementwise multiplication.

With keeping the theoretical background information in mind, we can now proceed to the practical part. The LSTM we are using for our UCT Doppelkopf player is based on Karpathys [1] character-level LSTM that generates text.

4.2 Character-level LSTM

Karpathy's character-level LSTM [1] served as a starting point for us. We used the torch implementation that can be found here [2]. Torch is a common library for machine learning (like tensorflow or theano for example).
The purpose of this LSTM is producing text character by character. It can be trained on any piece of text, e.g. Shakespeare, LaTeX source code, baby names etc. and will then produce similar text. Of course, this network is not able to produce real Shakespeare-like texts, but it does produce correct english words. Plenty of examples can be found here [1].
The training of this LSTM works similarly to ordinary neural networks. Basically, we are executing the following steps repeatedly on the training data (mini batchwise). Each element in the trainingsdata is represented by a 1-hot vector encoding (some more words about the training data in subsection 4.3).

1. For a given state and input compute an output sequence of certain length (every element in this sequence is a vector) using equations 6 to 11 (remember that, in addition to input and output, we also need to keep track of the hidden and cell state).

2. compute loss by applying softmax classifier to all of these output vectors simultaniously.

3. Use Adam [9] to update the weight matrices.

We are omitting the technical details of this process.

4.3 Adjusting the Character-level LSTM for Doppelkopf

The above mentioned LSTM learns how to produce text with particular characteristics. That is actually very close to what we need for our Doppelkopf AI. In order to use this LSTM for Doppelkopf, we need two things: representing our training games appropriately and choosing a suitable architecture and hyperparameters. We acquired 0.5 million online played Doppelkopf games for training. Each line in the training data was a sequence of 62 characters (12 for the handcards of a certain player, 1 for the position of this player, 48 for the game and a newline character, that indicates the end of the game). We represented the 24 distinguishable Doppelkopf cards with characters from a to x and the positions with 0 to 3. Together with the newline character this gives 29 different characters, where each character was represented with a 1-hot encoding (i.e. a 29 dimensional vector, that is 1 at the position associated with the character and 0 else).

Our LSTM consists of two hidden layers and 64 nodes per layer. We want the LSTM to remember the history of one game and hence, set the sequence length it is supposed to remember to 62. We trained for two days on a CPU (70 epochs). Details and a comparison with other architectures can be found in subsubsection 5.1.3

As a validation we estimated the accuracy of our LSTM by counting the number of correct predicted cards on a validation set of previously unseen games. To be precise: for each game in the validation set we are providing a game sequence of length 0, 1, 2, ... 47 and comparing the predicted next card with that one that was actually played (details about this can be found in section 5).

4.4 Does our LSTM learn how to play Doppelkopf?

To answer this question, we will have a look at some typical situations (Table 4 and Table 5) in a Doppelkopf game and collect the next card predicted by the LSTM. Note, that in the best case, the LSTM is supposed to understand that always four cards form a trick and from the history it extracts relevant information. Making predictions that are consistent with the game history is also more important than predicting exactly the card, that was played.

The first two examples of Table 4 are very common: we are playing from the first position and want to play our non-trump ace. The probabilities predicted by the LSTM match with the card, suggested by the author and indicate that this kind of situation is well learned. However, in situations 3 and 4 we are in the position that we cannot follow suit. Instead we need to trump or play another suit. While in example three, the basic idea of the LSTM is correct (apparently it is not aware of its own handcards), it is quite off in the fourth example (only 3% for the correct card). We saw many cases, where understanding who is playing together and drop valuable

our Handcards	Position	Game History	Human played next card (LSTM probability)	LSTM #1 Prediction (probability)
♡10, ♡Q, ♢Q, ♠J, ♢K, ♢9, ♣A, ♠K, ♠9, ♠9, ♡K, ♡9	1	∅	♣A (0.99)	♣A (0.99)
♡10, ♡Q, ♢Q, ♠J, ♢K, ♢9, ♣A, ♣10, ♣10, ♣9, ♣9, ♠A	1	∅	♠A (0.99)	♠A (0.99)
♡10, ♡Q, ♢Q, ♠J, ♢10, ♢9, ♢9, ♣A, ♣A, ♣9, ♡A, ♡9	2	♠A	♢10 (0.26)	♢A (0.46)
♣Q, ♡Q, ♠J, ♡J, ♢J, ♢K, ♢K, ♢9, ♢9, ♠10, ♡K, ♡9	4	♢K, ♡J, ♣Q, ♢K, ♣A	♠10 (0.03)	♣K (0.23)

Table 4: This table shows some typical game situations and illustrates how well our LSTM works in these particular situations. The two last columns compare the card, suggested by the author, with the card predicted by the LSTM and the probabilities of the LSTM. E.g. in the first two cases the LSTM is very certain, but in the last example it is quite uncertain (23% for the #1 guess) and assigns only 3% to the best card.

cards on the partner's tricks, is the most difficult part for our LSTM. But since we never explicitly taught the LSTM any rules, this level of performance is not bad.

Table 5 shows a typical Doppelkopf game and nicely illustrates situations, that are handled well (+) by the LSTM and those, it has difficulties with (-).

+ In the first three tricks the LSTM always predicted the first ace correctly.

+ In trick four the LSTM "remembered" that player two didn't follow club suit in the first trick and correctly guessed, that he will trump it.

+ In trick eight player one could not follow trump anymore and from that point on (trick 9 - 12), the LSTM correctly predicted non-trump cards for this player.

- The prediction accuracies for player 4 are comparably poor and this is the player who's handcards we know. So, we can deduct, that our LSTM doesn't learn its own handcards properly.

No. Trick	Player 1	Player 2	Player 3	Player 4 (handcards known by LSTM)
1	$\boxed{\clubsuit A}$ (40%, #1)	$\diamondsuit A$ (6%, #5)	$\clubsuit 10$ (21%, #3)	$\clubsuit 9$ (85%, #1)
2	$\spadesuit 9$ (51%, #1)	$\boxed{\spadesuit A}$ (65%, #1)	$\spadesuit K$ (15%, #3)	$\spadesuit A$ (32%, #1)
3	$\heartsuit K$ (34%, #2)	$\boxed{\heartsuit A}$ (22%, #1)	$\heartsuit A$ (13%, #3)	$\diamondsuit J$ (4%, #6)
4	$\clubsuit 9$ (15%, #2)	$\diamondsuit 10$ (21%, #1)	$\spadesuit 10$ (1%, #8)	$\boxed{\clubsuit K}$ (9%, #4)
5	$\diamondsuit A$ (27%, #2)	$\boxed{\heartsuit J}$ (8%, #4)	$\clubsuit Q$ (19%, #1)	$\heartsuit 10$ (21%, #2)
6	$\diamondsuit J$ (10%, #3)	$\clubsuit Q$ (5%, #5)	$\diamondsuit K$ (41%, #1)	$\boxed{\heartsuit Q}$ (1%, #18)
7	$\heartsuit J$ (13%, #4)	$\boxed{\spadesuit J}$ (12%, #3)	$\diamondsuit Q$ (15%, #3)	$\heartsuit Q$ (22%, #2)
8	$\spadesuit 10$ (0%, #14)	$\heartsuit 10$ (17%, #2)	$\diamondsuit 9$ (29%, #2)	$\boxed{\spadesuit Q}$ (11%, #2)
9	$\clubsuit A$ (23%, #1)	$\boxed{\clubsuit J}$ (6%, #10)	$\diamondsuit 9$ (31%, #1)	$\spadesuit Q$ (34%, #1)
10	$\heartsuit 9$ (25%, #2)	$\diamondsuit 10$ (33%, #1)	$\diamondsuit K$ (40%, #1)	$\boxed{\spadesuit J}$ (7%, #6)
11	$\clubsuit K$ (24%, #1)	$\diamondsuit Q$ (12%, #6)	$\spadesuit K$ (31%, #1)	$\boxed{\clubsuit J}$ (5%, #8)
12	$\heartsuit 9$ (99%, #1)	$\boxed{\heartsuit K}$ (36%, #1)	$\spadesuit 9$ (35%, #1)	$\clubsuit 10$ (46%, #2)

Table 5: This table shows one complete example game, played by human players and analyzes the predictions made by the LSTM. Each row is one trick und the circled card is the first card in each trick (you always need to read from left to right). For example, in trick 12 player 2 plays the first card ($\heartsuit K$), then player 3 plays a $\spadesuit 9$, player 4 a $\clubsuit 10$ and finally player 1 $\heartsuit 9$. For each card we also specified how the LSTM predicted this card, e.g. the $\diamondsuit A$ in the first line was the fifth likeliest card in this situation with a probabilty of 6%.

4.5 Adjusting the Weight Updating in the UCT Algorithm

Having an LSTM that predicts the next card in a game of Doppelkopf, the major question is how to exploit this information in our UCT algorithm. As mentioned before, the solution lies mainly in improving the selection step. That means we need to adjust the formula for computing the node weights. In contrast to the formula used by AlphaGo, we do not want to trust our LSTM alone for the exploration term, but instead combine the LSTM value with the original exploration term. We think, that the network needs to be trained more intensely in order to achieve more accurate and reliable predictions. Therefore, we chose the following convex combination for the exploration term:

$$exploration_{new} = \big(\alpha \cdot p_i + (1 - \alpha)\big) \cdot q_i$$

where p_i is the LSTM prediction probability (normalized to mean 1), q_i is the old exploration term and $\alpha = \frac{1}{\sqrt{1+n_i}}$. That leads to our new equation for updating the weights (compare with Equation 3):

$$w_i = (v_i)_{normalized} + \big(\alpha \cdot p_i + (1 - \alpha)\big) \cdot q_i \tag{12}$$

with

$$\alpha = \frac{1}{\sqrt{1 + n_i}}$$

$$p_i = \frac{LSTM_probability}{\sum legal_probabilities} \cdot number_of_legal_cards$$

$$q_i = \sqrt{\frac{2 \cdot \log(N_i)}{1 + n_i}}$$

$$(v_i)_{normalized} = \frac{v_i + 4000}{8240}$$

$$v_i = 500 \cdot s + e$$

n_i is the number of visits of the current node and N_i the number of visits of the parent node, s are the score points and e the game points. Note, that the numbers for normalizing the UCT rewards v_i are chosen according to playing without bidding and solos. In that case, we are assuming a minimal game score of -8 and a maximal game score of +8 ($-8 \cdot 500 + 0 = -4000$ and $8 \cdot 500 + 240 = 4240$). Thus, we are normalizing the v_i between 0 and 1.
The selection of unvisited nodes is decided by the LSTM and Equation 12 favors nodes predicted by the LSTM at the beginning and later uses the old exploration term.

5 Experiments

In this section we will report two types of experiments to determine the strength of our Doppelkopf AI. The first set of experiments is designed to test the accuracy of the LSTM's prediction in human played games. The predicted cards were compared to the cards played in the actual game. The accuracy of our LSTM was compared to the accuracy of a random, but consistent player and to a human Doppelkopf player.
The second set of experiments shows results, how our new Doppelkopf AI, that uses the combination of UCT and LSTM, performed against Siever's UCT player. Furthermore, we tested how the LSTM performed when it solely (without UCT) played against random and UCT players.
For simplicity, we chose the following basic setting for all experiments:

1. The official Doppelkopf rules, described in Table 3 were used for evaluating the score points.
2. Only regular games and marriages were played (that means no solos).
3. All games were played without bidding (except explicitly specified).

5.1 Prediction Accuracy

In this section we want to present three experiments, that compare the prediction accuracy of the LSTM to a random player and a human player.

5.1.1 Prediction Accuracy of the Random Player

Setup: The validation set consisted of 500 human played games (each game was predicted from four different positions, hence 2000 games in total). For each of these games, the random player was assigned the handcards and the position of one player and then guessed the next played card randomly. As always, the random player was only allowed to choose from cards, that are consistent with the game history. For example, if there were already two ♣*Q*s played, he could never guess a ♣*Q* as the next card or if a certain player didn't follow suit, the random player could not predict a card of this suit for this player.
After prediction of a card by the random player, the prediction was compared to the actual next card. Whenever those cards matched the prediction was considered to be correct. The actual played card was added to the game history and the random player predicted again. Thus, each game consisted of 48 guesses. Note, that especially at the end of the game, the number of consistent cards decreases rapidly.
This explains a surprisingly high prediction rate of 28.8%.

5.1.2 Prediction Accuracy of a Human Player

We performed a small self experiment for predicting the next card in a Doppelkopf game. Unfortunately, this test was quite time consuming and therefore, was only done on a small set of 12 games. In order to get reliable results, this experiment of course has to be redone on a larger scale. However, our results give an idea what

would be a desirable value for the LSTM to achieve.
Again, the human player was given the 12 handcards and the position of a player and then guessed the next card of the game. Note, that it makes not a huge difference, whether the so far played game history is layed open or (as in official rules) only the last trick, since a descent human player is able to remember the whole game more or less.
The human test subject, a descent amateur player, achieved a prediction rate of 47.6%.

5.1.3 Prediction Accuracy of the LSTM

Here we used the same validation set for which our LSTM had to predict cards as we used for the random player. For each game the LSTM was given an input string consisting of the handcards and position of a player as well as the current game history. We then computed the probability distribution for the next card and considered the LSTM to be correct when the highest value of legal cards (according to the Doppelkopf rules) in this distribution related to the card that was actually played. Note that sometimes this value was low since the LSTM assigned higher probability to non-legal cards.
We used this test to benchmark the performance of our LSTM and tested with a bunch of different architectures and hyperparameters. Table 6 compares some of these different settings.

LSTM Architecture	LSTM Hyperparameters	Prediction Accuracy
2 layers, 64 nodes per layer	batch size: 25, sequence length: 62 , number epochs: 10 , learning rate: 0.002	45.9 %
2 layers, 128 nodes per layer	batch size: 50, sequence length: 62 , number epochs: 10 , learning rate: 0.002	44.9 %
2 layers, 256 nodes per layer	batch size: 50, sequence length: 62 , number epochs: 5 , learning rate: 0.002	50 %

Table 6: Comparing different architectures. The prediction accuracy was computed on a set of 500 games. In the case of 256 nodes per layer, training took a lot more time and we therefore trained only 5 epochs.

Increasing the number of nodes per layer increases the complexity of the LSTM and yields better results, but has high costs. Using 256 nodes per layer was computationaly just infeasible for us. Hence, we chose the version with 64 nodes, as in the first line of Table 6, to give a proof of concept (tuning the LSTM later is absolutely necessary).
The sequence length determines the number of characters the LSTM is supposed to

remember. A length of 62 makes sense, since a game in our case consists of 62 characters (12 handcards, 1 position, 48 game play and 1 game end). We also played with the other hyperparameters and found the settings as in Table 6 to be good settings. We trained the LSTM with 64 nodes for another 60 epochs and improved the prediction accuracy to 47.4%.
Lets summarize the main results from the first part of this section in Table 7.

Player	**Prediction Accuracy**
human player	47.6%
LSTM	47.4%
random player	28.8%

Table 7: Summarizing prediction accuracies of random, LSTM and human players. The validation set of the human player consisted of only 12 games (that means $12 \cdot 48 = 576$ guesses) and 4×500 games for the LSTM and random player (i.e. 96,000 guesses).

5.1.4 Prediction Accuracy per Trick

Analyzing the prediction accuracies per trick (Figure 4), gives a hint in which part of the game it makes most sense to use the LSTM and where it is unnecessary computation.

It is not very surprising, that the accuracy of the random prediction gets closer to the LSTM prediction at the end of the game (Figure 4), since the number of possibilities decreases rapidly. It's also interesting to observe, that the accuracies in the first three tricks are better, than in the middle of the game. The reason for this is quite simple: Usually in a game of Doppelkopf, the non-trump cards are played at the beginning and the trump cards in the middle and at the end of the game. Since there are less non-trump cards of the same suit, its easier to guess the next card, than for trump cards.

5.2 LSTM Player versus UCT Player

This section describes our main experiments in which we analyzed the actual playing strength of the LSTM+UCT by playing Doppelkopf games. The following experiments have been done with a fixed seed and we changed the position every 1000 games and then played the same 1000 games with rotated positions. Thus, in each experiment we played 4000 games and averaged the scores. We used official DDV rules, but played without solos.

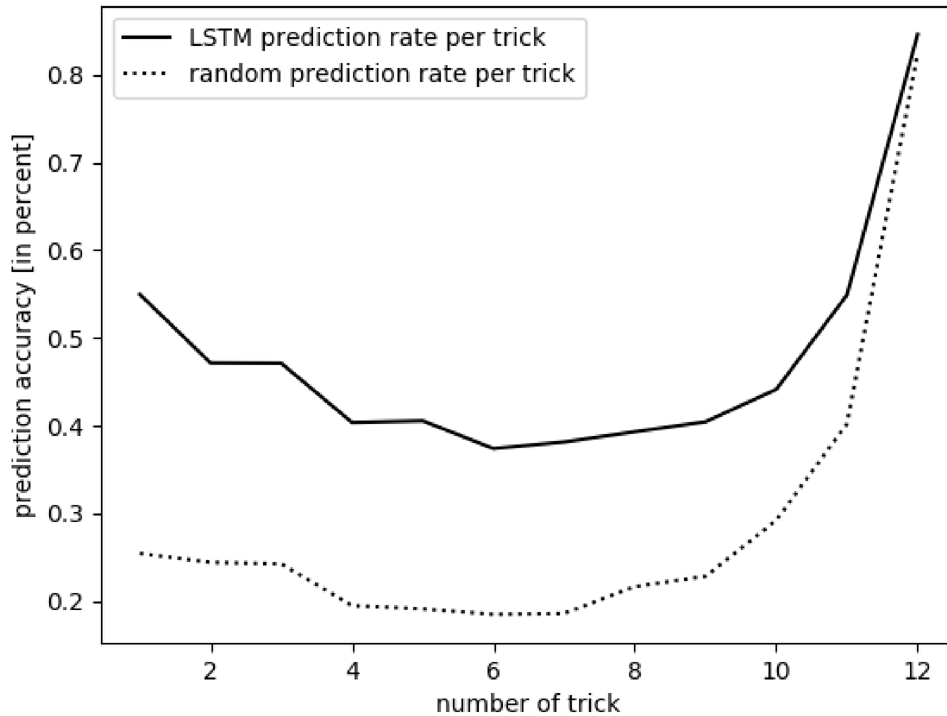

Figure 4: Prediction accuracies per trick. This diagram shows the prediction accuracies (y-axis) in each trick (x-axis) played in a Doppelkopf game. The accuracy is an average over 2000 games.

5.2.1 LSTM versus Random and UCT

This section describes our results, when the LSTM played solely (without using UCT) against UCT and random players (Table 8).

LSTM vs. random	UCT (10/1000/1) vs. random	LSTM vs. UCT (10/1000/1)
1.05 ± 0.13	1.70 ± 0.12	-0.36 ± 0.13

Table 8: In this experiment the LSTM player played without using UCT algorithm and thus, played the card predicted by the LSTM. The UCT player used 10 trees, 1000 rollouts and 1 simulation. The table illustrates the results of the LSTM player versus random players, UCT player versus random players, and LSTM player versus UCT players. The results are always the average points the first named player achieved against three players of the second type.

We can see that the LSTM without UCT already reaches a good level.
The 95% confidence interval was computed as follows (assuming a normal distribu-

tion):

$$average_score = M \pm 2.776 \cdot \sqrt{\frac{V}{N}},$$

where M is the sample mean, V the sample variance and N the number of scores in the sample (4000 in our case).

5.2.2 UCT+LSTM versus UCT

This is our main experiment. We let the LSTM+UCT player play against Siever's UCT player [13]. We used 10 UCT trees, 1000 rollouts per tree and 1 Monte Carlo simulation per rollout for all players.

UCT+LSTM (10/1000/1) vs. UCT (10/1000/1) with bidding
0.04 ± 0.24

Table 9: Results of the LSTM+UCT player performing against Siever's version of the UCT player.

As shown in Table 9 the LSTM didn't yield any improvement of the playing strength, what is a bit surprising after the promising results of the previous sections. However, we followed some of these games step by step to understand the process of the decision making and observed that the LSTM often gives the right impulse which nodes to visit. The problem is that our decision making is based on the expected score generated by the UCT algorithm and since this is the average over random sampling, it doesn't improve these scores when shifting the number of visits towards the promising nodes. That is an intrinsic problem of the UCT algorithm. We also observed that the expected scores for each legal next card do not differ a lot, e.g. when we had to choose between ♣A and ♣9 as our first card, the expected game score usually differed by less than 5 card points, what is highly unrealistic.

6 Conclusion

The aim of thesis was to implement a Doppelkopf AI that combines the upper confidence bounds applied to trees (UCT) algorithm with a policy network, that reduces the search space by predicting the next card. We implemented the policy network by using long short-term memory (LSTM), which predicted the next card at an average human level.

The first part of our experiments investigated how well the LSTM learned to predict cards in a game of Doppelkopf. We could nicely see, that it learned basic rules and concepts, like following suit. However, it had difficulties to "understand" more sophisticated concepts, especially in the middle of the game, like who plays together and does some mispredictions. Improving the LSTM will be crucial in order to build high level AI and is probably the most difficult part. The LSTM had difficulties to recognize the own handcards. Hence, the obvious step to improve our LSTM is to remove the information about the own handcards from the training data but instead feed this information directly into the LSTM using learnable weights, as done in [16] for example. Second, it is important to mention that we didn't have the best resources available, and trained most of the time on a single CPU. With better resources one could tune the hyperparameters more efficiently.
Another major issue is computational efficiency. Using the LSTM in each rollout is computationally very expensive and needs to be improved (subsubsection 5.1.3 illustrated the tradeof between complexity and prediction accuracy). Using the LSTM only in some parts of the game as suggested by Figure 4 is a reasonable option.

The second part of our experiments showed that the LSTM didn't improve the playing strength of the UCT player significantly. In this thesis we tried to adjust the exploration term with the policy network and it seems to be promising to further improve this network, but especially in combination with the single UCT player. The branching factor (average number of successors) for the ensemble UCT version isn't very high actually and hence, the policy network not very effective.
We observed that the playing strength of the ensemble UCT player crucially depends on the UCT rewards and expected scores. On the one hand, the expected scores are quite unsteady from trick to trick, what makes accurate bidding at the beginning impossible and on the other hand, the random sampling leads to vague and imprecise scores. Hence, our main conclusion is that implementing a value network is the more promising approach compared to the policy network. Furthermore, the ensemble UCT player would benefit from better solving the card assignment problem.

In a nutshell, our approach - using more general machine learning techniques instead of hardcoded rules and heuristics - seems to be a promising scratch on the surface so far and should be further investigated.

Bibliography

[1] http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[2] https://github.com/jcjohnson/torch-rnn.

[3] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, May 2002.

[4] Hyeong Soo Chang, Michael C. Fu, Jiaqiao Hu, and Steven I. Marcus. An adaptive sampling algorithm for solving markov decision processes. *Operations Research*, 53(1):126–139, 2005.

[5] Guillaume M. J-B. Chaslot, Mark H. M. Winands, H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4:343–357, 2008.

[6] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 273–280, New York, NY, USA, 2007. ACM.

[7] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649. IEEE, May 2013.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[10] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. *CoRR*, abs/1412.6564, 2014.

[12] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

[13] Silvan Sievers. Implementation of the UCT Algorithm for Doppelkopf. Master's thesis, Albert-Ludwigs-Universität Freiburg, 2012.

[14] Silvan Sievers and Malte Helmert. A Doppelkopf Player Based on UCT. In *38th Annual German Conference on Artificial Intelligence*, pages 151–165, 2015.

[15] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[16] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator, 2014. cite arxiv:1411.4555.